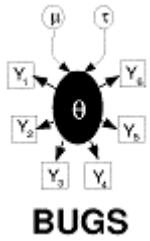


WinBUGS User Manual

Version 1.4, January 2003

David Spiegelhalter[1] Andrew Thomas[2] Nicky Best[2] Dave Lunn[2]

[1] MRC Biostatistics Unit,
Institute of Public Health,
Robinson Way,
Cambridge CB2 2SR, UK

[2] Department of Epidemiology & Public Health,
Imperial College School of Medicine,
Norfolk Place,
London W2 1PG, UK

e-mail: bugs@mrc-bsu.cam.ac.uk [general]
andrew.thomas@ic.ac.uk [technical]

internet: http://www.mrc-bsu.cam.ac.uk/bugs

Permission and Disclaimer

please click here to read the legal bit

More informally, potential users are reminded to be extremely careful if using this program for serious statistical analysis. We have tested the program on quite a wide set of examples, but be particularly careful with types of model that are currently not featured. If there is a problem, *WinBUGS* might just crash, which is not very good, but it might well carry on and produce answers that are wrong, which is even worse. Please let us know of any successes or failures.

Beware: MCMC sampling can be dangerous!

Contents

Introduction Ł

This manual
Advice for new users
MCMC methods
How *WinBUGS* syntax differs from that of *ClassicBUGS*
Changes from WinBUGS 1.3

ç

Compound Documents Ł

What is a compound document?
Working with compound documents
Editing compound documents

Compound documents and e-mail
Printing compound documents and Doodles
Reading in text files

Model Specification

Graphical models
Graphs as a formal language
The BUGS language: stochastic nodes
Censoring and truncation
Constraints on using certain distributions
Logical nodes
Arrays and indexing
Repeated structures
Data transformations
Nested indexing and mixtures
Formatting of data

DoodleBUGS: The Doodle Editor

General properties
Creating a node
Selecting a node
Deleting a node
Moving a node
Creating a plate
Selecting a plate
Deleting a plate
Moving a plate
Resizing a plate
Creating an edge
Deleting an edge
Moving a Doodle
Resizing a Doodle
Printing a Doodle

The Model Menu

General properties
Specification...
Update...
Monitor Metropolis
Save State
Seed...
Script

The Inference Menu

General properties
Samples...
Compare...
Correlations...
Summary...
Rank...
DIC...

The Info Menu

General properties
Open Log
Clear Log
Node info...
Components

The Options Menu

Output options...
Blocking options...
Update options...

Batch-mode: Scripts

Tricks: Advanced Use of the BUGS Language

Specifying a new sampling distribution
Specifying a new prior distribution
Specifying a discrete prior on a set of values
Using pD and DIC
Mixtures of models of different complexity
Where the size of a set is a random quantity
Assessing sensitivity to prior assumptions
Modelling unknown denominators
Handling unbalanced datasets
Learning about the parameters of a Dirichlet distribution
Use of the "cut" function

WinBUGS Graphics

General properties
Margins
Axis Bounds
Titles
All Plots
Fonts
Specific properties (via *Special...*)
Density plot
Box plot
Caterpillar plot
Model fit plot
Scatterplot

Tips and Troubleshooting

Restrictions when modelling
Some error messages
Some Trap messages
The program hangs
Speeding up sampling
Improving convergence

Tutorial

Introduction
Specifying a model in the BUGS language
Running a model in WinBUGS
Monitoring parameter values
Checking convergence
How many iterations after convergence?
Obtaining summaries of the posterior distribution

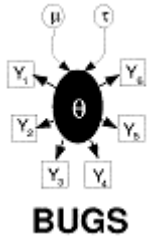
Changing MCMC Defaults (advanced users only)

Defaults for numbers of iterations
Defaults for sampling methods

Distributions

Discrete Univariate
Continuous Univariate
Discrete Multivariate
Continuous Multivariate

References



Introduction

Contents

This manual
Advice for new users
MCMC methods
How *WinBUGS* syntax differs from that of *Classic BUGS*
Changes from WinBUGS 1.3

This manual

[top | home]

This manual describes the *WinBUGS* software – an interactive Windows version of the *BUGS* program for Bayesian analysis of complex statistical models using Markov chain Monte Carlo (MCMC) techniques. *WinBUGS* allows models to be described using a slightly amended version of the *BUGS* language, or as Doodles (graphical representations of models) which can, if desired, be translated to a text-based description. The *BUGS* language is more flexible than the Doodles.

The sections cover the following topics:

Introduction: the software and how a new user can start using *WinBUGS*. Differences with previous incarnations of *BUGS* and *WinBUGS* are described.
Compound Documents: the use of the compound document interface that underlies the program, showing how documents can be created, edited and manipulated.
Model Specification: the role of graphical models and the specification of the *BUGS* language.
DoodleBUGS: The Doodle Editor: the *DoodleBUGS* software which allows complex Bayesian models to be

specified as Doodles using a graphical interface.
The Model Menu: the *Model* Menu permits models expressed as either Doodles or in the *BUGS* language to be parsed, checked and compiled.
The Inference Menu: the *Inference* Menu controls the monitoring, display and summary of the simulated variables: tools include specialized graphics and space-saving short-cuts for simple summaries of large numbers of variables.
The Info Menu: the *Info* menu provides a log of the run and other information.
The Options Menu: facility that allows the user some control over where the output is displayed and the various available MCMC algorithms.
Batch-mode: Scripts: how to run WinBUGS in batch-mode using 'scripts'.
Tricks: Advanced Use of the BUGS Language: special tricks for dealing with non-standard problems, e.g. specification of arbitrary likelihood functions.
WinBUGS Graphics: how to display and change the format of graphical output.
Tips and Troubleshooting: tips and troubleshooting advice for frequently experienced problems.
Tutorial: a tutorial for new users.
Changing MCMC Defaults (advanced users only): how to change some of the default settings for the MCMC algorithms used in WinBUGS.
Distributions: lists the various (closed-form) distributions available in WinBUGS.
References: references to relevant publications.

Users are advised that this manual only concerns the syntax and functionality of *WinBUGS*, and does not deal with issues of Bayesian reasoning, prior distributions, statistical modelling, monitoring convergence, and so on. If you are new to MCMC, you are strongly advised to use this software in conjunction with a course in which the strengths and weaknesses of this procedure are described. Please note the disclaimer at the beginning of this manual.

There is a large literature on Bayesian analysis and MCMC methods. For further reading, see, for example, Carlin and Louis (1996), Gelman et al (1995), Gilks, Richardson and Spiegelhalter (1996): Brooks (1998) provides an excellent introduction to MCMC. Chapter 9 of the *Classic BUGS* manual, 'Topics in Modelling', discusses 'non-informative' priors, model criticism, ranking, measurement error, conditional likelihoods, parameterisation, spatial models and so on, while the *CODA* documentation considers convergence diagnostics. Congdon (2001) shows how to analyse a very wide range of models using *WinBUGS*. The *BUGS* website provides additional links to sites of interest, some of which provide extensive examples and tutorial material.

Note that *WinBUGS* simulates each node in turn: this can make convergence very slow and the program very inefficient for models with strongly related parameters, such as hidden-Markov and other time series structures.

If you have the educational version of *WinBUGS*, you can run any model on the example data-sets provided (except possibly some of the newer examples). If you want to analyse your own data you will only be able to build models with less than 100 nodes (including logical nodes). However, the key for removing this restriction can be obtained by registering via the *BUGS* website, from which the current distribution policy can also be obtained.

Advice for new users

[top | home]

Although *WinBUGS* can be used without further reference to any of the *BUGS* project, experience with using *Classic BUGS* may be an advantage, and certainly the documentation on *BUGS* Version 0.5 and 0.6 (available from http://www.mrc-bsu.cam.ac.uk/bugs) contains examples and discussion on wider issues in modelling using MCMC methods. If you are using *WinBUGS* for the first time, the following stages might be reasonable:

1. Step through the simple worked example in the tutorial.
2. Try other examples provided with this release (see Examples Volume 1 and Examples Volume 2)
3. Edit the *BUGS* language to fit an example of your own.

If you are interested in using Doodles:

4. Try editing an existing Doodle (e.g. from Examples Volume 1), perhaps to fit a problem of your own.
5. Try constructing a Doodle from scratch.

Note that there are many features in the *BUGS* language that cannot be expressed with Doodles. If you wish to proceed to serious, non-educational use, you may want to dispense with *DoodleBUGS* entirely, or just use it for initially setting up a simplified model that can be elaborated later using the *BUGS* language. Unfortunately we do not have a program to back-translate from a text-based model description to a Doodle!

MCMC methods

[top | home]

Users should already be aware of the background to Bayesian Markov chain Monte Carlo methods: see for example Gilks *et al* (1996). Having specified the model as a full joint distribution on all quantities, whether parameters or observables, we wish to sample values of the unknown parameters from their conditional (posterior) distribution given those stochastic nodes that have been observed. The basic idea behind the Gibbs sampling algorithm is to successively sample from the conditional distribution of each node given all the others in the graph (these are known as full conditional distributions): the Metropolis-within-Gibbs algorithm is appropriate for difficult full conditional distributions and does not necessarily generate a new value at each iteration. It can be shown that under broad conditions this process eventually provides samples from the joint posterior distribution of the unknown quantities. Empirical summary statistics can be formed from these samples and used to draw inferences about their true values.

The sampling methods are used in the following hierarchies (in each case a method is only used if no previous method in the hierarchy is appropriate):

Continuous target distribution	**Method**
Conjugate	Direct sampling using standard algorithms
Log-concave	Derivative-free adaptive rejection sampling (Gilks, 1992)
Restricted range	Slice sampling (Neal, 1997)
Unrestricted range	Current point Metropolis
Discrete target distribution	**Method**
Finite upper bound	Inversion
Shifted Poisson	Direct sampling using standard algorithm

In cases where the graph contains a *Generalized Linear Model* (GLM) component, it is possible to request (see Blocking options...) that *WinBUGS* groups (or 'blocks') together the fixed-effect parameters and updates them via the multivariate sampling technique described in Gamerman (1997). This is essentially a Metropolis-Hastings algorithm where at each iteration the proposal distribution is formed by performing one iteration, starting at the current point, of *Iterative Weighted Least Squares* (IWLS).

If WinBUGS is unable to classify the full conditional for a particular parameter (p, say) according to the above hierarchy, then an error message will be returned saying "Unable to choose update method for p".

Simulations are carried out univariately, except for explicitly defined multivariate nodes and, if requested, blocks of fixed-effect parameters in GLMs (see above). There is also the option of using ordered over-relaxation (Neal, 1998), which generates multiple samples at each iteration and then selects one that is negatively correlated with the current value. The time per iteration will be increased, but the within-chain correlations should be reduced and hence fewer iterations may be necessary. However, this method is not always effective and should be used with caution.

A slice-sampling algorithm is used for non log-concave densities on a restricted range. This has an adaptive phase of 500 iterations which will be discarded from all summary statistics.

The current Metropolis MCMC algorithm is based on a symmetric normal proposal distribution, whose standard deviation is tuned over the first 4000 iterations in order to get an acceptance rate of between 20% and 40%. All summary statistics for the model will ignore information from this adapting phase.

It is possible for the user to change some aspects of the various available MCMC updating algorithms, such as the length of an adaptive phase – please see Update options... for details. It is also now possible to change the sampling methods for certain classes of distribution, although this is delicate and should be done carefully – see Changing MCMC Defaults (advanced users only) for details.

The shifted Poisson distribution occurs when a Poisson prior is placed on the order of a single binomial observation.

How *WinBUGS* syntax differs from that of *Classic BUGS*

[top | home]

Changes to the *BUGS* syntax have been kept, as far as possible, to simplifications. There is now:

- No need for constants (these are declared as part of the data).
- No need for variable declaration (but all names used to declare data must appear in the model).
- No need to specify files for data and initial values.
- No limitation on dimensionality of arrays.
- No limitation on size of problems (except those dictated by hardware).
- No need for semi-colons at end of statements (these were never necessary anyway!)

A major change from the *Classic BUGS* syntax is that when defining multivariate nodes, the range of the variable must be explicitly defined: for example

```
x[1:K] ~ dmnorm(mu[], tau[,])
```

must be used instead of `x[] ~ dmnorm(mu[], tau[,])`, and for precision matrices you must write, say

```
tau[1:K, 1:K] ~ dwish(R[,], 3)
```

rather than `tau[,] ~ dwish(R[,], 3)`.

The following format must now be used to invert a matrix:

```
sigma[1:K, 1:K] <- inverse(tau[,])
```

Note that `inverse(.)` is now a vector-valued function as opposed to the relatively inefficient component-wise evaluation required in previous versions of the software.

To convert *Classic BUGS* files to run under *WinBUGS*:

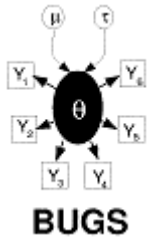
a) Open the `.bug` file as a text file, delete unnecessary declarations, and save as an `.odc` document.
b) Open `.dat` files: data has to be formatted as described in Formatting of data: eg
   * matrices in data files need to have the full '.structure' format
   * all data in datafile need to be described in the model
   * need data list of constants and file sizes
   * need column headings on rectangular arrays
   The data can be copied into the `.odc` file, or kept as a separate file.
c) Copy the contents of the `.in` file into the `.odc` file.

Changes from WinBUGS 1.3

[top | home]

- modular on-line manual;
- ability to run in batch-mode using scripts;
- running of default script on start-up to allow calling from other programs;
- new graphics (see here, for example) and editing of graphics – note that graphics from previous versions of the software will be incompatible with this version (1.4);
- missing data and range constraints allowed for multivariate normal;
- new distributions: negative binomial, generalized gamma, multivariate Student-t;
- DIC menu option for model comparison;
- Options menu, for advanced control of MCMC algorithms, for example;
- new syntax for (more efficient) 'inverse' function;
- "interp.lin" interpolation function, "cut" function;
- recursively- (and thus efficiently-) calculated running quantiles;

- MCMC algorithms: block updating of fixed effects – see here and/or here for details;
- non-integer binomial and Poisson data;
- Poisson as prior for continuous quantity;
- 'coverage' of random number generator;
- additional restrictions: END command for rectangular arrays;
- spatial (CAR) models moved to GeoBUGS;
- new display options;
- now possible to print out posterior correlation coefficients for monitored variables;
- new manual sections: Batch-mode: Scripts, Tricks, WinBUGS Graphics, Tutorial, and Changing MCMC Defaults.



Compound Documents

Contents

What is a compound document?
Working with compound documents
Editing compound documents
Compound documents and e-mail
Printing compound documents and Doodles
Reading in text files

What is a compound document?

[top | home]

A compound document contains various types of information (formatted text, tables, formulae, plots, graphs etc) displayed in a single window and stored in a single file. The tools needed to create and manipulate these information types are always available, so there is no need to continuously move between different programs. The *WinBUGS* software has been designed so that it produces output directly to a compound document and can get its input directly from a compound document. To see an example of a compound document click here. *WinBUGS* is written in Component Pascal using the BlackBox development framework: see http://www.oberon.ch.

In *WinBUGS* a document is a description of a statistical analysis, the user interface to the software, and the resulting output.

Compound documents are stored with the .odc extension.

Working with compound documents

[top | home]

A compound document is like a word-processor document that contains special rectangular embedded regions or elements, each of which can be manipulated by standard word-processing tools -- each rectangle behaves like a single large character, and can be focused, selected, moved, copied, deleted etc. If an element is focused the tools to manipulate its interior become available.

The *WinBUGS* software works with many different types of elements, the most interesting of which are Doodles, which allow statistical models to be described in terms of graphs. *DoodleBUGS* is a specialised graphics editor and is described fully in DoodleBUGS: The Doodle Editor. Other elements are rather simpler and are used to display plots of an analysis.

Editing compound documents

[top | home]

WinBUGS contains a built-in word processor, which can be used to manipulate any output produced by the software. If a more powerful editing tool is needed *WinBUGS* documents or parts of them can be pasted into a standard OLE enabled word processor.

Text is selected by holding down the left mouse button while dragging the mouse over a region of text. Warning: if text is selected and a key pressed the selection will be replaced by the character typed. The selection can be removed by pressing the "Esc" key or clicking the mouse.

A single element can be selected by clicking once into it with the left mouse button. A selected element is distinguished by a thin bounding rectangle. If this bounding rectangle contains small solid squares at the corners and mid sides it can be resized by dragging these with the mouse. An element can be focused by clicking twice into it with the left mouse button. A focused element is distinguished by a hairy grey bounding rectangle.

A selection can be moved to a new position by dragging it with the mouse. To copy the selection hold down the "control" key while releasing the mouse button.

These operations work across windows and across applications, and so the problem specification and the output can both be pasted into a single document, which can then be copied into another word-processor or presentation package.

The style, size, font and colour of selected text can be changed using the *Attributes* menu. The vertical offset of the selection can be changed using the *Text* menu.

The formatting of text can be altered by embedding special elements. The most common format control is the ruler: pick option *Show Marks* in menu *Text* to see what rulers look like. The small black up-pointing triangles are tab stops, which can be moved by dragging them with the mouse and removed by dragging them outside the left or right borders of the ruler. The icons above the scale control, for example, centering and page breaks.

Vertical lines within tables can be curtailed by inserting a ruler and removing the lines by selecting each tab-stop and then *ctrl*-left-mouse-click. (Warning: removing the left-most line requires care: there is a tab-stop hidden behind the upper left-most one that can cause a crash if deleted in the usual way - it seems to require a *ctrl*-**right**-mouse-click!).

Compound documents and e-mail

[top | home]

WinBUGS compound documents contain non-ascii characters, but the *Tools* menu contains a command *Encode Document* which produces an ascii representation of the focus document. The original document can be recovered from this encoded form by using the *Decode* command of the *Tools* menu. This allows, for example, Doodles to be sent by e-mail.

Printing compound documents and Doodles

[top | home]

These can be printed directly from the *File* menu. If postscript versions of Doodles or whole documents are wanted, you could install a driver for a postscript printer (say Apple LaserWriter), but set it up to *print to file* (checking the paper size is appropriate). Alternatively Doodles or documents could be copied to a presentation or word-processing package and printed from there.

Reading in text files

[top | home]

Open these from the *File* menu as text files. They can be copied into documents, or stored as documents.

Model Specification

Contents

Graphical models
Graphs as a formal language

The BUGS language: stochastic nodes
Censoring and truncation
Constraints on using certain distributions
Logical nodes
Arrays and indexing
Repeated structures
Data transformations
Nested indexing and mixtures
Formatting of data

Graphical models

[top | home]

We strongly recommend that the first step in any analysis should be the construction of a *directed graphical model*. Briefly, this represents all quantities as nodes in a directed graph, in which arrows run into nodes from their direct influences (parents). The model represents the assumption that, given its parent nodes pa[*v*], each node *v* is independent of all other nodes in the graph except descendants of *v*, where descendant has the obvious definition.

Nodes in the graph are of three types.

1. *Constants* are fixed by the design of the study: they are always founder nodes (*i.e.* do not have parents), and are denoted as rectangles in the graph. They must be specified in a data file.

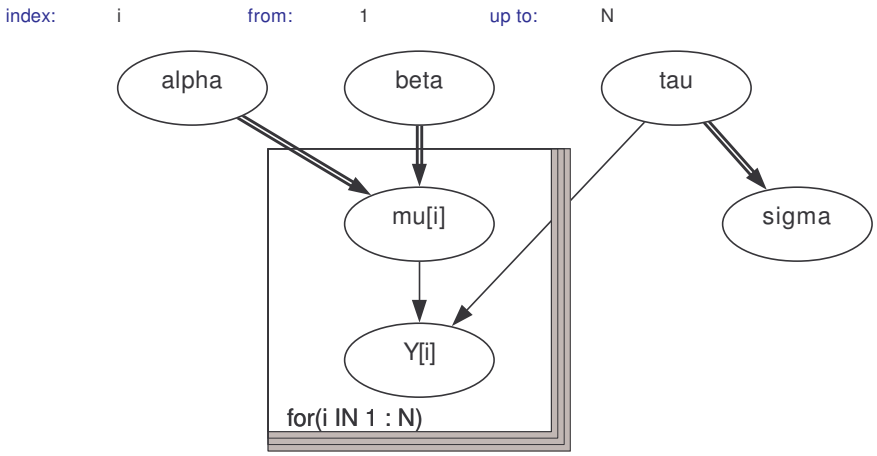
2. *Stochastic node*s are variables that are given a distribution, and are denoted as ellipses in the graph; they may be parents or children (or both). Stochastic nodes may be observed in which case they are *data*, or may be unobserved and hence be *parameters*, which may be unknown quantities underlying a model, observations on an individual case that are unobserved say due to censoring, or simply missing data.

3. *Deterministic nodes* are logical functions of other nodes.

Quantities are specified to be data by giving them values in a data file, in which values for constants are also given.

Directed links may be of two types: a solid arrow indicates a stochastic dependence while a hollow arrow indicates a logical function. An undirected dashed link may also be drawn to represent an upper or lower bound.

Repeated parts of the graph can be represented using a 'plate', as shown below for the range *(i in 1:N)*.



A simple graphical model, where Y[i] depends on mu[i] and tau, with mu[i] being a logical function of alpha and beta.

The conditional independence assumptions represented by the graph mean that the full joint distribution of all quantities *V* has a simple factorisation in terms of the conditional distribution p(*v* | parents[*v*]) of each node

given its parents, so that

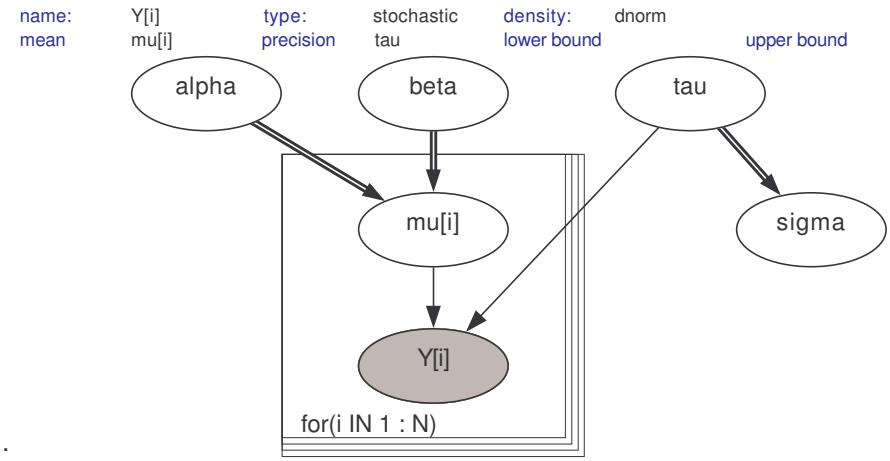
$$p(V) = \prod p(v \mid \text{parents}[v]) \qquad v \text{ in } V$$

The crucial idea is that we need only provide the parent-child distributions in order to fully specify the model, and *WinBUGS* then sorts out the necessary sampling methods directly from the expressed graphical structure.

Graphs as a formal language

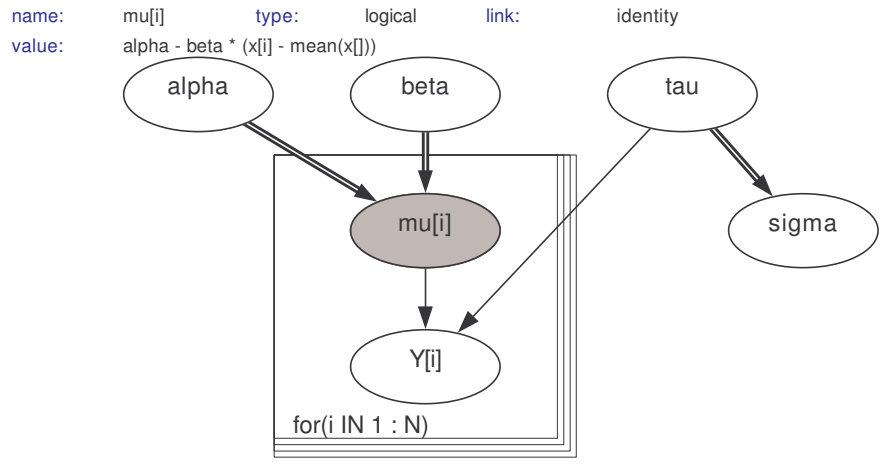
[top | home]

A special drawing tool *DoodleBUGS* has been developed for specifying graphical models, which uses a hyper-diagram approach to add extra information to the graph to give a complete model specification. Each stochastic and logical node in the graph must be given a name using the conventions explained in Creating a node.



.

The shaded node Y[i] is normally distributed with mean mu[i] and precision tau.



The shaded node mu[i] is a logical function of alpha, beta, and the constants x. (x is not required to be shown in the graph).

The value function of a logical node contains all the necessary information to define the logical node: the logical links in the graph are not strictly necessary.

As an alternative to the Doodle representation, the model can be specified using the text-based *BUGS* language, headed by the `model` statement:

```
model {
   text-based description of graph in BUGS language
}
```

The *BUGS* language: stochastic nodes [top | home]

In the text-based model description, stochastic nodes are represented by the node name followed by a twiddles symbol followed by the distribution name followed by a comma-separated list of parents enclosed in brackets *e.g.*

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

The distributions that can be used in *WinBUGS* are described in Distributions. Clicking on the name of each distribution should provide a link to an example of its use provided with this release. The parameters of a distribution must be explicit nodes in the graph (scalar parameters can also be numerical constants) and so may not be function expressions.

For distributions not featured in Distributions, see Tricks: Advanced Use of the BUGS Language.

Censoring and truncation [top | home]

Censoring is denoted using the notation `I(lower, upper)` *e.g.*

```
x ~ ddist(theta)I(lower, upper)
```

would denote a quantity `x` from distribution `ddist` with parameters `theta`, which had been observed to lie between `lower` and `upper`. Leaving either `lower` or `upper` blank corresponds to no limit, e.g. `I(lower,)` corresponds to an observation known to lie above `lower`. Whenever censoring is specified the censored node contributes a term to the full conditional distribution of its parents. This structure is only of use if `x` has not been observed (if `x` is observed then the constraints will be ignored).

It is vital to note that this construct does NOT correspond to a truncated distribution, which generates a likelihood that is a complex function of the basic parameters. Truncated distributions might be handled by working out an algebraic form for the likelihood and using the techniques for arbitrary distributions described in Tricks: Advanced Use of the BUGS Language.

It is also important to note that if `x`, `theta`, `lower` and `upper` are all unobserved, then `lower` and `upper` must not be functions of `theta`.

Constraints on using certain distributions [top | home]

Contiguous elements: Multivariate nodes must form contiguous elements in an array. Since the final element in an array changes fastest, such nodes must be defined as the final part of any array. For example, to define a set of K * K Wishart variables as a single multidimensional array `x[i,j,k]`, we could write:

```
for (i in 1:I) {
  x[i, 1:K, 1:K] ~ dwish(R[i,,], 3)
}
```

where `R[i,,]` is an array of specified prior parameters.

No missing data: Data defined as multinomial or as multivariate Student-t must be complete, in that missing values are not allowed in the data array. We realise this is an unfortunate restriction and we hope to relax it in the future. For multinomial data, it may be possible to get round this problem by re-expressing the multivariate likelihood as a sequence of conditional univariate binomial distributions.

Note that multivariate normal data may now be specified with missing values.

Conjugate updating: Dirichlet and Wishart distributions can only be used as parents of multinomial and multivariate normal nodes respectively.

Parameters you can't learn about and must specify as constants: The parameters of Dirichlet and Wishart distributions and the order (N) of the multinomial distribution must be specified and cannot be given prior distributions. (There is, however, a trick to avoid this constraint for the Dirichlet distribution – see here.)

Structured precision matrices for multivariate normals: these can be used in certain circumstances. If a

Wishart prior is not used for the precision matrix of a multivariate normal node, then the elements of the precision matrix are updated univariately without any check of positive-definiteness. This will result in a crash unless the precision matrix is parameterised appropriately. **This is the user's responsibility!**

Non-integer data for Poisson and binomial: Previously only integer-valued data were allowed with Poisson and binomial distributions – this restriction has now been lifted. More generally, it is now possible to specify a Poisson prior for any continuous quantity.

Range constraints – using the `I(.,.)` notation – cannot be used with multivariate nodes: except for multivariate normal distributions in which case the arguments to the `I(.,.)` function may be specified as 'blanks' or as vector-valued bounds.

Logical nodes

[top | home]

Logical nodes are represented by the node name followed by a left pointing arrow followed by a logical expression of its parent nodes e.g.

```
mu[i] <- beta0 + beta1 * z1[i] + beta2 * z2[i] + b[i]
```

Logical expressions can be built using the following operators: plus, multiplication, minus, division and unitary minus. The functions in Table I below can also be used in logical expressions.

In Table I, function arguments represented by *e* can be expressions, those by *s* must be scalar-valued nodes in the graph and those represented by *v* must be vector-valued nodes in a graph.

Table I: Functions

`abs(e)`	$\|e\|$
`cloglog(e)`	$\ln(-\ln(1-e))$
`cos(e)`	cosine(e)
`cut(e)`	cuts edges in the graph – see Use of the "cut" function
`equals(e1, e2)`	1 if $e1 = e2$; 0 otherwise
`exp(e)`	$\exp(e)$
`inprod(v1, v2)`	$\sum_i v1_i v2_i$
`interp.lin(e, v1, v2)`	$v2_p + (v2_{p+1} - v2_p) * (e - v1_p) / (v1_{p+1} - v1_p)$ where the elements of $v1$ are in ascending order and p is such that $v1_p < e < v1_{p+1}$
`inverse(v)`	v^{-1} for symmetric positive-definite matrix v
`log(e)`	$\ln(e)$
`logdet(v)`	$\ln(\det(v))$ for symmetric positive-definite v
`logfact(e)`	$\ln(e!)$
`loggam(e)`	$\ln(\Gamma(e))$
`logit(e)`	$\ln(e / (1-e))$
`max(e1, e2)`	$e1$ if $e1 > e2$; $e2$ otherwise
`mean(v)`	$n^{-1}\sum_i v_i$ $n = \dim(v)$
`min(e1, e2)`	$e1$ if $e1 < e2$; $e2$ otherwise
`phi(e)`	standard normal cdf
`pow(e1, e2)`	$e1^{e2}$
`sin(e)`	sine(e)
`sqrt(e)`	$e^{1/2}$
`rank(v, s)`	number of components of v less than or equal to v_s
`ranked(v, s)`	the s^{th} smallest component of v
`round(e)`	nearest integer to e
`sd(v)`	standard deviation of components of v (n – 1 in denominator)
`step(e)`	1 if $e >= 0$; 0 otherwise

`sum(v)`	$\sum_i v_i$
`trunc(e)`	greatest integer less than or equal to *e*

A link function can also be specified acting on the left hand side of a logical node *e.g.*

```
logit(mu[i]) <- beta0 + beta1 * z1[i] + beta2 * z2[i] + b[i]
```

The following functions can be used on the left hand side of logical nodes as link functions: `log`, `logit`, `cloglog`, and `probit` (where `probit(x) <- y` is equivalent to `x <- phi(y)`).

It is important to keep in mind that logical nodes are included only for notational convenience – they cannot be given data or initial values (except when using the data transformation facility described below).

Deviance: A logical node called "deviance" is created automatically by WinBUGS: this stores –2 * log(likelihood), where 'likelihood' is the conditional probability of all data nodes given their stochastic parent nodes. This node can be monitored, and contributes to the DIC function – see DIC...

Arrays and indexing

[top | home]

Arrays are indexed by terms within square brackets. The four basic operators +, -, *, and / along with appropriate bracketing are allowed to calculate an integer function as an index, for example:

```
Y[(i + j) * k, l]
```

On the left-hand-side of a relation, an expression that always evaluates to a fixed value is allowed for an index, whether it is a constant or a function of data. On the right-hand-side the index can be a fixed value or a named node, which allows a straightforward formulation for mixture models in which the appropriate element of an array is 'picked' according to a random quantity (see Nested indexing and mixtures). However, functions of unobserved nodes are not permitted to appear directly as an index term (intermediate deterministic nodes may be introduced if such functions are required).

The conventions broadly follow those of S-Plus:

`n:m`	represents *n*, *n* + 1, ..., *m*.
`x[]`	represents all values of a vector *x*.
`y[,3]`	indicates all values of the third column of a two-dimensional array *y*.

Multidimensional arrays are handled as one-dimensional arrays with a constructed index. Thus functions defined on arrays must be over equally spaced nodes within an array: for example `sum(i, 1:4, k)`.

When dealing with unbalanced or hierarchical data a number of different approaches are possible – see Handling unbalanced datasets. The ideas discussed in Nested indexing and mixtures may also be helpful in this respect; the user should bear in mind, however, the 'contiguous elements' restriction described in Constraints on using certain distributions.

Repeated structures

[top | home]

Repeated structures are specified using a "for-loop". The syntax for this is:

```
for (i in a:b) {
  list of statements to be repeated for increasing values of loop-variable i
}
```

Note that neither `a` nor `b` may be stochastic – see here for a possible way to get round this.

Data transformations

[top | home]

Although transformations of data can always be carried out before using *WinBUGS*, it is convenient to be able

to try various transformations of dependent variables within a model description. For example, we may wish to try both *y* and *sqrt(y)* as dependent variables without creating a separate variable *z = sqrt(y)* in the data file.

The BUGS language therefore permits the following type of structure to occur:

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

Strictly speaking, this goes against the declarative structure of the model specification, with the accompanying exhortation to construct a directed graph and then to make sure that each node appears once and only once on the left-hand side of a statement. However, a check has been built in so that, when finding a logical node which also features as a stochastic node (such as `z` above), a stochastic node is created with the calculated values as fixed data.

We emphasise that this construction is only possible when transforming observed data (not a function of data and parameters) with no missing values.

This construction is particularly useful in Cox modelling and other circumstances where fairly complex functions of data need to be used. It is preferable for clarity to place the transformation statements in a section at the beginning of the model specification, so that the essential model description can be examined separately. See the Leuk and Endo examples.

Nested indexing and mixtures

[top | home]

Nested indexing can be very effective. For example, suppose *N* individuals can each be in one of *I* groups, and *g[1:N]* is a vector which contains the group membership. Then "group" coefficients *beta[i]* can be fitted using *beta[g[j]]* in a regression equation.

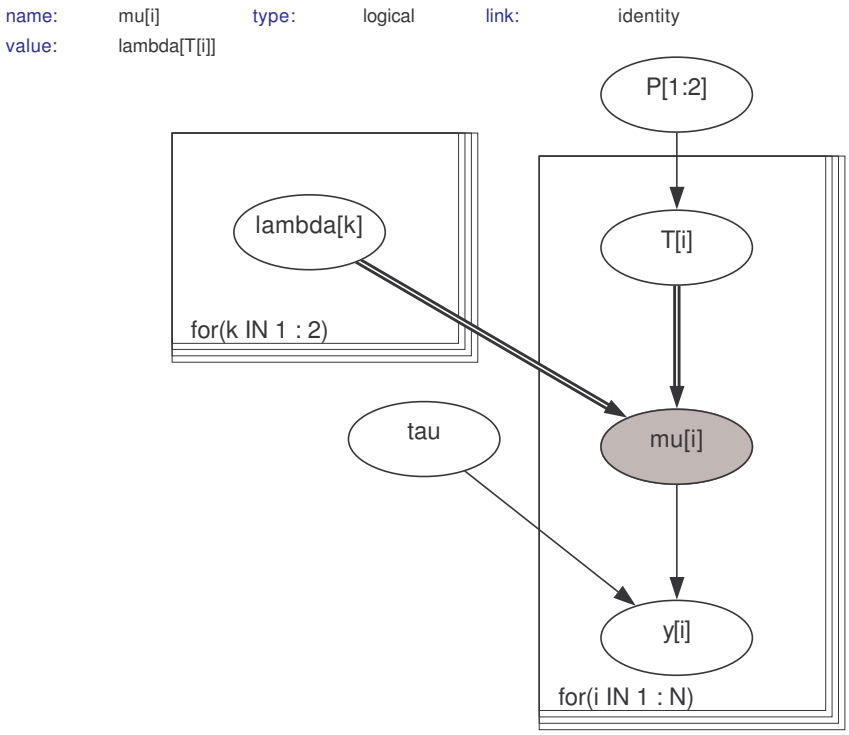
In the *BUGS* language, nested indexing can be used for the parameters of distributions: for example, the Eyes example concerns a normal mixture in which the i^{th} case is in an unknown group T_i which determines the mean λ_{T_i} of the measurement y_i. Hence the model is

$$T_i \sim \text{Categorical}(P)$$
$$y_i \sim \text{Normal}(\lambda_{T_i}, \tau)$$

which may be written in the BUGS language as

```
for (i in 1:N) {
    T[i] ~ dcat(P[])
    y[i] ~ dnorm(lambda[T[i]], tau)
}
```

However, when using Doodles the parameters of a distribution must be a node in the graph, and so an additional stage is needed to specify the mean $\mu_i = \lambda_{T_i}$, as shown in the graph below. (We emphasise the care required in establishing convergence of these notorious models.)



Vector parameters can also be identified dynamically, but currently only to a maximum of two dimensions. For example, if we wanted a two-state categorical variable `x` to have a vector of probabilities indexed by `i` and `j`, then we could write `x ~ dcat(p[i, j, 1:2])`. However, suppose we require three-level indexing, for example

```
a ~ dcat(p.a[1:2])
b ~ dcat(p.b[1:2])
c ~ dcat(p.c[1:2])
d ~ dcat(p.d[a, b, c, 1:2]
```

WinBUGS will not permit this, and so the index must be explicitly calculated:

```
d ~ dcat(p[k, 1:2])
k <- 8 * (a - 1) + 4 * (b - 1) + c
```

This 'calculated index' trick is useful in many circumstances.

Formatting of data

[top | home]

Data can be S-Plus format (see most of the examples) or, for data in arrays, in rectangular format.

The whole array must be specified in the file - it is not possible just to specify selected components.

Missing values are represented as `NA`.

All variables in a data file must be defined in a model, even if just left unattached to the rest of the model. In Doodles such variables can be left as constants: in a model description they can be assigned vague priors or allocated to dummy variables.

S-Plus format: This allows scalars and arrays to be named and given values in a single structure headed by key-word `list`. There must be no space after `list`.

For example, in the Rats example, we need to specify a scalar *xbar*, dimensions *N* and *T*, a vector *x* and a two-dimensional array *Y* with 30 rows and 5 columns. This is achieved using the following format:

```
list(
    xbar = 22, N = 30, T = 5,
```

```
        x = c(8.0, 15.0, 22.0, 29.0, 36.0),
        Y = structure(
            .Data = c(
                151, 199, 246, 283, 320,
                145, 199, 249, 293, 354,
                  ...................
                  ...................
                137, 180, 219, 258, 291,
                153, 200, 244, 286, 324),
            .Dim = c(30, 5)
            )
    )
```

See the examples for other use of this format.

WinBUGS reads data into an array by filling the right-most index first, whereas the S-Plus program fills the left-most index first. Hence *WinBUGS* reads the string of numbers c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) into a 2 * 5 dimensional matrix in the order

[i, j]th element of matrix	*value*
[1, 1]	1
[1, 2]	2
[1, 3]	3
.....	..
[1, 5]	5
[2, 1]	6
.....	..
[2, 5]	10

whereas S-Plus reads the same string of numbers in the order

[i, j]th element of matrix	*value*
[1, 1]	1
[2, 1]	2
[1, 2]	3
.....	..
[1, 3]	5
[2, 3]	6
.....	..
[2, 5]	10

Hence the ordering of the array dimensions must be reversed before using the S-Plus `dput` command to create a data file for input into *WinBUGS*.

For example, consider the 2 * 5 dimensional matrix

1	2	3	4	5
6	7	8	9	10

This must be stored in S-Plus as a 5 * 2 dimensional matrix:

```
> M
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

The S-Plus command

```
> dput(list(M=M), file="matrix.dat")
```

will then produce the following data file

```
list(M = structure(.Data = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10),
.Dim=c(5,2))
```

Edit the `.Dim` statement in this file from `.Dim=c(5,2)` to `.Dim=c(2,5)`. The file is now in the correct format to input the required 2 * 5 dimensional matrix into *WinBUGS*.

Now consider a 3 * 2 * 4 dimensional array

1	2	3	4
5	6	7	8

9	10	11	12
13	14	15	16

17	18	19	20
21	22	23	24

This must be stored in S-Plus as the 4 * 2 * 3 dimensional array:

```
> A
, , 1
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

, , 2
     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16

, , 3
     [,1] [,2]
[1,]   17   21
[2,]   18   22
[3,]   19   23
[4,]   20   24
```

The command

```
> dput(list(A=A), file="array.dat")
```

will then produce the following data file

```
list(A = structure(.Data = c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,
14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24), .Dim=c(4,2,3))
```

Edit the `.Dim` statement in this file from `.Dim=c(4,2,3)` to `.Dim=c(3,2,4)`. The file is now in the correct format to input the required 3 * 2 * 4 dimensional array into *WinBUGS* in the order

[i, j, k]th element of matrix	*value*
[1, 1, 1]	1
[1, 1, 2]	2

```
   .....          ..
[1, 1, 4]          4
[1, 2, 1]          5
[1, 2, 2]          6
   .....          ..
[2, 1, 3]         11
[2, 1, 4]         12
[2, 2, 1]         13
[2, 2, 2]         14
   .....          ..
[3, 2, 3]         23
[3, 2, 4]         24
```

Rectangular format: The columns for data in rectangular format need to be headed by the array name. The arrays need to be of equal size, and the array names must have explicit brackets: for example:

```
age[]    sex[]
 26        0
 52        1
     .....
 34        0
END
```

Note that the file must end with an 'END' statement, as shown above and below, and this must be followed by at least one blank line.

Multi-dimensional arrays can be specified by explicit indexing: for example, the Ratsy file begins

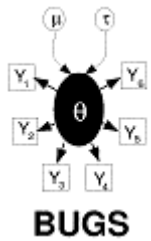
```
Y[,1]    Y[,2]    Y[,3]    Y[,4]    Y[,5]
 151      199      246      283      320
 145      199      249      293      354
 147      214      263      312      328
                  .......
 153      200      244      286      324
END
```

The first index position for any array must always be empty.

It is possible to load a mixture of rectangular and S-Plus format data files for the same model. For example, if data arrays are provided in a rectangular file, constants can be defined in a separate list statement (see also the Rats example with data files Ratsx and Ratsy).

(See here for details of how to handle unbalanced data.)

Note that programs exist for conversion of data from other packages: please see the BUGS resources web-page at `http://www.mrc-bsu.cam.ac.uk/bugs/weblinks/webresource.shtml`



DoodleBUGS: The Doodle Editor

Contents

General properties

Creating a node
Selecting a node
Deleting a node
Moving a node
Creating a plate
Selecting a plate
Deleting a plate
Moving a plate
Resizing a plate
Creating an edge
Deleting an edge
Moving a Doodle
Resizing a Doodle
Printing a Doodle

General properties

[top | home]

Doodles consist of three elements: *nodes*, *plates* and *edges*. The graph is built up out of these elements using the mouse and keyboard, controlled from the *Doodle* menu. The menu options are described below.

New: opens a new window for drawing Doodles. A dialog box opens allowing a choice of size of the Doodle graphic and the size of the nodes of the graph.

***Grid*:** the snap grid is on the centre of each node and each corner of each plate is constrained to lie on the grid.

Scale Model: shrinks the Doodle so that the size of each node and plate plus the separation between them is reduced by a constant factor. The Doodle will move towards the top left corner of the window. This command is useful if you run out of space while drawing the Doodle. Note that the Doodle will still be constrained by the snap grid, and if the snap is coarse then the Doodle could be badly distorted.

Remove Selection: removes the highlighting from the selected node or plate of the Doodle if any.

Write Code: opens a window containing the *BUGS* language equivalent to the Doodle. No variable declaration statement is produced as it is not needed within *WinBUGS*. **After constructing a Doodle, you are strongly recommended to use *Write Code* to check the structure is that which you intended.**

Creating a node

[top | home]

Point the mouse cursor to an empty region of the *Doodle* window and click. A flashing caret appears next to the blue word *name*. Typed characters will appear both at this caret and within the outline of the node.

Constant nodes can be given a name or a numerical value. The name of a node starts with a letter and can also contain digits and the period character "." The name must not contain two successive periods and must not end with a period. Vectors are denoted using a square bracket notation, with indices separated by commas. A colon-separated pair of integers is used to denote an index range of a multivariate node or plate.

When first created the node will be of type *stochastic* and have associated density *dnorm.*

The type of the node can be changed by clicking on the blue word *type* at the top of the doodle. A menu will drop down giving a choice of *stochastic*, *logical* and *constant* for the node type.

Stochastic: Associated with stochastic nodes is a density. Click on the blue word *density* to see the choice of densities available (this will not necessarily include all those available in the *BUGS* language and described in Distributions). For each density the appropriate name(s) of the parameters are displayed in blue. For some densities default values of the parameters will be displayed in black next to the parameter name. When edges are created pointing into a stochastic node these edges are associated with the parameters in a left to right order. To change the association of edges with parameters click on one of the blue parameter names, a menu will drop down from which the required edge can be selected. This drop down menu will also give the option of editing the parameter's default value.

Logical: Associated with logical nodes is a link, which can be selected by clicking on the blue word *link*. Logical nodes also require a *value* to be evaluated (modified by the chosen link) each time the value of the node is required. To type input in the value field click the mouse on the blue word *value*. The value field of a logical node corresponds to the right hand side of a logical relation in the *BUGS* language and can contain the same functions. The value field must be completed for all logical nodes.

We emphasise that the *value* determines the value of the node and the logical links in the Doodle are for cosmetic purposes only.

It is possible to define two nodes in the same Doodle with the same name – one as logical and one as stochastic – in order to use the data transformation facility described in Data transformations.

Constants: These need to be specified in the data file. If they only appear in value statements, they do not need to be explicitly represented in the Doodle.

Selecting a node [top | home]

Point the mouse cursor inside the node and left-mouse-click.

Deleting a node [top | home]

Select a node and press *ctrl + delete* key combination.

Moving a node [top | home]

Select a node and then point the mouse into selected node. Hold mouse button down and drag node. The cursor keys may also be used.

Creating a plate [top | home]

Point the mouse cursor to an empty region of the *Doodle* window and click while holding the *ctrl* key down.

Selecting a plate [top | home]

Point the mouse into the lower or right hand border of the plate and click.

Deleting a plate [top | home]

Select a plate and press *ctrl + delete* key combination.

Moving a plate [top | home]

Select a plate and then point the mouse into the lower or right hand border of the selected plate. Hold the mouse button down and drag the plate.

Resizing a plate [top | home]

Select a plate and then point the mouse into the small region at the lower right where the two borders intercept. Hold the mouse button down and drag to resize the plate.

Creating an edge [top | home]

Select node into which the edge should point and then click into its parent while holding down the *ctrl* key.

Deleting an edge [top | home]

Select node whose incoming edge is to be deleted and then click into its parent while holding down the *ctrl* key.

Moving a Doodle [top | home]

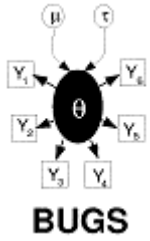
After constructing a Doodle, it can be moved into a document that may also contain data, initial values, and other text and graphics. This can be done by choosing *Select Document* from the *Edit* menu, and then either copying and pasting, or dragging, the Doodle.

Resizing a Doodle [top | home]

To change the size of Doodle which is already in a document containing text, click once into the Doodle with the left mouse button. A narrow border with small solid squares at the corners and mid-sides will appear. Drag one of these squares with the mouse until the Doodle is of the required size.

Printing a Doodle [top | home]

If a Doodle is in its own document, it may be printed directly from the *File* menu. If a postscript version of a Doodle is required, you could install a driver for a postscript printer (say Apple LaserWriter), but set it up to *print to file* (checking the paper size is appropriate). Alternatively Doodles can be copied to a presentation or word-processing package and printed from there.



The Model Menu

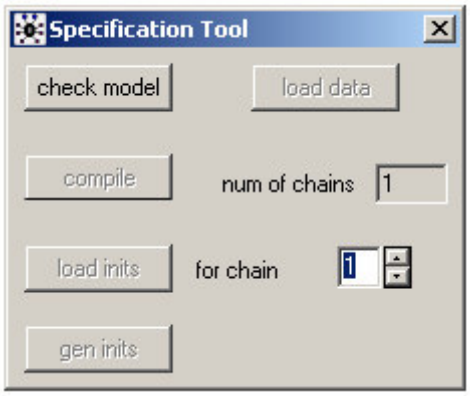
Contents

General properties
Specification...
Update...
Monitor Metropolis
Save State
Seed...
Script

General properties [top | home]

The commands in this menu either apply to the whole statistical model or open dialog boxes. This menu is only on display if the focus view is a text window or a Doodle.

Specification... [top | home]



This non-modal dialog box acts on the focus view.

check model: If the focus view contains text, *WinBUGS* assumes the model is specified in the *BUGS* language. The *check model* button parses the *BUGS* language description of the statistical model, as in the classic version of *BUGS*. If a syntax error is detected the cursor is placed where the error was found and a description of the error is given on the status line (lower left corner of screen). If a stretch of text is highlighted the parsing starts from the first character highlighted (*i.e.* highlight the word `model`) else parsing starts at the top of the window.

If the focus view contains a Doodle *(i.e.* the Doodle has been selected and is surrounded by a hairy border), *WinBUGS* assumes the model has been specified graphically. If a syntax error is detected the node where the error was found is highlighted and a description of the error is given on the status line.

load data: The *load data* button acts on the focus view; it will be greyed out unless the focus view contains text.

Data can be identified in two ways:

1) if the data is in a separate document, the window containing that document needs to be in the focus view (the windows title bar will be coloured not grey) when the *load data* command is used;
2) if the data is specified as part of a document, the first character of the data (either `list` if in S-Plus format, or the first array name if in rectangular format) must be highlighted and the data will be read from there on.

See here for details of data formats

Any syntax errors or data inconsistencies are displayed in the status line. Corrections can be made to the data without returning to the *check model* stage. When the data have been loaded successfully, 'Data Loaded' should appear in the status line.

The *load data* button becomes active once a model has been successfully checked, and ceases to be active once the model has been successfully compiled.

num of chains: The number of chains to be simulated can be entered into the text entry field next to the caption *num of chains.* This field can be typed in after the model has been checked and before the model has been compiled. By default, one chain is simulated.

compile: The *compile* button builds the data structures needed to carry out Gibbs sampling. The model is checked for completeness and consistency with the data.

A node called 'deviance' is automatically created which calculates minus twice the log-likelihood at each iteration, up to a constant. This node can be monitored by typing `deviance` in the *Samples* dialog box.

This command becomes active once the model has been successfully checked, and when the model has been successfully compiled, 'model compiled' should appear in the status line.

load inits: The *load inits* button acts on the focus view: it will be greyed out unless the focus view contains text. The initial values will be loaded for the chain indicated in the text entry field to the right of the caption *for chain*. The value of this text field can be edited to load initial values for any of the chains.

Initial values are specified in exactly the same way as data files. If some of the elements in an array are known (say because they are constraints in a parameterisation), those elements should be specified as missing (`NA`) in the initial values file.

This command becomes active once the model has been successfully compiled, and checks that initial values are in the form of an S-Plus object or rectangular array and that they are consistent with any previously loaded data. Any syntax errors or inconsistencies in the initial value are displayed.

If, after loading the initial values, the model is fully initialized this will be reported by displaying the message *initial values loaded: model initialized*. Otherwise the status line will show the message *initial values loaded: model contains uninitialized nodes*. The second message can have several meanings:

a) If only one chain is simulated it means that the chain contains some nodes that have not been initialized yet.
b) If several chains are to be simulated it could mean that no initial values have been loaded for one of the chains.

In either case further initial values can be loaded, or the *gen inits* button can be pressed to try and generate

initial values for *all* the uninitialized nodes in *all* the simulated chains.

Generally it is recommended to load initial values for all fixed effect nodes (founder nodes with no parents) for all chains, initial values for random effects can be generated using the *gen inits* button.

This *load inits* button can still be executed once Gibbs sampling has been started. It will have the effect of starting the sampler out on a new trajectory. A modal warning message will appear if the command is used in this context.

gen inits: The *gen inits* button attempts to generate initial values by sampling either from the prior or from an approximation to the prior. In the case of discrete variables a check is made that a configuration of zero probability is not generated. This command will generate extreme values if any of the priors are very vague. If the command is successful the message 'initial values generated: model initialized' is displayed otherwise the message 'could not generate initial values' is displayed.

The *gen inits* button becomes active once the model has been successfully compiled, and will cease to be active once the model has been initialized.

Update... [top | home]

This command will become active once the model has been compiled and initialized, and has fields:

updates: number of MCMC updates to be carried out.

refresh: the number of updates between redrawing the screen.

thin: the samples from every k^{th} iteration will be stored, where *k* is the value of ***thin***. Setting $k > 1$ can help to reduce the autocorrelation in the sample, but there is no real advantage in thinning except to reduce storage requirements and the cost of handling the simulations when very long runs are being carried out.

update: Click to start updating the model. Clicking on *update* during sampling will pause the simulation after the current block of iterations, as defined by *refresh*, has been completed; the number of updates required can then be changed if needed. Clicking on update again will restart the simulation. This button becomes active when the model has been successfully compiled and given initial values.

iteration: shows the total number of iterations stored after thinning – not the actual number of iterations carried out. In this respect *updates* represents the required number of samples rather than MCMC updates: for example, if 100 samples are requested (via *updates* = 100) and *thin* is set equal to 10, then 10 * 100 = 1000 iterations will actually be carried out, of which 100 (every 10th) will be stored.

over relax: click on this box (a tick will then appear) to select an over-relaxed form of MCMC (Neal, 1998) which will be executed where possible. This generates multiple samples at each iteration and then selects one that is negatively correlated with the current value. The time per iteration will be increased, but the within-chain correlations should be reduced and hence fewer iterations may be necessary. However, this method is not always effective and should be used with caution. The auto-correlation function may be used to check whether the mixing of the chain is improved.

adapting: This box will be ticked while the Metropolis or slice-sampling MCMC algorithm is in its initial tuning phase where some optimization parameters are tuned. All summary statistics for the model will ignore information from this adapting phase. The Metropolis and slice-sampling algorithms have adaptive phases of 4000 and 500 iterations respectively which will be discarded from all summary statistics. For details of how to

change these default settings please see Update options...

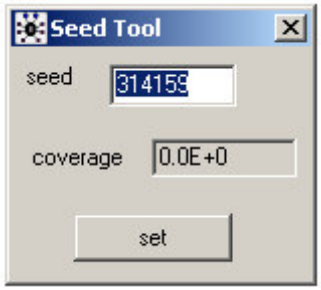
Monitor Metropolis [top | home]

This command is only active if the Metropolis algorithm is being used for the model. This shows the minimum, maximum and average acceptance rate averaged over 100 iterations as the Metropolis algorithm adapts over the first N iterations. The rate should lie between the two horizontal lines. The first N iterations of the simulation cannot be used for statistical inference. The default value of N is 4000 (see Update options... for details of how to change this value).

Save State [top | home]

Opens a window showing the current state of all the stochastic variables in the model, displayed in S-Plus format. This could then be used as an initial value file for future runs.

Seed... [top | home]



Opens a non-modal dialog box containing:

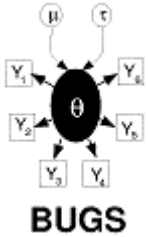
seed: a text entry field where the new seed of the random number generator can be typed.

coverage: the pseudo-random number generator used by *WinBUGS* generates a finite (albeit very long) sequence of distinct numbers, which would eventually be repeated if the sampler were run for a sufficiently long time. The *coverage* field shows the percentage of this sequence covered during the current *WinBUGS* session.

set: sets the seed to the value entered into the dialog box. The seed must be set *after* the model is checked (via 'check model' – see Specification...) in order for the new value to apply to the current analysis.

Script [top | home]

The Script command is used to run "model scripts" in batch-mode: if the focus-view contains a series of *WinBUGS* batch-mode commands then selecting this command from the *Model* menu will cause the script to be executed. Please see Batch-mode: Scripts for full details.



The Inference Menu

Contents

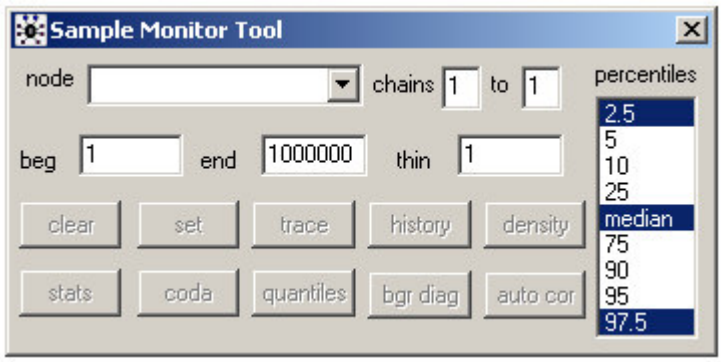
General properties
Samples...
Compare...
Correlations...
Summary...

Rank...
DIC...

General properties [top | home]

These menu items open dialog boxes for making inferences about parameters of the model. The commands are divided into three sections: the first three commands concern an entire set of monitored values for a variable; the next two commands are space-saving short-cuts that monitor running statistics; and the final command, DIC..., concerns evaluation of the *Deviance Information Criterion* proposed by Spiegelhalter *et al.* (2002). **Users should ensure their simulation has converged before using Summary..., Rank... or DIC...** Note that if the MCMC simulation has an adaptive phase it will not be possible to make inference using values sampled before the end of this phase.

Samples... [top | home]



This command opens a non-modal dialog for analysing stored samples of variables produced by the MCMC simulation. The fields are:

node: The variable of interest must be typed in this text field. If the variable of interest is an array, slices of the array can be selected using the notation `variable[lower0:upper0, lower1:upper1, ...]`. The buttons at the bottom of the dialog act on this variable. A star '*' can be entered in the node text field as shorthand for all the stored samples.

WinBUGS generally automatically sets up a logical node to measure a quantity known as *deviance*; this may be accessed, in the same way as any other variable of interest, by typing its name, i.e. "deviance", in the *node* field of the *Sample Monitor Tool*. The definition of deviance is –2 * log(likelihood): 'likelihood' is defined as p(*y*| *theta*), where *y* comprises all stochastic nodes given values (i.e. data), and *theta* comprises the *stochastic parents* of *y* – 'stochastic parents' are the stochastic nodes upon which the distribution of *y* depends, when collapsing over all logical relationships.

beg and ***end:*** numerical fields are used to select a subset of the stored sample for analysis.

thin: numerical field used to select every k^{th} iteration of each chain to contribute to the statistics being calculated, where *k* is the value of the field. Note the difference between this and the thinning facility on the *Update Tool* dialog box: when thinning via the *Update Tool* we are *permanently* discarding samples as the MCMC simulation runs, whereas here we have already generated (and stored) a suitable number of (posterior) samples and may wish to discard some of them only temporarily. Thus, setting $k > 1$ here will not have any impact on the storage (memory) requirements of processing long runs; if you wish to reduce the number of samples actually stored (to free-up memory) you should thin via the *Update Tool*.

chains . to .: can be used to select the chains which contribute to the statistics being calculated.

clear: removes the stored values of the variable from computer memory.

set: must be used to start recording a chain of values for the variable.

trace: plots the variable value against iteration number. This trace is dynamic, being redrawn each time the screen is redrawn.*

history: plots out a complete trace for the variable.*

The next six buttons will be greyed out if the MCMC simulation is in an adaptive phase.

density: plots a smoothed kernel density estimate for the variable if it is continuous or a histogram if it is discrete.*

auto cor: plots the autocorrelation function of the variable out to lag-50. The values underlying these can be listed to a window by double-clicking on the plot followed by *ctrl*-left-mouse-click: if multiple chains are being simulated the values for each chain will be given.*

stats: produces summary statistics for the variable, pooling over the chains selected. The required percentiles can be selected using the *percentile* selection box. The quantity reported in the MC error column gives an estimate of $\sigma / N^{1/2}$, the Monte Carlo standard error of the mean. The batch means method outlined by Roberts (1996; p.50) is used to estimate σ.

coda: dumps out an ascii representation of the monitored values suitable for use in the *CODA* S-Plus diagnostic package. An output file for each chain is produced, corresponding to the `.out` files of *CODA*, showing the iteration number and value (to four significant figures). There is also a file containing a description of which lines of the `.out` file correspond to which variable – this corresponds to the *CODA* `.ind` file. These can be named accordingly and saved as text files for further use. (Care may be required to stop the Windows system adding a `.txt` extension when saving: enclosing the required file name in quotes should prevent this.)

quantiles: plots out the running mean with running 95% confidence intervals against iteration number.*

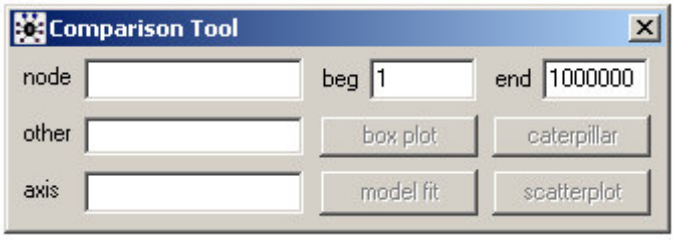
bgr diag: calculates the Gelman-Rubin convergence statistic, as modified by Brooks and Gelman (1998). The width of the central 80% interval of the pooled runs is green, the average width of the 80% intervals within the individual runs is blue, and their ratio *R* (= pooled / within) is red - for plotting purposes the pooled and within interval widths are normalised to have an overall maximum of one. The statistics are calculated in bins of length 50: *R* would generally be expected to be greater than 1 if the starting values are suitably over-dispersed. Brooks and Gelman (1998) emphasise that one should be concerned both with convergence of *R* to 1, and with convergence of both the pooled and within interval widths to stability.

The values underlying these plots can be listed to a window by double-clicking on the plot followed by *ctrl*-left-mouse-click on the window.*

* See WinBUGS Graphics for details of how to customize these plots.

Compare... [top | home]

Select *Compare...* from the *Inference* menu to open the *Comparison Tool* dialog box. This is designed to facilitate comparison of elements of a vector of nodes (with respect to their posterior distributions).



node: defines the vector of nodes to be compared with each other. As the comparisons are with respect to posterior distributions, *node* must be a monitored variable.

other: where appropriate, *other* defines a vector of reference points to be plotted alongside each element of *node* (on the same scale), for example, *other* may be the observed data in a *model fit* plot (see below). The elements of *other* may be either monitored variables, in which case the posterior mean is plotted, or they may be observed/known.

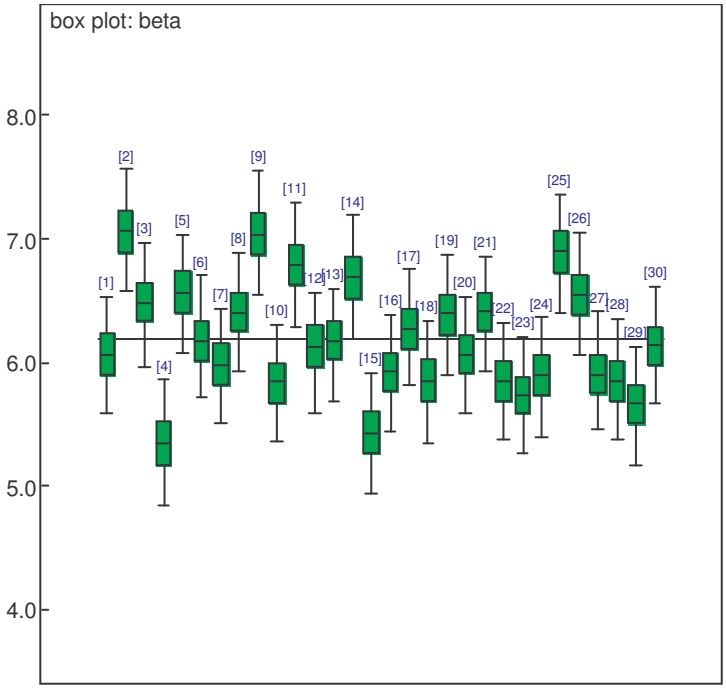
axis: where appropriate, *axis* defines a set of values against which the elements of *node* (and *other*) should be

plotted. Each element of *axis* must be either known/observed or, alternatively, a monitored variable, in which case the posterior mean is used.

Note: *node*, *other*, and *axis* should all have the same number of elements!

beg and ***end:*** are used to select the subset of stored samples from which the desired plot should be derived.

box plot: this command button produces a single plot in which the posterior distributions of all elements of *node* are summarised side by side. For example,



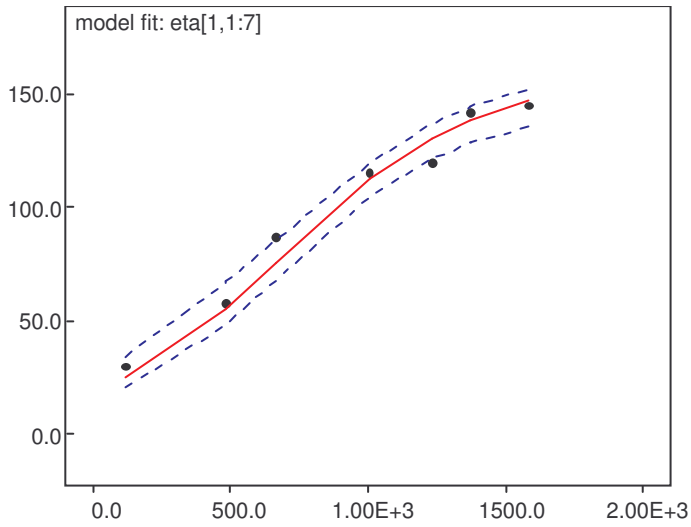
By default, the distributions are plotted in order of the corresponding variable's index in *node* and are also labelled with that index. Boxes represent inter-quartile ranges and the solid black line at the (approximate) centre of each box is the mean; the arms of each box extend to cover the central 95 per cent of the distribution – their ends correspond, therefore, to the 2.5% and 97.5% quantiles. (Note that this representation differs somewhat from the traditional.)

(The default value of the baseline shown on the plot is the global mean of the posterior means.)

There is a special "property editor" available for box plots, as indeed there is for all graphics generated via the *Comparison Tool*. This can be used to interact with the plot and change the way in which it is displayed, for example, it is possible to rank the distributions by their means or medians and/or plot them on a logarithmic scale. For full details, please see Box plot.

caterpillar: a "caterpillar" plot is conceptually very similar to a box plot. The only significant differences are that the inter-quartile ranges are not shown and the default scale axis is now the x-axis – each distribution is summarised by a horizontal line representing the 95% interval and a dot to show where the mean is. (Again, the default baseline – in red – is the global mean of the posterior means.) Due to their greater simplicity caterpillar plots are typically preferred over box plots when the number of distributions to be compared is large. See Caterpillar plot for details of how to change the properties of caterpillar plots.

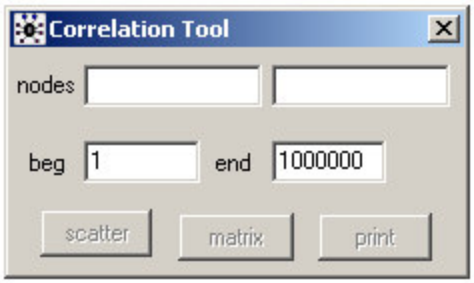
model fit: the elements of *node* (and *other* if specified) are treated as a time-series, defined by (increasing values of) the elements of *axis*. The posterior distribution of each element of *node* is summarised by the 2.5%, 50% and 97.5% quantiles. Each of these quantities is joined to its direct neighbours (as defined by *axis*) by straight lines (solid red in the case of the median and dashed blue for the 95% posterior interval) to form a piecewise linear curve – the 'model fit'. In cases where *other* is specified, its values are also plotted, using black dots, against the corresponding values of *axis*, e.g.



Where appropriate, either or both axes can be changed to a logarithmic scale via a property editor – see Model fit plot.

scatterplot: by default, the posterior means of *node* are plotted (using blue dots) against the corresponding values of *axis* and an exponentially weighted smoother is fitted. Numerous properties of a scatterplot can be modified using the editor described in Scatterplot – for example, the smoothing parameter may be changed, or the smoother may be replaced by a different type of line, or 95% posterior intervals for *node* may be displayed, etc.

Correlations... [top | home]



This non-modal dialog box is used to plot out the relationship between the simulated values of selected variables, which must have been monitored.

nodes: scalars or arrays may be entered in each box, and all combinations of variables entered in the two boxes are selected. If a single array is given, all pairwise correlations will be plotted.

scatter: produces a scatter plot of the individual simulated values.

matrix: produces a matrix summary of the cross-correlations.

print: opens a new window containing the coefficients for all possible correlations among the selected variables.

The calculations may take some time.

Summary... [top | home]

This non modal dialog box is used to calculate running means, standard deviations and quantiles. The commands in this dialog are less powerful and general than those in the *Sample Monitor Tool*, but they also require much less storage (an important consideration when many variables and/or long runs are of interest).

node: The variable of interest must be typed in this text field.

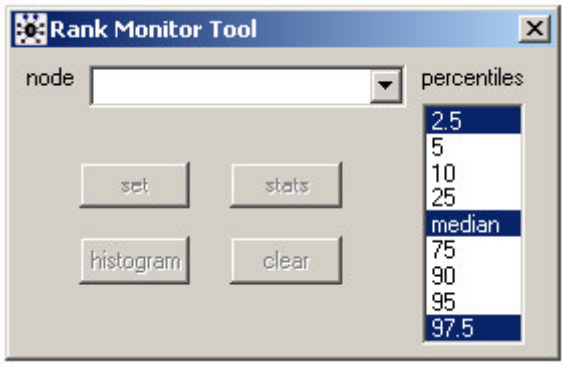
set: starts recording the running totals for *node*.

stats: displays the running means, standard deviations, and 2.5%, 50% (median) and 97.5% quantiles for *node*. **Note that these running quantiles are calculated via an approximate algorithm (see here for details) and should therefore be used with caution.**

means: displays the running means for *node* in a comma delimited form. This can be useful for passing the results to other statistical or display packages.

clear: removes the running totals for *node*.

Rank... [top | home]



This non-modal dialog box is used to store and display the ranks of the simulated values in an array.

node: the variable to be ranked must be typed in this text field (must be an array).

set: starts building running histograms to represent the rank of each component of *node*. An amount of storage proportional to the square of the number of components of *node* is allocated. Even when *node* has thousands of components this can require less storage than calculating the ranks explicitly in the model specification and storing their samples, and it is also much quicker.

stats: summarises the distribution of the ranks of each component of the variable *node*. The quantiles highlighted in the percentile selection box are displayed.

histogram: displays the empirical distribution of the simulated rank of each component of the variable *node*.

clear: removes the running histograms for *node*.

DIC... [top | home]

The *DIC Tool* dialog box is used to evaluate the *Deviance Information Criterion* (DIC; Spiegelhalter *et al.*, 2002) and related statistics – these can be used to assess model complexity and compare different models. Most of the examples packaged with *WinBUGS* contain an example of their usage.

It is important to note that DIC assumes the posterior mean to be a good estimate of the stochastic parameters. If this is not so, say because of extreme skewness or even bimodality, then DIC may not be appropriate. There are also circumstances, such as with mixture models, in which WinBUGS will not permit the calculation of DIC and so the menu option is greyed out. Please see the WinBUGS 1.4 web-page for current restrictions:

http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml

set: starts calculating DIC and related statistics – the user should ensure that convergence has been achieved before pressing *set* as all subsequent iterations will be used in the calculation.

clear: if a DIC calculation has been started (via *set*) this will clear it from memory, so that it may be restarted later.

DIC: displays the calculated statistics, as described below; please see Spiegelhalter *et al.* (2002) for full details; the section Tricks: Advanced Use of the BUGS Language also contains some comments on the use of DIC.

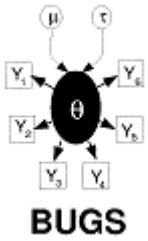
The DIC button generates the following statistics:

Dbar: this is the posterior mean of the deviance, which is exactly the same as if the node 'deviance' had been monitored (see here). This deviance is defined as –2 * log(likelihood): 'likelihood' is defined as p(*y*|*theta*), where *y* comprises all stochastic nodes given values (i.e. data), and *theta* comprises the *stochastic parents* of *y* – 'stochastic parents' are the stochastic nodes upon which the distribution of *y* depends, when collapsing over all logical relationships.

Dhat: this is a point estimate of the deviance (–2 * log(likelihood)) obtained by substituting in the posterior means *theta.bar* of *theta*: thus Dhat = –2 * log(p(*y*|*theta.bar*)).

pD: this is 'the effective number of parameters', and is given by **pD = Dbar – Dhat**. Thus pD is the posterior mean of the deviance minus the deviance of the posterior means.

DIC: this is the 'Deviance Information Criterion', and is given by **DIC = Dbar + pD = Dhat + 2 ∗ pD**. The model with the smallest DIC is estimated to be the model that would best predict a replicate dataset of the same structure as that currently observed.



The Info Menu

Contents

General properties
Open Log
Clear Log

Node info...
Components

General properties [top | home]

This menu allows the user to obtain more information about how the software is working.

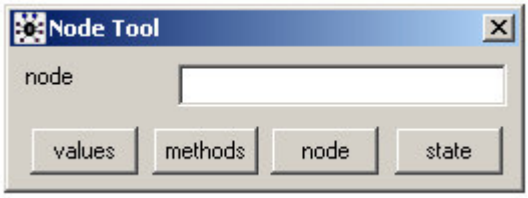
Open Log [top | home]

This option opens a log window to which error and status information is written.

Clear Log [top | home]

This option clears all the information displayed in the log window.

Node info... [top | home]



This dialog box allows information about particular nodes in the model to be obtained.

node: the name of the node for which information is required should be typed here.

values: displays the current value of the node (for each chain) in the log window.

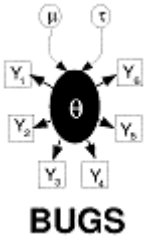
methods: displays the type of updater used to sample from the node (if appropriate) in the log window.

node: displays the type used to represent the node in the log window.

state: opens a trap window showing the current state of the internal data structures used to represent the node; this is for debugging purposes only – it is of no practical use to the user!

Components [top | home]

Displays all the components (dynamic link libraries) in use.

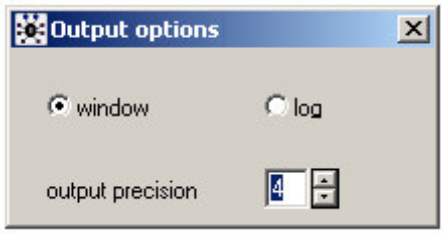


The Options Menu

Contents

Output options...
Blocking options...
Update options...

Output options... [top | home]



window or ***log:*** if *window* is selected then a new window will be opened for each new piece of output (statistics, traces, etc.); if *log* is selected then all output will be pasted into a single log file.

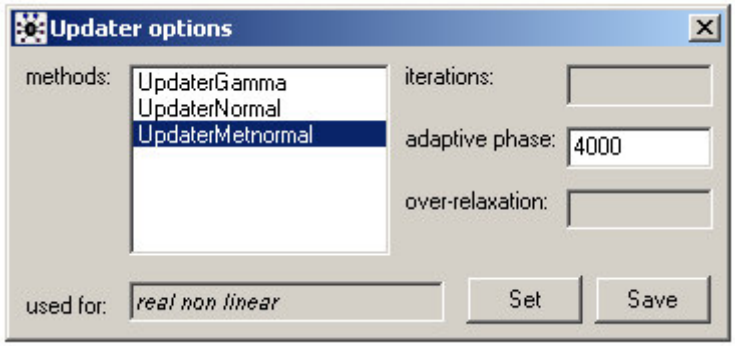
output precision: the number of significant digits in numerical output.

Blocking options... [top | home]

If the box is checked then, where possible, WinBUGS will use the multivariate updating method described in Gamerman (1997) to generate new values for blocks of fixed-effect parameters.

Update options... [top | home]

Once a model has been compiled, the various updating algorithms required in order to perform the MCMC simulation may be 'tuned' somewhat via the *Updater options* dialog box (select *Update options...* from the *Options* menu):



methods: this "selection box" shows the system names of all updating methods required/selected for the current problem. All other components of the *Updater options* dialog box (i.e. fields and command buttons) pertain to the currently selected item in *methods*.

used for: this text field simply describes what type of node the currently selected method is generally used for.

Note: it is now possible to change the sampling methods for certain classes of distribution, although this is delicate and should be done carefully – please see Changing MCMC Defaults (advanced users only) for details.

iterations: some updating algorithms entail iterative procedures that terminate when some relevant criterion is satisfied. It is always possible, however, that within a given Gibbs iteration this criterion cannot be satisfied in reasonable time. In such cases, rather than allow the computer to 'hang', it is preferable to specify a maximum number of iterations allowed before an error message is generated. This maximum number of iterations is displayed in the *iterations* field, which may be edited. In cases where iterative procedures are not required the *iterations* field will be greyed-out.

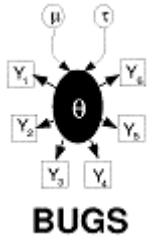
adaptive phase: some updating methods, such as Metropolis-Hastings, have an adaptive phase during which their internal parameters are tuned based on information gained from the chain(s) generated so far. All samples

generated during an adaptive phase should be discarded when drawing inferences but sometimes the default adaptive phase is longer than necessary, meaning that the sampler is somewhat wasteful. (Alternatively, the default adaptive phase may not be sufficiently long to allow proper tuning.) When the *adaptive phase* field is not greyed-out (indicating that the currently selected method in *methods* requires tuning) it displays the length of the adaptive phase in iterations (Gibbs cycles) – this may be edited by the user.

over-relaxation: many updating methods are capable of generating *over-relaxed* samples. Here, at each iteration, a number of candidate samples is generated and one that is negatively correlated with the current value is selected – the time per iteration will be increased but within-chain correlations should be reduced. The number of candidate samples (including the current value) is displayed in the *over-relaxation* field.

Set: The *Set* command button applies the values shown in *iterations*, *adaptive phase*, and *over-relaxation* to the updating method currently selected in *methods* for the current model – if a new model is loaded or if WinBUGS is shut-down and re-started then the software will revert to its default values.

Save: The *Save* button also applies the values shown in *iterations*, *adaptive phase*, and *over-relaxation* to the currently selected method, but it also saves those values as defaults for that method - the next time that that method is used the new values will be selected automatically.



Batch-mode: Scripts

The Scripting Language

As an alternative to the menu/dialog box interface of *WinBUGS* a scripting language has been provided. This language can be useful for automating routine analysis. The language works in effect by writing values into fields and clicking on buttons in relevant dialog boxes. Note that it is possible to combine use of this scripting language with use of the menu/dialog box interface.

To make use of the scripting language for a specific problem, a minimum of four files are required: the script itself; a file containing the BUGS language representation of the model; a file (or several) containing the data; and for each chain a file containing initial values. Each file may be in either native *WinBUGS* format (.odc) or text format, in which case it must have a .txt extension.

The shortcut *BackBUGS* has been set up to run the commands contained in the file script.odc (in the root directory of *WinBUGS*) when it is double-clicked. Thus a *WinBUGS* session may be embedded within any software component that can execute the *BackBUGS* shortcut.

Below is a list of currently implemented commands in the scripting language. Alongside each is a brief synopsis of its menu/dialog box equivalent: first we have the menu name and then the associated menu option; then, an underlined name corresponds to a button in a dialog box, whereas a name without an underline corresponds to a dialog box field set equal to the value of the quantity in italics that the => points to.

If the menu/dialog box equivalent of the specified script command would normally be greyed out, because of inappropriate timing, for example, then the script command will not execute and an error message will be produced instead.

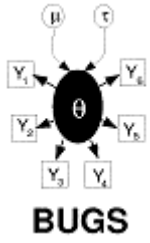
script command	**menu/dialog box equivalent**
display(*option*)	Options –> Output options... –> *option* = "window" or "log"
check(*model file*)	Model –> Specification... –> check model

data(*data file*)	Model –> Specification... –> load data
blockfe(*option*)	Options –> Blocking options... –> fixed effects => *option*
compile(*chains*)	Model –> Specification... –> num of chains => *chains* + compile
inits(*chain*, *inits file*)	Model –> Specification... –> for chain => *chain* + load inits
gen.inits()	Model –> Specification... –> gen inits
update(*iterations*)	Model –> Update... –> updates => *iterations* + update
refresh(*every*)	Model –> Update... –> refresh => *every*
over.relax(*option*)	Model –> Update... –> over relax => *option*
thin.updater(*thin*)	Model –> Update... –> thin => *thin*
beg(*iter*)	Inference –> Samples... –> beg => *iter*
end(*iter*)	Inference –> Samples... –> end => *iter*
first(*iter*)	Inference –> Samples... –> chains => *iter*
last(*iter*)	Inference –> Samples... –> to => *iter*
thin.samples(*thin*)	Inference –> Samples... –> thin => *thin*
set(*node*)	Inference –> Samples... –> node => *node* + set
clear(*node*)	Inference –> Samples... –> node => *node* + clear
stats(*node*)	Inference –> Samples... –> node => *node* + stats
density(*node*)	Inference –> Samples... –> node => *node* + density
autoC(*node*)	Inference –> Samples... –> node => *node* + auto cor
trace(*node*)	Inference –> Samples... –> node => *node* + trace
history(*node*)	Inference –> Samples... –> node => *node* + history
quantiles(*node*)	Inference –> Samples... –> node => *node* + quantiles
gr(*node*)	Inference –> Samples... –> node => *node* + bgr diag
coda(*node*, *file stem*)[1]	Inference –> Samples... –> node => *node* + coda
set.summary(*node*)	Inference –> Summary... –> node => *node* + set
stats.summary(*node*)	Inference –> Summary... –> node => *node* + stats
mean.summary(*node*)	Inference –> Summary... –> node => *node* + mean
clear.summary(*node*)	Inference –> Summary... –> node => *node* + clear
set.rank(*node*)	Inference –> Rank... –> node => *node* + set
stats.rank(*node*)	Inference –> Rank... –> node => *node* + stats
hist.rank(*node*)	Inference –> Rank... –> node => *node* + histogram
clear.rank(*node*)	Inference –> Rank... –> node => *node* + clear
dic.set()	Inference –> DIC... –> set
dic.stats()	Inference –> DIC... –> DIC
quit()	File –> Exit
save(*file*)[2]	File –> Save As... –> File name: => *file* + Save
script(*file*)[3]	Model –> Script

[1] If the *file stem* is left blank the CODA output is written to windows (one for each chain plus one for the index). If the *file stem* is not blank then the CODA output is written to separate text files, each starting with *file stem.*

[2] If the file ends with ".txt" the log window is saved to a text file (with all graphics, fonts, etc., stripped out); otherwise the log window is saved as is.

[3] Runs the script stored in *file*.



Tricks: Advanced Use of the BUGS Language

Contents

Specifying a new sampling distribution
Specifying a new prior distribution
Specifying a discrete prior on a set of values
Using pD and DIC
Mixtures of models of different complexity
Where the size of a set is a random quantity
Assessing sensitivity to prior assumptions
Modelling unknown denominators
Handling unbalanced datasets
Learning about the parameters of a Dirichlet distribution
Use of the "cut" function

Specifying a new sampling distribution

[top | home]

Suppose we wish to use a sampling distribution that is not included in the list of standard distributions, in which an observation x[i] contributes a likelihood term L[i]. We may use the "zeros trick": a Poisson(phi) observation of zero has likelihood exp(-phi), so if our observed data is a set of 0's, and phi[i] is set to –log(L[i]), we will obtain the correct likelihood contribution. (Note that phi[i] should always be > 0 as it is a Poisson mean, and so we may need to add a suitable constant to ensure that it is positive.) This trick is illustrated by an example new-sampling in which a normal likelihood is constructed (using the zeros trick) and compared to the standard analysis.

```
C <- 10000            # this just has to be large enough to ensure all phi[i]'s > 0
for (i in 1:N) {
    zeros[i] <- 0
    phi[i] <- -log(L[i]) + C
    zeros[i] ~ dpois(phi[i])
}
```

This trick allows arbitrary sampling distributions to be used, and is particularly suitable when, say, dealing with truncated distributions.

A new observation x.pred can be predicted by specifying it as missing in the data-file and assigning it a uniform prior, e.g.

```
x.pred ~ dflat()        # improper uniform prior on new x
```

However our example shows that this method can be very inefficient and give a very high MC error.

An alternative to using 'zeros' is the "ones trick", where the data is a set of 1's assumed to be the results of Bernoulli trials with probabilities p[i]. By making each p[i] proportional to L[i] (i.e. by specifying a scaling constant large enough to ensure all p[i]'s are < 1) the required likelihood term is provided.

```
C <- 10000            # this just has to be large enough to ensure all p[i]'s < 1
for (i in 1:N) {
    ones[i] <- 1
    p[i] <- L[i] / C
    ones[i] ~ dbern(p[i])
}
```

Specifying a new prior distribution

[top | home]

If, for a parameter *theta*, say, we want to use a prior distribution that is not part of the standard set, then we can use the 'zeros' trick (see above) at the prior level. A single zero Poisson observation (with mean phi = phi(*theta*)) contributes a term exp(-phi) to the likelihood for *theta*; when this is combined with a 'flat' prior for *theta* the correct prior distribution results.

```
zero <- 0
theta ~ dflat()
```

```
phi <- expression for –log(desired prior for theta)
zero ~ dpois(phi)
```

This is illustrated in new-prior by an example in which a normal prior is constructed using the zeros trick and the results are compared to the standard formulation. It is important to note that this method produces high auto-correlation, poor convergence and high MC error, so it is computationally slow and long runs are necessary.

Specifying a discrete prior on a set of values

[top | home]

Suppose you want a parameter D to take one of a set of values, d[1], ..., d[K], say, with probabilities p[1], ..., p[K]. Then specify the arrays d[1:K] and p[1:K] in a data-file (or in the model) and use:

```
M ~ dcat(p[])        # choose which element of d[] to use.
D <- d[M]
```

This is illustrated by an example t-df of learning about the degrees of freedom of a t-distribution.

If the discrete prior is put on too coarse a grid, then there may be numerical problems (crashes), unless good initial values are provided, and very poor mixing. It is therefore advised to either use a continuous prior or a discrete prior on a fine grid – see the t-df example.

Using pD and DIC

[top | home]

Here we make a number of observations regarding the use of DIC and pD – for a full discussion see Spiegelhalter *et al.* (2002):

1) DIC is intended as a generalisation of Akaike's Information Criterion (AIC). For non-hierarchical models, pD should be approximately the true number of parameters.

2) Slightly different values of Dhat (and hence pD and DIC) can be obtained depending on the parameterisation used for the prior distribution. For example, consider the precision *tau* (1 / variance) of a normal distribution. The two priors

   ```
   tau ~ dgamma(0.001, 0.001) and
   log.tau ~ dunif(-10, 10); log(tau) <- log.tau
   ```

   are essentially identical but will give slightly different results for Dhat: for the first prior the stochastic parent is *tau* and hence the posterior mean of *tau* is substituted in Dhat, while in the second parameterisation the stochastic parent is *log.tau* and hence the posterior mean of log(*tau*) is substituted in Dhat.

3) For sampling distributions that are log-concave in their stochastic parents, pD is guaranteed to be positive (provided the simulation has converged). However, it is theoretically possible to get negative values. We have obtained negative pD's in the following situations:

   i) with non-log-concave likelihoods (e.g. Student-t distributions) when there is substantial conflict between prior and data;
   ii) when the posterior distribution for a parameter is symmetric and bimodal, and so the posterior mean is a very poor summary statistic and gives a very large deviance.

4) No MC error is available on the DIC. MC error on Dbar can be obtained by monitoring deviance and is generally quite small. The primary concern is to ensure convergence of Dbar – it is therefore worthwhile checking the stability of Dbar over a long chain.

5) The minimum DIC estimates the model that will make the best short-term predictions, in the same spirit as Akaike's criterion. However, if the difference in DIC is, say, less than 5, and the models make very different inferences, then it could be misleading just to report the model with the lowest DIC.

6) DICs are comparable only over models with exactly the same observed data, but there is no need for

them to be nested.

7) DIC differs from Bayes factors and BIC in both form and aims.

8) Caution is advisable in the use of DIC until more experience has been gained. **It is important to note that the calculation of DIC will be disallowed for certain models. Please see the WinBUGS 1.4 web-page for details:**

http://www.mrc-bsu.cam.ac.uk/bugs/winbugs/contents.shtml

Mixtures of models of different complexity

[top | home]

Suppose we assume that each observation, or group of observations, is from one of a set of distributions, where the members of the set have different complexity. For example, we may think data for each person's growth curve comes from either a linear or quadratic line. We might think we would require 'reversible jump' techniques, but this is not the case as we are really only considering a single mixture model as a sampling distribution. Thus standard methods for setting up mixture distributions can be adopted, but with components having different numbers of parameters.

The mixtures example illustrates how this is handled in WinBUGS, using a set of simulated data.

Naturally, the standard warnings about mixture distributions apply, in that convergence may be poor and careful parameterisation may be necessary to avoid some of the components becoming empty.

Where the size of a set is a random quantity

[top | home]

Suppose the size of a set is a random quantity: this naturally occurs in 'changepoint' problems where observations up to an unknown changepoint K come from one model, and after K come from another. Note that we **cannot** use the construction

```
for (i in 1:K) {
    y[i] ~ model 1
}
for (i in (K + 1):N) {
    y[i] ~ model 2
}
```

since the index for a loop cannot be a random quantity. Instead we can use the step function to set up an indicator as to which set each observation belongs to:

```
for (i in 1:N) {
    ind[i] <- 1 + step(i - K - 0.01)      # will be 1 for all i <= K, 2 otherwise
    y[i] ~ model ind[i]
}
```

This is illustrated in random-sets by the problem of adding up terms in a series of unknown length, and in Stagnant by a changepoint problem.

Assessing sensitivity to prior assumptions

[top | home]

One way to do this is to repeat the analysis under different prior assumptions, but within the same simulation in order to aid direct comparison of results. Assuming the consequences of K prior distributions are to be compared:

a) replicate the dataset K times within the model code;
b) set up a loop to repeat the analysis for each prior, holding results in arrays;
c) compare results using the 'compare' facility.

The example prior-sensitivity explores six different suggestions for priors on the random-effects variance in a meta-analysis.

Modelling unknown denominators [top | home]

Suppose we have an unknown Binomial denominator for which we wish to express a prior distribution. It can be given a Poisson prior but this makes it difficult to express a reasonably uniform distribution. Alternatively a continuous distribution could be specified and then the 'round' function used. For example, suppose we are told that a fair coin has come up heads 10 times – how many times has it been tossed?

```
model {
    r <- 10
    p <- 0.5
    r ~ dbin(p, n)
    n.cont ~ dunif(1, 100)
    n <- round(n.cont)
}
```

node	mean	sd	MC error	2.5%	median	97.5%	start	sample
n	21.08	4.794	0.07906	13.0	21.0	32.0	1001	5000
n.cont	21.08	4.804	0.07932	13.31	20.6	32.0	1001	5000

Assuming a uniform prior for the number of tosses, we can be 95% sure that the coin has been tossed between 13 and 32 times. A discrete prior on the integers could also have been used in this context.

Handling unbalanced datasets [top | home]

Suppose we observe the following data on three individuals:

Person 1: 13.2
Person 2: 12.3 , 14.1
Person 3: 11.0, 9.7, 10.3, 9.6

There are three different ways of entering such 'ragged' data into WinBUGS:

1. Fill-to-rectangular: Here the data is 'padded out' by explicitly including the missing data, i.e.

```
y[,1]   y[,2]   y[,3]   y[,4]
 13.2     NA      NA      NA
 12.3    14.1     NA      NA
 11.0     9.7    10.3     9.6
END
```

or list(y = structure(.Data = c(13.2, NA, NA, NA, 12.3, 14.1, NA, NA, 11.0, 9.7, 10.3, 9.6), .Dim = c(3, 4)).

A model such as `y[i, j] ~ dnorm(mu[i], 1)` can then be fitted. This approach is inefficient unless one explicitly wishes to estimate the missing data.

2. Nested indexing: Here the data are stored in a single array and the associated person is recorded as a factor, i.e.

```
 y[]   person[]
13.2       1
12.3       2
14.1       2
11.0       3
 9.7       3
10.3       3
 9.6       3
 END
```

or list(y = c(13.2, 12.3, 14.1, 11.0, 9.7, 10.3, 9.6), person = c(1, 2, 2, 3, 3, 3, 3)).

A model such as `y[k] ~ dnorm(mu[person[k]], 1)` can then be fitted. This seems an efficient and

clear way to handle the problem.

3. Offset: Here an 'offset' array holds the position in the data array at which each person's data starts. For example, the data might be

list(y = c(13.2, 12.3, 14.1, 11.0, 9.7, 10.3, 9.6), offset = c(1, 2, 4, 8))

and lead to a model containing the code

```
for (k in offset[i]:(offset[i + 1] - 1)) {
    y[k] ~ dnorm(mu[i], 1)
}
```

The danger with this method is that it relies on getting the offsets correct and they are difficult to check. See the ragged example for a full worked example of these methods.

Learning about the parameters of a Dirichlet distribution

[top | home]

Suppose as part of a model there are J probability arrays p[j, 1:K], j = 1, ..., J, where K is the dimension of each array and sum(p[j, 1:K]) = 1 for all j. We give each of them a Dirichlet prior:

```
p[j, 1:K] ~ ddirch(alpha[])
```

and we would like to learn about alpha[]. However, the parameters alpha[] of a Dirichlet distribution cannot be stochastic nodes. The trick is to note that if delta[k] ~ dgamma(alpha[k], 1), then the vector with elements delta[k] / sum(delta[1:K]), k = 1, ..., K, is Dirichlet with parameters alpha[k], k = 1, ..., K. So the following construction should allow learning about the parameters alpha[]:

```
for (k in 1:K) {
    p[j, k] <- delta[j, k] / sum(delta[j,])
    delta[j, k] ~ dgamma(alpha[k], 1)
}
```

A prior can be put directly on the alpha[k]'s.

Use of the "cut" function

[top | home]

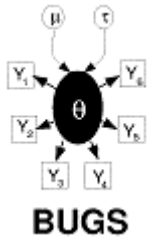
Suppose we observe some data that we do not wish to contribute to the parameter estimation and yet we wish to consider as part of the model. This might happen, for example:

a) when we wish to make predictions on some individuals on whom we have observed some partial data that we do not wish to use for parameter estimation;
b) when we want to use data to learn about some parameters and not others;
c) when we want evidence from one part of a model to form a prior distribution for a second part of the model, but we do not want 'feedback' from this second part.

The "cut" function forms a kind of 'valve' in the graph: prior information is allowed to flow 'downwards' through the cut, but likelihood information is prevented from flowing upwards.

For example, the following code leaves the distribution for theta unchanged by the observation y.

```
model {
    y <- 2
    y ~ dnorm(theta.cut, 1)
    theta.cut <- cut(theta)
    theta ~ dnorm(0, 1)
}
```



WinBUGS Graphics

Contents

General properties
Margins
Axis Bounds
Titles
All Plots
Fonts
Specific properties (via *Special...*)
Density plot
Box plot
Caterpillar plot
Model fit plot
Scatterplot

General properties

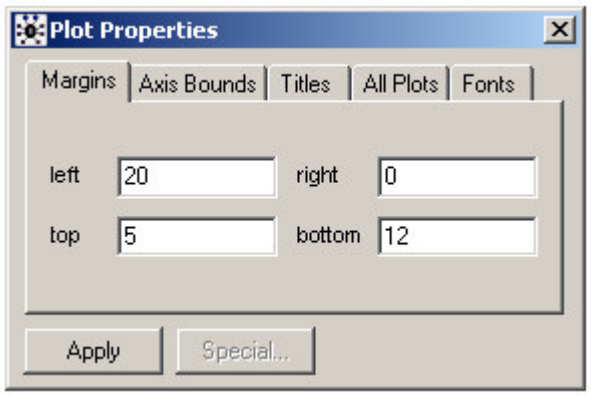
[top | home]

All WinBUGS graphics have a set of basic properties that can be modified using the *Plot Properties* dialog box. This is opened as follows: first, focus the relevant plot by left-clicking on it; then select *Object Properties...* from the *Edit* menu. Alternatively, right-clicking on a focused plot will reveal a pop-up menu from which *Properties...* may equivalently be selected. The *Plot Properties* dialogue box comprises a "tab-view" and two command buttons, namely *Apply* and *Special...*. The tab-view contains five tabs that allow the user to make different types of modification – these are discussed below. The *Apply* command button applies the properties displayed in the currently selected tab to the focused plot. And *Special...* opens any special property editors that have been designed so that the user may further interact with specific types of plot – these are discussed below.

Margins

[top | home]

The *Margins* tab displays the plot's left, right, top and bottom margins in millimetres (mm). The left and bottom margins are used for drawing the y- and x-axes respectively. The top margin provides room for the plot's title and the right margin is typically used for plots that require a legend. (Note that top margins (and hence titles) are always inside the plotting rectangle, i.e. there is no gap between the plotting rectangle and the top edge of the graphic.)

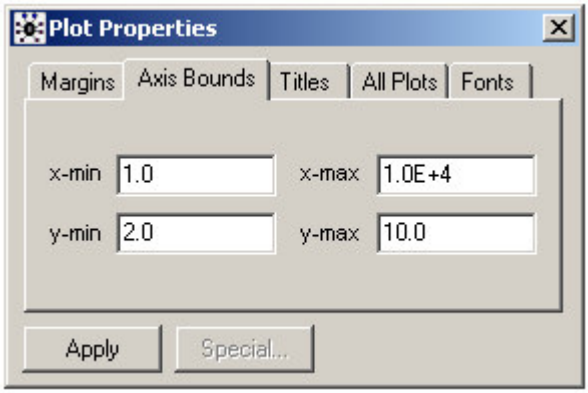


In cases where it is not inappropriate to alter a plot's margins the user may enter his/her preferred values and click on *Apply* to effect the desired change. If the specified values are not appropriate, e.g. if left + right is greater than the width of the graphic (which would result in a plotting rectangle of negative width) or if any

margin is negative, etc., then either nothing will happen and the *Margins* tab will reset itself or some form of compromise will be made.

Axis Bounds [top | home]

The user may specify new minimum and maximum values for either or both axes using the *Axis Bounds* tab (followed by the *Apply* button).

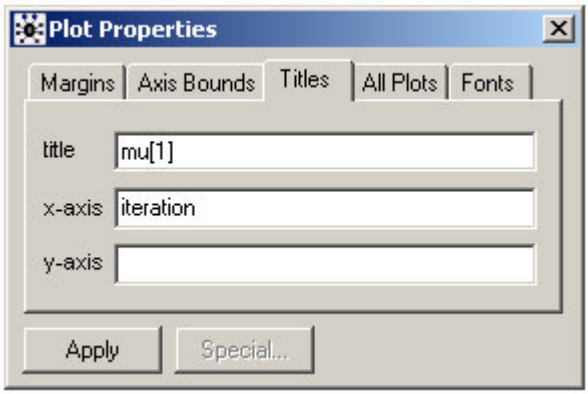


Note that the resulting minima and maxima may not be exactly the same as the values specified because WinBUGS always tries to ensure that axes range between 'nice' numbers and also have a sufficient number of tick marks. Note also that if max < min is specified then WinBUGS will ignore it and the *Axis Bounds* tab will reset itself, but there is no guard against specifying a range that does not coincide with the data being plotted, and so the contents of the plot may disappear!

(Some types of plot, such as trace plots, do not allow the user to change their axis bounds, because it would be inappropriate to do so.)

Titles [top | home]

The *Titles* tab should be self-explanatory.



Note, however, that because WinBUGS does not (yet) support vertical text, a substantial amount of space may be required in the left margin in order to write the y-axis title horizontally – if sufficient space is not available then the y-axis title may not appear at all (clearly, this can be rectified by increasing the left margin).

All Plots [top | home]

The idea behind the *All Plots* tab is to allow the user to apply some or all of the properties of the focused plot to all plots of the same type (as the focused plot) in the same window (as the focused plot).

The user should first configure the focused plot in the desired way and then decide which of the plot's various properties are to be applied to all plots (of the same type, in the same window). The user should then check all relevant check-boxes and click on *Apply all*. Be careful! It is easy to make mistakes here and there is no undo option – the best advice for users who go wrong is to reproduce the relevant graphics window via the *Sample Monitor Tool* or the *Comparison Tool*, for example.

Fonts [top | home]

The *Fonts* tab allows the user to change the font of the plot's title, its axes, and any other text that may be present.

First select the font to be modified and click on *Modify...*. (Note that it will only be possible to select *other* if the focused plot has a third font, i.e. if text other than the title and axes is present on the plot, e.g. the labels on a box plot – see Compare...) The self-explanatory *Font* dialogue box should appear – select the required font and click on *OK* (or *Cancel*).

In order to apply the same font to an arbitrarily large group of plots (not necessarily of the same type), rather than using the *All Plots* tab we use a "drag-and-pick" facility. First focus a single plot and select which font is to be modified for the whole group: *title*, *axes*, or *other* (if available). Now highlight the group of plots using the mouse. Hold down the ALT key and then the left-hand mouse button and drag the mouse over an area of text with the desired font; then release the mouse button and the ALT key in turn and the required changes should be made. As an alternative to dragging the mouse over a piece of text with the desired font, the user may instead drag over another plot (even one in the group to be modified) – the group will adopt that plot's properties for the selected font on the *Fonts* tab.

Specific properties (via *Special...*) [top | home]

Below we describe the special property editors that are available for certain types of plot – these allow user-interaction beyond that afforded by the *Plot Properties* dialogue box and are accessed via its *Special...* button.

Density plot [top | home]

When the *density* button on the *Sample Monitor Tool* is pressed, the output depends on whether the specified variable is discrete or continuous – if the variable is discrete then a histogram is produced whereas if it is continuous a kernel density estimate is produced instead. The specialized property editor that appears when

Special... on the *Plot Properties* dialogue box is selected also differs slightly depending on the nature of the specified variable. In both cases the editor comprises a numeric field and two command buttons (*apply* and *apply all*) but in the case of a histogram the numeric field corresponds to the "bin-size" whereas for kernel density estimates it is the smoothing parameter* (see below):

property editor for histogram

property editor for kernel density estimate

In either case, the *apply* button sets the bin-size or smoothing parameter of the focused plot to the value currently displayed in the numeric field. The *apply all* button, on the other hand, applies the value currently displayed in the numeric field to *all* plots of the same type (as the focused plot) in the same window (as the focused plot).

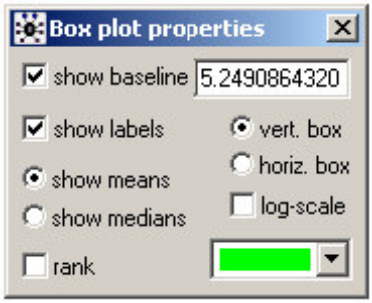
Note*: We define the smoothing parameter for kernel density estimates, σ, via the definition of band-width. Suppose our posterior sample comprises m realisations of variable z – denote these by z_i, i = 1,..., m:

$$\text{band-width} = V^{1/2} / m^{\sigma}; \qquad V = m^{-1}\sum_i z_i^2 - \left(m^{-1}\sum_i z_i\right)^2$$

where the summations are from i = 1 to i = m. The default setting for σ is 0.2.

Box plot [top | home]

The box plot property editor, opened by pressing *Special...* on the *Plot Properties* dialogue box when a box plot is focused, is described as follows:



show baseline: this check-box should be used to specify whether or not a baseline should be shown on the plot; the numeric field to the immediate right of the check-box gives the value of that baseline. The default setting is that a baseline equal to the global mean of the posterior means will be shown – the user may specify an alternative baseline simply by editing the displayed value.

show labels: check-box that determines whether or not each distribution/box should be labelled with its index in *node* (that is *node* on the *Comparison Tool*). The default setting is that labels should be shown.

show means or ***show medians:*** these radio buttons specify whether the solid black line at the approximate centre of each box is to represent the posterior mean or the posterior median – mean is the default.

rank: use this check-box to specify whether the distributions should be ranked and plotted in order. The basis for ranking is either the posterior mean or the posterior median, depending on which is chosen to be displayed in the plot (via *show means* or *show medians*).

vert. box or ***horiz. box:*** these radio buttons determine the orientation of the plot. The default is "vertical boxes", which means that the scale axis (i.e. that which measures the 'width' of the distributions) is the y-axis.

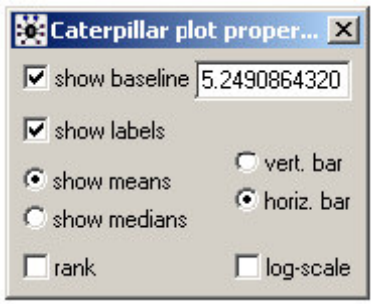
log-scale: the scale axis can be given a logarithmic scale by checking this check-box.

Finally, in the bottom right-hand corner there is a colour field for selecting the fill-colour of the displayed boxes.

Caterpillar plot

[top | home]

The caterpillar plot property editor is virtually identical with the box plot property editor except that there is no fill-colour field on the former:



Model fit plot

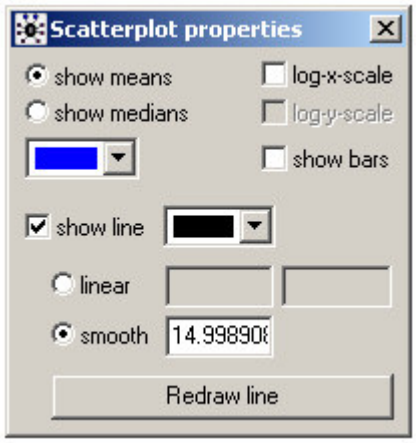
[top | home]

In cases where an axis is defined on a strictly positive range, it may be given a logarithmic scale by checking the appropriate check-box:

Scatterplot

[top | home]

With a scatterplot focused, select *Special...* on the *Plot Properties* dialogue box to obtain the following property editor:



show means or ***show medians:*** these radio buttons determine whether it is the posterior means or medians of *node* that are plotted/scattered against *axis* to form the plot (that is *node* and *axis* on the *Comparison Tool*) – the default is means.

Immediately beneath the ***show means*** or ***show medians*** radio buttons is a colour field for selecting the colour of the scattered points.

log-x-/log-y-scale: either or both axes may be given a logarithmic scale (assuming they are defined on strictly positive ranges) by checking the appropriate check-box(es).

show bars: 95 per cent posterior intervals (2.5% – 97.5%) for *node* will be shown on the plot (as vertical bars) if this box is checked.

show line: use this check-box to specify whether or not a reference line (either a straight line specified via its intercept and gradient or an exponentially weighted smoother – see below) is to be displayed.

To the immediate right of the ***show line*** check-box is a colour field that determines the reference line's colour (if displayed).

linear or ***smooth:*** these radio buttons are used to specify whether the reference line (if it is to be displayed) should be linear or whether an exponentially weighted smoother should be fitted instead. In the case where a linear line is required, its intercept and gradient should be entered in the two numeric fields to the right of the *linear* radio button (in that order). If a smoother is required instead then the desired degree of smoothing* (see below) should be entered in the numeric field to the right of the *smooth* radio button. To save unnecessary redrawing of the plot as the various numeric parameters are changed, the *Redraw line* command button is used to inform WinBUGS of when all alterations to the parameters have been completed.

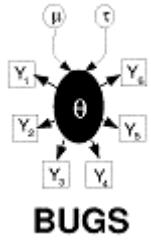
Note*: The exponentially weighted smoother that WinBUGS uses on scatterplots is defined as:

$$s_i = \sum_j w_{ij} y_j \Big/ \sum_j w_{ij} \qquad i = 1,\ldots, n$$

where n is the number of scattered points and the summations are from j = 1 to j = n. The weights w_{ij} are given by

$$w_{ij} = \exp(-|x_i - x_j| / \sigma)$$

where σ is the smoothing parameter defined in the *Scatterplot properties* dialogue box (next to the *smooth* radio button). The default setting for σ is (somewhat arbitrarily) {max(x) – min(x)} / 20.



Tips and Troubleshooting

Contents

Restrictions when modelling
Some error messages
Some Trap messages
The program hangs
Speeding up sampling
Improving convergence

This section covers a range of problems that might arise:

1) restrictions in setting up the model;
2) error messages generated when setting up the model or when sampling;
3) Traps: these corresponds to an error which has not been picked up by *WinBUGS*. It is difficult to interpret the Trap message!
4) the program just hanging;
5) slow sampling;
6) lack of convergence.

Restrictions when modelling

[top | home]

Restrictions have been stated throughout this manual. A summary list is as follows:

a) Each stochastic and logical node must appear once and only once on the left-hand-side of an

expression. The only exception is when carrying out a data transformation (see Data transformations). This means, for example, that it is generally not possible to give a distribution to a quantity that is specified as a logical function of an unknown parameter.

b) Truncated sampling distributions cannot be handled using the `I(.,.)` construct – see Censoring and truncation.

c) Multivariate distributions: Dirichlet and Wishart distributions may only be used as conjugate priors and must have known parameters (although there does exist a trick for allowing unknown Dirichlet parameters); in contrast, multinomial distributions may not be used as priors and the order *N* of the distribution must be known; multivariate normal and Student-t distributions can be used anywhere in the graph (i.e. as prior or likelihood) and there are no restrictions regarding their parameters. See Constraints on using certain distributions.

d) Logical nodes cannot be given data or initial values. This means, for example, that it is not possible to model observed data that is the sum of two random variables. (See Logical nodes.)

e) Poisson priors can only be specified for the (unknown) order (n) of a *single* binomial observation. In this case, the sampling method for a 'shifted Poisson' distribution will be used (see MCMC methods). If more than one binomial response is assumed to share the same order parameter, then the algorithm for a shifted Poisson cannot be used to update the order parameter and WinBUGS will return the error message "Unable to choose update method for n". A suitably constrained continuous prior may be used for the order parameter, provided that it is used in conjunction with the `round(.)` function to ensure integer values (see here for an example of using the 'round' function to this end).

Some error messages

[top | home]

a) **'expected variable name'** indicates an inappropriate variable name.

b) **'linear predictor in probit regression too large'** indicates numerical overflow. See possible solutions below for Trap 'undefined real result'.

c) **'logical expression too complex'** - a logical node is defined in terms of too many parameters/constants or too many operators: try introducing further logical nodes to represent parts of the overall calculation; for example, a1 + a2 + a3 + b1 + b2 + b3 could be written as A + B where A and B are the simpler logical expressions a1 + a2 + a3 and b1 + b2 + b3, respectively. Note that linear predictors with many terms should be formulated by 'vectorizing' parameters and covariates and by then using the `inprod(.,.)` function (see Logical nodes).

d) **'invalid or unexpected token scanned' -** check that the *value* field of a logical node in a Doodle has been completed.

e) **'unable to choose update method'** indicates that a restriction in the program has been violated – see Restrictions when modelling above.

f) **'undefined variable'** - variables in a data file must be defined in a model (just put them in as constants or with vague priors). If a logical node is reported *undefined*, the problem may be with a node on the 'right hand side'.

g) **'index out of range'** - usually indicates that a loop-index goes beyond the size of a vector (or matrix dimension); sometimes, however, appears if the # has been omitted from the beginning of a comment line.

Some Trap messages

[top | home]

a) **'undefined real result'** indicates numerical overflow. Possible reasons include:

- initial values generated from a 'vague' prior distribution may be numerically extreme - specify appropriate initial values;
- numerically impossible values such as log of a non-positive number - check, for example, that no zero expectations have been given when Poisson modelling;
- numerical difficulties in sampling. Possible solutions include:
    - better initial values;
    - more informative priors - uniform priors might still be used but with their range restricted to plausible values;
    - better parameterisation to improve orthogonality;
    - standardisation of covariates to have mean 0 and standard deviation 1.
- can happen if all initial values are equal.

Probit models are particularly susceptible to this problem, i.e. generating undefined real results. If a probit is a stochastic node, it may help to put reasonable bounds on its distribution, e.g.

```
probit(p[i]) <- delta[i]
delta[i] ~ dnorm(mu[i], tau)I(-5, 5)
```

This trap can sometimes be escaped from by simply clicking on the update button. The equivalent construction

```
p[i] <- phi(delta[i])
```

may be more forgiving.

b) **'index array out of range' -** possible reasons include:
- attempting to assign values beyond the declared length of an array;
- if a logical expression is too long to evaluate - break it down into smaller components.

c) **'stack overflow'** can occur if there is a recursive definition of a logical node.

d) **'NIL dereference (read)'** can occur at compilation in some circumstances when an inappropriate transformation is made, for example an array into a scalar.

e) Trap messages referring to **'DFreeARS'** indicate numerical problems with the derivative-free adaptive rejection algorithm used for log-concave distributions. One possibility is to change to "Slice" sampling – see here for details.

The program hangs

[top | home]

This could be due to:

a) a problem that seems to happen with NT - rebooting is a crude way out;
b) interference with other programs running - try to run WinBUGS on its own;
c) a particularly ill-posed model - try the approaches listed above under Trap messages.

Speeding up sampling

[top | home]

The key is to reduce function evaluations by expressing the model in as concise a form as possible. For example, take advantage of the functions provided and use nested indexing wherever possible. Look at the packaged examples and others provided on users' web sites.

Improving convergence

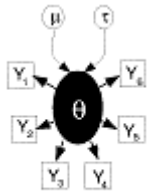
[top | home]

Possible solutions include:

a) better parameterisation to improve orthogonality of joint posterior;
b) standardisation of covariates to have mean 0 and standard deviation 1;
c) use of ordered over-relaxation.

For other problems, try the FAQ pages of the web site and the Classic BUGS manual. Then try mailing us on bugs@mrc-bsu.cam.ac.uk.

Please try and keep the discussion list for modelling issues and problems other than *my program won't run!*



Tutorial

Contents

Introduction
Specifying a model in the BUGS language
Running a model in WinBUGS
Monitoring parameter values
Checking convergence
How many iterations after convergence?
Obtaining summaries of the posterior distribution

Introduction

[top | home]

This tutorial is designed to provide new users with a step-by-step guide to running an analysis in WinBUGS. It is not intended to be prescriptive, but rather to introduce you to the main tools needed to run an MCMC simulation in WinBUGS, and give some guidance on appropriate usage of the software.

The Seeds example from Volume I of the WinBUGS examples will be used throughout this tutorial. This example is taken from Table 3 of Crowder (1978) and concerns the proportion of seeds that germinated on each of 21 plates arranged according to a 2 by 2 factorial layout by seed and type of root extract. The data are shown below, where r_i and n_i are the number of germinated and the total number of seeds on the i^{th} plate, i = 1, ..., N. These data are also analysed by, for example, Breslow and Clayton (1993).

seed O. aegyptiaco 75						*seed O. aegyptiaco 73*					
Bean			Cucumber			Bean			Cucumber		
r	n	r/n	r	n	r/n	r	n	r/n	r	n	r/n
10	39	0.26	5	6	0.83	8	16	0.50	3	12	0.25
23	62	0.37	53	74	0.72	10	30	0.33	22	41	0.54
23	81	0.28	55	72	0.76	8	28	0.29	15	30	0.50
26	51	0.51	32	51	0.63	23	45	0.51	32	51	0.63
17	39	0.44	46	79	0.58	0	4	0.00	3	7	0.43
			10	13	0.77						

The model is essentially a random effects logistic regression, allowing for over-dispersion. If p_i is the probability of germination on the i^{th} plate, we assume

$$r_i \sim \text{Binomial}(p_i, n_i)$$

$$\text{logit}(p_i) = \alpha_0 + \alpha_1 x_{1i} + \alpha_2 x_{2i} + \alpha_{12} x_{1i} x_{2i} + b_i$$

$$b_i \sim \text{Normal}(0, \tau)$$

where x_{1i} and x_{2i} are the seed type and root extract of the i^{th} plate, and an interaction term $\alpha_{12} x_{1i} x_{2i}$ is included.

Specifying a model in the BUGS language [top | home]

The BUGS language allows a concise expression of the model, using the 'twiddles' symbol ~ to denote stochastic (probabilistic) relationships, and the left arrow ('<' sign followed by '–' sign) to denote deterministic (logical) relationships. The stochastic parameters α_0, α_1, α_2, α_{12}, and τ are given proper but minimally informative prior distributions, while the logical expression for `sigma` allows the standard deviation (of the random effects distribution) to be estimated.

```
model {
    for (i in 1:N) {
        r[i] ~ dbin(p[i], n[i])
        b[i] ~ dnorm(0, tau)
        logit(p[i]) <- alpha0 + alpha1 * x1[i] + alpha2 * x2[i]
            + alpha12 * x1[i] * x2[i] + b[i]
    }
    alpha0 ~ dnorm(0, 1.0E-6)
    alpha1 ~ dnorm(0, 1.0E-6)
    alpha2 ~ dnorm(0, 1.0E-6)
    alpha12 ~ dnorm(0, 1.0E-6)
    tau ~ dgamma(0.001, 0.001)
    sigma <- 1 / sqrt(tau)
}
```

More detailed descriptions of the BUGS language along with lists of the available logical functions and stochastic distributions can be found in Model Specification and Distributions. See also the on-line examples: Volume 1 and Volume 2. An alternative way of specifying a model in WinBUGS is to use the graphical interface known as DoodleBUGS.

Running a model in WinBUGS [top | home]

The WinBUGS software uses compound documents, which comprise various different types of information (formatted text, tables, formulae, plots, graphs, etc.) displayed in a single window and stored in a single file. This means that it is possible to run the model for the "seeds" example directly from this tutorial document, since the model code can be made 'live' just by highlighting it. However, it is more usual when creating your own models to have the model code and data etc. in separate files. We have therefore created a separate file with the model code in it for this tutorial – you will find the file in Manuals/Tutorial/seeds_model.odc (or click here).

Step 1

Open the seeds_model file as follows:

* Point to File on the tool bar and click once with the left mouse button (LMB).
* Highlight the Open... option and click once with LMB.
* Select the appropriate directory and double-click on the file to open.

Step 2

To run the model, we first need to check that the model description does fully define a probability model:

* Point to Model on the tool bar and click once with LMB.
* Highlight the Specification... option and click once with LMB.
* Focus the window containing the model code by clicking the LMB once anywhere in the window – the top panel of the window should then become highlighted in blue (usually) to indicate that the window is currently in focus.
* Highlight the word `model` at the beginning of the code by dragging the mouse over the word whilst holding down the LMB.
* Check the model syntax by clicking once with LMB on the check model button in the Specification Tool window. A message saying "model is syntactically correct" should appear in the bottom left of the WinBUGS program window.

Step 3
We next need to load in the data. The data can be represented using S-Plus object notation (file Manuals/Tutorial/seeds_S_data.odc), or as a combination of an S-Plus object and a rectangular array with labels at the head of each column (file Manuals/Tutorial/seeds_mix_data.odc).

* Open one of the data files now.
* To load the data in file Manuals/Tutorial/seeds_S_data.odc:
  - Highlight the word `list` at the beginning of the data file.
  - Click once with the LMB on the load data button in the Specification Tool window. A message saying "data loaded" should appear in the bottom left of the WinBUGS program window.
* To load the data in file Manuals/Tutorial/seeds_mix_data.odc:
  - Highlight the word `list` at the beginning of the data file.
  - Click once with the LMB on the load data button in the Specification Tool window. A message saying "data loaded" should appear in the bottom left of the WinBUGS program window.
  - Next highlight the whole of the header line (i.e. column labels) of the rectangular array data.
  - Click once with the LMB on the load data button in the Specification Tool window. A message saying "data loaded" should appear in the bottom left of the WinBUGS program window.

Step 4
Now we need to select the number of chains (i.e. sets of samples to simulate). The default is 1, but we will use 2 chains for this tutorial, since running multiple chains is one way to check the convergence of your MCMC simulations.

* Type the number 2 in the white box labelled num of chains in the Specification Tool window. In practice, if you have a fairly complex model, you may wish to do a pilot run using a single chain to check that the model compiles and runs and obtain an estimate of the time taken per iteration. Once you are happy with the model, re-run it using multiple chains (say 2–5 chains) to obtain a final set of posterior estimates.

Step 5
Next compile the model by clicking once with the LMB on the compile button in the Specification Tool window. A message saying "model compiled" should appear in the bottom left of the WinBUGS program window. This sets up the internal data structures and chooses the specific MCMC updating algorithms to be used by WinBUGS for your particular model.

Step 6
Finally the MCMC sampler must be given some initial values for each stochastic node. These can be arbitrary values, although in practice, convergence can be poor if wildly inappropriate values are chosen. You will need a different set of initial values for each chain, i.e. two sets are needed for this tutorial since we have specified two chains – these are stored in file Manuals/Tutorial/seeds_inits.odc.

* Open this file now.
* To load the initial values:
  - Highlight the word `list` at the beginning of the first set of initial values.
  - Click once with the LMB on the load inits button in the Specification Tool window. A message saying "initial values loaded: model contains uninitialized nodes (try running gen inits or loading more files)" should appear in the bottom left of the WinBUGS program window.
  - Repeat this process for the second initial values file. A message saying "initial values loaded: model initialized" should now appear in the bottom left of the WinBUGS program window.

Note that you do not need to provide a list of initial values for every parameter in your model. You can get WinBUGS to generate initial values for any stochastic parameter not already initialized by clicking with the LMB on the gen inits button in the Specification Tool window. WinBUGS generates initial values by forward sampling from the prior distribution for each parameter. Therefore, you are advised to **provide your own initial values for parameters with vague prior distributions** to avoid wildly inappropriate values.

Step 7

Close the Specification Tool window. You are now ready to start running the simulation. However, before doing so, you will probably want to set some monitors to store the sampled values for selected parameters. For the seeds example, set monitors for the parameters `alpha0`, `alpha1`, `alpha2`, `alpha12` and `sigma` – see here for details on how to do this.

To run the simulation:

* Select the Update... option from the Model menu.
* Type the number of updates (iterations of the simulation) you require in the appropriate white box (labelled updates) – the default value is 1000.
* Click once on the update button: the program will now start simulating values for each parameter in the model. This may take a few seconds – the box marked iteration will tell you how many updates have currently been completed. The number of times this value is revised depends on the value you have set for the refresh option in the white box above the iteration box. The default is every 100 iterations, but you can ask the program to report more frequently by changing refresh to, say, 10 or 1. A sensible choice will depend on how quickly the program runs. For the seeds example, experiment with changing the refresh option from 100 to 10 and then 1.
* When the updates are finished, the message "updates took *** s" will appear in the bottom left of the WinBUGS program window (where *** is the number of seconds taken to complete the simulation).
* If you previously set monitors for any parameters you can now check convergence and view graphical and numerical summaries of the samples. Do this now for the parameters you monitored in the seeds example – see Checking convergence for tips on how to do this (for the seeds example, you should find that at least 2000 iterations are required for convergence).
* Once you're happy that your simulation has converged, you will need to run some further updates to obtain a sample from the posterior distribution. The section How many iterations after convergence? provides tips on deciding how many more updates you should run. For the seeds example, try running a further 10000 updates.
* Once you have run enough updates to obtain an appropriate sample from the posterior distribution, you may summarise these samples numerically and graphically (see section Obtaining summaries of the posterior distribution for details on how to do this). For the seeds example, summary statistics for the monitored parameters are shown below:

node	mean	sd	MC error	2.5%	median	97.5%	start	sample
alpha0	-0.5553	0.1904	0.003374	-0.9365	-0.5563	-0.1763	2001	20000
alpha1	0.08693	0.3123	0.006873	-0.5502	0.09157	0.6964	2001	20000
alpha12	-0.8358	0.4388	0.01105	-1.731	-0.8253	-0.002591	2001	20000
alpha2	1.359	0.2744	0.005812	0.8204	1.354	1.923	2001	20000
sigma	0.2855	0.146	0.005461	0.04489	0.2752	0.6132	2001	20000

* To save any files created during your WinBUGS run, focus the window containing the information you want to save, and select the Save As... option from the File menu.
* To quit WinBUGS, select the Exit option from the File menu.

Monitoring parameter values

[top | home]

In order to check convergence and obtain posterior summaries of the model parameters, you first need to set **monitors** for each parameter of interest. This tells WinBUGS to store the values sampled for those parameters; otherwise, WinBUGS automatically discards the simulated values.

There are two types of monitor in WinBUGS:

Samples monitors: Setting a samples monitor tells WinBUGS to store *every* value it simulates for that parameter. You will need to set a samples monitor if you want to view trace plots of the samples to check convergence (see Checking convergence) or if you want to obtain 'exact' posterior quantiles, for example, the posterior 95% Bayesian credible interval for that parameter. (Note that you can also obtain *approximate* 2.5%, 50% and 97.5% quantiles of the posterior distribution for each parameter using summary monitors.)

To set a samples monitor:

* Select Samples... from the Inference menu;
* Type the name of the parameter to be monitored in the white box marked node;
* Click once with the LMB on the button marked set;
* Repeat for each parameter to be monitored.

Summary monitors: Setting a summary monitor tells WinBUGS to store the running mean and standard deviation for the parameter, plus approximate running quantiles (2.5%, 50% and 97.5%). The values saved contain less information than saving each individual sample in the simulation, but require much less storage. This is an important consideration when running long simulations (e.g. 1000's of iterations) and storing values for many variables.

We recommend setting summary monitors on long vectors of parameters such as random effects in order to store posterior summaries, and then also setting full samples monitors on a small subset of the random effects, plus other relevant parameters (e.g. means and variances), to check convergence.

To set a summary monitor:

* Select Summary... from the Inference menu;
* Type the name of the parameter to be monitored in the white box marked node;
* Click once with the LMB on the button marked set;
* Repeat for each parameter to be monitored.

Note: you should not set a summary monitor until you are happy that convergence has been reached (see Checking convergence), since it is not possible to discard any of the pre-convergence ('burn-in') values from the summary once it is set, other than to clear the monitor and re-set it.

Checking convergence

[top | home]

Checking convergence requires considerable care. **It is very difficult to say conclusively that a chain (simulation) has converged, only to diagnose when it definitely hasn't!**

The following are practical guidelines for assessing convergence:

* For models with many parameters, it is impractical to check convergence for every parameter, so just choose a random selection of relevant parameters to monitor. For example, rather than checking convergence for every element of a vector of random effects, just choose a random subset (say, the first 5 or 10).
* Examine trace plots of the sample values versus iteration to look for evidence of when the simulation appears to have stabilised:

  To obtain 'live' trace plots for a parameter:

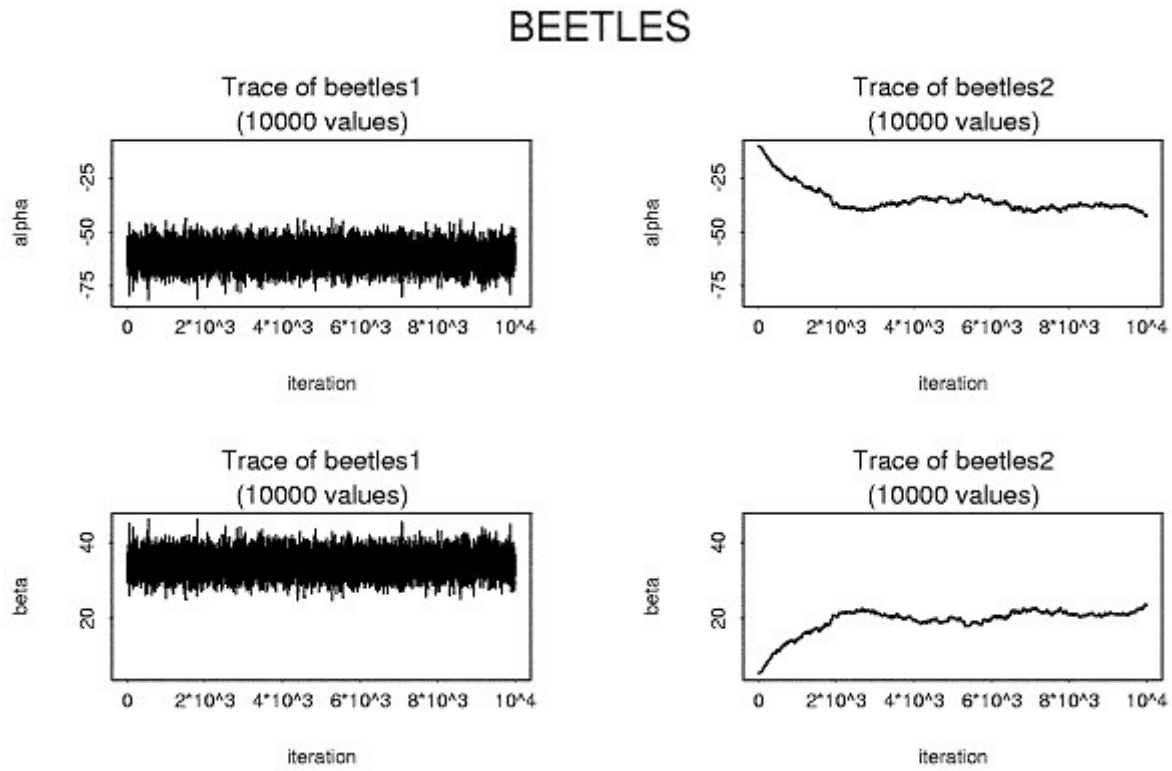
  - Select Samples... from the Inference menu.
  - Type the name of the parameter in the white box marked node.
  - Click once with the LMB on the button marked trace: an empty graphics window will appear on screen.
  - Repeat for each parameter of interest.
  - Once you start running the simulations (using the Update... tool from the Model menu), trace plots for these parameters will appear 'live' in the graphics windows.

  To obtain a trace plot showing the full history of the samples *for any parameter for which you have previously set a samples monitor and carried out some updates*:

  - Select Samples... from the Inference menu.
  - Type the name of the parameter in the white box marked node (or select the name from the pull down list of currently monitored nodes – click once with the LMB on the downward-facing arrowhead to the immediate right of the node field).

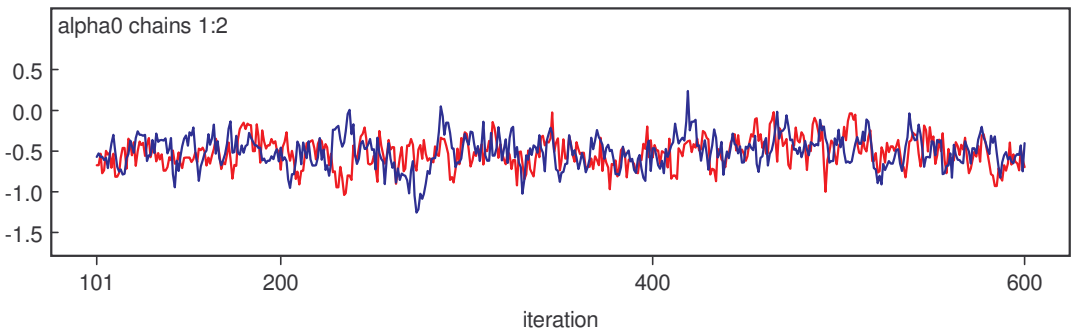
- Click once with the LMB on the button marked history: a graphics window showing the sample trace will appear.
- Repeat for each parameter of interest.

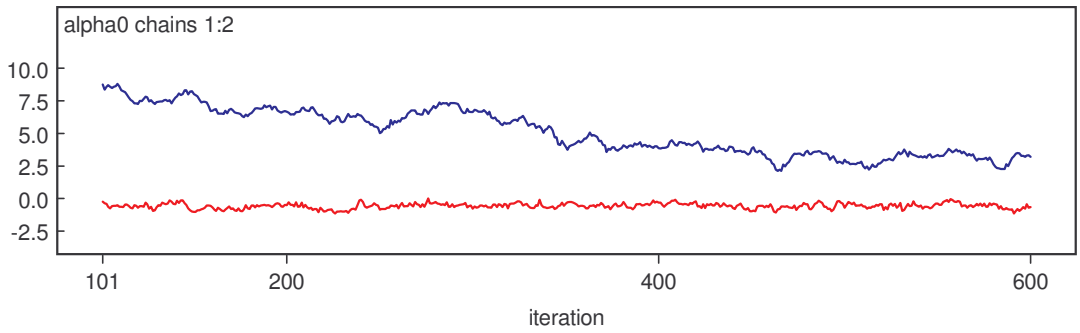
The following plots are examples of: (i) chains for which convergence (in the pragmatic sense) looks reasonable (left-hand-side); and (ii) chains which have clearly not reached convergence (right-hand-side).



If you are running more than one chain simultaneously, the trace and history plots will show each chain in a different colour. In this case, we can be reasonably confident that convergence has been achieved if all the chains appear to be overlapping one another.

The following plots are examples of: (i) multiple chains for which convergence looks reasonable (top); and (ii) multiple chains which have clearly not reached convergence (bottom).





For a more formal approach to convergence diagnosis the software also provides an implementation (see here) of the techniques described in Brooks & Gelman (1998), and a facility for outputting monitored samples in a format that is compatible with the *CODA* software – see here.

How many iterations after convergence?

[top | home]

Once you are happy that convergence has been achieved, you will need to run the simulation for a further number of iterations to obtain samples that can be used for posterior inference. The more samples you save, the more accurate will be your posterior estimates.

One way to assess the accuracy of the posterior estimates is by calculating the Monte Carlo error for each parameter. This is an estimate of the difference between the mean of the sampled values (which we are using as our estimate of the posterior mean for each parameter) and the true posterior mean.

As a rule of thumb, the simulation should be run until the Monte Carlo error for each parameter of interest is less than about 5% of the sample standard deviation. The Monte Carlo error (MC error) and sample standard deviation (SD) are reported in the summary statistics table (see section Obtaining summaries of the posterior distribution).

Obtaining summaries of the posterior distribution

[top | home]

The posterior samples may be summarised either graphically, e.g. by kernel density plots, or numerically, by calculating summary statistics such as the mean, variance and quantiles of the sample.

To obtain summaries of the monitored samples, to be used for posterior inference:

* Select Samples... from the Inference menu.
* Type the name of the parameter in the white box marked node (or select the name from the pull down list, or type "*" (star) to select all monitored parameters).
* Type the iteration number that you want to start your summary from in the white box marked beg: this allows the pre-convergence 'burn-in' samples to be discarded.
* Click once with the LMB on the button marked stats: a table reporting various summary statistics based on the sampled values of the selected parameter will appear.
* Click once with the LMB on the button marked density: a window showing kernel density plots based on the sampled values of the selected parameter will appear.

Please see the manual entry Samples... for a detailed description of the Sample Monitor Tool dialog box (i.e. that which appears when Samples... is selected from the Inference menu).

Changing MCMC Defaults (advanced users only)

Contents

Defaults for numbers of iterations
Defaults for sampling methods

This section shows how to change some of the defaults for the MCMC algorithms used in WinBUGS. Users do so at their own risk, and should make a back-up copy of the relevant default file first (although in case of disaster a new copy of WinBUGS can always be downloaded). The program should be restarted after any edits.

Defaults for numbers of iterations

[top | home]

Options for iteration numbers can be changed temporarily using the *Options* menu (see here for details), but the default values are held in the WinBUGS file System/Rsrc/Registry.odc which may be edited.

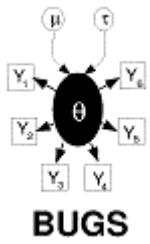
Examples include:

- **UpdaterMetnormal_AdaptivePhase** (default 4000) is the length of the adaptive phase of the general normal-proposal Metropolis algorithm;
- **UpdaterSlice_AdaptivePhase** (default 500) is the adaptive phase of the slice sampling algorithm;
- **UpdaterSlice_Iterations** (default 100000) is how many tries the slice sampling algorithm has before it gives up and produces an error message.

Defaults for sampling methods

[top | home]

It is now possible to change the sampling methods for certain classes of distribution, although this is delicate and should be done carefully. The sampling methods are held in Updater/Rsrc/Methods.odc and can be edited. For example, if there are problems with WinBUGS' adaptive rejection sampler (DFreeARS), then the method "UpdaterDFreeARS" for "log concave" could be replaced by "UpdaterSlice" (normally used for "real non linear") – this has been known to sort out some Traps. However, take care and don't forget to keep a copy of the original Methods.odc file!



Distributions

Contents

Discrete Univariate

Bernoulli
Binomial
Categorical
Negative Binomial
Poisson

Continuous Univariate

Beta
Chi-squared
Double Exponential
Exponential
Gamma
Generalized Gamma

Log-normal
Logisitic
Normal
Pareto
Student-t
Uniform
Weibull

Discrete Multivariate

Multinomial

Continuous Multivariate

Dirichlet
Multivariate Normal
Multivariate Student-t
Wishart

Discrete Univariate

[top | home]

Bernoulli

`r ~ dbern(p)`

$$p^r(1-p)^{1-r}; \quad r = 0, 1$$

Binomial

`r ~ dbin(p, n)`

$$\frac{n!}{r!(n-r)!}p^r(1-p)^{n-r}; \quad r = 0, ..., n$$

Categorical

`r ~ dcat(p[])`

$$p[r]; \quad r = 1, 2, ..., \dim(p); \quad \sum_i p[i] = 1$$

Negative Binomial

`x ~ dnegbin(p, r)`

$$\frac{(x+r-1)!}{x!(r-1)!}p^r(1-p)^x; \quad x = 0, 1, 2, ...$$

Poisson

`r ~ dpois(lambda)`

$$e^{-\lambda}\frac{\lambda^r}{r!}; \quad r = 0, 1, ...$$

Continuous Univariate

[top | home]

Beta

`p ~ dbeta(a, b)`

$$p^{a-1}(1-p)^{b-1}\frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}; \quad 0 < p < 1$$

Chi-squared

`x ~ dchisqr(k)`

$$\frac{2^{-k/2}x^{k/2-1}e^{-x/2}}{\Gamma(\frac{k}{2})}; \quad x > 0$$

Double Exponential

`x ~ ddexp(mu, tau)`

$$\frac{\tau}{2}\exp(-\tau\,|x-\mu|); \quad -\infty < x < \infty$$

Exponential

`x ~ dexp(lambda)`

$$\lambda e^{-\lambda x}; \quad x > 0$$

Gamma

`x ~ dgamma(r, mu)`

$$\frac{\mu^r x^{r-1}e^{-\mu x}}{\Gamma(r)}; \quad x > 0$$

Generalized Gamma

`x ~ gen.gamma(r, mu, beta)`

$$\frac{\beta}{\Gamma(r)}\mu^{\beta r}x^{\beta r-1}\exp\left[-(\mu x)^{\beta}\right]; \quad x > 0$$

Log-normal

`x ~ dlnorm(mu, tau)`

$$\sqrt{\frac{\tau}{2\pi}}\frac{1}{x}\exp\left(-\tfrac{\tau}{2}(\log x-\mu)^2\right); \quad x > 0$$

Logistic

`x ~ dlogis(mu, tau)`

$$\frac{\tau\exp(\tau(x-\mu))}{(1+\exp(\tau(x-\mu)))^2}; \quad -\infty < x < \infty$$

Normal

`x ~ dnorm(mu, tau)`

$$\sqrt{\frac{\tau}{2\pi}}\exp\left(-\tfrac{\tau}{2}(x-\mu)^2\right); \quad -\infty < x < \infty$$

Pareto

`x ~ dpar(alpha, c)`

$$\alpha c^{\alpha}x^{-(\alpha+1)}; \quad x > c$$

Student-t

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

$$\frac{\Gamma\left(\frac{k+1}{2}\right)}{\Gamma\left(\frac{k}{2}\right)}\sqrt{\frac{\tau}{k\pi}}\left[1+\tfrac{\tau}{k}(x-\mu)^2\right]^{-(k+1)/2};$$

$$-\infty < x < \infty; \quad k \geq 2$$

Uniform

`x ~ dunif(a, b)`

$$\frac{1}{b-a}; \quad a < x < b$$

Weibull

`x ~ dweib(v, lambda)`

$$v\lambda x^{v-1}\exp(-\lambda x^{v}); \quad x > 0$$

Discrete Multivariate

[top | home]

Multinomial

`x[] ~ dmulti(p[], N)`

$$\frac{(\sum_i x_i)!}{\prod_i x_i!}\prod_i p_i^{x_i};$$

$$\sum_i x_i = N; \quad 0 < p_i < 1; \quad \sum_i p_i = 1$$

Continuous Multivariate

[top | home]

Dirichlet

`p[] ~ ddirch(alpha[])`

$$\frac{\Gamma(\sum_i \alpha_i)}{\prod_i \Gamma(\alpha_i)}\prod_i p_i^{\alpha_i - 1};$$

$$0 < p_i < 1; \quad \sum_i p_i = 1$$

Multivariate Normal

`x[] ~ dmnorm(mu[], T[,])`

$$(2\pi)^{-d/2}|T|^{1/2}\exp\left(-\tfrac{1}{2}(x-\mu)'T(x-\mu)\right);$$

$$-\infty < x < \infty$$

Multivariate Student-t

`x[] ~ dmt(mu[], T[,], k)`

$$\frac{\Gamma((k+d)/2)}{\Gamma(k/2)\,k^{d/2}\pi^{d/2}}|T|^{1/2}$$

$$\times\left[1+\tfrac{1}{k}(x-\mu)'T(x-\mu)\right]^{-(k+d)/2};$$

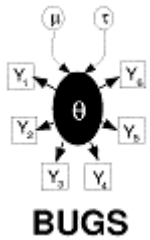
$$-\infty < x < \infty; \quad k \geq 2$$

Wishart

`x[,] ~ dwish(R[,], k)`

$$|R|^{k/2}|x|^{(k-p-1)/2}\exp\left(-\tfrac{1}{2}\operatorname{Tr}(Rx)\right);$$

x symmetric & positive definite



References

Best N G, Cowles M K and Vines S K (1997) *CODA: Convergence diagnosis and output analysis software for Gibbs sampling output, Version 0.4*. MRC Biostatistics Unit, Cambridge: http://www.mrc-bsu.cam.ac.uk/bugs/classic/coda04/readme.shtml

Breslow N E and Clayton D G (1993) Approximate inference in generalized linear mixed models. *Journal of the American Statistical Association*. **88**, 9-25.

Brooks S P (1998) Markov chain Monte Carlo method and its application. *The Statistician*. **47**, 69-100.

Brooks S P and Gelman A (1998) Alternative methods for monitoring convergence of iterative simulations. *Journal of Computational and Graphical Statistics*. **7**, 434-455.

Carlin B P and Louis T A (1996) *Bayes and Empirical Bayes Methods for Data Analysis*. Chapman and Hall, London, UK.

Congdon P (2001) *Bayesian Statistical Modelling*. John Wiley & Sons, Chichester, UK.

Crowder M J (1978) Beta-binomial Anova for proportions. *Applied Statistics*. **27**, 34-37.

Gamerman D (1997) Sampling from the posterior distribution in generalized linear mixed models. *Statistics and Computing*. **7**, 57-68.

Gelman A, Carlin J C, Stern H and Rubin D B (1995) *Bayesian Data Analysis.* Chapman and Hall, New York.

Gilks W (1992) Derivative-free adaptive rejection sampling for Gibbs sampling. In *Bayesian Statistics 4*, (J M Bernardo, J O Berger, A P Dawid, and A F M Smith, eds), Oxford University Press, UK, pp. 641-665.

Gilks W R, Richardson S and Spiegelhalter D J (Eds.) (1996) *Markov chain Monte Carlo in Practice*. Chapman and Hall, London, UK.

Neal R (1997) Markov chain Monte Carlo methods based on 'slicing' the density function. *Technical Report 9722*, Department of Statistics, University of Toronto, Canada: http://www.cs.utoronto.ca/~radford/publications.html

Neal R (1998) Suppressing random walks in Markov chain Monte Carlo using ordered over-relaxation. In *Learning in Graphical Models*, (M I Jordan, ed). Kluwer Academic Publishers, Dordrecht, pp. 205-230. http://www.cs.utoronto.ca/~radford/publications.html

Roberts G O (1996). Markov chain concepts related to sampling algorithms. In W R Gilks, S Richardson and D J Spiegelhalter (Eds.) *Markov chain Monte Carlo in Practice*. Chapman and Hall, London, UK.

Spiegelhalter D J, Best N G, Carlin B P and van der Linde A (2002) Bayesian measures of model complexity and fit (with discussion). *J. Roy. Statist. Soc. B*. **64**, 583-640.

Tierney L (1983) A space-efficient recursive procedure for estimating a quantile of an unknown distribution. *SIAM J. Sci. Stat. Comput*. **4**, 706-711.