# Introduction to BUGS

BUGS (Bayes Using Gibbs Sampling) is a statistical software package designed specifically to do Bayesian analyses of simple to intermediate complexity using Markov Chain Monte Carlo (MCMC) numerical simulation methods. The great thing about BUGS is that it keeps a lot of the mathematical and computational details "under the hood" so that you can focus on the structure of your model at a high level rather than being bogged down in the details. Also, since BUGS is designed just to do Bayesian MCMC computation it is very efficient at this, which is nice since Bayesian MCMC computations can be time consuming. In this tutorial we will work through the basics of using BUGS: writing and compiling a model, loading data, executing the model, and evaluating the numerical output of the model.

## Resources

BUGS is open-source software, and can be freely downloaded from the OpenBUGS website at http://openbugs.net/. Source code, several Linux binaries, and a Windows port are available in the *Downloads* section, and the latter can be run under Mac OS X via WINE. There is also a separate *User Contributed Code* section, which contains information on the R packages BRUGS and R2OpenBUGS, which allow the BUGS engine to be run directly from R.

The OpenBUGS website is also a great first stop for information on using the software. The *Documentation* section contains the definitive OpenBUGS User Manual, tutorials, and other useful information, and the *Community* page links to the BUGS email list and the obsolete WinBUGS site which contains many useful resources for learning more about BUGS specifically and Bayes in general.

## Software

Click on the OpenBUGS icon to start the software. This will open up a fairly uninteresting software window. Let's start by looking at some of the WinBUGS menus. First click ***File > New*** to open up a new script window in BUGS. If you are familiar with R (or similar), you'll soon notice that one of the major differences between R and BUGS lies in how you input code to be evaluated. In R you can type commands directly at the command prompt (the ">" symbol), or write it in a script to be evaluated line by line, in blocks, or all at once. By contrast, BUGS has no command prompt and so requires all code to be written in scripts. Furthermore, the script can only be evaluated all at once, so in general each script contains a single discrete analysis.

Next, choose the menu ***Manuals > OpenBUGS User Manual***. This will open up a window that shows the BUGS manual, which is the same as you would have seen on the BUGS website. Click on *Contents* and take a look at the list of topics covered. There's a lot of information in the manual, but we will focus our attention on a few sections that you'll find yourself coming back to repeatedly. The first of these is *Model Specification,* which provides a lot of detail on writing models and formatting data in BUGS, and also lists all of the functions and distributions available in the program. The naming convention for distributions in BUGS is very similar to the convention in R, but there are a few cases where the parameterizations are different (e.g. Weibull), so it is always good to check the manual to see that the values you are passing to a BUGS distribution are what you think they are. One very important example of this is that the Normal distribution in BUGS, *dnorm*, is parameterized in terms of a mean and **precision** (1/variance), rather than a mean and standard deviation as in R.

Next, select the menu ***Examples > Examples Vol 1***. BUGS has three volumes of examples that provided written explanations of analyses with the BUGS code and data embedded in them so that you can run the code directly from the example. Working through these examples is a great way to learn more about how BUGS works, and *when you are analyzing your own data it is often easiest to start from an existing example and modify it to meet your needs rather than starting from scratch.*

### Analysis in BUGS at a glance

All analyses in BUGS follow the same basic outline. This section will present a general overview of the steps involved and then work through a simple example to explain the details of each step. Please run this example as you read along.

## BUGS "recipe"

1. Write model
2. Specify data
3. Specify initial conditions (*optional*)
4. *Don't forget to save your script before running!*
5. Open **Model Specification Tool (*Model > Specification*)**
6. Highlight model, click *check model*
7. Highlight data, click *Load Data*
8. Set number of chains (typically 3-5)
9. Click *compile*
10. Click *gen inits* and/or highlight initial conditions and click *load inits*
11. Open **Sample Monitor Tool (*Inference > Samples*)**
12. Specify variables to track (enter variable name in window, click *set*)
13. Open **Update Tool (*Model > Update*)**
14. Click *update* to run sampler
15. Use **Sample Monitor Tool (*Inference > Samples...*)** to evaluate model fit
16. Use **Update Tool** to run longer if necessary

### 1. Write model

As our first example, let's consider the simple case of finding the mean of a normal distribution with known variance. This problem has an exact analytical solution that we can compare the numerical results. Here, we're considering a case where both the likelihood of the data and the prior for the mean are assumed to be normal:

$$L = p(y|\mu) = N(y|\mu,\sigma^2)$$

$$\text{prior} = p(\mu) = N(\mu|\mu_0,\tau^2)$$

In BUGS the specification of any model begins with the word "model" and then encapsulates the rest of the model code in curly brackets:

```
model {
  ## model goes here ##
}
```

When writing models in BUGS we have to specify the *data model*, *process model*, and *parameter model*, but we don't need to specify explicitly the connections between them—BUGS figures them out based on the conditional probabilities involved. Deterministic calculations (e.g. process model) always make use of an *arrow* ( **<-** ) for assignment like in R, while assignment of random variables is always done with a *tilde* ( **~** ). The equal sign ( **=** ) is not used for either process or data models. Furthermore, deterministic calculations and distributions can not be combined in the same line of code. For example, a regression model where we're trying to predict data *y* based on observation *x* might include

```
mu <- b0 + b1* x        # process model

y ~ dnorm(mu,tau)       # data model
```

but in BUGS the same model can NOT be expressed as

```
y ~ dnorm(b0 + b1* x, tau)
```

Putting this all together, to specify our first model of a normal likelihood and normal prior, the BUGS code is:

```
model {
  mean ~ dnorm(53,0.04)  # prior on the mean
  prec <- 1/185.0        # known variance
  Y ~ dnorm(mean,prec)   # data model
}
```

The first line of this model specifies the prior and says that *mean* is a random variable (~) that is Normally distributed (*dnorm*) with an expected value of 53 and a precision of 0.04 (equivalent to variance of 25, standard deviation of 5). The second line says that the *prec* is calculated deterministically as 1/185. Remember that in this model we're assuming that the variance is known (and in this case it is 185). The third line specifies the likelihood and says that there is a single data point *Y* that is an instance of the Normally-distributed random variable with expected value *mean* and precision *prec*.

### 2. Specify data

The next step is to specify the data for this model. There are two ways to do this in BUGS, either as a *list* or as a *table*. We'll start with the list format, which is the same as the R list format and more flexible. All in all, BUGS is not designed for loading and manipulating data, and we recommend that you do that in R or in a spreadsheet first and then cut and paste into your BUGS script file. (Aside: it is possible to download R packages that call BUGS directly from R—this is the easiest approach if you are dealing with *large* data sets or a large number of runs.)

The data list in BUGS begins with the word "list" and a set of parentheses. The data variable names and the data themselves go between these parentheses. For this example with a single data point, we have a line that follows the model that simply says:

```
list(Y=42)
```

It is important that the variable names in the list match those in the model, because BUGS will check that the data set has the right names and is of the correct length.

### 3. Specify initial conditions (optional)

The next step is to specify the initial conditions for the model. In BUGS this is (sometimes) optional because if initial conditions are not specified then the code will draw them randomly from the priors. If you use informative priors this is not usually a problem. If you use uninformative priors, however, the initial parameter value can start far from the center of the distribution and take a very long time to converge, as in other optimization contexts. In addition, some BUGS models of even intermediate complexity may produce cryptic errors if you try to let BUGS generate all initial conditions on its own. With that said, a nice feature in BUGS is that we only have to specify some of the initial conditions, and those that we don't specify will be initialized based on the priors. Because of this, we can often get away with only initializing a subset of variables while still avoiding errors and convergence problems.

BUGS initial conditions are specified using a list format similar to that described above for entering data. We will often want to run multiple independent MCMC chains that each start from different initial conditions, and to do so we provide one initial condition list for each chain. In this example we’ll run three chains starting at three different values for the *mean* parameter. We specify this in BUGS as:

```
list(mean = 40)
list(mean = 45)
list(mean = 50)
```

### 4. *Don’t forget to save your script before running!*

Remember to save early and often in BUGS, which is a rather aptly-named piece of software. It is particularly important to save your script before you run it in case your model crashes the program.

### 5. Open Model Specification Tool

In the menus, choose ***Model > Specification***. Running a model in BUGS requires opening a bunch of small windows to access different tools. It’s helpful to arrange them so they're not overlapping and you can see them all without closing or moving them.

### 6. Check model (Specification Tool)

In the script, highlight the word “model”, and then in the **Specification Tool** click *check model*. When you do this the message bar at the very bottom of BUGS will either say “model is syntactically correct” or will give an error message that indicates that there is a bug in the code. You'll need to keep an eye on the message bar because, in a true reflection of the software's British origin, the information is vital but understated (a U.S. software designer probably would have made a large & colorful pop-up to make sure you didn't miss these messages).

A very important behavior of BUGS to note is that any time you encounter an error you have to start back at the model specification step. For example, if your data is specified incorrectly and you get an error message, you can't just fix the data and hit *load data* again; you have to restart from *check model*.

### 7. Load data (Specification Tool)

Highlight your data in the script and click *load data* in the **Specification Tool**. For our data list all you have to do is highlight the word “list” before clicking *load data*. If data are correctly specified and loaded the message bar will say “data loaded”.

### 8. Set number of MCMC chains (Specification Tool)

Set the number of chains in the **Specification Tool**. We'll use three for this example; 3-5 is typical.

### 9. Compile (Specification Tool)

Click *compile* in the **Specification Tool**. If this is successful the message bar will say "model compiled." Otherwise it will give you an error. In addition to syntax errors, common errors are having data listed in your model that's not in your data set, or having variables in your data that are not used in the model.

### 10. Load initial conditions (Specification Tool)

Begin by highlighting "list" for the first initial condition in your script and then click *load inits* in the **Specification Tool**. When you do this the *for chain* counter in the tool window will switch from 1 to 2, automatically readying you to load initial conditions for chain 2. The message bar will also indicate that you have initialized one chain but others remain uninitialized.

Repeat the highlight, *load inits* procedure for the second and third initial conditions. When you are done the message bar should say "model is initialized." For our model, the "gen inits" button will also become greyed out because there are no initial conditions left to generate. However, if the model had additional variables that we had not initialized, we could now hit *gen inits* to ask BUGS to set the remaining values randomly based on the priors.

As an alternative to specifying some or all of the initial conditions and using *load inits*, we could have just hit *gen inits* from the beginning to generate all necessary initial parameter values from the priors. However, as mentioned above that could have large implications for the rate/success of convergence, and in some cases a model will crash entirely if poor initial parameter values lead to numerical instabilities. In fact, initial conditions are one of the first things to suspect when a model is failing to converge or otherwise misbehaving.

### 11. Open Sample Monitor Tool

Choose ***Inferences > Samples...*** from the menu. The **Sample Monitor Tool** allows you to define which variables in the model are tracked, and later to produce various plots of the tracked variables.

### 12. Specify variables to track (Sample Monitor Tool)

BUGS will only store values for the variables that you ask it to. To request a variable to be tracked, type its name in text field labeled *node*, and then hit the *set* button. You can check the list of variables currently set for tracking by clicking the dropdown triangle on the *node* box. If one of the variables in your model is a vector, you can specify either the name of the whole vector or just specific values to track (e.g. "alpha[3]" would only record the third value of the *alpha* variable, whereas "alpha" would track all values).

In general, variables tracked have to be variables in your model (case-sensitive, typos forbidden). One important exception is that you can always ask BUGS to track the "deviance" of the model (defined as -2 times the log-likelihood). This is particularly useful in multiple-chain models in case not all the chains converge to the same value, in which case you will be interested in the chains with the lowest deviance.

For the model we've specified lets track the variables "mean" and "deviance".

### 13. Open Update Tool

Choose ***Model* > *Update*** from the menus. This simple tool is just for setting the number of iterations to run, and then running them.

### 14. Run sampler (Update Tool)

Initially, you'll want to set *updates* to a small number (e.g. 10) to make sure everything works. Then hit the *update* button to actually run the model. If these initial few steps are successful, you can change *updates* to a larger number (e.g. 1000) and hit *update* again. Note that the second update adds additional steps to the first, rather than starting from scratch. In this way you can repeatedly check the progress of the MCMC and run it longer if need be. The total number of samples you need to run depends on how quickly the model converges, the acceptance rate and autocorrelation of the samples, and the quantities you are interested in estimating (e.g. a good estimate of the posterior mean takes far fewer samples than a good estimate of the confidence intervals).

### 15. Evaluate model fit (Sample Monitor Tool),

BUGS has a number of options for generating default graphs and statistics within the **Sample Monitor Tool**. To access these, you begin by specifying the variable you want to look at in the *node* field by typing its name or selecting it from the dropdown. Note that you must have previously asked BUGS to track the variable of interest, and then performed updates; you can't look at output of a variable that hasn't been tracked for some number of updates, because there won't be any yet.

As an alternative to viewing output for variables one at a time, you can specify an asterisk (*) in the node field to look at all of your tracked variables at once. This is a bad idea if you're tracking lots of variables, but in this case we're not so lets enter a "*".

The remaining fields in the **Sample Monitor Tool** window control settings that apply to all of the available tools (*see below*). The *chains* settings let you specify which MCMC chains to analyze, but in most cases you should leave this at its default (use all chains). The *beg* and *end* values set the endpoints of the chain segment to analyze. Importantly, *beg* (short for "beginning") determines how many samples to skip at the beginning of the chains, which is how you tell BUGS to ignore the pre-convergence burn-in period in your model output. The *thin* field is also very important, because it sets the degree to which BUGS will thin the chains to reduce autocorrelation before analyzing them. Intuitively, setting *thin* to 10 retains only every 10th sample for further analysis, etc.

Several tools below will help you choose appropriate values for *beg* and *thin* (identified by asterisks). Note that statistics and plots that you've already generated *are not updated when you choose new* beg *and* thin *values*; if you determine that either value should be changed, you'll have to rerun the plots and statistics you want to see. Thus, it's a good idea to start your analysis of BUGS output by determining your *beg* and *thin* values.

The rest of the **Sample Monitor Tool** is a collection of buttons that each run a different analysis and/or generate a figure. Details for each are described below. We underlined the names of the tools we consider most important for analyzing your MCMC output, and asterisked those that directly help you choose appropriate *beg* and *thin* values.

### STATS

Generates a table showing the posterior mean, standard deviation, median, and 95% CI of the selected variable(s). It will also show the MCMC sample size and the MCMC standard error, which is a reflection of the *precision* of the numerical approximation. As you run an MCMC longer and longer this error will decrease but the posterior standard deviation will not, because its width is a reflection of the sample size of the DATA rather than the sample size of the MCMC. The MCMC error gives you guidance as to how many digits you should interpret from numerical approximation. Information from the stats window can be copied and pasted elsewhere for safe keeping.

### DENSITY

Creates histogram graphs of the posterior of each selected variable. We're looking for smooth densities and in general distributions that are unimodal. While truly multimodal posterior densities can exist, much more commonly they are indicative of a lack of convergence. All plots in BUGS, including the density plots, can be copied to the clipboard by right clicking on the graph and selecting *copy* and then pasted to another program (e.g. Microsoft Word). You can also change the format of a graph by right clicking and selecting *properties*.

### CODA

Opens one window for each chain and displays the raw MCMC values. This is useful if you want to save these values for subsequent analysis (e.g. in R). The “coda” file format is that used by the “Convergence Diagnostics and Output Analysis” software, which exists as an R package named “coda” and includes a number of additional convergence statistics and graphing options beyond those built into BUGS.

### TRACE

Plots the recent history of the MCMC chains, with each chain in a different color. In a model that has converged these chains will be overlapping and will bounce around at random, preferably looking like white noise but sometimes showing longer-term trends. The plot updates in real time if you hit *update* again in the **Update Tool**. It can be very useful to open the trace window at the start of a model run to follow the progress of the MCMC.

### JUMP

Creates a plot of the squared jump distance of the MCMC. This allows you to see if the model is taking large or small steps

### BGR DIAGRAM*

Plots the Brooks–Gelman–Rubin statistic, which is based on the ratio of within-chain to among-chain variability and used to assess whether/when an MCMC simulation has converged. Generally, convergence is indicated by the red line in the BGR plot converging to 1 and the other lines to approximately constant values. Be aware that you should check all the variables in your model for convergence because sometimes one appears to have converged while another hasn't yet.

**HISTORY***

Is similar to trace but shows the full history of the MCMC and does not update dynamically. You will want to assess when the model has converged based on this diagram, the BGR diagram, and the quantile diagram, and then set that time point of convergence as the *beg* value in the **Sample Monitor Tool**. This will cause all values prior to convergence to be ignored in making graphs and calculating summary statistics.

**ACCEPT**

Graph of the acceptance rate of the MCMC. Generally models that use a Gibbs sampler will always be at 1 (100% acceptance) while other numerical methods typically accept around 30-50% of proposed values once they tune their step size. In general you'll want to thin the MCMC by at least 1/(acceptance rate).

**QUANTILES**

Provides a moving-average estimate of the median and CI. This is a useful convergence diagnostic if you are interested in estimating the CI, because it shows whether these statistics have converged for the different chains, and as noted previously this takes longer than convergence of the mean or other central-tendency statistics.

**AUTO COR***

Generates an autocorrelation diagram for the MCMC chains, with lag on the x-axis and correlation coefficient on the y-axis. At lag 0 the chain is always perfectly correlated with itself (autocorrelation = 1). A lag of one shows the correlation between each value in the MCMC with the one next to it, a lag of two gives the correlation between values that are separated by two, etc. Typically the diagram looks roughly exponential, and we are looking for the lag where it reaches approximately zero (in practice, ≤0.2 is a good rule of thumb). You want to set *thin* in the **Sample Monitor Tool** to this lag, which will make your MCMC samples approximately independent after thinning. This method of choosing how much to thin is usually more conservative (i.e. it throws out more samples) than evaluating the acceptance rate. Again, since the *thin* value affects all other statistics, you need to recompute any output you are keeping (e.g. density plots or summary statistics) if you change it.

### 16. Run more samples if necessary (Update Tool)

Both burn-in and thinning reduce the number of samples in the chains you finally analyze. As a broad rule of thumb, you'll want to run the MCMC until you have at least 5,000 samples left *after* accounting for *beg* and *thin*. If your diagnostics using the tools described above lead to too few samples in the resulting chains, add updates using the **Update Tool** and then re-evaluate the output.

## Case Study: Forest Stand Characteristics

Now that we have a basic feel for BUGS we'll look at ways to progressively increase the complexity of our model. For the two examples below, we provide model code for estimating the distribution of tree diameters in a dataset from the Duke loblolly pine FACE site. Once input into a BUGS script, the models can be run by following the recipe outlined above. Data for both analyses is given below in BUGS list format, which can be copied and pasted into your script:

```
list(Y = c(20.9, 13.6, 15.7, 6.3, 2.7, 25.6, 4, 20.9, 7.8, 27.1, 25.2, 19, 17.8, 22.8, 12.5, 21.1,
22, 22.4, 5.1, 16, 20.7, 15.7, 5.5, 18.9, 22.9, 15.5, 18.6, 19.3, 14.2, 12.3, 11.8, 26.8, 17, 5.7,
12, 19.8, 19, 23.6, 19.9, 8.4, 22, 18.1, 21.6, 17, 12.4, 2.9, 22.6, 20.8, 18.2, 14.2, 17.3, 14.5,
8.6, 9.1, 2.6, 19.8, 20, 22.2, 10.2, 12.9, 20.9, 21.1, 7.3, 5.8, 23.1, 17, 21.5, 10.1, 18.4, 22.6,
21.2, 21.5, 22.4, 17.3, 16, 25, 22.4, 23.9, 23, 21.9, 19, 28.6, 16, 22.5, 23.2, 8.7, 23.4, 15.3,
25.6, 19.2, 17.4, 23.8, 20.4, 19, 3.6, 23.4, 19.6, 17.5, 16.5, 22, 19.7, 7.35, 18, 17.8, 9.6, 15, 12,
17.7, 21.4, 17, 22.1, 18.9, 15.05, 12.9, 19.3, 15.3, 13.6, 15.4, 10.6, 11.3, 11.8, 22.2, 22.2, 13.1,
7.4, 4.5, 11.7, 19.5, 19.9, 11.6, 13.9, 15.5, 11, 18.6, 17.6, 12.7, 20.9, 18.8, 22.4, 21.2, 18.2,
15.3, 13.6, 7.3, 17.4, 17.4, 10.5, 22.9, 23.2, 13.8, 14.8, 22.2, 20.9, 13, 18.9, 19, 15.2, 16.8, 18,
24.6, 15.4, 17.2, 23.2, 22.8, 25.5, 7.8, 6, 6.4, 19, 13.5, 23.7, 18, 22.2, 22.4, 9.3, 13.7, 18.9,
20.5, 23.3, 20.8, 18.4, 4.5, 12.2, 16.9, 13.5, 17.8, 16.9, 20.4, 19.5, 22.2, 24.5, 21.2, 16.5, 18,
16.4, 3.9, 17.9, 22, 12.9, 21, 18, 9.2, 15.9, 8.1, 8.3, 10.7, 12, 19.9, 13.6, 17.3, 11.5, 12.4, 15.1,
22, 19.3, 17.5, 14.5, 14.7, 17.5, 19.6, 12.9, 20.3, 17.9, 20.2, 18.3, 9.5, 19, 21, 13.1, 20.4, 16.3,
18.3, 11.8, 23.3, 15.2, 20, 17.9, 12, 19.6, 18.5, 16.2, 10.9, 17.8, 13.8, 10, 17.9, 15.6, 20.3, 14.9,
18.6, 12.5, 18.2, 16, 18.7, 18, 15.3, 19, 17.9, 15.8, 17.7, 14.4, 19.6, 18.3, 18.7, 17.8, 18, 10.1,
18.8, 16.4, 21.2, 16.6, 16.7, 17.8, 16.5, 19.3, 16.3, 14.2, 13, 9.4, 19.7, 13.4, 2.6, 17.6, 16.7,
17.6, 5.8, 17.6, 20.1, 18.2, 16.7, 14, 13.9, 5.1, 16.6, 3.9, 17.5, 18))
```

We'll begin by expanding the model we specified above to account for more than one observation. As a first step let's still assume that the variance is known. Our data set has 297 values, so when specifying the model in BUGS we'll need to loop over each value to calculate the likelihood of each data point and use the vector index notation, `[i]` , to specify which value we're computing.

```
model {
  mean ~ dnorm(20,0.01)        # prior mean
  prec <- 1/27.373             # known variance
  for(i in 1:297){             # loop over observations
    Y[i] ~ dnorm(mean,prec)    # data model
  }
}
```

Next we'll modify our model to account for uncertainty in the stem-diameter variance. This only requires modifying one line—changing the precision from a constant to a prior. We've also added another line that calculates standard deviation from the precision, because it is often more intuitive to interpret. The nice thing about working with Monte Carlo samples is that any such calculations we perform produce a valid transformation of the parameter distribution. Similar calculations done analytically require tedious amounts of calculus to account for nonlinearities and Jensen's Inequality.

```
model {
  mean ~ dnorm(20,0.01)
  prec ~ dgamma(0.1,0.1)
  sd <- sqrt(1/prec)
  for(i in 1:297){
    Y[i] ~ dnorm(mean,prec)
  }
}
```

**Lab Report Task 1:**
Run the unknown mean/fixed variance model to estimate the mean tree diameter. Include in your report the following:

- Table of summary statistics
- Graph of parameter densities
- Plot of MCMC "history"
- Report the length of the MCMC (i.e. the total number of "updates"), the number of chains, "beg", and "thin" you used and any graphs/statistics you used to **justify those settings**

**Lab Report Task 2:**
Run the unknown mean/unknown variance model to estimate both the mean tree diameter and the standard deviation. Include in your report the following:

- Table of summary statistics for both the mean and sd
- Graph of parameter densities
- Plot of MCMC "history"
- Report the length of the MCMC, number of chains, "beg", and "thin" you used and any graphs/statistics you used to justify those settings
- Note any changes in the distribution of the mean (shape, location, tails) compared to Task 1

Unfortunately, BUGS does not support Markdown, so you will need to cut-and-paste figures and results from BUGS software into a document and submit that document (pdf, docx, etc).