# Bayesian Data Analysis

Vanda Inácio & Ken Newman

University of Edinburgh

Semester 2, 2017/2018

## The BUGS language

- ↪ We will learn now the basics of the BUGS language.
- ↪ BUGS stands for **B**ayesian Inference **U**sing **G**ibbs **S**ampling and began in 1989.
- ↪ Its basic philosophy was to separate the language used to describe a model from the actual programs used to carry out the computations (the *engine*).
- ↪ This approach has two attractive advantages:
  - ↪ One can easily specify complex models without requiring extensive knowledge of Bayesian computational tools.
  - ↪ The language has remained stable and consistent over time, even as the underlying programs which actually do the sampling are constantly changing.

## The BUGS language

- The syntax of the BUGS language is relatively straightforward. Every line does one of the following things:
  - Defines a stochastic node
    ```
    theta~dbeta(1,1)
    ```
  - Defines a deterministic node
    ```
    tau=pow(sigma,-2)
    ```
  - Provides a comment
    ```
    ## Prior
    ```
  - Defines a loop
    ```
    for (i in 1:n)
    ```
- Common distributions in the BUGS language are `dbin`, `dpois`, `dunif`, `dnorm`, `dgamma`.
- The normal is parameterised in terms of its mean and precision (=1/ variance).
- Functions cannot be used as arguments in distributions (one needs to create new nodes).
- For more details on the distributions and functions available in BUGS, check the online manual.

## The BUGS language

- ↪ Historically, the most widely used BUGS engine has been a program called WinBUGS. It began in the early 1980s under David Spiegelhalter at the Medical Research Council Biostatistics Unit in Cambridge, UK.

- ↪ Further developments by the developers of WinBUGS, however, have focused on a somewhat different program called OpenBUGS; the features and appearance of the programs are similar, but OpenBUGS is more compatible with different operating systems.

- ↪ Information about WinBUGS and OpenBUGS available at

  `https://www.mrc-bsu.cam.ac.uk/software/bugs/`

- ↪ A separate project, JAGS (Just Another Gibbs Sampler), can be thought of as an engine for running BUGS models, although strictly speaking, the language syntax for the two programs is not always identical.

  `http://mcmc-jags.sourceforge.net/`

## The BUGS language

$\hookrightarrow$ Another two recent engines are `STAN` and `NIMBLE`, whose links are, respectively

`http://mc-stan.org/`

`https://r-nimble.org/nimble-a-new-way-to-do-mcmc-and-more-from-bugs-code-in-r-3`

## The BUGS language

$\hookrightarrow$ All engines carry out the following steps for fitting BUGS models:

1. Checking model syntax.
2. Reading in data.
3. Compiling the model.
4. Initialising the simulation.
5. Sampling.
6. Report results.

## The BUGS language
### Running WinBUGS/OpenBUGS

1. Go to the *File* menu and click on *New*. This opens a window entitled '*untitled1*'.
2. In this new window specify the model, the data, and starting values for the computations. We will see that, in fact, starting values are not necessarily needed.
3. Back in the (Win/Open)BUGS window, open the menu *Model* and click on *Specification*. Highlight `model` by double-click. Click on *Check model*. Highlight start of data. Click on *Load* data. Click on *Compile*. Highlight start of initial values. Click on *Load* inits. Click on *Gen Inits* if more initial values needed. Note that messages are printed at the bottom left describing whether or not these operations are working.
4. Open *Update* from *Model* menu, and *Samples* from *Inference* menu. Type nodes to be monitored into *Sample Monitor*, and click set each. Click on *Update* to generate samples.
5. Type * into *Sample Monitor*, and click *stats*, etc to see results on all monitored nodes. Perform more updates.

# The BUGS language

## Running WinBUGS/OpenBUGS

- Let us come back to the drug example of lecture and practical 1. Remember that we have assumed $Y \sim \text{Binomial}(n, \theta)$ and $\theta \sim \text{Beta}(a, b)$, with $a = 9.2$ and $b = 13.8$. The observed $y$ is 15 and $n$ is 20.

- We have seen that the posterior is also a beta distribution with parameters 24.2 and 18.8. We have also seen that the posterior mean is 0.563.

- In BUGS, the model syntax is

```
#Model
model{
y~dbin(theta,n)
theta~dbeta(a,b)
}

#Data
list(y=15, n=20, a=9.2, b=13.8)

#Initial values ( a different one for each chain, 2 in this case)
list(theta=0.1)
list(theta=0.5)
```

## The BUGS language

### Running WinBUGS/OpenBUGS

↪ We can also easily compute the posterior predictive probability as well as the probability of exceeding a critical threshold. For instance, in the first practical, we've computed the probability of observing at least 25 positive responses in 40 new trials. The BUGS model is as follows:

```
#Model
model{
y~dbin(theta,n)
theta~dbeta(a,b)
ypred~dbin(theta,m)
pcrit<-step(ypred-ncrit)
}

#Data
list(n=20, y=15,a=9.2, b=13.8, m=40, ncrit=25)
```

↪ Note that `step(a-b)` equals 1 if $a - b \geq 0$ and 0 otherwise.

## The BUGS language

### Running WinBUGS/OpenBUGS

↪ To recap, the basic steps are

1. Specify the model
   - (a) Load the model
   - (b) Load the data
   - (c) Load/generate initial values
2. Run the sampler
   - (a) Monitor nodes
   - (b) Update model
   - (c) Repeat as necessary

↪ For further and more detailed instructions we refer to the WinBUGS 14 manual (file `winbugs_user_manual14.pdf` on Learn).

# The BUGS language

## Running WinBUGS/OpenBUGS

- ↪ One can run WinBUGS/OpenBUGS directly, we don't need `R` installed on our machine to do so, but it's both easier and more useful to use `R` as an interface.
- ↪ Several `R` interfaces exist, namely `R2WinBUGS`, `BRugs`, and `R2OpenBUGS`.
- ↪ The main problem is that they do not work well (or at all) in MAC computers.
- ↪ Fortunately, JAGS runs smoothly on all operating systems (mac os, windows, unix/linux) and `R` interface is also available, through the packages `R2jags`, `rjags`, and `runjags`.
- ↪ We also argue that JAGS is also more contemporaneous and actively maintained (by Martin Plummer).

## The BUGS language

### Running JAGS via `rjags`

↪ In general, the steps to using JAGS to produce samples are:

1. Download JAGS at

   `http://sourceforge.net/projects/mcmc-jags/files/`

2. Install the `rjags` package in `R`, which should automatically find your JAGS installation.
3. Specify the statistical model (likelihood and prior) using the model command.
4. Compile the model using `jags.model`.
5. Generate samples using `update` and `coda.samples`.

↪ As it was already mentioned, there are, at least, 3 `R` interfaces for JAGS. The model syntax is the same for the three interfaces, what changes is how relevant information is extracted. We will stick with `rjags`, although you are free to choose your preferred interface.

# The BUGS language

## Running JAGS via `rjags`

- We will learn `rjags` as we go! As an introductory example, we will go back to the example where we have normally distributed data where both the mean and the variance are unknown, i.e., $Y_i \overset{\text{iid}}{\sim} \text{N}(\mu, \sigma^2)$, with $\mu$ and $\sigma^2$ unknown (see `R` script `jags_unknown_mean_and_variance`).

- Note that BUGS language uses the precision, instead of variance, in the parameterisation of the normal distribution. Therefore, we assume $\mu \sim \text{N}(\mu_0, \sigma_0^2)$ and $\sigma^{-2} \sim \text{Gamma}(a, b)$, with $\mu_0 = 0$, $\sigma_0^2 = 100$, and $a = b = 0.1$.

- First we must load the `rjags` package

```
require(rjags)
```

## The BUGS language

### Running JAGS via `rjags`

↪ The program specifying the model (BUGS) code must be put in a separate file which is then read by JAGS. When working in `R` this is most conveniently done by saving the model in an object, e.g., `model_string="model{...}"` and then using the `R` function `textConnection` when passing the model to the `jags.model` function.

```
model_string <- "model{

  # Likelihood
  for(i in 1:n){
    y[i]~dnorm(mu,inv.var)
  }

  # Prior for mu
  mu~dnorm(mu0,inv.var0)
  inv.var0=1/sigma02

  # Prior for the inverse variance
  inv.var~dgamma(a, b)

  # Compute the variance
  sigma2=1/inv.var
}"
```

# The BUGS language

## Running JAGS via `rjags`

↪ Alternatively we can use the `R` function `cat`, to put the model in a file, say, `m1.jag`

```
cat("model{

# Likelihood
for(i in 1:n){
y[i]~dnorm(mu,inv.var)
}

# Prior for mu
mu~dnorm(mu0,inv.var0)
inv.var0=1/sigma02

# Prior for the inverse variance
inv.var~dgamma(a, b)

# Compute the variance
sigma2=1/inv.var
}",
file="m1.jag")
```

↪ Personally, I prefer the first option.

## The BUGS language

### Running JAGS via `rjags`

- ↪ The BUGS language is declarative, i.e., it is not executed as the program runs. Instead it is a specification of the model structure, and after the model is set up JAGS will decide how best to go about the MCMC simulation.
- ↪ We now specify the parameters we will feed into JAGS (data and priors)

```
data=list(y=y,n=n,mu0=mu0,sigma02=sigma02,a=a,b=b)
```

- ↪ Regarding starting values, we can supply them, although JAGS in most cases will however be able to generate them. In case we want to monitor convergence we will normally run several chains, so in case we supply the initial values, we must supply starting values for each chain. Here, in this example, we use three chains, hence the initial values is a list of three lists. Each os these lists has as elements one named value for each parameter – in this case there two parameters `mu` (mean) and `inv.var` (precision).

```
inits=list(list(mu=mean(y),inv.var=1/var(y)), list(mu=0,inv.var=1), list(mu=10,inv.var=0.1))
```

## The BUGS language

### Running JAGS via `rjags`

↪ Once the model, the data, and the initial values (in case we want to supply them) we ask JAGS, through the function `jags.model` to compile and initialise the model.

```
model=jags.model(textConnection(model_string),n.chains=3,data=data,inits=inits)
```

↪ As this step, we obtain the following

```
Compiling model graph
   Resolving undeclared variables
   Allocating nodes
Graph information:
   Observed stochastic nodes: 150
   Unobserved stochastic nodes: 2
   Total graph size: 160

Initializing model
```

↪ In case we have used the function `cat` to define the model and saved it as file `m1.jag`, we would instead write

```
model=jags.model("m1.jag",n.chains=3,data=data,inits=inits)
```

# The BUGS language

## Running JAGS via `rjags`

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

↪ For `coda.samples` we must specify

- ↪ The variables (nodes) that we want to monitor in the subsequent cycles of the chain. This is done using the argument variable.names. In our case the variables we want to monitor are called `mu` and `sigma2`.
- ↪ How many iterations to run the chain (n.iter).
- ↪ How often we sample the specified parameters and retain the results in memory (thin, by default equal to one).

## The BUGS language

### Running JAGS via `rjags`

↪ We thus type

```
update(model,1000,progress.bar="none")
res=coda.samples(model,variable.names=c("mu","sigma2"),n.iter=10000, progress.bar="none")
```

↪ As always in `R`, the most useful overview comes from the `summary` function

```
> summary(res)

Iterations = 12001:22000
Thinning interval = 1
Number of chains = 3
Sample size per chain = 10000

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

        Mean     SD Naive SE Time-series SE
mu     1.926 0.2336 0.001348       0.001336
sigma2 8.215 0.9591 0.005537       0.005640

2. Quantiles for each variable:

        2.5%   25%   50%   75%  97.5%
mu     1.468 1.767 1.926 2.083  2.384
sigma2 6.541 7.541 8.144 8.812 10.283
```

## The BUGS language

### Running JAGS via `rjags`

- ↪ We can also type `plot(res)` to visualise the traceplots and densityplots.

- ↪ For inspection of the autocorrelation function, type `autocorr.plot(res)` and for knowing the effective sample size `effectiveSize(res)`. The Gelman Rubin statistics is obtained typing `gelman.plot(res)` (requires that the number of chains is equal or greater than 2).

- ↪ We can also extract results for each chain and each parameter using the object `res`. For instance, to obtain the values of `mu` of the first chain, we type `res[[1]][,1]`, while for obtaining the `sigma2` values for chain 2, we do `res[[1]][,2]`.

- ↪ We can also collect all the posterior samples from the different chains in one matrix by typing `resmat=as.matrix(res)`.