One of the reasons why I like BUGS and all related dialects has been put nicely in a very good book, i.e. “Introduction to WinBUGS for ecologists” (Kery, 2010); at page 11, the author says: *“WinBUGS helps free the modeler in you”*. Ultimately, that statement is true: when I have fully understood a model with all its components (and thus I have become a modeler), I can very logically translate it to BUGS code. The drawback is that, very often, the final coding appears to be rather ‘problem-specific’ and difficult to be reused in other situations, without an extensive editing work.

For example, consider the ‘ANOVA models’ with all their ‘flavors’: one-way, two-ways with interactions, nested, and so on. These models are rather common in agricultural research and they are relatively easy to code in BUGS, following the suggestions provided in Kery’s book. However, passing from one model to the others requires some editing, which is often prone to errors. And errors in BUGS are difficult to spot in short time… Therefore, I have been wondering: “*Can I write general BUGS code, which can be used for all ANOVA models with no substantial editing?*”.

Finally, I have found a solution and, as it took me awhile to sort things out, I thought I might share it, for the benefit of those who would like to fit ANOVA models in the Bayesian framework. It works with JAGS (JUST ANOTHER GIBBS SAMPLER), a BUGS dialect running also in Mac OS and developed by Marty Plummer. JAGS can be used from R, thanks to the ‘rjags’ package (Plummer, 2019), which I will use in this post.

Let’s start from a working example.

A genotype experiment

The yields of seven wheat genotypes were compared in one experiment laid down in three randomised complete blocks. The data is available in an external repository as a ‘csv’ file and it can be loaded by using the code below.

```
fileName <- "https://www.casaonofri.it/_datasets/WinterWheat2002.csv"
dataset <- read.csv(fileName, header = T)
head(dataset)
##   Plot Block Genotype Yield
## 1   57     A COLOSSEO  4.31
## 2   61     B COLOSSEO  4.73
## 3   11     C COLOSSEO  5.64
## 4   60     A    CRESO  3.99
## 5   10     B    CRESO  4.82
## 6   42     C    CRESO  4.17
```

This is the typical situation were we might be interested in fitting an ANOVA model, with the yield as the response variable and the blocks and genotypes as the explanatory factors. The ultimate aim is to estimate genotype means and credible intervals, which calls for the Bayesian approach.

For the sake of simplicity, let’s take both the block and genotype effects as fixed; in matrix notation, a general linear fixed effects model can be written as:

\[Y = X \, \beta + \varepsilon\]

where \(Y\) is the vector of the observed response, \(\beta\) is the vector of estimated

parameters, X is the design matrix and ε is the vector of residuals, which we assume as gaussian and homoscedastic ($\varepsilon \sim N(0, \sigma^2 I)$; N is the multivariate gaussian distribution). The same model can also be written as:

$$ Y \sim N \left(X \, \beta, \sigma^2 I \right)$$

In JAGS (and maybe also in other BUGS dialects), we can code every linear model using the above specification, as long as we can provide the correct design matrix X. Luckily, we will see that this is a rather simple task; but… let's do it one step at a time!

Specification of a JAGS model

First of all, we open R and code a general linear JAGS model, as shown in the box below.

```
# Coding a JAGS model
modelSpec <- "
data {
n <- length(Y)
np <- dim(X)
nk <- dim(K)
}

model {
# Model
for (i in 1:n) {
   expected[i] <- inprod(X[i,], beta)
   Y[i] ~ dnorm(expected[i], tau)
  }

# Priors
beta[1] ~ dunif(0, 1000000)
for (i in 2:np[2]){
  beta[i] ~ dnorm(0, 0.000001)
  }
sigma ~ dunif(0, 100)

# Derived quantities (model specific)
tau <- 1 / ( sigma * sigma)

# Contrasts of interest
for (i in 1:nk[1]) {
   mu[i] <- inprod(K[i,], beta)
  }
}"
writeLines(modelSpec, con="ModelAOV.txt")
```

Let's see some more detail; you may notice that the code above consists of two fundamental parts, surrounded by curly brackets:

1. a 'data' part
2. a 'model' part

In the data part, we create three variables, i.e. the number of data (n), the number of

estimated parameters (\(np\)) and the number of contrasts (see later). All variables are used in successive model steps and they are obtained, respectively, by counting the number of observations in the vector \(Y\), the number of columns in the design matrix \(X\) and the number of rows in the contrast matrix \(K\).

In the model part we have three further components:

1. the model specification
2. the priors
3. the derived quantities

The model specification contains 'deterministic' and 'stochastic' statements (nodes). The 'deterministic' node returns the expected values for all observations, based on multiplying the design matrix \(X\) by the vector of estimated parameters \(\beta\). In practice, we use a 'for()' loop and, for each i^{th} observation, we sum the products of all element in the i^{th} row of \(X\) by the corresponding elements in the vector of estimated parameters \(\beta\). This sum of products is accomplished by using the function `inprod(X[i,], beta)`.

In the 'stochastic' node we specify that the observed values in \(Y\) are sampled from a gaussian distribution ('dnorm'), with mean equal to the expected value and precision equal to 'tau'. In JAGS, WinBUGS and all related software, the normal distribution is parameterised by using the precision \(\tau = 1/\sigma^2\), instead of standard deviation.

Next, we have to define the priors for all the estimands. For those who are not very much into Bayesian inference, I will only say that priors represent our expectations about model parameters before looking at the data; in this example, we use very vague priors, meaning that, before looking at the data, we have no idea about the values of these unknown quantities. In detail, for the intercept we specify a uniform distribution from 0 to 10000 (`beta[1] ~ dunif(0, 1000000)`), meaning that the overall mean might be included between 0 and 10000 and we have no preference for any values within that range (a very vague prior, indeed). For all other effects in the vector \(\beta\), our prior expectation is that they are normally distributed with a mean of 0 and very low precision (`beta[i] ~ dnorm(0, 0.000001)`). For the residual standard deviation, we expect that it is uniformly distributed from 0 to 100. The selection of priors is central to Bayesian inference and, in other circumstances, you may like to adopt more informative priors. We do not discuss this important item here.

In the end, we also specify some quantities that should be derived from estimated parameters. As we have put a prior on standard deviation, we need to derive the precision (`tau <- 1 / ( sigma * sigma)`), that is necessary for the stochastic node in the specification of our linear model. Afterwards, we add a set of contrasts, which are specified by way of a matrix of contrast coefficients (\(K\); one row per each contrast). This is useful to calculate, e.g., the means of treatment levels or pairwise differences between means as linear combinations of model parameters.

The model definition in the box above is assigned to a text string (`modelSpec`) and it is finally written to an external text file ('modelAOV.txt'), using the function `writeLines()`.

I conclude this part by saying that, based on our model specification, JAGS requires three input ingredients: the \(Y\) vector of responses, the \(X\) matrix and the \(K\) contrast matrix. Furthermore, JAGS requires initial values for all estimands, i.e. for all quantities for which we have specified our prior expectations (the 'beta' vector and the 'sigma' scalar).

Fitting the JAGS model from within R

JAGS models can be fitted from R by using the `rjags` package (Plummer, 2019). However, we have some preliminary steps to accomplish:

1. loading the dataset (see the first box above);
2. creating the \(Y\) vector of responses
3. creating the \(X\) matrix
4. creating the \(K\) matrix

The first two steps are obvious. The third step can be accomplished by using the `model.matrix()` function: the call is very similar to an 'lm()' call, although we do not need to explicitly indicate the response variable (see the box below). In order to create the \(K\) matrix of contrasts, we might prefer to work with the sum-to-zero parameterisation (`options(contrasts=c("contr.sum", "contr.poly"))`), so that the intercept represents the overall mean (for balanced designs) and the effects of blocks and genotypes represent differences with respect to the overall mean. In the box below we specify a set of eight contrasts returning the means for all genotypes.

```
options(contrasts=c("contr.sum", "contr.poly"))
Y <- dataset$Yield
X <- model.matrix( ~ Block + Genotype, data = dataset)

k1 <- c(1, 0, 0, 1, 0, 0, 0, 0, 0, 0)
k2 <- c(1, 0, 0, 0, 1, 0, 0, 0, 0, 0)
k3 <- c(1, 0, 0, 0, 0, 1, 0, 0, 0, 0)
k4 <- c(1, 0, 0, 0, 0, 0, 1, 0, 0, 0)
k5 <- c(1, 0, 0, 0, 0, 0, 0, 1, 0, 0)
k6 <- c(1, 0, 0, 0, 0, 0, 0, 0, 1, 0)
k7 <- c(1, 0, 0, 0, 0, 0, 0, 0, 0, 1)
k8 <- c(1, 0, 0,-1,-1,-1,-1,-1,-1,-1)
K <- rbind(k1, k2, k3, k4, k5, k6, k7, k8)
```

If you need further explanation about the \(X\) and \(K\) matrices and their role in the analysis, I have added an appendix below. Otherwise, we are ready to fit the model. To this aim, we:

1. load the `rjags` library;
2. create two lists: a list of all the data needed for the analysis (`dataList`) and a list of the initial values for the parameters to be estimated (`initList`). Initial values need not be particularly precise;
3. send the model specification and the other data to JAGS, using the function `jags.model()` from the `rjags` package;
4. start the sampler, using the `coda.samples()` function. In this step, we specify which parameters we want to obtain estimates for and the number of samples we want to draw (`n.iter`).
5. obtain the number of required samples, using the `window()` function. In this step, we specify how many samples should be discarded as burn.in. These samples might have been produced before reaching the convergence, so they might not come from the correct posterior distribution and we need to get rid of them.

From the posterior, we obtain the mean and median as measures of central tendency, the standard deviation as a measure of uncertainty and credible intervals, which are the Bayesian analog to confidence intervals. Due to our vague priors, the results are very similar to those obtained with a traditional frequentist analysis (see the appendix below).

```
library(rjags)

# Create lists
dataList <- list(Y = Y, X = X, K = K)
initList <- list(beta = c(4.3, rep(0, 9)), sigma = 0.33)

# Start sampler
mcmc <- jags.model("modelAOV.txt",
                   data = dataList, inits = initList,
                   n.chains = 4, n.adapt = 100)
## Compiling data graph
##    Resolving undeclared variables
##    Allocating nodes
##    Initializing
##    Reading data back into data table
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 24
##    Unobserved stochastic nodes: 11
##    Total graph size: 451
##
## Initializing model
# Get samples
res <- coda.samples(mcmc, variable.names = c("beta", "sigma", "mu"),
                    n.iter = 1000)

out <- summary(window(res, start = 110))
res <- cbind(out$statistics[,1:2], out$quantiles[,c(1,5)])
res
##                  Mean         SD        2.5%       97.5%
## beta[1]   4.42459380 0.07450192  4.27331180  4.56808970
## beta[2]  -0.21830958 0.10451119 -0.42114931 -0.01073556
## beta[3]  -0.01067826 0.10489729 -0.22122939  0.19833336
## beta[4]   0.46390007 0.19802745  0.04693158  0.83344935
## beta[5]  -0.09827277 0.19450502 -0.48923642  0.27830973
## beta[6]  -0.18856790 0.19460922 -0.58070525  0.19744961
## beta[7]  -0.07897598 0.20104587 -0.46988505  0.32557805
## beta[8]   0.54737970 0.19813294  0.15200081  0.93771826
## beta[9]   0.07786131 0.19946009 -0.31222840  0.48406509
## beta[10] -1.09035526 0.19353015 -1.47151174 -0.70790010
## mu[1]     4.88849386 0.21213313  4.44865961  5.29774010
## mu[2]     4.32632102 0.20825090  3.91707525  4.71763694
## mu[3]     4.23602590 0.20766216  3.83381632  4.64905719
## mu[4]     4.34561782 0.21404163  3.93107938  4.77594175
## mu[5]     4.97197349 0.21222639  4.54813702  5.38971184
## mu[6]     4.50245511 0.21323351  4.09000675  4.93210416
## mu[7]     3.33423854 0.20600397  2.92628945  3.73812600
## mu[8]     4.79162462 0.21207498  4.36009497  5.21086914
## sigma     0.35632989 0.08036709  0.24276414  0.55078418
```

Reusing the code for a multi-environment experiment

The JAGS model above is very general and can be easily reused for other situations. For example, if the above genotype experiment is replicated across years, we might like to fit an ANOVA model by considering the blocks (within years), the genotypes, the years and the 'year by genotype' interaction. The dataset is available in the same external repository, as the 'WinterWheat.csv' file.

The JAGS specification for this multienvironment model does not change, we only need to update the \(Y\), \(X\) and \(K\) matrices, as shown in the box below. In order to obtain the contrast matrix for the means of the 'genotype x environment' combinations we need to write some cumbersome code, as shown below (but, perhaps, some of you could suggest better alternatives…).

```
library(tidyverse)

# Loading the data
fileName <- "https://www.casaonofri.it/_datasets/WinterWheat.csv"
dataset <- read_csv(fileName)
dataset <- dataset %>%
  mutate(across(c(Block, Year, Genotype), .fns = factor))
dataset
## # A tibble: 168 x 5
##     Plot Block Genotype Yield Year
##
##  1     2 1     COLOSSEO  6.73 1996
##  2   110 2     COLOSSEO  6.96 1996
##  3   181 3     COLOSSEO  5.35 1996
##  4     2 1     COLOSSEO  6.26 1997
##  5   110 2     COLOSSEO  7.01 1997
##  6   181 3     COLOSSEO  6.11 1997
##  7    17 1     COLOSSEO  6.75 1998
##  8   110 2     COLOSSEO  6.82 1998
##  9   256 3     COLOSSEO  6.52 1998
## 10    18 1     COLOSSEO  7.18 1999
## # … with 158 more rows
# Create input matrices
Y <- dataset$Yield
X <- model.matrix(~ Genotype*Year +  Block:Year, data = dataset)

# Workaround to get K matrix
asgn <- attr(X, "assign")
tmp1 <- expand.grid(Genotype = unique(dataset$Genotype),
                    Year = unique(dataset$Year))
K1 <- model.matrix(~ Genotype*Year, data = tmp1)
K2 <- matrix(0, nrow = nrow(K1), ncol = length(asgn[asgn==4]))
colnames(K2) <- colnames(X)[asgn==4]
K <- cbind(K1, K2)
row.names(K) <- with(tmp1, interaction(Genotype:Year))
# K
```

```
# Create lists
dataList <- list(Y = Y, X = X, K = K)
initList <- list(beta = c(4.3, rep(0, length(X[1,])-1)), sigma = 0.33)

# Start sampler
mcmc <- jags.model("modelAOV.txt",
                   data = dataList, inits = initList,
                   n.chains = 4, n.adapt = 100)
## Compiling data graph
##    Resolving undeclared variables
##    Allocating nodes
##    Initializing
##    Reading data back into data table
## Compiling model graph
##    Resolving undeclared variables
##    Allocating nodes
## Graph information:
##    Observed stochastic nodes: 168
##    Unobserved stochastic nodes: 71
##    Total graph size: 16519
##
## Initializing model
# Get samples
res <- coda.samples(mcmc, variable.names = c("beta", "sigma", "mu"),
                    n.iter = 1000)

out <- summary(window(res, start = 110))
res <- cbind(out$statistics[,1:2], '50%'=out$quantiles[,3],
             out$quantiles[,c(1, 5)])
head(res)
##               Mean         SD        50%         2.5%      97.5%
## beta[1]  6.2677640 0.03043537  6.2673982  6.208390004  6.3300080
## beta[2]  0.1464792 0.07854643  0.1461239 -0.009216115  0.2979934
## beta[3] -0.2947195 0.07764813 -0.2955634 -0.446178189 -0.1401161
## beta[4]  0.3248028 0.07746617  0.3246501  0.174748219  0.4733587
## beta[5] -0.1843623 0.08221827 -0.1854063 -0.349242266 -0.0245657
## beta[6]  0.4269969 0.07989058  0.4260559  0.272364897  0.5842687
#....
tail(res)
##             Mean         SD       50%      2.5%    97.5%
## mu[52] 4.3419716 0.22876493 4.3422616 3.8980334 4.792529
## mu[53] 4.9626900 0.22571023 4.9634626 4.5132893 5.392722
## mu[54] 4.5076303 0.22951179 4.5094695 4.0518930 4.955052
## mu[55] 3.3378495 0.22618643 3.3342110 2.9010714 3.769008
## mu[56] 4.7907886 0.22765164 4.7854894 4.3385675 5.241458
## sigma  0.3915724 0.02904505 0.3893148 0.3428075 0.454776
```

The discovery of the `inprod()` function was a very big hit for me: the above approach is very flexible and lend itself to a lot of potential uses, including fitting mixed models. I will show some examples in future posts.