glm to MCMCglmm

Kevin Healy

14 July 2016

This is a short example of linear modeling starting with a glm and working up towards a MCMCglmm. However, we will mainly focus on the basics needed to run and check MCMCglmm models today with some extra more complex models including adding random effects. Tomorrow we will use this to do some more complex models, such as correcting for phylogeny and imputing missing data.

Installation

First we need to install some packages including the lme4 to run mixed models using a frequentist approach and MCMCglmm to run a Bayesian approach to mixed models.

```r
if(!require(MCMCglmm)) install.packages("MCMCglmm")
if(!require(lme4)) install.packages("lme4")
```

We will also install from GitHub the `MulTree` package which is still under development (so watch out for BUGS) but contains some handy data and also will allow us to use MCMCglmm to include the error associated with building phylogenies in tomorrow's session. To do so we need to get them from GitHub and so we need to run.

```r
if(!require(devtools)) install.packages("devtools")
library(devtools)
install_github("TGuillerme/mulTree", ref = "master")
```

If you have not heard of GiTHub it is a great way to share code, build packages and use version control to back up your work. For more check out here

Next we load up the packages we just installed from the library and we are good to go.

```r
library(MCMCglmm)
library(lme4)
library(mulTree)
```

Data

For the duration of the tutorials we will use some data that is part of a `MulTree` package. To get it just run

```r
data(lifespan)
```

This data file contains a subset of the data used in an analysis on the role of flying (volant) in the evolution of maximum lifespan in birds and mammals Link to paper. Note that these data have been log transformed, mean centered and expressed in units of standard deviation. The original lifespan data were taken from the Anage database. We will come back to why it is often useful to transform data into z-scores later but for now we will simply assume our data is well behaved. Lets have a look at it now.

```
#data have been log transformed, mean centered and
#expressed in units of standard deviation.
head(lifespan_volant)
```

```
##                species    class  longevity       mass    volant
## 1 Dolichotis_patagonum Mammalia -0.1490041  1.0875446 nonvolant
## 2       Eidolon_helvum Mammalia  0.4686111 -0.2748337    volant
## 3      Elephas_maximus Mammalia  2.1071286  3.1220340 nonvolant
## 4         Equus_asinus Mammalia  1.6128024  2.0352764 nonvolant
## 5     Equus_burchellii Mammalia  1.2962194  2.2295299 nonvolant
## 6       Equus_caballus Mammalia  1.9001076  2.2548716 nonvolant
```

Lets run some models

Let's first start off running a simple glm for a subset of data for mammals

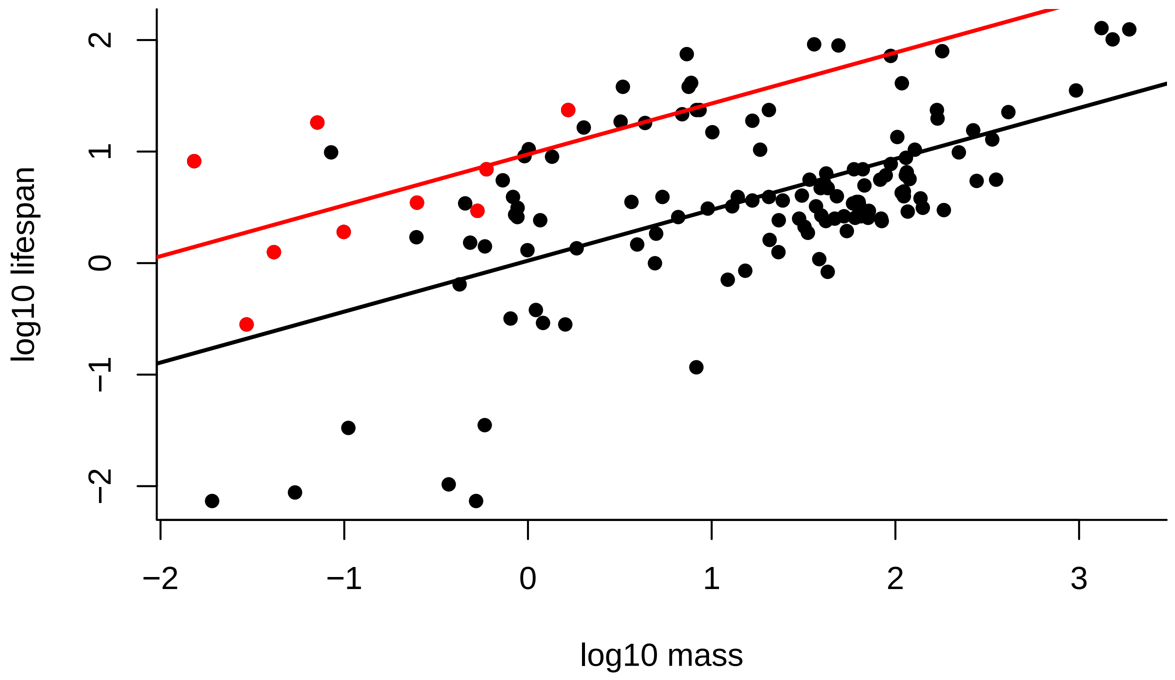
```
#subset for mammals
lifespan_mammals <- lifespan_volant[lifespan_volant$class == "Mammalia",]

#### and run a simple glm
glm_mod <- glm(formula = longevity ~ mass + volant, family = "gaussian", data = lifespan_mammals)
summary(glm_mod)
```

```
##
## Call:
## glm(formula = longevity ~ mass + volant, family = "gaussian",
##     data = lifespan_mammals)
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -2.02569  -0.38641  -0.07613   0.41818   1.46075
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.02144    0.09241   0.232 0.816885
## mass          0.45655    0.05844   7.812  1.6e-12 ***
## volantvolant  0.95325    0.25568   3.728 0.000286 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.423457)
##
##     Null deviance: 81.317  on 133  degrees of freedom
## Residual deviance: 55.473  on 131  degrees of freedom
## AIC: 270.09
##
## Number of Fisher Scoring iterations: 2
```

We can plot the results

```r
#simple plots
plot(longevity ~ mass, data = lifespan_mammals, pch = 16, bty = "l",
                       xlab = "log10 mass", ylab = "log10 lifespan")
#add in the volant species as red
points(lifespan_mammals[lifespan_mammals$volant == "volant","longevity"] ~
        lifespan_mammals[lifespan_mammals$volant == "volant","mass"],
        col = "red", pch = 16)
#add in the nonvolant regression line
abline(glm_mod, lwd = 2)
#add in the volant regression line
abline(glm_mod$coefficients[1] + glm_mod$coefficients[3],
       glm_mod$coefficients[2], lwd = 2, col = "red")
```



Most people will be familiar with lm and glm models so we won't spend any more time here. One thing that we might do is account for some of the structure in the error term. To do so we would need to include a random term.

glmm

While our linear model looks good we might also want to try to control for the structure of our data. For example, maybe all the data points are not fully independent with some data points more likely to have values closer to other ones. In this case we might want to add a random term to control for this. We can imagine such a case in our data above were two species from the same genus might show more similar values of lifespan in comparison to other species.

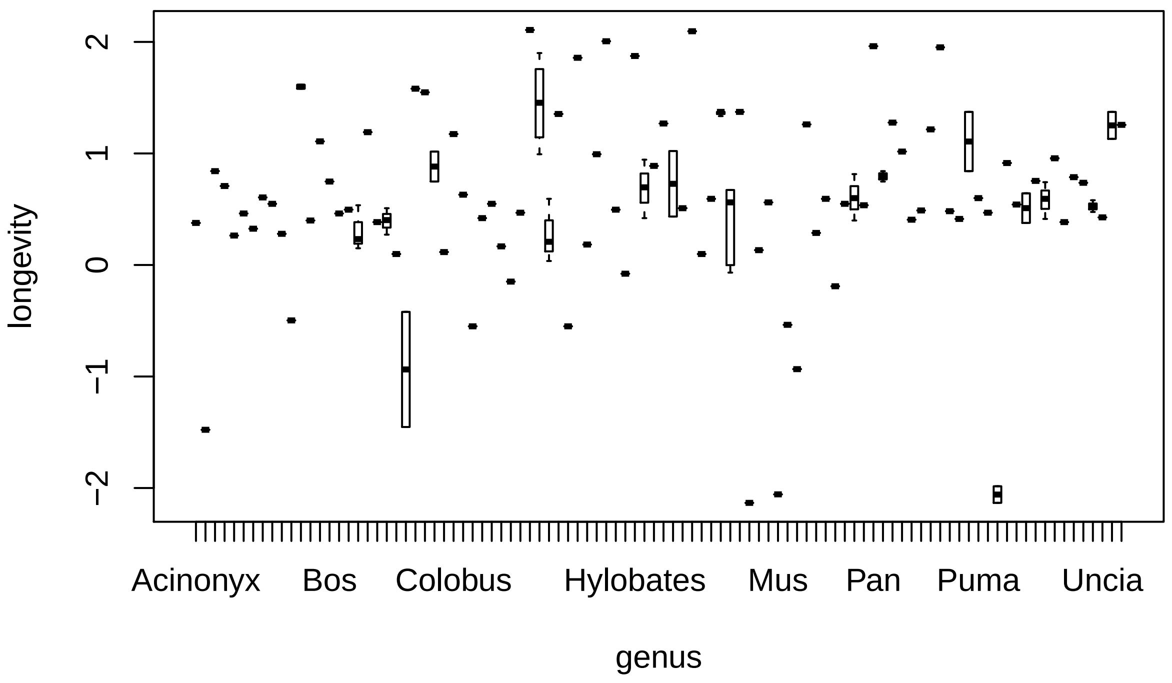
Let's plot it out and have a look

```r
#use the gsub function to create a vector of the genus for each species
genus <- (gsub("_.*","", lifespan_mammals$species))

#bind it back into the database
```

```r
lifespan_mammals <- data.frame(lifespan_mammals, genus = genus)

#plot out lifespan ~ genus to get an idea of whether genus is structured
#randomly
plot(longevity ~  genus, data = lifespan_mammals)
```



This looks to me like genus is something I would like to control for, however I am not really interested in which groups are different and for practical reasons I don't want to fit every single group as a fixed factor. Hence I could include it as a random term using a lmer model which is used to fit mixed models using a maximum likelihood approach.

```r
#Lets fit our model. For lmer the random term is fitted using (1|genus)
#to indicate that you want to fit seperate intercepts
#to each of group in your random term.
lmer_mod  <-  lmer(longevity ~ mass + volant  + (1|genus), data = lifespan_mammals)
summary(lmer_mod)
```

```
## Linear mixed model fit by REML ['lmerMod']
## Formula: longevity ~ mass + volant + (1 | genus)
##    Data: lifespan_mammals
##
## REML criterion at convergence: 220.9
##
## Scaled residuals:
##     Min      1Q  Median      3Q     Max
## -2.0997 -0.2667 -0.0240  0.3312  1.8262
##
## Random effects:
##  Groups   Name        Variance Std.Dev.
##  genus    (Intercept) 0.37772  0.6146
##  Residual             0.05953  0.2440
## Number of obs: 134, groups:  genus, 98
##
## Fixed effects:
##               Estimate Std. Error t value
## (Intercept)  -0.01585    0.10366  -0.153
## mass          0.48637    0.06365   7.641
## volantvolant  1.00386    0.27950   3.592
##
## Correlation of Fixed Effects:
##             (Intr) mass
## mass        -0.746
## volantvolnt -0.534  0.496
```

Like before we see estimates for each of the fixed effects which look similar to our glm model. The next section that we are interested in is the random effects were we see the terms genus and Residual. Our residual term is acting as in the glm model telling how much of the variance is unexplained while the genus term tells us how much of the variance is due to variance associated with the genus groupings. We can see here for example that genus accounts for more variation than our random term after accounting for our fixed effects.

I will leave lmer models here, however if you are interested in more on mixed effects models using this approach check out `this tutorial`. For now lets move on to the main event with MCMCglmm.

MCMCglmm

So far we have fitted a very simple glm and a glmm model, now we will fit a linear model and a mixed model using the MCMCglmm package. We will start first with a model similar to our glm.

First things first, since we are using a Bayesian approach we will need to set up the priors. In most cases we want to use a non-informative prior that doesn't influence the estimated posterior distribution. We are basically saying that we don't know anything about the expected values for our parameters. That is we have no prior information.

To give priors for MCMCglmm we need to make an object that is in a list format that includes terms of B (fixed effect), R (residual terms) and G (random terms which we will come to later).

For now let's build a prior with just a fixed term and a residual term.

```r
prior <- list(B = list(mu= diag(3)*0, V=diag(3)*1e+10),
            R = list(nu=0.002, V=1))
```

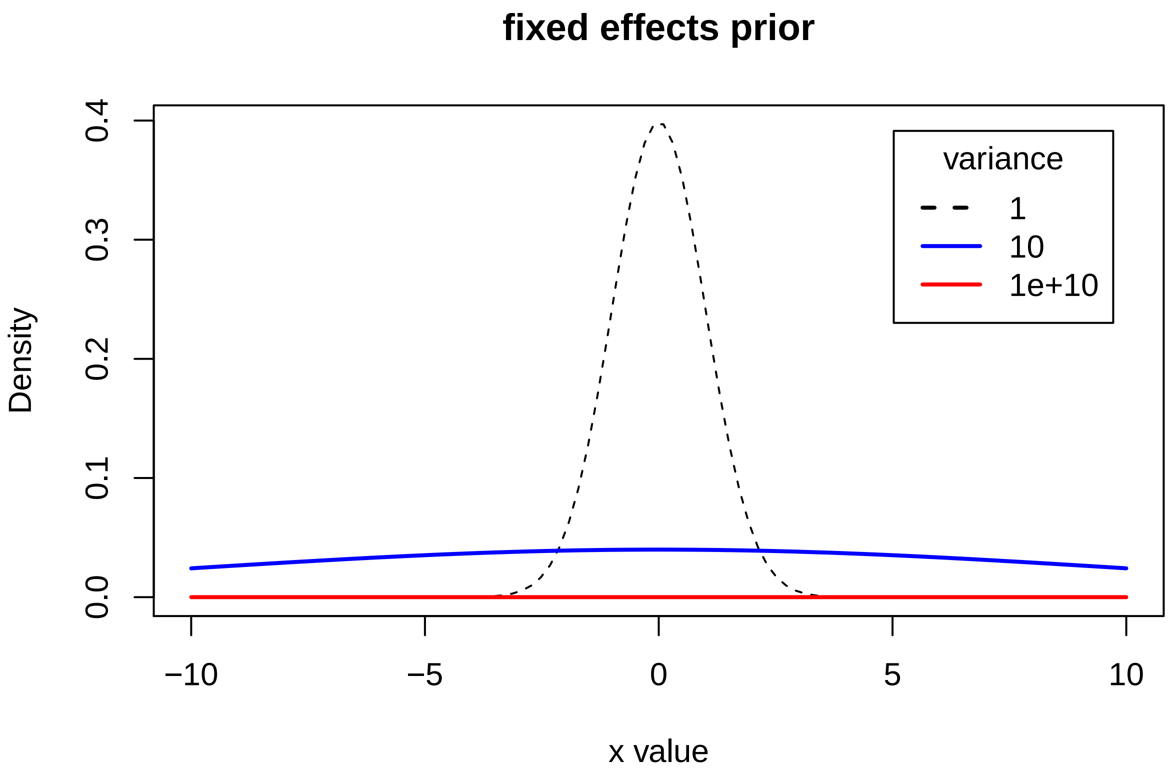
For fixed effects (B) the terms V and mu give the variance and mean of a normal distribution. Here we set mu as 0 and the variance as a large number to make these priors uninformative. Since we have three fixed terms (two intercepts and one slope) we can use the diag function to create a matrix to store a prior for each. Normally we don't need to set this as MCMCglmm will set non-informative priors automatically for fixed terms.

If we plot it out, we can see that higher values for V give less informative priors.

```r
#create some normal distributions over some range x
x <- seq(-10, 10, length=100)
#V = 1
hx_1 <- dnorm(x, 0,1 )
#V = 10
hx_10 <- dnorm(x, 0,10 )
#V = 1e+10
```

```r
hx_1e10 <- dnorm(x, 0,1e+10 )
plot(x, hx_1, type="l", lty=2, xlab="x value",
     ylab="Density", main="fixed effects prior")

lines(x, hx_10, lwd=2, col="blue")
lines(x, hx_1e10, lwd=2, col="red")
labels <- c("1","10","1e+10")
legend("topright", inset=.05, title="variance",labels,
       lwd=2, lty=c(2, 1, 1), col=c("black", "blue", "red"))
```



For the variance terms we need to make sure that the distribution is bounded at zero as the variance term needs to be positive. To do this MCMCglmm uses an inverse-Gamma distribution. In MCMCglmm this is described by two parameters `nu` and `V`. These terms are related to the shape (alpha) and scale (beta) parameters on an inverse-Gamma with `alpha = nu/2`, and `Beta = (nu*V)/2`. As we don't want our estimates to be heavily influenced by our prior we will use weakly informative prior values such as descripted as `V = 1` and `nu = 0.002`. (For more on priors the see course notes)

Next we need to decide on the parameters relating to running the mcmc chain in the model. We need to include how many iterations we want to run the chain for (nitt), the burnin we want to discard at the start of the chain (burnin) and also how often we want to sample and store from the chain (`thin`). We discard a burnin as we don't want the starting point of the chain to over-influence our final estimates. For now let's just use a burnin of 1/6 of the `nitt`, just to be safe. The thinning is used to help reduce autocorrelation in our sample, how much you use often depends on how much autocorrelation you find.

To save time we will only run this model over 12000 iterations (However, much larger `nitt` is often required).

```r
#no. of interations
nitt <- c(12000)
#length of burnin
burnin <- c(2000)
```

```r
#amount of thinning
thin <- c(5)
```

Now we can run the model using our data. Let's run a model similar to our first glm

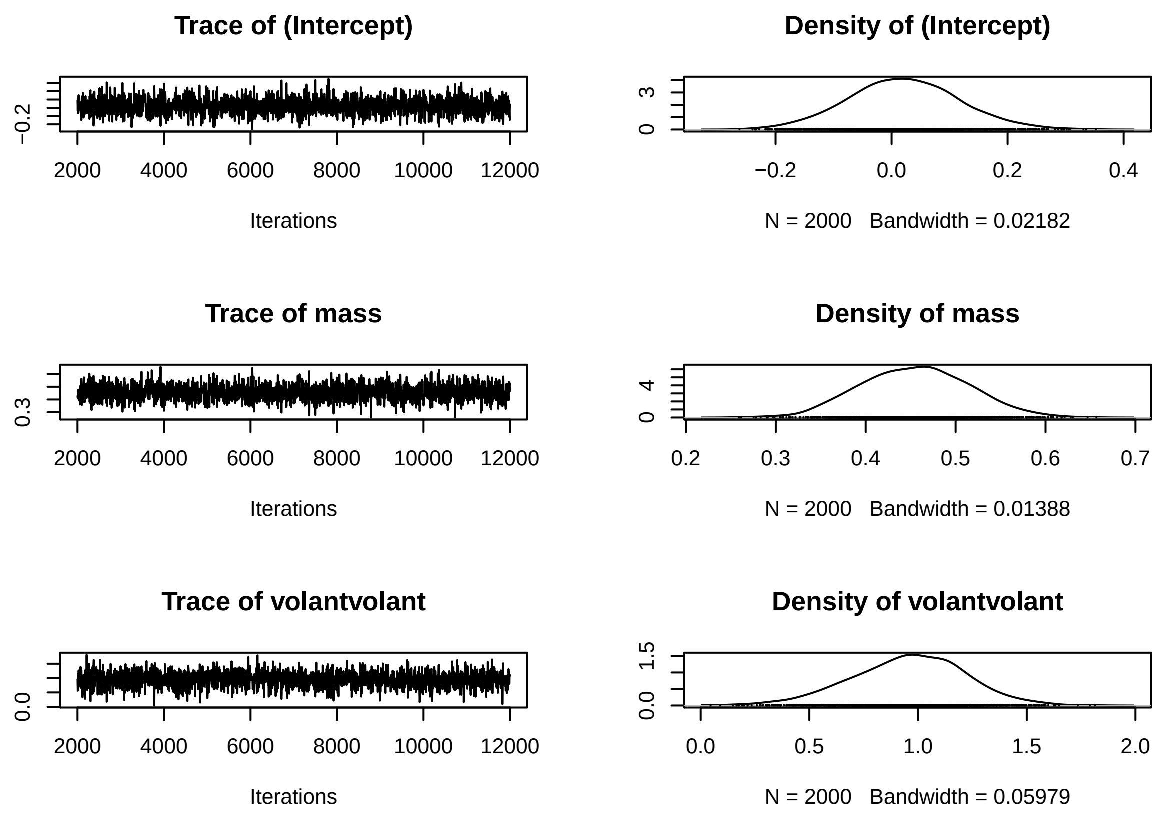
```r
mod_mcmc_fix <- MCMCglmm(fixed =  longevity ~ mass + volant,
                    family="gaussian",
                    data = lifespan_mammals,
                    nitt = nitt,
                    burnin = burnin,
                    thin = thin,
                    prior = prior)
```

```
##
##                        MCMC iteration = 0
##
##                        MCMC iteration = 1000
##
##                        MCMC iteration = 2000
##
##                        MCMC iteration = 3000
##
##                        MCMC iteration = 4000
##
##                        MCMC iteration = 5000
##
##                        MCMC iteration = 6000
##
##                        MCMC iteration = 7000
##
##                        MCMC iteration = 8000
##
##                        MCMC iteration = 9000
##
##                        MCMC iteration = 10000
##
##                        MCMC iteration = 11000
##
##                        MCMC iteration = 12000
```

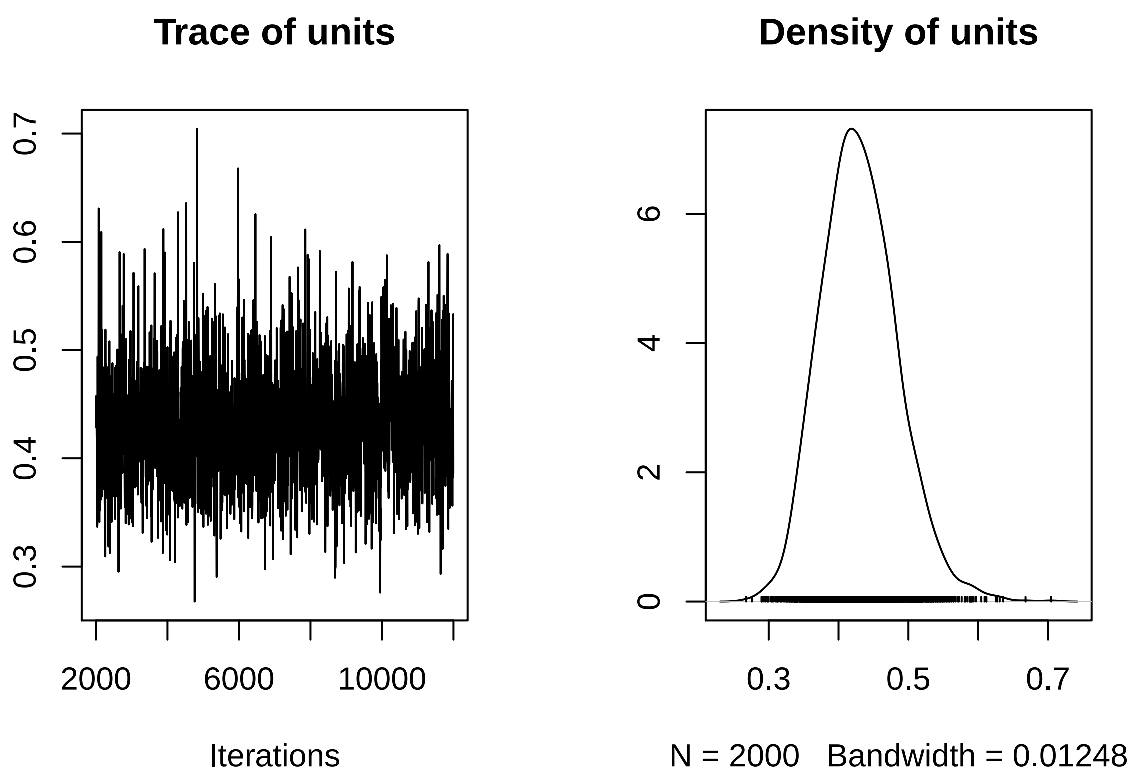
As the model runs we see the iterations print out. These chains can take some time to run, depending on the model, however, since we only ran our chains for 12000 iterations it doesn't take long here.

Before we even look at our model we need to check if the model ran appropriately. We can do this by visually inspecting the chains to make sure there has been no unruly behavior! We can extract the full chains using `model$Sol` for the fixed effects and `model$VCV` for the variance terms. So `Sol[,1]` will give you the first fixed term, in this case the intercept, and `VCV[,1]` will give you the first random term, which is just the residual term here. As our model is an mcmc object when we use the plot function we get a trace plot.

```r
#plot the fist fixed term, the intercpet.
plot(mod_mcmc_fix$Sol)
```

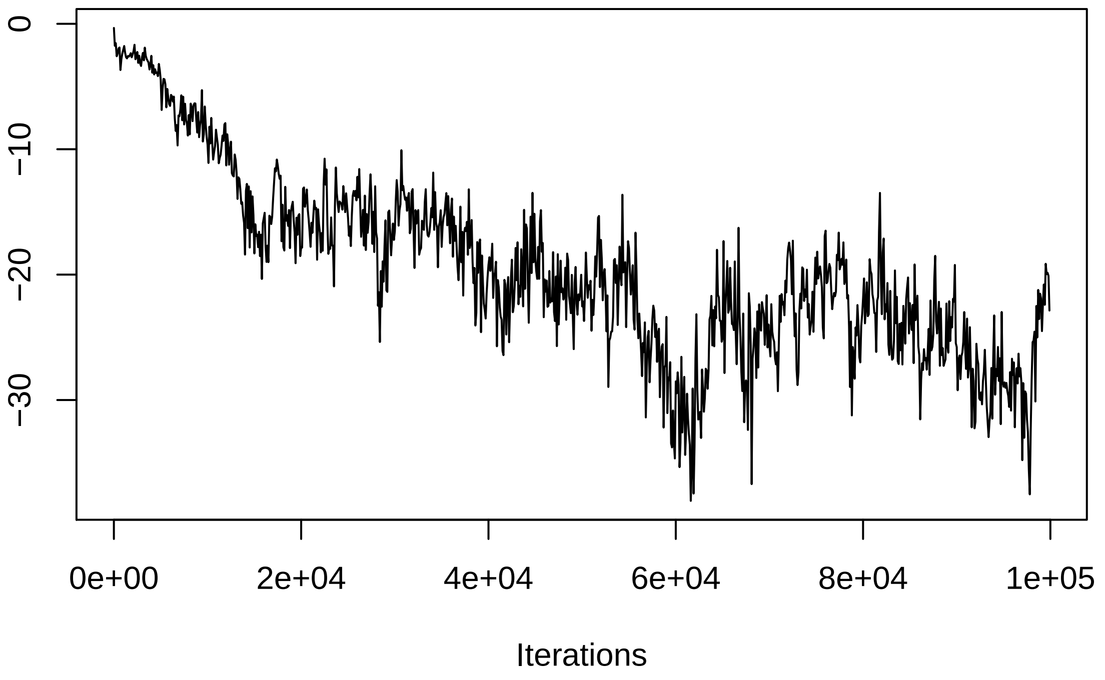


```
#plot the fist variance term, the residual error term.
plot(mod_mcmc_fix$VCV)
```



On the right hand side of the plots is the posterior distributions for each of the terms. On the left side of these plots are the traces of the mcmc chain for each estimate. What we want to see in these trace plots is "hairy caterpillars" (not my phrase!). That is a trace with no obvious trend that is bouncing around some stable point.

What we don't want to see in the trace plots can be demonstrated if we only run a model over a very short chain (itt == 10000) or more difficult model fit (we will see these more difficult models later). Notice in the example below that without a burnin the start of trace is well outside the area that the chain is converging towards.



So far in our simple model everything looks good visually, however we also want to check the level of auto-correlation in these traces. We can do this using autocorr.diag() which gives the level of correlation along the chain between some lag sizes.

```
autocorr.diag(mod_mcmc_fix$Sol)
```

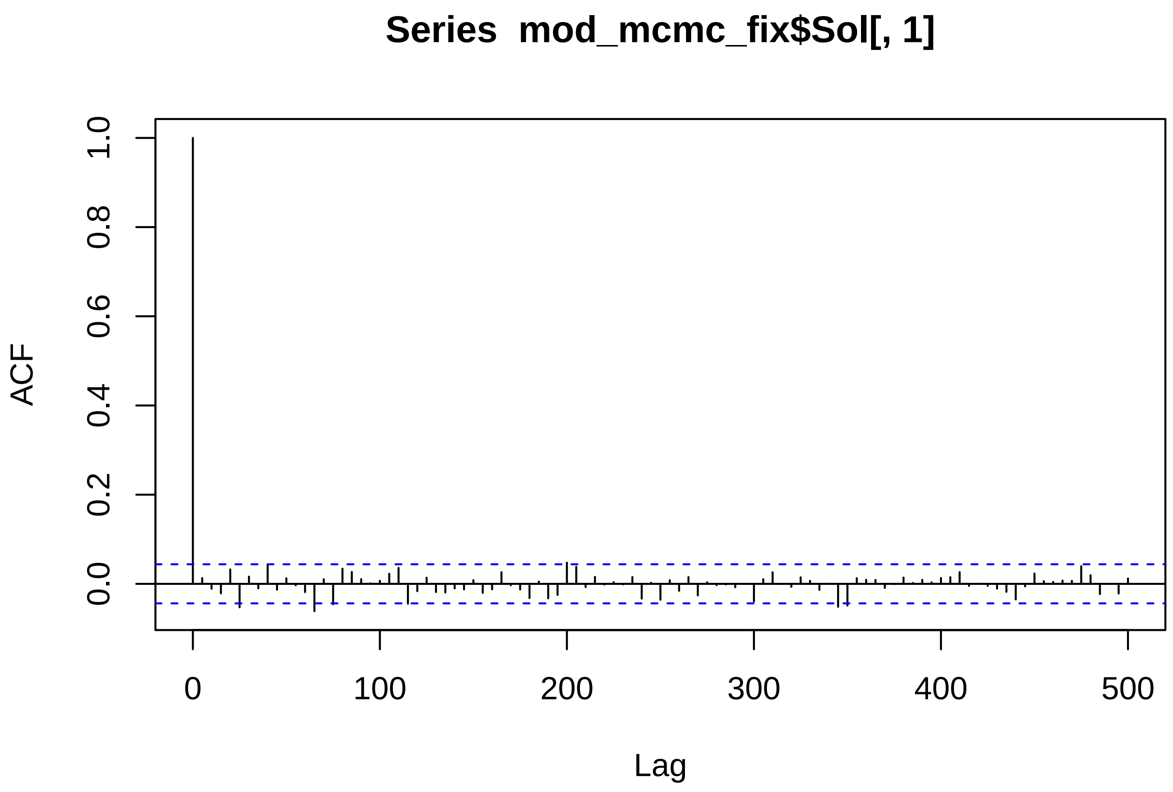
```
##         (Intercept)         mass volantvolant
## Lag 0    1.00000000  1.000000000   1.00000000
## Lag 5    0.01286601 -0.009051209   0.03570829
## Lag 25  -0.05263699 -0.019151706  -0.03097609
## Lag 50   0.01246101 -0.005556991  -0.03231956
## Lag 250 -0.03570952 -0.039018990   0.02571834
```

```
autocorr.diag(mod_mcmc_fix$VCV)
```

```
##                units
## Lag 0    1.000000000
## Lag 5   -0.007042144
## Lag 25   0.017343293
## Lag 50   0.020467950
## Lag 250 -0.007250844
```

or we can look at autocorrelation plots for each of the traces. For example, let's check the auto-correlation in the intercept chain using the `acf` function

```r
#acf plot for the first fixed estimate in our model (the intercept)
acf(mod_mcmc_fix$Sol[,1], lag.max =100)
```



For our intercept the auto-correlation plot looks good. However if there is bad auto-correlation one quick way to deal with this is to simply increase the thinning. While we don't need to in our case an example would be running something like

```r
nitt2 <- 240000
burnin2 = 40000
thin2 = 100
mod_mcmc_long <- MCMCglmm(fixed = longevity ~ mass + volant,
                    family="gaussian",
                    data = lifespan_mammals,
                    nitt = nitt2,
                    burnin = burnin2,
                    thin = thin2,
                    prior = prior,
                    verbose=FALSE)
```

Noticed I also increased the number of iterations. One rough and ready rule that I like to use is to aim for an effective sample size of somewhere between 1000-2000 for all my estimates. The effective sample size is the number of samples in the posterior after the burnin, thinning and autocorrelation are accounted for.

```r
#acf plot for the first fixed estimate in our model (the intercept)
effectiveSize(mod_mcmc_long$Sol)
```

```
##   (Intercept)         mass volantvolant
##          2000         2000         2000
```

```
effectiveSize(mod_mcmc_long$VCV)
```

```
## units
##  2000
```

*One thing to note is that while thinning might help autocorrelation it wont solve it and you might have to use parameter expanded priors. These are priors that help weight the chain away from zero, a common problem when variance is low or with certain phylogenetic structures. They work by splitting the prior into 2 components with one component weighing the chain away from zero. We will come back to such a prior shortly.

One last thing to check is that our MCMC chain has properly converged and that our estimate is not the result of some type of transitional behaviour. That is have our chains "found" the optimum or do we need to let them run longer before they settle around some estimate. To check this we will run a second model and see if it converges on the same estimates as our first model.

```
mod_mcmc_2 <- MCMCglmm(fixed = longevity ~ mass + volant,
                     family="gaussian",
                     data = lifespan_mammals,
                     nitt = nitt2,
                     burnin = burnin2,
                     thin = thin2,
                     prior = prior,
                     verbose=FALSE)
```

We can now check the convergence of the two chains using the Gelman and Rubin Multiple Sequence Diagnostic. This calculates the within-chain and between-chain variance of the chains and then gives a scale reduced factor, (for more see here. When this number is close to one (something below 1.1 is usually good) the chains are indistinguishable and hence can be considered to be converged.

```
#checking convergence for our fixed factors
gelman.diag(mcmc.list(mod_mcmc_long$Sol, mod_mcmc_2$Sol))
```

```
## Potential scale reduction factors:
##
##              Point est. Upper C.I.
## (Intercept)           1          1
## mass                  1          1
## volantvolant          1          1
##
## Multivariate psrf
##
## 1
```

```
#checking convergence for our random terms
gelman.diag(mcmc.list(mod_mcmc_long$VCV, mod_mcmc_2$VCV))
```

```
## Potential scale reduction factors:
##
##       Point est. Upper C.I.
## units          1          1
```

Since everything looks good, we will finally look at the results of our model.

```r
summary(mod_mcmc_long)
```

```
##
##  Iterations = 40001:239901
##  Thinning interval  = 100
##  Sample size  = 2000
##
##  DIC: 270.1105
##
##  R-structure:  ~units
##
##       post.mean l-95% CI u-95% CI eff.samp
## units    0.4298   0.3345   0.5417     2000
##
##  Location effects: longevity ~ mass + volant
##
##              post.mean l-95% CI u-95% CI eff.samp  pMCMC
## (Intercept)    0.02307 -0.14891  0.20181     2000  0.782
## mass           0.45641  0.34238  0.57652     2000 <5e-04 ***
## volantvolant   0.95424  0.43435  1.44529     2000 <5e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

First off we can find the estimates for the fixed factors are under the Location effects section (Notice the similarity to our glm and nlme model). Each parameter has a measure of the effect size under post.mean and a lower and higher 95% credible interval (CI). These are simply calculated from the posterior distributions we looked at in the above plots, so if you would rather calculated the median instead of using the mean we can simple use

```r
median(mod_mcmc_long$Sol[,1])
```

```
## [1] 0.02407152
```

We also have the effective sample size (`eff.samp`) and the `pMCMC` which calculated as two times the probability that the estimate is either > or < 0, using which ever one is smaller. However, since our data has been mean centred and expressed in units of standard deviation we can simply look at what proportion of our posterior is on either side of zero. This mean centering and expression in of our data units of standard deviation hence allows us to use a cut off point like a p-value but without boiling down the whole distribution to one value.

We also have the DIC which is a Bayesian version of AIC. Like AIC it is a measure of the trade-off between the "fit" of the model and the number of parameters, with a lower number better.

Putting the m in MCMCglmm

Since we can run a simple linear MCMCglmm lets add a random term of genus in the example above. Like before we need to set up the prior however we will let the model estimte the fixed effects this time. To add a random term we now add a `G structure` that acts just like the other random varience term and is defined using `nu` and `V`.

```r
prior <- list(G = list(G1 = list(nu=0.002, V=1)),
              R = list(nu=0.002, V=1))
```

We can now include the random term in the model in the section `random= ~`.

```r
nitt_m <- 1000
burnin_m <- 1
thin_m <- 1

mod_mcmc_mixed <- MCMCglmm(fixed = longevity ~ mass + volant,
                           rcov=~ units,
                           random= ~ genus,
                           family="gaussian",
                           data = lifespan_mammals,
                           nitt = nitt_m,
                           burnin = burnin_m,
                           thin = thin_m,
                           prior = prior,
                           verbose=FALSE)
summary(mod_mcmc_mixed)
```

```
##
##  Iterations = 2:1000
##  Thinning interval  = 1
##  Sample size  = 999
##
##  DIC: 88.87016
##
##  G-structure:  ~genus
##
##       post.mean l-95% CI u-95% CI eff.samp
## genus     0.377   0.2608   0.5141    443.7
##
##  R-structure:  ~units
##
##       post.mean l-95% CI u-95% CI eff.samp
## units   0.06322  0.03875  0.09445      142
##
##  Location effects: longevity ~ mass + volant
##
##              post.mean l-95% CI u-95% CI eff.samp  pMCMC
## (Intercept)   -0.01742 -0.21179  0.18344      999  0.879
## mass           0.48769  0.35582  0.60426      999 <0.001 **
## volantvolant   1.00371  0.46826  1.54143      999  0.002 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

As an exercise check the output of this model.

Once you are happy with the above model you have essentially got the basics of running a Bayesian mixed effects model. Building on the complexity involves adding more terms which are covered in good detail in the course notes and vignette.

If we have time I will just show a couple of extra bits below.

Bonus materials

Non-normal response variables

Running a MCMCglmm seems like a lot of work to get the same result so why bother. One reason is that you are fundamentally team Bayesian. Another is that MCMCglmm can give you a lot of flexibility in your models. For example, we can run models with different types of response variables such as categorical by simply changing the family input to "categorical".

So if we wanted to run a model with volant as a response variable we could run something like

```
mod_mcmc_cat <- MCMCglmm(fixed = volant ~ mass,
                         rcov=~ units,
                         random= ~ genus,
                         family="categorical",
                         data = lifespan_mammals,
                         nitt = nitt2,
                         burnin = burnin2,
                         thin = thin2,
                         prior = prior,
                         verbose=FALSE)
summary(mod_mcmc_cat)
```

```
##
##  Iterations = 40001:239901
##  Thinning interval  = 100
##  Sample size  = 2000
##
##  DIC: 0.2200763
##
##  G-structure:  ~genus
##
##       post.mean l-95% CI u-95% CI eff.samp
## genus    282670 0.004191   531211    20.79
##
##  R-structure:  ~units
##
##       post.mean l-95% CI u-95% CI eff.samp
## units     12293  0.05405    76096    201.6
##
##  Location effects: volant ~ mass
##
##             post.mean l-95% CI u-95% CI eff.samp  pMCMC
## (Intercept)    -801.2  -1231.5   -376.5    8.281 <5e-04 ***
## mass           -627.1  -1023.1   -327.3    4.747 <5e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

*Unfortunately this model runs terrible and it may converge eventually but it may need a long time. However, this does still demonstrate the ability to run such a model.

This is particularly useful as random effects can also easily be incorporated in such models unlike frequentist approaches that require more complex maths to solve over more complex likelihood spaces where Bayesian approaches can "find" such estimates using computational power to search such complex spaces. To get a full list of the families `MCMCglmm` can deal with check out `??MCMCglmm`

Multiple response in a MCMCglmm

Another advantage of using MCMCglmm is that you can build multiple response models. These might be useful when you are interested in the effects on two variables simultaneously hence avoiding the need to treat one as an explanatory variable. As before we need to set some priors.

```
prior_mul<-list(R = list(V = diag(2)/2, nu=0.002), G = list(G1=list(V = diag(2)/2,nu = 0.002)))
```

Since we want to fit separate parameters for each response variable (i.e we want an slope/intercept for both longevity and volancy) we use the diag function again.

Now we are set to run a multiple response model. We need to use the `us()` and `idh()` aspects of the model. The notation `us()` specifies parameters for all variances and co-variances, giving the same random-effects parameters that lme4 would give us by default. The idh() tells MCMCglmm to estimate parameters for just the variances for the random term but not the co-variances.

```
mod_mul_r <- MCMCglmm(cbind(longevity,volant) ~ trait:mass,
                     rcov=~us(trait):units,
                     random= ~ idh(trait):genus,
                     family= c("gaussian","catagorical"),
                     data = lifespan_mammals,
                     nitt = nitt2,
                     burnin = burnin2,
                     thin = thin2,
                     prior = prior_mul,
                     verbose=FALSE)
summary(mod_mul_r)
```

```
##
##  Iterations = 40001:239901
##  Thinning interval  = 100
##  Sample size  = 2000
##
##  DIC: -623.4387
##
##  G-structure:  ~idh(trait):genus
##
##                     post.mean l-95% CI u-95% CI eff.samp
## traitlongevity.genus    0.4301   0.3004   0.5779 2000.000
## traitvolant.2.genus   165.9941  29.1879 484.1854    7.284
##
##  R-structure:  ~us(trait):units
##
##                                    post.mean   l-95% CI u-95% CI eff.samp
## traitlongevity:traitlongevity.units   0.06683  0.0383074   0.1028  2019.92
## traitvolant.2:traitlongevity.units    0.24161 -0.3665782   1.1642    42.78
## traitlongevity:traitvolant.2.units    0.24161 -0.3665782   1.1642    42.78
## traitvolant.2:traitvolant.2.units     3.59285  0.0005674  18.9359    67.75
##
##  Location effects: cbind(longevity, volant) ~ trait:mass
##
##                     post.mean  l-95% CI  u-95% CI eff.samp  pMCMC
## (Intercept)           0.17114  -0.01725   0.35247 1566.197  0.078 .
## traitlongevity:mass   0.37746   0.26205   0.49015 1098.428 <5e-04 ***
## traitvolant.2:mass  -12.80205 -22.52952  -8.04155    4.281 <5e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Like before we get our fixed effects now under the Location effects section, however now we get an estimate for both mass against both longevity and volancy (again the results of this model have not converged at all so effect sizes are not to be trusted).

Parameter expanded priors

I noted above that one problem that may occur is a difficulty getting rid of auto-correlation. One reason this may happen is that when values for variance terms are close to zero chains can become "stuck" affecting the mixing of the model. This would be seen as trace plots that show poor mixing, and don't look like hairy caterpillars. One potential solution is to use parameter expanded priors. These work by splitting the prior into two components, one part of the prior deals with values very close to zero and the other part of the prior deals with all other values. This split allows the chain to be bounced out of very low values and help with the chain mixing.

To set one, we include an `alpha.mu` and an `alpha.V` which essentially set the extra part of the prior. To set this part of the prior we put `alpha.mu` as a vector of zeros equal to the number of random effects and set `alpha.V` as a nxn matrix with very large numbers were `n` is the number of random effects. In this case we only have one random effect so its easy 1 zero and 1 large number.

```
prior_exp<-list(R = list(V = 1, nu=0.002), G = list(G1=list(V = 1,n = 1, alpha.mu=rep(0,1), alpha.V= di
```

We can use this in our categorical model for example.

```
mod_mcmc_cat_exp <- MCMCglmm(fixed = volant ~ mass,
                      random= ~ genus,
                      family="categorical",
                      data = lifespan_mammals,
                      nitt = nitt2,
                      burnin = burnin2,
                      thin = thin2,
                      prior = prior_exp,
                      verbose=FALSE)
summary(mod_mcmc_cat)
```

```
##
##  Iterations = 40001:239901
##  Thinning interval  = 100
##  Sample size  = 2000
##
##  DIC: 0.2200763
##
##  G-structure:  ~genus
##
##       post.mean l-95% CI u-95% CI eff.samp
## genus    282670 0.004191   531211    20.79
##
##  R-structure:  ~units
##
```

```
##       post.mean l-95% CI u-95% CI eff.samp
## units     12293  0.05405    76096    201.6
##
##  Location effects: volant ~ mass
##
##             post.mean l-95% CI u-95% CI eff.samp  pMCMC
## (Intercept)    -801.2  -1231.5   -376.5    8.281 <5e-04 ***
## mass           -627.1  -1023.1   -327.3    4.747 <5e-04 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Again however this model would need to be run for much longer even with the parameter expanded prior it may have problems converging.