# Owls example: a zero-inflated, generalized linear mixed model for count data

Ben Bolker, Mollie Brooks, Beth Gardner, Cleridy Lennert, Mihoko Minami

October 23, 2012

## 1 The model

Data collected in ecological studies are often complex. Studies may involve repeat observations on the same units (e.g., individuals, quadrats, stations). Predictor variables may be categorical or continuous, and interactions may be of interest. In addition, such data may contain excess zero-valued observations (with respect to a Poisson model) because of sampling variability and/or incomplete knowledge of processes that generated the data (e.g., factors that define suitable species habitat) Zuur et al. (2009). *Zero-inflated generalized linear mixed-effects models* (ZIGLMMs) are a class of models, incorporating aspects of generalized linear models, mixed models, and zero-inflated models, that are both flexible and computationally efficient tools for data of this sort.

The data for this example, taken from Zuur et al. (2009) and ultimately from Roulin and Bersier (2007), quantify the amount of sibling negotiation (vocalizations when parents are absent) by owlets (owl chicks) in different nests as a function of food treatment (deprived or satiated), the sex of the parent, and arrival time of the parent at the nest. Since the same nests are observed repeatedly, it is natural to consider a mixed-effects model for these data, with the nest as a random effect. Because we are interested in explaining variability in the number of vocalizations *per chick*, the total brood size in each nest is used as an offset in the model.

Since the same nests are measured repeatedly, Zuur et al. (2009) fitted a Poisson generalized linear mixed-effects model to these data (see their Section 13.2.2). We extend that example by considering zero-inflation.

We use a zero-inflated Poisson model with a log link function for the count (Poisson) part [i.e., the inverse link is exponential] of the model and a logit link for the binary part [i.e., the inverse link is logistic]

$$Y_{ij} \sim \text{ZIPois}(\lambda = \exp(\eta_{ij}), p = \text{logistic}(z_i)) \tag{1}$$

where ZIPois represents the zero-inflated Poisson distribution, and $\text{logistic}(z) \equiv (1+\exp(-z))^{-1}$. The zero-inflated Poisson with parameters $\lambda$, $p$ has probability $p + (1-p)\exp(-\lambda)$ if $Y = 0$ and $(1-p)\text{Pois}(X, \lambda)$ if $Y > 0$.

In this case we use only a single, overall zero-inflation parameter for the whole model — $z_i = z_0$. (In R's Wilkinson-Rogers notation, this would be written as `z~1`.)

The model for the vector $\boldsymbol{\eta}$, the linear predictor of the Poisson part of the distribution, follows a standard linear mixed model formulation:

$$(\boldsymbol{\eta}|\mathcal{B} = \mathbf{b}) = \mathbf{X}\boldsymbol{\beta} + \mathbf{Z}\mathbf{b} + \mathbf{f} \tag{2}$$

and

$$\mathcal{B} \sim \text{Normal}(\mathbf{0}, \boldsymbol{\Sigma}_\theta) \tag{3}$$

where $\boldsymbol{\eta}$ is the vector of linear predictors; $\mathcal{B}$ is the random variable representing a vector of random effects; $\mathbf{b}$ is a *particular* value of this random variable; $\mathbf{X}$ is the fixed-effect design matrix; $\boldsymbol{\beta}$ is the vector of fixed-effect parameters; $\mathbf{Z}$ is the random-effect design matrix; $\boldsymbol{\Sigma}_\theta$ is the variance-covariance matrix of the random effects, with parameters $\boldsymbol{\theta}$; and $\mathbf{f}$ is an offset vector.

In this case, the fixed-effect design matrix $\mathbf{X}$ includes columns for the intercept, difference between males and females, between food treatments, arrival times, and their interactions. The random effect design matrix $\mathbf{Z}$ is a simple dummy matrix encoding nest identity. There is only a single random effect, so $\boldsymbol{\Sigma}_{\boldsymbol{\theta}}$ is a diagonal matrix with the random effects (nest) variance $\sigma^2_{\text{nest}}$ on the diagonal (the nest random effects are independent and identically distributed) and the random-effects parameter vector is just $\boldsymbol{\theta} = \{\sigma^2_{\text{nest}}\}$. The offset $\mathbf{f}$, equal to the log of the number of chicks in the nest, accounts for the fact that the data give the total number of vocalizations per *nest*, while we are ultimately interested in the total number of vocalizations per *chick*.

For the purposes of model comparison (timing, accuracy etc.) we will stick with the offset, zero-inflated Poisson model described by eqs. (2, 1), but in the discussion below we will also consider alternative formulations that seem to fit the data well (e.g. using overdispersed distributions such as the lognormal-Poisson or negative binomial; allowing the response to brood size to be other than strictly proportional).

## 2 Preliminaries

Load packages:

```
> library(coda)
> library(reshape)
> library(MCMCglmm)
> library(coefplot2)
> library(glmmADMB)
> library(bbmle)
> library(ggplot2)
> theme_set(theme_bw())  ## set color scheme
> library(RColorBrewer)
> library(R2jags)
> ## library(lme4)  ## wait to load -- glmmADMB/lme4 conflicts
```

Package versions:

```
      bbmle         coda         coda    coefplot2      ggplot2     glmmADMB 
    1.0.5.2       0.15-2       0.15-2        0.1.3      0.9.2.1    0.7.2.12 
       lme4     MCMCglmm       R2jags    R2WinBUGS RColorBrewer      reshape 
0.99999911-0        2.16      0.03-08       2.1-18        1.0-5        0.8.4 
      rjags 
        3-7 
```

Load the data and use the `rename` function from the `reshape` package to change the name of the response variable:

```
> load("../DATA/Owls.rda")
> library(reshape)
> Owls <- rename(Owls,c(SiblingNegotiation="NCalls"))
```

Scale arrival time: necessary for WinBUGS, useful to have it done up front for the sake of parameter comparisons[1]

```
> Owls <- transform(Owls,ArrivalTime=scale(ArrivalTime,center=TRUE,scale=FALSE))
```

## 3 R

There are (at least) three different ways to do this problem in R, although (as far as we know) there is no simple, out-of-the-box method that is built purely on R (or underlying C or FORTRAN code) that solves the problem as we have stated it.

- The reference method, used for comparisons with ADMB and WinBUGS, is the `MCMCglmm` package; this method is the only one that works "out of the box" in R without recourse to other (non-R) tools. Its only disadvantage is that it fits a lognormal-Poisson version of the model (i.e., with an observation-level random effect added (Elston et al., 2001) rather than the reference Poisson model. In terms of equation 2, we add a parameter $\sigma^2_{\text{obs}}$ to the random-effects parameter vector $\boldsymbol{\theta}$ and expand $\boldsymbol{\Sigma_\theta}$ to incorporate an additional set of diagonal components $\sigma^2_{\text{obs}}$ (the observation-level random effects are also independent and identically distributed).

  (Because it allows for overdispersion in the count part of the model, this model is actually superior to the ZIP we originally proposed, but it is not the reference model.)

[1]here we *replace* the original variable with the scaled version, rather than creating a new variable called (say) `scArrivalT`. This can potentially lead to confusion if we forget whether we are dealing with the scaled or the unscaled version ...

- One can cheat a little bit and make use of ADMB (but without having to do any explicit AD Model Builder coding) by using the `glmmADMB` package, which encapsulates a subset of ADMB's capabilities into a pre-compiled binary that can be run from within R using a fairly standard formula syntax.
- With a bit of hacking, one can write up a fairly simple, fairly generic implementation of the expectation-maximization (EM) algorithm that alternates between fitting a GLMM (using `glmer`) with data that are weighted according to their zero probability, and fitting a binary GLM (using `glm`) for the probability that a data point is zero. We have implemented this in the code in `owls_R_funs.R`, as a function called `zipme` (see below).

## 3.1 MCMCglmm

As mentioned above, we use `MCMCglmm` for our reference R implementation.

```
> library(MCMCglmm)
```

Set up a variable that will pick out the offset (log brood size) parameter, which will be in position 3 of the parameter vector:

```
> offvec <- c(1,1,2,rep(1,5))
```

While `MCMCglmm` can easily fit a ZIGLMM, specifying the fixed effect is a little bit tricky. For zero-inflated models, `MCMCglmm` internally constructs an augmented data frame: if the original data frame is

```
resp      f1  x1
1         A   1.1
0         A   2.3
3         B   1.7
```

where `resp` is a Poisson response, `f1` is a factor, and `x1` is a continuous predictor, then the augmented data frame will look like this:

```
resp      trait   f1  x1
1         resp    A   1.1
0         resp    A   2.3
3         resp    B   1.7
1         zi_resp A   1.1
0         zi_resp A   2.3
1         zi_resp B   1.7
```

The `trait` column distinguishes two types of "pseudo-observations": the `resp` values give the actual observed values, for modeling the count portion of the data, while the `zi_resp` values reduce the results to binary (0/1) form.

`MCMCglmm` provides a special helper function, `at.level`, which enables us to specify that some covariates affect only the count part of the model (`resp`), or only the binary part (zi_resp) of the model: the first would be specified as an interaction of at.level(trait,1) with a covariate (i.e., the covariate affects only responses for level 1 of trait, which are the count responses).

Here is the model specified by Zuur et al. (2009), which includes an offset effect of brood size and most but not all of the interactions between FoodTreatment, SexParent, and ArrivalTime on the count aspect of the model, but only a single fixed (intercept) effect on the binary part of the model (i.e., the zero-inflation term):

```
> fixef2 <- NCalls~trait-1+ ## intercept terms for both count and binary terms
   at.level(trait,1):logBroodSize+
   at.level(trait,1):((FoodTreatment+ArrivalTime)*SexParent)
```

(If we wanted to include covariates in the model for the level of zero-inflation we would use an interaction with at.level(trait,2): for example, e.g. we would add a term at.level(trait,2):SexParent to the fixed-effect model if we wanted to model a situation where the zero-inflation proportion varied according to parental sex.)

The next complexity is in the specification of the priors, which (ironically) we have to do in order to make the model simple enough. By default, MCMCglmm will fit the same random effect models to both parts of the model (count and binary). Here, we want to turn off the random effect of nest for the binary aspect of the model. In addition, MCMCglmm always fits residual error terms for both parts of the model. In our first specification, we first fix the residual error (R) of the binary part of the data to 1 (because it is not identifiable) by setting fix=2; the parameter nu=0.002 specifies a very weak prior for the other (non-fixed) variance term. (In order to get reasonable mixing of the chain we have to fix it to a non-zero value.) We also essentially turn off the random effect on the binary part of the model by fixing its variance to $10^{-6}$, in the same way.

```
> prior_overdisp  <- list(R=list(V=diag(c(1,1)),nu=0.002,fix=2),
                         G=list(list(V=diag(c(1,1e-6)),nu=0.002,fix=2)))
```

prior_overdisp will serve as a base version of the prior, but we also want to specify that the log brood size enters the model as an offset. We do this by making the priors for all *but* the log-brood-size effect (nearly) equal to the default value for fixed effects [$N(\mu = 0, \sigma^2 = 10^8)$ — the variance for the log brood size effect is an (extremely) weak prior centered on zero] and setting a very strong prior centered at 1 [$N(\mu = 1, \sigma^2 = 10^{-6})$] for the log brood size effect.[2]

```
> prior_overdisp_broodoff <- c(prior_overdisp,
                               list(B=list(mu=c(0,1)[offvec],
                                 V=diag(c(1e8,1e-6)[offvec]))))
```

[2]If we set $\sigma^2 = 10^{10}$, the default value, for the non-offset predictors, we get an error saying that fixed effect V prior is not positive definite — the difference in variances is so large that the smaller variance underflows to zero in a calculation.

Now we fit the lognormal version of the ZIPois model as follows:

```
> mt1 <- system.time(mfit1 <- MCMCglmm(fixef2,
                                       rcov=~idh(trait):units,
                                       random=~idh(trait):Nest,
                                       prior=prior_overdisp_broodoff,
                                       data=Owls,
                                       family="zipoisson",
                                       verbose=FALSE))
```

For comparison, we'll also try the model with the log-brood-size parameter allowed to vary from 1 (i.e. dropping the prior specification that fixes its value at 1):

```
> mt2 <- system.time(mfit2 <- MCMCglmm(fixef2,
                                       rcov=~idh(trait):units,
                                       random=~idh(trait):Nest,
                                       prior=prior_overdisp,
                                       data=Owls,
                                       family="zipoisson",
                                       verbose=FALSE))
```

These fits take 28.7 and 26.4 seconds respectively.

Define a utility function to abbreviate the variable names from the slightly verbose MCMCglmm results and apply it to the relevant portions of the fits:

```
> abbfun <- function(x) {
   gsub("(Sol\\.)*(trait|at.level\\(trait, 1\\):)*","",
        gsub("FoodTreatment","FT",x))
 }
> colnames(mfit1$Sol) <- abbfun(colnames(mfit1$Sol))
> colnames(mfit2$Sol) <- abbfun(colnames(mfit2$Sol))
```

Now look at the results:

```
> summary(mfit1)

 Iterations = 3001:12991
 Thinning interval  = 10
 Sample size  = 1000

 DIC: 3055.7

 G-structure:  ~idh(trait):Nest

              post.mean l-95% CI u-95% CI eff.samp
NCalls.Nest     0.095764 0.020663 0.203210    439.1
zi_NCalls.Nest  0.000001 0.000001 0.000001      0.0
```

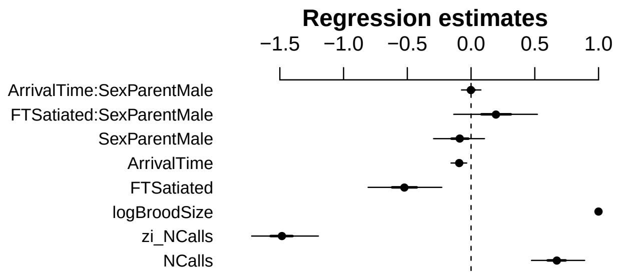
```
 R-structure:  ~idh(trait):units

                post.mean l-95% CI u-95% CI eff.samp
NCalls.units       0.4268   0.3357   0.5276    325.4
zi_NCalls.units    1.0000   1.0000   1.0000      0.0

 Location effects: NCalls ~ trait - 1 + at.level(trait, 1):logBroodSize + at.level(trait, 1)

                            post.mean   l-95% CI   u-95% CI eff.samp  pMCMC    
NCalls                      0.6736582  0.4730089  0.8873843   671.15 <0.001 ***
zi_NCalls                  -1.4825091 -1.7326731 -1.2121988    77.92 <0.001 ***
logBroodSize                0.9999923  0.9981527  1.0020144  1000.00 <0.001 ***
FTSatiated                 -0.5222586 -0.7947399 -0.2239459   330.67 <0.001 ***
ArrivalTime                -0.0924656 -0.1523303 -0.0319161   548.98 <0.001 ***
SexParentMale              -0.0885357 -0.2946377  0.1053200   811.02  0.390    
FTSatiated:SexParentMale    0.1956149 -0.1269305  0.5252096   569.55  0.254    
ArrivalTime:SexParentMale -0.0001698 -0.0761542  0.0754351   540.74  0.968    
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

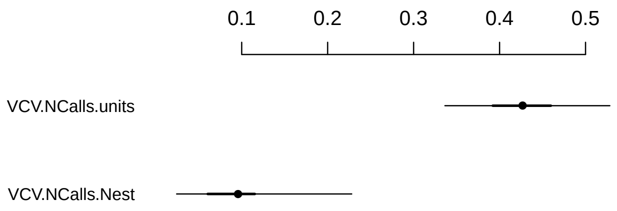
Moving through this in sequence:

- the first three lines give information about the chain parameters: using the MCMCglmm default, 3000 “burn-in” samples have been taken, followed by 10000 steps that are sampled every 10 steps
- the DIC, or deviation information criterion, is useful (with caveats) for model comparison (Spiegelhalter et al., 2002)
- the $G$ (random effects) variance for the count part of the model (`zi_NCalls.Nest`) has been succesfully fixed to a small value; the variance among nests is small but non-zero
- the residual (among-unit) variance is quite large for number of calls (suggesting overdispersion); it is fixed to 1.0 for the count part of the model, as described above
- the fixed-effect parameter table gives the mean, 95% credible intervals, effective sample size, and Bayesian $p$-value (a two-tailed test of the more extreme of the fraction of samples above or below zero) for both the binary intercept (the logit of the zero-inflation probability $z_0$, named `zi_NCall` here) and the the fixed effects of the parameters. It looks like parental sex is not doing much, at least as measured in terms of $p$ values.

Alternatively, we can represent the results graphically:

```
> library(coefplot2)
> op <- par(mfrow=c(2,1))
> vn1 <- abbfun(rownames(coeftab(mfit2)))
> coefplot2(mfit1,intercept=TRUE,varnames=vn1)
> coefplot2(mfit1,var.idx=c(1,3),ptype="vcov",
          main="")
> par(op)
```

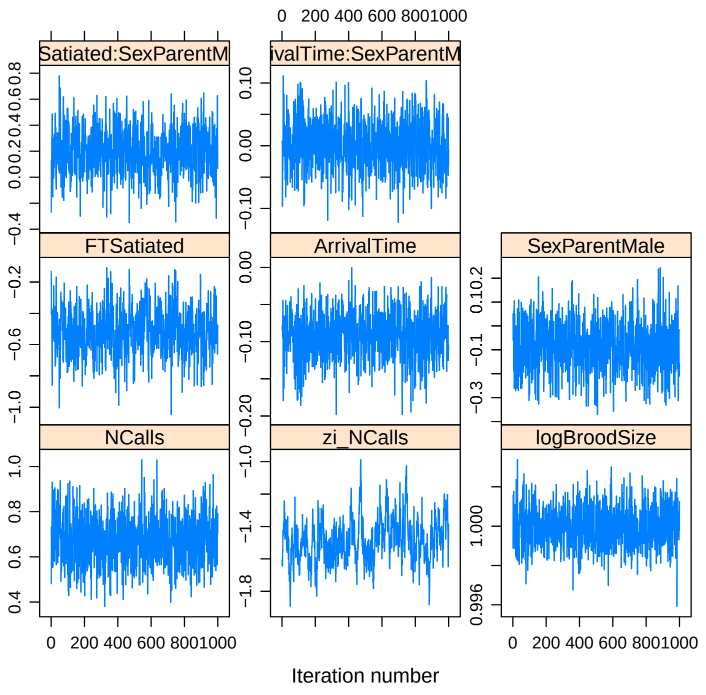




We have to look at the *trace plots* — plots of the time series of the Markov chain — to make sure that the fits behaved appropriately. We are hoping that the trace plots look like white noise, with rapid and featureless variation:

For the fixed effects:

```
> print(xyplot(mfit1$Sol,layout=c(3,3)))
```



The one parameter that looks slightly questionable is `zi_NCalls`, and indeed its effective size is rather lower than we'd like (we should be aiming for at least 200 if we want reliable confidence intervals):

```
> round(sort(effectiveSize(mfit1$Sol)))

              zi_NCalls              FTSatiated ArrivalTime:SexParentMale
                     78                     331                       541
            ArrivalTime  FTSatiated:SexParentMale                    NCalls
                    549                     570                       671
          SexParentMale            logBroodSize
                    811                    1000
```

Note that although `logBroodSize` is varying, it's varying over a very small range (0.998-1.002) because of the strong prior we imposed.

We can also run a quantitative check on convergence, using `geweke.diag` which gives a $Z$-statistic for the similarity between the first 10% and the last 50% of the chain:

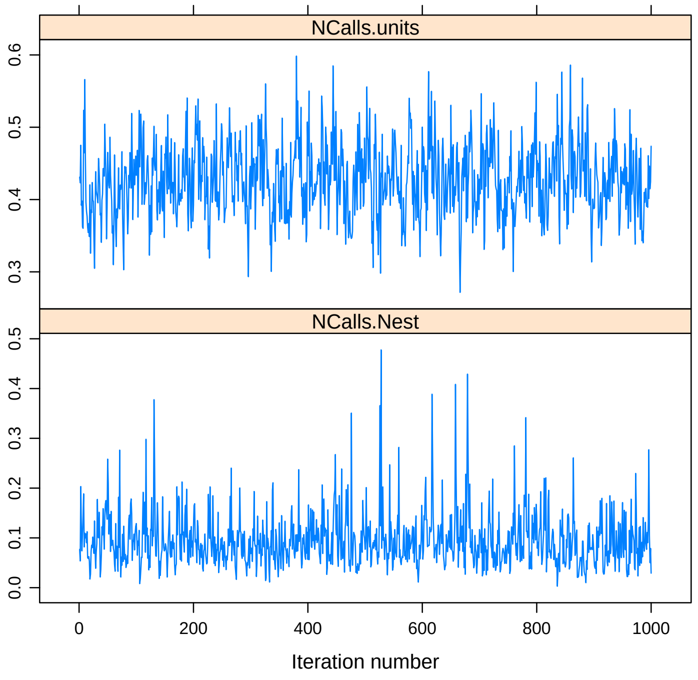
```
> geweke.diag(mfit1$Sol)

Fraction in 1st window = 0.1
Fraction in 2nd window = 0.5
```

```
                   NCalls                 zi_NCalls              logBroodSize
                  0.94798                  -1.75351                  -0.07509
               FTSatiated               ArrivalTime             SexParentMale
                  0.13767                  -0.58302                  -0.41654
FTSatiated:SexParentMale ArrivalTime:SexParentMale
                  0.63148                   1.25217
```

All the values are well within the 95% confidence intervals of the standard normal (i.e., $|x| < 1.96$).

For variance parameters:

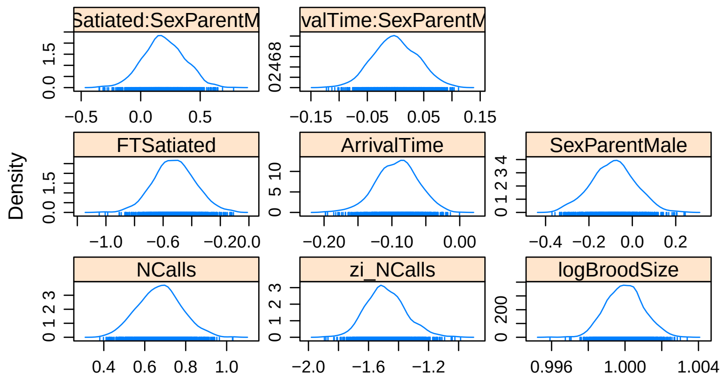
```
> vv <- mfit1$VCV
> ## drop uninformative ZI random effects
> vv <- vv[,c("NCalls.Nest","NCalls.units")]
> print(xyplot(vv,layout=c(1,2)))
> effectiveSize(vv)

 NCalls.Nest NCalls.units
    439.1036     325.4316
```



Density plots display the posterior distributions of the parameters, useful for checking whether the distribution of sampled points is somehow odd (strongly skewed, bimodal, etc.). Symmetry and approximate normality of the posterior are useful for inference (e.g. the DIC is based on an assumption of approximate normality, and quantiles and highest posterior density intervals give the same results for symmetry), but not absolutely necessary.

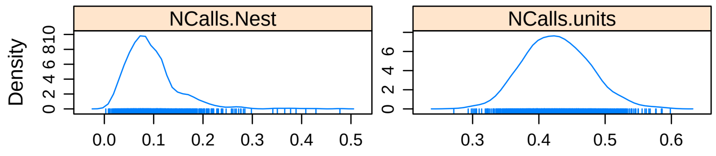
For fixed effects:

```
> print(densityplot(mfit1$Sol,layout=c(3,3)))
```



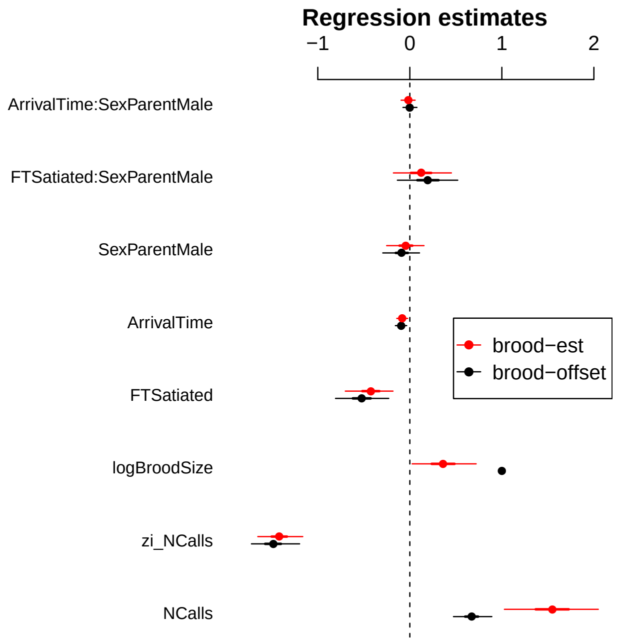
For variance parameters:

```
> print(densityplot(vv,layout=c(2,1)))
```



Comparing the fits of the model with and without log brood size:

```
> coefplot2(list("brood-offset"=mfit1,"brood-est"=mfit2),
            intercept=TRUE,
            varnames=vn1,
            legend=TRUE,legend.x="right")
```



With the exception of the log brood size and intercept parameters, the parameters are similar. The log brood size parameter is quite far from 1:

```
> coda::HPDinterval(mfit2$Sol)["logBroodSize",]

     lower      upper
0.03850979 0.73115415
```

We can also create versions of the model that attempt to eliminate the overdispersion in the count part of the model. We can't fix both variance parameters, but we can make the variance prior informative (by setting `nu=100`, or `nu=1000`) and make the (1,1) element of the variance matrix small. However, if we try too hard to do this, we sacrifice a well-mixed chain (this mixing property is the reason that `MCMCglmm` automatically adds an observation-level variance to every model). Since the model runs *reasonably* fast, for this case we might be able to use brute force and just run the model 10 or 100 times as long ...

We could set up prior specifications for this non-overdispersed case as follows:

```
> prior_broodoff <- within(prior_overdisp_broodoff,
                           R <- list(V=diag(c(1e-6,1)),nu=100,fix=2))
> prior_null <- within(prior_overdisp,
                       R <- list(V=diag(c(1e-6,1)),nu=100,fix=2))
```

### 3.2 glmmADMB

This problem can also be done with `glmmADMB`. The model is about equally fast, and the coding is easier! On the other hand, there is arguably some advantage to having the MCMC output (which would take longer to get with ADMB).

```
> library(glmmADMB)
```

```
> gt1 <- system.time(gfit1 <- glmmadmb(NCalls~(FoodTreatment+ArrivalTime)*SexParent+
                                        offset(logBroodSize)+(1|Nest),
                                        data=Owls,
                                        zeroInflation=TRUE,
                                        family="poisson"))
```

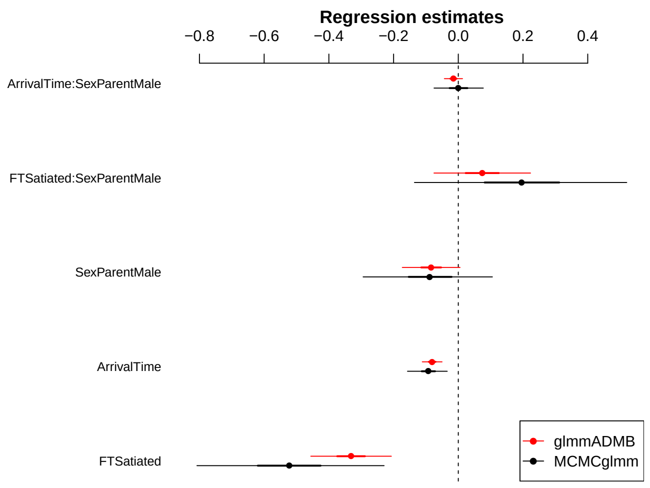
It takes 26.9 seconds, slightly faster than `MCMCglmm`.

```
> summary(gfit1)
```

```
Call:
glmmadmb(formula = NCalls ~ (FoodTreatment + ArrivalTime) * SexParent +
    offset(logBroodSize) + (1 | Nest), data = Owls, family = "poisson",
    zeroInflation = TRUE)


Coefficients:
                                    Estimate Std. Error z value Pr(>|z|)
(Intercept)                           0.8571     0.0823   10.41  < 2e-16 ***
FoodTreatmentSatiated                -0.3314     0.0635   -5.22  1.8e-07 ***
ArrivalTime                          -0.0807     0.0156   -5.18  2.3e-07 ***
SexParentMale                        -0.0838     0.0455   -1.84    0.066 .
FoodTreatmentSatiated:SexParentMale   0.0740     0.0761    0.97    0.331
ArrivalTime:SexParentMale            -0.0150     0.0143   -1.05    0.295
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Number of observations: total=599, Nest=27
Random effect variance(s):
Group=Nest
            Variance StdDev
(Intercept)     0.14 0.3742
Zero-inflation: 0.25833  (std. err.:  0.018107 )

Log-likelihood: -1985.3
```

The results are quite similar, although MCMCglmm gives wider confidence intervals in general because it considers the uncertainties in all model components, and because it allows for overdispersion.

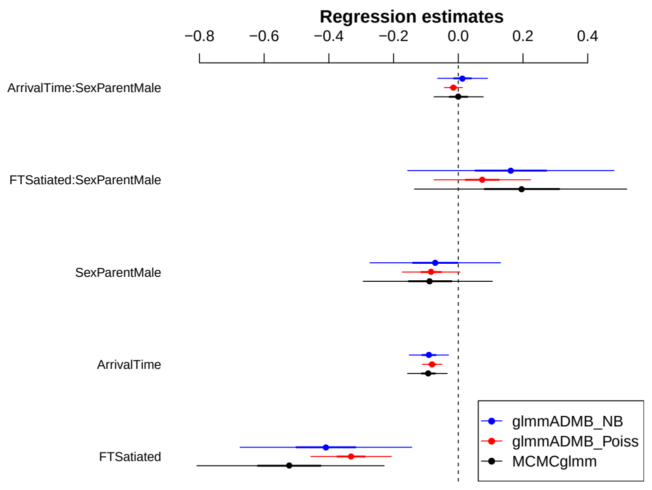
```
> cm1tab <- coeftab(mfit1)[4:8,]
> cg1tab <- coeftab(gfit1)[2:6,]
> coefplot2(list(MCMCglmm=cm1tab,glmmADMB=cg1tab),merge.names=FALSE,intercept=TRUE,
          varnames=abbfun(rownames(cg1tab)),
          legend=TRUE)
```



```
> gt2 <- system.time(gfit2 <- glmmadmb(NCalls~(FoodTreatment+ArrivalTime)*SexParent+
                                       offset(logBroodSize)+(1|Nest),
                                       data=Owls,
                                       zeroInflation=TRUE,
                                       family="nbinom"))
```

This takes a little longer than the Poisson model (20.9 seconds). The parameters are a little bit closer to the MCMCglmm fit (which also allows for overdispersion).

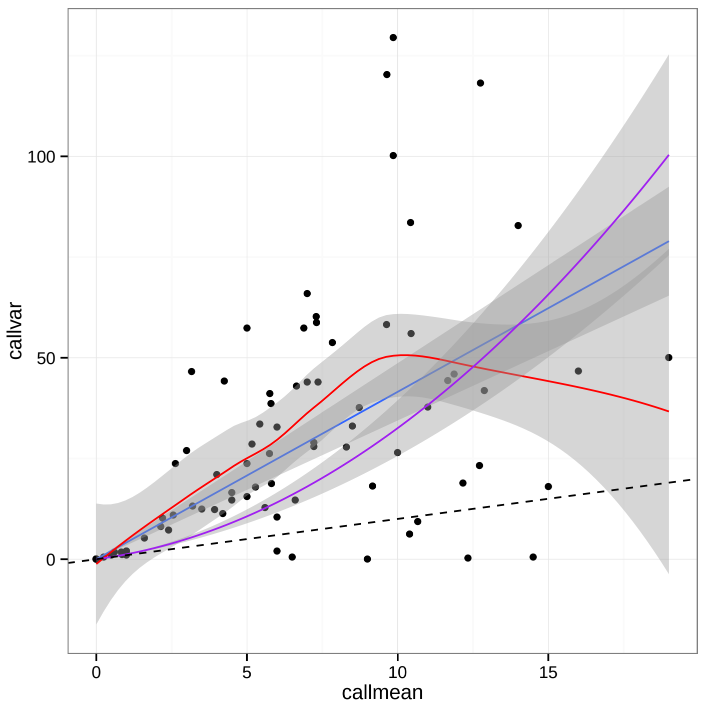
```
> cg2tab <- coeftab(gfit2)[2:6,]
> coefplot2(list(MCMCglmm=cm1tab,glmmADMB_Poiss=cg1tab,glmmADMB_NB=cg2tab),
          merge.names=FALSE,intercept=TRUE,
          varnames=abbfun(rownames(cg1tab)),
          legend=TRUE,
          col=c(1,2,4))
```



It seems that most of the difference in confidence interval size was due to overdispersion (or lack of it), rather than to what components of uncertainty were included.

To explore the variance-mean relationship, we can calculate the mean and variance by group and look at the relationship. A linear relationship implies that a quasi-Poisson or negative binomial "type 1" with variance $V$ proportional the mean $\mu$ (Hardin and Hilbe, 2007) is likely to be best, while a relationship of the form $V = \mu + C\mu^2$ implies that a negative binomial type 2 (the standard in R and most other statistics packages) or lognormal-Poisson fit is likely to be best.

```
> library(plyr)
> mvtab <- ddply(Owls,
                 .(FoodTreatment:SexParent:Nest),
                 summarise,
                 callmean=mean(NCalls),
                 callvar=var(NCalls))
> q1 <- qplot(callmean,callvar,data=mvtab)
> print(q1+
      ## linear (quasi-Poisson/NB1) fit
      geom_smooth(method="lm",formula=y~x-1)+
      ## smooth (loess)
      geom_smooth(colour="red")+
      ## semi-quadratic (NB2/LNP)
      geom_smooth(method="lm",formula=y~I(x^2)+offset(x)-1,colour="purple")+
      ## Poisson (v=m)
      geom_abline(a=0,b=1,lty=2))
```



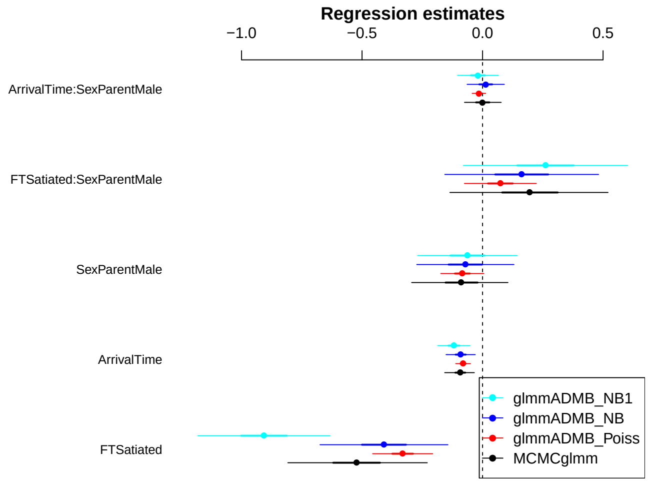
(black=Poisson [$V = \mu$]; purple=quadratic/NB2; blue=linear/NB1; red=nonparametric)

Of the parametric choices, it looks like NB1 is better than NB2. We can fit this in `glmmADMB` via `family="nbinom1"`:

```
> gt3 <- system.time(gfit3 <- glmmadmb(NCalls~(FoodTreatment+ArrivalTime)*SexParent+
                                        offset(logBroodSize)+(1|Nest),
                                        data=Owls,
                                        zeroInflation=TRUE,
                                        family="nbinom1"))
```

It doesn't change the coefficients very much, although it does make the main effect of food stronger:

```
> cg3tab <- coeftab(gfit3)[2:6,]
> coefplot2(list(MCMCglmm=cm1tab,glmmADMB_Poiss=cg1tab,glmmADMB_NB=cg2tab,
                 glmmADMB_NB1=cg3tab),
           merge.names=FALSE,intercept=TRUE,
           varnames=abbfun(rownames(cg1tab)),
           legend=TRUE,
           col=c(1,2,4,5))
```



AIC suggests that (ZI)NB1 is a much better fit to the data.

```
> AICtab(gfit1,gfit2,gfit3)

      dAIC  df
gfit3   0.0 9
gfit2  67.5 9
gfit1 635.8 8
```

## 3.3 EM algorithm

The expectation-maximization (EM) algorithm (e.g., Dempster et al. (1977); Minami et al. (2007)) is an iterative procedure for finding maximum likelihood estimates of model parameters. To find parameter estimates, the EM algorithm iterates between expectation and maximization steps until convergence. The response variable of a zero-inflated Poisson model can be viewed as arising from one of two states (e.g., Lambert (1992)): a 'perfect' ('zero') state where only the value zero is possible or an imperfect state where any non-negative value is possible. In this conceptualization, the zero-valued observations of a data set could come from either state, whereas positive counts could only come from the imperfect state. At each iteration of the EM algorithm, the purpose of the expectation step is to estimate the probability that each zero- valued observation is in the 'perfect' state, given the parameters of the logistic and log-linear models (which are estimated in the maximization steps). In our implementation, we assume that the logistic and log-linear models have no shared parameters.

In our case, we use the `glmer` function from `lme4` to fit the Poisson part of the regression, and either the `glmer` function (if the zero-inflation model contains a random effect) or the `glm` function (if not) for the logistic regression ($Z$) part of the algorithm. The algorithm is encapsulated in a `zipme` function we have written which takes a standard R model formula for the Poisson part (`cformula`); a model formula with `z` on the left-hand side for the zero-inflation part; and addition parameters `data`, `maxitr` (maximum number of iterations), `tol` (tolerance criterion), and `verbose` (whether to print out progress messages).

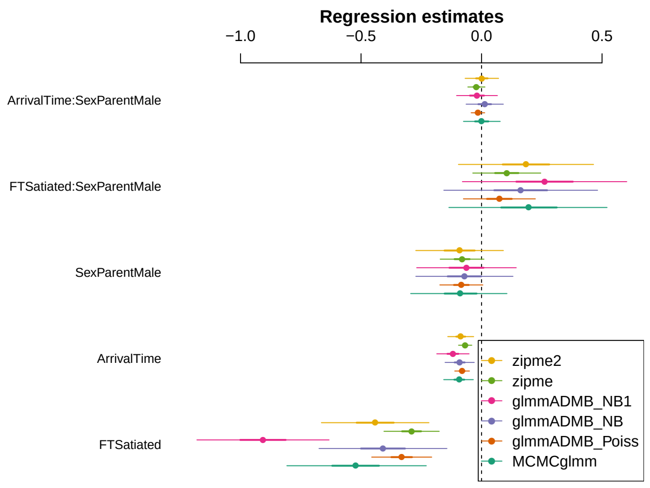
```
> source("../R/owls_R_funs.R")
> library(lme4)

> zt1 <- system.time(ofit_zipme <-
                   zipme(cformula=NCalls~(FoodTreatment+ArrivalTime)*SexParent+
                         offset(logBroodSize)+(1|Nest),
                         zformula=z ~ 1,
                         data=Owls,maxitr=20,tol=1e-6,
                         verbose=FALSE))

> Owls$obs <- seq(nrow(Owls))
> zt2 <- system.time(ofit2_zipme <-
                   zipme(cformula=NCalls~(FoodTreatment+ArrivalTime)*SexParent+
                         offset(logBroodSize)+(1|Nest)+(1|obs),
                         zformula=z ~ 1,
                         data=Owls,maxitr=20,tol=1e-6,
                         verbose=FALSE))
```

The EM fits take 9.9 and 91.2 seconds without and with observation-level random effects, respectively.

The EM results are similar to the corresponding fits from other approaches: the Poisson fit looks most similar to the `glmmADMB` Poisson fit (this is the reference model, which doesn't account for overdispersion) and the lognormal-Poisson fit (with an observation-level random effect) looks most like the `MCMCglmm` fit.

The only innovation here is using `brewer.pal(6,"Dark2")` from the `RColorBrewer` package to get some nicer colors.

```
> cg4tab <- coeftab(ofit_zipme$cfit)[2:6,]
> cg5tab <- coeftab(ofit2_zipme$cfit)[2:6,]
> library(RColorBrewer)
> cvec <- brewer.pal(6,"Dark2")
> coefplot2(list(MCMCglmm=cm1tab,glmmADMB_Poiss=cg1tab,glmmADMB_NB=cg2tab,
                 glmmADMB_NB1=cg3tab,
                 zipme=cg4tab,
                 zipme2=cg5tab),
           merge.names=FALSE,intercept=TRUE,
           varnames=abbfun(rownames(cg1tab)),
           legend=TRUE,
           col=cvec)
```


Regression estimates
−1.0
−0.5
0.0
0.5
ArrivalTime:SexParentMale
FTSatiated:SexParentMale
SexParentMale
ArrivalTime
FTSatiated
zipme2
zipme
glmmADMB_NB1
glmmADMB_NB
glmmADMB_Poiss
MCMCglmm

# 4 ADMB

## 4.1 Code in owls.tpl

```
1  DATA_SECTION
2          init_int nobs       // # of observations (599)
3          init_int nnests     // # of Random Effect Levels (Nests) (27)
4          init_int nf         // # of fixed effects incl interactions and intercept
5          init_matrix fdesign(1,nobs,1,nf)       //Design matrix (intercept & fixed effects)
6          init_matrix rdesign(1,nobs,1,nnests)   //Design matrix (random effects)
7          init_vector obs(1,nobs)                //Response variable (number of calls)
8          init_vector Lbrood(1,nobs)             //Log of brood size (offset)
9
10 PARAMETER_SECTION
11         init_vector fcoeffs(1,nf)                 //Intercept & fixed effect coeffs
12         init_number logit_p                        //Logit of zero inflation parameter
13         init_bounded_number sigma(0.001,100)
14         vector eta(1,nobs)
15         vector mu(1,nobs)
16         number p
17         objective_function_value jnll
18         random_effects_vector rcoeffs(1,nnests)
19
20 PROCEDURE_SECTION
21         jnll = 0.0;
22         p = exp(logit_p)/(1.0+exp(logit_p));
23         eta = Lbrood + fdesign*fcoeffs + rdesign*(rcoeffs*sigma);
24         mu = exp(eta);
25
26         for(int i=1; i<=nobs; i++)
27         {
28                 if(obs(i)==0)
29                 {
30                         jnll-=log(p + (1-p)*exp(-mu(i) ) );
31                 }
32                 else//obs(i)>0
33                 {
34                         jnll-=log(1-p) + obs(i)*log(mu(i)) - mu(i)- gammln(obs(i)+1);
35                 }
36         }
37
38         jnll+=0.5*(rcoeffs*rcoeffs)+0.5*nnests*log(2.*M_PI); // for unscaled [N(0,1)] REs
39
40 TOP_OF_MAIN_SECTION
41         gradient_structure::set_MAX_NVAR_OFFSET(764);
42 GLOBALS_SECTION
43
```

- `sigma` is bounded in (0.001,100)
- line 21 initializes the objective function to zero. `jnll` is where we'll sum up

the joint negative log-likelihood of the observation model and the random effects.

- line 22 transforms the logit of the zero-inflation parameter back to the probability scale
- line 23 computes the linear predictor (offset + fixed effects + random effects): in `rcoeffs*sigma` (scaling the random effects by the random-effects standard deviation: vector × scalar), `*` acts as elementwise multiplication; in the other cases (multiplying fixed-effect and random-effect design matrices by the corresponding parameter vectors) it acts as matrix multiplication
- line 24 converts the linear predictor from the log to the count scale;
- lines 26–36 loop over the observations, subtracting the ZIP log-likelihood (computed on lines 28–35) for each observation to the objective function value (joint negative log-likelihood) `jnll`
- line 38 adds the negative log-likelihood of the random effects to `jnll`
- line 42 will make sure there's enough space. ADMB told us to use this number when we tried to run the program.

## 4.2 Running the model from within R

Load the `R2admb` package and tell R where to find the `admb` executable:

```
> library(R2admb)
> setup_admb()
```

```
[1] "/usr/local/admb"
```

Read in the data (if we hadn't already done so):

```
> load("../DATA/Owls.rda")
> ## don't forget to transform!
> Owls <- transform(Owls,ArrivalTime=scale(ArrivalTime,center=TRUE,scale=FALSE))
```

Organize the data — these definitions need to be defined in advance in order for `R2admb` to check them properly ...

```
> LBrood <- log(Owls$BroodSize)
> mmf <- model.matrix(~(FoodTreatment+ArrivalTime)*SexParent,
                      data=Owls)
> mmr <- model.matrix(~Nest-1, data=Owls)
> response <- Owls$SiblingNegotiation
> nf <- ncol(mmf)
> nobs <- nrow(Owls)
> nnests <- length(levels(Owls$Nest))
```

Combine these objects into a list and write them to a file in the appropriate format for ADMB:

```
> regdata=list(nobs=nrow(Owls),
      nnests=length(levels(Owls$Nest)),
      nf=ncol(mmf),
      fdesign=mmf,
      rdesign=mmr,
      obs=response,
      Lbrood=LBrood)
> write_dat("owls", L=regdata)
```

Define and write a list of starting values for the coefficients:

```
> regparams=list(fcoeffs=rep(0,ncol(mmf)),
   logit_p=0,
   sigma=1,
   rcoeffs=rep(0.001,length(levels(Owls$Nest))))
> write_pin("owls", L=regparams)
```

Compile the model, specifying that it contains random effects:

```
> compile_admb("owls", re=TRUE)
```

Run the compiled executable:

```
> xargs <- "-noinit -nox -l1 40000000 -nl1 40000000 -ilmn 5"
> run_admb("owls",extra.args=xargs)
```

The extra argument `-noinit` tells ADMB to start the random effects from the last optimum values, instead of the `pin` file values, when doing the Laplace approximation. `-nox` reduces the amount of information output while it's running. `-l1` allocates memory to prevent ADMB from creating the temporary storage file `f1b2list1`, which is much slower than using RAM. `-nl1` is similar to `-l1`, but for the temporary file `nf1b2list1`. Users add these command line options when they see temporary files created. The amount to allocate is done by trial and error or experience. In this user's experience, 40000000 is a good value to try. `-ilmn 5` is used to make ADMB run faster when there are a lot of random effects; it tells ADMB to use a limited memory quasi-Newton optimization algorithm and only save 5 steps. (See the ADMB and ADMB-RE manuals, and `http://admb-project.org/community/tutorials-and-examples/memory-management`, for more information.)

Read in the results and clean up files that were produced by ADMB:

```
> fit_admb <- read_admb("owls")
> ## rename fixed-effect parameters according to column
> ##  names of model matrix
> names(fit_admb$coeflist$fcoeffs) <-
   names(fit_admb$coefficients)[1:ncol(mmf)] <- colnames(mmf)
> clean_admb("owls")
```

## 4.3 Results

The `fit_admb` object read in by `R2admb` works with many of the standard accessor methods in R: `coef`, `stdEr`, `vcov`, `confint`, etc.:

```
> methods(class="admb")
```

```
 [1] AIC.admb*      coef.admb*     coeftab.admb*  confint.admb*  deviance.admb*
 [6] logLik.admb*   print.admb*    stdEr.admb*    summary.admb*  vcov.admb*

   Non-visible functions are asterisked
```

However, the coefficients vector contains the full set of fixed and random effect coefficients (as well as the zero-inflation parameter and the random-effects variance):

```
> logLik(fit_admb)
```

```
[1] -1999.82
```

```
> coef(fit_admb)[1:8]
```

```
                       (Intercept)              FoodTreatmentSatiated
                         0.8544950                         -0.2911060
                       ArrivalTime                      SexParentMale
                        -0.0680782                         -0.0809064
FoodTreatmentSatiated:SexParentMale           ArrivalTime:SexParentMale
                         0.1047250                         -0.0213974
                           logit_p                              sigma
                        -1.0575332                          0.3596638
```

These coefficients are in the same order as the columns of the model matrix `mmf`

```
> colnames(mmf)
```

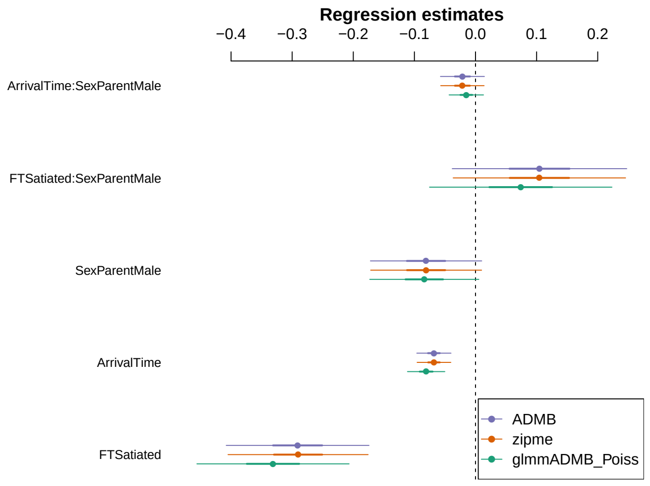
```
[1] "(Intercept)"                         "FoodTreatmentSatiated"
[3] "ArrivalTime"                         "SexParentMale"
[5] "FoodTreatmentSatiated:SexParentMale" "ArrivalTime:SexParentMale"
```

Checking equivalence:

```
> write_pin("owls",as.list(c(coef(gfit1),
                             logitpz=qlogis(gfit1$pz),
                             sigma=sqrt(gfit1$S[[1]]))))
```

Comparing the three reference implementations (with `glmmADMB`, the EM algorithm (`zipme`), and ADMB): **do we care that the results from glmmADMB and ADMB aren't quite identical? Is this due to "robustification"? glmmADMB has a better log-likelihood (1985 vs 1999) — either ADMB got stuck or the likelihoods are calculated differently (robustification etc.)**

```
> ## rearrange order to match
> a1tab <- coeftab(fit_admb)[2:6,]
> cvec <- brewer.pal(6,"Dark2")
> coefplot2(list(glmmADMB_Poiss=cg1tab,
                 zipme=cg4tab,
                 ADMB=a1tab),
           merge.names=FALSE,intercept=TRUE,
           varnames=abbfun(rownames(cg1tab)),
           legend=TRUE,
           col=cvec[1:3])
```



## 4.4 Faster version using separable functions

The same model can be written in a much more efficient way if we exploit the separability of its parameters. We do this by operating on each observation separately and only sending the necessary parameters to the separable function. This lets ADMB know how sparse the Hessian will be. The fewer parameters get sent together, the sparser the Hessian. So instead of using a matrix for the random effects, we use an index vector for which nest corresponds to each observation. Then, for each observation, only the random effect for the relevant nest gets sent to the separable function.

The new TPL file is `owls_sep.tpl`.

```
> sepdat=list(nobs=nrow(Owls),
      nnests=length(levels(Owls$Nest)),
      nf=ncol(mmf),
      fdesign=mmf,
      nests=as.integer(Owls$Nest),
      obs=response,
      Lbrood=LBrood)
> write_dat("owls_sep", L=sepdat)
```

The `DATA_SECTION` of the new tpl file looks like this:

```
DATA_SECTION
        init_int nobs    // # obs (=599)
        init_int nnests  // # random effects levels (=nests=27)
        init_int nf      // # fixed effects (incl intercept & interaction)
        init_matrix fdesign(1,nobs,1,nf) // Design matrix for fixed effects + intercept
        init_ivector nests(1,nobs)   // Nest indices
        init_vector obs(1,nobs)      // Response variable (# calls)
        init_vector Lbrood(1,nobs)   // Log of brood size (offset)
```

The `PARAMETER_SECTION` and par files are the same.

```
> write_pin("owls_sep", L=regpars)
```

The new `PROCEDURE_SECTION` is followed by two separable function definitions: one to calculate the negative log-likelihood of the observations, and another to calculate the negative log-likelihood of the random effects. These negative log-likelihoods get added to the joint negative log-likelihood `jnll`. Note that the definition of a `SEPARABLE_FUNCTION` must all be on one line, but it's split in this document to fit on the page.

```
PROCEDURE_SECTION
        jnll = 0.0;
        for(int i=1; i<=nobs; i++)
        {
                pois(i, fcoeffs, rcoeffs(nests(i)), logit_p, sigma);
        }
        for(int n=1; n<=nnests; n++)
        {
                rand(rcoeffs(n));
        }
SEPARABLE_FUNCTION void pois(int i, const dvar_vector& fcoeffs, const dvariable& r,
const dvariable& logit_p, const dvariable& sigma)

        dvariable p = exp(logit_p)/(1.0+exp(logit_p));
        dvariable eta = Lbrood(i) + fdesign(i)*fcoeffs + r*sigma;
        dvariable mu = exp(eta);

        if(obs(i)==0)
        {
                jnll-=log(p + (1-p)*exp(-mu) );
        }
        else//obs(i)>0
        {
                jnll-=log(1-p) + obs(i)*log(mu) - mu- gammln(obs(i)+1);
        }
SEPARABLE_FUNCTION void rand(const dvariable& r)
        jnll+=0.5*(r*r)+0.5*log(2.*M_PI); //for the random effects distributed N(0,1)
```

Then we compile and run the code from R and read back the results

```
> compile_admb("owls_sep", safe=FALSE, re=TRUE, verbose=FALSE)
> run_admb("owls_sep",verbose=FALSE,
 extra.args="-shess -noinit -nox -l1 40000000 -nl1 40000000 -ilmn 5")
> fit_admb_sep <- read_admb("owls_sep")
```

The extra argument `-shess` tells ADMB to use algorithms that are efficient for sparse Hessian matrices.

We get the same answer, at least up to a precision of $10^{-7}$:

```
> all.equal(coef(fit_admb),coef(fit_admb_sep),tol=1e-7)
```

```
[1] "Names: 6 string mismatches"
```

## 5 BUGS

```
1  model {
2
3    ## PRIORS
4    for (m in 1:5){
5      beta[m] ~ dnorm(0, 0.01)           # Linear effects
6    }
7    alpha ~ dnorm(0, .01)
8    sigma ~ dunif(0, 5)
9    tau <- 1/(sigma*sigma)
10   psi ~ dunif(0, 1)
11   for(j in 1:nnests){
12      a[j] ~ dnorm(0, tau)
13    }
14
15    for(i in 1:N){
16        SibNeg[i] ~ dpois(mu[i])
17        mu[i] <- lambda[i]*z[i]+0.00001 ## hack required for Rjags -- otherwise 'incompatible'
18        z[i] ~ dbern(psi)
19        log(lambda[i]) <- offset[i] + alpha + inprod(X[i,],beta) + a[nest[i]]
20    }
21 }
```

- lines 4–13 define the priors
  - (4–8) weak $N(\mu = 0, \tau = 0.01)$ for the fixed effect coefficients `beta` and the intercept `alpha` (remember that WinBUGS parameterizes the normal distribution in terms of the precision, or inverse variance);

  - (8–9) a uniform $(0, 5)$ distribution for the random-effects standard deviation `sigma` (which is converted to a precision `tau` on line 9);
  - (10) $U(0, 1)$ for the zero-inflation probability `psi`;
  - (11–13) and $N(0, \tau)$ priors for the per-nest random effects (these last values could be thought of as part of the model rather than as priors, since their variance is controlled by the parameter `sigma`).

- line 16 defines the conditional probability of observation `i` based on the conditional mean `mu[i]`
- line 17 defines `mu[i]`, which is either equal to zero (if the observation is a structural zero) or to the `lambda[i]` value relevant to observation `i`; `JAGS` complains if we ever have a non-zero observation from a Poisson with mean zero (because the likelihood is exactly zero), so we add a small fudge factor
- line 18 picks a Bernoulli (`dbern`) variable to decide whether the observation is a structural zero or not
- line 19 defines the expected mean of the sampling part of the distribution, based on the design matrix and the fixed effects (`inprod(X[i,],beta)`); the intercept and offset (`alpha+offset[i]`); and the random nest effect (`a[nest[i]]`).

For convenience, we packaged the R code to run the BUGS model (in JAGS) as a function.

```
> owls_BUGS_fit <- function(data, ni=25000, nb=2000, nt=10, nc=3) {
   ## Bundle data
   data$ArrivalTime <- scale(data$ArrivalTime,center=TRUE,scale=FALSE)
   fmm <- model.matrix(~(FoodTreatment+ArrivalTime)*SexParent,data=data)[,-1]
   bugs.data <- with(data,list(N=nrow(data),
                            nnests=length(levels(Nest)),
                            offset = logBroodSize,  ## nb specified on original scale
                            SibNeg = SiblingNegotiation,
                            nest = as.numeric(Nest),
                            X=fmm))

   ## Inits function
   inits <- function(){list(a=rnorm(27, 0, .5),
                           sigma=runif(1,0,.5), alpha=runif(1, 0, 2), beta = rnorm(5))}

   ## Parameters to estimate
   params <- c("alpha", "beta", "sigma", "psi")  ## DON'T save b, overdispersion latent vari
                                                  ## or a, nest random effect

   jags(data = bugs.data, inits = inits, parameters.to.save = params, model.file = "owls.txt
        n.thin = nt, n.chains = nc, n.burnin = nb, n.iter = ni)
}
```

The jags command (from the R2jags package) is compatible with the bugs command from the R2WinBUGS package; that should be all you have to change to run with WinBUGS instead of JAGS.

The only important point we had to take account specifically for a BUGS solution to the problem is the issue of centering the continuous predictor. Centering continuous predictors is generally a good idea for reasons of both interpretability and numerical stability (Gelman and Hill, 2006; Schielzeth, 2010). While deterministic, derivative- or linear-algebra-based approaches such as those implemented by lme4 in R or AD Model Builder often have built-in safeguards against strongly correlated parameters, centering is helpful in borderline cases or when convergence appears to fail. However, centering can be critically important for MCMC analyses: with a few exceptions (the optional glm module in JAGScan handle this case), mixing will be very slow for models with uncentered predictors.

```
> library(R2jags) ## loads 'rjags', 'R2WinBUGS', 'coda' as well

> ofit <- owls_BUGS_fit(Owls)
```

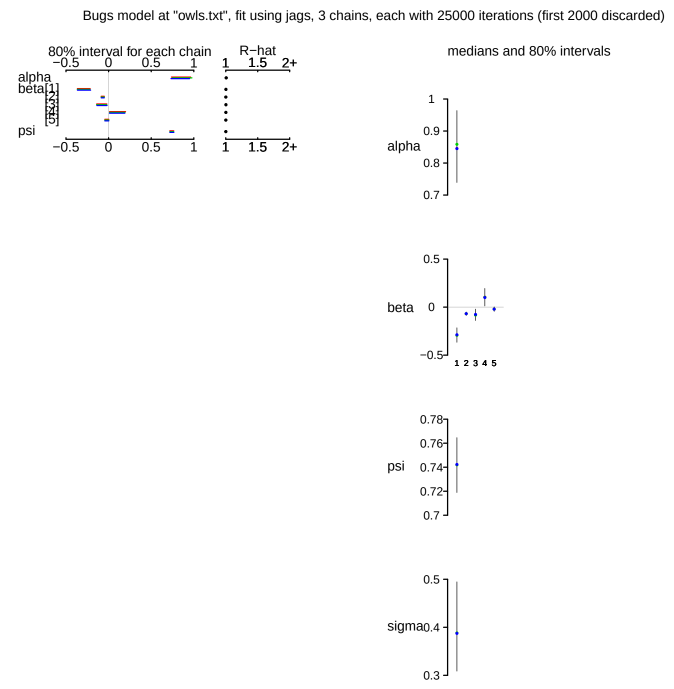
ofit is an object of class rjags. For a start, we can print it out:

```
> print(ofit,digits=2)

Inference for Bugs model at "owls.txt", fit using jags,
 3 chains, each with 25000 iterations (first 2000 discarded), n.thin = 10
 n.sims = 6900 iterations saved
         mu.vect sd.vect    2.5%     25%     50%     75%   97.5% Rhat n.eff
alpha       0.85    0.09    0.68    0.79    0.85    0.91    1.03 1.01   330
beta[1]    -0.29    0.06   -0.41   -0.33   -0.29   -0.25   -0.17 1.00  1300
beta[2]    -0.07    0.01   -0.10   -0.08   -0.07   -0.06   -0.04 1.00  6900
beta[3]    -0.08    0.05   -0.17   -0.11   -0.08   -0.05    0.01 1.00   990
beta[4]     0.10    0.07   -0.04    0.06    0.10    0.15    0.24 1.00  2400
beta[5]    -0.02    0.02   -0.06   -0.03   -0.02   -0.01    0.02 1.00  6900
psi         0.74    0.02    0.71    0.73    0.74    0.75    0.78 1.00  6900
sigma       0.40    0.07    0.28    0.34    0.39    0.44    0.57 1.00  6900
deviance 3262.99   15.45 3237.42 3251.94 3261.38 3272.43 3297.29 1.00  6400

For each parameter, n.eff is a crude measure of effective sample size,
and Rhat is the potential scale reduction factor (at convergence, Rhat=1).

DIC info (using the rule, pD = var(deviance)/2)
pD = 119.3 and DIC = 3382.3
DIC is an estimate of expected predictive error (lower deviance is better).
```

The results give information about the fit (number of chains, length of burn-in, number of samples, number saved) as well as estimated means, standard deviations, quantiles for each saved parameter. We use `print(ofit,digits=2)` to reduce the resolution a bit so the print-out fits on the page better. (By default, `R2jags` prints results with 3 digits of precision, while `R2WinBUGS` prints them with only 1 digit of precision: you can override either of these defaults by using `print` with an explicit `digits` option set.) In addition, the results contain two useful diagnostics, the Gelman-Rubin statistic (`Rhat`) and the effective size (`n.eff`). Because we have run multiple chains, we can use the (preferred) Gelman-Rubin test of convergence, which assesses whether the variance among and within chains is consistent with those chains sampling the same space (i.e., whether the chains have converged). The G-R statistics should be close to 1 (equal within- and between-chain variance), with a general rule of thumb that $\hat{R} < 1.2$ is acceptable. (In this case it looks like we might have done a bit of overkill, with the maximum `Rhat` value equal to 1.006.) The effective sample size corrects for autocorrelation in the chain, telling how many equivalent samples we would have if the chains were really independent. In this case the intercept parameter mixes most poorly ($n = 390$), while $\sigma$ is uncorrelated ($n = 6900$, which is equal to the total number of samples saved). In general $n > 1000$ is overkill, $n > 200$ is probably acceptable.

The `R2jags` package provides a `plot` method for `rjags` object, which falls back on the method defined in `R2WinBUGS` for `bugs` objects; `plot(ofit)` would give us a nice, rich plot that compares the chains, but the effect here is a bit ruined by the deviance, which is on a very different scale from the other variables. We extract the `BUGSoutput` component of the fit (which is what gets plotted anyway) and use a utility function `dropdev` (defined elsewhere) to drop the deviance component from the fit: We use a utility function

```
> source("../R/owls_R_funs.R")
> oo <- dropdev(ofit$BUGSoutput)
> plot(oo)
```



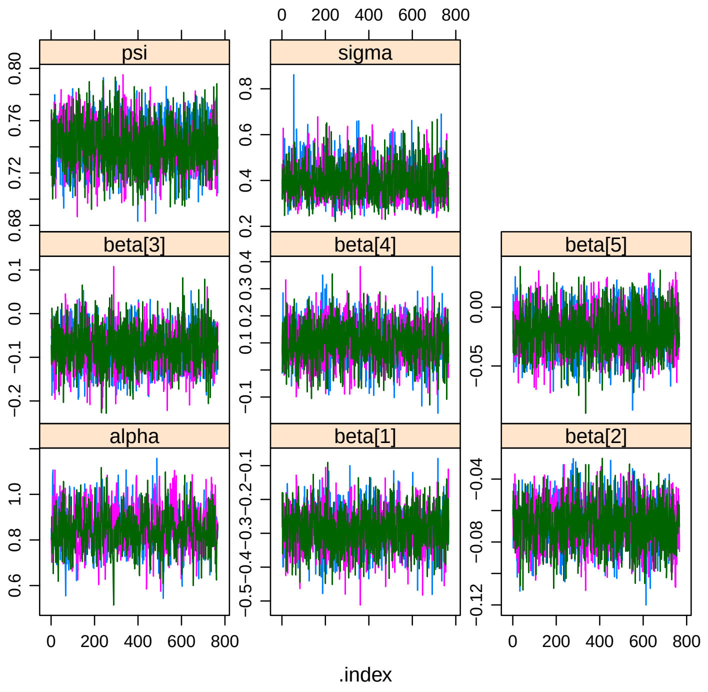
Adding the optional argument `display.parallel=TRUE` would compare the distributions of the estimated parameters across chains. This plot method is most useful if we have large vectors of parameters (e.g. if we had included the nest random effects in our list of parameters to save), and if the predictors have all been adjusted to have similar scales (e.g. if we had divided arrival time [our only continuous predictor variable] by its standard deviation). While putting all of the input variables into a model matrix is convenient from one point of view, it does limit the usefulness of our output because the names of the variables are `beta[1]`, `beta[2]`, . . . instead of being more informative. Note that `plot.bugs` displays the 50% credible intervals, not the 95% intervals, so these plots are not particularly useful for standard ("is it significant??") types of inference.

If we want more detail, or different types of plots, we can convert the `bugs` object we have extracted from the JAGS fit to an `mcmc` object (or an `mcmc.list` object in this case, since it contains multiple chains) and use plotting and diagnostic tools from the `coda` package:

```
> mm <- as.mcmc.list(oo)
```

As in the `MCMCglmm` case we would usually like to look at the trace plots: here the traces of the three different chains we ran are plotted together for each parameter.

```
> ## thin slightly for convenience
> mm2 <- as.mcmc.list(lapply(mm,function(x) {as.mcmc(x[seq(1,nrow(x),by=3),])}))
> print(xyplot(mm2,layout=c(3,3)))
```



We could plot density plots with `densityplot`, or run the Gelman-Rubin diagnostic with `gelman.diag`.

## 6 Combined summary

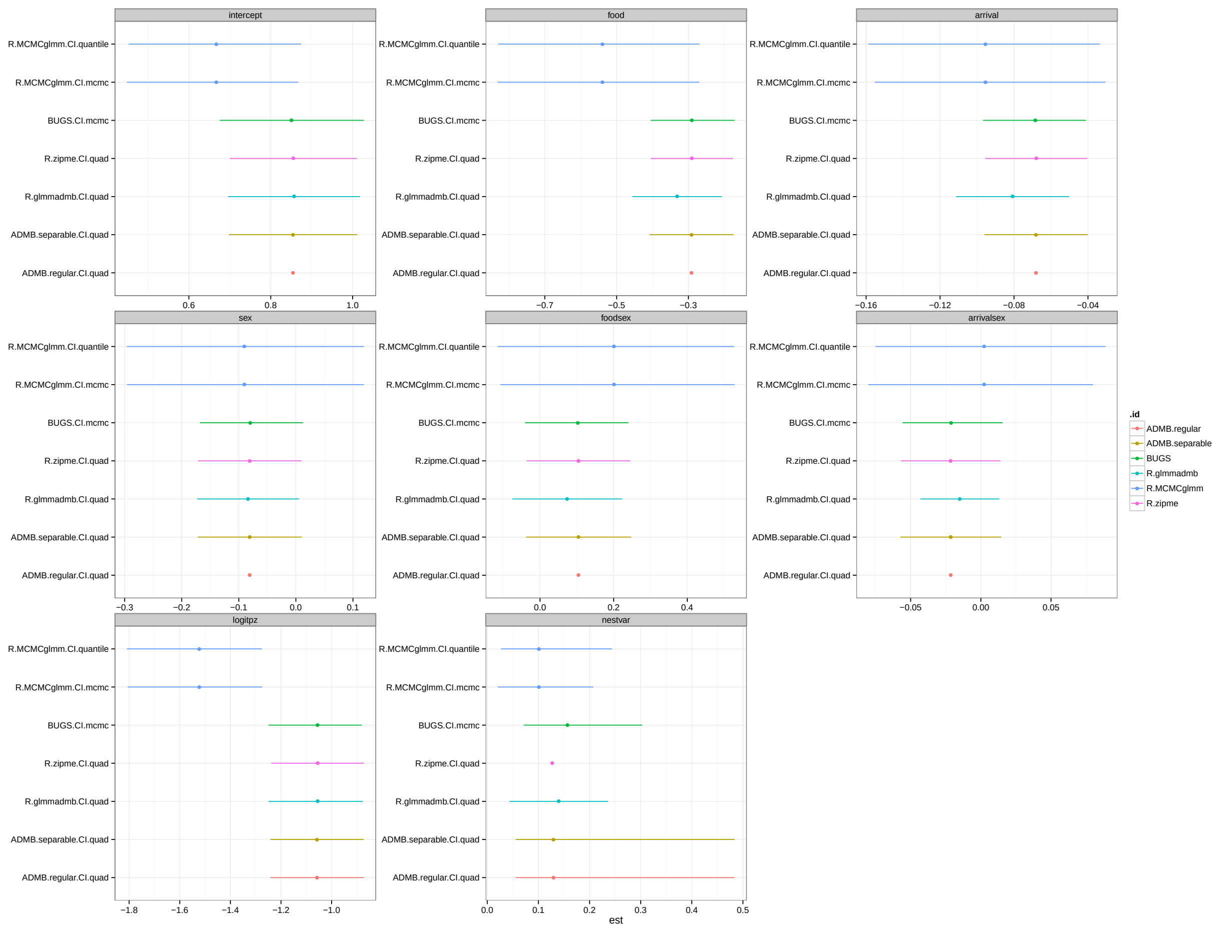
Finally, we will use some utility functions to summarize and compare all of the parameter estimates from the methods described, for model variants as close as we can get to the reference model (i.e. zero-inflated Poisson, log-brood-size offset).

```
> cwd <- setwd("../../TOOLS")
> source("tools.R")
> setwd(cwd)  ## restore
> library(reshape)
> library(ggplot2)
> ovals <- get_proj("owls",start.dir="../..")
```

```
owls
```

```
> b <- basesum(ovals,ncolparam=3)
```

```
> print(b$plots$params)
```



The MCMCglmm fit is the most noticeably different from all the rest, (presumably) because it is fitting a zero-inflated lognormal-Poisson (overdispersed) model rather than a zero-inflated Poisson model.

The next most obvious discrepancy is in the confidence intervals for the among-nest variance (`nestvar`). The ADMB fit was done on the standard deviation scale, and the confidence intervals then transformed by squaring. The more symmetric appearance of the MCMCglmm and BUGS confidence intervals (which are based on the posterior distribution of the variance, and hence are invariant across changes in parameter scales) suggest that the sampling/posterior distribution of the variance is actually more symmetric on the variance scale than on the standard deviation scale.

Beyond this, there are slight differences in the parameter estimates from glmmADMB, possibly because of some difference in stopping criteria. **?? could try restarting glmmADMB at consensus parameter values . . .**

Finally, we look at the timings of the various methods. The R methods are all in the same ballpark, along with the non-separable variant ADMB model (with E-M/zipme slightly faster than the others). BUGS is much slower ($\approx 12$ minutes), while the separable ADMB model is much faster ($\approx 4$ seconds):

We could use `print(b$times)` to look at the plot, but the values (times in seconds) are easy enough to read:

```
> ff <- subset(ovals$base,select=c(.id,time))
> ff$time <- round(ff$time,1)
> ff[order(ff$time),]
```

```
             .id  time
2 ADMB.separable   4.5
7        R.zipme  15.5
4     R.MCMCglmm  25.5
5     R.MCMCglmm  25.5
6     R.glmmadmb  27.9
1   ADMB.regular  31.7
3           BUGS 725.9
```

## References

Dempster, A., N. Laird, and D. Rubin (1977). Maximum likelihood from incomplete data via the EM algorithm (with discussion). *J. R. Stat. Soc. B 39*, 1–38.

Elston, D. A., R. Moss, T. Boulinier, C. Arrowsmith, and X. Lambin (2001). Analysis of aggregation, a worked example: numbers of ticks on red grouse chicks. *Parasitology 122*(5), 563–569.

Gelman, A. and J. Hill (2006). *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge, England: Cambridge University Press.

Hardin, J. W. and J. Hilbe (2007, February). *Generalized linear models and extensions*. Stata Press.

Lambert, D. (1992). Zero-inflated Poisson regression, with an application to defects in manufacturing. *Technometrics 34*, 1–14.

Minami, M., C. Lennert-Cody, W. Gao, and M. Roma¡n-Verdesoto (2007, April). Modeling shark bycatch: The zero-inflated negative binomial regression model with smoothing. *Fisheries Research 84*(2), 210–221.

Roulin, A. and L. Bersier (2007). Nestling barn owls beg more intensely in the presence of their mother than in the presence of their father. *Animal Behaviour 74*, 1099–1106.

Schielzeth, H. (2010). Simple means to improve the interpretability of regression coefficients. *Methods in Ecology and Evolution 1*, 103–113.

Spiegelhalter, D. J., N. Best, B. P. Carlin, and A. Van der Linde (2002). Bayesian measures of model complexity and fit. *Journal of the Royal Statistical Society B 64*, 583–640.

Zuur, A. F., E. N. Ieno, N. J. Walker, A. A. Saveliev, and G. M. Smith (2009, March). *Mixed Effects Models and Extensions in Ecology with R* (1 ed.). Springer.