

Using AD Model Builder and R together: getting started with the **R2admb** package

Ben Bolker

November 16, 2010

1 Introduction

AD Model Builder (ADMB: <http://admb-project.org>) is a standalone program, developed by Dave Fournier continuously since the 1980s and released as an open source project in 2007, that takes as input an objective function (typically a negative log-likelihood function) and outputs the coefficients that minimize the objective function, along with various auxiliary information. AD Model Builder uses *automatic differentiation* (that's what "AD" stands for), a powerful algorithm for computing the derivatives of a specified objective function efficiently and without the typical errors due to finite differencing. Because of this algorithm, and because the objective function is compiled into machine code before optimization, ADMB can solve large, difficult likelihood problems efficiently. ADMB also has the capability to fit random-effects models (via Laplace approximation).

To an R user, however, ADMB represents a challenge. The first (unavoidable) challenge is that the objective function needs to be written in a superset of C++; the second is learning the particular sequence of steps that need to be followed in order to output data in a suitable format for ADMB; compile and run the ADMB model; and read the data into R for analysis. The **R2admb** package aims to eliminate the second challenge by automating the R-ADMB interface as much as possible.

2 Installation

The **R2admb** package can be installed in R in the standard way (with `install.packages()` or via a Packages menu, depending on your platform: at the moment, since it's on the development platform r-forge, you'll have to say

```
> install.packages("R2admb", repos="http://r-forge.r-project.org")
```

You can also download the file and use R CMD INSTALL from the command line.

However, you'll also need to install ADMB: see one of the following links:

- <http://admb-project.org/>
- <http://admb-project.org/downloads>
- <http://code.google.com/p/admb-project/>
- <http://code.google.com/p/admb-project/downloads/list>

You may also need to install a C++ compiler (in particular, the MacOS installation instructions will probably ask you to install gcc/g++ from the Xcode package). You will need to have the scripts `admb`, `adcomp`, and `adlink` in the `bin` directory of your ADMB installation (I hope this Just Works once you have installed ADMB, but there's a chance that things will have to be tweaked).

3 Quick start (for the impatient)

1. Write the function that computes your negative log-likelihood function (see the ADMB manual, or below, for examples) and save it in a file with extension `.tpl` (hereafter “the TPL file”) in your working directory.
2. run `setup_admb()` to set up your ADMB environment appropriately.
3. run `do_admb(fn,data,params)`, where `fn` is the base name (without extension) of your TPL file, `data` is a list of the input data, and `params` is a list of the starting parameter values; if you want R to generate the PARAMETERS and DATA section of your TPL file automatically, use `do_admb(fn,data,params,checkparam="write",checkdata="write")`. Save the results as an R object.
4. use the standard R model accessor methods (`coef`, `summary`, `vcov`, `logLik`, `AIC`) to explore the results.

4 Basics

Here's a very simple example that can easily be done completely within R; we show how to do it with R2admb as well.

```
> library(R2admb)
> library(ggplot2) ## for pictures
```

The data are from Vonesh and Bolker (2005), describing the numbers of reed frog (*Hyperolius spinigularis*) tadpoles killed by predators as a function of size (TBL is total body length, Kill is the number killed out of 10 tadpoles exposed to predation). Figure 1 shows the data.

So if $p(\text{kill}) = c((S/d) \exp(1 - (S/d)))^g$ (a function for which the peak occurs at $S = d$, peak height= c) then a reasonable starting set of estimates would be $c = 0.45$, $d = 13$.

```
> ReedfrogSizepred <-
  data.frame(TBL = rep(c(9,12,21,25,37),each=3),
             Kill = c(0,2,1,3,4,5,0,0,0,0,1,0,0,0,0L))
```

Here is the code to fit a binomial model with mle2 using these starting points:

```
> library(bbmle)
> m0 <- mle2(Kill~dbinom(c*((TBL/d)*exp(1-TBL/d))~g,size=10),
             start=list(c=0.45,d=13,g=1),data=ReedfrogSizepred,
             method="L-BFGS-B",
             lower=c(c=0.003,d=10,g=0),
             upper=c(c=0.8,d=20,g=20),
             control=list(parscale=c(c=0.5,d=10,g=1)))
```

```
0.4138351 13.35082 18.24812
```

Generate predicted values:

```
> TBLvec = seq(9.5,36,length=100)
> predfr <-
  data.frame(TBL=TBLvec,
             Kill=predict(m0,newdata=data.frame(TBL=TBLvec)))
```

Here is a minimal TPL (AD Model Builder definition) file:

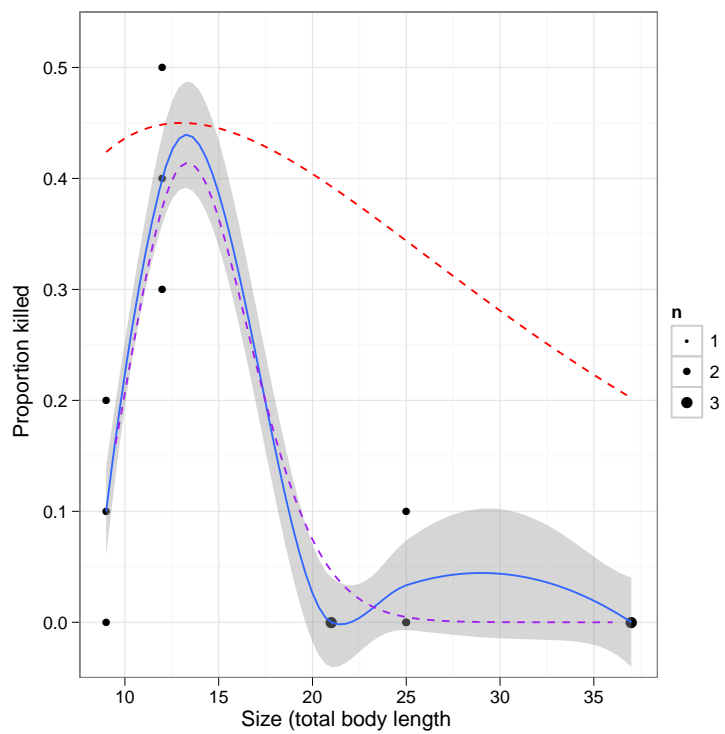


Figure 1: Proportions of reed frogs killed by predators, as a function of total body length in mm. Red: starting estimate.

```

1 PARAMETER_SECTION
2
3     vector prob(1,nobs)    // per capita mort prob
4
5 PROCEDURE_SECTION
6
7     dvariable fpen=0.0;      // penalty variable
8     // power-Ricker
9     prob = c*pow(elem_prod(TBL/d,exp(1-TBL/d)),g);
10    // penalties: constrain 0.001 <= prob <= 0.999
11    prob = posfun(prob,0.001,fpen);
12    f += 1000*fpen;
13    prob = 1-posfun(1-prob,0.001,fpen);
14    f += 1000*fpen;
15    // binomial negative log-likelihood
16    f -= sum( log_comb(nexposed,Kill)+
17             elem_prod(Kill,log(prob))+
18             elem_prod(nexposed-Kill,log(1-prob)));

```

- Comments are written in C++ format: everything on a line after `//` is ignored.
- lines 1–4 are the `PARAMETER` section; most of the parameters will get filled in automatically by `R2admb` based on the input parameters you specify, but you should include this section if you need to define any additional utility variables. In this case we define `prob` as a vector indexed from 1 to `nobs` (we will specify `nobs`, the number of observations, in our data list).
- most of the complexity of the `PROCEDURE` section (lines 7 and 11–14) has to do with making sure that the mortality probabilities do not exceed the range (0,1), which is not otherwise guaranteed by this model specification. Line 7 defines a utility variable `fpen`; lines 11–14 use the built-in ADMB function `posfun` to adjust low probabilities up to 0.001 (line 11) and high probabilities down to 0.999 (line 13), and add appropriate penalties to the negative log-likelihood to push the optimization away from these boundaries (lines 12 and 14).
- the rest of the `PROCEDURE` section simply computes the mortality probabilities as $c((S/d) \exp(1 - (S/d)))^g$ as specified above (line 9) and computes the binomial log-likelihood on the basis of these probabilities (lines 16–18). Because this is a log-likelihood and we want to

compute a negative log-likelihood, we *subtract* it from any penalty terms that have already accrued. The code is written in C++ syntax, using `=` rather than `<-` for assignment, `+=` to increment a variable and `-=` to decrement one. The power operator is `pow(x,y)` rather than `x^y`; elementwise multiplication of two vectors uses `elem_prod` rather than `*`.

To run this model, we save it in a text file called `ReedfrogSizepred0.tpl`; run `setup_admb()` to locate the AD Model Builder binaries and libraries on our system; and run `do_admb` with appropriate arguments.

```
> setup_admb()
> m1 <- do_admb("ReedfrogSizepred0",
               data=c(list(nobs=nrow(ReedfrogSizepred),
                           nexposed=rep(10,nrow(ReedfrogSizepred))),
                           ReedfrogSizepred),
               params=list(c=0.45,d=13,g=1),
               bounds=list(c=c(0,1),d=c(0,50),g=c(-1,25)),
               checkparam="write",
               checkdata="write")
> unlink(c("reedfrogsizedpred0.tpl",
           "reedfrogsizedpred0_gen.tpl",
           "reedfrogsizedpred0")) ## clean up leftovers
```

The `data`, `params`, and `bounds` (parameter bounds) arguments should be reasonably self-explanatory. When `checkparam="write"` and `checkdata="write"` are specified, `R2admb` attempts to write appropriate DATA and PARAMETER sections into a modified TPL file, leaving the results with the suffix `_gen.tpl` at the end of the run.

Here's the augmented file:

```
1 DATA_SECTION
2
3   init_int nobs
4   init_vector nexposed(1,15)
5   init_vector TBL(1,15)
6   init_vector Kill(1,15)
7
8 PARAMETER_SECTION
9
10  objective_function_value f
11  init_bounded_number c(0,1)
```

```

12   init_bounded_number d(0,50)
13   init_bounded_number g(-1,25)
14   vector prob(1,nobs)    // per capita mort prob
15 PROCEDURE_SECTION
16
17   dvariable fpen=0.0;      // penalty variable
18   // power-Ricker
19   prob = c*pow(elem_prod(TBL/d,exp(1-TBL/d)),g);
20   // penalties: constrain 0.001 <= prob <= 0.999
21   prob = posfun(prob,0.001,fpen);
22   f += 1000*fpen;
23   prob = 1-posfun(1-prob,0.001,fpen);
24   f += 1000*fpen;
25   // binomial negative log-likelihood
26   f -= sum( log_comb(nexposed,Kill)+
27            elem_prod(Kill,log(prob))+
28            elem_prod(nexposed-Kill,log(1-prob)));

```

Lines 1–7, 10–13 are new and should (I hope) be reasonably self-explanatory.

Now that we have fitted the model, here are some of the things we can do with it:

- Get basic information about the fit and coefficient estimates:

```
> m1
```

```

Model file: reedfrogsizepred0
Negative log-likelihood: 12.8938
Coefficients:
           c           d           g
0.4138331 13.3508215 18.2479066

```

- Get vector of coefficients only:

```
> coef(m1)
```

```

           c           d           g
0.4138331 13.3508215 18.2479066

```

- Get a coefficient table including standard errors and (approximate!!) *p* values:

```
> summary(m1)
```

```

Model file: reedfrogsizedpred0
Negative log-likelihood: 12.8938
Coefficients:
      Estimate Std. Error z value Pr(>|z|)
c    0.4138     0.1257   3.292 0.000996 ***
d   13.3508     0.8111  16.461 < 2e-16 ***
g   18.2479     6.0331   3.025 0.002489 **
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

```

(you can use `coef(summary(m1))` to extract just the table).

- Variance-covariance matrix of the parameters:

```

> vcov(m1)

           c           d           g
c 0.01580552 0.0578055 0.5043901
d 0.05780550 0.6578345 2.2464986
g 0.50439009 2.2464986 36.3982956

```

Log-likelihood, deviance, AIC:

```

> c(logLik(m1),deviance(m1),AIC(m1))

[1] -12.8938 25.7876 31.7876

```

4.1 Profiling

You can also ask ADMB to compute likelihood profiles for a model. If you code it yourself in the TPL file you need to add variables of type `likeprof_number` to keep track of the values: `R2admb` handles these details for you. You just need to specify `profile=TRUE` and give a list of the parameters you want profiled.

```

> m1P <- do_admb(fn,
  data=c(list(nobs=nrow(ReedfrogSizepred),
    nexposed=rep(10,nrow(ReedfrogSizepred))),
    ReedfrogSizepred),
  params=list(c=0.45,d=13,g=1),
  bounds=list(c=c(0,1),d=c(0,50),g=c(-1,25)),
  checkparam="write",

```



```

    checkdata="write",
    profile=TRUE,
    profpars=c("c","d","g"))

```

The profile information is stored in a list `m1P$prof` with entries for each variable to be profiled. Each entry in turn contains a list with elements `prof` (a 2-column matrix containing the parameter value and profile log-likelihood), `ci` (confidence intervals derived from the profile), `prof_norm` (a profile based on the normal approximation), and `ci_norm` (confidence intervals, ditto).

Let's compare ADMB's profiles to those generated from R:

```
> m0prof <- profile(m0)
```

```

12.998 15.1534
12.55568 11.44483
11.93926 7.116073
11.40261 2.930945
13.65335 20
13.91794 20
14.16285 20
14.38588 20
14.42789 20
14.44579 20
0.3673063 16.08703
0.3279814 13.46467
0.2957032 10.57105
0.2732866 8.06168
0.2603835 6.219009
0.2587049 5.920876
0.2572751 5.642461
0.2560763 5.382318
0.2552462 5.180081
0.4685466 20
0.500649 20
0.5162224 20
0.4276931 16.80234
0.2764666 10.15403
0.3533864 13.01403
0.2937065 12.41511

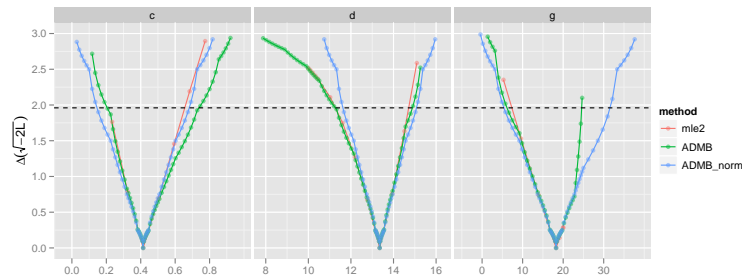
```

```

0.2445030 11.01926
0.1551972 10
0.4257947 13.40218
0.4377133 13.44989
0.4380248 13.45109

```

(A little bit of magic [hidden] gets everything into the same data frame and expressed in the same scale that R uses for profiles, which is the square root of the change in deviance ($-2L$) between the best fit and the profile: this scale provides a quick graphical assessment of the profile shape, because quadratic profiles will be V-shaped on this scale.)



Notice that R evaluates the profile at a smaller number of locations, using spline interpolation to compute confidence intervals.

4.2 MCMC

Another one of ADMB's features is that it can use Markov chain Monte Carlo (starting at the maximum likelihood estimate and using a candidate distribution based on the approximate sampling distribution of the parameters) to get more information about the uncertainty in the estimates. This procedure is especially helpful for complex models (high-dimensional or containing random effects) where likelihood profiling becomes problematic.

To use MCMC, just add `mcmc=TRUE` and specify the parameters you want to keep track of with `mcmcpars`:

```

> m1MC <- do_admb("ReedfrogSizepred",
  data=c(list(nobs=nrow(ReedfrogSizepred),
    nexposed=rep(10,nrow(ReedfrogSizepred))),
    ReedfrogSizepred),
  params=list(c=0.45,d=13,g=1),
  bounds=list(c=c(0,1),d=c(0,50),g=c(-1,25)),
  checkparam="write",

```

```

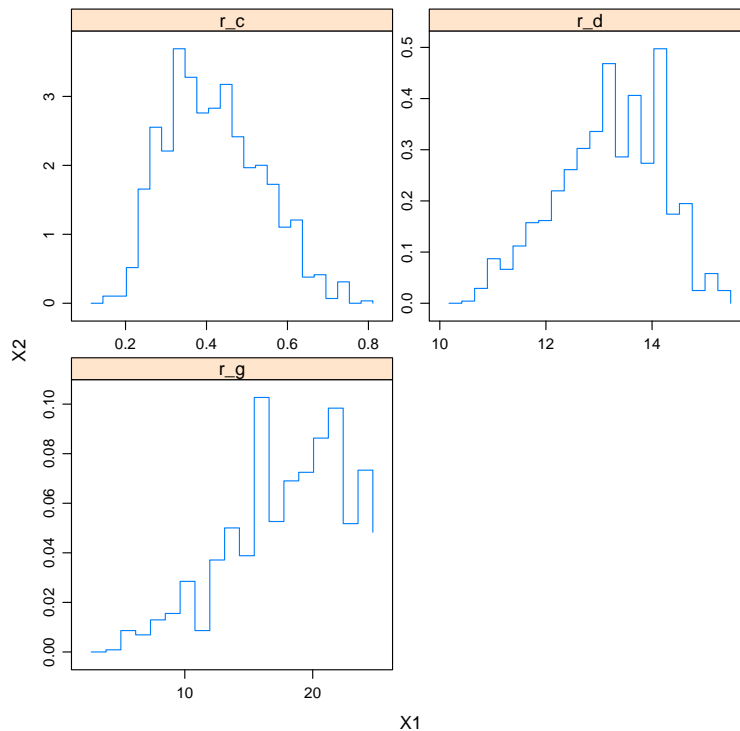
        checkdata="write",
        mcmc=TRUE,
        mcmcpars=c("c","d","g"))
> ## clean up leftovers:
> unlink(c("reedfrogsizpred0.tpl",
          "reedfrogsizpred0_gen.tpl",
          "reedfrogsizpred0"))

```

The output of MCMC is stored in two ways.

(1) ADMB internally computes a histogram of the MCMC sampled density; this is stored in a list element called `$hist`, as an object of class `admb_hist`. It has its own plot method:

```
> print(plot(m1MC$hist))
```



(2) Alternatively, the full set of samples is stored (as a data frame) in list element `$mcmc`. If you load the `coda` package, you can convert this into an object of class `mcmc`, and then use the various methods implemented in `coda` to analyze it.

```
> library(coda)
> mmc <- as.mcmc(m1MC$mcmc)
```

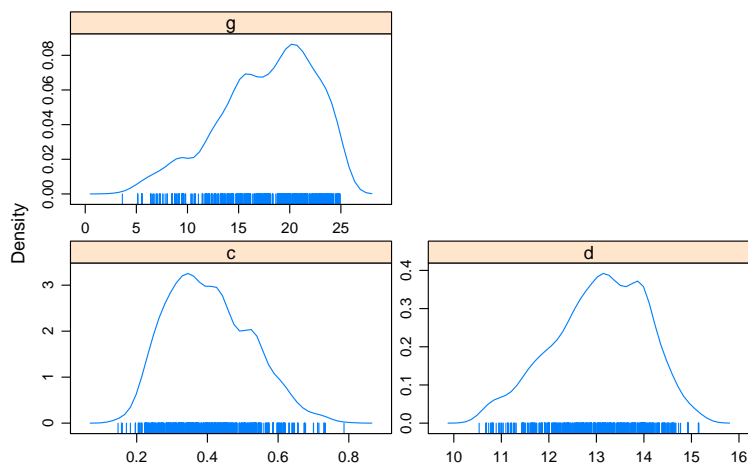
Highest posterior density (i.e. Bayesian credible) intervals:

```
> HPDinterval(mmc)

      lower      upper
c 0.2071513 0.6190827
d 10.9495253 14.6161983
g  8.7644415 24.8812550
attr("Probability")
[1] 0.95
```

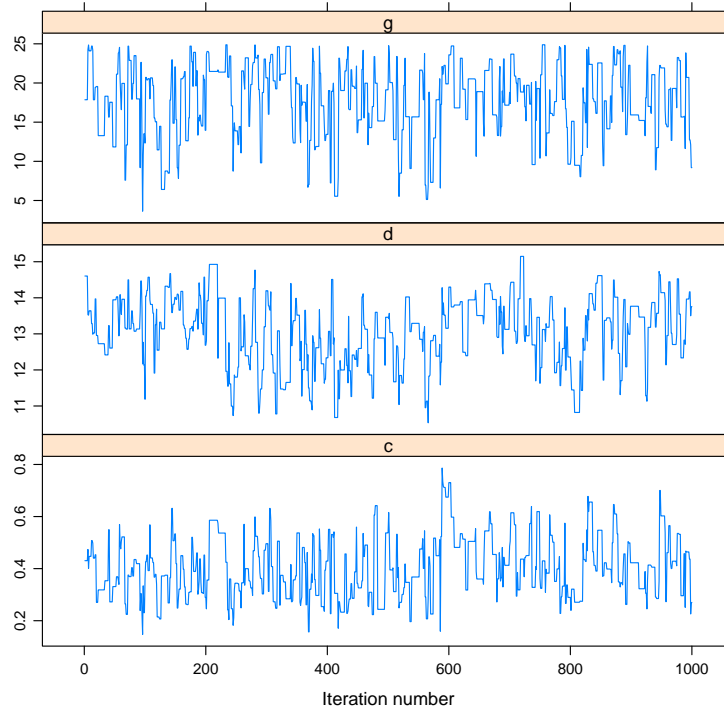
Density plots:

```
> print(densityplot(mmc))
```



Trace plots:

```
> print(xyplot(mmc))
```



See the documentation for the `coda` package. (You don't need to use `print` to see these plots in an interactive session — it's just required for generating documents.)

5 Incorporating random effects

One of ADMB's big advantages is the capability to fit flexible random-effects models — they need not fit within the generalized linear mixed model (GLMM) framework, they can use non-standard distributions, and so forth.

Here, however, we show a very basic example, one of the GLMM examples used in the `lme4` package.

Here's the `lme4` code to fit the model:

```
> library(lme4)
> gm1 <- glmer(cbind(incidence, size - incidence) ~ period + (1 | herd),
               family = binomial, data = cbpp)
```

To adapt this for ADMB, we first construct design matrices for the fixed and random effects:

```
> X <- model.matrix(~period,data=cbpp)
> Zherd <- model.matrix(~herd-1,data=cbpp)
```

Include these design matrices in the list of data to pass to ADMB:

```
> tmpdat <- list(X=X,Zherd=Zherd,
                 incidence=cbpp$incidence,size=cbpp$size,
                 nobs=nrow(cbpp))
```

Here is the bare-bones TPL file:

```
1  PARAMETER_SECTION
2
3      vector herdvec(1,nobs)
4      vector eta(1,nobs)
5      vector mu(1,nobs)
6
7  PROCEDURE_SECTION
8
9      herdvec = sigma_herd*(Zherd*u_herd);
10     eta = X*beta;                // form linear predictor
11     eta += herdvec;              // augment with random effects
12     mu = pow(1.0+exp(-eta),-1.0); // logistic transform
13     // binomial log-likelihood (unnormalized)
14     f -= sum(elem_prod(incidence,log(mu))+
15              elem_prod(size-incidence,log(1.0-mu)));
16
17     f+=0.5*norm2(u_herd); // log-prior (standard normal)
```

Only a few new things to note here:

- in the appropriate (matrix \times vector) context, `*` denotes matrix multiplication (rather than elementwise multiplication as in R)
- the random effects vector `u_herd` is unnormalized, i.e. drawn from a standard normal $N(0, 1)$. Line 9 constructs the vector of herd effects by (1) multiplying by the random-effects design matrix `Zherd` and (2) scaling by `sigma_herd`. (This approach is not very efficient, especially when the design matrix is sparse, but it's easy to code.)
- line 17 accounts for the random effects in the likelihood.

See the ADMB-RE manual (<http://admb-project.googlecode.com/files/admb-re.pdf>) for more detail.

The only changes in the `do_admb` call are that we have to specify `re=TRUE` (to turn on random effects) and we have to use the `re_vectors` argument to specify the names and lengths of each of the random effects vectors — only one (`u_herd`) in this case.

```
> d1 <- do_admb("toy1",
  data=tmpdat,
  re=TRUE,
  params=list(beta=rep(0,ncol(X)),sigma_herd=0.1),
  bounds=list(sigma_herd=c(0.0001,20)),
  re_vectors=c(u_herd=ncol(Zherd)),
  checkdata="write",checkparam="write",
  mcmc=TRUE,
  mcmcpars=c("beta","sigma_herd"))
```

Comparing `glmer` and `R2admb` results:

```
> coef(summary(gm1))
```

	Estimate	Std. Error	z value	Pr(> z)
(Intercept)	-1.3985351	0.2278906	-6.136871	8.416284e-10
period2	-0.9923347	0.3053852	-3.249452	1.156274e-03
period3	-1.1286754	0.3260491	-3.461673	5.368286e-04
period4	-1.5803739	0.4288037	-3.685542	2.282169e-04

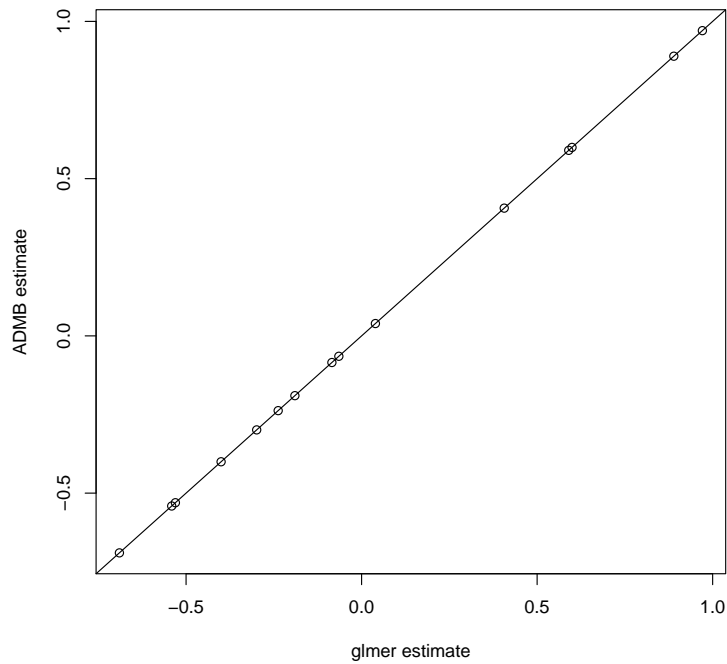
```
> coef(summary(d1))[1:5,]
```

	Estimate	Std. Error	z value	Pr(> z)
beta1	-1.3985300	0.23247	-6.015959	1.788248e-09
beta2	-0.9923330	0.30664	-3.236150	1.211538e-03
beta3	-1.1286700	0.32664	-3.455394	5.494886e-04
beta4	-1.5803100	0.42744	-3.697150	2.180331e-04
sigma_herd	0.6422615	0.17871	3.593875	3.257956e-04

(The full table would include the estimates of the random effects as well.)

Confirm that the random effects estimates are the same (note that the ADMB estimates are not scaled by the estimated standard deviation, so we do that by hand).

```
> plot(ranef(gm1)$herd[,1],coef(d1)[6:20]*coef(d1)["sigma_herd"],
  xlab="glmer estimate",ylab="ADMB estimate")
> abline(a=0,b=1)
```



We can get confidence (credible) intervals based on the MCMC run:

```
> detach("package:lme4") ## HPDinterval definition gets in the way
> HPDinterval(as.mcmc(d1$mcmc[,6:20]))
```

	lower	upper
u_herd01	0.26040589	1.206323475
u_herd02	-0.83294218	0.006624788
u_herd03	0.02043034	0.981741058
u_herd04	-0.29279222	0.377716760
u_herd05	-0.70323283	0.140222764
u_herd06	-1.08272994	-0.204100416
u_herd07	0.66292084	1.830446174
u_herd08	0.24791360	1.339227267
u_herd09	-0.65077048	-0.033571146
u_herd10	-1.34797903	-0.418856814
u_herd11	-0.58846759	0.327037459
u_herd12	-0.42418903	0.219284561
u_herd13	-1.39103985	-0.472251939


```
u_herd14  0.82725388  1.967255524
u_herd15 -1.28799325 -0.434780118
attr(,"Probability")
[1] 0.95
```

That's all for now.

References

Vonesh, J. R. and B. M. Bolker. 2005. Compensatory larval responses shift tradeoffs associated with predator-induced hatching plasticity. *Ecology* **86**:1580–1591.