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

Ben Bolker

May 1, 2013

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 platorm: at the moment, since it's on the development platform R-forge, you'll have to use

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

to install it. 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`

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)

3.1 For non-ADMB users

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, run.opts = run.control(checkparam = "write", checkdata
   ```

   Assign the result to an R object.
4. use the standard R model accessor methods (`coef`, `summary`, `vcov`, `logLik`, `AIC` (etc.)) to explore the results.

3.2 For ADMB users

If you are already familiar with ADMB (e.g. you already have your TPL files written with appropriate PARAMETERS and DATA sections), or if you prefer a more granular approach to controlling ADMB (for example, if you are going to compile a TPL file once and then run it for lots of different sets of input parameters), you can instead use `R2admb` as follows:

1. Write your TPL file, set up your input and data files.
2. `setup_admb()` as above.
3. `compile_admb(fn)` to compile your TPL file, specifying `re=TRUE` if the model has random effects (or do this outside R)
4. `run_admb(fn)` to run the executable
5. `results <- read_admb(fn)` to read (and save) the results
6. `clean_admb(fn)` to clean up the files that have been generated
7. as before, use the standard R model accessor methods to explore the results.

There are more steps this way, but you have a bit more control of the process.

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

## Loading required package:  methods

theme_set(theme_bw())
```

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")

## Loading required package:  stats4
##
## Attaching package:  'bbmle'
## The following object is masked from 'package:R2admb':
##
##    stdEr

m0 <- mle2(Kill ~ dbinom(c * ((TBL/d) * exp(1 - TBL/d))^g, size = 10), start = list(c
    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
    d = 10, g = 1)))
```

Generate predicted values:

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

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

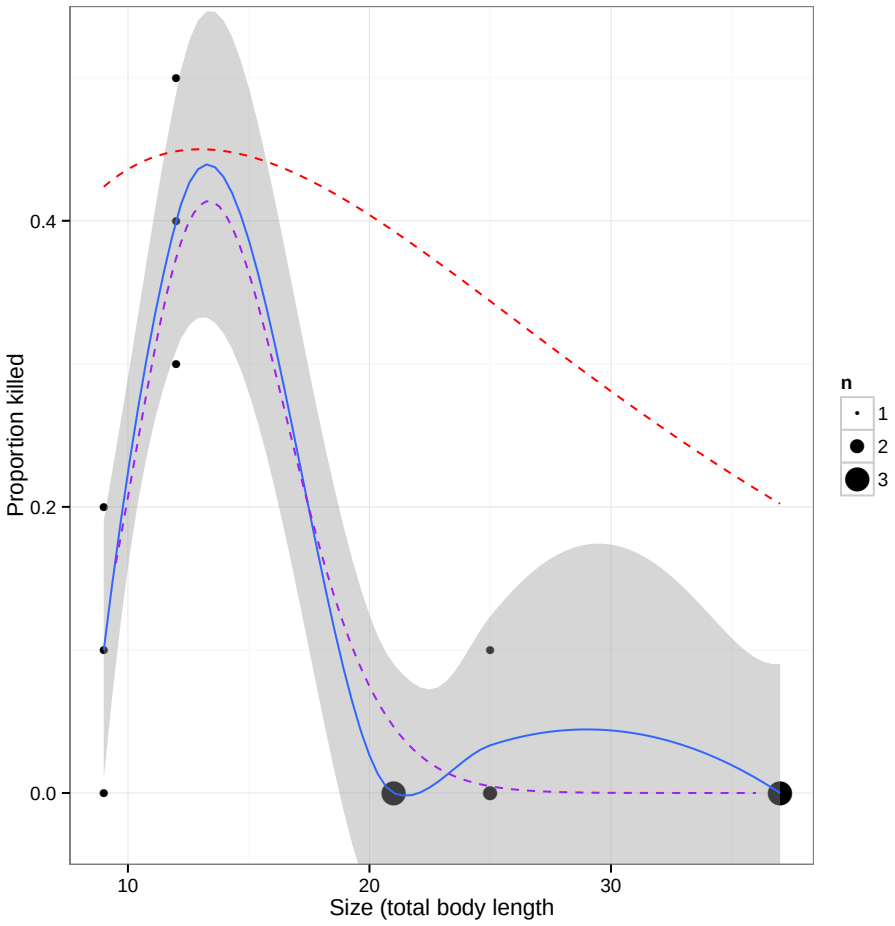


Figure 1: Proportions of reed frogs killed by predators, as a function of total body length in mm. Red: starting estimate.

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

```
## load in data from previously executed runs
load_doc <- function(x) {
    load(system.file("doc", x, package = "R2admb"), envir = .GlobalEnv)  ##
    ## would like parent.frame(2)) ...
}
zz <- load_doc("Reedfrog_runs.RData")
```

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()
```

```
rfs_params <- list(c = 0.45, d = 13, g = 1)  ## starting parameters
rfs_bounds <- list(c = c(0, 1), d = c(0, 50), g = c(-1, 25))  ## bounds
rfs_dat <- c(list(nobs = nrow(ReedfrogSizepred), nexposed = rep(10, nrow(ReedfrogSize
    ReedfrogSizepred)
```

```
m1 <- do_admb("ReedfrogSizepred0", data = rfs_dat, params = rfs_params, bounds = rfs_
    run.opts = run.control(checkparam = "write", checkdata = "write"))
unlink(c("reedfrogsizepred0.tpl", "reedfrogsizepred0_gen.tpl", "reedfrogsizepred0"))
```

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:

```
DATA_SECTION

  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_number c
12  init_number d
13  init_number g
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_gen
## Negative log-likelihood: 12.89
## Coefficients:
##       c       d       g
##  0.4138 13.3508 18.2478
```

- Get vector of coefficients only:

```
coef(m1)

##       c       d       g
##  0.4138 13.3508 18.2478
```

- Get a coefficient table including standard errors and p values. (The p values provided are from a Wald test, which is based on an assumption that the log-likelihood surface is quadratic. Use them with caution.)

```
summary(m1)

## Model file: ReedfrogSizepred0_gen
## Negative log-likelihood:  12.9   AIC:  19.8
## Coefficients:
##   Estimate Std. Error z value Pr(>|z|)
## c    0.414      0.126    3.29   0.0010 ***
## d   13.351      0.811   16.46   <2e-16 ***
## g   18.248      6.033    3.02   0.0025 **
## ---
## 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 0.01581 0.05781  0.5044
## d 0.05781 0.65783  2.2465
## g 0.50439 2.24650 36.3983
```

Log-likelihood, deviance, AIC:

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

## [1] -12.89  31.79
```

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("ReedfrogSizepred0", data = c(list(nobs = nrow(ReedfrogSizepred),
    nexposed = rep(10, nrow(ReedfrogSizepred))), ReedfrogSizepred), params = rfs_para
    bounds = rfs_bounds, run.opts = run.control(checkparam = "write", checkdata = "wr
    profile = TRUE, profpars = c("c", "d", "g"))
```

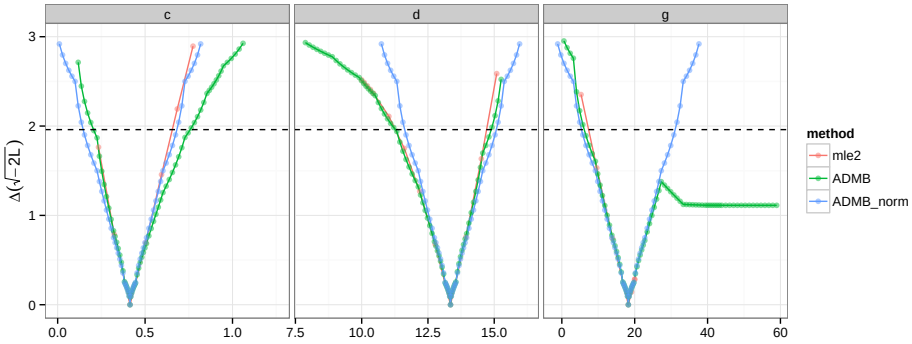
The profile information is stored in a list `m1P$prof` with entries for each variable to be profiled. Each entry in turn contains a list with elements `prof` (a 2-column matrix containing the parameter value and profile log-likelihood), `ci` (confidence intervals derived from the profile), `prof_norm`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

(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.)

```
## Warning:  Removed 44 rows containing missing values (geom_path).
## Warning:  Removed 35 rows containing missing values (geom_path).
## Warning:  Removed 28 rows containing missing values (geom_path).
## Warning:  Removed 44 rows containing missing values (geom_point).
## Warning:  Removed 35 rows containing missing values (geom_point).
## Warning:  Removed 28 rows containing missing values (geom_point).
```



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 for which you want histograms [see below] via `mcmcpars` (you must specify at least one).

```
m1MC <- do_admb("ReedfrogSizepred0", data = rfs_dat, params = rfs_params, bounds = rf
    run.opts = run.control(checkparam = "write", checkdata = "write"), mcmc = TRUE,
    mcmc.opts = mcmc.control(mcmcpars = c("c", "d", "g")))
## clean up leftovers:
unlink(c("reedfrogsizepred0.tpl", "reedfrogsizepred0_gen.tpl", "reedfrogsizepred0"))
```

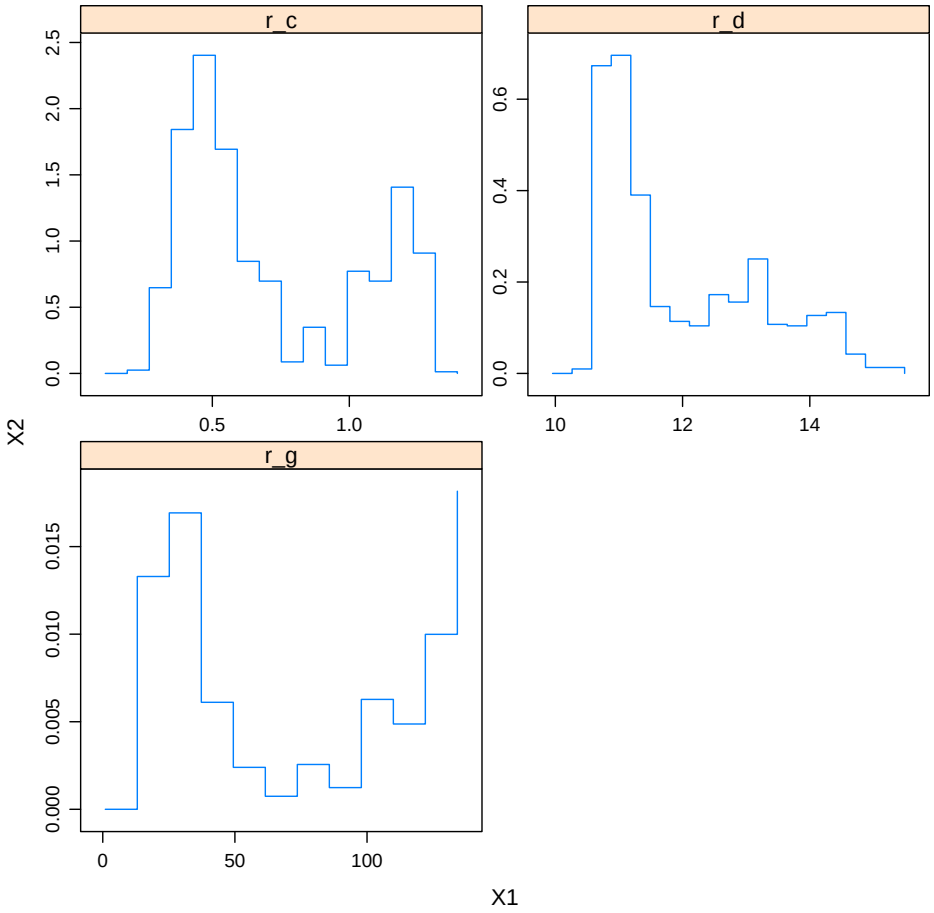
The output of MCMC is stored in two ways.

(1) ADMB internally computes a histogram of the MCMC sampled densities, for `sdreport` parameters only (if you don't know what these are, that's OK — appropriate parameters are auto-generated when you specify `mcmcpars`). This information is stored in a list element called `$hist`, as an object of class `admb_hist`.

It has its own plot method:

```
plot(m1MC$hist)

## Loading required package:  lattice
```

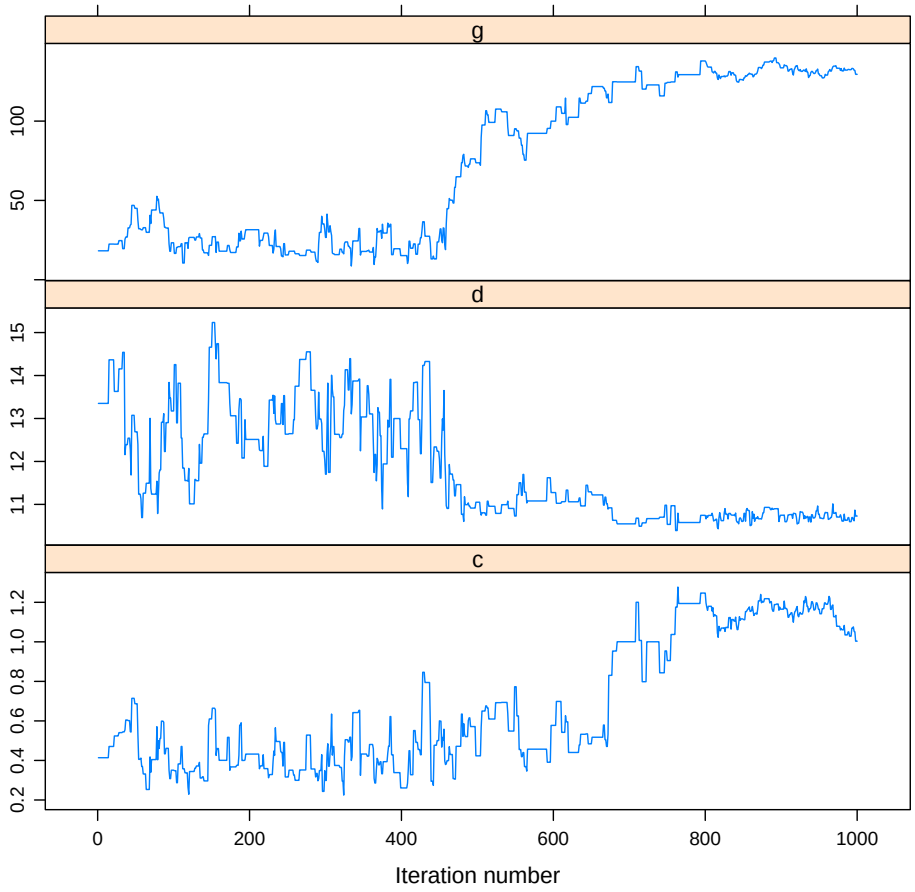


(2) In addition the full set of samples, sampled as frequently as specified in `mcsave` (by default, the values are sampled at a frequency that gives a total of 1000 samples for the full run) is stored as a data frame in list element `$mcmc`. If you load the `coda` package, you can convert this into an object of class `mcmc`, and then use the various methods implemented in `coda` to analyze it.

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

Trace plots give a graphical diagnostic of the behavior of the MCMC chain. In this case the diagnostics are *not* good — you should be looking for traces that essentially look like white noise.

```
xyplot(mmc)
```



(for larger sets of parameters you may want to specify a layout other than the default 1-row-by-n-columns, e.g. `xyplot(mmc,layout=c(2,2))`).

If you want a numerical summary of the chain behavior you can use `raftery.diag` or `geweke.diag` (the most common diagnostic, the Gelman-

Rubin statistic (`gelman.diag`) doesn't work here because it requires multiple chains and ADMB only runs a single chain):

```
raftery.diag(mmc)

##
## Quantile (q) = 0.025
## Accuracy (r) = +/- 0.005
## Probability (s) = 0.95
##
## You need a sample size of at least 3746 with these values of q, r and s

geweke.diag(mmc)

##
## Fraction in 1st window = 0.1
## Fraction in 2nd window = 0.5
##
##      c      d      g
## -2.615  3.451 -9.293
```

`geweke.diag` returns Z scores for the equality of (by default) the first 10% and the last 50% of the chain. For example, the value of -2.615 here is slightly high: `pnorm(abs(v),lower.tail=FALSE)*2` computes a two-tailed Z-test (with p value 0.009 in this case).

You can also compute the effective size of the sample, i.e. corrected for autocorrelation:

```
effectiveSize(mmc)

##      c      d      g
##  3.423 10.410  1.625
```

This value should be at least 100, and probably greater than 200, for reasonable estimation of confidence intervals.

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

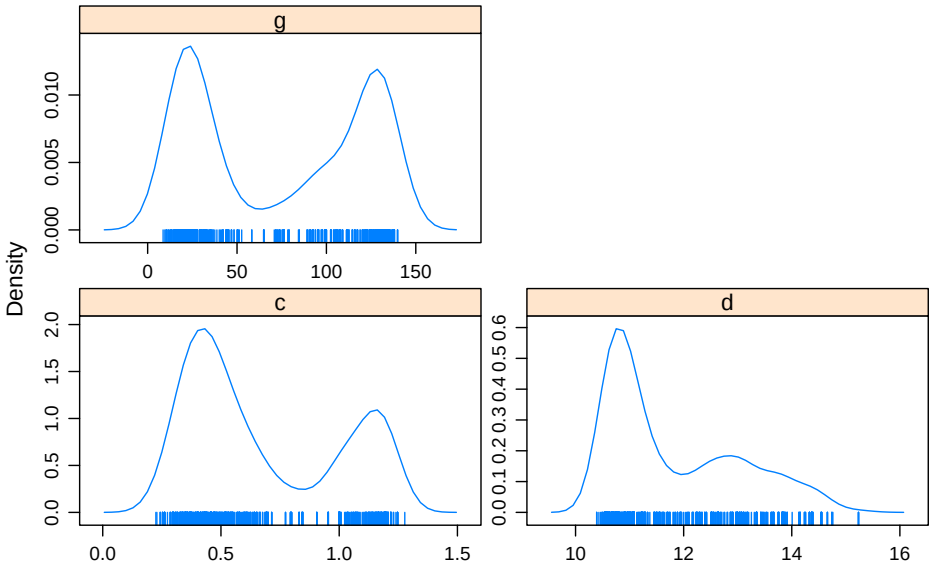
```
HPDinterval(mmc)

##     lower   upper
## c  0.2952   1.208
```

```
## d 10.5075  14.327
## g 14.9532 135.716
## attr(,"Probability")
## [1] 0.95
```

Density plots show you the estimated posterior density of the variables:

```
densityplot(mmc)
```



See the documentation for the `coda` package for more information about these methods. (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)
if (as.numeric(R.version$major) < 3) {
    ## FIXME
    gm1 <- glmer(cbind(incidence, size - incidence) ~ period + (1 | herd), family = b
        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 × 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.

```
zz2 <- load_doc("toy1_runs.RData")
```

The only changes in the `do_admb` call are that we have to use the `re` 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, params = list(beta = rep(0, ncol(X)), sigma_herd
    bounds = list(sigma_herd = c(1e-04, 20)), re = list(u_herd = ncol(Zherd)),
    run.opts = run.control(checkdata = "write", checkparam = "write"), mcmc = TRUE,
    mcmc.opts = mcmc.control(mcmc = 20, mcmcpars = c("beta", "sigma_herd")))
```

Comparing `glmer` and `R2admb` results:

```
## FIXME coef(summary(gm1))
```

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

##            Estimate Std. Error z value  Pr(>|z|)
## beta1       -1.3985     0.2325  -6.016 1.788e-09
## beta2       -0.9923     0.3066  -3.236 1.212e-03
## beta3       -1.1287     0.3266  -3.455 5.495e-04
## beta4       -1.5803     0.4274  -3.697 2.180e-04
## sigma_herd   0.6423     0.1787   3.594 3.258e-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).

```
## FIXME 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  0.60038  0.91801
## u_herd02 -0.48658 -0.38510
## u_herd03  0.36705  0.63192
## u_herd04 -0.02227  0.06114
## u_herd05 -0.34290 -0.22529
## u_herd06 -0.69625 -0.51892
## u_herd07  0.92868  1.38363
```

```
## u_herd08  0.54744  0.93242
## u_herd09 -0.44171 -0.35540
## u_herd10 -0.88836 -0.70766
## u_herd11 -0.23191 -0.11277
## u_herd12 -0.16964 -0.10091
## u_herd13 -1.11149 -0.91426
## u_herd14  1.04312  1.51039
## u_herd15 -0.87400 -0.72084
## 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.