Tadpole writeup

Ben Bolker

July 30, 2012

Contents

1 Summary

This is a relatively simple nonlinear fitting problem. With reasonable starting values (which are fairly easy to guess by direct graphical examination of the data), none of the approaches is problematic. This example is a good introduction to the basics of input (formatting and data), running, and output (retrieving results) using the different approaches.

- ADMB works fastest and has the most reliable 'basic' (quadratic) confidence intervals.

- With box constraints set, the standard optimizers in R also work fine, and quickly.
- BUGS (here using JAGS) is much slower but still, for this simple problem, pretty quick and robust, although there is no clear advantage to using BUGS for this problem (unless e.g. one wanted to introduce informative priors).

2 Introduction

The data are originally from Vonesh and Bolker (2005) (these data are also described and analyzed in Bolker (2008)), describing the numbers of reed frog (*Hyperolius spinigularis*) tadpoles killed by predators as a function of size in a small-scale field trial. (`TBL` is total body length in mm, `Kill` is the number killed out of 10 tadpoles exposed to predation). Figure 1 shows the data.

Our main interest is in a quantitative description of the "window of vulnerability" — the unimodal pattern of proportion killed as a function of size. In various contexts, we can use this description either to describe and test differences among treatments (e.g., does the window of vulnerability differ by predator size, or with tadpoles exposed to different predator cues?) or to project the effects of growth and mortality rates through a life stage (see references above and McCoy et al. (2011) for more details and examples).

This is one of the most basic examples developed by the NCEAS nonlinear modeling working group, with few of the more challenging components incorporated in the more complex examples. In particular, the data are

- small (there were 3 trials for each of 6 size classes);
- well-behaved (no missing data, no extreme outliers);
- fully specified (no observation errors, latent variables, or random effects);
- low-dimensional (a single predictor variable (TBL) and a single response (number killed);
- easy to describe mechanistically (it is reasonable to assume a binomial distribution or some variant of it).

3 Basics

Load all the packages we'll need, up front (not all of these are absolutely necessary, but it will be most convenient to make sure you have them installed now).

```
> library(ggplot2)    ## pictures
> theme_update(theme_bw()) ## I dislike the default gray background
> library(bbmle)      ## MLE fitting in R (wrapper)
> library(optimx)     ## additional R optimizers
> library(MCMCpack)   ## for post-hoc MCMC in R
> library(coda)       ## analysis of MCMC runs (R, BUGS, ADMB)
> library(scapeMCMC)  ## prettier MCMC diagnostics
> library(R2admb)     ## R interface to AD Model Builder
> library(R2jags)     ## R interface to JAGS (BUGS dialect)
> source("../R/tadpole_R_funs.R")
```

This version uses R Under development (unstable) (2012-07-27 r60007) and package versions:

```
     bbmle       coda    ggplot2   MCMCpack     optimx     R2admb     R2jags
   1.0.5.1     0.15-1      0.9.1      1.2-4 2012.04.01    0.7.5.3    0.03-07
```

The data are very simple:

```
> (ReedfrogSizepred <- read.table("../DATA/tadpole.dat"))
```

```
   TBL Kill Exposed
1    9    0      10
2    9    2      10
3    9    1      10
4   12    3      10
5   12    4      10
6   12    5      10
7   21    0      10
8   21    0      10
9   21    0      10
10  25    0      10
11  25    1      10
12  25    0      10
13  37    0      10
14  37    0      10
15  37    0      10
```

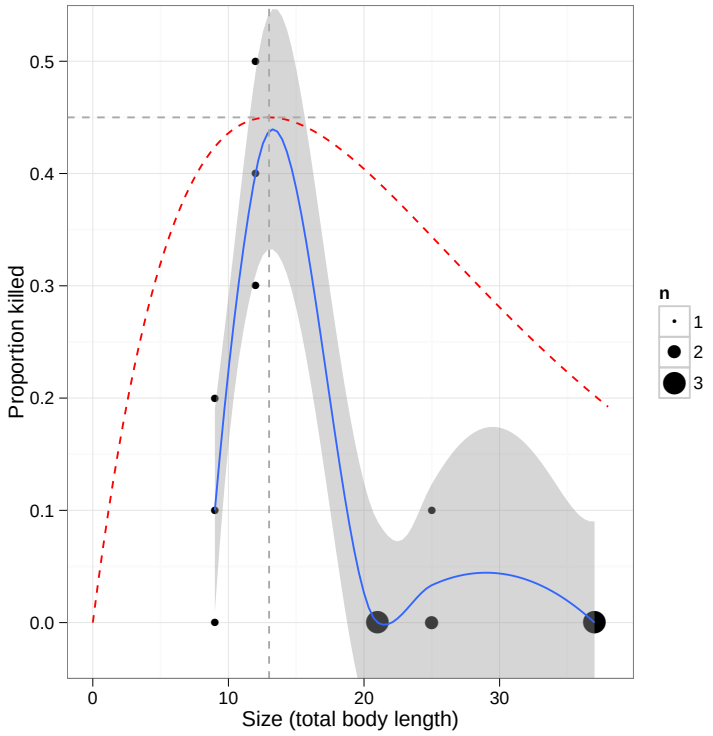


Figure 1: Proportions of reed frogs killed by predators, as a function of total body length in mm. Red: starting estimate. Blue/gray fill: nonparametric (loess) fit.

We need a nonlinear function that is flexible enough to increase, decrease, and change its shape. We chose

$$P(\text{kill}) = c((S/d)\exp(1-(S/d)))^g, \tag{1}$$

sometimes called the "power-Ricker" function (in its simplest form, it is a variant of the Ricker function $y = x\exp(-x)$ raised to a power: $y = (x\exp(-x))^g$. The d and c parameters adjust the scale of the function in the x and y directions respectively. A bit of calculation or numerical experimentation will show that this function is equal to zero when $S = 0$; declines to zero as S gets large; initially increases as $c(S/d)^g$ (e.g. linearly if $g = 1$ (Ricker), quadratically if $g = 2$); and peaks when $S = d$ at a height c. [1]. Looking at the plot (a luxury we have in the low-dimensional case), we can see that a reasonable starting set of estimates would be

- $g = 1$ (the Ricker model; linear increase near $S = 0$);
- $c = 0.45$,
- $d = 13$

(Figure 1 illustrates the initial values).

The remainder of the model formulation describes the stochastic part of the model. Although it is slightly questionable whether tadpoles within a tank are identically vulnerable and killed independently of each other, we will go ahead and use a binomial model anyway:

$$\text{Killed} \sim \text{Binomial}(P(\text{kill}), N) \tag{2}$$

4 R

4.1 Fitting

In R we'll use the `mle2` function from the `bbmle` package. There is no particular computational advantage to `mle2` over the various optimizers in R (`optim`, `nlm`, and `nlminb` in base R, or the variants found in the `optimx` package) — it is just a wrapper that provides various conveniences (profiling, confidence intervals, predictions, etc.) for maximum likelihood estimation problems.

[1] A different form of "generalized Ricker" was used in Vonesh and Bolker (2005); as described by Bolker (2008)), that turned out to be a poor choice, because the same data could be described in quite different ways by a single function (i.e., the likelihood surface has multiple maxima.

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

```
> library(bbmle)
> tadpole_R_fit

function (data = ReedfrogSizepred, start = list(c = 0.45, d = 13,
    g = 1))
{
    mle2(Kill ~ dbinom(c * ((TBL/d) * exp(1 - TBL/d))^g, size = Exposed),
        start = start, data = data, method = "L-BFGS-B", lower = c(c = 0.003,
            d = 10, g = 0), upper = c(c = 0.8, d = 20, g = 50),
        control = list(parscale = c(c = 0.5, d = 10, g = 1)))
}

> mle2_fit <- tadpole_R_fit()
```

Notes:

- It is not absolutely necessary to set bounds on the optimization in this case, but it is generally good practice and will help avoid many problems when the optimizer wants to wander off to strange values on its way to its ultimate best solution (which may be perfectly sensible).
- The tradeoff for setting bounds is that there is a more limited set of optimizers available. I used `L-BFGS-B`, which is available within `optim`, but is finicky
  - it may fail if your parameters are not scaled appropriately (which you can do by hand by redefining your parameters so they all have an expected magnitude between 1 and 10, or as above by setting `parscale` to the approximate magnitude of each parameter);
  - it will fail if it encounters a non-finite value (`Inf`, `NA`, or `NaN`) when calling the negative log-likelihood function;
  - it sometimes steps slightly over the specified lower/upper bounds when computing the derivatives of the function, so it is best to set these slightly in from any set of parameter values that will return a non-finite value (e.g. set the boundary for c at 0.002 or above rather than exactly at zero so the probability always stays positive).

Alternatives to `L-BFGS-B` within R include `nlminb` (base R) and `bobyqa` (derivative-free, in `optimx`) [neither supports `parscale`]. In a quick test, `nlminb` and `L-BFGS-B` were approximately the same speed, while `bobyqa` (which may be more robust in some situations) was about 12 times slower ...

Printing out the fit gives just the original function call, the coefficients, and the log-likelihood:

```
> mle2_fit

Call:
mle2(minuslogl = Kill ~ dbinom(c * ((TBL/d) * exp(1 - TBL/d))^g,
    size = Exposed), start = start, method = "L-BFGS-B", data = data,
    lower = c(c = 0.003, d = 10, g = 0), upper = c(c = 0.8, d = 20,
        g = 50), control = list(parscale = c(c = 0.5, d = 10,
        g = 1)))

Coefficients:
         c          d          g
 0.4138351 13.3508185 18.2481176

Log-likelihood: -12.88
```

`summary(...)` gives estimates, approximate standard errors (based on a quadratic approximation to the likelihood surface), Z values (estimate/std. error), and *Wald p-values* against the null hypotheses that each parameter is separately equal to zero (based again on a quadratic approximation). `coef(summary(...))` will extract the table printed here.

```
> summary(mle2_fit)

Maximum likelihood estimation

Call:
mle2(minuslogl = Kill ~ dbinom(c * ((TBL/d) * exp(1 - TBL/d))^g,
    size = Exposed), start = start, method = "L-BFGS-B", data = data,
    lower = c(c = 0.003, d = 10, g = 0), upper = c(c = 0.8, d = 20,
        g = 50), control = list(parscale = c(c = 0.5, d = 10,
        g = 1)))
```

```
Coefficients:
  Estimate Std. Error z value     Pr(z)
c  0.41384    0.12572  3.2918 0.0009953 ***
d 13.35082    0.81106 16.4609 < 2.2e-16 ***
g 18.24812    6.03276  3.0248 0.0024877 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

-2 log L: 25.75741
```

Other accessor methods allow you to do inference (`AIC`, `logLik`, `deviance`, `anova` (Likelihood Ratio Test vs alternative models) ...)

`mle2` allows you to generate predicted values, *if* you have used the formula interface rather than writing your own objective function:

```
> TBLvec = seq(9.5,36,length=100)
> predfr <- data.frame(TBL=TBLvec,
                       Kill=predict(mle2_fit,
                         newdata=data.frame(TBL=TBLvec,
                           Exposed=10)))
```

4.2 Profiling

Compute likelihood profiles and profile confidence intervals:

```
> mle2_profile <- profile(mle2_fit)
> mle2_profile_confint <- confint(mle2_profile)
> ## or just confint(mle2_fit) if you don't want to do anything
> ## else with the profile (such as plotting it or extracting
> ## confidence intervals at multiple alpha levels)
```

4.3 MCMC

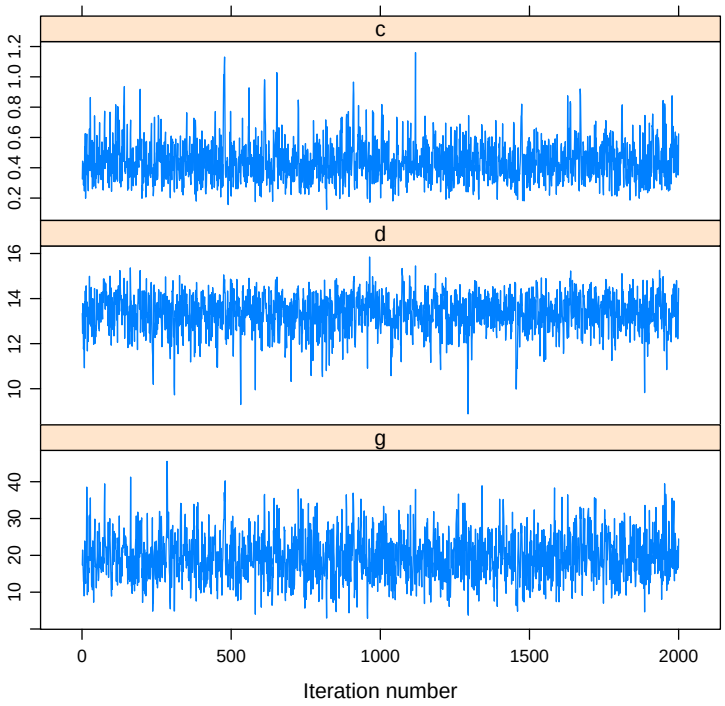
Using the `MCMCpack` package, which contains a general-purpose Metropolis-Hastings sampler (`MCMCmetrop1R`), we can get a *post hoc* MCMC chain by extracting the negative log-likelihood function from the fitted model and putting a wrapper around it to (1) pass the parameters as a list rather than a vector (`do.call(...,as.list(p))`); (2) compute the (positive) log-likelihood rather than the negative log-likelihood; (3) intercept bad (non-finite) values and replace them with large negative values (the specific "bad" value given here might have to be adjusted for other applications).

We specify the starting values from our previous fit; the length of the chain; and the thinning fraction.

```
> tadpole_R_mcmc

function (fit)
{
    require(MCMCpack)
    mcmcfun <- function(p) {
        r <- -do.call(fit@minuslogl, as.list(p))
        if (!is.finite(r))
            -100
        else r
    }
    MCMCmetrop1R(mcmcfun, theta.init = coef(fit), mcmc = 18000,
        thin = 9)
}

> mle2_mcmc <- tadpole_R_mcmc(mle2_fit)

@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
The Metropolis acceptance rate was 0.43719
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@

> colnames(mle2_mcmc) <- names(coef(mle2_fit))
```

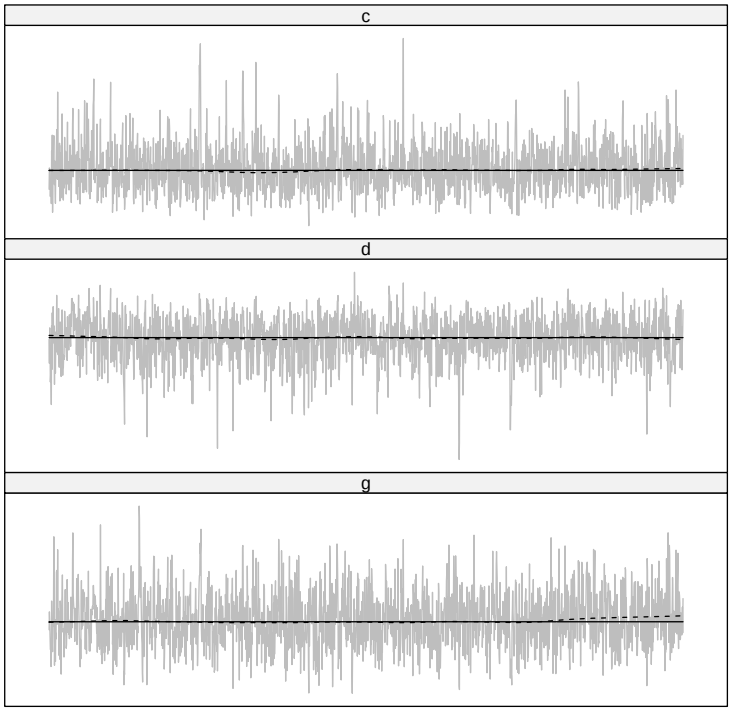
The default `coda` function for trace plots shows the plots in a one-column, stacked layout: this is generally good for longitudinal data like trace plots, but can be hard to read for models (unlike this one) with lots of parameters. Use the `layout=c(m,n)` argument for an $m \times n$ layout; `aspect="fill"` to allow the aspect ratio of the subplots to be flexible); and `as.table=TRUE` if you want the parameters arranged top-to-bottom rather than bottom-to-top.

```
> print(xyplot(mle2_mcmc,as.table=TRUE))
```



`scapeMCMC` offers a slightly prettier version of the trace plot (although its versions do not work with multiple MCM chains):

```
> plotTrace(mle2_mcmc,layout=c(1,3))
```




```
> gd <- geweke.diag(mle2_mcmc)                ## Z-scores
> (gp <- 2*pnorm(abs(gd$z),lower.tail=FALSE)) ## p values

        c         d         g
0.3878674 0.2466525 0.7013971

> effectiveSize(mle2_mcmc)

       c        d        g
1082.187 1085.380 1021.520
```

We are aiming for an effective size of 1000 and non-significant p values from the Geweke diagnostic.

5 AD Model Builder (via R2admb)

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

```
1  DATA_SECTION
2
3    init_int nobs
4    init_vector nexposed(1,nobs)
5    init_vector TBL(1,nobs)
6    init_vector Kill(1,nobs)
7    init_ivector Exposed(1,nobs)
8
9  PARAMETER_SECTION
10
11   objective_function_value f
12   init_bounded_number c(0,1)
13   init_bounded_number d(0,50)
14   init_bounded_number g(-1,25)
15   vector prob(1,nobs)    // per capita mort prob
16 PROCEDURE_SECTION
17
18   f=0.0;                          // initialize objective function
19   dvariable fpen=0.0;         // penalty variable
20   // power-Ricker
21   prob = c*pow(elem_prod(TBL/d,exp(1-TBL/d)),g);
22   // penalties: constrain 0.001 <= prob <= 0.999
23   prob = posfun(prob,0.001,fpen);
24   f += 1000*fpen;
25   prob = 1-posfun(1-prob,0.001,fpen);
26   f += 1000*fpen;
27   // binomial negative log-likelihood
28   f -= sum( log_comb(nexposed,Kill)+
29             elem_prod(Kill,log(prob))+
30             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 `tadpole.tpl`; run `setup_admb()` to locate the AD Model Builder binaries and libraries on our system; and run `do_admb` with appropriate arguments.

```
function ()
{
    run_admb("tadpole")
    L <- read_admb("tadpole")
    clean_admb("tadpole")
    return(L)
}
```

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. (In this example, we could leave out all of the `DATA` section and everything in the `PARAMETER` sections except the utility variable `prob`.)

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:

```
> tfit_admb
```

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

- Get vector of coefficients only:

```
> coef(tfit_admb)
```

```
         c          d          g
 0.4138331 13.3508215 18.2479066
```

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

```
> summary(tfit_admb)
```

```
Model file: tadpole
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(tfit_admb))` to extract just the table).

- Variance-covariance matrix of the parameters:

```
> vcov(tfit_admb)
```

```
           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=logLik(tfit_admb),deviance=deviance(tfit_admb),
   AIC=AIC(tfit_admb))

  logLik deviance      AIC
-12.8938  25.7876  31.7876
```

5.1 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.

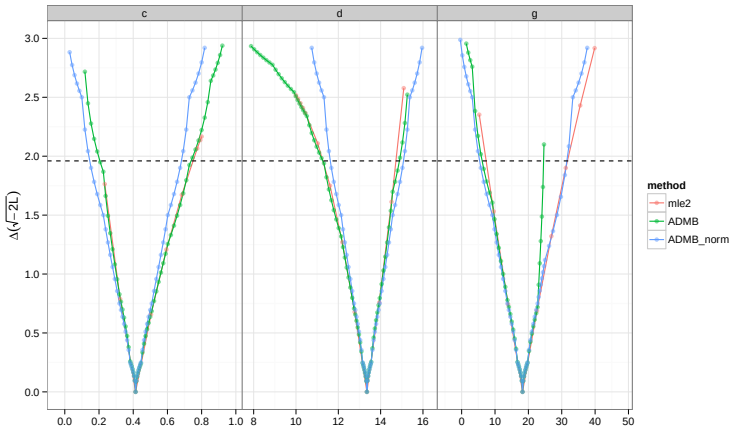
```
> tfit_admb_prof <- do_admb("ReedfrogSizepred0",
                 data=c(list(nobs=nrow(ReedfrogSizepred),
                   nexposed=rep(10,nrow(ReedfrogSizepred))),
                   ReedfrogSizepred),
                 params=list(c=0.45,d=13,g=1),
                 bounds=list(c=c(0,1),d=c(0,50),g=c(-1,25)),
                 run.opts=run.control(checkparam="write",
                   checkdata="write"),
                 profile=TRUE,
                 profpars=c("c","d","g"),
                 admb_errors="warn")
```

The profile information is stored in a list `tfit_admb_prof$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(mle2_fit)
```

(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.

5.2 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.

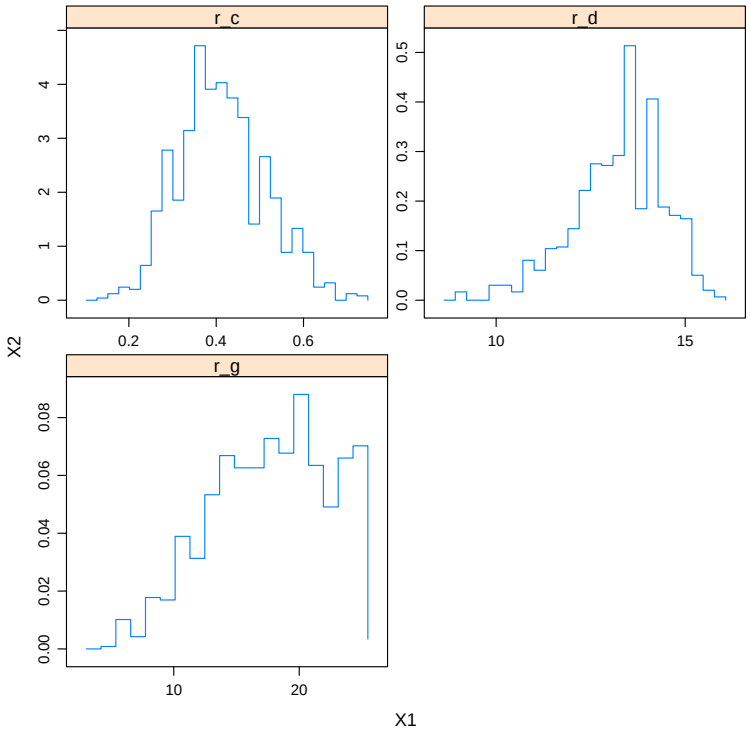
To use MCMC, just add `mcmc=TRUE` and specify the parameters you want to keep track of with `mcmcpars`:

```
> tfit_admb_mcmc <- 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)),
                  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 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(tfit_admb_mcmc$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(tfit_admb_mcmc$mcmc)
```

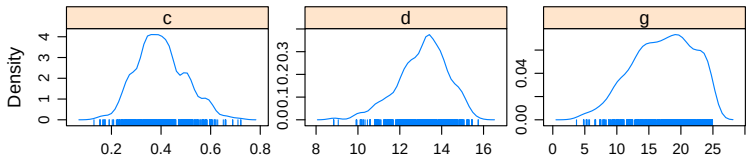
Highest posterior density (i.e. Bayesian credible) intervals:

```
> HPDinterval(mmc)
```

```
       lower      upper
c  0.2386734  0.6037669
d 10.8056715 15.2300318
g  9.2973212 24.9245569
attr(,"Probability")
[1] 0.95
```

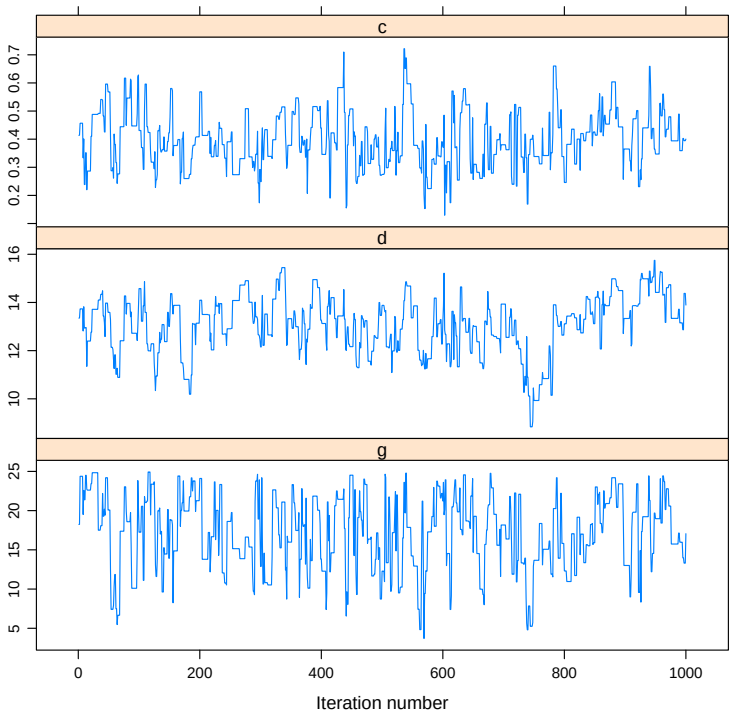
Density plots:

```
> print(densityplot(mmc,layout=c(3,1)))
```



Trace plots:

```
> print(xyplot(mmc,as.table=TRUE))
```



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

See the `coda` and `scapeMCMC` packages for more information on quantitative and graphical diagnostics for MCMC.

The jaggedness of this trace plot indicates that the chain is not actually mixing very well. ADMB does offer some options for tuning the chain: see the ADMB documentation for more details. If you have an ADMB-compiled executable, then (e.g. `./tadpole -? | grep mcmc` at a Unix or MacOS terminal command line will give you a terse list of the possibilities:

```
-mcdiag          use diagonal covariance matrix for mcmc with diagonal values 1
-mcmc [N]        perform markov chain monte carlo with N simulations
-mcmult N        multiplier N for mcmc default
-mcr             resume previous mcmc
-mcrb  N         reduce the amount of correlation in the covariance matrix 1<=N<=9
-mcnoscale       don't rescale step size for mcmc depending on acceptance rate
-nosdmcmc        turn off mcmc histogram calcs to make mcsave run faster
-mcgrope N       use probing strategy for mcmc with factor N
-mcseed N        seed for random number generator for markov chain monte carlo
```

```
-mcscale N       rescale step size for first N evaluations
-mcsave N       save the parameters for every N'th simulation
-mceval         Go throught the saved mcmc values from a previous mcsave
```

6 BUGS

The BUGS input file is very simple:

```
1
2  model {
3    for (i in 1:N) {
4      killprob[i] <- c*((TBL[i]/d)*exp(1-TBL[i]/d))^g
5      Kill[i] ~ dbin(killprob[i],10)
6    }
7    ##  priors match bounds on MLE fits
8    c ~ dunif(0,1)
9    d ~ dunif(0,50) ## dnorm(0,0.001)I(0,)
10   g ~ dunif(-1,40) ## dnorm(0,0.001)I(0,)
11 }
```

- BUGS is not vectorized — all calculations on vectors must be written out with explicit `for` loops
- BUGS uses different names and (more dangerously) different parameterizations for distributions even when they have the same names (see `http://tinyurl.com/bugsparams`): the most common trap is that it parameterizes the normal distribution by the *precision* (inverse variance) rather than the standard deviation
- the priors used here are a bit unusual – we set them as uniform priors with the same range as the box constraints on the parameters used in the previous fits. Bayesians would more typically use a Normal distribution with a large variance (small precision, as specified in BUGS) for parameters that need not be positive, and a log-Normal, Gamma or other positive distribution with a large variance for vague priors on parameters that must be positive. There is much discussion of priors: if you're feeling lost, start with McCarthy (2007) for a relatively gentle discussion, and then continue with other references (Gelman et al., 1996; Lambert et al., 2005). Gelman and co-authors (Gelman and Hill, 2006; Gelman, 2006) warn about situations where the traditional Gamma prior for Normal precisions can be problematic, such as in estimating variances from small samples. The debate over priors continues

in the blogosphere (see e.g. `http://tinyurl.com/gelmanprior` and other posts about priors on Andrew Gelman's blog).

- it is general better practice to run multiple chains (i.e. to run BUGS from several different starting points); it makes slightly stronger tests of convergence, such as the Gelman-Rubin statistic, possible (see e.g. the Weeds example). (This setting goes in the R code to call BUGS, not in the BUGS model file itself.)

Using JAGS:

```
> jags_fit <- jags(model="../BUGS/tadpole_bugs.txt",
                   data=c(list(N=10),as.list(subset(ReedfrogSizepred,
                            select=-c(Exposed)))),
                   parameters.to.save=c("c","d","g"),
                   n.chains=2,
                   inits=list(list(c=0.45,d=13,g=1),
                     list(c=0.4,d=10,g=2)),
                   progress.bar="none")

Compiling model graph
   Resolving undeclared variables
   Allocating nodes
   Graph Size: 64

Initializing model

> jags_fit

Inference for Bugs model at "../BUGS/tadpole_bugs.txt", fit using jags,
 2 chains, each with 2000 iterations (first 1000 discarded)
 n.sims = 2000 iterations saved
         mu.vect sd.vect   2.5%    25%    50%    75%  97.5%  Rhat n.eff
c          0.428   0.117  0.252  0.349  0.409  0.487  0.726 1.007  2000
d         12.466   0.848 11.006 11.848 12.401 13.004 14.266 1.001  2000
g         29.572   6.834 14.706 24.737 30.404 35.282 39.586 1.014   250
deviance  18.894   2.242 16.493 17.275 18.234 19.951 24.647 1.019   100

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 = 2.5 and DIC = 21.4
DIC is an estimate of expected predictive error (lower deviance is better).
```

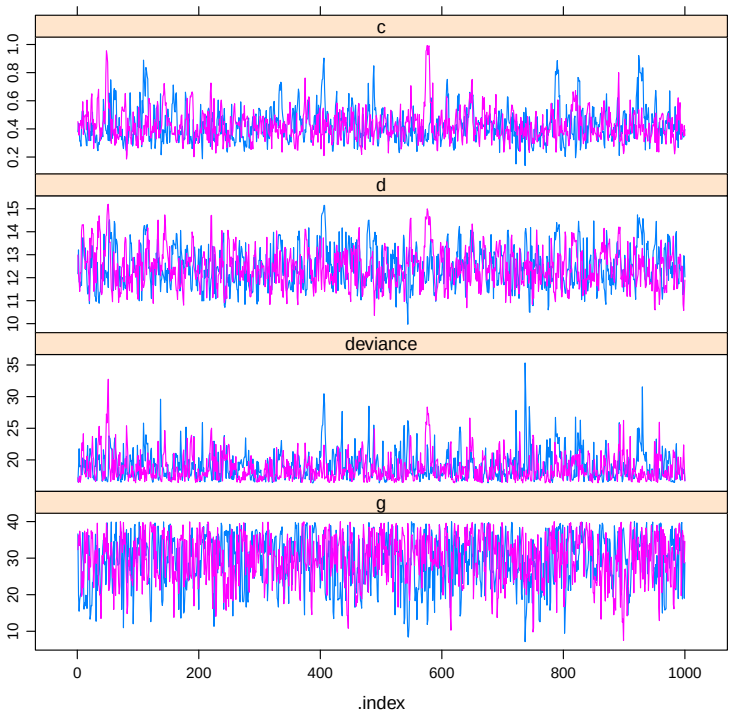
```
> jags_fit_mcmc <- as.mcmc(jags_fit)
```

```
> summary(jags_fit_mcmc)
```

```
Iterations = 1:1000
Thinning interval = 1
Number of chains = 2
Sample size per chain = 1000

1. Empirical mean and standard deviation for each variable,
   plus standard error of the mean:

            Mean     SD Naive SE Time-series SE
c         0.4279 0.1174 0.002624       0.006334
d        12.4662 0.8481 0.018964       0.037523
deviance 18.8935 2.2416 0.050124       0.101192
g        29.5719 6.8336 0.152803       0.261642

2. Quantiles for each variable:

            2.5%     25%    50%     75%   97.5%
c         0.2518  0.3493  0.409  0.4871  0.7263
d        11.0063 11.8480 12.401 13.0044 14.2664
deviance 16.4933 17.2746 18.234 19.9509 24.6467
g        14.7063 24.7373 30.404 35.2821 39.5862
```

Since `jags_fit_mcmc` is an `mcmc` object, we can run the same plots and diagnostics as on the previous MCMC runs: `xyplot`, `densityplot`, `geweke.diag`, `effectiveSize`, `HPDinterval`, etc..

```
> source("../../TOOLS/misc_funs.R")  ## get dropdev() tool to ignore deviance
> print(xyplot(dropdev(jags_fit_mcmc),as.table=TRUE))
```



This is better behaved than the ADMB trace plot, although we would like it to look still more like white noise ...

7 Simulation results

(See figures.)

References

Bolker, B. M. (2008, July). *Ecological Models and Data in R*. Princeton University Press.

Gelman, A. (2006). Prior distributions for variance parameters in hierarchical models. *Bayesian Analysis 1*(3), 515–533.

Gelman, A., J. Carlin, H. S. Stern, and D. B. Rubin (1996). *Bayesian data analysis*. New York, New York, USA: Chapman and Hall.

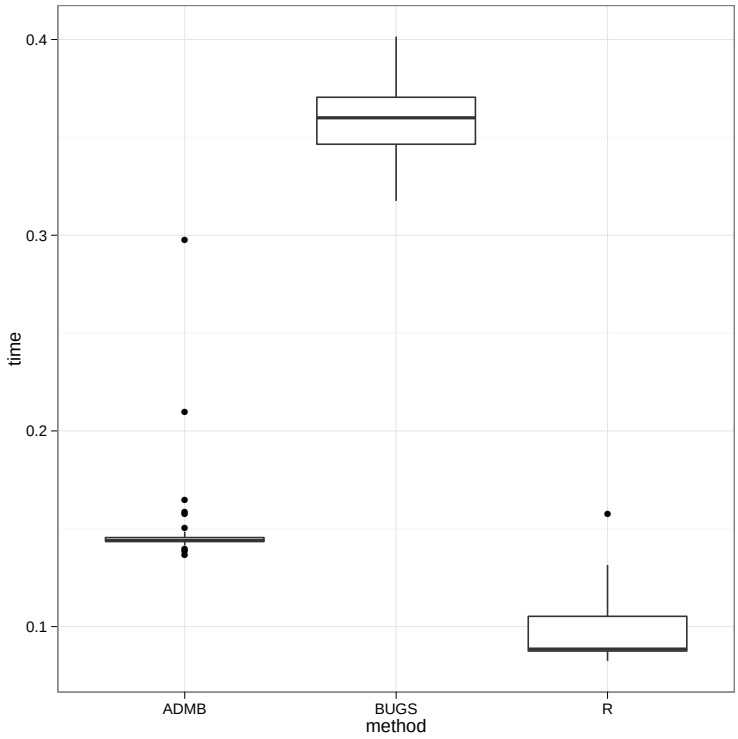


Figure 2: Timings

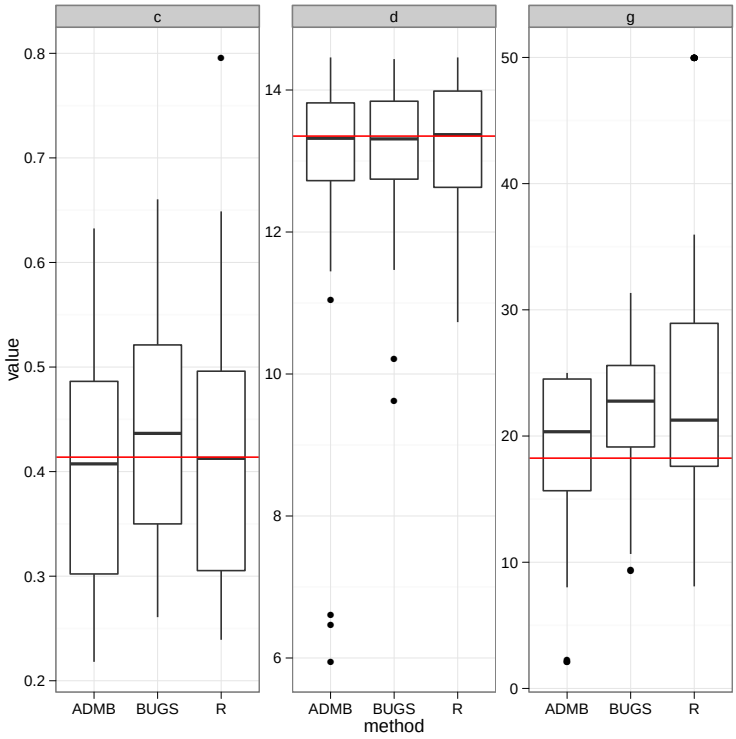


Figure 3: Distribution of parameter estimates

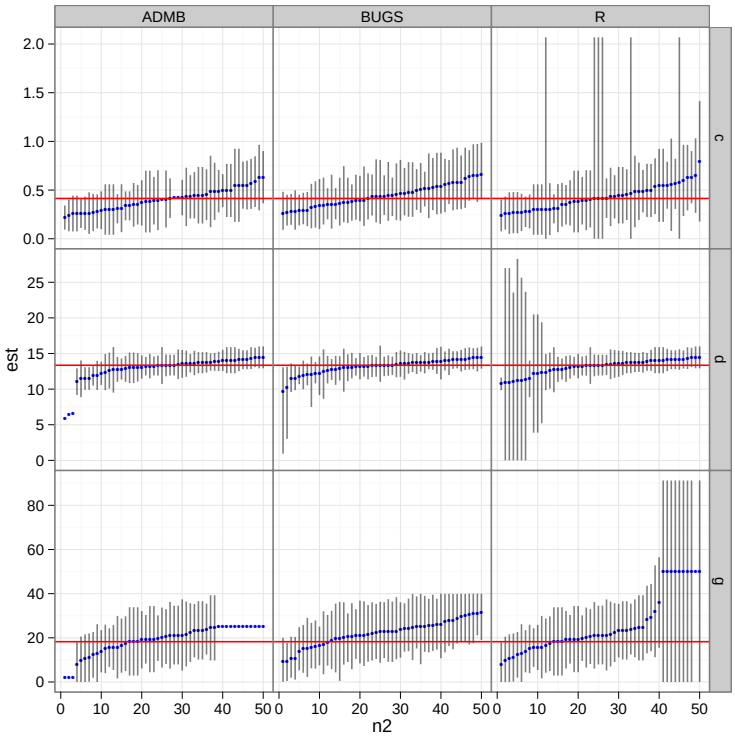


Figure 4: Full distribution of parameters+CIs

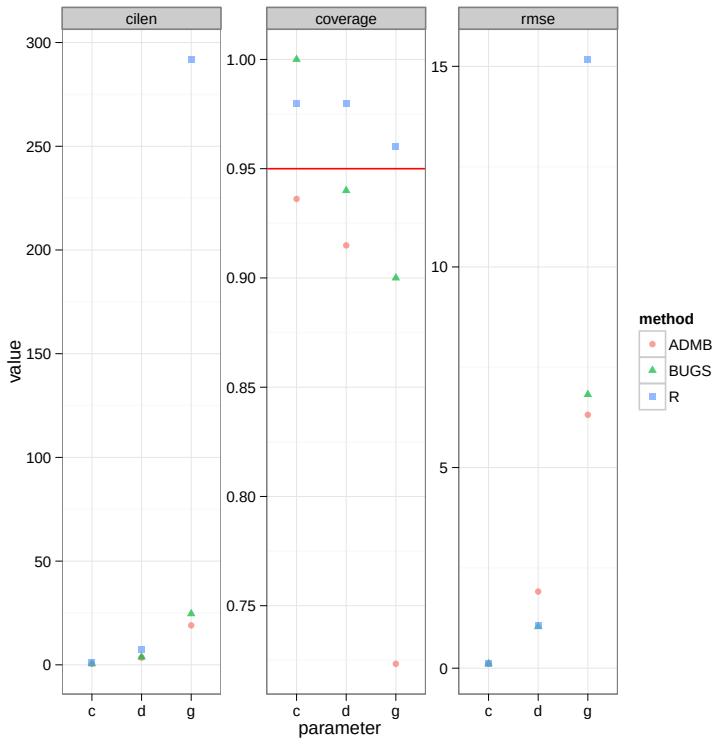


Figure 5: Confidence interval summaries

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

Lambert, P., A. Sutton, P. Burton, K. Abrams, and D. Jones (2005). How vague is vague? A simulation study of the impact of the use of vague prior distributions in MCMC using WinBUGS. *Statistics in Medicine 24*(15), 2401–2428.

McCarthy, M. (2007). *Bayesian methods for ecology*. Cambridge, England: Cambridge University Press.

McCoy, M. W., B. M. Bolker, K. M. Warkentin, and J. R. Vonesh (2011, June). Predicting predation through prey ontogeny using size-dependent functional response models. *The American Naturalist 177*(6), 752–766. PMID: 21597252.

Vonesh, J. R. and B. M. Bolker (2005). Compensatory larval responses shift tradeoffs associated with predator-induced hatching plasticity. *Ecology 86*(6), 1580–1591.