

# The Hobbs Weed Infestation Problem

John C. Nash, Telfer School of Management, University of Ottawa, Ottawa, Canada  
Anders Nielsen, Technical University of Denmark, Charlottenlund, Denmark  
Ben Bolker, McMaster University, Hamilton, Canada

May 2011

## Note

This (partially finished) vignette borrows heavily on a Scaling-Optim vignette by John Nash and some material is duplicated so that the flow of ideas here is self-contained. We will use the statistical programming system R (<http://www.r-project.org>) to present most of our calculations, but will use R, AD Model Builder (abbreviated ADMB) and the Bayesian estimation approach called BUGS (via either JAGS or OpenBUGS) to attempt estimation of models for the data of this problem.

This document is intended as a repository of a range of different attempts to approach a problem in nonlinear estimation. As such it has sections that are heavy reading. It is an archive of what we did and thought about rather than a summary, though the presentation is not necessarily in order.

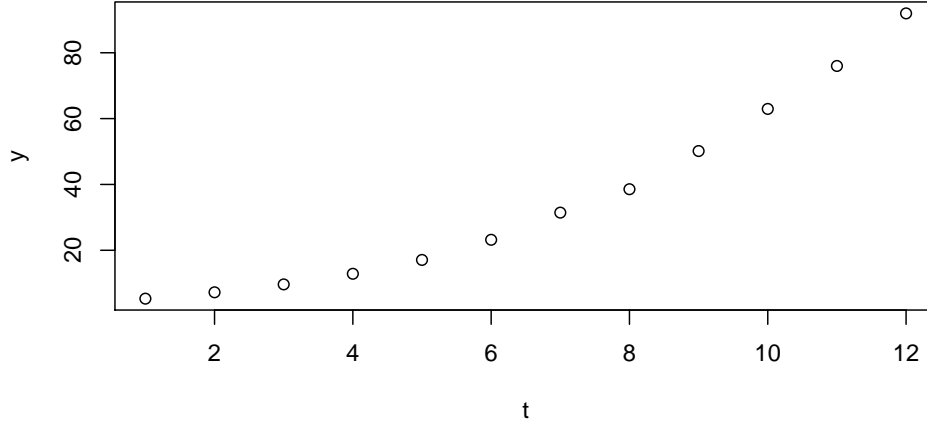
## 1 The initial Hobbs weed infestation problem

This problem came across the desk of John Nash sometime in 1974 when he was working on the development of a program to solve nonlinear least squares estimation problems. He had written several variants of Gauss-Newton methods in BASIC for a Data General NOVA system that offered a very limited environment of a 10 character per second teletype with paper tape reader and punch that allowed access to a maximum 8K byte (actually 4K word) segment of the machine. Arithmetic was particularly horrible in that floating point used six hexadecimal digits in the mantissa with no guard digit.

The problem was supplied by Mr. Dave Hobbs of Agriculture Canada. As told to John Nash, the observations ( $y$ ) are weed densities per unit area over 12 growing periods. We were never given the actual units of the observations. Here is the data.

```
> # draw the data
> y<-c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443,
+      38.558, 50.156, 62.948, 75.995, 91.972)
> t<-1:12
> plot(t,y)
> title(main="Hobbs' weed infestation data", font.main=4)
```

### ***Hobbs' weed infestation data***



It was suggested that the appropriate model was a 3-parameter logistic, that is,

$$y_i = b_1 / (1 + b_2 \exp(-b_3 t_i)) + \varepsilon_i \quad (1)$$

where  $\varepsilon_i \sim N(0, \sigma^2)$ ,  $t_i$  is the growing period, and  $i = 1, \dots, 26$ .

The problem can be approached as a nonlinear least squares problem. In R, the `nls()` function could be used, but fails for many sets of starting parameters, and optimization turns out to be more robust in this case. Examples of the use of `nls()` are presented later. The problem has been published in a number of places, notably Nash (1979) and Nash and Walker-Smith (1987).

Most discussions of the problem have centered on solving the three parameter nonlinear least squares problem, either as such a problem using Gauss-Newton (ref?) or Marquardt (ref?) methods, or else via general nonlinear optimization of the sum of squares. While minimizing the sum of squares of residuals is one approach to “fitting” a function to data, statisticians frequently prefer the maximum likelihood approach so that the variance can also be estimated at the same time. This is conventionally accomplished by minimizing the negative log likelihood.

For our problem,

$$\begin{aligned} nll &= 0.5 \sum_{i=1}^n (\log(2\pi\sigma^2) + (res[i]^2)/\sigma^2) \\ &\quad \text{or} \\ nll &= 0.5(n \log(2\pi\sigma^2) + \sum_{i=1}^n (res[i]^2/\sigma^2)) \end{aligned}$$

There are a number of annoyances about the particular logistic problem and data:

- The scale of the problem is such that the upper asymptote  $b_1$  has scale of the order of 100,  $b_2$  has scale of the order 10, while the coefficient of time  $b_3$  is scaled by 0.1. We can explicitly put such scaling factors into the model so that the coefficients all come out with roughly similar scale. As we shall show, this eases the computational task.
- It is useful if all the coefficients are positive. We can use explicit bounds in some optimization and nonlinear least squares methods, but will see that writing the model in terms of the logs of the parameters achieves the same goal and also provides a scaling of the problem.

- Other transformations of the problem are possible; at least one will be mentioned later.

## 2 Preparation

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
> 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(R2admb)  ## R interface to AD Model Builder
> library(R2jags)  ## R interface to JAGS (BUGS dialect)
> # source("../R/tadpole_R_funs.R")
>
```

## 3 Solving the maximum likelihood problem – log-form parameters

We will solve the 4-parameter maximum likelihood problem by ADMB, R, and JAGS (a BUGS variant, Plummer (2003)). The parameters our estimation or optimization tools seek will be the logs of the quantities that enter the models, thereby forcing positivity on these quantities.

### 3.1 Solving the ML problem in ADMB

The we use a small script to prepare the data file for use by AD Model Builder:

```
> source("../ADMB/weeds_ADMB_funs.R")
> weeds_ADMB_setup()
```

This creates the file

```
# "../ADMB/weed.dat" produced by dat_write() from R2admb Tue Aug 14 20:11:45 2012
# noObs
12

#
1 5.308
2 7.24
3 9.638
4 12.866
5 17.069
6 23.192
7 31.443
8 38.558
9 50.156
10 62.948
11 75.995
12 91.972
```

The implementation follows the typical AD Model Builder template, first data is read in, then model parameters are declared, and finally the negative log likelihood is coded.

```
DATA_SECTION
  init_int noObs
  init_matrix obs(1,noObs,1,2)

PARAMETER_SECTION
  init_number logb1;
  init_number logb2;
  init_number logb3;
  init_number logSigma;

  sdreport_number sigma2;
  sdreport_number b1;
  sdreport_number b2;
  sdreport_number b3;
  sdreport_vector pred(1,noObs);
  objective_function_value nll;

PRELIMINARY_CALCS_SECTION

PROCEDURE_SECTION
  b1=exp(logb1);
  b2=exp(logb2);
  b3=exp(logb3);
  sigma2=exp(2.0*logSigma);

  for(int i=1; i<=noObs; ++i){
    pred(i)=b1/(1.0+b2*exp(-b3*obs(i,1)));
    nll+=0.5*(log(2.0*M_PI*sigma2)+square((obs(i,2)-pred(i)))/sigma2);
  }

REPORT_SECTION
  report<<"rss " <<norm2(column(obs,2)-pred)<<endl;
```

The model can be run from the command line by compiling and then executing the produced binary, but this can also be accomplished from within the R console like this as long as we ensure that the R working directory is set to the directory containing the AD Model Builder code for the problem `weed.tpl` and the corresponding data file `weed.dat`.

```
> file.copy("../ADMB/weed.tpl","weed.tpl")
```

```
[1] TRUE
```

```
> file.copy("../ADMB/weed.dat","weed.dat")
```

```
[1] TRUE
```

```
> system('admb weed')
> system.time(system('./weed > weedout.txt'))
```

```
user  system elapsed
0.004  0.024   0.035
```

```
> unlink("weed.tpl") # for cleanup of WRITEUP directory
> unlink("weed.dat")
```

```
Initial statistics: 4 variables; iteration 0; function evaluation 0; phase 1
Function value  1.1771317e+04; maximum gradient component mag -2.3509e+04
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
```

```

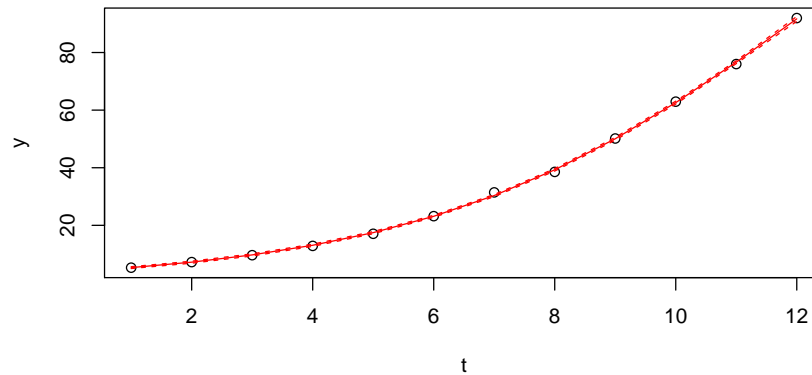
1  0.00000 -4.12021e+02 | 2  0.00000  2.38244e+00 | 3  0.00000 -5.51269e+00
4  0.00000 -2.35086e+04 |

Intermediate statistics: 4 variables; iteration 10; function evaluation 25; phase 1
Function value  6.0649538e+01; maximum gradient component mag -2.2037e+02
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  4.44243 -2.20366e+02 |  2  5.12862  4.57014e+01 |  3 -0.46331 -1.84794e+02
  4  0.88196 -6.60776e+01 |
4 variables; iteration 20; function evaluation 36; phase 1
Function value  2.4384978e+01; maximum gradient component mag -2.8737e+01
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  5.04861 -1.91374e+01 |  2  3.77868  6.10380e+00 |  3 -1.08066 -2.87369e+01
  4  1.05750  1.06646e+01 |
4 variables; iteration 30; function evaluation 46; phase 1
Function value  2.2788992e+01; maximum gradient component mag -2.5509e+01
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  5.57992 -8.96350e+00 |  2  4.04802  7.33044e+00 |  3 -1.25408 -2.55090e+01
  4  0.92887  1.07695e+01 |
4 variables; iteration 40; function evaluation 69; phase 1
Function value  1.0842488e+01; maximum gradient component mag -7.4469e+02
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  5.39693 -3.33655e+02 |  2  3.94939  2.38746e+02 |  3 -1.20164 -7.44692e+02
  4 -0.50648  2.13965e-01 |
4 variables; iteration 50; function evaluation 80; phase 1
Function value  7.8214592e+00; maximum gradient component mag  1.9830e-04
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  5.27906  1.03439e-04 |  2  3.89369 -6.21736e-05 |  3 -1.15973  1.98303e-04
  4 -0.76715  2.11426e-07 |

- final statistics:
4 variables; iteration 51; function evaluation 81
Function value  7.8215e+00; maximum gradient component mag  9.4619e-07
Exit code = 1; converg critr  1.0000e-04
Var  Value      Gradient  |Var  Value      Gradient  |Var  Value      Gradient
  1  5.27906  4.42720e-07 |  2  3.89369 -2.36799e-07 |  3 -1.15973  9.46188e-07
  4 -0.76715  9.89104e-09 |
Estimating row 1 out of 4 for hessian
Estimating row 2 out of 4 for hessian
Estimating row 3 out of 4 for hessian
Estimating row 4 out of 4 for hessian

```

After running the model we can plot the fit to make sure all went well.



Estimates and standard deviations of model parameters and derived quantities are located in the file `weed.std`.

index	name	value	std dev
1	logb1	5.2791e+00	5.0086e-02
2	logb2	3.8937e+00	2.9896e-02
3	logb3	-1.1597e+00	1.8982e-02
4	logSigma	-7.6715e-01	2.0412e-01
5	sigma2	2.1561e-01	8.8021e-02
6	b1	1.9619e+02	9.8262e+00
7	b2	4.9092e+01	1.4676e+00
8	b3	3.1357e-01	5.9523e-03
9	pred	5.3199e+00	1.5835e-01
10	pred	7.2072e+00	1.7632e-01
11	pred	9.7300e+00	1.8936e-01
12	pred	1.3075e+01	1.9478e-01
13	pred	1.7462e+01	1.9168e-01
14	pred	2.3134e+01	1.8478e-01
15	pred	3.0337e+01	1.8781e-01
16	pred	3.9274e+01	2.1279e-01
17	pred	5.0048e+01	2.4618e-01
18	pred	6.2600e+01	2.5541e-01
19	pred	7.6648e+01	2.5149e-01
20	pred	9.1684e+01	4.2122e-01

### 3.2 Solving the ML problem in R

We first load functions to compute the residuals (actually in natural rather than log scale – the logs are only needed in the optimization).

```
> # source("../R/shobbs.R", echo=TRUE)
> source("../R/lhobbs.R", echo=TRUE)

> lhobbs.f <- function(xl, y) {
+   if (abs(12 * exp(xl[3])) > 50) {
+     fbad <- .Machine$double.xmax
+     return(fbad)
+   }
+   res .... [TRUNCATED]

> lhobbs.res <- function(xl, y) {
+   x <- exp(xl)
+   if (abs(12 * x[3]) > 50) {
+     rbad <- rep(.Machine$double.xmax, length(x))
+     .... [TRUNCATED]

> lhobbs.jac <- function(xl, y) {
+   x <- exp(xl)
+   jj <- matrix(0, 12, 3)
+   t <- 1:12
+   yy <- exp(-x[3] * t)
+   zz <- 1/(1 + x[2] * .... [TRUNCATED]
```

```

> lhobbs.g <- function(xl, y) {
+   shj <- lhobbs.jac(xl, y)
+   shres <- lhobbs.res(xl, y)
+   shg <- as.vector(2 * (shres %*% shj))
+   retu .... [TRUNCATED]

```

Then we define the log likelihood function.

```

> source("../R/lhobbslik.R", echo=TRUE)

> lhobbs.lik <- function(xaug, y = y0) {
+   xl <- xaug[1:3]
+   logSigma <- xaug[4]
+   sigma2 = exp(2 * logSigma)
+   res <- lhobbs.res(xl, .... [TRUNCATED]

> lhobbs.lg <- function(xaug, y = y0) {
+   xl <- xaug[1:3]
+   logSigma <- xaug[4]
+   sigma2 = exp(2 * logSigma)
+   res3 <- lhobbs.res(xl, .... [TRUNCATED]

```

It is a good idea to test the function and gradient to make sure we have our code working properly. There is still some room for error, but at least the numerical gradients of the function match the values from the analytic expressions.

```

> source("../R/lhobbsliktest.R", echo=TRUE)

> source("../R/lhobbs.R")

> source("../R/lhobbslik.R")

> require(numDeriv)

> y0 <- c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192,
+   31.443, 38.558, 50.156, 62.948, 75.995, 91.972)

> xxax <- c(2, 5, 3, 1)

> xxa <- log(xxax)

> xl3 <- xxa[1:3]

> res0 <- lhobbs.res(xl3, y = y0)

> print(res0)
[1] -3.706636 -5.264484 -7.639233 -10.866061 -15.069003 -21.192000

```

```

[7] -29.443000 -36.558000 -48.156000 -60.948000 -73.995000 -89.972000

> ss0 <- lhobbs.f(xl3, y = y0)

> print(ss0)
[1] 22701.32

> jj0 <- lhobbs.jac(xl3, y = y0)

> jj0n <- jacobian(lhobbs.res, xl3, y = y0)

> print(jj0 - jj0n)
      [,1]      [,2]      [,3]
[1,] 1.622902e-11 8.335888e-12 -7.149836e-12
[2,] -1.407408e-11 -4.927860e-12 4.276995e-11
[3,] 5.253353e-12 8.200460e-12 -7.176780e-12
[4,] 3.741807e-11 5.181383e-11 2.631627e-11
[5,] 8.541101e-11 -7.939929e-12 4.143344e-11
[6,] 1.165295e-10 1.190895e-11 5.231059e-11
[7,] -1.559814e-10 1.098205e-11 5.086183e-13
[8,] -1.250731e-10 -2.302512e-11 5.640126e-12
[9,] 4.160114e-10 -1.880307e-11 -9.938316e-12
[10,] -2.587837e-10 -9.357623e-13 2.616794e-11
[11,] 4.347960e-10 -4.658886e-14 1.537432e-12
[12,] 4.287295e-10 -2.319523e-15 8.350282e-14

> g0 <- lhobbs.g(xl3, y = y0)

> g0n <- grad(lhobbs.f, xl3, y = y0)

> print(g0)
[1] -1608.004405      2.641076     -8.813356

> print(g0 - g0n)
[1] -9.966129e-08 6.350435e-08 -5.101980e-08

> f <- lhobbs.lik(xxa, y = y0)

> f
[1] 11361.69

> ga <- lhobbs.lg(xxa, y = y0)

> ga
[1] -804.002202      1.320538     -4.406678 -22689.316307

```



```
> gn <- grad(lhobbs.lik, xxa, y = y0)
```

```
> gn
```

```
[1] -804.002202      1.320538     -4.406678 -22689.316307
```

Finally, we run the optimization. Here we try a number of methods both with and without analytic gradients.

```
> source("../R/lhobbslikrun.R", echo=TRUE)
```

```
> require("optimx")
```

```
> source("../R/lhobbs.R")
```

```
> source("../R/lhobbslik.R")
```

```
> source("../R/lhobbslikn.R")
```

```
> cat("Using analytic derivatives\n")
```

```
Using analytic derivatives
```

```
> test <- try(ansR5 <- optimx(log(c(2, 5, 3, 1)), lhobbs.lik,
+   lhobbs.lg, control = list(all.methods = TRUE)))
end topstuff in optimxCRA
```

```
> if (class(test) != "try-error") {
```

```
+   print(ansR5)
```

```
+ } else {
```

```
+   cat("ML attempt with optimx on scaled likelihood and analytic gradients failed ..." ...
```

	par	fvalues	method	fns
4	NA, NA, NA, NA	8.988466e+307	L-BFGS-B	NA
9	NA, NA, NA, NA	8.988466e+307	Rcgmin	NA
11	5.2790641, 3.8936885, -1.1597334, -0.7671477	7.821459	newuoa	1968
2	5.206357, 3.861371, -1.133986, 0.492384	17.51391	CG	358
3	5.2783751, 3.8933393, -1.1595111, -0.7669179	7.821567	Nelder-Mead	417
5	5.2790329, 3.8936753, -1.1597222, -0.7671501	7.821459	nlm	NA
7	5.2790649, 3.8936871, -1.1597343, -0.7670325	7.821459	spg	652
10	NA, NA, NA, NA	NULL	Rvmin	NA
1	5.2790644, 3.8936887, -1.1597335, -0.7671526	7.821459	BFGS	597
6	5.2790645, 3.8936887, -1.1597335, -0.7671502	7.821459	nlminb	74
8	5.2790645, 3.8936887, -1.1597335, -0.7671503	7.821459	ucminf	59
	grs itns conv KKT1 KKT2 xtimes			
4	NULL NULL 9999 NA NA 0.004			
9	NA NA 9999 NA NA 0.004			
11	NA NULL 0 TRUE TRUE 0.06			

```

2  101 NULL      1 FALSE TRUE  0.016
3   NA NULL      0 FALSE TRUE  0.008
5   NA  69       0 FALSE TRUE  0.012
7   NA 437       0  TRUE TRUE  0.112
10  NA  NA 9999   NA  NA      0
1   66 NULL      0  TRUE TRUE  0.016
6   45  44       0  TRUE TRUE  0.008
8   59 NULL      0  TRUE TRUE  0.004

```

```
> cat("Using numerical derivative approximations\n")
```

```
Using numerical derivative approximations
```

```

> test <- try(ansR5n <- optimx(log(c(2, 5, 3, 1)), lhobbs.lik,
+   control = list(all.methods = TRUE)))
end topstuff in optimxCRA
function (xaug, y = y0)

```

```

{
  x1 <- xaug[1:3]
  logSigma <- xaug[4]
  sigma2 = exp(2 * logSigma)
  res <- lhobbs.res(x1, y)
  nll <- 0.5 * (length(res) * log(2 * pi * sigma2) + sum(res *
    res)/sigma2)
}

```

```

> if (class(test) != "try-error") {
+   print(ansR5n)
+ } else {

```

```
+   cat("ML attempt with optimx on scaled likelihood and numerical gradients fail ..." ... [T
```

```

                                par      fvalues      method  fns
4                                NA, NA, NA, NA 8.988466e+307  L-BFGS-B   NA
9                                NA, NA, NA, NA 8.988466e+307   Rcgmin   NA
11 5.2790641, 3.8936885, -1.1597334, -0.7671477    7.821459   newuoa 1968
2   5.1381561, 3.8288219, -1.1088996, 0.7803977    20.88628      CG   358
3   5.2783751, 3.8933393, -1.1595111, -0.7669179    7.821567 Nelder-Mead 417
8   5.2784607, 3.8933534, -1.1595431, -0.7671387    7.821539   ucminf   54
1   5.2792349, 3.8937746, -1.1597901, -0.7671011    7.821465    BFGS   260
5   5.2790329, 3.8936753, -1.1597222, -0.7671501    7.821459      nlm   NA
7   5.2790635, 3.8936842, -1.1597345, -0.7670815    7.821459      spg   809
10                                NA, NA, NA, NA      NULL   Rvmmin   NA
6   5.2790645, 3.8936887, -1.1597335, -0.7671502    7.821459   nlminb   72
  grs itns conv  KKT1 KKT2 xtmes
4  NULL NULL 9999   NA  NA  0.012
9   NA  NA 9999   NA  NA  0.008
11  NA NULL    0  TRUE TRUE  0.06

```

2	101	NULL	1	FALSE	TRUE	0.024
3	NA	NULL	0	FALSE	TRUE	0.008
8	54	NULL	0	FALSE	TRUE	0.004
1	57	NULL	0	FALSE	TRUE	0.016
5	NA	69	0	FALSE	TRUE	0.012
7	NA	545	0	FALSE	TRUE	0.164
10	NA	NA	9999	NA	NA	0.004
6	208	47	0	TRUE	TRUE	0.004

### 3.3 Solving the ML problem in JAGS

The BUGS/JAGS control file we will use is given below. This simply provides the (scaled) model and very simple uniform priors. The range for the deviance was widened based on estimates found.

```
model {
## attempt to model weeds data using JAGS/BUGS
  b1<-exp(logb1)
  b2<-exp(logb2)
  b3<-exp(logb3)
  sigma<-exp(logsigma)
  for (i in 1:N) {
    yhat[i] <- b1/(1+b2*exp(-b3*t[i]))
    y[i] ~ dnorm(yhat[i],tau)
  }
  tau <- 1/(sigma*sigma)
  ## priors -- note must include 0 because of one chain
  logb1 ~ dunif(-10,10)
  logb2 ~ dunif(-10,10)
  logb3 ~ dunif(-10,10)
  logsigma ~ dunif(-10,10)
}
```

We run JAGS from R as follows.

```
> # rm(list=ls()) ## Clear workspace. NOT in vignette.
> library(rjags) # ? Is this needed?
> library(R2jags)
> y<-c(5.308, 7.24, 9.638, 12.866, 17.069, 23.192, 31.443, 38.558, 50.156, 62.948,
+      75.995, 91.972)
> t<-1:12
> # Note that n.iter=10000 and n.burning=1000 was not sufficient.
>
> tfit_jags <- jags(model="../BUGS/weeds_bugs.txt",
+   data=list(N=length(y),t=t, y=y),
+   parameters.to.save=c("logb1","logb2","logb3","logsigma"),
+   n.iter=10000,
+   n.burnin=1000,
+   n.thin=10,
+   n.chains=3,
+   inits=list(list(logb1=1,logb2=1,logb3=1,logsigma=1,
+     .RNG.name="base::Wichmann-Hill", .RNG.seed=654321),
+     list(logb1=-1,logb2=-1,logb3=-1,logsigma=-1,
```

```
+ .RNG.name="base::Wichmann-Hill", .RNG.seed=321654),
+ list(logb1=0,logb2=0,logb3=0,logsigma=0,
+ .RNG.name="base::Wichmann-Hill", .RNG.seed=123456)))
```

```
Compiling model graph
  Resolving undeclared variables
  Allocating nodes
  Graph Size: 99
```

```
Initializing model
```

```
> # wpred <- .... stuff to create predicted model
> cat("output of jags() run\n")
```

```
output of jags() run
```

```
> tfit_jags
```

```
Inference for Bugs model at "../BUGS/weeds_bugs.txt", fit using jags,
  3 chains, each with 10000 iterations (first 1000 discarded), n.thin = 10
  n.sims = 2700 iterations saved
```

	mu.vect	sd.vect	2.5%	25%	50%	75%	97.5%	Rhat	n.eff
logb1	5.633	0.595	5.178	5.270	5.324	5.877	7.253	1.747	6
logb2	4.162	0.471	3.835	3.888	3.923	4.260	5.461	1.732	7
logb3	-1.223	0.099	-1.440	-1.303	-1.176	-1.155	-1.119	1.711	6
logsigma	-0.235	0.566	-0.963	-0.662	-0.427	0.199	0.966	1.499	8
deviance	28.311	12.613	16.349	18.788	21.921	39.698	53.840	1.524	8

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,  $pD = \text{var}(\text{deviance})/2$ )

$pD = 53.9$  and  $DIC = 82.2$

DIC is an estimate of expected predictive error (lower deviance is better).

This shows us the estimates found from our MCMC iterations. The effective sample sizes for all four estimated parameters are reasonably large and the Gelman Rhat statistics are less than 1.2, suggesting that the chains have converged. The following code simply extracts part of the output that we will use for some graphs.

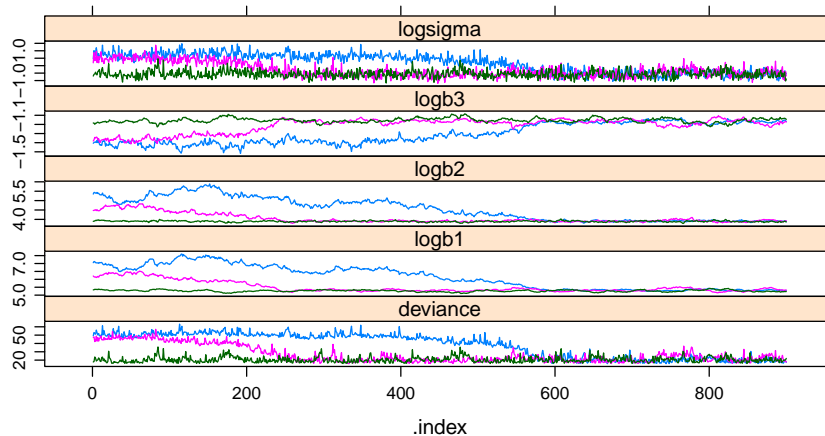
```
> tfit_jags_b <- tfit_jags$BUGSoutput
```

We graph the progress of the MCMC process to check if it has stabilized ("converged"). While there is some noise in the plots, they are reasonably stable.

```
> library(emdbook) ## for as.mcmc.bugs
> tfit_jags_m <- as.mcmc.bugs(tfit_jags_b)
```

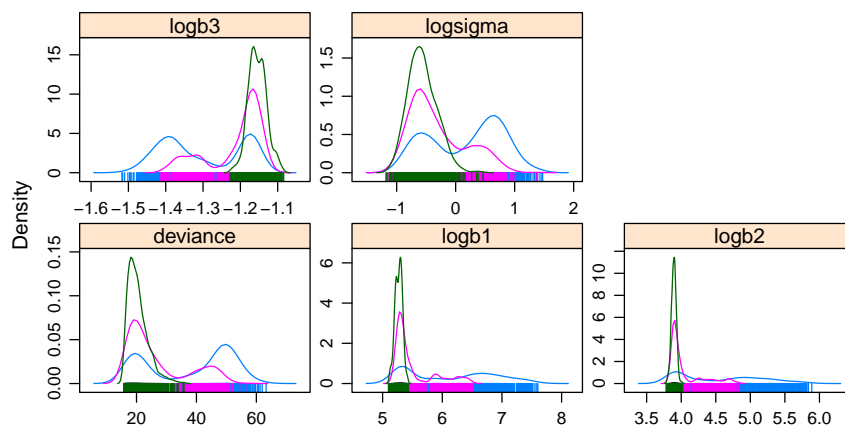
We can also look at the parameter distributions. Again, these are not too different, and we can treat the results as reasonable.

```
> library(coda)
> print(xyplot(tfit_jags_m))
```



Clearly the Bayesian (JAGS) approach takes a very different point of view to the optimization (R and ADMB) methods. Nevertheless, the model parameters are not very different, and the JAGS outputs give us some idea of the possible distribution of the parameters.

```
> ## x11()
> print(densityplot(tfit_jags_m))
```



## 4 Other views of the Weeds problem

### 4.1 Nonlinear least squares

The original problem was, with apparent simplicity, to find the three unscaled logistic parameters that provided the best fit to the 12 observations. This turned out to be surprisingly difficult. Indeed,

the Statistical Research Service of Agriculture Canada was, in 1974, one of the most sophisticated and well-connected biostatistics units in the world, but staff found no computer program that would crack this problem. John Nash was, at the time, trying to develop programs in BASIC for a Data General Nova minicomputer and an HP 9830A desktop calculator, and offered to give his codes a try. These succeeded in finding a solution.

The Weeds problem was, in fact, a prime motivator to prepare very robust nonlinear least squares and optimization codes for what were, in today's view, rather poor computational platforms. However, even today, the original problem creates difficulties. Let us see what R can do.

```
> source("../R/weeds_nls1.R", echo=TRUE)

> cat("nls tests on Hobbs weeds problem\n")
nls tests on Hobbs weeds problem

> tdata <- read.table("../DATA/weeds.dat", header = TRUE)

> t <- tdata$t

> y <- tdata$y

> rm(tdata)

> cat("Test with original 3-parameter model, unscaled parameters -- good starting parameters\n")
Test with original 3-parameter model, unscaled parameters -- good starting parameters

> xx <- c(200, 50, 0.3)

> q <- try(ansxx <- nls(y ~ x1/(1 + x2 * exp(-x3 * t)),
+   data = list(y, t), start = list(x1 = xx[1], x2 = xx[2], x3 = xx[3])))

> if (class(q) == "try-error") cat("Failed\n") else print(ansxx)
Nonlinear regression model
  model: y ~ x1/(1 + x2 * exp(-x3 * t))
  data: list(y, t)
         x1      x2      x3
196.1863  49.0916  0.3136
residual sum-of-squares: 2.587

Number of iterations to convergence: 4
Achieved convergence tolerance: 1.918e-07

> cat("Same problem, but a very crude starting point\n")
Same problem, but a very crude starting point

> xy <- c(1, 1, 1)
```

```

> q <- try(ansxy <- nls(y ~ x1/(1 + x2 * exp(-x3 * t)),
+   data = list(y, t), start = list(x1 = xy[1], x2 = xy[2], x3 = xy[3])))

> if (class(q) == "try-error") cat("Failed\n") else print(ansxy)
Failed

> cat("Scaled problem, and a crude but not random starting point\n")
Scaled problem, and a crude but not random starting point

> xx <- c(2, 1, 1)

> q <- try(ansxxs <- nls(y ~ 100 * x1/(1 + 10 * x2 *
+   exp(-0.1 * x3 * t)), data = list(y, t), start = list(x1 = xx[1],
+   x2 = xx[2], x3 = x .... [TRUNCATED])

> if (class(q) == "try-error") cat("Failed\n") else print(ansxxs)
Nonlinear regression model
  model: y ~ 100 * x1/(1 + 10 * x2 * exp(-0.1 * x3 * t))
 data: list(y, t)
      x1      x2      x3
1.962 4.909 3.136
residual sum-of-squares: 2.587

Number of iterations to convergence: 20
Achieved convergence tolerance: 1.253e-06

> hobbs.f <- function(x) {
+   if (abs(12 * x[3]) > 50) {
+     fbad <- .Machine$double.xmax
+     return(fbad)
+   }
+   res <- hobbs.r .... [TRUNCATED]

> hobbs.res <- function(x) {
+   if (abs(12 * x[3]) > 50) {
+     rbad <- rep(.Machine$double.xmax, length(x))
+     return(rbad)
+   }
+   .... [TRUNCATED]

> hobbsl.f <- function(xlog) {
+   x <- exp(xlog)
+   if (abs(12 * x[3]) > 50) {
+     fbad <- .Machine$double.xmax
+     return(fbad)

```

```

+ .... [TRUNCATED]

> cat("For comparison, try Nelder-Mead on the failed case\n")
For comparison, try Nelder-Mead on the failed case

> fval <- hobbs.f(c(1, 1, 1))

> cat("Initial hobbs.f function at c(1,1,1) =", fval,
+     "\n")
Initial hobbs.f function at c(1,1,1) = 23520.58

> anm1 <- optim(c(1, 1, 1), hobbs.f, control = list(trace = 0))

> anm1
$par
[1] 31.830911 -16.423012  4.166667

$value
[1] 10071.14

$counts
function gradient
      250         NA

$convergence
[1] 0

$message
NULL

> cat("And again, try Nelder-Mead on the failed case but in log form\n")
And again, try Nelder-Mead on the failed case but in log form

> fval <- hobbsl.f(c(0, 0, 0))

> cat("Initial log form hobbsl.f function at c(0,0,0) =",
+     fval, "\n")
Initial log form hobbsl.f function at c(0,0,0) = 23520.58

> anml1 <- optim(c(0, 0, 0), hobbsl.f, control = list(trace = 0))

> anml1
$par
[1] 5.279378 3.893328 -1.160023

```



```
$value
[1] 2.587462
```

```
$counts
function gradient
      270      NA
```

```
$convergence
[1] 0
```

```
$message
NULL
```

```
> print(exp(anml1$par))
[1] 196.2478215  49.0739370  0.3134789
```

```
> cat("And again, try Nelder-Mead on the failed case but scaled start\n")
And again, try Nelder-Mead on the failed case but scaled start
```

```
> fval <- hobbs.f(c(100, 10, 0.1))
```

```
> cat("Initial hobbs.f function at c(100,10,0.1) =",
+     fval, "\n")
Initial hobbs.f function at c(100,10,0.1) = 10685.29
```

```
> anm2 <- optim(c(100, 10, 0.1), hobbs.f, control = list(trace = 0))
```

```
> anm2
$par
[1] 195.8632785  49.0609496  0.3137564
```

```
$value
[1] 2.587545
```

```
$counts
function gradient
      247      NA
```

```
$convergence
[1] 0
```

```
$message
NULL
```

Here we see that the very crude Nelder-Mead optimizer does reasonably well if we give it a scaled start or work with the logs of the parameters. the `nls()` does less well than we might hope. However, truthfully, this is a "bad" problem.

## 4.2 Reparametrization

Our model is an S curve, but our data only has the early up-turned part of the curve, so we can anticipate that we are going to have difficulty estimating one or both of the upper limit of the growth and the rate of getting there. These are the  $b_1$  and  $b_3$  parameters in the original model.

Doug Bates has suggested the model

$$yy = c_1 / (1 + \exp((c_2 - tt)/c_3))$$

where the  $c_1$  should be the same as  $b_1$  of the original model. However, now it is much clearer that  $c_2$  is the time at which we reach the half-way point to  $c_1$ . Moreover,  $c_3$  is now scaled in time units and controls how fast the curve rises. (The same arguments can be applied to declining data with  $c_3$  having its sign reversed.)

Looking at the graph of the data, we can provide a rough guess at the half-way point as (13, 100), making a rough guess of  $c_1 = 100$  and  $c_2 = 13$ . Plugging these values and the first data point (1, 5.308) into the model above gives us

$$c_3 = (c_2 - 1) / \log(c_1/y[1] - 1) = 3.331294$$

This gives the output

```
> dbn<-nls(y~c1/(1+exp((c2-t)/c3)), start=list(c1=200,c2=13,c3=3.33), trace=TRUE)
132.4117 : 200.00 13.00 3.33
3.087018 : 193.510427 12.315751 3.173038
2.587485 : 196.075031 12.414664 3.188793
2.587277 : 196.184968 12.417261 3.189077
2.587277 : 196.186251 12.417298 3.189083
```

where the starting function value and parameters are very close to the solution. However, it should be noted that random starts to `nls()` seem to give singular gradient or similar failures to those using the standard model.

## References

- Nash, J. C. (1979). *Compact numerical methods for computers : linear algebra and function minimisation*. Hilger, Bristol .:
- Nash, J. C. and M. Walker-Smith (1987). *Nonlinear parameter estimation: an integrated system in BASIC*. Marcel Dekker Inc.: New York. This book includes the complete software on diskette. Republished combined with the previous item in electronic form by Nash Information Services Inc.: Ottawa, 1996.
- Plummer, M. (2003). JAGS: A program for analysis of Bayesian graphical models using Gibbs sampling. Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003), March 20–22, Vienna, Austria.