Nonlinear regression plays an important role in my research and teaching activities. While I often use the ‘drm()’ function in the ‘drc’ package for my research work, I tend to prefer the ‘nls()’ function for teaching purposes, mainly because, in my opinion, the transition from linear models to nonlinear models is smoother, for beginners. One problem with ‘nls()’ is that, in contrast to ‘drm()’, it is not specifically tailored to the needs of biologists or students in biology. Therefore, now and then, I have to build some helper functions, to perform some specific tasks; I usually share these functions within the ‘aomisc’ package.

In this post, I would like to describe one of these helper functions, i.e. ‘gnlht()’, which is aimed at helping students (and practitioners; why not?) with one of their tasks, i.e. making some simple manipulations of model parameters, to obtain relevant biological information. Let’s see a typical example.

**Motivating example**

This is a real-life example, taken from a research published by Vischetti et al. in 1996. That research considered three herbicides for weed control in sugar beet, i.e. metamitron (M), phenmedipham (P) and cloridazon (C). Four soil samples were contaminated, respectively with: (i) M alone, (ii) M + P (iii) M + C and (iv) M + P + C. The aim was to assess whether the degradation speed of metamitron in soil depended on the presence of co-applied herbicides. To reach this aim, the soil samples were incubated at 20°C and sub-samples were taken in different times after the beginning of the experiment. The concentration of metamitron in those sub-samples was measured by HPLC analyses, performed in triplicate. The resulting dataset is available within the ‘aomisc’ package; we can load it and use the ‘lattice’ package to visualise the observed means over time.

# library(devtools)

# install\_github("OnofriAndreaPG/aomisc")

library(aomisc)

library(lattice)

data(metamitron)

xyplot(Conc ~ Time|Herbicide, data = metamitron,

xlab = "Time (d)", ylab = "Concentration",

scales = list(alternating = F),

panel = function(x, y, ...) {

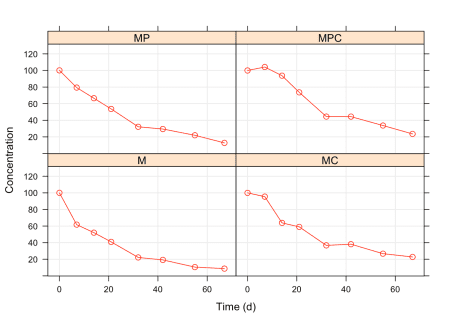
panel.grid(h = -1, v = -1)

fmy <- tapply(y, list(factor(x)), mean)

fmx <- tapply(x, list(factor(x)), mean)

panel.xyplot(fmx, fmy, col="red", type="b", cex = 1)

})



Considering this exemplary dataset, let’s see how we can answer the following research questions.

1. What is the degradation rate for metamitron, in the four combinations?
2. Is there a significant difference between the degradation rate of metamitron alone and with co-applied herbicides?
3. What is the half-life for metamitron, in the four combinations?
4. What are the times to reach 70 and 30% of the initial concentration, for metamitron in the four combinations?
5. Is there a significant difference between the half-life of metamitron alone and with co-applied herbicides?

**Fitting a degradation model**

The figure above shows a visible difference in the degradation pattern of metamitron, which could be attributed to the presence of co-applied herbicides. The degradation kinetics can be described by the following (first-order) model:

\[ C(t, h) = A\_h \, \exp \left(-k\_h \, t \right) \]

where \(C(t, h)\) is the concentration of metamitron at time \(t\) in each of the four combinations \(h\), \(A\_h\) and \(k\_h\) are, respectively, the initial concentration and degradation rate for metamitron in each combination.

The model is nonlinear and, therefore, we can use the ‘nls()’ function for nonlinear least squares regression. The code is given below: please, note that the two parameters are followed by the name of the factor variable in square brackets (i.e.: A[Herbicide] and k[Herbicide]). This is necessary to fit a different parameter value for each level of the ‘Herbicide’ factor.

#Fit nls grouped model

modNlin <- nls(Conc ~ A[Herbicide] \* exp(-k[Herbicide] \* Time),

start=list(A=rep(100, 4), k=rep(0.06, 4)),

data=metamitron)

summary(modNlin)

##

## Formula: Conc ~ A[Herbicide] \* exp(-k[Herbicide] \* Time)

##

## Parameters:

## Estimate Std. Error t value Pr(>|t|)

## A1 9.483e+01 4.796e+00 19.77 <2e-16 \*\*\*

## A2 1.021e+02 4.316e+00 23.65 <2e-16 \*\*\*

## A3 9.959e+01 4.463e+00 22.31 <2e-16 \*\*\*

## A4 1.116e+02 4.184e+00 26.68 <2e-16 \*\*\*

## k1 4.260e-02 4.128e-03 10.32 <2e-16 \*\*\*

## k2 2.574e-02 2.285e-03 11.26 <2e-16 \*\*\*

## k3 3.034e-02 2.733e-03 11.10 <2e-16 \*\*\*

## k4 2.186e-02 1.822e-03 12.00 <2e-16 \*\*\*

## ---

## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1

##

## Residual standard error: 9.701 on 88 degrees of freedom

##

## Number of iterations to convergence: 5

## Achieved convergence tolerance: 7.136e-06

For the sake of simplicity, I will neglige an accurate model check, although I need to point out that this is highly wrong. I’ll come back to this issue in another post.

**Working with model parameters**

Considering the research questions, it is clear that the above output answers the first one, as it gives the four degradation rates, \(k1\), \(k2\), \(k3\) and \(k4\). All the other questions can be translated into sets of linear/nonlinear functions (combinations) of model parameters. If we use the naming of parameter estimates in the nonlinear regression object, for the second question we can write the following functions: \(k1 – k2\), \(k1 – k3\) and \(k1 – k4\). The third question requires some slightly more complex math: if we invert the equation above for one herbicide, we get to the following inverse:

\[ t = \frac{- log \left[\frac{C(t)}{A} \right] }{k} \]

I do not think this is complex enough to scare the biologists, is it? The half-life is the time required for C(t) to drop to half of the initial value, so that \(C(t)/A\) is equal to \(0.5\). Thus:

\[ t\_{1/2} = \frac{- \log \left[0.5 \right] }{k} \]

Analogously, we can answer the question 4, by replacing \(0.5\) respectively with \(0.7\) and \(0.3\). The difference between the half-lives of metamitron alone and combined with the second herbicide can be calculated by:

\[ \frac{- \log \left[0.5 \right] }{k\_1} – \frac{- \log \left[0.5 \right] }{k\_2} = \frac{k\_2 – k\_1}{k\_1 \, k\_2} \, \log(0.5)\]

The other differences are obtained analogously.

**Inferences and hypotheses testing**

All parameter estimates are characterised by some uncertainty, which is summarised by way of the standard errors (see the code output above). Clearly, such an uncertainty propagates to their combinations. As for the first question, the combinations are linear, as only subtraction is involved. Therefore, the standard error for the difference can be easily calculated by the usual law of propagation of errors,

In R, linear combinations of model parameters can be built and tested by using the ‘glht()’ function in the ‘multcomp’ package. However, I wanted to find a general solution, that could handle both linear and nonlinear combinations of model parameters. Such a solution should be based on the ‘delta method’. Unfortunately, the function ‘deltaMethod()’ in the ‘car’ package is not flexible enough to the aims of my students and mine.

Therefore, I wrote a wrapper for the ‘deltaMethod()’ function, which I named ‘gnlht()’, as it might play for nonlinear combinations the same role as ‘glht()’ for linear combinations. To use this function, apart from loading the ‘aomisc’ package, we need to prepare a list of formulas. Care needs to be taken to make sure that the element in the formulas correspond to the names of the estimated parameters in the model object, as returned by the ‘coef()’ method. In the box below, I show how we can calculate the differences between the degradation rates.

funList <- list(~k1 - k2, ~k1 - k3, ~k1 - k4)

gnlht(modNlin, funList)

## form Estimate SE t-value p-value

## 1 k1 - k2 0.01686533 0.004718465 3.574325 5.727311e-04

## 2 k1 - k3 0.01226241 0.004951372 2.476568 1.517801e-02

## 3 k1 - k4 0.02074109 0.004512710 4.596150 1.430392e-05

The very same code can be used for nonlinear combinations of model parameters. In order to calculate the half-lives, we can use the following code:

funList <- list(~ -log(0.5)/k1, ~ -log(0.5)/k2,

~ -log(0.5)/k3, ~ -log(0.5)/k4)

gnlht(modNlin, funList)

## form Estimate SE t-value p-value

## 1 -log(0.5)/k1 16.27089 1.576827 10.31876 7.987827e-17

## 2 -log(0.5)/k2 26.93390 2.391121 11.26413 9.552915e-19

## 3 -log(0.5)/k3 22.84747 2.058588 11.09861 2.064093e-18

## 4 -log(0.5)/k4 31.70942 2.643329 11.99601 3.257067e-20

Instead of writing ‘0.5’, we can introduce a new model term, e.g. ‘prop’, as a ‘constant’, in the sense that it is not an estimated parameter. We can pass a value for this constant in a data frame, by using the ‘const’ argument:

funList <- list(~ -log(prop)/k1, ~ -log(prop)/k2,

~ -log(prop)/k3, ~ -log(prop)/k4)

gnlht(modNlin, funList, const = data.frame(prop = 0.5))

## form prop Estimate SE t-value p-value

## 1 -log(prop)/k1 0.5 16.27089 1.576827 10.31876 7.987827e-17

## 2 -log(prop)/k2 0.5 26.93390 2.391121 11.26413 9.552915e-19

## 3 -log(prop)/k3 0.5 22.84747 2.058588 11.09861 2.064093e-18

## 4 -log(prop)/k4 0.5 31.70942 2.643329 11.99601 3.257067e-20

This is very flexible, because it lets us to calculate, altogether, the half-life and the times required for the concentration to drop to 70 and 30% of the initial value:

funList <- list(~ -log(prop)/k1, ~ -log(prop)/k2,

~ -log(prop)/k3, ~ -log(prop)/k4)

gnlht(modNlin, funList, const = data.frame(prop = c(0.7, 0.5, 0.3)))

## form prop Estimate SE t-value p-value

## 1 -log(prop)/k1 0.7 8.372564 0.8113927 10.31876 7.987827e-17

## 2 -log(prop)/k1 0.5 16.270892 1.5768267 10.31876 7.987827e-17

## 3 -log(prop)/k1 0.3 28.261979 2.7388937 10.31876 7.987827e-17

## 4 -log(prop)/k2 0.7 13.859465 1.2304069 11.26413 9.552915e-19

## 5 -log(prop)/k2 0.5 26.933905 2.3911214 11.26413 9.552915e-19

## 6 -log(prop)/k2 0.3 46.783265 4.1532956 11.26413 9.552915e-19

## 7 -log(prop)/k3 0.7 11.756694 1.0592942 11.09861 2.064093e-18

## 8 -log(prop)/k3 0.5 22.847468 2.0585881 11.09861 2.064093e-18

## 9 -log(prop)/k3 0.3 39.685266 3.5756966 11.09861 2.064093e-18

## 10 -log(prop)/k4 0.7 16.316814 1.3601864 11.99601 3.257067e-20

## 11 -log(prop)/k4 0.5 31.709415 2.6433295 11.99601 3.257067e-20

## 12 -log(prop)/k4 0.3 55.078163 4.5913724 11.99601 3.257067e-20

The differences between the half-lives (and other degradation times) can be calculated as well:

funList <- list(~ (k2 - k1)/(k1 \* k2) \* log(prop),

~ (k3 - k1)/(k1 \* k3) \* log(prop),

~ (k4 - k1)/(k1 \* k4) \* log(prop))

gnlht(modNlin, funList, const = data.frame(prop = c(0.7, 0.5, 0.3)))

## form prop Estimate SE t-value

## 1 (k2 - k1)/(k1 \* k2) \* log(prop) 0.7 5.486900 1.473859 3.722813

## 2 (k2 - k1)/(k1 \* k2) \* log(prop) 0.5 10.663013 2.864235 3.722813

## 3 (k2 - k1)/(k1 \* k2) \* log(prop) 0.3 18.521287 4.975078 3.722813

## 4 (k3 - k1)/(k1 \* k3) \* log(prop) 0.7 3.384130 1.334340 2.536183

## 5 (k3 - k1)/(k1 \* k3) \* log(prop) 0.5 6.576577 2.593100 2.536183

## 6 (k3 - k1)/(k1 \* k3) \* log(prop) 0.3 11.423287 4.504125 2.536183

## 7 (k4 - k1)/(k1 \* k4) \* log(prop) 0.7 7.944250 1.583814 5.015900

## 8 (k4 - k1)/(k1 \* k4) \* log(prop) 0.5 15.438524 3.077917 5.015900

## 9 (k4 - k1)/(k1 \* k4) \* log(prop) 0.3 26.816185 5.346236 5.015900

## p-value

## 1 3.468973e-04

## 2 3.468973e-04

## 3 3.468973e-04

## 4 1.297111e-02

## 5 1.297111e-02

## 6 1.297111e-02

## 7 2.718445e-06

## 8 2.718445e-06

## 9 2.718445e-06

The possibility of passing constants in a data.frame adds flexibility with respect to the ‘deltaMethod()’ function in the ‘car’ package. For example, we can use this method to make predictions:

funList <- list(~ A1 \* exp (- k1 \* Time), ~ A2 \* exp (- k2 \* Time),

~ A3 \* exp (- k3 \* Time), ~ A4 \* exp (- k4 \* Time))

pred <- gnlht(modNlin, funList, const = data.frame(Time = seq(0, 67, 1)))

head(pred)

## form Time Estimate SE t-value p-value

## 1 A1 \* exp(-k1 \* Time) 0 94.83198 4.795948 19.77336 3.931107e-34

## 2 A1 \* exp(-k1 \* Time) 1 90.87694 4.381511 20.74101 1.223613e-35

## 3 A1 \* exp(-k1 \* Time) 2 87.08684 4.015039 21.69016 4.511113e-37

## 4 A1 \* exp(-k1 \* Time) 3 83.45482 3.695325 22.58389 2.205772e-38

## 5 A1 \* exp(-k1 \* Time) 4 79.97427 3.421034 23.37722 1.623774e-39

## 6 A1 \* exp(-k1 \* Time) 5 76.63888 3.190531 24.02072 2.050113e-40

tail(pred)

## form Time Estimate SE t-value p-value

## 267 A4 \* exp(-k4 \* Time) 62 28.78518 2.657182 10.83297 7.138133e-18

## 268 A4 \* exp(-k4 \* Time) 63 28.16278 2.648687 10.63273 1.824651e-17

## 269 A4 \* exp(-k4 \* Time) 64 27.55384 2.639403 10.43942 4.525865e-17

## 270 A4 \* exp(-k4 \* Time) 65 26.95807 2.629361 10.25270 1.090502e-16

## 271 A4 \* exp(-k4 \* Time) 66 26.37517 2.618594 10.07227 2.555132e-16

## 272 A4 \* exp(-k4 \* Time) 67 25.80489 2.607131 9.89781 5.827812e-16

Although this is not very fast, in contrast to the ‘predict()’ method for ‘nls’ objects, it has the advantage of reporting standard errors.

Hope this is useful. Happy coding!