Pharmacokinetics and pharmacodynamics

Peter Dalgaard

April/May 2003

The `nls` function

1–12

Overview

Overview

- Simple usage

Overview

- Simple usage
- Summary output

Overview

- Simple usage
- Summary output
- Profiling

Overview

- Simple usage
- Summary output
- Profiling
- Confidence intervals

Overview

- Simple usage
- Summary output
- Profiling
- Confidence intervals
- Self-starting models

Overview

- Simple usage
- Summary output
- Profiling
- Confidence intervals
- Self-starting models
- Writing self-starting models

Simple usage

```
nlsout <- nls(y ~ A*exp(-alpha*t), start=list(A=2,alpha=0.05))
```

Simple usage

```
nlsout <- nls(y ~ A*exp(-alpha*t), start=list(A=2,alpha=0.05))
```

- Right side of model formula is arithmetic expression (no special interpretation for factors, etc.)

Simple usage

```
nlsout <- nls(y ~ A*exp(-alpha*t), start=list(A=2,alpha=0.05))
```

- Right side of model formula is arithmetic expression (no special interpretation for factors, etc.)
- Notice that this is a *vectorized* expression (very useful for ODE solvers)

Simple usage

```
nlsout <- nls(y ~ A*exp(-alpha*t), start=list(A=2,alpha=0.05))
```

- Right side of model formula is arithmetic expression (no special interpretation for factors, etc.)
- Notice that this is a *vectorized* expression (very useful for ODE solvers)
- The `start` argument defines which are parameters to be estimated and starting values for the iterative algorithm

Summary output

```
> summary(nlsout)

Formula: y ~ A * exp(-alpha * t)

Parameters:
      Estimate Std. Error t value Pr(>|t|)
A      4.75402    0.29408  16.166 5.88e-08 ***
alpha  0.18364    0.02023   9.079 7.95e-06 ***

Residual standard error: 0.3892 on 9 degrees of freedom

Correlation of Parameter Estimates:
           A
alpha 0.6724
```

Profiling

```
par(mfrow=c(2,1))
plot(profile(nlsout))
```

Profiling

```
par(mfrow=c(2,1))
plot(profile(nlsout))
```

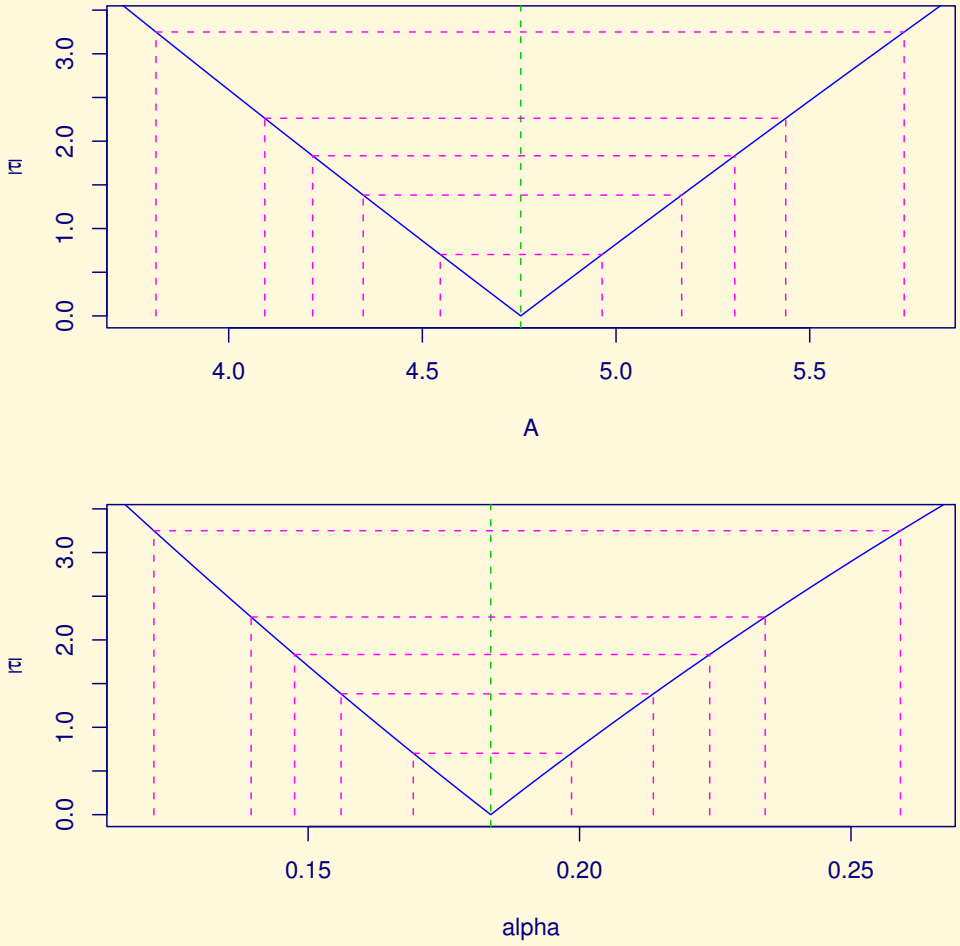
- Calculate profile t statistics, i.e. signed values of $\sqrt{\Delta\text{SSD}}/\text{SE}(\hat{\theta})$ for varying values of θ, maximized over other parameters and signed according to which side of $\hat{\theta}$ you're on.

Profiling

```
par(mfrow=c(2,1))
plot(profile(nlsout))
```

- Calculate profile t statistics, i.e. signed values of $\sqrt{\Delta\text{SSD}}/\text{SE}(\hat{\theta})$ for varying values of θ, maximized over other parameters and signed according to which side of $\hat{\theta}$ you're on.
- Plots of $|t|$ with indication of approximate confidence levels (.99, .95, .90, .80, .50)

Profile plots



Confidence intervals

```
> confint(nlsout)
Waiting for profiling to be done...
            2.5%     97.5%
A      4.0931837 5.4381261
alpha 0.1395008 0.2342045
```

Confidence intervals

```
> confint(nlsout)
Waiting for profiling to be done...
           2.5%     97.5%
A     4.0931837 5.4381261
alpha 0.1395008 0.2342045
```

- The same procedure as in profile plots, but showing results numerically

Supplying gradient information

Supplying gradient information

- Often not worth it. . . If not supplied, numeric differentiation will be done.

Supplying gradient information

- Often not worth it. . . If not supplied, numeric differentiation will be done.
- Right hand side must evaluate to something that has a `gradient` attribute, e.g.

Supplying gradient information

- Often not worth it. . . If not supplied, numeric differentiation will be done.
- Right hand side must evaluate to something that has a `gradient` attribute, e.g.

```
nls(y~structure(A*exp(-alpha*t),
          gradient=cbind(A=exp(-alpha*t),
                         alpha=-A*t*exp(-alpha*5))),
    start=list(A=10,alpha=3.5))
```

Supplying gradient information

- Often not worth it...If not supplied, numeric differentiation will be done.
- Right hand side must evaluate to something that has a `gradient` attribute, e.g.

```
nls(y~structure(A*exp(-alpha*t),
          gradient=cbind(A=exp(-alpha*t),
                         alpha=-A*t*exp(-alpha*5))),
    start=list(A=10,alpha=3.5))
```

- The `deriv()` function can do much of the hard work for you if the model is given as a simple arithmetic expression.

Selfstarting models

Selfstarting models

- How to get starting values?

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations
- Typical tricks:

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations
- Typical tricks:
  - transform to linearity

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations
- Typical tricks:
  - transform to linearity
  - calculate "landmarks" as function of parameters (AUC, initial slope, position and value of maximum) estimate them empirically and solve for parameters

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations
- Typical tricks:
  - transform to linearity
  - calculate "landmarks" as function of parameters (AUC, initial slope, position and value of maximum) estimate them empirically and solve for parameters
- Idea: Store algorithm for starting value within model object

Selfstarting models

- How to get starting values?
- Mostly an art, but can be worked out for typical situations
- Typical tricks:
  - transform to linearity
  - calculate "landmarks" as function of parameters (AUC, initial slope, position and value of maximum) estimate them empirically and solve for parameters
- Idea: Store algorithm for starting value within model object
- Standard models supplied: `SSfol`, etc.

Writing your own selfstart models

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`
- `mCall` is the matched call. Used to access the names passed for the parameters.

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`
- `mCall` is the matched call. Used to access the names passed for the parameters.
- `LHS` is the left hand side (response). NB: Unevaluated.

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`
- `mCall` is the matched call. Used to access the names passed for the parameters.
- `LHS` is the left hand side (response). NB: Unevaluated.
- `data` is the modelling frame.

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`
- `mCall` is the matched call. Used to access the names passed for the parameters.
- `LHS` is the left hand side (response). NB: Unevaluated.
- `data` is the modelling frame.
- Return value should be a named list.

Writing your own selfstart models

- Main thing to supply is the initializer. Must be `function(mCall, LHS, data)`
- `mCall` is the matched call. Used to access the names passed for the parameters.
- `LHS` is the left hand side (response). NB: Unevaluated.
- `data` is the modelling frame.
- Return value should be a named list.
- `selfStart()` constructor function does the rest. If used on a model formula, `deriv` is used for gradient info.

Selfstart example

```
initexp <-
function(mCall, data, LHS) {
     y <- eval(LHS,data)
     cc <- coef(lm(log(y)~t,data=data))
     l <- list(exp(cc[1]),-cc[2])
     names(l) <- mCall[c("A","a")]
     l
}

SSexp<-selfStart(~A*exp(-a*t),
                 initial = initexp,
                 parameters = c("A","a"))

summary(nls(y~SSexp(t,A,a)))
summary(nls(y~SSexp(t,B,b))) # won't work without the mCall stuff
```

Partially linear models

Partially linear models

- Some models have parameter that act linearly on the response

Partially linear models

- Some models have parameter that act linearly on the response
- Better algorithms available

Partially linear models

- Some models have parameter that act linearly on the response
- Better algorithms available
- *May* be worth trying if there are speed or convergence problems

Partially linear models

- Some models have parameter that act linearly on the response
- Better algorithms available
- *May* be worth trying if there are speed or convergence problems
- `algorithm="plinear"` argument

Partially linear models

- Some models have parameter that act linearly on the response
- Better algorithms available
- *May* be worth trying if there are speed or convergence problems
- `algorithm="plinear"` argument
- Arrange that r.h.s. becomes a *matrix* to be multiplied by the linear parameters.

Partially linear models

- Some models have parameter that act linearly on the response
- Better algorithms available
- *May* be worth trying if there are speed or convergence problems
- `algorithm="plinear"` argument
- Arrange that r.h.s. becomes a *matrix* to be multiplied by the linear parameters.

```
nls(y~cbind(exp(-a*t),exp(-b*t)),start=list(a=.3,b=.01),
    algorithm="plinear")
```

Deep tricks

Deep tricks

- `nlsModel()` sets up an object that can be modified and queried. `nls()` returns one of these in its `$m` component

Deep tricks

- `nlsModel()` sets up an object that can be modified and queried. `nls()` returns one of these in its `$m` component
- Current status is kept in an internal environment

Deep tricks

- `nlsModel()` sets up an object that can be modified and queried. `nls()` returns one of these in its `$m` component
- Current status is kept in an internal environment
- `m$setPar()` changes the parameter; `m$deviance()` gets the residual sum of squares

Deep tricks

- `nlsModel()` sets up an object that can be modified and queried. `nls()` returns one of these in its `$m` component
- Current status is kept in an internal environment
- `m$setPar()` changes the parameter; `m$deviance()` gets the residual sum of squares
- NB: Using `setPar()` destroys the fit. Cannot just save the fit in a different variable because of the embedded environment

Deep tricks

- `nlsModel()` sets up an object that can be modified and queried. `nls()` returns one of these in its `$m` component
- Current status is kept in an internal environment
- `m$setPar()` changes the parameter; `m$deviance()` gets the residual sum of squares
- NB: Using `setPar()` destroys the fit. Cannot just save the fit in a different variable because of the embedded environment
- Used by the profiler, and can also be used for drawing approximate confidence regions.