AIFFD Preliminaries

Derek H. Ogle

Contents

I prefer to not show significance stars for hypothesis test output and set contrasts in such a manner as to force R output to match SAS output for linear model summaries (discussed in Section 0.1.5. These options are set below.

```
> options(show.signif.stars=FALSE,contrasts=c("contr.sum","contr.poly"))
```

I also set the random number seed so that that random data used below remains consistent from run-to-run.

```
> set.seed(983452)
```

0.1 Some Basics of R (and SAS)

As the authors of AIFFD noted in the preface, they had to make a choice to use a common software for the book, they chose SAS, and any choice they would have made would have had some limitations. One of the goals of this portion of the fishR website is to attempt to perform all boxed example analyses in the book with R rather than SAS. R and SAS are completely different "animals" that can basically perform the same analyses. Converting from SAS to R will provide some difficulties because of these differences. The items identified in the sections below are differences or difficulties that are more general in nature and will, therefore, permeate through all analyses. I have chosen to pull them together in this one place to keep the further analyses streamlined. Please consider these items carefully before continuing with the other analyses.

0.1.1 Learning R

Most of the AIFFD R examples on the fishR website assume that you have a working familiarity with R, just as the book assumes you have a working familiarity with SAS. Thus, the examples demonstrate how to perform specific analyses in R but they don't necessarily demonstrate basic functioning of R. Therefore, you may have to familiarize yourself with R with some of the following resources:

- An Introduction to R
- Dalgaard's Introductory Statistics with R. Springer, 2nd edition.
- Verzani's Using R for Introductory Statistics.} Chapman & Hall/CRC
- Crawley's Statistics: An Introduction using R.} Wiley, 2005.

0.1.2 Reading Data into R

In most of the boxed examples the data are entered directly into SAS with the `CARDS` statement. In the R versions of the boxed examples I have chosen to enter these data into external text files that can then be read into R with `read.table()`. All of these data files are available via links in the corresponding example. The most efficient way to read these data into R is to first set the R working directory to where you saved the data file with `setwd()`. The `setwd()` function requires one argument which is the drive letter followed by the path to the folder containing the data file. This drive and path must be contained in double-quotes and the "slashes" between the drive and the path and between all folders in the path must be forward-slashes. For example, I would change the R working directory to where I saved the data file for Box 5.6 with the following code.

```
> setwd("c:/aaaWork/web/fishR/BookVignettes/AIFFD/")
```

It **MUST** be noted that this is the drive and path to the data file on **MY** computer. This will not work on YOUR computer unless your directory structure is exactly the same as mine (which is unlikely). Thus, if you are attempting to recreate the R box analyses, then you will have to make sure NOT to copy this command exactly; rather you will have to change it to exactly where you saved the data file.

You should also note that all of the data files have been saved such that the first row contains the names for the variables. In the R language it is said that these files have a *header*. The default behavior for `read.table()` is to assume that there is not a header; thus, you must make sure to tell `read.table()` that the data file has a header. Thus, every time you use `read.table()` with examples that I have authored you will need to include the file name (in quotes) as the first argument and `header=TRUE` as the second argument. For example, after setting the working directory above, I would read the data file for Box 5.6 with the following code.

```
> d6 <- read.table("data/box5_6.txt",header=TRUE)
```

IFAR REFERENCE HERE

0.1.3 Required Packages

One of the major advantages of R is that the base R can be easily extended with contributed packages. Packages are sets of routines written by anybody in the world that either modify or, more likely, extend the abilities of base R. Many such packages are stored in the Comprehensive R Archive Network (CRAN), while others are stored on "Forge" sites (e.g., r-forge.r-project, rforge.net, or sourceforge) or GitHub, while still others are stored on personal webpages. To use a particular package, it must be installed on your computer. Installing packages from CRAN is fairly easy using built-in menu options in the R GUI (e.g., on Windows go to the *Packages* menu and then *Install package(s)...* sub-menu, choose a mirror site, and then choose

the package(s) you want to install). Most packages stored on a "Forge" site or GitHub will have a small set of commands or a script on the specific site that will install the package when copied into R. Packages on personal websites will likely have to be downloaded from that website and installed via a menu item in the R GUI (e.g., on Windows go to the *Packages* menu and then *Install package(s) from local zip file(s)...* sub-menu item and then browse to where you downloaded the ZIP file). Packages only have to be installed once on a computer – except that it will have to be re-installed when upgrading.

Once a package is installed on a computer then R must be told to load that package for use. A package must be loaded every time you open R and want to use the package but it only needs to be loaded once per R session. A package is loaded by including the package name (without quotes) as the single argument to `library()`. For example, the `car` package is loaded as follows.

```
> library(car)
```

In the R analysis box examples, all required packages will be loaded at the beginning of the example.

0.1.4 Philosophical Difference between R and SAS (just one!)

A critical philosophical difference between R and SAS is that SAS will generally print out a great deal of information from every PROC that is run. R, on the other hand, prints out very little. The philosophy of R is that you need to explicitly ask for the results, whereas the philosophy of SAS is to print out everything it knows and let you sort through to find what you want.

The philosophy of R is apparent in the two main types of functions – constructor and extractor functions. A constructor function is a function that performs some task and likely stores a great deal of information in a seemingly invisible way. The results of a constructor function are usually saved into an R object. An extractor function is a function that acts on an R object to extract specific pieces of information from it. The typical analysis then is to use a constructor function to perform some analysis and then to use an extractor function to extract just the information that you require.

For example, suppose that you have two variables – `Y` and `X` – for which you want to construct a linear regression model to predict `Y` from `X`. The `lm()` function is a constructor function that performs the linear regression (this function is described in much more detail in the SECTION XXX). In the example below, the results are stored into an object called `lm1` but you do not immediately see any results. If you would like to see the coefficients from the regression fit you can submit the saved `lm` object to the `coef()` extractor function. If you want to see the ANOVA table then submit the saved `lm` object to the `anova()` extractor function.

```
> lm1 <- lm(Y~X)
> coef(lm1)
```

```
(Intercept)           X
  26.973878    2.927415
```

```
> anova(lm1)
```

```
Analysis of Variance Table

Response: Y
          Df Sum Sq Mean Sq F value    Pr(>F)
X          1 645.07  645.07  653.42 < 2.2e-16
Residuals 48  47.39    0.99
```

The point is that you will have to change your attitude about getting results from R as compared to SAS. SAS prints everything you might ever possibly want (and a lot that you probably don't understand) whereas R will only print results (and very specific results) when you ask for them.

0.1.5 Types (I,II,III) of Sums-of-Squares

One of the major arguments between proponents of R and proponents of SAS is over the definitions and uses of the variety of types of sums-of-squares (SS). SAS defines four types of SS, two of which are used throughout the AIFFD book – Type-I and Type-III. I will attempt to describe each of these SS below and briefly describe how these will be handled in R.

SAS' type-I SS are sometimes called *sequential* or *variables-in-order*SS because they are the SS explained by adding a variable to the model AFTER all previous variables have been added and BEFORE adding any subsequent variables. For example, consider a situation where there are three factor variables (`A`, `B`, and `C`) and all of the two- and three-way interactions between these variables. In general, this model can be written as `Y = A + B + C + A:B + A:C + B:C + A:B:C}`.

The type-I SS for `A:B`, for example, would be the SS explained by `A:B` AFTER the main `A`, `B`, and `C` effects have been considered but withOUT considering the effects of `A:C`, `B:C`, or `A:B:C`. The type-I SS for `B:C` would be the SS explained by `B:C` AFTER all of the other terms except for `A:B:C` have been considered. In other words, the type-I SS are the SS explained by that term in the order that that term was entered into the model. Which, of course, means that the order that terms are entered into the model is important.

SAS' type-III SS are sometimes called *marginal* or *variables-added-last* SS because they are the SS explained by adding a variable to the model AFTER ALL other variables have been included in the model, regardless of the order the variables were entered into the model. Using the same example as before, the type-III SS for `A:B` would be the SS explained by `A:B` AFTER ALL other variables have been considered.

If the experiment has equal numbers of experimental units in each cell defined by each combination of the factors (a so-called balanced design) then the type-I and type-III SS will be identical. However, if the design is unbalanced, some cells are missing observations, or a quantitative explanatory variables is included in the model (i.e., ANCOVA) then the type-I and type-III SS will not be equivalent.

The type-I SS are computed in R by submitting a linear model or analysis of variance object to the `anova()` extractor function.

Type type-III SS as defined by SAS are a bit more difficult to extract in R. The `car` package has `Anova()` (note the capital "A") which has a `type=` argument which can be set equal to `III`. However, under the default contrast structure of R (i.e., `contr.treatment`), this function will NOT return the SAS type-III SS. However, if the contrast structure is changed (to either `contr.sum` or `contr.helmert` then `Anova()` with the `type="III"` argument will produce SAS type-III SS ONLY for the situation where the explanatory variables are all factors. In situations where at least one of the explanatory variables is quantitative (e.g., ANCOVA) then the type-III SS of SAS cannot be computed in R. As changing the contrast structure does not affect the type-I or type-II (see below) SS, then it is generally a good idea to change the contrast structure early in your R script. Thus, in many of the AIFFD examples, you will see the command below at the beginning of the vignette to change the contrast structure.

```
> options(contrasts=c("contr.sum","contr.poly"))
```

Fox suggested that any default SS and hypothesis testing (in contrast to very carefully defined hypothesis tests, model fitting, and SS calculations) should follow the principle-of-marginality where one ignores all higher-order "relatives" when testing lower-order terms. For example, in the three-factor example above, one would ignore `A:B`, `A:C`, and `A:B:C` when assessing the significance of `A`. In other words, the type-II SS for `A` would be the SS explained by `A` AFTER considering `B`, `C`, and `B:C` but withOUT considering `A:B`, `A:C`, and `A:B:C`. Similarly, the SS for `A:B` would be the SS explained by `A:B` AFTER considering `A`, `B`, `C`, `A:C`, and `B:C` but not `A:B:C` (i.e., after considering all terms that do NOT contain `A:B` as part of the term).

Further discussion of these different types of SS can be found in the following links:

- Fox's An R and S-Plus companion to applied regression
- Littell *et al.*'s SAS for Linear Models

- These notes from the University of Minnesota

Noe that in ANCOVA situations I will use the type-II SS from `Anova()` despite the fact that the R results will not perfectly match the SAS results shown in the AIFFD book. In addition, for all other linear model analyses I will show the type-II results, in addition to what was illustrated in the AIFFD book (type-I or type-III).

0.1.6 Least-Squares Means

The second major argument between proponents of R and proponents of SAS is over the use of so-called "least-squares means." The terminology "least-squares means" is largely a SAS construct and near synonyms include "adjusted means", "marginal means", or "estimated marginal means." In general, a "least-squares mean" is the mean for a group after having controlled for other variables – i.e., other factors or quantitative covariates. The most common "least-squares mean" is the calculation of adjusted group means after holding a quantitative covariate at a typical value (say the mean) in an ANCOVA.

In general, users of R argue for a more general approach that has been implemented in Fox's `effects` package. However, a method to produce least-squares means has been provided by `lsmeans()` in the `lsmeans` package. The use of this function will be demonstrated in the various chapter vignettes as least-squares means are introduced.

More resources on least-squares means can be found at following links.

- Lenth's `lsmeans` package vignette
- On Biostatistics and Clinical Trial blog
- SAS support

0.2 Linear Models in R

Linear models – e.g., analysis of variance, simple linear regression, analysis of covariance, multiple linear regression – are used throughout the boxed examples in the AIFFD book. In R, all of these models are implemented with one constructor function – `lm()` – that can receive a variety of formula types. The `lm()` function will then fit one of the linear models depending on the types of variables present in the formula. In this section, I briefly describes the use of `lm()` for these variety of models.

0.2.1 R Formulae

An R formula consists of a left-hand-side (the response or dependent; LHS) and a right-hand-side (the explanatory, predictor, or independent; RHS) separated by a tilde. For the purposes of the boxed examples in AIFFD, the LHS will (nearly always) consist of a continuous response variable. The RHS, on the other hand, will consist of a single explanatory variable or some function of several explanatory variables. For our purposes, we need to note that explanatory variables can be "added" to the RHS by including a "plus sign" followed by the variable name and interaction terms are symbolized by the two variables forming the interaction separated by a colon (e.g., `A:B` represents the interaction between `A` and `B`). Finally, note that R uses a short-hand notation of `A*B` to note that the RHS should include the two main effect terms and an interaction term (i.e., `A + B + A:B`).

0.2.2 `lm()`

The `lm()` function requires two arguments. The first argument is a model formula as described in the previous section. Different model formulae provide different analyses depending on the variables in the formula. The

second argument, the `data=` argument, tells R which data frame the variables in the formula can be found. The results of `lm()` should be saved to an object so that that object can be submitted to a variety of extractor functions to return specific results.

The variety of linear models produced by different formulae supplied to `lm()`. Note that the generic variables in these formula are defined as follows: `Y` is continuous response variable, `X1` and `X2` are continuous explanatory variables, and `G1` and `G2` are categorical group factor explanatory variables.

R Formula	Linear Model	Example
`Y~X1`	Simple Linear Regression	Box 6.4
`Y~G1`	One-Way ANOVA	Box 3.8
`Y~G1*G2`	Two-way ANOVA (with interaction)	Box 3.13
`Y~G1+G2`	Two-way ANOVA (withOUT interaction)	Box 5.5 (last section)
`Y~X1*G1`	One-way Indicator Variable Regression (ANCOVA-like model)	Box 3.11
`Y~X1+G1`	One-way ANCOVA	
`Y~X1*G1*G2`	Two-way Indicator Variable Regression	
`Y~X1*X2`	Multiple Linear Regression (with interaction)	Box 7.6

0.2.3 `lm()` Extractor Functions

A number of functions can be used to extract specific information from an object saved from a `lm()` call. Extractor functions, and corresponding packages, for a linear model object (note that `lm1` represents a saved linear model object). Note that functions in the `base` package do not require any extra packages to be loaded.

Function Call	Package	Description
`anova(lm1)`	`base`	Extracts the ANOVA table using type-I SS. See Section 0.1.5.
`coef(lm1)`	`base`	Extracts the values of the parameter coefficients.
`confint(lm1)`	`base`	Extracts confidence intervals for the parameter coefficients.
`summary(lm1)`	`base`	Extracts the parameter coefficient values, SEs, and default t-test and p-values. Also, extracts coefficient of determination (unadjusted and adjusted), overall F-test and p-value, and rMSE.
`predict(lm1)`	`base`	Extracts predictions using the linear model for each individual in the data frame. Modifications (i.e., other arguments) allow predicting other values.
`Anova(lm1,type="III")`	`car`	Extracts the ANOVA table using type-III SS. See Section 0.1.5.
`Anova(lm1,type="II")`	`car`	Extracts the ANOVA table using type-II SS. See Section 0.1.5.
`lsmeans(lm1,~factor)`	`FSA`	Extracts least-squares means for the given factor.
`fitPlot(lm1)`	`FSA`	Constructs a "fitted-line plot" (specifics depends on model; does not work for all model types)
`residPlot(lm1)`	`FSA`	Constructs a residual plot.
`hist(lm1\$residuals)`	`base`	Constructs histogram of model residuals.
`ad.test(lm1\$residuals)`	`nortest`	Performs Anderson-Darling test of normality on model residuals.
`leveneTest(lm1)`	`car`	Performs Levene's Homogeneity of Variance test on model groups.
`outlierTest(lm1)`	`car`	Performs a test for outliers on model residuals.

0.3 Model Optimization in R

A wide variety of fisheries models require the use of an optimization algorithm in order to find the suite of parameters that result in the "best fit" of the model to the data. These optimizations usually are the minimization of a *least-squares* or the maximization of a *maximum likelihood* criterion. Some models and optimizations are so well formed that they are "optimized" with traditional statistical models based on

well-known theoretical underpinnings. Other models are more complex or the underlying theory is less well-known and the "best-fit" suite of parameters are found by first defining a function to be minimized or maximized and then "searching" through a large list of parameter combinations to find the set that minimizes or maximizes the given function. On this page I briefly introduce how to perform optimizations in these latter situations.

There are a wide variety of optimization algorithms and the theory behind them is complex. Most of these algorithms need starting values for the parameters to "get going." The optimization function is evaluated at this point and the algorithm "searches" for another set of parameters, based on some criterion specific to that algorithm, that are "downhill" (when minimizing) of the initial set. The optimization function is then evaluated at this new parameter set, the algorithm "searches" for another set of parameters "downhill", etc. This iterative process is continued until a "convergence critierion" is met. The set of parameters at this iterative step is then considered the "best-fit" solution. Each algorithm uses a different searching procedure, different convergence criterion, etc. that makes a full description more complex than what is given here. However, here I give an overall impression of how most of the procedures work.

0.3.1 Excel's Solver

Many fisheries scientists are familiar with Excels "solver" routine. Indeed, the routine is referred to several times in the AIFFD book. The solver routine apparently uses the Generalized Reduced Gradient (GRG2) algorithm as described here. This algorithm has not been implemented in R.

0.3.2 Optimization in R

0.3.2.1 Writing a Function to be Optimized Optimization of functions in R generally uses two functions: `optimize()` for one parameter models and `optim()` for more complex models. Both of these functions requires a vector containing the starting values for the parameters and a function to be optimized (minimized or maximized). Generally, the function to be optimized must be set up prior to calling `optimize()` or `optim()`. Functions are created in R with `function()`. The arguments to `function()` are the arguments that will be required by the function you are creating. The results of `function()` are assigned to an object that will be the name of the function you are creating. The calculations are then contained within a "{" and a "}" and the last line is the object that is returned by the function. For example, the function below, called `SSE`, computes and returns the sum of squared deviations from the parameter, `theta`,

```
> SSE <- function(theta,obs) {
    sum((obs-theta)^2)
  }
```

As a second example, the following function, called `normLH`, computes the log-likelihood, assuming a normal distribution, for two parameters, `mu` and `sig`.

```
> normLH <- function(pars,obs) {
    mu <- pars[1]
    sig <- pars[2]
    sum(log(dnorm(obs,mean=mu,sd=sig)))
  }
```

0.3.2.2 `optimize()` for Single Parameter Models The `optimize()` function requires the function to be optimized as the first argument and a range of values to be searched over for the single parameter as the second argument. In addition, `maximize=TRUE`, may be used to force `optimize()` to maximize rather than minimize (the default) the optimization function. Finally, any other arguments that need to be sent to

the optimization function can be included in `optimize()`. The result of `optimize()` should be saved to an object.

To illustrate the use of `optimize()`, the code below first creates 1000 random observations from a normal distribution with a mean of 0 and a standard deviation of 1, then finds the mean of these 1000 observations (this should be the value of theta that minimizes the sum of squared deviations), and then uses `optimize()` to minimize `theta` from all `theta`s between -2 and 2 given the observations in `dat`. The value of `theta` that minimizes the sum-of-squared deviations is returned with the `\$minimum` object of the saved `optimize` object (this will be the `\$maximum` object if `maximum=TRUE` was used) and the corresponding sum-of-squared deviations is given in the `\$objective` object of the saved `optimize` object.

```
> dat <- rnorm(1000)
> mean(dat)
```

```
[1] -0.0004261087
```

```
> opt1 <- optimize(SSE,c(-2,2),obs=dat)
> opt1$minimum
```

```
[1] -0.0004261087
```

```
> opt1$objective
```

```
[1] 1035.996
```

0.3.2.3 `optim()` for Multi-Parameter Models The `optim()` function requires a vector of starting values for each parameter as the first argument and the function to be optimized as the second argument. In addition, an optional `method=` argument can be used to control the specific algorithm used to do the optimization. The default method used is the "Nelder-Mead" method. Names of other methods can be found in the corresponding help file (see `?optim`). Various characteristics of the alorithm used can be controlled with a list submitted to the `control=` argument. Descriptions of these controls can be found on the help page for `optim()`; however, three of general interest are listed below.

- `fnscale=`: a constant multiple for the result from the optimization function. If `fnscale=-1` is used then `optim()` performs a *maximization* rather than the default *minimization*.
- `maxit=`: an integer that controls the maximum number of iterations that the algorithm will perform.
- `parscale=`: a vector of constants that each parameter should be divided by. These constants can be very useful when attempting to solve for parameters that are on very different scales. An example is shown below.

For example, the code below will maximize (note use of `fnscale=1`) the `normLH()` function with starting values for `mu` and `sig` of 0 and 1, respectively, given the data in `dat`. The values of `mu` and `sig` that maximize the log likelihood is returned with the `\$par` object of the saved `optim` object and the corresponding maximum log likelihood is given in the `\$value` object of the saved `optim` object.

```
> opt2 <- optim(c(0,1),normLH,control=list(fnscale=-1),obs=dat)
> opt2$par
```

```
[1] -0.0003969545  1.0177934869
```

```
> opt2$value
```

```
[1] -1436.62
```

0.3.2.4 Constraining Parameter Estimates In some instances, the optimization routine may find values for the parameters that are outside of the range of reasonable values for that parameter (e.g., a negative catchability coefficient). In these instances, it may be necessary to constrain the parameter estimates within a particular range. The `method="L-BFGS-B"` optimization method allows for so-called "box constraints" for each parameter. If this method is used in `optim()`, then the user must also supply a vector of minimum values for each parameter in the `lower=` argument and a vector of maximum values for each parameter in the `upper=` argument. For example, if one wants to constain the possible values of `sig` to be between 0 and 2 but not constrain the possible values of `mu` then one would use the following code for `optim()`,

```
> opt3 <- optim(c(0,1),normLH,method="L-BFGS-B",lower=c(NA,0),upper=c(NA,2),
                control=list(fnscale=-1),obs=dat)
> opt3$par
```

```
[1] -0.000426349  1.017840086
```

```
> opt3$value
```

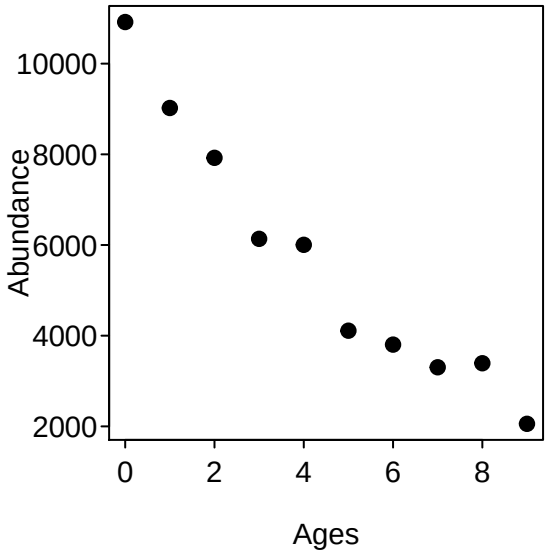
```
[1] -1436.62
```

Note that constraining the values in this case is not really needed and the results are not substantively different from what was done above. The minor differences that did occur in the parameter estimates is likely due to the fact that a different optimization algorithm was used. For an example where constraining the parameters was crucial see Box 8.5 in the Chapter 8 vignette.

0.3.2.5 Rescaling Parameter Estimates In some situations, the scales of the parameters may be radically different. Many algorithms, including the "Nelder-Mead" algorithm, may have difficulty finding the optimal solution in these situations. The `optim()` function provides the `parscale=` argument for re-scaling the parameters during the optimization iterations such that a "better" solution can be achieved. Before illustrating the use of the arguments let's consider a different situation then that described above.

Suppose that we have random data that consists of the abundance of animals at 10 ages given an initial number of 10000 animals (e.g., N_0), an instantaneous mortality rate of 0.2 (e.g., Z), and an (unrealistic) additive normal error with a mean of 800 animals and standard deviation of 300 animals. These random data is constructed and visualized as below.

```
> ages <- 0:9
> Nt <- round(10000*exp(-0.2*ages)+rnorm(length(ages),800,300),0)
> plot(Nt~ages,pch=19,xlab="Ages",ylab="Abundance")
```



A function that computes the sum-of-squared deviations from an exponential decay model given parameters N_0 and Z is created below.

```
> edSSE <- function(pars,t,y) {  # t=age, y=abundance
    N0 <- pars[1]                # isolate N0 param from first position
    Z <- pars[2]                 # isolate Z param from second position
    sum((y-N0*exp(-Z*t))^2)      # compute and return SSE
  }
```

In this situation, the parameter starting values (shown below in `par.start` differ by approximately five orders of magnitude. In this situation, "better" parameter estimates and a faster convergence may be achieved by re-scaling the parameter values. As a general rule-of-thumb, the parameter scale values can be set at 1 times the order of magnitude of the parameter estimate (see `par.scale` below). These parameter scale values are used in `optim()` by including the `parscale=` argument in the `control=` argument list.

```
> par.start <- c(15000,0.3)
> par.scale <- c(1e4,1e-1)
> opt4 <- optim(par.start,edSSE,control=list(parscale=par.scale),t=ages,y=Nt)
> opt4$par
```

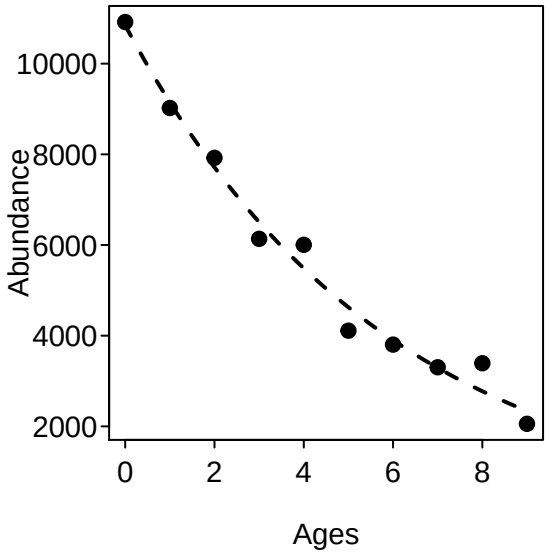
```
[1] 1.084634e+04 1.704109e-01
```

```
> opt4$value
```

```
[1] 1207409
```

A graphic of the optimal fit is constructed and shown below.

```
> plot(Nt~ages,pch=19,xlab="Ages",ylab="Abundance")
> curve(opt4$par[1]*exp(-opt4$par[2]*x),c(0,9),lwd=2,lty=2,add=TRUE)
```



The rescaling was not particularly important in this very simple example. For an example where rescaling was crucial see Box 8.7 in the Chapter 8 vignette.

```
Reproducibility Information
  Compiled Date: Sun Apr 26 2015
  Compiled Time: 11:23:25 AM
  Code Execution Time: 0.22 s

  R Version: R version 3.2.0 (2015-04-16)
  System: Windows, i386-w64-mingw32/i386 (32-bit)
  Base Packages: base, datasets, graphics, grDevices, methods, stats, utils
  Required Packages: FSA and its dependencies (car, dplyr, gdata, gplots,
    Hmisc, knitr, lmtest, multcomp, plotrix, relax, sciplot)
  Other Packages: FSA_0.6.12, knitr_1.9, rmarkdown_0.5.1
  Loaded-Only Packages: acepack_1.3-3.3, assertthat_0.1, bitops_1.0-6,
    car_2.0-25, caTools_1.17.1, cluster_2.0.1, codetools_0.2-11,
    colorspace_1.2-6, DBI_0.3.1, digest_0.6.8, dplyr_0.4.1, evaluate_0.6,
    foreign_0.8-63, formatR_1.1, Formula_1.2-1, gdata_2.13.3,
    ggplot2_1.0.1, gplots_2.16.0, grid_3.2.0, gtable_0.1.2, gtools_3.4.2,
    highr_0.4.1, Hmisc_3.15-0, htmltools_0.2.6, KernSmooth_2.23-14,
    lattice_0.20-31, latticeExtra_0.6-26, lme4_1.1-7, lmtest_0.9-33,
    magrittr_1.5, MASS_7.3-40, Matrix_1.2-0, mgcv_1.8-6, minqa_1.2.4,
    multcomp_1.4-0, munsell_0.4.2, mvtnorm_1.0-2, nlme_3.1-120,
    nloptr_1.0.4, nnet_7.3-9, parallel_3.2.0, pbkrtest_0.4-2,
    plotrix_3.5-11, plyr_1.8.1, proto_0.3-10, quantreg_5.11,
    RColorBrewer_1.1-2, Rcpp_0.11.5, relax_1.3.15, reshape2_1.4.1,
    rpart_4.1-9, sandwich_2.3-3, scales_0.2.4, sciplot_1.1-0, SparseM_1.6,
    splines_3.2.0, stringr_0.6.2, survival_2.38-1, TH.data_1.0-6,
    tools_3.2.0, yaml_2.1.13, zoo_1.7-12
```