MathSoft

S-PLUS 4
Guide to Statistics

March 1998

Data Analysis Products Division
MathSoft, Inc.
Seattle, Washington

Proprietary Notice

The correct bibliographical reference for this document is as follows:

S-PLUS 4 Guide to Statistics, Data Analysis Products Division, MathSoft, Seattle.

Printed in the United States.

Copyright Notice

Acknowledgments

S-PLUS would not exist without the pioneering research of the Bell Labs S team at AT&T (now Lucent Technologies): Richard A. Becker, John M. Chambers, Allan R. Wilks, William S. Cleveland, and colleagues.

This release of S-PLUS includes specific work from a number of scientists:

The cluster library was written by Mia Hubert, Peter Rousseeuw and Anja Struyf (University of Antwerp).

Updates to functions provided to this and earlier releases of S-PLUS were provided by Brian Ripley (Oxford University) and Terry Therneau (Mayo Clinic, Rochester).

GUIDE TO STATISTICS CONTENTS OVERVIEW

Introduction

Estimation and Inference

Regression and Smoothing

Analysis of Variance

Multivariate Techniques

.

CONTENTS

PREFACE

Introduction

Welcome to the *S-PLUS Guide to Statistics.*

This book is designed as a reference tool for S-PLUS users wanting to use the powerful statistical techniques in S-PLUS. The *Guide to Statistics* covers a wide range of statistical and mathematical modeling; no one user is likely to tap all of these resources since advanced topics such as survival analysis and time series are complete fields of study in themselves.

All examples in this guide are run using input through the Commands window—the traditional method of accessing the power of S-PLUS. Many of the functions can also be run through the Statistics menu and dialogs available in the graphical user interface. We hope you will find this book a valuable aid for exploring both the theory and practice of statistical modeling.

On-line version

This Guide is also available on-line, through the On-line Manuals entry of the main Help menu. It can be viewed using Adobe Acrobat Reader, which is included with S-PLUS.

The on-line version is identical in content to the printed one, but with some particular advantages. First, you can cut-and-paste example S-PLUS code directly into the Commands window, and can run these examples without having to type them. Be careful not to cut-and-paste the ">" prompt character and notice that distinct colors differentiate between command language input and output.

Secondly, the on-line text can be searched for any character string. If you wish information on a certain function, for example, you can easily browse through all occurrences of it in the guide.

Also, contents and index entries in the on-line version are hot-links; click on them to go to the appropriate page.

Evolution of S-PLUS

S-PLUS has evolved considerably from its beginnings as a research tool, and the contents of this guide has grown steadily, and will continue to grow, as the language is improved and expanded. This may mean that some examples in the text do not match your output from S-PLUS in every formatting detail. However, the underlying theory and computations, are as described here.

In addition to the huge range of functionality covered in this guide, there are additional modules, libraries, and user-written functions available from a number of sources. Refer to the *S-PLUS* User's Guide for more details.

Companion Guides

The *Guide to Statistics* is a companion volume to the *S-PLUS* User's Guide and the *S-PLUS* Programmer's Guide. All three are available both in printed form and on-line through the help system.

INTRODUCTION TO STATISTICAL ANALYSIS IN S-PLUS

1

The power of S-PLUS lies in its wide variety of modeling techniques available for statistical analysis.

INTRODUCTION TO STATISTICAL ANALYSIS IN S-PLUS

All statistical analysis has, at its heart, a *model* which attempts to describe the structure or relationships in some objects or phenomena on which measurements (the data) are taken. Estimation, hypothesis testing, and inference, in general, are based on the data at hand and a conjectured model which you may define implicitly or explicitly. You specify many types of models in S-PLUS using *formulas*, which express the conjectured relationships between observed variables in a natural way. The power of S-PLUS as a statistical modeling language lies in its convenient and useful way of organizing data, its wide variety of classical and modern modeling techniques, and its way of specifying models.

The goal of this chapter is to give you a feel for data analysis in S-PLUS: examining the data, selecting a model, and displaying and summarizing the fitted model.

1.1 DEVELOPING STATISTICAL MODELS

The process of developing a statistical model varies depending on whether you follow a classical, hypothesis-driven approach (confirmatory data analysis) or a more modern, data-driven approach (exploratory data analysis). In many data analysis projects, both approaches are frequently used. For example, in classical regression analysis, you usually examine residuals using exploratory data analytic methods for verifying whether underlying assumptions of the model hold. The goal of either approach is a model which imitates, as closely as possible, in as simple a way as possible, the properties of the objects or phenomena being modeled. Creating a model usually involves the following steps:

1. Determine the variables to observe. In a study involving a classical modeling approach, these variables correspond to the hypothesis being tested. For data-driven modeling, these variables are the link to the phenomena being modeled.
2. Collect and record the data observations.
3. Study graphics and summaries of the collected data to discover and remove mistakes and to reveal low-dimensional relationships between variables.
4. Choose a model describing the important relationships seen or hypothesized in the data.

5. Fit the model using the appropriate modeling technique.
6. Examine the fit using model summaries and diagnostic plots.
7. Repeat steps 4–6 until you are satisfied with the model.

There are a wide range of possible modeling techniques to choose from when developing statistical models in S-PLUS. Among these are linear models (`lm`), analysis of variance models (`aov`), generalized linear models (`glm`), generalized additive models (`gam`), local regression models (`loess`), and tree-based models (`tree`).

1.2 DATA USED FOR MODELS

This section provides descriptions of the most common types of data objects used when developing models in S-PLUS. There are also brief descriptions and examples of common S-PLUS functions used for developing and displaying models.

Data Frame Objects

Statistical models allow inferences to be made about objects by modeling associated observational or experimental data, organized by *variables*. A *data frame* is an object that represents a sequence of observations on some chosen set of variables. Data frames are like matrices, with variables as columns and observations as rows. They allow computations where variables can act as *separate objects* and can be referenced simply by naming them. This makes data frames very useful in modeling.

Variables in data frames are generally of three forms:

- Numeric vectors.
- Factors and ordered factors.
- Numeric matrices.

Continuous and Discrete Data

The type of data you have when developing a model is important for deciding which modeling technique best suits your data. *Continuous* data represent quantitative data having a continuous range of values. *Categorical* data, by contrast, represent *qualitative* data, and are discrete, meaning they can assume only certain fixed numeric or nonnumeric values.

In S-PLUS, you represent categorical data with *factors*, which keep track of the *levels* or different values contained in the data and the level each data point corresponds to. For example, you might have a factor `gender` in which every element assumed one of the two values `"male"` and `"female"`. You represent continuous data with numeric objects. Numeric objects are vectors, matrices, or arrays of numbers. Numbers can take the form of decimal numbers (such as `11`, `-2.32`, or `14.955`) and exponential numbers

expressed in scientific notation (such as `.002` expressed as `2e-3`).

A statistical model expresses a *response* variable as some function of a set of one or more *predictor* variables. The type of model you select depends on whether the response and predictor variables are continuous (numeric) or categorical (factor). For example, the classical regression problem has a continuous response and continuous predictors, but the classical ANOVA problem has a continuous response and categorical predictors.

Summaries and Plots for Examining Data

Before you fit a model, you should examine the data. Plots provide important information on mistakes, outliers, distributions and relationships between variables. Numerical summaries provide a statistical synopsis of the data in a tabular format.

Among the most common functions to use for generating plots and summaries are the following:

- `summary`: provides a synopsis of an object. The following example displays a summary of the `kyphosis` data frame:

```
> summary(kyphosis)
   Kyphosis         Age              Number
 absent  :64  Min.   :  1.00  Min.   : 2.000
 present :17  1st Qu.: 26.00  1st Qu.: 3.000
              Median : 87.00  Median : 4.000
              Mean   : 83.65  Mean   : 4.049
              3rd Qu.:130.00  3rd Qu.: 5.000
              Max.   :206.00  Max.   :10.000

      Start
 Min.    : 1.00
 1st Qu. : 9.00
 Median  :13.00
 Mean    :11.49
 3rd Qu. :16.00
 Max.    :18.00
```

- `plot`: a generic plotting function, `plot` produces different kinds of plots depending on the data passed to it. In its most common use, it produces a scatter plot of two numeric objects.
- `hist`: creates histograms.
- `qqnorm`: creates quantile-quantile plots.
- `pairs`: creates, for multivariate data, a matrix of scatter plots showing each variable plotted against each of the other variables. To

create the pairwise scatter plots for the data in the matrix `longley.x`, use `pairs` as follows:

```
> pairs(longley.x)
```

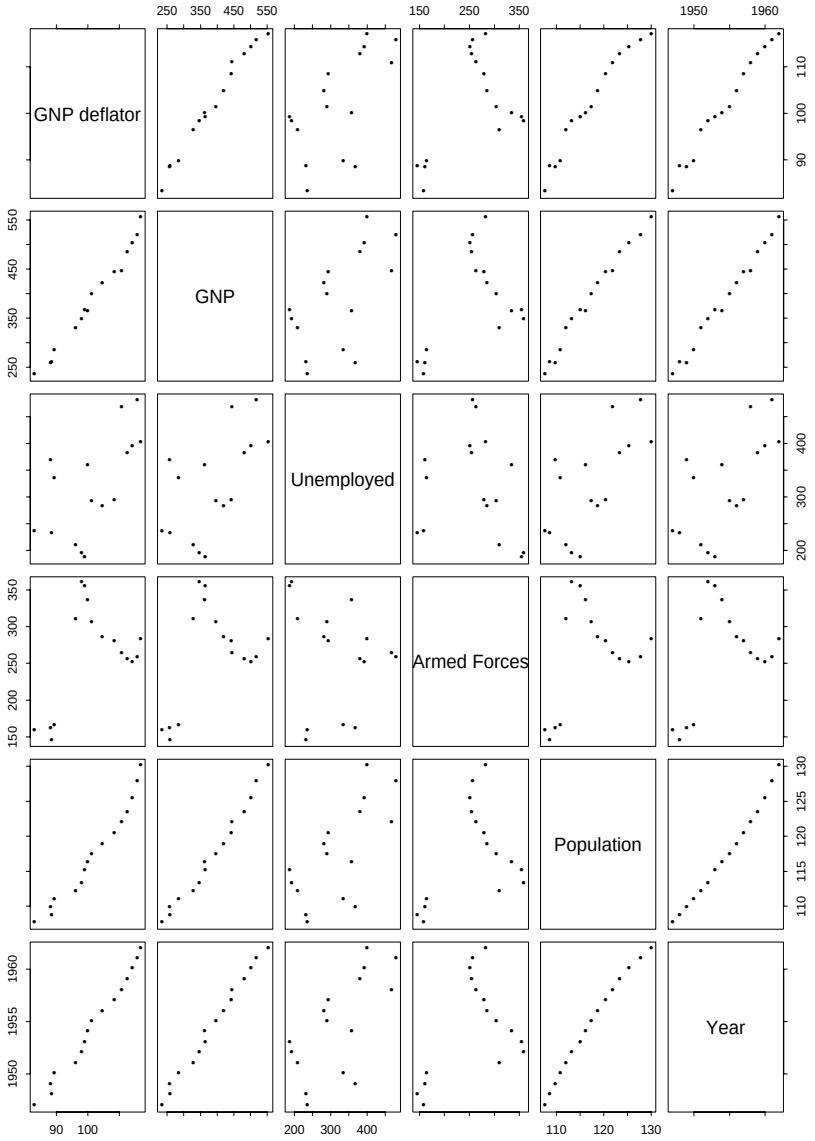

Figure 1.1: *Pairwise scatter plots for* `longley.x`.

The resulting plot appears as in figure 1.1.

- `coplot`: provides a graphical look at cross-sectional relationships, which enable you to assess potential interaction effects. The following example shows the effect of the interaction between `C` and `E` on values of `NOx`. The resulting plots appear as in figure 1.2.

```
> attach(ethanol)
> E.intervals <- co.intervals(E, 9, 0.25)
> coplot(NOx ~ C | E, given.values = E.intervals,
+ data = ethanol, panel = function(x, y) panel.smooth(x,
+ y, span = 1, degree = 1))
```

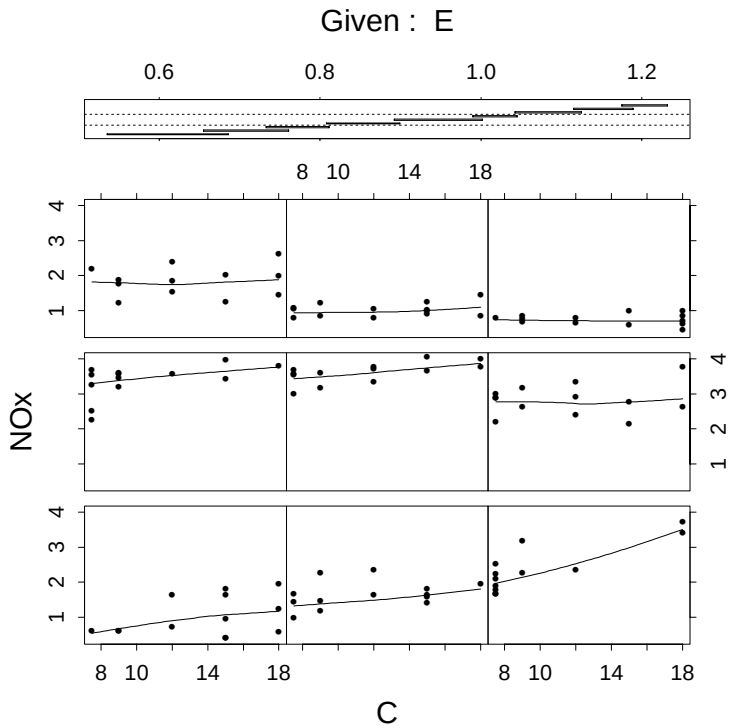

Figure 1.2: *Coplot of response and predictors.*

1.3 STATISTICAL MODELS IN S-PLUS

The development of statistical models is, in many ways, *data dependent*. The choice of the modeling technique you use depends upon the type and structure of your data and what you want the model to test or explain. A model may predict new responses, show general trends, or uncover underlying phenomena. This section gives general selection criteria to help you develop a statistical model.

The fitting procedure for each model is based on a unified modeling paradigm in which:

- A data frame contains the data for the model.
- A *formula* object specifies the relationship between the response and predictor variables.
- The formula and data frame are passed to the fitting function.
- The fitting function returns a fit object.

There is a relatively small number of functions to help you fit and analyze statistical models in S-PLUS.

- Fitting models:
  - `lm`: linear (regression) models.
  - `aov` and `varcomp`: analysis of variance models.
  - `glm`: generalized linear models.
  - `gam`: generalized additive models.
  - `loess`: local regression models.
  - `tree`: tree models.
- Extracting information from a fitted object:
  - `fitted`: returns fitted values.
  - `coefficients` or `coef`: returns the coefficients (if present).
  - `residuals` or `resid`: returns the residuals.
  - `summary`: provides a synopsis of the fit.
  - `anova`: for a single fit object, produces a table with rows corresponding to each of the terms in the object, plus a row for residuals. If two or more fit objects are used as arguments, `anova` returns a table showing the tests for differences between the models, sequentially, from first to last.
- Plotting the fitted object:
  - `plot`: plot a fitted object.
  - `qqnorm`: produces a normal probability plot, frequently used in analysis of residuals.
  - `coplot`: provides a graphical look at cross-sectional relationships

for examining interaction effects.

- For minor modifications in a model, use the `update` function (adding and deleting variables, transforming the response, etc.).
- To compute the predicted response from the model, use the `predict` function.

The Unity of Models in Data Analysis

Because there is usually more than one way to model your data, you should learn which type(s) of model are best suited to various types of response and predictor data. When deciding on a modeling technique, it helps to ask: "What do I want the *data* to explain? What hypothesis do I want to test? What am I trying to show?"

Some methods should or should not be used depending on whether the response and predictors are continuous, factors, or a combination of both. Table 1.1 organizes the methods by the type of data they can handle.

Table 1.1: *Criteria for developing models.*

Model	Response	Predictors
`lm`	Continuous	Both
`aov`	Continuous	Factors
`glm`	Both	Both
`gam`	Both	Both
`loess`	Continuous	Both
`tree`	Both	Both

Linear regression models a continuous response variable, y, as a linear combination of predictor variables x_j, for j=1,...,p. For a single predictor, the data fit by a linear model scatter about a straight line or curve. A linear regression model has the mathematical form

$$y_i = \beta_0 + \sum_{j=1}^{p} \beta_j x_{ij} + \varepsilon_i$$

where ε_i, referred to, generally, as the error, is the difference between the ith observation and the model. On average, for given values of the predictors, you predict the response best with the equation

$$y = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

Analysis of variance models are also linear models, but all predictors are categorical, which contrasts with the typically continuous predictors of regression. For designed experiments, use analysis of variance to estimate and test for effects due to the factor predictors. For example, consider the `catalyst` data frame, which contains the data below:

```
  Temp Conc Cat Yield
1  160   20   A    60
2  180   20   A    72
3  160   40   A    54
4  180   40   A    68
5  160   20   B    52
6  180   20   B    83
7  160   40   B    45
8  180   40   B    80
```

Each of the predictor terms, `Temp`, `Conc`, and `Cat`, is a factor with two possible levels, and the response term, `Yield`, contains numeric data. Use analysis of variance to estimate and test for the effect of the predictors on the response.

Linear models produce estimates with good statistical properties when the relationships are, in fact, linear, and the errors are normally distributed. In some cases, when the distribution of the response is skewed, you can transform the response, using, for example, square root, logarithm, or reciprocal transformations, and produce a better fit. In other cases, you may need to include polynomial terms of the predictors in the model. However, if linearity or normality does not hold, or if the variance of the observations is not constant, and transformations of the response and predictors do not help, you should explore other techniques such as generalized linear models, generalized additive models, or classification and regression trees.

Generalized linear models generalize linear models by assuming a *transformation* of the expected (or average) response is a linear function of the predictors, and the variance of the response is a function of the mean response:

$$\eta(E(y)) = \beta_0 + \sum_{j=1}^{p} \beta_j x_j$$

$$\mathrm{VAR}(y) = \phi V(\mu)$$

Generalized linear models, fitted using the `glm` function, allow you to model data with distributions including normal, binomial, Poisson, gamma, and inverse normal, but still require a linear relationship in the parameters.

When the linear fit provided by `glm` does not produce a good fit, an alternative is the generalized additive model, fit with the `gam` function. In contrast to `glm`, `gam` allows you to fit nonlinear data-dependent functions of the predictors. The mathematical form of a generalized additive model is

$$\eta(E(y)) = \sum_{j=1}^{p} f_j(x_j)$$

where the f_j term represent functions to be estimated from the data. The form of the model assumes a low-dimensional additive structure. That is, the pieces represented by functions, f_i, of each predictor added together predict the response without interaction.

In the presence of interactions, if the response is continuous and the errors about the fit are normally distributed, local regression (or *loess*) models, allow you to fit a multivariate function which includes interaction relationships. The form of the model is

$$y_i = g(x_{i1}, x_{i2}, ..., x_{ip}) + \varepsilon_i$$

where g represents the regression surface.

Tree-based models have gained in popularity because of their flexibility in fitting all types of data. Tree models are generally used for exploratory analysis. They allow you to study the structure of data, creating nodes or clusters of data with similar characteristics. The variance of the data within each node is relatively small, since the characteristics of the contained data is similar. The following example displays a tree-based model using the data frame `car.test.frame`:

```
> car.tree <- tree(Mileage ~ Weight, car.test.frame)
> plot(car.tree, type = "u")
> text(car.tree)
> title("Tree-based Model")
```

The resulting plot appears as in figure 1.3.

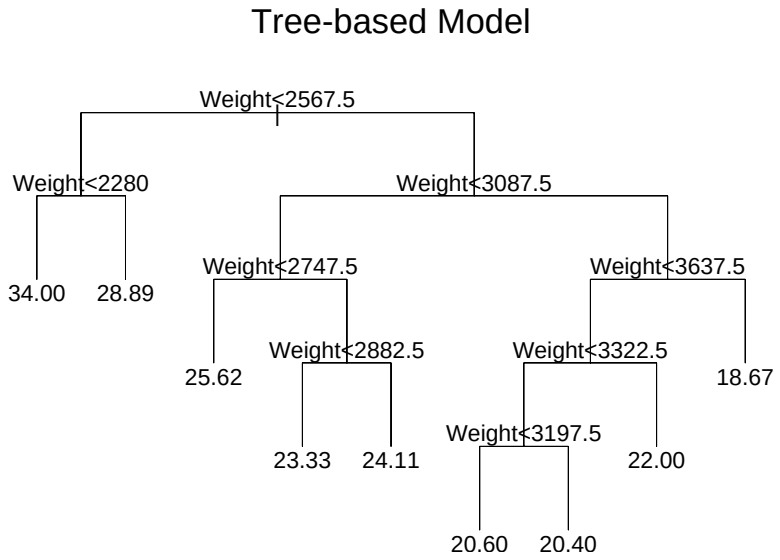

Figure 1.3: *A tree-based model for* `Mileage` *versus* `Weight`.

1.4 EXAMPLE OF DATA ANALYSIS

The example that follows describes only one way of analyzing data through the use of statistical modeling. There is no perfect cookbook approach to building models, as different techniques do different things, and not all of them use the same arguments when doing the actual fitting.

The Iterative Process of Model Building

As was discussed at the beginning of this chapter, there are some general steps you can take when building a model:

1. Determine the variables to observe. In a study involving a classical modeling approach, these variables correspond directly to the hypothesis being tested. For data-driven modeling, these variables are the link to the phenomena being modeled.
2. Collect and record the data observations.
3. Study graphics and summaries of the collected data to discover and remove mistakes and to reveal low-dimensional relationships between variables.
4. Choose a model describing the important relationships seen or hypothesized in the data.
5. Fit the model using the appropriate modeling technique.
6. Examine the fit through model summaries and diagnostic plots.

7. Repeat steps 4–6 until you are satisfied with the model.

At any point in the modeling process, you may find that your choice of a model does not appropriately fit the data. In some cases, diagnostic plots may give you clues to improve the fit. Sometimes you may need to try transformed variables or entirely different variables. You may need to try a different modeling technique that will, for example, allow you to fit nonlinear relationships, interactions, or different error structures. At times, all you need to do is remove outlying, influential data, or fit the model robustly. A point to remember is that there is no one answer on how to build good statistical models. By iteratively fitting, plotting, testing, changing something and then refitting, you will arrive at the best fitting model for your data.

Exploring the Data

The following analysis uses the built-in data set `auto.stats`, which contains a variety of data for car models between the years 1970–1982, including price, miles per gallon, weight, and more. Suppose we want to model the effect that `Weight` has on the gas mileage of a car. The object, `auto.stats`, is not a data frame, so we start by coercing it into a data frame object:

```
> auto.dat <- data.frame(auto.stats)
```

Attach the data frame to treat each variable as a separate object:

```
> attach(auto.dat)
```

Look at the distribution of the data by plotting a histogram of the two variables, `Weight` and `Miles.per.gallon`. First, split the graphics screen into two portions to display both graphs:

```
> par(mfrow = c(1, 2))
```

Plot the histograms:

```
> hist(Weight)
> hist(Miles.per.gallon)
```

The resulting histograms appear as in figure 1.4.

Subsetting (or subscripting) gives you the ability to look at only a portion of the data. For example, type the following to look at only those cars with mileage greater than 34 miles per gallon:

```
> auto.dat[Miles.per.gallon > 34,]
                Price Miles.per.gallon Repair (1978)
    Datsun 210  4589               35             5
        Subaru  3798               35             5
Volk Rabbit(d)  5397               41             5
```

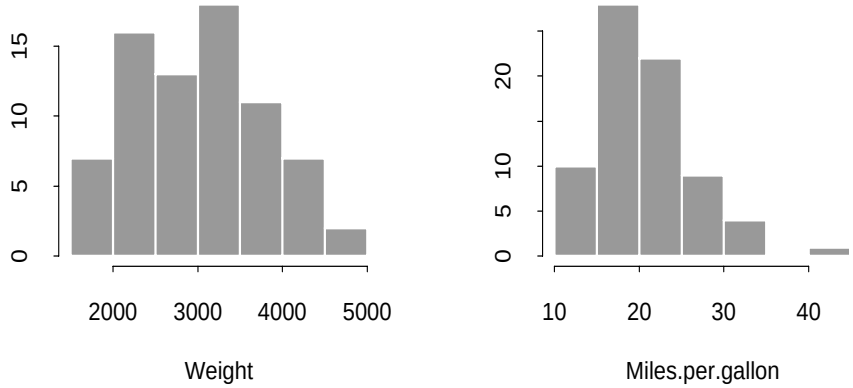

Figure 1.4: *Histograms of* `Weight` *and* `Miles.per.gallon`.

```
                Repair (1977) Headroom Rear.Seat Trunk Weight
    Datsun 210             5      2.0      23.5     8   2020
        Subaru             4      2.5      25.5    11   2050
Volk Rabbit(d)             4      3.0      25.5    15   2040
                Length Turning.Circle Displacement Gear.Ratio
    Datsun 210     165             32           85       3.70
        Subaru     164             36           97       3.81
Volk Rabbit(d)     155             35           90       3.78
```

Suppose you want to predict the gas mileage of a particular auto based upon its weight. Create a scatter plot of `Weight` versus `Miles.per.gallon` to examine the relationship between the variables. First, reset the graphics window to display only one graph:

```
> par(mfrow = c(1,1))
```

Plot `Weight` versus `Miles.per.gallon`. The plot appears as in figure 1.5:

```
> plot(Weight, Miles.per.gallon)
```

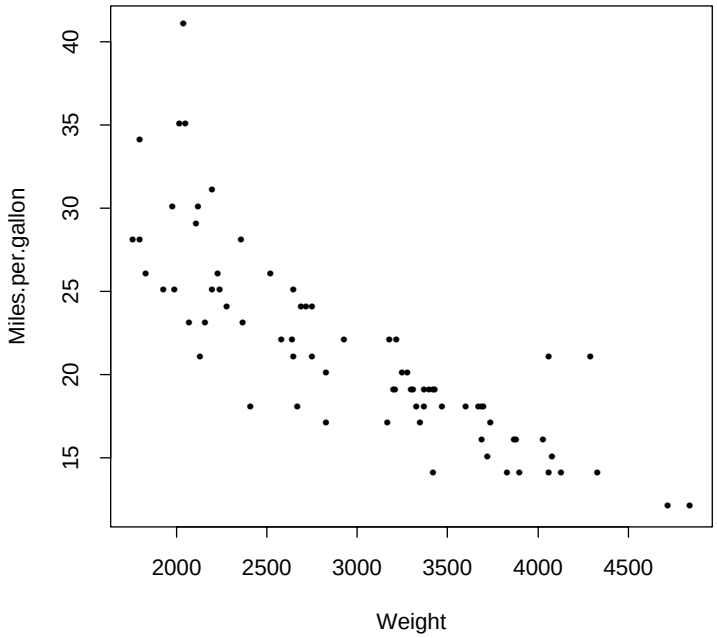

Figure 1.5: *Scatter plot:* `Weight` *versus* `Miles.per.gallon`.

The resulting figure displays a curved scattering of the data. This might suggest a nonlinear relationship. Create a plot from a different perspective, giving gallons per mile (`1/Miles.per.gallon`) as the vertical axis:

```
> plot(Weight, 1/Miles.per.gallon)
```

The resulting scatter plot appears as in figure 1.6.

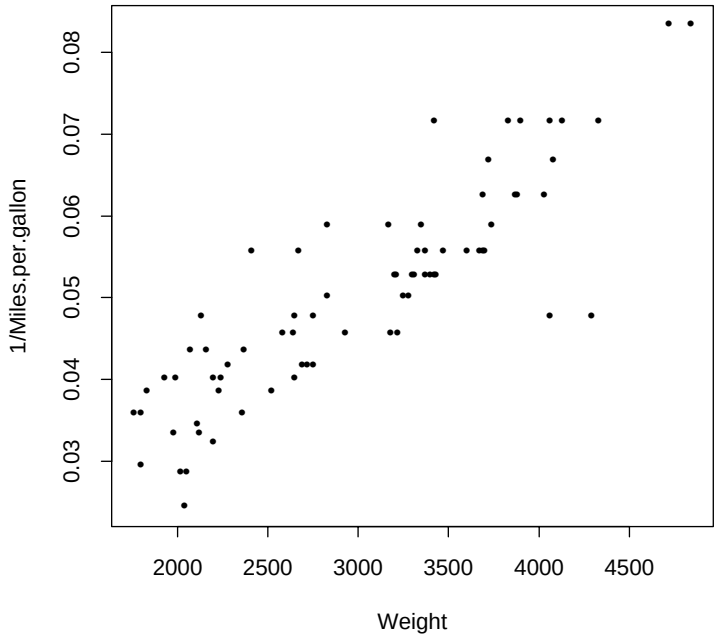

Figure 1.6: *Scatter plot of* `Weight` *versus* `1/Miles.per.gallon`.

Fitting the Model

Gallons per mile is more linear with respect to weight, suggesting that you can fit a linear model to `Weight` and `1/Miles.per.gallon`. The formula `1/Miles.per.gallon ~ Weight` describes this model. Fit the model by using the `lm` function, and name the fitted object `fit1`:

```
> fit1 <- lm(1/Miles.per.gallon ~ Weight)
```

As with any S-PLUS object, when you type the name, `fit1`, S-PLUS prints the object, in this case, using the specific print method for `"lm"` objects:

```
> fit1
Call:
lm(formula = 1/Miles.per.gallon ~ Weight)

Coefficients:
 (Intercept)       Weight
 0.007447302 1.419734e-05
Degrees of freedom: 74 total; 72 residual
Residual standard error: 0.006363808
```

Plot the regression line to see how well it fits the data. The resulting line appears as in figure 1.7.

```
> abline(fit1)
```

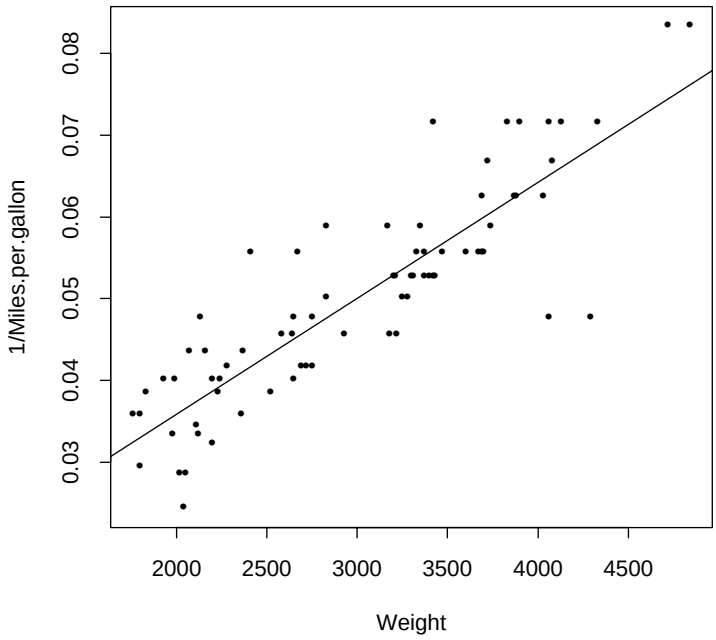

Figure 1.7: *Regression line of* `fit1`.

Judging from figure 1.7, the regression line does not fit well in the upper range of `Weight`. Plot the residuals versus the fitted values to see more clearly where the model does not fit well.

```
> plot(fitted(fit1), residuals(fit1))
```

The plot appears as in figure 1.8.

Note that with the exception of two outliers in the lower right corner, the residuals become more positive as the fitted value increases. You can identify the outliers by typing the following command, then interactively clicking on the outliers with your mouse:

```
> outliers <- identify(fitted(fit1), residuals(fit1),
+ labels = names(Weight))
```

The `identify` function allows you to interactively select the points on the plot. The `labels` argument and `names` function label the points with their names in the object. For more information on the `identify` function, see chapter Traditional Graphics in the S-PLUS Programmer's Guide The plot

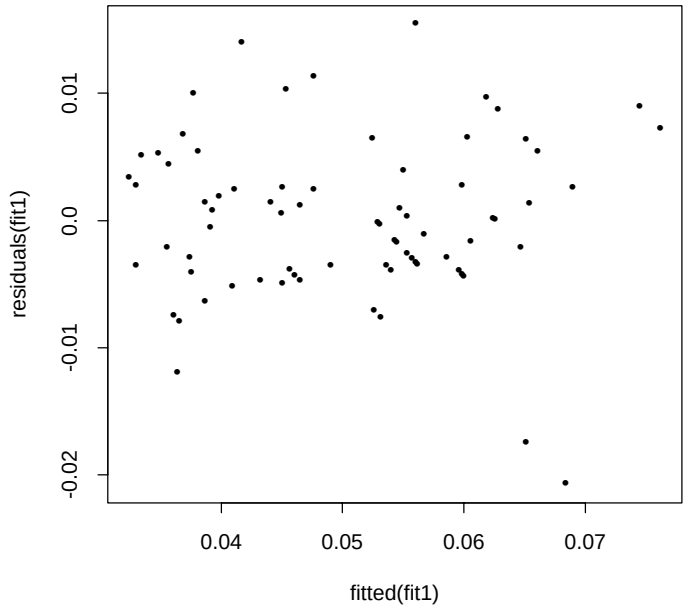

Figure 1.8: *Plot of residuals for* `fit1`.

appears as in figure 1.9.

These outliers correspond to cars with *better* gas mileage than other cars in the study with similar weights. You can remove the outliers using the `subset` argument to `lm`.

```
> fit2 <- lm(1/Miles.per.gallon ~ Weight,
+ subset = -outliers)
```

Plot `Weight` versus `1/Miles.per.gallon`, and also two regression lines, one for the `fit1` object and one for the `fit2` object. Use the `lty=` argument to differentiate between the regression lines:

```
> plot(Weight, 1/Miles.per.gallon)
> abline(fit1, lty=2)
> abline(fit2)
```

The two lines appear with the data in figure 1.10.

A plot of the residuals versus the fitted values shows a better fit. The plot appears as in figure 1.11:

```
> plot(fitted(fit2), residuals(fit2))
```

To see a synopsis of the fit contained in `fit2`, use `summary` as follows:

```
> summary(fit2)
```

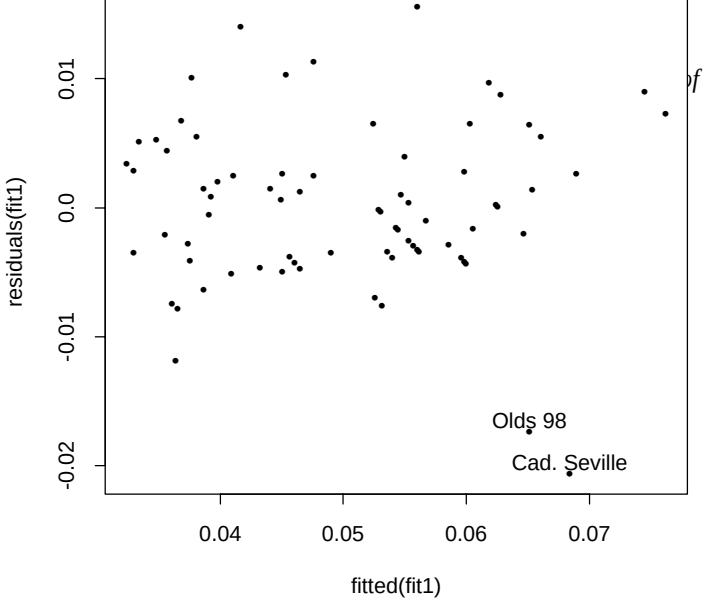

Figure 1.9: *Plot with labeled outliers.*

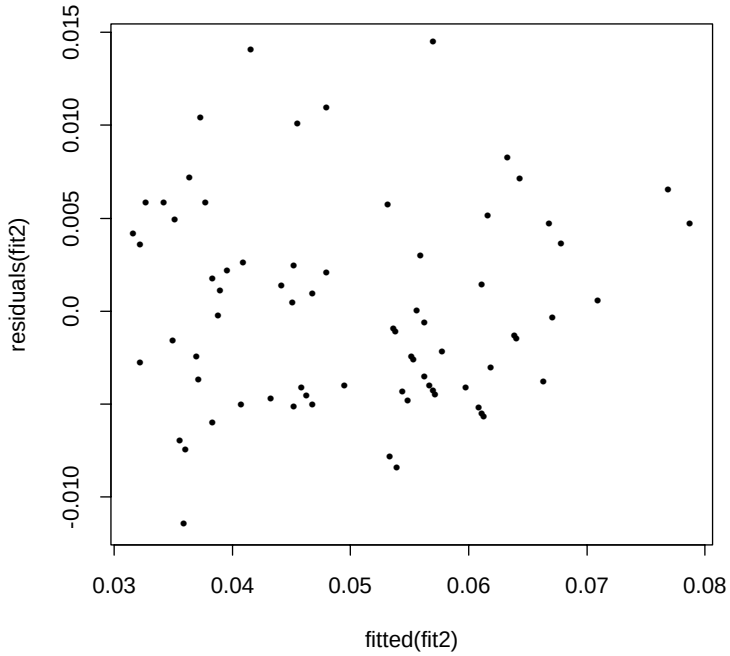

Figure 1.11: *Plot of residuals for* `fit2`.

```
Call: lm(formula = 1/Miles.per.gallon ~ Weight,
subset = - outliers)
Residuals:
      Min        1Q     Median       3Q     Max
 -0.01152 -0.004257 -0.0008586 0.003686 0.01441

Coefficients:
             Value Std. Error t value Pr(>|t|)
(Intercept)  0.0047     0.0026  1.8103   0.0745
     Weight  0.0000     0.0000 18.0625   0.0000

Residual standard error: 0.00549 on 70 degrees of freedom
Multiple R-squared: 0.8233
F-statistic: 326.3 on 1 and 70 degrees of freedom, the p-
value is 0
Correlation of Coefficients:
       (Intercept)
Weight -0.9686
```

The summary displays information on the spread of the residuals, coefficients, standard errors, and tests of significance for each of the variables

in the model (it includes an intercept by default), and overall regression statistics for the fit. As expected, `Weight` is a very significant predictor of `1/Miles.per.gallon`. The amount of the variability of `1/Miles.per.gallon` explained by `Weight` is about 82%, and the residual standard error is .0055, down about 14% from that of `fit1`.

To see the individual coefficients for `fit2`, use `coef` as follows:

```
> coef(fit2)
 (Intercept)       Weight
 0.004713079 1.529348e-05
```

Fitting an Alternative Model

Now consider an alternative approach. Recall the plot in figure 1.5 showed curvature in the scatter plot of `Weight` versus `Miles.per.gallon`, indicating that a straight line fit is an inappropriate model. You can fit a nonparametric nonlinear model to the data using

`gam` using a cubic spline smoother to model the curvature in the data:

```
> fit3 <- gam(Miles.per.gallon ~ s(Weight))
> fit3
Call:
gam(formula = Miles.per.gallon ~ s(Weight))

Degrees of Freedom: 74 total; 69.00244 Residual
Residual Deviance: 704.7922
```

The resulting plot of `fit3` appears as in figure 1.12:

```
> plot(fit3, residuals = T, scale =
+ diff(range(Miles.per.gallon)))
```

The cubic spline smoother in the plot appears to give a good fit to the data. You can check the fit with diagnostic plots of the residuals as we did for the linear models. You should also compare the `gam` model with a linear model using `aov` to produce a statistical test.

Use the `predict` function to make predictions from models. One of the arguments to `predict`, `newdata`, specifies a data frame containing the values at which the predictions are required. If `newdata` is not supplied, the `predict` function will make predictions at the data originally supplied to fit the `gam` model, as in the following example:

```
> predict.fit3 <- predict(fit3)
```

Create a new object `predict.high` and print it to display cars with predicted miles per gallon greater than 30:

```
> predict.high <- predict.fit3[predict.fit3 > 30]
> predict.high
```

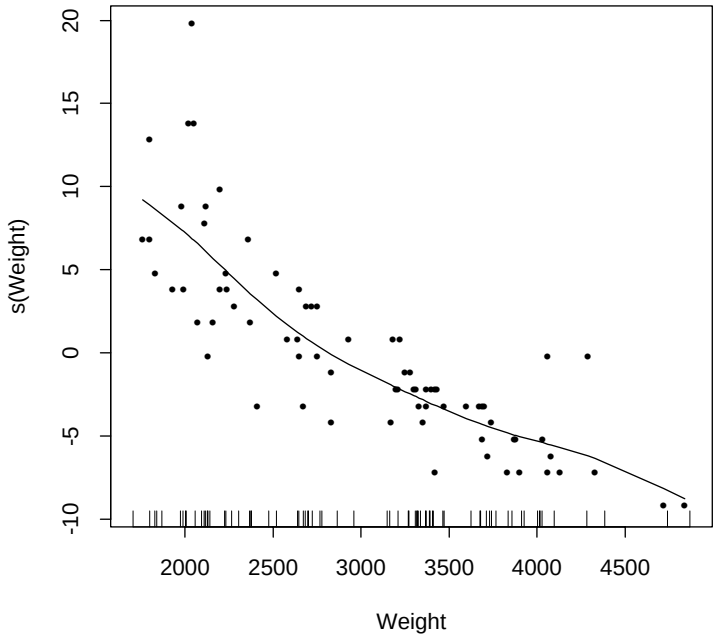

Figure 1.12: *Plot of additive model with smoothed spline term.*

```
Ford Fiesta Honda Civic Plym Champ
   30.17946    30.49947   30.17946
```

Conclusions

The previous examples show a few simple methods for taking data and iteratively fitting models until achieving desired results. The chapters that follow discuss the previously mentioned modeling techniques in far greater detail. Before proceeding further, it is good to remember that:

- General formulas define the structure of models.
- Data used in model-fitting are generally in the form of data frames.
- Different methods can be used on the same data.
- A variety of functions are available for diagnostic study of the fitted models.
- The S-PLUS functions, like model-fitting in general, are designed to be very flexible for users. Handling different preferences and procedures in model-fitting are what make S-PLUS very effective for data analysis.

SPECIFYING MODELS IN S-PLUS 2

Models are specified in S-PLUS using formulas; often one formula is all that is needed for many modeling techniques.

SPECIFYING MODELS IN S-PLUS

Models are specified in S-PLUS using *formulas*, which express the conjectured relationships between observed variables in a natural way. Once you begin building models in S-PLUS, you quickly discover that formulas specify models for the wide variety of modeling techniques available in S-PLUS. You can use the same formula to specify a model for linear regression (`lm`), analysis of variance (`aov`), generalized linear modeling (`glm`), generalized additive modeling (`gam`), local regression (`loess`), and tree-based regression (`tree`).

For example, consider the following formula:

```
> mpg ~ weight + displ
```

This formula can specify a least squares regression with `mpg` regressed on two predictors, `weight` and `displ`, or a generalized additive model with purely linear effects.

You can also specify smoothed fits for `weight` and `displ` in the generalized additive model as follows:

```
> mpg ~ s(weight) + s(displ)
```

and compare the resulting fit with the purely linear fit to see if some nonlinear structure must be built into the model.

Thus, formulas provide the means for you to specify models for all modeling techniques: parametric or nonparametric, classical or modern. This chapter provides you with an introduction to the syntax used for specifying statistical models.

The chapters that follow make use of this syntax in a wide variety of specific examples.

2.1 BASIC FORMULAS

A formula is an S-PLUS expression that specifies the form of a model in terms of the variables involved. For example, to specify that `mpg` is modeled as a linear and additive model of the two predictors `weight` and `displ`, you use the following formula:

```
> mpg ~ weight + displ
```

The tilde (~) character separates the response variable from the explanatory variables. For something to be interpretable as a variable it must be one of the following:

- numeric vector
- factor or ordered factor

- matrix

For numeric vectors, one coefficient is fit; for matrices, a coefficient for each column is fit; for factors, the *equivalent* of one coefficient is fit for *each* level of the factor.

You can use any acceptable S-PLUS expression in the place of any of the variables, provided the expression evaluates to something *interpretable as one or more variables*. Thus, the formula

```
> log(mpg) ~ weight + poly(displ,2)
```

specifies that the log of `mpg` is modeled as a linear function of `weight` and a quadratic polynomial of `displ`.

Continuous Data

Each continuous variable you provide generates one coefficient in the fitted model. Thus the formula

```
> mpg ~ weight + displ
```

fits the model

$$\text{mpg} = \beta_0 + \beta_1 \text{ weight} + \beta_2 \text{ displ} + \varepsilon$$

A formula *always* implicitly includes an intercept term (β_0 in the above formula).

You can, however, remove the intercept term by specifying the model with `-1` as an explicit predictor:

```
> mpg ~ -1 + weight + displ
```

Similarly, you can explicitly include an intercept with a `+ 1`.

When you provide a numeric matrix as one term in a formula, each column of the matrix is taken to be a separate variable in the model. Any names associated with the columns are carried along as labels in the subsequent fits.

Categorical Data

When you specify categorical variables (factors, ordered factors, or categories) as predictors in the formulas, the modeling functions fit a coefficient for each level of the variable. For example, to model `salary` as a linear model of `age` (continuous) and `gender` (factor) you specify it as follows:

```
> salary ~ age + gender
```

However, a different parameter is fitted for each of the two levels of gender. This is equivalent to fitting two dummy variables—one for males and one for females. Thus you need not create and specify dummy variables in the model. (In actuality only one additional parameter is fitted, because the parameters are not independent of the intercept term. More details on over-parameterization and the defining of contrasts between factor levels is provided in section 2.7, Contrasts: The Coding of Factors.

General Formula Syntax

This section provides a table summarizing the meanings of the operators in formulas, and shows how to create and save formulas.

Table 2.1, based on page 29 of *Statistical Models in S*, summarizes the syntax of formulas.

Table 2.1: *A summary of formula syntax.*

Expression	Meaning
T ~ F	T is modeled as F
F_a + F_b	Include both F_a and F_b in the model
F_a - F_b	Include all of F_a except what is in F_b in the model
F_a : F_b	The interaction between F_a and F_b
F_a * F_b	F_a + F_b + F_a : F_b
F_b `%in%` F_a	F_b is nested within F_a
F_a / F_b	F_a + F_b `%in%` F_a
F^m	All terms in F crossed to order m

You can create and save formulas as objects using the `formula` function:

```
> form.eg.1 <- formula(Fuel ~ poly(Weight, 2) + Disp. +
+ Type)
> form.eg.1
Fuel ~ poly(Weight, 2) + Disp. + Type
```

2.2 INTERACTIONS IN FORMULAS

You can specify interactions for categorical data (e.g., factors), continuous data, or a mixture of the two. In each case, additional parameters are fitted that are appropriate for the different types of variables specified in the model. The syntax for specifying the interaction is the same in each case, but the interpretation varies depending on the data types.

To specify a particular interaction between two or more variables use a colon (`:`) between the variable names. Thus to specify the interaction between `gender` and `race`, use the following term:

```
gender:race
```

You can use an asterisk (*) to specify *all* terms in the model created by the subsets of the variables named along with the *. Thus

```
salary ~ age * gender
```

is equivalent to

```
salary ~ age + gender + age:gender
```

You can remove terms with a minus or hyphen (-). Thus

```
salary ~ gender*race*education - gender:race:education
```

is equivalent to

```
salary ~ gender + race + education + gender:race +
gender:education + race:education
```

the model consisting of all the terms in the full model except the three-way interaction. A third way to specify this model is by using the power notation to get all terms of order two or less:

```
salary ~ (gender + race + education) ^ 2
```

Categorical Data

For categorical data, interactions add coefficients for each combination of the levels of the named factors. Thus, for two factors, `Opening` and `Mask`, with three and five levels, respectively, the `Opening:Mask` term in a model adds 15 additional parameters to the model. (In practice, because of dependencies among the parameters, only some of the total number of parameters specified by a model are fitted.)

You can specify, for example, a two-way analysis of variance with the simplified notation as follows:

```
skips ~ Opening * Mask
```

The fitted model is

$$\text{skips} = \mu + \text{Opening}_i + \text{Mask}_j + (\text{Opening} : \text{Mask})_{ij} + \varepsilon$$

Continuous Data

You can specify interactions between continuous variables in the same way as you do for categorical and a mixture of categorical and continuous variables. However, the interaction specified is multiplicative. Thus

```
mpg ~ weight * displ
```

fits the model

$$\text{mpg} = \beta_0 + \beta_1 \text{weight} + \beta_2 \text{displ} + \beta_3 (\text{weight})(\text{displ}) + \varepsilon$$

2.3 NESTING IN FORMULAS

Nesting arises in models when the levels of one or more factors make sense only *within* the levels of one or more other factors. For example, in sampling

the U.S. population, a sample of states is drawn, from which a sample of counties is drawn, from which a sample of cities is drawn, from which a sample of families or households is drawn. Counties are nested within states, cities are nested within counties, and households are nested within cities.

There is special syntax to emphasize the nesting of factors within others. You can write the county within state model as:

```
state + county
```

You can state the model more succinctly with

```
state / county
```

which means "state and then county within state." The slash (`/`) used for nested models is the counterpart of the asterisk (`*`) which is used for factorial models.

You can specify the full state-county-city-household example as follows:

```
state / county / city / household
```

2.4 INTERACTIONS BETWEEN CATEGORICAL AND CONTINUOUS VARIABLES

For categorical data combined with continuous data, *interactions* add a coefficient for the continuous variable for each level of the categorical variable. So, for example, you can easily fit a model with different slope estimates for different groups where the categorical variables specify the groups.

When you combine categorical and continuous data using the *nesting* syntax, you can specify analysis of covariance models simply. If gender (categorical) and age (continuous) are predictors in a model, you can fit separate slopes for each gender by nesting. First, make gender a factor (i.e., `gender <- factor(gender)`). Then the analysis of covariance model is:

```
salary ~ gender / age
```

This fits a model equivalent to:

$$\mu + \text{gender}_i + \beta_i \text{age}$$

This is also equivalent to `gender * age`. However, the *parameterization* for the two models is different. When you fit the nested model, you get estimates of the individual slopes for each group. When you fit the factorial model, you get an overall slope estimate plus the deviations in the slope for the different group contrasts. For example, in `gender / age`, the formula expands into main effects for `gender` followed by `age` within each level of `gender`. One coefficient is fitted for `age` from each level of `gender`. Another coefficient estimates the contrast between the two levels of `gender`. Thus, the nesting

model fits the following type of model:

$$\text{Salary}_M = \mu + \alpha_g + \beta_1 \times \text{age}$$

$$\text{Salary}_F = \mu - \alpha_g + \beta_2 \times \text{age}$$

The intercept is μ, the contrast is α_g, and the model has coefficients β_i for `age` within each level of `gender`. Thus, you have separate slope estimates for each group. Conversely, the factorial model `gender * age` fits the following model:

$$\text{Salary}_M = \mu - \alpha_g + \beta \times \text{age} - \gamma \times \text{age}$$

$$\text{Salary}_F = \mu + \alpha_g + \beta \times \text{age} + \gamma \times \text{age}$$

You get the overall slope estimate β plus the deviations in the slope for the different group contrasts.

You can fit the "equal slope, separate intercept" model by specifying:

```
salary ~ gender + age
```

This fits a model equivalent to:

$$\mu + \text{gender}_i + \beta \times \text{age}$$

2.5 USING THE PERIOD OPERATOR IN FORMULAS

A single period (".") operator can act as a default left or right side of a formula. There are numerous ways you can use "." in formulas. To see how "." is used, consider the function `update`, which allows you to modify existing models. The following example uses the data frame `fuel.frame` to display the usage of the single "." in formulas:

```
> fuel.null <- lm(Fuel ~ 1, fuel.frame)
```

If `Weight` is the single best predictor, use `update` to add it to the model:

```
> fuel.wt <- update(fuel.null, . ~ . + Weight)
> fuel.wt
Call:
lm(formula = Fuel ~ Weight, data = fuel.frame)

Coefficients:
 (Intercept)     Weight
   0.3914324 0.00131638

Degrees of freedom: 60 total; 58 residual
Residual standard error: 0.387715
```

The single dots "." in the above example are replaced on the left and right side of the tilde "~" by the left and right sides of the formula used to fit the object `fuel.null`. Two additional methods use "." in reference to data frame objects. In the following example, a linear model is fit using the data frame `fuel.frame`:

```
> lm(Fuel ~ ., data = fuel.frame)
```

Here, the new model includes all the predictors in `fuel.frame`. In the example,

```
> lm(skips ~ .^2, data = solder.balance)
```

all main effects and second-order interactions in `solder.balance` are used to fit the model.

2.6 COMBINING FORMULAS WITH FITTING PROCEDURES

Once you specify a model with its associated formula, you can fit it to a given data set by passing the formula and the data to the appropriate fitting procedure. For the following example, you create the data frame `auto.dat` from the data set `auto.stats` by typing,

```
> auto.dat <- data.frame(auto.stats)
```

To fit a linear model to `Miles.per.gallon ~ Weight + Displacement`, when `Miles.per.gallon`, `Weight`, and `Displacement` are columns in a data frame named `auto.dat`, you type:

```
> lm(Miles.per.gallon ~ Weight + Displacement, auto.dat)
```

You could fit a *smoothed* model to the same data with:

```
> loess(Miles.per.gallon ~ s(Weight) + s(Displacement),
+ auto.dat)
```

All the fitting procedures take a formula and an optional data set (actually a data frame) as the first two arguments. If the individual variables are in your search path, or you attached the data frame `auto.dat`, you can omit the data specification and type more simply:

```
> lm(Miles.per.gallon ~ Weight + Displacement)
```

or

```
> loess(Miles.per.gallon ~ s(Weight) + s(Displacement))
```

Warning

If you attach a data frame for fitting models and have objects in your **.Data** directory with names that match those in the data frame, the data frame variables are *masked* and are not used in the actual model fitting.

Composite Terms in Formulas

As was previously mentioned, certain operators have special meaning when used in formula expressions. They must appear only at the top level in the formulas and only on the right side of the "~". However, if the operators appear within arguments to functions within the formula, then they work as they normally do in S-PLUS. In the formula

```
Kyphosis ~ poly(Age, 2) + I((Start > 12) * (Start - 12))
```

the '*' and '-' operators evaluate as they normally do in S-PLUS, without the special meaning they have when used at the top level within the formula because they appear within arguments to the `I` function. The `I` function's sole purpose, in fact, is to protect special operators on the right side of formulas.

You can use any acceptable S-PLUS expression in the place of any variable within the formula, provided the expression evaluates to something *interpretable* as *one or more variables*. The expression must be one of the following:

- numeric vector
- factor, ordered factor, or category
- matrix

Thus, certain composite terms (among them `poly`, `I`, and `bs`) can be used as formula variables. Matrices used in formulas are treated as single terms. You can also use functions that produce factors and categories as formula variables.

2.7 CONTRASTS: THE CODING OF FACTORS

A coefficient for each level of a factor cannot usually be estimated because of dependencies among the coefficients of the overall model. An example of this is the sum of all the *dummy* variables for any factor, which is a vector of all ones. This corresponds to the term used for fitting an intercept. Overparametrization induced by dummy variables is removed prior to fitting, by replacing the dummy variables with a set of linear combinations of the dummy variables, which are:

- functionally independent of each other, and
- functionally independent of the *sum* of the dummy variables.

A factor with k levels has k-1 possible independent linear combinations. A particular choice of linear combinations of the dummy variables is called a set of *contrasts*. Any choice of contrasts for a factor alters the specific individual coefficients in the model, but *does not change the overall contribution of the term to the fit*.

Built-in Contrasts

S-PLUS provides four different kinds of contrasts as built-in functions:

- **Helmert Contrasts**

  The function `contr.helmert` implements Helmert contrasts. The *j*th linear combination is the difference between the level *j* + 1 and the average of the first *j*. The following example returns a Helmert parametrization based upon four levels:

```
> contr.helmert(4)
  [,1] [,2] [,3]
1   -1   -1   -1
2    1   -1   -1
3    0    2   -1
4    0    0    3
```

- **Orthogonal Polynomials**

  The function `contr.poly` implements polynomial contrasts. Individual coefficients represent orthogonal polynomials if the levels of the factor are equally spaced numeric values. In general, the function produces *k* - 1 orthogonal contrasts representing polynomials of degree 1 to *k* - 1. The following example uses four levels:

```
> contr.poly(4)
               L     Q          C
[1,] -0.6708204   0.5 -0.2236068
[2,] -0.2236068  -0.5  0.6708204
[3,]  0.2236068  -0.5 -0.6708204
[4,]  0.6708204   0.5  0.2236068
```

- **Sum**

  The function `contr.sum` implements sum contrasts. This produces contrasts between each of the first *k* - 1 levels and level *k*:

```
> contr.sum(4)
  [,1] [,2] [,3]
1    1    0    0
2    0    1    0
3    0    0    1
4   -1   -1   -1
```

- **Treatment**

  The function `contr.treatment` implements treatment contrasts. This is not really a contrast but simply includes each level as a

dummy variable excluding the first one. *This generates statistically dependent coefficients even in balanced experiments.*

```
> contr.treatment(4)
  2 3 4
1 0 0 0
2 1 0 0
3 0 1 0
4 0 0 1
```

This is not a true set of contrasts, for the columns do not sum to zero and thus are not orthogonal to the vector of ones.

Specifying Contrasts

Use the functions `C`, `contrasts`, and `options` to specify contrasts. Use `C` to specify a contrast as you type the formula; it is the simplest way to alter the choice of contrasts. Use `contrasts` to specify a contrast attribute on a factor. Use `options` to specify the default choice of contrasts for all factors.

The `C` Function

As was previously stated, the `C` function is the simplest way to alter the choice of contrasts. The arguments to the function are `C(object, contr)` where `object` is a factor or ordered factor, and `contr` is the contrast to alter. An optional argument, `how.many`, is for the number of contrasts to assign to the factor. The value returned by `C` is the factor with a `"contrasts"` attribute equal to the specified contrast matrix.

For example, with the soldering experiment contained in `solder.balance` you could specify `sum` contrasts for `Mask` with `C(Mask, sum)`. You could also have your own contrast function, `special.contrast`, that returns a matrix of the desired dimension with the call `C(Mask, special.contrast)`.

Note

If you create your own contrast function it must return a matrix with the following properties:

1. The number of rows must be equal to the number of levels specified and the number of columns one less than the number of rows.
2. The columns must be linearly independent of each other and of the vector of all ones.

You can also specify contrasts by supplying the contrast matrix directly. For example, `quality` is a factor with four levels:

```
> levels(quality)
[1] "tested-low"  "low"      "high"      "tested-high"
```

You can contrast levels 1 and 4 with levels 2 and 3 by including `quality` in the model formula as `C(quality, c(1, -1, -1, 1))`. Two additional contrasts are generated, orthogonal to the one supplied.

To contrast the "low" values versus the "high" values, provide the contrasts as a matrix:

```
> contrast.mat
     [,1] [,2]
[1,]    1    1
[2,]   -1    1
[3,]   -1   -1
[4,]    1   -1
```

The `contrasts` Function

Use the `contrasts` function to *set* the contrasts for a particular factor whenever it appears. The `contrasts` function extracts contrasts from a factor and returns them as a matrix. The following sets the contrasts for the `quality` factor:

```
> contrasts(quality) <- contrast.mat
> contrasts(quality)
            [,1] [,2] [,3]
 tested-low    1    1 -0.5
        low   -1    1  0.5
       high   -1   -1 -0.5
tested-high    1   -1  0.5
```

Now `quality` has the `contrast.mat` parametrization by default any time it appears in the formula. To override this new default setting, supply a new contrast specification through the `C` function.

Setting the `contrasts` Option

Use the contrast `options` function to change the default choice of contrasts for *all* factors, as in the following example:

```
> options()$contrasts
         factor      ordered
 "contr.helmert" "contr.poly"
> options(contrasts = c(factor = "contr.treatment",
+ ordered = "contr.poly"))
> options()$contrasts
[1] "contr.treatment" "contr.poly"
```

In summary, the `options` function sets the default choice of contrasts globally (on all factors); the `contrasts` function sets the default choice of contrasts on a particular factor; and the `C` function overrides the default.

2.8 USEFUL FUNCTIONS FOR MODEL FITTING

As model building proceeds, you'll find several functions useful for adding and deleting terms in formulas. The `update` function starts with an existing fit and adds or removes terms as you specify. For example, create a data frame from the data set `fuel.frame` by typing:

```
> fuel.fit <- data.frame(fuel.frame)
```

Suppose you save the result of `lm` as follows:

```
> fuel.lm.fit <- lm(Mileage ~ Weight + Disp., fuel.fit)
```

You can use `update` to change, for example, the response to `Fuel`. Use a period on either side of the tilde to represent the current state of the model in the fit object (`fuel.lm.fit` below).

```
> update(fuel.lm.fit, Fuel ~ . )
```

Recall that the period (".") means to include every predictor that is in `fuel.lm.fit` in the new model. Only the response changes.

You could drop the `Disp.` term, keeping the response the same by:

```
> update(fuel.lm.fit, . ~ . - Disp.)
```

Another useful function is `drop1`, which produces statistics obtained from dropping each term out of the model one at a time. For example:

```
> drop1(fuel.lm.fit)
Single term deletions

Model: Mileage ~ Weight + Disp.
       Df Sum of Sq   RSS    Cp
<none>              380.3 420.3
Weight  1     323.4 703.7 730.4
Disp.   1       0.6 380.8 407.5
```

Each line presents model summary statistics corresponding to dropping the term indicated in the first column. The first line in the table corresponds to the original model, that is, no terms (<none>) are deleted.

There is also an `add1` function which adds one term at a time. The second argument to `add1` provides the *scope* for added terms. The scope argument can be a formula or a character vector indicating the terms to be added. The resulting table prints a line for each term indicated by the scope argument.

```
> add1(fuel.lm.fit, c("Type", "Fuel"))
Single term additions

Model: Mileage ~ Weight + Disp.
```

```
       Df Sum of Sq     RSS      Cp
<none>              380.271 420.299
  Type  5   119.722 260.549 367.292
  Fuel  1   326.097  54.173 107.545
```

2.9 OPTIONAL ARGUMENTS TO MODEL-FITTING FUNCTIONS

In most model-building calls, you will need to specify the data frame to use. You may need arguments that check for missing values in the data frame, or select only particular portions of the data frame to use in the fit. The following list summarizes standard optional arguments for most model-fitting functions (other than nonlinear models) you can use in the model fit:

- `data`: specifies a data frame to interpret the variables named in the formula, or in the `subset` and `weights` arguments. The following example fits a linear model to data in the `fuel.frame` data frame:

  ```
  > fuel.lm <- lm(Fuel ~ Weight + Disp., data = fuel.frame)
  ```

- `weights`: specifies a vector of observation of weights. If `weights` is supplied, the algorithm fits to minimize the sum of the squared residuals multiplied by the weights:

  $$\sum w_i r_i^2$$

  Negative weights generate an S-PLUS error. We recommend that the weights be strictly positive, since zero weights give no residuals. The following example fits a linear model to the `claims` data frame, and uses `number` with the `weights` argument:

  ```
  > claims.fit <- lm(cost ~ age + type + car.age, claims,
  + weights = number, na.action = na.omit)
  ```

The `number` in the preceding call corresponds to the number of claims per type of car in the `claims` data frame.

- `subset`: indicates a subset of the rows of the data to be used in the fit. The expression should evaluate to a logical or numeric vector, or a character vector with appropriate row names. The following example removes outliers and fits a linear model to data in the `auto.dat` data frame:

  ```
  > fit <- lm(1/Miles.per.gallon ~ Weight,
  + subset = -outliers)
  ```

- `na.action`: a missing-data filter function, applied to the model frame, after any subset argument has been used. The following

example uses `na.omit` with the `na.action` argument to drop any row of the data frame that contains a missing value:

```
> ozone.lm <- lm(ozone ~ temperature + wind, data= air,
+             subset=wind > 8, na.action=na.omit)
```

Each model fitting function has nonstandard optional arguments, not listed above, which you can use to fit the appropriate model. The following chapters describe the available arguments for each model type.

STATISTICAL INFERENCE FOR ONE AND TWO SAMPLE PROBLEMS

3

Sample means and variances give information about underlying distributions.

STATISTICAL INFERENCE FOR ONE AND TWO SAMPLE PROBLEMS

Suppose you have one or two samples of data that are *continuous* in the sense that the individual observations can take on any possible value in an interval. You often want to draw conclusions from your data concerning underlying "population" or distribution model parameters which determine the character of the observed data. The parameters which are most often of interest are the mean and variance in the case of one sample, and the relative means and variances and the correlation coefficient in the case of two samples. This chapter shows you how to use S-PLUS to carry out statistical inference for these parameters.

Often, your samples of data are assumed to come from a distribution that is *normal*, or *Gaussian*. A normal distribution has the familiar bell-shaped population "frequency" curve (or *probability density*) shown by the solid line in figure 3.1. Another common assumption is that the observations *within* a sample are *serially uncorrelated* with one another. In fact, the data seldom come from an exactly normal distribution. Usually, a more accurate assumption is that the samples are drawn from a *nearly normal* distribution—that is, a nearly bell-shaped curve whose tails do not go to zero in quite the same way as those of the true normal distribution, as shown by the dotted line in figure 3.1.

It is important that you be aware that nearly normal distributions, which have "heavier tails" than a normal distribution, give rise to *outliers*, i.e., unusually aberrant or deviant data values. For example, in figure 3.1 the left-hand tail of the nearly normal distribution is heavier than the tail of the normal distribution, but the right hand tail is not, and so this nearly normal distribution generates outliers which fall to the left (smaller values than) the bulk of the data.

Even though your data has only a nearly normal distribution, rather than a normal distribution, you can use a normal distribution as a good "nominal" model, as indicated by figure 3.1. Thus you are interested in knowing the values of the parameters of a normal distribution, (or of two normal distributions in the case of two samples), that provides a good nominal distribution model for your data.

A normal distribution is characterized by two parameters: the *mean* μ and the *variance* σ^2, or, equivalently, the mean and the *standard deviation* σ (the square root of the variance). The mean *locates* the center of symmetry of the normal distribution, and so the parameter μ is sometimes referred to as the

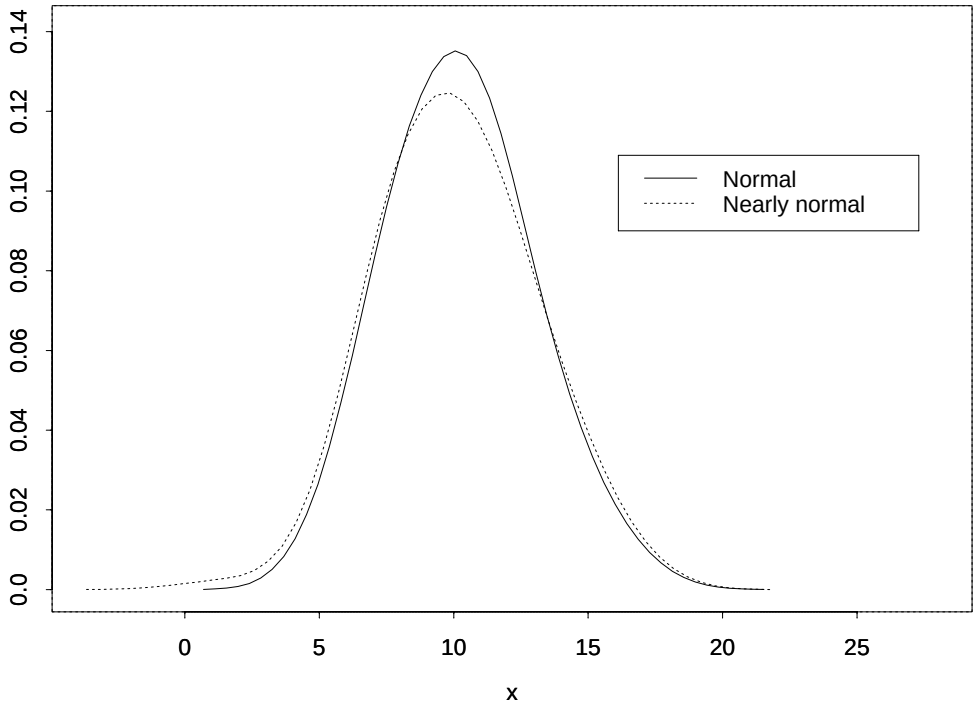

Figure 3.1: *Normal and nearly normal densities.*

location. Similarly, the standard deviation provides a measure of the spread of the distribution, and thus can be thought of as a *scale* parameter.

In the case of two samples, X_1, X_2, …, X_n and Y_1, Y_2, …, Y_n, for two variables X and Y, you may also be interested in the value of the *correlation coefficient* ρ. The parameter ρ measures the correlation (or linear dependency) between the variables X and Y. The value of ρ is reflected in the *scatter plot* obtained by plotting Y_i versus X_i for i=1,2, …,n. A scatterplot of Y_i versus X_i which has a roughly elliptical shape, with the values of Y_i increasing with increasing values of X_i, corresponds to positive correlation ρ (see for example, figure 3.7). An elliptically-shaped scatter plot with the values of Y_i decreasing with increasing values of X_i corresponds to negative correlation r. A circular shape to the scatter plot corresponds to a zero value for the correlation coefficient ρ.

Keep in mind that the correlation between two variables X and Y, as just described, is quite distinct from serial correlation between the observations within one or both of the samples when the samples are collected over time.

Whereas the former reveals itself in a scatterplot of the Y_i versus the X_i, the latter reveals itself in scatter plots of the observations versus *lagged* values of the observations; for example, a scatter plot of Y_i versus Y_{i+1} or a scatter plot of X_i versus X_{i+1}. If these scatter plots have a circular shape, the data is serially uncorrelated. Otherwise, the data has some serial correlation.

Generally, you must be careful not to assume that data collected over time is serially uncorrelated. You need to check this assumption carefully, because the presence of serial correlation invalidates most of the methods of this chapter.

To summarize: You want to draw conclusions from your data concerning the population mean and variance parameters μ and σ^2 for one sample of data, and you want to draw conclusions from your data concerning the population means μ_1, μ_2, the population variances σ_1^2, σ_2^2 and the population correlation coefficient ρ for two samples of data. You frame your conclusions about the above parameters in one of the following two types of *statistical inference* statements, illustrated for the case of the population mean μ in a one sample problem:

- *A CONFIDENCE INTERVAL.* With probability 1 - α, the mean μ lies within the *confidence interval* (L,U).
- *A HYPOTHESIS TEST.* The computed statistic T compares the *null hypothesis* that the mean μ has the specified value μ_0 with the *alternative hypothesis* that $\mu \neq \mu_0$. At any level of significance greater than the reported *p-value* for T, we *reject* the null hypothesis in favor of the alternative hypothesis.

A more complete description of confidence intervals and hypothesis tests is provided in the section Statistical Inference on page 46.

Classical methods of statistical inference, such as *Student's t* methods, rely on the assumptions that the data come from a normal distribution and the observations within a sample are serially uncorrelated. If your data contain outliers, or are strongly non-normal, or if the observations within a sample are serially correlated, the classical methods of statistical inference can give you very misleading results. Fortunately, there are *robust* and *nonparametric* methods which give reliable statistical inference for data that contain outliers or are strongly non-normal. Special methods are needed for dealing with data that are serially correlated. See, for example, Heidelberger and Welch (1981).

In this chapter, you learn to use S-PLUS functions for making both classical and robust or nonparametric statistical inference statements for the population means and variances for one and two samples, and for the

population correlation coefficient for two samples. The basic steps in using S-PLUS functions are essentially the same no matter which of the above parameters you are interested in. They are as follows:

1. *Setting up your data.*
   Before S-PLUS can be used to analyze the data, you must put the data in a form that S-PLUS recognizes.
2. *Exploratory data analysis, or EDA.*
   EDA is a graphically-oriented method of data analysis which helps you determine whether the data support the assumptions required for the classical methods of statistical inference: an outlier-free nearly normal distribution and serially uncorrelated observations
3. *Statistical inference.*
   Once you've verified that your sample or samples are nearly normal, outlier-free, and uncorrelated, you can use classical methods of statistical inference which assume a normal distribution and uncorrelated observations, to draw conclusions from your data.

If your data are not nearly normal and outlier-free, the results of the classical methods of statistical inference may be misleading. Hence, you often need "robust" or "nonparametric" methods, as described in the section Robust and Nonparametric Methods on page 48.

3.1 BACKGROUND

This section prepares you for using the S-PLUS functions in the remainder of the chapter by providing brief background information on the following three topics: *exploratory data analysis*, *statistical inference*, and *robust and nonparametric methods*.

Exploratory Data Analysis

The classical methods of statistical inference depend heavily on the assumption that your data is outlier-free and nearly normal, and that your data is serially uncorrelated. *Exploratory data analysis* (EDA) uses graphical displays to help you obtain an understanding of whether or not such assumptions hold. Thus you should always carry out some graphical exploratory data analysis (EDA) to answer the following questions:

- Do the data come from a nearly normal distribution?
- Do the data contain outliers?
- If the data were collected over time, is there any evidence of serial correlation (correlation between successive values of the data)?

You can get a pretty good picture of the shape of the distribution generating your data, and also detect the presence of outliers, by looking at the following collection of four plots: a *histogram*, a *boxplot*, a *density plot*, and a *normal qqplot*. Examples of these four plots are provided by figure 3.2.

Density plots are essentially smooth versions of histograms, which provide smooth estimates of population *frequency*, or *probability density* curves; for example, the normal and nearly normal curves of figure 3.1. Since the latter are smooth curves, it is both appropriate and more pleasant to look at density plots than at histograms.

A normal qqplot (or quantile-quantile plot) consists of a plot of the ordered values of your data versus the corresponding quantiles of a *standard* normal distribution; that is, a normal distribution with mean zero and variance one. If the qqplot is fairly linear, your data are reasonably Gaussian; otherwise, they are not.

Of these four plots, the histogram and density plot give you the best picture of the distribution shape, while the boxplot and normal qqplot give the clearest display of outliers. The boxplot also gives a clear indication of the median (the solid dot inside the box), and the upper and lower quartiles (the upper and lower ends of the box).

A simple S-PLUS function can create all four suggested distributional shape EDA plots, and displays them all on a single screen or a single hard copy plot. Define the function as follows:

```
> eda.shape <- function(x)
+ {
+       par(mfrow = c(2, 2))
+       hist(x)
+       boxplot(x)
+       iqd <- summary(x)[5] - summary(x)[2]
+       plot(density(x,width=2*iqd), xlab = "x",
+       ylab = "", type = "l")
+       qqnorm(x)
+       qqline(x)
+ }
```

This function is used to make the EDA plots you see in the remainder of this chapter. (The argument `width=2*iqd` to `density` sets the degree of smoothness of the density plot in a good way. For more details on writing functions, see the S-PLUS Programmer's Guide.)

If you have collected your data over time, the data may contain serial correlation. That is, the observations may be correlated with one another at different times. The assessment of whether or not there is any time series

correlation in the context of confirmatory data analysis for location and scale parameters (and more generally) is an often-neglected task.

You can check for obvious time series features, such as trends and cycles, by looking at a plot of your data against time, using the function `ts.plot` (see the chapter entitled "Visualizing One and Two Dimensional Data" in the S-PLUS User's Guide). You can check for the presence of less obvious serial correlation by looking at a plot of the autocorrelation function for the data, using the `acf` function. These plots can be created, and displayed one above the other, with the following S-PLUS function:

```
> eda.ts <- function(x)
+ {
+       par(mfrow=c(2,1))
+       ts.plot(x)
+       acf(x)
+       invisible()
+ }
```

This function is used to make the time series EDA plots you find in the remainder of this chapter. See, for example, figure 3.3. The discussion of figure 3.3 includes a guideline for interpreting the `acf` plot. (See the chapter Analyzing Time Series for a complete description of `acf`.)

Warning

If either the time series plot or the acf plot suggests the presence of serial correlation, then you can place little credence in the results computed in this chapter, using either the Student's *t*-statistic approach or using the nonparametric Wilcoxon approach! A method for estimating the population mean in the presence of serial correlation is described by Heidelberger and Welch (1981). Seek expert assistance, as needed.

Statistical Inference

Formal methods of *statistical inference* provide probability-based statements about population parameters such as the mean, variance, and correlation coefficient for your data. You may be interested in a simple *(point) estimate* of a population parameter. For example, the sample mean is a point estimate of the population mean. However, a point estimate neither conveys any uncertainty about the value of the estimate, nor indicates whether a hypothesis about the population parameter is to be rejected. To address these two issues, you will usually use one or both of the following methods of statistical inference: *confidence intervals* and *hypothesis tests*.

We define these two methods for you, letting θ represent any one of the parameters you may be interested in; for example, θ may be the mean μ, or the difference between two means $\mu_1 - \mu_2$, or the correlation coefficient ρ.

CONFIDENCE INTERVALS.

A (1-α)100% confidence interval for the true but unknown parameter θ is any interval of the form (*L*,*U*), such that the probability is 1 - α that (*L*,*U*) contains θ. The probability a with which the interval (*L*,*U*) fails to cover q is sometimes called the *error rate* of the interval. The quantity (1-α)$\times$100% is called the *confidence level* of the confidence interval. Common values of α are α=.01, .05, .1, which yield 99%, 95%, and 90% confidence intervals, respectively.

HYPOTHESIS TESTS. A hypothesis test is a probability-based method for making a decision concerning the value of a population parameter θ (for example, the population mean μ or standard deviation σ in a one-sample problem), or the relative values of two population parameters θ_1 and θ_2 (for example, the difference between the population means μ_1 - μ_2 in a two sample problem). You begin by forming a *null hypothesis* and an *alternative hypothesis*. For example, in the two sample problem your null hypothesis is often the hypothesis that $\theta_1 = \theta_2$, and your alternative hypothesis is one of the following:

- the *two-sided* alternative: $\theta_1 \neq \theta_2$
- the *greater-than* alternative: $\theta_1 > \theta_2$
- the *less-than* alternative: $\theta_1 < \theta_2$

Your decision to *accept* the null hypothesis, or to *reject* the null hypothesis in favor of your alternative hypothesis is based on the observed value $T = t_{obs}$ of a suitably chosen test statistic *T*. The probability that the statistic *T* "exceeds" the observed value t_{obs} when your null hypothesis is in fact true, is called the *p-value*.

For example, suppose you are testing the null hypothesis that $\theta = \theta_0$ against the alternative hypothesis that $\theta \neq \theta_0$ in a one-sample problem. The *p*-value is the probability that the absolute value of *T* exceeds the absolute value of t_{obs} for your data, when the null hypothesis is true.

In *formal* hypothesis testing, you proceed by choosing a "good" statistic *T* and specifying a *level of significance*, which is the probability of rejecting a null hypothesis when the null hypothesis is in fact true.

In terms of formal hypothesis testing, your *p*-value has the following interpretation: the *p*-value is the level of significance for which your observed test statistic value t_{obs} lies on the boundary between acceptance and rejection of the null hypothesis. At any significance level greater than the *p*-value, you

reject the null hypothesis, and at any significance level less than the *p*-value you accept the null hypothesis. For example, if your *p*-value is .03, you reject the null hypothesis at a significance level of .05, and accept the null hypothesis at a significance level of .01.

Robust and Nonparametric Methods

Two problems frequently complicate your statistical analysis. For example, Student's *t*-test, which is the basis for most statistical inference on the mean-value locations of normal distributions, relies on two critical assumptions:

1. The observations have a common normal (or Gaussian) distribution with mean μ and variance σ^2.
2. The observations are independent.

However, one or both of these assumptions often fail to hold in practice.

For example, if the actual distribution for the observations is an outlier-generating, heavy-tailed deviation from an assumed Gaussian distribution, the confidence level remains quite close to $(1-\alpha)100\%$, but the average confidence interval length is considerably larger than under normality. The *p*-values based on the Student's *t* test are also heavily influenced by outliers.

In this example, and more generally, you would like to have statistical methods with the property that the conclusions you draw are not much affected if the distribution for the data deviates somewhat from the assumed model; for example, if the assumed model is a normal, or Gaussian distribution, and the actual model for the data is a nearly normal distribution. Such methods are called *robust*. In this chapter you will learn how to use an S-PLUS function to obtain robust point estimates and robust confidence intervals for the population correlation coefficient.

For one and two-sample location parameter problems (among others), there exist *strongly* robust alternatives to classical methods, in the form of *nonparametric* statistics. The term "nonparametric" means that the methods work even when the actual distribution for the data is far from normal; that is, when the data do not have to have even a nearly normal distribution. In this chapter, you will learn to use one of the best of the nonparametric methods for constructing a hypothesis test *p*-value, namely the Wilcoxon rank method, as implemented in the S-PLUS function `wilcox.test`.

It is important to keep in mind that serial correlation in the data can quickly invalidate the use of both classical methods (such as Student's *t*) and nonparametric methods (such as the Wilcoxon rank method) for computing confidence intervals and *p*-values. For example, a 95% Student's *t* confidence interval can have a much higher error rate than 5% when there is a small amount of positive correlation in the data. Also, most modern robust

methods are oriented toward obtaining insensitivity toward outliers generated by heavy-tailed nearly normal distributions, and are not designed to cope with serial correlation. For information on how to construct confidence intervals for the population mean when your data are serially correlated and free of outliers, see Heidelberger and Welch (1981).

3.2 ONE SAMPLE: DISTRIBUTION SHAPE, LOCATION, AND SCALE

In 1876, the French physicist Cornu reported a value of 299,990 km/sec for *c*, the speed of light. In 1879, the American physicist A. A. Michelson carried out several experiments to verify and improve on Cornu's value.

Michelson obtained the following 20 measurements of the speed of light:

```
 850  740  900  1070  930  850  950  980  980  880
1000  980  930   650  760  810 1000 1000  960  960
```

To obtain Michelson's actual measurements in km/sec, add 299,000 km/sec to each of the above values.

The twenty observations can be thought of as observed values of twenty random variables with a common but unknown mean-value location μ. If the experimental setup for measuring the speed of light is free of bias, then it is reasonable to assume that μ is the true speed of light.

In evaluating this data, we seek answers to at least five questions:

1. What is the speed of light μ?
2. Has the speed of light changed relative to our best previous value μ_0?
3. What is the uncertainty associated with our answers to (1) and (2)?
4. What is the shape of the distribution of the data?
5. The measurements were taken over time. Is there any evidence of serial correlation?

The first three questions were probably in Michelson's mind when he gathered his data. The last two must be answered to determine which techniques can be used to obtain valid statistical inferences from the data. For example, if the shape of the distribution indicates a nearly normal distribution without outliers, we can use the Student's *t* tests in attempting to answer question (2). If the data contain outliers or are far from normal, we should use a robust method or a nonparametric method such as the Wilcoxon signed-rank test. On the other hand, if serial correlation exists, neither the Student's *t* nor the Wilcoxon test offers valid conclusions.

In this section, we use S-PLUS to carefully analyze the Michelson data. Identical techniques can be used to explore and analyze any set of one-sample data.

Setting Up the Data

The data form a single, ordered set of observations, so they are appropriately described in S-PLUS as a vector. Use the `scan` function to create the vector `mich`:

```
> mich <- scan()
1: 850 740 900 1070 930
6: 850 950 980 980 880
11: 1000 980 930 650 760
16: 810 1000 1000 960 960
21:
```

Exploratory Data Analysis

To start, we can evaluate the shape of the distribution, by making a set of four EDA plots, using the `eda.shape` function described in the section Exploratory Data Analysis on page 44:

```
> eda.shape(mich)
```

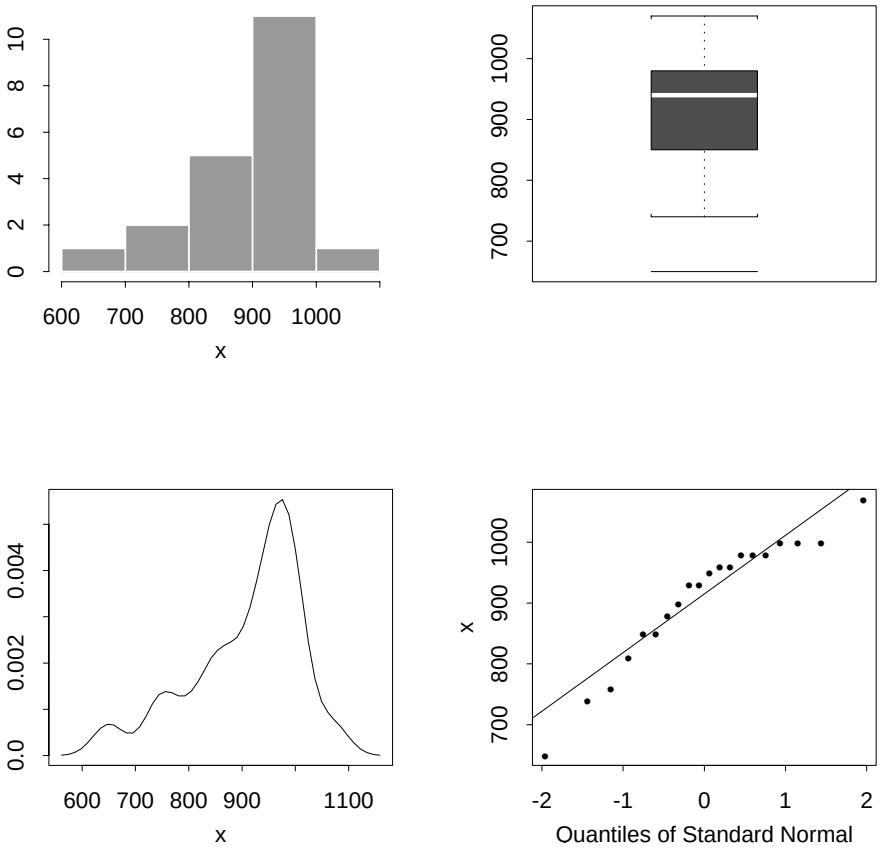

Figure 3.2: *Exploratory data analysis plots.*

The plots in figure 3.2 reveal a distinctly skewed distribution, skewed toward the left (that is, toward smaller values), but rather normal in the middle region. The distribution is thus not normal, and probably not even "nearly" normal.

The solid horizontal line in the box plot is located at the *median* of the data, and the upper and lower ends of the box are located at the *upper quartile* and *lower quartile* of the data, respectively. To get precise values for the median and quartiles, use the `summary` function:

```
> summary(mich)
 Min. 1st Qu. Median Mean 3rd Qu. Max.
  650     850    940  909     980 1070
```

The summary shows, from left to right, the smallest observation, the first quartile, the median, the mean, the third quartile, and the largest observation. From this summary you can compute the interquartile range, $IQR = 3Q - 1Q$. The interquartile range provides a useful criterion for identifying outliers—any observation which is more than $1.5 \times IQR$ above the third quartile or below the first quartile is a suspected outlier.

To examine possible serial correlation, or dependency, make two plots using the `eda.ts` function defined in the section Exploratory Data Analysis on page 44.

```
> eda.ts(mich)
```

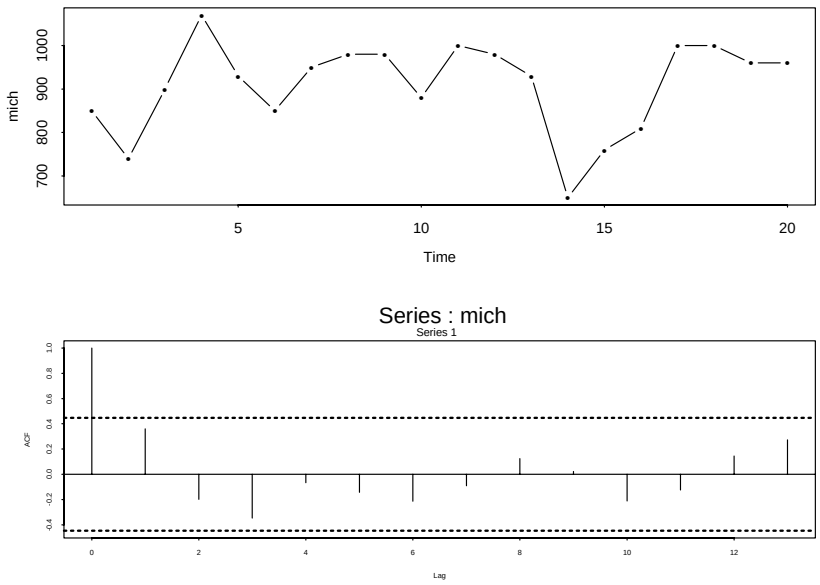

Figure 3.3: *Time series plots.*

The top plot in figure 3.3 reveals a somewhat unusual excursion at observations 14, 15, 16, and perhaps a slightly unusual oscillation in the first 6 observations. However, the autocorrelation function plot in the lower part of figure 3.3 reveals no significant serial correlations—all values lie within the horizontal dashed lines for lags greater than 0.

Statistical Inference

Because the Michelson data are not normal, you should probably use the Wilcoxon signed-rank test rather than the Student's *t*-test for your statistical inference. For illustrative purposes, we'll use both.

To compute Student's *t* confidence intervals for the population mean-value location parameter μ, and to compute Student's *t* significance test *p*-values for the parameter μ_0, use the function `t.test`.

To perform the test, you specify the confidence level, the hypothesized mean-value location μ, and the hypothesis being tested, as follows:

- `conf.level=` specifies the confidence level of the confidence interval. Usual values are `0.90`, `0.95`, or `0.99`. The default is `0.95`.
- `mu=` specifies the null hypothesis value μ_0 of μ. The default is μ_0=0, which is often inappropriate for one sample problems. You should choose μ carefully, using either a previously accepted value or a value suggested by the data before sampling.
- `alternative=` specifies the specific hypothesis being tested. There are three options:
  - `"two.sided"` tests the hypothesis that the true mean is not equal to μ_0. This is the default alternative.
  - `"greater"` tests the hypothesis that the true mean is greater than μ_0.
  - `"less"` tests the hypothesis that the true mean is less than μ_0.

For Michelson's data, suppose you want to test the null hypothesis value μ_0 = 990 (plus 299,000) against a two-sided alternative. Then you use `t.test` with the argument `mu=990`:

```
> t.test(mich,mu=990)

        One-sample t-Test

data:  mich
t = -3.4524, df = 19, p-value = 0.0027
alternative hypothesis: true mean is not equal to 990
```

```
95 percent confidence interval:
 859.8931 958.1069
sample estimates:
 mean of x
       909
```

The *p*-value is 0.0027, which is highly significant. S-PLUS returns other useful information besides the *p*-value, including the *t*-statistic value, the degrees of freedom (`df`), the sample mean, and the confidence interval.

Our example used the default confidence level of .95. If you specify a different confidence level, as in the following command:

```
> t.test(mich,conf.level=.90,mu=990)
```

you obtain a new confidence interval of (868,950), which is shorter than before, but nothing else changes in the output from `t.test`.

Wilcoxon Signed Rank Test p-Values

To perform the Wilcoxon signed rank nonparametric test, use the function `wilcox.test`. As with `t.test`, the test is completely determined by the confidence level, the hypothesized mean μ_0, and the hypothesis to be tested. These options are specified for `wilcox.test` exactly as for `t.test`.

For example, to test the hypothesis that μ = 990 (plus 299,000), use `wilcox.test` as follows:

```
> wilcox.test(mich,mu=990)

Wilcoxon signed-rank test

data:  mich
signed-rank normal statistic with correction Z = -3.0715,
 p-value = 0.0021
alternative hypothesis: true mu is not equal to 990
Warning messages:
  cannot compute exact p-value with ties in:
  wil.sign.rank(dff, alternative, exact, correct)
```

The *p*-value of .0021 compares with the *t*-test *p*-value of .0027 for testing the same null hypothesis with a two-sided alternative.

Michelson's data have several tied values. Because exact *p*-values cannot be computed if there are tied values (or if the null hypothesis mean is equal to one of the data values), a normal approximation is used and the associated *Z*-statistic value is reported.

3.3 TWO SAMPLES: DISTRIBUTION SHAPES, LOCATIONS, AND SCALES

Suppose you are a nutritionist interested in the relative merits of two diets, one featuring high protein, the other low protein. Do the two diets lead to differences in mean weight gain? Consider the data in table 3.1, which shows the weight gains (in grams) for two lots of female rats, under the two diets. The first lot, consisting of 12 rats, was given the high protein diet, and the second lot, consisting of 7 rats, was given the low protein diet. These data appear in section 6.9 of Snedecor and Cochran (1980).

Table 3.1: *Weight gain data.*

High Protein	Low Protein
134	70
146	118
104	101
119	85
124	107
161	132
107	94
83	
113	
129	
97	
123	

The high protein and low protein samples are presumed to have mean-value location parameters μ_H and μ_L, and standard deviation scale parameters σ_H and σ_L, respectively. While you are primarily interested in whether there is any difference in the μ's, you may also be interested in whether or not the two diets result in different variabilities, as measured by the standard deviations (or their squared values, the variances). This section shows you how to use S-PLUS functions to answer such questions.

Setting Up the Data

In the two sample case, each sample forms a set of data. Thus, you begin by creating two data vectors, say `gain.high` and `gain.low`, containing, respectively, the first and second column of data from the table 3.1:

```
> gain.high <- scan()
1: 134 146 104 119 124 161 107 83 113 129 97 123
13:
> gain.low <- scan()
1: 70 118 101 85 107 132 94
8:
```

Exploratory Data Analysis

For each sample, make a set of EDA plots, consisting of a histogram, a boxplot, a density plot and a normal qq-plot, all displayed in a two-by-two plot layout, using the `eda.shape` function defined in the section Exploratory Data Analysis on page 44:

```
> eda.shape(gain.high)
> eda.shape(gain.low)
```

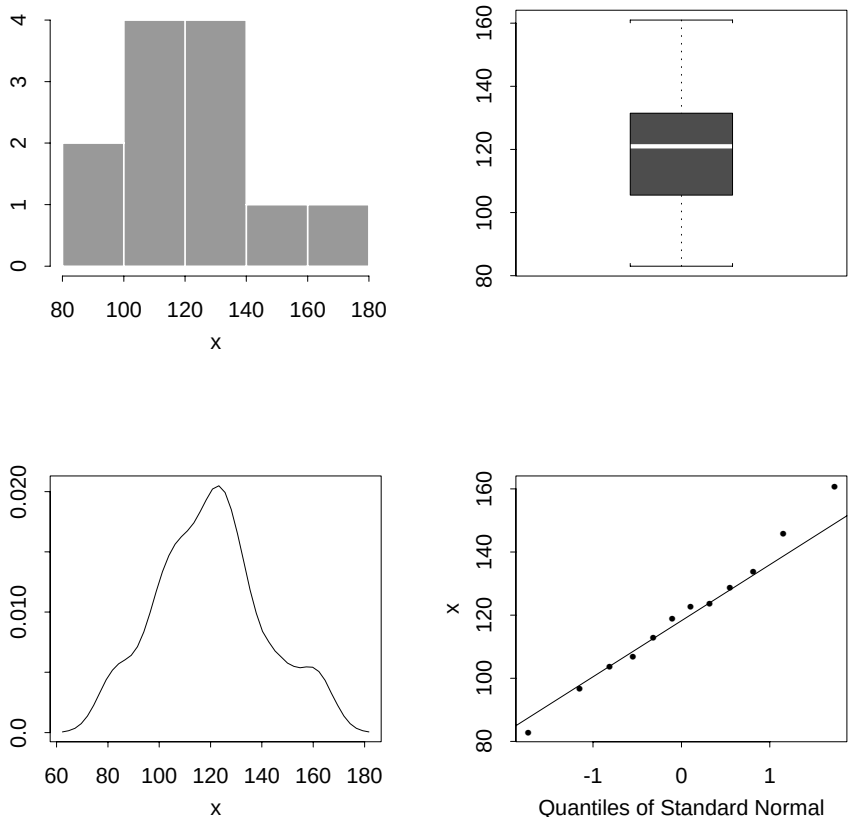

Figure 3.4: *EDA plots for high-protein group.*

The resulting plots for the high-protein group are shown in figure 3.4. They indicate that the data come from a nearly normal distribution, and there is no indication of outliers. The plots for the low-protein group, which we do not show, support the same conclusions.

Since the data were not collected in any specific time order, you need not make any exploratory time series plots to check for serial correlation.

Statistical Inference

Is the mean weight gain the same for the two groups of rats? Specifically, does the high-protein group show a higher average weight gain? From our exploratory data analysis, we have good reason to believe that Student's *t*-test will provide a valid test of our hypotheses. As in the one-sample case, you can get confidence intervals and hypothesis test *p*-values for the difference μ_1-μ_2 between the two mean-value location parameters μ_1 and μ_2 using the functions `t.test` and `wilcox.test`.

As before, each test is specified by a confidence level, a hypothesized μ_0 (which now refers to the *difference* of the two sample means), and the hypothesis to be tested. However, because of the possibility that the two samples may be from different distributions, you may also specify whether the two samples have equal variances.

You define the test to be performed using the following arguments to `t.test`:

- `conf.level=` specifies the confidence level of the confidence interval. Usual values are `0.90`, `0.95`, or `0.99`. The default is `0.95`.
- `mu=` specifies the null hypothesis value μ_0 of $\mu_{diff} = \mu_H - \mu_L$. The default is μ_0=0.
- `alternative=` specifies the hypothesis being tested. There are three options:
  - `"two.sided"` tests the hypothesis that the difference of means is not equal to μ_0. This is the default alternative.
  - `"greater"` tests the hypothesis that the difference of means is greater than μ_0.
  - `"less"` tests the hypothesis that the difference of means is less than μ_0.
- `var.equal=` specifies whether equal variances are assumed for the two samples. The default is `var.equal=TRUE`.

To determine the correct setting for the option `var.equal`, you can either use informal inspection of the EDA boxplots or use the function `var.test` for a more formal test. If the heights of the boxes in the two boxplots are approximately the same, then so are the variances of the two outlier-free samples. The `var.test` function performs the *F* test for variance equality on the vectors representing the two samples. For the weight gain data:

```
> var.test(gain.high,gain.low)
         F test for variance equality
data:  gain.high and gain.low
F = 1.0755, num df = 11, denom df = 6, p-value = 0.9788
alternative hypothesis: true ratio of variances is not
 equal to 1
95 percent confidence interval:
 0.1988088 4.1737320
sample estimates:
 variance of x variance of y
      457.4545      425.3333
```

The evidence supports the assumption that the variances are the same, so `var.equal=T` is a valid choice.

We are interested in two alternative hypotheses: the two-sided alternative that $\mu_H - \mu_L = 0$ and the one-sided alternative that $\mu_H - \mu_L > 0$. To test these, we run the standard two-sample *t*-test twice, once with the default two-sided alternative and a second time with the one-sided alternative `alt="g"`.

You get both a confidence interval for $\mu_H - \mu_L$, and a two-sided test of the null hypothesis that $\mu_H - \mu_L = 0$, by the following simple use of `t.test`:

```
> t.test(gain.high,gain.low)
         Standard Two-Sample t-Test
data: gain.high and gain.low
t = 1.8914, df = 17, p-value = 0.0757
alternative hypothesis: true difference in means is
 not equal to 0
95 percent confidence interval:
  -2.198905 40.198905
sample estimates:
 mean of x mean of y
       120       101
```

The *p*-value is .0757, so the null hypothesis is rejected at the .10 level, but not at the .05 level. The confidence interval is (-2.2, 40.2).

To test the one-sided alternative that $\mu_H - \mu_L > 0$, use `t.test` again with the argument `alternative="greater"` (abbreviated below for ease of typing):

```
t.test(gain.high,gain.low,alt="g")

        Standard Two-Sample t-Test

data:  gain.high and gain.low
t = 1.8914, df = 17, p-value = 0.0379
alternative hypothesis: true difference in means
 is greater than 0
95 percent confidence interval:
 1.521055       NA
sample estimates:
mean of x mean of y
      120       101
```

In this case, the *p*-value is just half of the *p*-value for the two-sided alternative. This relationship between the *p*-values of the one-sided and two-sided alternatives holds in general. You also see that when you use the `alt="g"` argument, you get a lower confidence bound. This is the natural one-sided confidence interval corresponding to the "greater than" alternative.

Hypothesis Test p-Values using `wilcox.test`

To get a two-sided hypothesis test *p*-value for the "two-sided" alternative, based on the Wilcoxon rank sum test statistic, use `wilcox.test`, which takes the same arguments as `t.test`:

```
> wilcox.test(gain.high,gain.low)

            Wilcoxon rank-sum test

data: gain.high and gain.low
rank-sum normal statistic with correction Z = 1.691,
 p-value = 0.0908
alternative hypothesis: true mu is not equal to 0

Warning messages:
  cannot compute exact p-value with ties
```

The above *p*-value of .0908, based on the normal approximation (used because of ties in the data), is rather close to the *t*-statistic *p*-value of .0757.

3.4 TWO PAIRED SAMPLES

Often two samples of data are collected in the context of a *comparative* study. A comparative study is designed to determine the *difference* between effects, rather than the individual effects. For example, consider the data in table 3.2, which gives values of wear for two kinds of shoe sole material, A and B, along with the differences in values.

Table 3.2: *Comparing shoe sole material*

boy	wear.A	wear.B	wear.A-wear.B
1	14.0(R)	13.2(L)	0.8
2	8.8(R)	8.2(L)	0.6
3	11.2(L)	10.9(R)	0.3
4	14,2(R)	14.3(L)	-0.1
5	11.8(L)	10.7(R)	1.1
6	6.4(R)	6.6(L)	-0.2
7	9.8(R)	9.5(L)	0.3
8	11.3(R)	10.8(L)	0.5
9	9.3(L)	8.8(R)	0.5
10	13.6(R)	13.3(L)	0.3

In the table, (L) indicates the material was used on the left sole; (R), that it was used on the right sole.

The experiment leading to this data, described in Box, Hunter, and Hunter (1978), was carried out by taking 10 pairs of shoes and putting a sole of material A on one shoe and a sole of material B on the other shoe in each pair. Which material type went on each shoe was determined by randomizing, with equal probability that material A was on the right shoe or left shoe. A group of 10 boys then wore the shoes for a period of time, after which the amount of wear was measured. The problem is to determine whether shoe material A or B is longer wearing.

You could treat this problem as a two sample location problem and use either `t.test` or `wilcox.test`, as described in the section Two Samples: Distribution Shapes, Locations, and Scales on page 54, to test for a difference in the means of wear for material A and material B. However, you will not be very successful with this approach because there is considerable variability in wear of *both* materials types A and B from individual to individual, and this variability tends to mask the difference in wear of material A and B when you use an ordinary two sample test.

However, the above experiment uses *paired* comparisons. Each boy wears one shoe with material A and one shoe with material B. In general, *pairing* involves selecting similar individuals or things. One often uses *self-pairing* as in the above experiment, in which two procedures, often called *treatments*, are applied to the same individual (either simultaneously or at two closely spaced time intervals) or to similar material. The goal of pairing is to make a comparison more sensitive by measuring experimental outcome differences on each pair, and combining the differences to form a statistical test or confidence interval. When you have paired data, you use `t.test` and `wilcox.test` with the optional argument `paired=T`.

The use of paired versions of `t.test` and `wilcox.test` leads to improved sensitivity over the usual versions when the variability of differences is smaller than the variability of each sample; for example, when the variability of differences of material wear between materials A and B is smaller than the variability in wear of material A and material B.

Setting Up the Data

In paired comparisons you start with two samples of data, just as in the case of ordinary two sample comparisons. You begin by creating two data vectors, `wear.A` and `wear.B`, containing the first and second columns of table 3.2:

```
> wear.A <- scan()
1: 14.0 8.8 11.2 14.2 11.8 6.4 9.8 11.3 9.3 13.6
10:
> wear.B <- scan()
1: 13.2 8.2 10.9 14.3 10.7 6.6 9.5 10.8 8.8 13.3
10:
```

Exploratory Data Analysis

You can carry out exploratory data analysis on each of the two paired samples $x_1, \ldots, x_n$ and $y_1, \ldots, y_n$, as for an ordinary two sample problem, as described in the section Exploratory Data Analysis on page 55. However, since your analysis is based on differences, it is appropriate to carry out EDA based on a single sample of differences $d_i = x_i - y_i$, i=1, …, n.

In the shoe material wear experiment, you use `eda.shape` on the difference `wear.A-wear.B`:

```
> eda.shape(wear.A - wear.B)
```

The results are displayed in figure 3.5. The histogram and density indicate some deviation from normality that is difficult to judge because of the small sample size.

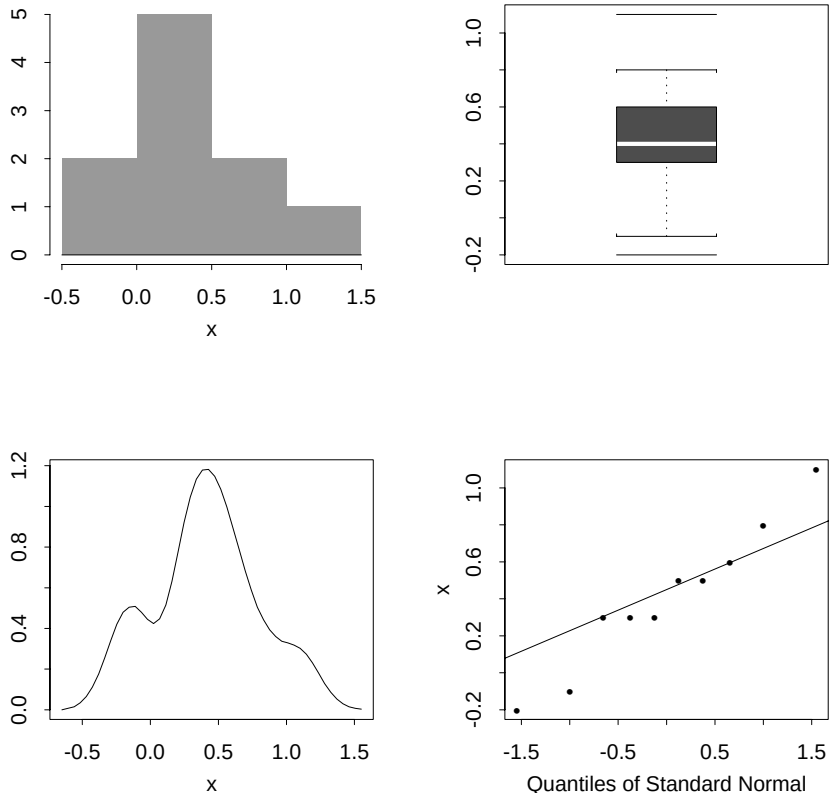

Figure 3.5: *EDA plots for differences in shoe sole material wear.*

You might also want to make a scatter plot of `wear.B` versus `wear.A`, using `plot(wear.A,wear.B)`, as a visual check on correlation between the two variables. Strong correlation is an indication that within-sample variability is considerably larger than the difference in means, and hence that the use of pairing will lead to greater test sensitivity. To obtain the scatter plot of figure 3.6, use the following S-PLUS expression:

```
> plot(wear.A,wear.B)
```

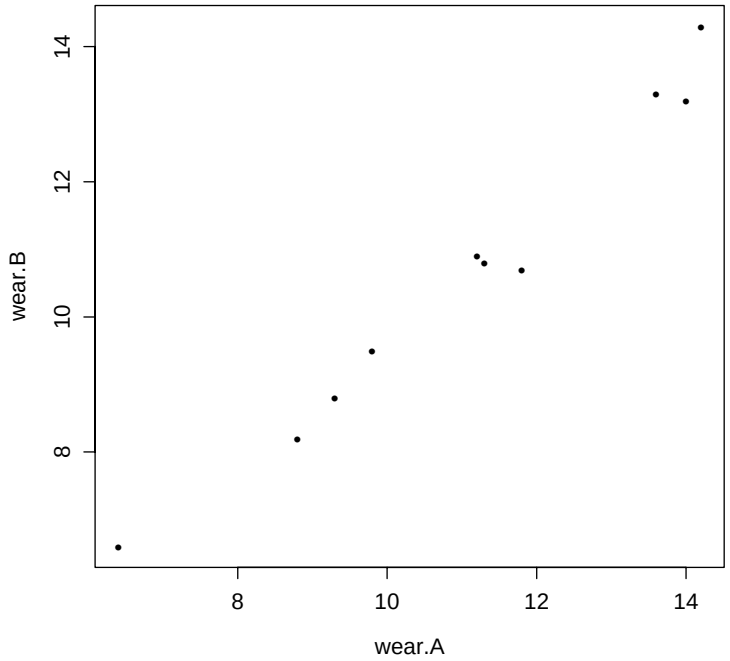

Figure 3.6: *Scatter plot of* `wear.A` *versus* `wear.B`.

Statistical Inference

To perform a paired *t*-test on the shoe material wear data, with the default two-sided alternative use `t.test` with the `paired` argument, as follows:

```
> t.test(wear.A,wear.B,paired=T)

        Paired t-Test

data: wear.A and wear.B
t = 3.3489, df = 9, p-value = 0.0085
alternative hypothesis: true mean of differences is not
 equal to 0
95 percent confidence interval:
 0.1330461 0.6869539
sample estimates:
 mean of x - y
          0.41
```

The *p*-value of .0085 is highly significant for testing the difference in mean wear of materials A and B. You also get the 95% confidence interval (.13,.67) for the difference in mean values. You can control the type of alternative hypothesis with the `alt=` optional argument, and you can control the confidence level with the `conf.level=` optional argument, as usual. To perform a paired Wilcoxon test (often called the *Wilcoxon signed rank test*) on the shoe material data, with the default two-sided alternative use `wilcox.test` with the `paired` argument, as follows:

```
> wilcox.test(wear.A,wear.B,paired=T)

Wilcoxon signed-rank test

data:  wear.A and wear.B
signed-rank normal statistic with correction Z = 2.4495,
 p-value = 0.0143
alternative hypothesis: true mu is not equal to 0

Warning messages:
  cannot compute exact p-value with ties in:
  wil.sign.rank(dff, alternative, exact, correct)
```

The *p*-value of .0143 is highly significant for testing the null hypothesis of equal centers of symmetry for the distributions of `wear.A` and `wear.B`. You can control the type of alternative hypothesis by using the optional argument `alt=` as usual.

3.5 CORRELATION

What effect, if any, do housing starts have on the demand for residential telephone service? If there is some useful association, or *correlation*, between the two, you may be able to use housing start data as a predictor of growth in demand for residential phone lines. Consider the data displayed in table 3.3 (in coded form), which relates to residence telephones in one area of New York City.

The first column of data, labeled “Diff. HS", shows annual first differences in new housing starts over a period of fourteen years. The first differences are calculated as the number of new housing starts in a given year, minus the number of new housing starts in the previous year. The second column of data, labeled “Phone Increase", shows the annual increase in the number of “main" residence telephone services (excluding extensions), for the same fourteen-year period.

The general setup for analyzing the association between two samples of data

Table 3.3: *The phone increase data.*

Diff. HS	Phone Increase
.06	1.135
.13	1.075
.14	1.496
-.07	1.611
-.05	1.654
-.31	1.573
.12	1.689
.23	1.850
-.05	1.587
-.03	1.493
.62	2.049
.29	1.942
-.32	1.482
.71	1.382

such as those above is as follows. You have two samples of observations, of equal sizes n, of the random variables $X_1, X_2, \ldots, X_n$ and $Y_1, Y_2, \ldots, Y_n$. Let's assume that each of the two-dimensional vector random variables (X_i, Y_i), i=1, 2, …, n, have the same joint distribution.

The most important, and commonly used measure of association between two such random variables is the (population) *correlation coefficient* parameter ρ, defined as

$$\rho = \frac{E(x-\mu_1)(Y-\mu_2)}{\sigma_1\sigma_2},$$

where μ_1, μ_2 and σ_1, σ_2 are the means and standard deviations, respectively, of the random variables X and Y. The E appearing in the numerator denotes the statistical *expected value*, or *expectation* operator, and the quantity

$E(X - \mu_1)(Y - \mu_2)$ is the *covariance* between the random variables X and Y. The value of ρ is always between 1 and -1.

Your main goal is to use the two samples of observed data to determine the value of the correlation coefficient ρ. In the process you want to do sufficient graphical EDA to feel confident that your determination of ρ is reliable.

Setting Up the Data

The data form two distinct data sets, so we create two vectors with the suggestive names `diff.hs` and `phone.gain`:

```
> diff.hs <- scan()
1: .06 .13 .14 -.07 -.05 -.31 .12
8: .23 -.05 -.03 .62 .29 -.32 -.71
15:
> phone.gain <- scan()
1: 1.135 1.075 1.496 1.611 1.654 1.573 1.689
8: 1.850 1.587 1.493 2.049 1.943 1.482 1.382
15:
```

Exploratory Data Analysis

If two variables are strongly correlated, that correlation may appear in a scatter plot of one variable against the other. For example, plot `phone.gain` versus `diff.hs` using the following command:

```
> plot(diff.hs, phone.gain)
```

The results are shown in figure 3.7. The plot reveals a strong positive correlation, except for two obvious outliers. To identify the observation numbers associated with the outliers in the scatter plot, along with that of a third suspicious point, we used `identify` as follows:

```
> identify(diff.hs, phone.gain, n=3)
```

See the on-line help for a complete discussion of `identify`.

The obvious outliers occur at the first and second observations. In addition, the suspicious point (labeled “3” in the scatter plot) occurs at the third observation time.

Since you have now identified the observation times of the outliers, you can gain further insight by making a time series plot of each series:

```
> plot(diff.hs,type="b")
> plot(phone.gain,type="b")
```

You should also make an autocorrelation plot for each series:

```
> acf(diff.hs)
> acf(phone.gain)
```

The results are shown in figure 3.8. Except for the first three observations of the two series `phone.gain` and `diff.hs`, there is a strong similarity of

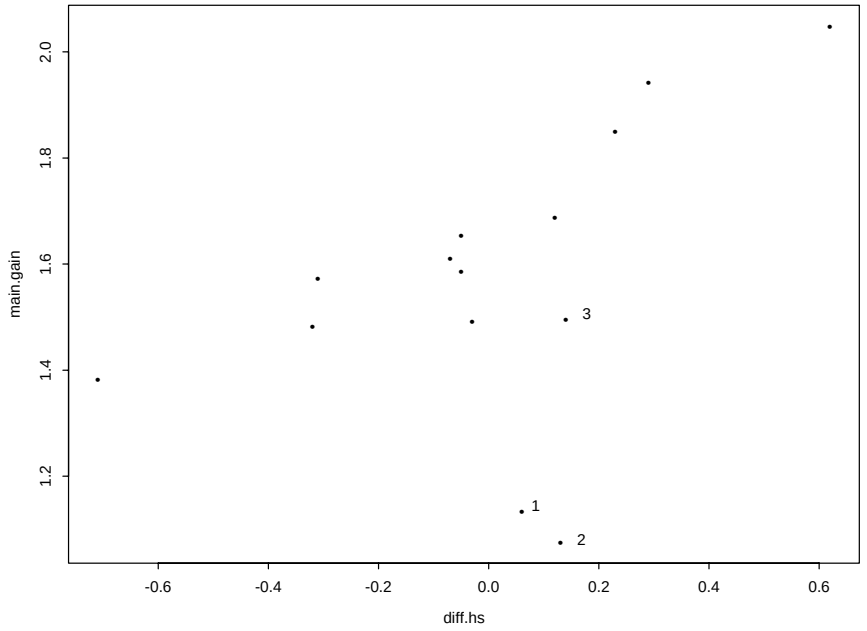

Figure 3.7: *Scatter plot of* `phone.gain` *versus* `diff.hs`.

shape exhibited in the two time series plots. This accounts for the strong positive correlation between the two variables `diff.hs` and `phone.gain` shown in figure 3.7. The dissimilar behavior of the two time series plots for the first three observations produces the two obvious outliers, and the suspicious point, in the scatter plot of `phone.gain` versus `diff.hs`.

The ACF plots show little evidence of serial correlation within each of the individual series.

Statistical Inference

From your exploratory data analysis, two types of questions present themselves for more formal analysis. If the evidence for correlation is inconclusive, you may want to test whether there is correlation between the two variables of interest by testing the null hypothesis that $\rho = 0$. On the other hand, if your EDA convinces you that correlation exists, you might prefer a point estimate $\hat{\rho}$ of the correlation coefficient ρ, or a confidence interval for ρ.

Hypothesis Test p-Values

You can get *p*-values for the null hypothesis that $\rho = 0$ by using the function `cor.test`. To perform this test, you specify the alternative hypothesis to be tested and the test method to use, as follows:

- `alternative=` specifies the alternative hypothesis to be tested.

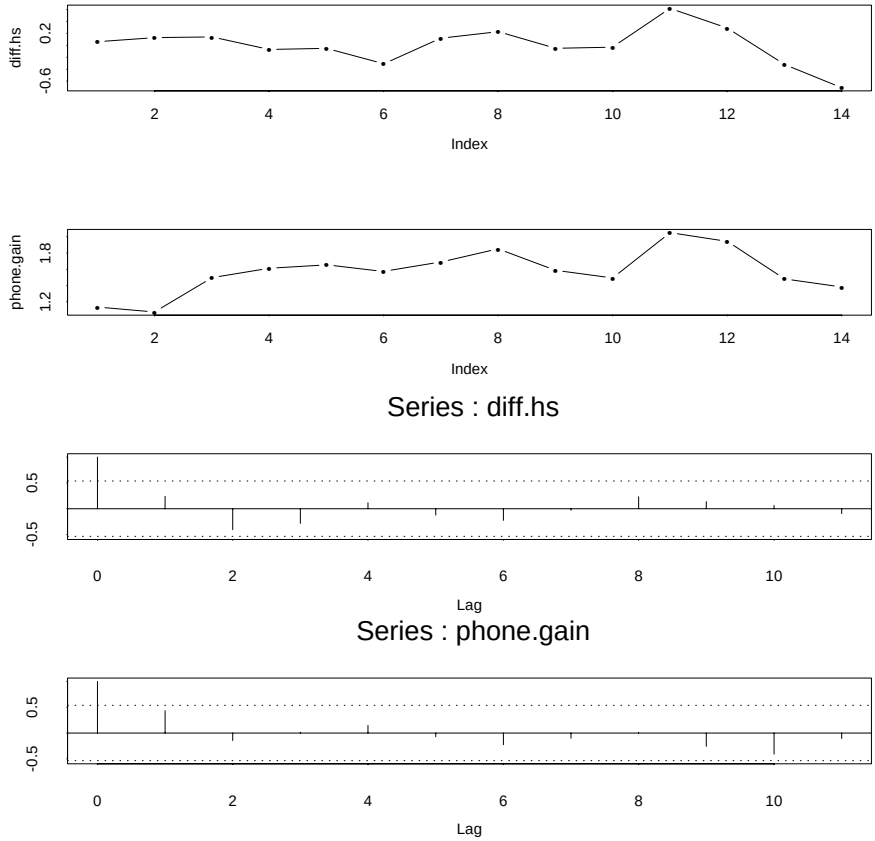

Figure 3.8: *Time series and ACF plots of phone increase data* .

There are three options:

- `"two.sided"` (the default alternative) tests the alternative hypothesis that $\rho \neq 0$.
- `"greater"` tests the alternative hypothesis that $\rho > 0$.
- `"less"` tests the alternative hypothesis that $\rho < 0$.

You can also use the abbreviated forms `alt="g"` and `alt="l"`.

- `method=` specifies which of the following methods is used:
  - `"pearson"` (the default) uses the standard Pearson sample correlation coefficient.
  - `"kendall"` uses the rank-based Kendall's τ measure of correlation.
  - `"spearman"` uses the rank-based Spearman's ρ measure of correlation.

  You can abbreviate these methods by using enough of the character string to determine a unique match; here `"p"`, `"k"`, and `"s"` work.

Because both Kendall's τ and Spearman's ρ methods are based on ranks, they are not so sensitive to outliers and non-normality as the standard Pearson estimate.

Here is a simple use of `cor.test` to test the alternative hypothesis that there is a positive correlation in the phone gain data. We use the default choice of the classical Pearson estimate with the one-sided alternative `alt="g"`:

```
> cor.test(diff.hs,phone.gain,alt="g")

        Pearson product-moment correlation

data:  diff.hs and phone.gain
t = 1.9155, df = 12, p-value = 0.0398
alternative hypothesis: true coef is greater than 0
sample estimates:
      cor
 0.4839002
```

You get a normal theory *t*-statistic having the modest value of 1.9155, and a *p*-value of .0398. The estimate of ρ is .48, to two decimal places. There are 14 bivariate observations, and since the mean is estimated for each sample under the null hypothesis that $\rho > 0$, the number of degrees of freedom (df) is 12.

Since your EDA plots reveal two obvious bivariate outliers in the phone gain data, the non-parametric alternatives, either Kendall's τ or Spearman's ρ, are preferable in determining *p*-values for this case. Using Kendall's method, we obtain the following results:

```
> cor.test(diff.hs,phone.gain,alt="g",method="k")
```

```
        Kendall's rank correlation tau

data: diff.hs and phone.gain
normal-z = 2.0256, p-value = 0.0214
alternative hypothesis: true tau is greater than 0
sample estimates:
      tau
 0.4065934
```

The *p*-value obtained from Kendall's method is smaller than that obtained from the Pearson method. The null hypothesis is rejected at a level of 0.05. Spearman's ρ, by contrast, yields a *p*-value similar to that of the standard Pearson method.

Warning

The values returned for "tau" and "rho" (.407 and .504, respectively, for the phone gain data) do not provide unbiased estimates of the true correlation ρ. Transformations of "tau" and "rho" are required to obtain unbiased estimates of ρ.

Point Estimates and Confidence Intervals for ρ

You may want an estimate $\hat{\rho}$ of ρ, or a confidence interval for ρ. The function `cor.test` gives you the classical sample correlation coefficient estimate *r* of ρ, when you use the default Pearson's method. However, `cor.test` does not provide you with a robust estimate of ρ, (since neither Kendall's τ or Spearman's ρ provide an *unbiased* estimate of ρ). Furthermore, `cor.test` does not provide any kind of confidence interval for ρ.

To obtain a robust point estimate of ρ, use the function `cor` with a non-zero value for the optional argument `trim=`. You should specify a fraction α between 0 and .5 for the value of this argument. This results in a robust estimate which consists of the ordinary sample correlation coefficient based on the fraction (1-α) of the data remaining after "trimming" away a fraction α. In most cases, set `trim=.2`. If you think your data contain more than 20% outliers, you should use a larger fraction in place of `.2`. The default value is `trim=0`, which yields the standard Pearson sample correlation coefficient.

Applying `cor` to the phone gain data, you get:

```
> cor(diff.hs,phone.gain,trim=.2)
[1] 0.715215
```

Comparing this robust estimate to our earlier value for ρ obtained using `cor.test`, we see the robust estimate yields a larger estimate of ρ. This is what you expect, since the two outliers cause the standard sample correlation coefficient to have a value smaller than that of the "bulk" of the data.

To obtain a confidence interval for ρ, we'll use the following procedure (as in Snedecor and Cochran (1980)). First, transform ρ using Fisher's "z-transform," which consists of taking the inverse hyperbolic tangent transform $z = atanh(\rho)$. Then, construct a confidence interval for the correspondingly transformed true correlation coefficient $\bar{\rho} = atanh(\rho)$. Finally, transform this interval back to the original scale by transforming each endpoint of this interval with the hyperbolic tangent transformation *tanh*.

To implement the procedure just described as an S-PLUS function, create the function `cor.confint` as follows:

```
> cor.confint <- function(x, y, conf.level = .95, trim = 0)
+ {
+          z <- atanh(cor(x, y, trim))
+          b <- qnorm((1 - conf.level)/2)/(length(x) - 3)^.5
+          ci.z <- c(z - b, z + b)
+          conf.int <- tanh(ci.z)
+          conf.int
+ }
```

You can now use your new function `cor.confint` on the phone gain data:

```
> cor.confint(diff.hs,phone.gain)
[1]  0.80722631 -0.06280418
> cor.confint(diff.hs,phone.gain,trim=.2)
[1] 0.9028239 0.2962303
```

When you use the optional argument `trim=.2`, you are basing the confidence interval on a robust estimate of ρ, and consequently you get a robust confidence interval. Since the robust estimate has the value .72, which is larger than the standard (non-robust) estimate value of .48, you should be reassured to get an interval which is shifted upward, and is also shorter, than the non-robust interval you got by using `cor.confint` with the default option `trim=0`.

3.6 REFERENCES

Bishop, Y. M. M. and Fienberg, S. J. and Holland, P. W. (1980). *Discrete Multivariate Analysis: Theory and Practice*. The MIT Press, Cambridge, MA.

Box, G.E.P. and Hunter, W.G. and Hunter, J.S. (1978). *Statistics for Experimenters: An Introduction to Design, Data Analysis and Model Building*. John Wiley, New York.

Conover, W. J. (1980). *Practical Nonparametric Statistics, 2nd edition*. John Wiley, New York.

Heidelberger, P. and Welch, P. D. (1981). A Spectral Method for Confidence Interval Generation and Run-length Control in Simulations. *Communications of the ACM*, 24:233–245.

Hogg, R. V. and Craig, A. T. (1970). *Introduction to Mathematical Statistics*, 3rd edition. Macmillan, Toronto, Canada.

Mood, A. M. and Graybill, F. A. and Boes, D. C. (1974). *Introduction to the Theory of Statistics*, 3rd edition. McGraw-Hill, New York.

Snedecor, G. W. and Cochran, W. G. (1980). *Statistical Methods*, 7th edition. Iowa State University Press, Ames, IA.

GOODNESS OF FIT TESTS

4

Goodness of fit tests are a formal method for assessing the evidence in assuming a certain underlying distribution.

GOODNESS OF FIT TESTS

Most S-PLUS functions for hypothesis testing assume a certain distributional form—often normal—and then use data to make conclusions about certain *parameters* of the distribution—often the mean or variance. In chapter 3, Statistical Inference for One and Two Sample Problems, we describe EDA techniques to informally test the assumptions of these procedures. Goodness of fit (GOF) tests are another, more formal, tool to assess the evidence for assuming a certain distribution.

There are two types of GOF problems—corresponding to the number of samples—which ask the following questions:

1. *One sample.* Does the sample arise from a hypothesized distribution?
2. *Two sample.* Do two independent samples arise from the same distribution?

S-PLUS implements the two best known GOF tests:

- Chi-square, in the `chisq.gof` function.
- Kolmogorov-Smirnov, in the `ks.gof` function.

The chi-square test applies only in the one-sample case; Kolmogorov-Smirnov can be used in both the one-sample and the two-sample cases. This chapter describes both tests, together with a graphical function, `cdf.compare`, that can be used in both the one-sample and two-sample cases as an exploratory tool for evaluating goodness of fit.

4.1 CUMULATIVE DISTRIBUTION FUNCTIONS

For a random variable X, a cumulative distribution function (cdf), $F(x) = P[X \leq x]$, assigns a measure (between 0 and 1) of the probability that $X \leq x$. If $X_1, \ldots, X_n$ form a random sample from a continuous distribution, with observed values $x_1, \ldots, x_n$, an *empirical distribution function* F_n can be defined for all x, $-\infty < x < \infty$, so that $F_n(x)$ is the proportion of observed values less than or equal to x. The empirical distribution function estimates the unknown cdf. To decide whether two samples arise from the same unknown distribution, a natural procedure is to compare their empirical distribution functions. Likewise, for one sample, you can compare its empirical distribution function with a hypothesized cdf.

A graphical comparison of either one empirical distribution function with a known cdf, or of two empirical distribution functions, can be obtained easily in S-PLUS using the function `cdf.compare`.

For example, consider the plot shown in figure 4.1. In this example, the

empirical distribution function and a hypothetical cdf are quite close. This plot is produced using the `cdf.compare` function as follows:

```
> z <- rnorm(100)
> cdf.compare(z, distribution="normal")
```

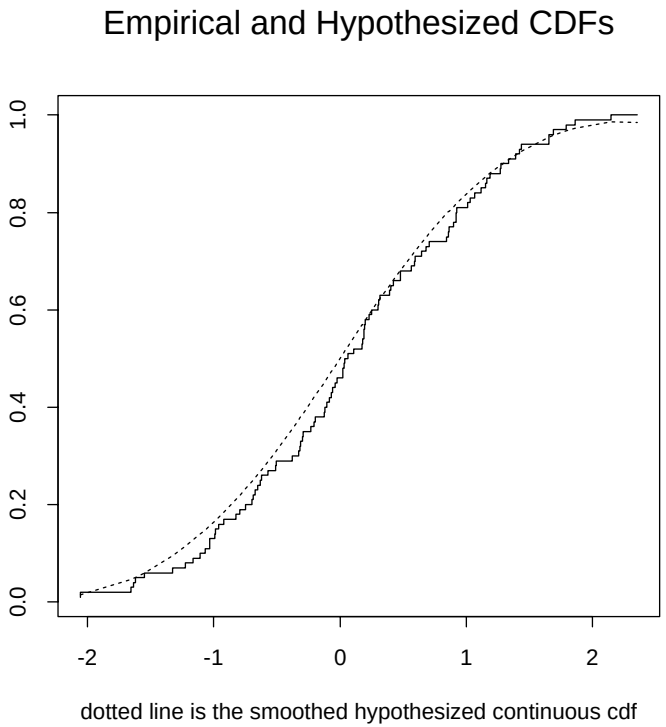

Figure 4.1: *The empirical distribution function of a sample of size 100 generated from a N(0,1) distribution. The dotted line is the smoothed theoretical N(0,1) distribution evaluated at the sample points.*

You may also compare distributions using quantile-quantile plots (Q-Q plots) generated by either of the following functions:

- `qqnorm` to compare one sample with a normal distribution
- `qqplot` to compare two samples

For our normal sample `z`, `qqnorm(z)` produces the plot shown in figure 4.2. Departures from linearity show how the sample differs from the normal, or how the two sample distributions differ. To compare samples with distributions other than the normal, you may produce Q-Q plots using the function `ppoints`. For more details, see the chapter Traditional Graphics in

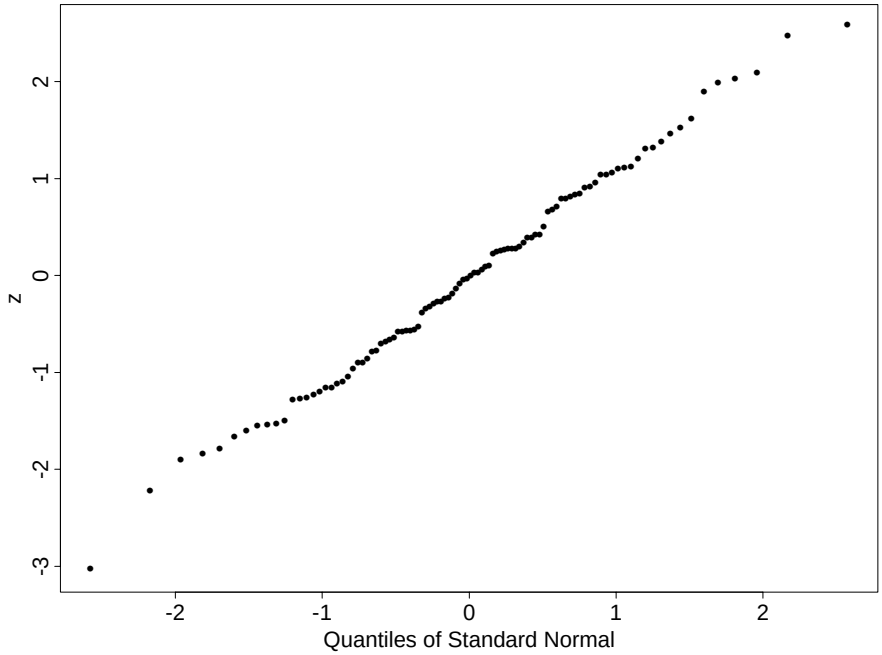

Figure 4.2: *A* `qqnorm` *plot of a sample from a normal distribution.*

the S-PLUS Programmer's Guide.

In many cases, the graphical conclusions drawn from `cdf.compare` or the Q-Q plots make more formal tests such as the chi square or Kolmogorov-Smirnov unnecessary. For example, consider the two empirical distributions compared in figure 4.3—they clearly have different distribution functions:

```
> x <- rnorm(30)
> y <- runif(30)
> cdf.compare(x,y)
```

4.2 THE CHI-SQUARE TEST OF GOODNESS OF FIT

The chi-square test, the oldest and best known goodness-of-fit test, is a one-sample test that examines the frequency distribution of n observations grouped into k classes. The observed counts c_i in each class are compared to the expected counts C_i from the hypothesized distribution. The test statistic, due to Pearson, is

$$\hat{\chi}^2 = \sum_{i=1}^{k} (c_i - C_i)^2 / C_i \quad .$$

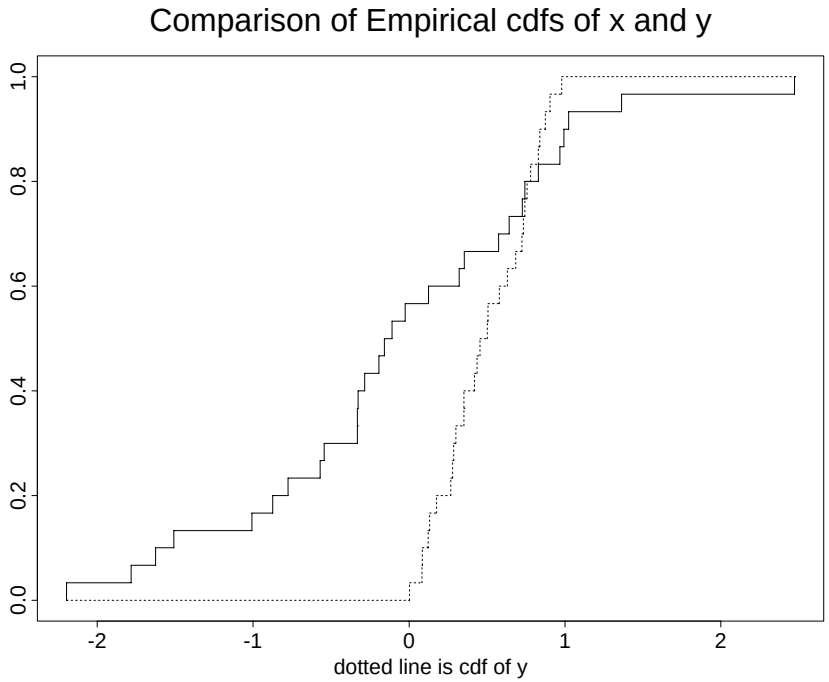

Figure 4.3: *Two clearly different empirical distribution functions.*

Under the null hypothesis that the sample comes from the hypothesized distribution, it has a χ^2 distribution with k-1 degrees of freedom. For any significance level α, reject the null hypothesis if $\hat{\chi}^2$ is greater than the critical value ν for which $P(\chi^2 > \nu) = \alpha$.

You perform the chi-square goodness of fit test with the `chisq.gof` function. In the simplest case, you specify a test vector and a hypothesized distribution:

```
> z.chisq <- chisq.gof(z, distribution="normal")
> z.chisq
        Chi-square Goodness of Fit Test

data: z

Chi-square = 8.94, df = 12,
p-value = 0.708
alternative hypothesis:
True cdf does not equal the normal Distn.
     for at least one sample point.
```

Since we created `z` as a random sample from a normal distribution, it is not surprising that we cannot reject the null hypothesis. If we hypothesize a different distribution, we see that the chi-square correctly rejects the hypothesis:

```
> chisq.gof(z, distribution="exponential")

        Chi-square Goodness of Fit Test

data: z

Chi-square = 324.84, df = 12,
p-value = 0
alternative hypothesis:
True cdf does not equal the exponential Distn.
     for at least one sample point.
```

The allowable values for the `distribution` argument are the following strings (the default is `"normal"`):

```
"beta"            "binomial"     "cauchy"    "chisquare"
"exponential"     "f"              "gamma"    "geometric"
"hypergeometric"  "lognormal"  "logistic"  "negbinomial"
"normal"          "poisson"          "t"      "uniform"
"weibull"         "wilcoxon"
```

When the sample being tested is from a continuous distribution, one factor affecting the outcome is the choice of partition for determining the grouping of the observations. This becomes particularly important when the expected count in one or more cells drops below 1, or the average expected cell count drops below five (Snedecor and Cochran (1980), p. 77). You can control the choice of partition using either the `n.classes` or `cut.points` argument to `chisq.gof`. By default, `chisq.gof` uses a default value for `n.classes` due to Moore (1986).

Use the `n.classes` argument to specify the number of equal-width classes:

```
> chisq.gof(z, n.classes=5)
```

Use the `cut.points` argument to specify the end points of the cells; the specified points should span the observed values:

```
> cuts.z <- c(floor(min(z))-1, -1,0,1, ceiling(max(z))+1)
> chisq.gof(z, cut.points=cuts.z)
```

Chi-square tests apply to any type of variable: continuous, discrete, or a combination of these. For large sample sizes ($n \geq 50$), if the hypothesized distribution is discrete, the chi-square is the *only* valid test. In addition, the

chi-square test easily adapts to the situation when parameters of a distribution are estimated. However, especially for continuous variables, information is lost by grouping the data.

When the hypothesized distribution is continuous, the *Kolmogorov-Smirnov* test is more likely to reject the null hypothesis when it should; it is *more powerful* than the chi-square test.

4.3 THE KOLMOGOROV-SMIRNOV TEST

Suppose F_1 and F_2 are two cdfs. In the one-sample situation, F_1 is the empirical distribution function, and F_2 is a hypothesized cdf. In the two-sample situation, F_1 and F_2 are both empirical distribution functions.

Possible hypotheses and alternatives concerning these cdfs are:

- Two-sided:

  H_0: $F_1(x) = F_2(x)$ for all x

  H_A: $F_1(x) \neq F_2(x)$ for at least one value of x.

- *One-sided ("less" alternative):*

  H_0: $F_1(x) \geq F_2(x)$ for all x

  H_A: $F_1(x) < F_2(x)$ for at least one value of x.

- *One-sided ("greater" alternative):*

  H_0: $F_1(x) \leq F_2(x)$ for all x

  H_A: $F_1(x) > F_2(x)$ for at least one value of x.

The *Kolmogorov-Smirnov* test is a method for testing the above hypotheses. Corresponding to each alternative hypothesis is a Kolmogorov-Smirnov test statistic, as follows:

- *Two-sided Test:* $T = sup_x \left| F_1(x) - F_2(x) \right|$
- *Less Alternative:* $T^- = sup_x \left| F_2(x) - F_1(x) \right|$
- *Greater Alternative:* $T^+ = sup_x \left| F_1(x) - F_2(x) \right|$

If the test statistic is greater than some critical value, the null hypothesis is rejected.

To perform a KS test, use the function `ks.gof`. By default, the one-sample `ks.gof` test compares the sample `x` to a normal with mean `mean(x)` and standard deviation `sqrt(var(x))`:

```
> ks.gof(z)

 One sample Kolmogorov-Smirnov Test of Composite Normality
data: z

ks = 0.0457, p-value = 0.5
alternative hypothesis:
True cdf does not equal the normal Distn.
     for at least one sample point.

sample estimates:
   mean of x standard deviation of x
 -0.04593973                1.103777
Warning messages:
  The Dallal-Wilkinson approximation, used to calculate
     the p-value in testing composite normality,
     is most accurate for p-values <= 0.10 .
   The calculated p-value is 0.881
     and so is set to 0.5 . in: dall.wilk(test, nx)
```

In the one-sample case, `ks.gof` can test any of the three alternative hypotheses (`"two-sided"`, `"greater"`, `"less"`). In the two-sample case, `ks.gof` can test only the two-sided hypothesis. The default hypothesized distribution is the normal, as for the `chisq.gof` function. You can specify a different distribution using the `distribution` argument. Allowable values for the `distribution` argument are as follows:

```
"beta"            "binomial"     "cauchy"    "chisquare"
"exponential"     "f"              "gamma"    "geometric"
"hypergeometric"  "lognormal"  "logistic"  "negbinomial"
"normal"          "poisson"           "t"      "uniform"
"weibull"         "wilcoxon"
```

4.4 ONE SAMPLE TESTS

In a real situation, we do *not* know the true source of the data. Suppose, instead, that we think the underlying distribution is chi-square with 2 degrees of freedom. The KS test gives strong evidence against this assumption:

```
> ks.gof(z,alternative="greater",dist="chisquare",df=2)

        One-sample Kolmogorov-Smirnov Test;
                    hypothesized distribution = chisquare
```

```
data: z
ks = 0.4741, p-value = 0
alternative hypothesis: True cdf is greater than the
       chisquare Distn. for at least one sample point.
```

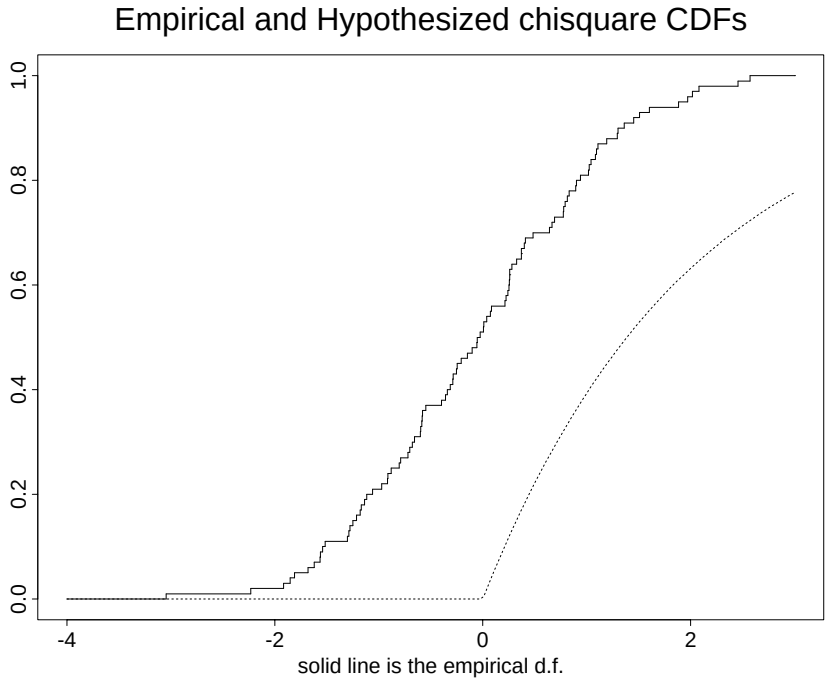

Figure 4.4: *Like the previous figure, except the dotted line shows a chi-square cdf with 2 degrees of freedom.*

Figure 4.4, created as follows, further shows that this assumption is not reasonable:

```
> cdf.compare(z, dist="chisquare", df=2)
```

The `chisq.gof` test gives further confirmation:

```
> chisq.gof(z,dist="chisquare",n.param.est=0,df=2)

        Chi-square Goodness of Fit Test
data: z
Chi-square = 282.98, df = 12, p-value = 0
alternative hypothesis: True cdf does not equal the
        chisquare Distn. for at least one sample point.
```

Note that `chisq.gof` tests only the two sided alternative.

Composite Tests for a Family of Distributions

When the parameters are *estimated* from the sample, rather than specified in advance, the tests described above are no longer adequate. Different tables of critical values are needed. In fact, for the KS test, the tables vary for different distributions, parameters estimated, methods of estimation, and sample sizes. The null hypothesis is now *composite*: rather than hypothesizing that the data have a distribution with specific parameters, we hypothesize only that the distribution belongs to the family of distributions with a certain form, such as normal. This family of distributions results from allowing all possible parameter values.

The two functions `ks.gof` and `chisq.gof` use different strategies to solve composite tests: `ks.gof` explicitly calculates the required parameters in two cases (described below), but otherwise forbids composite hypotheses, while `chisq.gof` requires the user to pass both the number of estimated parameters and the estimates themselves as arguments, then reduces the degrees of freedom for the chi-square by the number of estimated parameters.

The function `ks.gof` estimates parameters in the following two cases:

- Normal, with both mean and variance estimated.
- Exponential, with mean estimated.

As an example, we test whether we can reasonably assume that the Michelson data (see the section One Sample: Distribution Shape, Location, and Scale on page 49) arises from a normal distribution. We start with an exploratory plot using `cdf.compare` (figure 4.5), and then use `ks.gof` with estimated mean and variance:

```
> cdf.compare(mich, dist="normal", mean=mean(mich),
+                 sd=sqrt(var(mich)))
> ks.gof(mich,dist="normal")

One-sample Kolmogorov-Smirnov Test of Composite Normality

data: mich
ks = 0.1793, p-value = 0.0914
alternative hypothesis: True cdf does not equal
         the normal Distn. for at least one sample point.
```

For the function `chisq.gof`, if parameters are estimated, the degrees of freedom depend on the method of estimation. In practice, you may estimate the parameters from the original (that is, not grouped) data, and then set the argument `n.param.est` to the number of parameters estimated. The function then subtracts one degree of freedom for each parameter estimated.

In truth, if the parameters are estimated by maximum likelihood, the degrees of freedom are bounded between (m-1) and (m-1-k), where m is the number

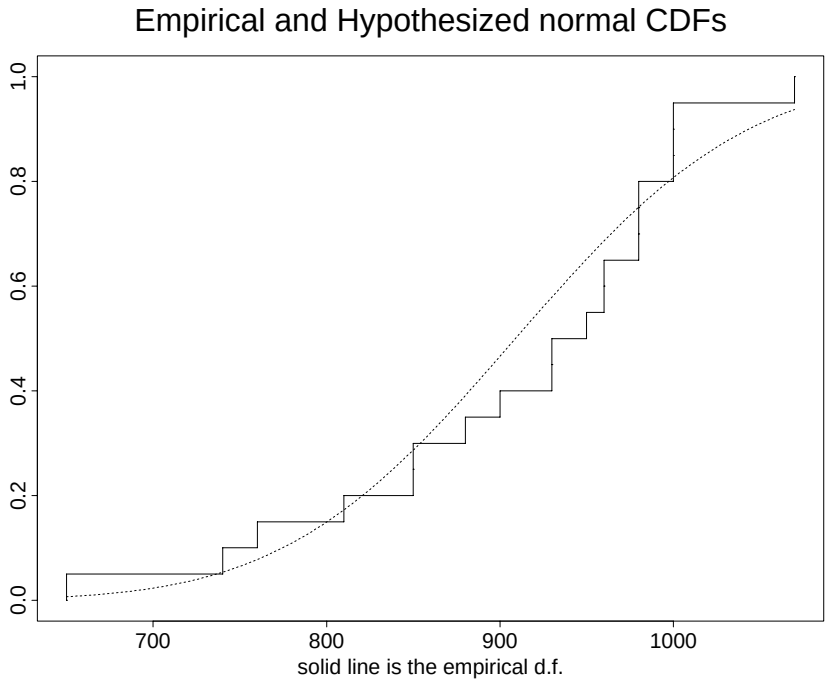

Figure 4.5: *Exploratory plot of cdf of* `mich` *data.*

of cells, and k is the number of parameters estimated. Therefore, especially when the sample size is small, it is important to compare the test statistic to the chi-square distribution with both (m-1) and (m-1-k) degrees of freedom. See Kendall and Stuart (1979), for a more complete discussion.

We again test the normal assumption for the Michelson data using `chisq.gof`:

```
> chisq.gof(mich, dist="normal", n.param.est=2,
+                 mean=mean(mich), sd=sqrt(var(mich)))

        Chi-square Goodness of Fit Test
data: mich
Chi-square = 8.7, df = 4, p-value = 0.0691
alternative hypothesis: True cdf does not equal
      the normal Distn. for at least one sample point.

Warning messages:
Expected counts < 5. Chi-square approximation may not
    be appropriate. in: chisq.gof(mich, dist = "normal",
    n.param.est = 2, mean = mean(mich), sd = sqrt( ....
```

Both goodness-of-fit tests return results which make us suspect the null hypothesis, but don't allow us to firmly reject it at 95% or 99% confidence levels.

Note that the distribution theory of the chi-square test is a large sample theory. Therefore when any expected cell counts are small, `chisq.gof` issues a warning message. You should regard *p*-values with caution in this case.

4.5 TWO SAMPLE TESTS

In the two-sample case, you can use `ks.gof` as for the one-sample case (with the second sample `y` filling in for the hypothesized distribution).

The assumptions of the two-sample KS test are:

- the samples are random samples,
- the two samples are mutually independent, and
- the data are measured on at least an ordinal scale.

In addition, the test gives exact results only if the underlying distributions are continuous.

For example, compare the cdfs of vectors `x` and `y` generated from a normal and exponential distribution, respectively:

```
> x <- rnorm(30)
> y <- rexp(100)
> qqplot(x,y)
> cdf.compare(x,y)
```

Figure 4.6 shows a Q-Q plot which is not linear, and cdfs which are quite different. This graphical evidence is verified by a formal KS test:

```
> ks.gof(x,y)

        Two-Sample Kolmogorov-Smirnov Test

data: x and y

ks = 0.3667, p-value = 0.0028
alternative hypothesis:
cdf of x does not equal the
             cdf of y for at least one sample point.
```

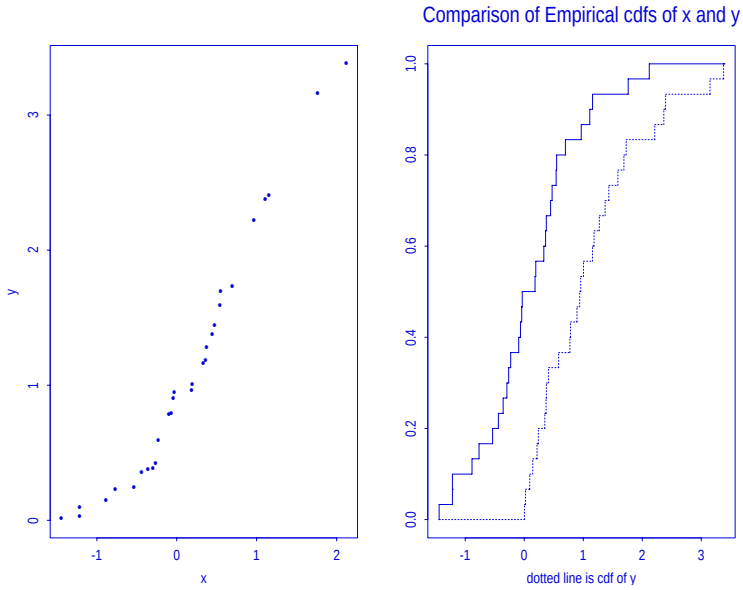

Figure 4.6: *A normal and exponential (dotted line) sample compared.*

4.6 REFERENCES

Kendall, M. G. and Stuart, A. (1979). *The Advanced Theory of Statistics*, 4th edition, 2:Inference and Relationship. Oxford University Press, New York.

Moore, D. S. (1986). Tests of Chi-Squared Type. In D'Agostino, R. B. and Stevens, M. A.", editors, *Goodness-of-Fit Techniques*. Marcel Dekker, New York.

Snedecor, G. W. and Cochran, W. G. (1980). *Statistical Methods*, 7th edition. Iowa State University Press, Ames, Iowa.

STATISTICAL INFERENCE FOR COUNTS AND PROPORTIONS

5

There is a range of tools in S-PLUS to obtain confidence intervals for unknown population parameters.

STATISTICAL INFERENCE FOR COUNTS AND PROPORTIONS

This chapter shows you how to use S-PLUS statistical inference functions for two types of problems that involve *counts* or *proportions*. With these functions, you can obtain confidence intervals for the unknown population parameters and *p*-values for hypothesis tests of the parameter values.

The first type of problem is one where you have one or more samples, or sets of trials, in which the count for each sample represents the number of times that a certain interesting outcome occurs. By common convention, we refer to the occurrence of the outcome of interest as a "success". For example, if you play roulette 100 times at a casino, and you bet on red each time, you are interested in counting the number of times that the color red comes up. This count is a number between 0 and 100. When you divide this count by 100 you get a proportion (that is, a number between 0 and 1). This proportion is a natural estimate of the probability that red comes up on the roulette wheel.

Another example is provided by the famous Salk vaccine trials. These trials involved two groups, one of which received the vaccine and one of which received a placebo. For each group, the count of interest was the number of individuals who contracted polio. The ratio of the number of individuals who contracted polio to the total number of individuals in the group is a proportion that provides a natural estimate of the probability of contracting polio within that group.

The underlying probability model for problems of this first type is the binomial distribution. For each set of trials *i*, this distribution is characterized by the number of trials and the probability p_i that a success occurs on each trial. This probability is called a *proportion* parameter. Your main interest is in making statistical inference statements concerning the probabilities p_1, p_2, …, p_m of occurrence of the event of interest for each of the m sets of trials.

The second type of problem is one where you have counts on the number of occurrences of several possible outcomes for each of two variables. For example, you may be studying three types of cancer and three types of diet (such as low, medium and high fat diets). The two variables of interest may be "type of cancer" and "type of diet". For a fixed set of individuals, you have counts on the number of individuals who fall jointly in each of the categories defined by the simultaneous occurrence of a type of cancer and a diet classification. For problems of this kind, the data is arranged in a two-way table called a *contingency table*.

In this second kind of problem, your main interest is to determine whether there is any *association* between the two variables of interest. The probability model for such problems is one of *statistical independence* between the two variables.

The first three sections of this chapter cover the first type of problem described above, for which the proportion parameters are the probabilities of success, $p_1, p_2, \ldots, p_m$ in m sets of binomial trials. The last section covers the second type of problem, where you are interested in testing the null hypothesis of independence between two variables.

5.1 PROPORTION PARAMETER FOR ONE SAMPLE

When you play roulette and bet on red, you expect your probability of winning to be close to, but slightly less than, 0.5. You expect this because (in the United States) a roulette wheel has 18 red slots, 18 black slots, and two additional slots labeled "0" and "00", for a total of 38 slots into which the ball can fall. Thus, for a "fair" (that is, perfectly balanced) wheel, you expect the probability of red to be $p_0 = 18/38 = .474$. You hope that the house is not cheating you by altering the roulette wheel so that the probability of red is less than .474.

To test whether a given sample has a particular proportion parameter, use the `binom.test` function.

Setting Up the Data

In the roulette case there is little to do, since the only data are the number n of trials and the number x of successes. These two values can be directly supplied as arguments to `binom.test`, as shown in the examples below.

Hypothesis Testing

You can test the null hypothesis that $p = p_0$ using the function `binom.test`. For example, if you bet on red 100 times and red comes up 42 times, you get a p-value for this null hypothesis against the two-sided alternative that $p \neq .474$ as follows:

```
> binom.test(42,100,p=.474)$p.value
[1] 0.3167881
```

The two-sided alternative is the default alternative for `binom.test`. You can get a p-value for a one-sided alternative by using the optional argument `alt=`. For example, in the roulette wheel example you are concerned that the house might cheat you in some way so that $p < p_0$. Use the following to test the null hypothesis against this one-sided alternative:

```
> binom.test(42,100,p=.474,alt="l")$p.value
[1] 0.1632416
```

Here `alt="l"` specifies the "less than" alternative $p < p_0$. To specify the "greater than" alternative $p > p_0$, use `alt="g"`.

The default for the optional argument `p=`, which specifies the null hypothesis value for p, is `p=.5`. For example, suppose you toss a coin 1000 times, with heads coming up 473 times. To test the coin for "fairness;" that is, to test that the probability of heads equals .5, use the following:

```
> binom.test(473,1000)$p.value
[1] 0.09368729
```

Confidence Intervals

The function `binom.test` does not compute a confidence interval for the probability p of success. You can get a confidence interval for p by using the function `prop.test`. For example, the following shows how to obtain the 95% confidence interval for p:

```
> prop.test(45,100)$conf.int
[1] 0.3514281 0.5524574
attr(, "conf.level"):
[1] 0.95
```

The function `prop.test` uses a normal approximation to the binomial distribution for such computations.

You get different confidence intervals by using the optional argument `conf.level=` with different values. For example, to get a 90% confidence interval:

```
> prop.test(45,100,conf.level=.9)$conf.int
[1] 0.3657761 0.5370170
attr(, "conf.level"):
[1] 0.9
```

5.2 PROPORTION PARAMETERS FOR TWO SAMPLES

In the Salk vaccine trials, two large groups were involved in the placebo-control phase of the study. The first group, which received the vaccination, consisted of 200,745 individuals. The second group, which received a placebo, consisted of 201,229 individuals. There were 57 cases of polio in the first group and 142 cases of polio in the second group.

You assume a binomial model for each group, with a probability p_1 of contracting polio in the first group and a probability p_2 of contracting polio in the second group. You are mainly interested in knowing whether or not the vaccine is effective. Thus you are mainly interested in knowing the relationship between p_1 and p_2.

You can use `prop.test` to obtain hypothesis test *p*-values concerning the values of p_1 and p_2, and to obtain confidence intervals for the difference between the values p_1 and p_2.

Setting Up the Data

The first two arguments to prop.test are vectors containing, respectively, the number of successes and the total number of trials. For consistency with the one-sample case, we name these vectors `x` and `n`. In the case of the Salk vaccine trials, a "success" corresponds to contracting polio (although one hardly thinks of this as a literal success!) Thus, you create the vectors `x` and `n` as follows:

```
> x <- c(57,142)
> n <- c(200745,201229)
```

Hypothesis Testing

For two-group problems, you can use either of two null hypotheses: an equal probabilities null hypothesis that $p_1 = p_2$, with no restriction on the common value of these probabilities other than that they be between 0 and 1, or a completely specified probabilities null hypothesis, where you provide specific probabilities for both p_1 and p_2, and test whether the true probabilities are equal to those hypothesized.

The Equal Probabilities Null Hypothesis

When using the equal probabilities null hypothesis, a common alternative hypothesis is the two-sided alternative $p_1 \neq p_2$. These null and alternative hypotheses are the defaults for `prop.test`.

In the Salk vaccine trials, your null hypothesis is that the vaccine has no effect. For the two-sided alternative that the vaccine has some effect, either positive or negative, you get a *p*-value by extracting the `p.value` component of the list returned by `prop.test`:

```
> prop.test(x,n)$p.value
[1] 2.86606e-09
```

The extremely small *p*-value clearly indicates that the vaccine has some effect.

To test the one-sided alternative that the vaccine has a positive effect; that is, that $p_1 < p_2$, use the argument `alt="l"` to `prop.test`:

```
> prop.test(x,n,alt="l")$p.value
[1] 1.43303e-09
```

Here the *p*-value is even smaller, indicating that the vaccine is highly effective in protecting against the contraction of polio.

Completely Specified Null Hypothesis Probabilities

You can also use `prop.test` to test "completely" specified null hypothesis probabilities. For example, suppose you have some prior belief that the probabilities of contracting polio with and without the Salk vaccine are p_{01} = .0002 and p_{02} = .0006, respectively. Then you supply these null hypothesis probabilities as a vector object, using the optional argument `p=`. The *p*-value returned is for the joint probability that both probabilities are equal to the hypothesized probabilities; that is, .0002 and .0006 .

```
> prop.test(x,n,p=c(.0002,.0006))$p.value
[1] 0.005997006
```

The above *p*-value is very small and indicates that the null hypothesis that the joint probability that the underlying population probabilities with and without the Salk vaccine are .0002 and .0006 is very unlikely and should be rejected.

Confidence Intervals

You obtain a confidence interval for the difference p_1 - p_2 in the probabilities of success for the two samples by extracting the `conf.int` component of `prop.test`. For example, to get a 95% confidence interval for the difference in probabilities for the Salk vaccine trials:

```
> prop.test(x,n)$conf.int
[1] -0.0005641810 -0.0002792617
attr(, "conf.level"):
[1] 0.95
```

The 95% confidence level is the default confidence level for `prop.test`. You get a different confidence level by using the optional argument `conf.level=`. For example, to get a 99% confidence interval, use:

```
> prop.test(x,n,conf.level=.99)$conf.int
[1] -0.0006073705 -0.0002360723
attr(, "conf.level"):
[1] 0.99
```

You get a confidence interval for the difference p_1 - p_2 by using `prop.test` only when you use the default null hypothesis that $p_1 = p_2$.

You get all the information provided by `prop.test` as follows:

```
> prop.test(x,n,conf.level=.90)

        2-sample test for equality of proportions with
 continuity correction

data: x out of n
X-squared = 35.2728, df = 1, p-value = 0
```

```
alternative hypothesis: two.sided
90 percent confidence interval:
 -0.0005420769 -0.0003013659
sample estimates:
 prop'n in Group 1 prop'n in Group 2
      0.0002839423      0.0007056637
```

5.3 PROPORTION PARAMETERS FOR THREE OR MORE SAMPLES

Sometimes you may have three or more samples of subjects, with each subject characterized by the presence or absence of some characteristic. An alternative, but equivalent, terminology is that you have three or more sets of trials, with each trial resulting in a success or failure. For example, consider the data shown in table 5.1 for four different studies of lung cancer patients, as presented by Fleiss (1981).

Table 5.1: *Smoking status among lung cancer patients in four studies.*

Study	Number of Patients	Number of Smokers
1	86	83
2	93	90
3	136	129
4	82	70

Each study has a certain number of patients, as shown in the second column of the table, and for each study a certain number of the patients were smokers, as shown in the third column of the table. For this data, you are interested in whether the probability of a patient being a smoker is the same in each of the four studies, that is, whether each of the studies involve patients from a homogeneous population.

Setting Up the Data

The first argument to `prop.test` is a vector containing the number of subjects having the characteristic of interest for each of the groups (or the number of successes for each set of trials). The second argument to `prop.test` is a vector containing the number of subjects in each group (or the number of trials for each set of trials). As in the one and two sample cases, we call these vectors `x` and `n`.

For the smokers data in table 5.1, you create the vectors `x` and `n` as follows:

```
> x <- c(83,90,129,70)
> n <- c(86,93,136,82)
```

Hypothesis Testing

For problems with three or more groups, you can use either an equal probabilities null hypothesis or a completely specified probabilities null hypothesis.

The Equal Probabilities Null Hypothesis

In the lung cancer study, the null hypothesis is that the probability of being a smoker is the same in all groups. Because the default null hypothesis for `prop.test` is that all groups (or sets of trials) have the same probability of success, you get a *p*-value as follows:

```
> prop.test(x,n)$p.value
[1] 0.005585477
```

The *p*-value of .006 is highly significant, so you can not accept the null hypothesis that all groups have the same probability that a patient is a smoker. To see all the results returned by `prop.test`, use:

```
> prop.test(x,n)

        4-sample test for equality of proportions without
 continuity correction

data: x out of n
X-squared = 12.6004, df = 3, p-value = 0.0056
alternative hypothesis: two.sided
sample estimates:
prop'n in Group 1 prop'n in Group 2 prop'n in Group 3
        0.9651163         0.9677419         0.9485294
prop'n in Group 4
        0.8536585
```

Completely Specified Null Hypothesis Probabilities

If you want to test a completely specified set of null hypothesis probabilities, you need to supply the optional argument p=, with the value of this argument being a vector of probabilities having the same length as the first two arguments, x and n.

For example, in the lung cancer study, to test the null hypothesis that the first three groups have a common probability .95 of a patient being a smoker, while the fourth group has a probability .90 of a patient being a smoker, create the vector p as follows, the use it as an argument to `prop.test`:

```
> p <- c(.95,.95,.95,.90)
> prop.test(x,n,p)$p.value
[1] 0.5590245
Warning messages:
  Expected counts < 5. Chi-square approximation may not be
appropriate.
```

Alternatively, you could use

```
prop.test(x,n,p=c(.95,.95,.95,.90))$p.value
```

Confidence Intervals

Confidence intervals are not computed by `prop.test` when you have three or more groups (or sets of trials).

5.4 CONTINGENCY TABLES AND TESTS FOR INDEPENDENCE

The Salk vaccine trials in the early 1950s resulted in the data presented in table 5.2.

Table 5.2: *Contingency table of Salk vaccine trials data.*

	No Polio	Non-paralytic Polio	Paralytic Polio	Totals
Vaccinated	200,688	24	33	200,745
Placebo	201,087	27	115	201,229
Totals	401,775	51	148	401,974

There are two categorical variables for the Salk trials: vaccination status, which has the two levels “vaccinated” and “placebo,” and polio status, which has the three levels “no polio,” “non-paralytic polio,” and “paralytic polio.” Of 200,745 individuals who were vaccinated, 24 contracted non-paralytic polio, 33 contracted paralytic polio, and the remaining 200,688 did not contract any kind of polio. Of 201,229 individuals who received the placebo, 27 contracted non-paralytic polio, 115 contracted paralytic polio, and the remaining 201,087 did not contract any kind of polio.

Tables such as table 5.2 are called contingency tables. A contingency table lists the number of counts for the joint occurrence of two levels (or possible outcomes), one level for each of two categorical variables. The levels for one of the categorical variables correspond to the columns of the table, and the levels for the other categorical variable correspond to the rows of the table.

When working with contingency table data, your primary interest is most often determining whether there is any association in the form of statistical dependence between the two categorical variables whose counts are displayed in the table. The null hypothesis is that the two variables are statistically independent. You can test this null hypothesis with the functions `chisq.test` and `fisher.test`. The function `chisq.test` is based on the classic chi-square test statistic, and the associated *p*-value computation entails some approximations. The function `fisher.test` computes an

exact *p*-value for tables having at most 10 levels for each variable. The function `fisher.test` also entails a statistical conditioning assumption.

For contingency tables involving confounding variables, which are variables related to both variables of interest, you can test for independence using the function `mantelhaen.test`, which performs the Mantel-Haenszel test. For contingency tables involving matched pairs, use the function `mcnemar.test` to perform McNemar's chi-square test.

The functions for testing independence in contingency tables do not compute confidence intervals, only *p*-values and the associated test statistic.

The Chi-square and Fisher Tests of Independence

The chi-square and Fisher's exact tests are familiar methods for testing independence. The Fisher test is often recommended when expected counts in any cell are below 5, as the chi-square probability computation becomes increasingly inaccurate when the expected counts in any cell are low. (S-PLUS produces a warning message in that case). Other factors may also influence your choice of which test to use, however. Refer to a statistics text for further discussion if you are unsure which test to use.

Setting Up the Data

You can set up your contingency table data in several ways. Which way you choose depends to some extent on the original form of the data and whether the data involves a large number of counts or a small to moderate number of counts.

Two-Column Matrix Objects

If you already have the data in the form of a contingency table in printed form, as in table 5.2, the easiest thing to do is to put the data in matrix form (excluding the marginal totals, if provided in the original data). For example,

```
> salk.mat <- rbind(c(200688,24,33),c(201087,27,115))
> salk.mat
       [,1] [,2] [,3]
[1,] 200688   24   33
[2,] 201087   27  115
```

You could obtain the same result in a slightly different way as follows:

```
> salk.mat <- matrix(c(200688, 24, 33, 201087, 27, 115),
+ 2, 3, byrow=T)
```

Two Numeric Vector Objects

You may be given the raw data in the form of two equal-length coded vectors, one for each variable. In such cases, the length of the vectors corresponds to the number of individuals, with each entry indicating the level by a numeric coding. For example, suppose you have two variables from a clinical trial of the drug propranolol. (The data was reported by P. J. D. Snow in *Lancet*, (Snow 1965)). The vector `drug` is coded for control or propranolol status, and the vector `status` is coded yes or no indicating whether the patient

survived at least 28 days. The raw data is as follows:

```
> drug
 [1] "control" "control" "control" "control" "prop"
 [6] "control" "prop"    "control" "prop"    "control"
[11] "prop"    "prop"    "control" "prop"    "prop"
[16] "control" "control" "prop"    "prop"    "prop"
[21] "prop"    "control" "prop"    "control" "control"
[26] "prop"    "control" "control" "control" "control"
[31] "control" "control" "prop"    "control" "prop"
[36] "control" "prop"    "prop"    "prop"    "control"
[41] "prop"    "control" "prop"    "control" "prop"
[46] "control" "prop"    "control" "control" "prop"
[51] "prop"    "prop"    "control" "prop"    "prop"
[56] "prop"    "control" "control" "control" "prop"
[61] "prop"    "control" "prop"    "control" "prop"
[66] "control" "prop"    "control" "prop"    "control"
[71] "prop"    "control" "prop"    "control" "prop"
[76] "control" "prop"    "control" "prop"    "control"
[81] "prop"    "control" "prop"    "control" "prop"
[86] "control" "prop"    "control" "control" "prop"
[91] "prop"
> status
 [1] "yes" "yes" "yes" "no"  "yes" "yes" "yes" "yes" "yes"
[10] "yes" "yes" "no"  "no"  "yes" "yes" "no"  "no"  "yes"
[19] "yes" "yes" "yes" "no"  "yes" "yes" "no"  "yes" "no"
[28] "yes" "no"  "yes" "no"  "yes" "no"  "yes" "yes" "no"
[37] "no"  "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes"
[46] "no"  "yes" "no"  "yes" "yes" "yes" "yes" "yes" "yes"
[55] "yes" "yes" "yes" "yes" "yes" "no"  "yes" "yes" "yes"
[64] "no"  "no"  "no"  "yes" "yes" "yes" "yes" "no"  "no"
[73] "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes" "yes"
[82] "yes" "yes" "yes" "yes" "yes" "yes" "no"  "no"  "yes"
[91] "no"
```

To obtain the contingency table (without marginal count totals) use the function `table` with `status` and `drug` as arguments:

```
> table(status,drug)
     control prop
no        17    7
yes       29   38
```

Two Factor Objects

Your data may already be in the form of two factor objects, or you may want to put your data in that form for further analysis in S-PLUS. For example, to put `drug` and `status` into factor form, use the `factor` command as follows:

```
> drug.fac <- factor(drug)
> drug.fac
 [1] control control control control prop    control
 [7] prop    control prop    control prop    prop
[13] control prop    prop    control control prop
[19] prop    prop    prop    control prop    control
[25] control prop    control control control control
[31] control control prop    control prop    control
[37] prop    prop    prop    control prop    control
[43] prop    control prop    control prop    control
[49] control prop    prop    prop    control prop
[55] prop    prop    control control control prop
[61] prop    control prop    control prop    control
[67] prop    control prop    control prop    control
[73] prop    control prop    control prop    control
[79] prop    control prop    control prop    control
[85] prop    control prop    control control prop
[91] prop
> status.fac <- factor(status)
> status.fac
 [1] yes yes yes no  yes yes yes yes yes yes yes no
[13] no  yes yes no  no  yes yes yes yes no  yes yes
[25] no  yes no  yes no  yes no  yes no  yes yes no
[37] no  yes yes yes yes yes yes yes yes no  yes no
[49] yes yes yes yes yes yes yes yes yes yes yes no
[61] yes yes yes no  no  no  yes yes yes yes no  no
[73] yes yes yes yes yes yes yes yes yes yes yes yes
[85] yes yes yes no  no  yes no
```

Then use `drug.fac` and `status.fac` as arguments to the functions described below.

The Chi-square Test of Independence

You use the function `chisq.test` to perform a classical chi-square test of the null hypothesis that the categorical variables of interest are independent. For example, using the matrix form of data object `salk.mat` for the Salk vaccine trials

```
> chisq.test(salk.mat)$p.value
[1] 1.369748e-10
```

which yields an exceedingly small *p*-value. This leads to rejection of the null hypothesis of no association between polio status and vaccination status.

To get all the information computed by `chisq.test`, use `chisq.test` without specifying a return component, as usual:

```
> chisq.test(salk.mat)

          Pearson's chi-square test without Yates'
 continuity correction

data: salk.mat
X-squared = 45.4224, df = 2, p-value = 0
```

You could also use the two factor objects, such as `drug.fac` and `status.fac` as follows:

```
> chisq.test(drug.fac,status.fac)

         Pearson's chi-square test with Yates'
 continuity correction

data: drug.fac and status.fac
X-squared = 4.3198, df = 1, p-value = 0.0377
```

The results are the same no matter which way you have set up the data.

Fisher's Exact Test of Independence

You can perform an exact test of indepence by using the S-PLUS function `fisher.test`. You can use any data object type that can be used with `chisq.test`. For example, using the factor object for the propranolol clinical trial:

```
> fisher.test(drug.fac,status.fac)

         Fisher's exact test

data: drug.fac and status.fac
p-value = 0.0314 alternative hypothesis: two.sided
```

When using `fisher.test` you should be aware that the *p*-value is computed conditionally on the fixed marginal counts of the contingency table you are analyzing That is, the inference does not extend to all possible tables that might be obtained by repeating the experiment and getting different marginal counts.

The Mantel-Haenszel Test of Independence

A cancer study produced the data shown in table 5.3 and table 5.4, as reported by Rosner (1986). In these tables, "case" refers to an individual who had cancer and "control" refers to an individual who did not have cancer. A "passive" smoker is an individual who lives with a smoker. A smoker can also be a passive smoker if that smoker lives with a spouse who also smokes.

Table 5.3: *Nonsmokers in cancer study.*

case-control status	passive smoker	not a passive smoker
case	120	111
control	80	155

Table 5.4: *Smokers in cancer study.*

case-control status	passive smoker	not a passive smoker
case	161	117
control	130	124

For each of these tables, you can use `chisq.test` or `fisher.test` to test for independence between cancer status and passive smoking status. The data is presented in separate tables because "smoking status;" that is, being a smoker or not being a smoker, could be a *confounding variable*, because both smoking status and passive smoking status are related to the outcome, cancer status, and because smoking status may be related to the smoking status of the spouse. You would like to be able to combine the information in both tables so as to produce an overall test of independence between cancer status and passive smoking status. You can do so for *two or more two-by-two* tables, by using the function `mantelhaen.test`, which performs the *Mantel-Haenszel* test.

Since the data is now associated with three categorical variables, the two main variables of interest plus a confounding variable, you can prepare your data in any one of the following forms:

- a three-dimensional array which represents the three dimensional contingency table (two-by-two tables stacked on top of one another)

- three numerical vectors representing each of the three categorical variables, two of primary interest and one a confounding variable
- three factor objects for the three categorical variables

Which form you use depends largely on the form in which the data is presented to you. For example, the data in tables 5.3 and 5.4 are ideal for use with a three-dimensional array:

```
> x.array <- array(c(120, 80, 111, 155, 161, 130,
+ 117, 124),c(2,2,2))
> x.array

, , 1
     [,1] [,2]
[1,]  120  111
[2,]   80  155

, , 2
     [,1] [,2]
[1,]  161  117
[2,]  130  124
> mantelhaen.test(x.array)$p.value
[1] 0.0001885083
> mantelhaen.test(x.array)

         Mantel-Haenszel chi-square test with
 continuity correction

data: x.array
Mantel-Haenszel chi-square = 13.9423, df = 1,
p-value = 2e-04
```

McNemar Test for Symmetry using Matched Pairs

In some experiments with two categorical variables, one of the variables specifies two or more groups of individuals who receive different treatments. In such situations, matching of individuals is often carried out in order to increase the precision of statistical inference. However, when matching is carried out the observations usually are not independent. In such cases, the inference obtained from `chisq.test`, `fisher.test` and `mantelhaen.test` is not valid because these tests all assume independent observations. The function `mcnemar.test` allows you to obtain a valid inference for experiments where matching is carried out.

Consider, for example, the data in table 5.5, as reported by Rosner (1986). In this table, each entry represents one pair. For instance, the “5” in the lower

left cell means that in 5 pairs, the individual with treatment A died, while the individual that that person was paired with, who received treatment B, survived.

Table 5.5: *Matched pair data for cancer study.*

	survive with treatment B	die with treatment B
survive with treatment A	90	16
die with treatment A	5	510

Your interest is in the relative effectiveness of treatments A and B in treating a rare form of cancer. Each count in the table is associated with a matched pair of individuals.

A pair in the table for which one member of a matched pair survives while the other member dies is called a discordant pair. There are 16 discordant pairs in which the individual who received treatment A survives and the individual who received treatment B dies. There are 5 discordant pairs with the reverse situation in which the individual who received treatment A dies and the individual who received treatment B survives.

If both treatments are equally effective, then you expect these two types of discordant pairs to occur with "nearly" equal frequency. Put in terms of probabilities, your null hypothesis is that p1 = p2, where p1 is the probability that the first type of discordancy occurs in a matched pair of individuals, and p2 is the probability that the second type of discordancy occurs.

We illustrate the use of `mcnemar.test` on the above data, putting the data into the form of a matrix object:

```
> x.matched <- cbind(c(90,5),c(16,510))
> x.matched
     [,1] [,2]
[1,]   90   16
[2,]    5  510
> mcnemar.test(x.matched)$p.value
[1] 0.02909633
> mcnemar.test(x.matched)
        McNemar's chi-square test with continuity
 correction

data: x.matched
McNemar's chi-square = 4.7619, df = 1, p-value = 0.0291
```

You can use `mcnemar.test` with two numeric vector objects, or two factor objects, as the data arguments (just as with the other functions in this section). You can also use `mcnemar.test` with matched pair tables having more than two rows and more than two columns. In such cases, the null hypothesis is symmetry of the probabilities pij associated with each row and column of the table; that is, the null hypothesis is that pij = pji for each combination of i and j.

5.5 REFERENCES

Bishop, Y. M. M. and Fienberg, S. J. and Holland, P. W. (1980). *Discrete Multivariate Analysis: Theory and Practice*. The MIT Press, Cambridge, MA.

Conover, W. J. (1980). *Practical Nonparametric Statistics, 2nd edition*. John Wiley, New York.

Fienberg, S. E. (1983). *The Analysis of Cross-Classified Categorical Data*, 2nd edition. The MIT Press, Cambridge, MA.

Fleiss, J. L. (1981). Statistical Methods for Rates and Proportions, 2nd edition. Wiley, New York.

Lehmann, E.L. (1975). *Nonparametrics: Statistical Methods Based on Ranks*. Holden-Day, San Francisco.

Rosner, B. (1986). Fundamentals of Biostatistics. Duxbury Press, Boston.

Snedecor, G. W. and Cochran, W. G. (1980). *Statistical Methods*, 7th edition. Iowa State University Press, Ames, Iowa.

Snow, P. J. D. (1965). Lancet.

CROSS-CLASSIFIED DATA AND CONTINGENCY TABLES

6

S-PLUS easily handles categorical data, enabling cross-classification and cross-tabulation of the data.

CROSS-CLASSIFIED DATA AND CONTINGENCY TABLES

Much data of interest is categorical in nature. Did patients receive treatment A, B, or C; did they survive? Do the people in a sample population smoke? Do they have high cholesterol counts? Have they had heart trouble? These data are stored in S-PLUS as *factors*, that is, as vectors where the elements indicate one of a number of *levels*. A useful way of looking at this data is to *cross-classify* it and get a *count* of the number of cases sharing a given combination of levels, and then create a multi-way contingency table (a *cross-tabulation*) showing the levels and the counts.

Consider the data set `claims`. It contains the number of claims for auto insurance received broken down by the following variables: age of claimant, age of car, type of car, and the average cost of the claims. We can disregard the costs for the moment, and consider the question of which groups of claimants generate the most claims. To make the work easier we create an new data frame `claims.src` which does not contain the `cost` variable:

```
> claims.src <- claims[,-4]
> summary(claims.src)
            age    car.age  type        number
 17-20       :16   0-3:32   A:32   Min.   :  0.00
 21-24       :16   4-7:32   B:32   1st Qu.:  9.00
 25-29       :16   8-9:32   C:32   Median : 35.50
 30-34,35-39 :32   10+:32   D:32   Mean   : 69.86
 40-49       :16                   3rd Qu.: 96.25
 50-59       :16                     Max. :434.00
 60+         :16
```

Use the function `crosstabs` to generate tables of *cross-classified* data. Table 6.1 shows car age vs. car types, the output generated by the following call to `crosstabs`:

```
> crosstabs(number~car.age+type, claims.src)
```

The first argument to `crosstabs` is a *formula* that tells which variables to include in the table. The second argument is the data set where the variables are found. The complete call to `crosstabs` is stored in the resulting object as the attribute `"call"`, and is printed at the top of the table.

The next item of information is the number of cases, that is, the total count of all the variables considered. In this example, this is the total of the `number` variable; that is, `sum(claims.src$number)`.

Then you get a key which tells you how to interpret the cells of the table. `N` is

the count; below it are the proportions of the whole that the count represents: the proportion of the row total, the proportion of the column total and the proportion of the table total. If there are only two terms in the formula, the table total will be the same as the number of cases. A quick look at the counts, and in particular at the row totals (`4134`, `3549`, `822`, `437`), shows that there are fewer older cars than newer cars; relatively few cars survive to be eight or nine years old, and the number of cars over ten years old is a tenth that of cars three years or newer. It is slightly more surprising to note the four types of cars don't seem to age equally. You can get an inkling of this by comparing the cells near the top of the table with those near the bottom, but if you compare the third figure in each cell, the one the key tells us is `N/ColTotal`, the progression becomes clear. Of cars of type `D`, 64% are no more than three years old, while only 4% are eight or nine, and less than 2% are over 10. Compare this to type `A` cars, where there are slightly more in the four to seven year age group than in the under three year, the proportion between eight and nine is 0.147 and the proportion over ten years is 12. It seems as if the the type of car is related to its age, and if we look below the table where the results of the χ^2 test for independence are written, we see that the *p*-value is so small it appears as 0.

Of course, we must remember these data are from insurance claims forms—this is not a sample of all the cars on the road, just those that got into accidents and had insurance policies with the company that collected the data.

There may also be an interaction between car type/car age and the age of the owner (which seems likely) and between the age of the owner and the likelihood of a automobile accident.

With `crosstabs`, it is possible to tabulate all this data at once, and print the resulting table in a series of layers, each showing two variables. Thus when we type `crosstabs(number~car.age+type+age, claims.src)`, we get a series of 8 layers, one for each factor (age group) in the variable `age`. The variable represented by the first term in the formula to the left of the `~`, `age.car`, is represented by the rows of each layer, the second term, `car.age` is represented by the columns, and each level of the third, `type`, produces a separate layer. If there were more than three variables, there would be one layer for each possible combination of levels in the variables after the first two. Part of the first of these layers is shown in table 6.2. Note that the number written in the bottom right margin is the sum of the row totals, and is not the same as the number of cases in the entire table, which is still found at the top of the display and which is used to compute `N/Total`, the fourth

Table 6.1: *Output from call to* `crosstabs`.

```
Call:
crosstabs(number ~ car.age + type, claims.src)
8942 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
car.age|type
       |A      |B      |C      |D      |RowTotl|
-------+-------+-------+-------+-------+-------+
0-3    | 391   |1538   |1517   | 688   |4134   |
       |0.0946 |0.3720 |0.3670 |0.1664 |0.462  |
       |0.3081 |0.3956 |0.5598 |0.6400 |       |
       |0.0437 |0.1720 |0.1696 |0.0769 |       |
-------+-------+-------+-------+-------+-------+
4-7    | 538   |1746   | 941   | 324   |3549   |
       |0.1516 |0.4920 |0.2651 |0.0913 |0.397  |
       |0.4240 |0.4491 |0.3472 |0.3014 |       |
       |0.0602 |0.1953 |0.1052 |0.0362 |       |
-------+-------+-------+-------+-------+-------+
8-9    | 187   | 400   | 191   |  44   |822    |
       |0.2275 |0.4866 |0.2324 |0.0535 |0.092  |
       |0.1474 |0.1029 |0.0705 |0.0409 |       |
       |0.0209 |0.0447 |0.0214 |0.0049 |       |
-------+-------+-------+-------+-------+-------+
10+    | 153   | 204   |  61   |  19   |437    |
       |0.3501 |0.4668 |0.1396 |0.0435 |0.049  |
       |0.1206 |0.0525 |0.0225 |0.0177 |       |
       |0.0171 |0.0228 |0.0068 |0.0021 |       |
-------+-------+-------+-------+-------+-------+
ColTotl|1269   |3888   |2710   |1075   |8942   |
       |0.14   |0.43   |0.30   |0.12   |       |
-------+-------+-------+-------+-------+-------+
Test for independence of all factors
        Chi^2 = 588.2952 d.f.=9 (p=0)
        Yates' correction not used
```

figure in each cell.

Table 6.2: *Further cross-tabulations of* `claims` *data.*

```
type=A
age     |car.age                                    8942 cases
        |0-3     |4-7     |8-9     |10+     |RowTotl|  in table
--------+--------+--------+--------+--------+-------+ +----------+
17-20   |  8     |  8     |  4     |  1     |21     | |N         |
        |0.38095 |0.38095 |0.19048 |0.04762 |0.0165 | |N/RowTotal|
        |0.02046 |0.01487 |0.02139 |0.00654 |       | |N/ColTotal|
        |0.00089 |0.00089 |0.00045 |0.00011 |       | |N/Total   |
--------+--------+--------+--------+--------+-------+ +----------+
21-24   | 18     | 31     | 10     |  4     |63     |
        |                                           |
                      .  .  .
        |                                           |
        |                                           |
--------+--------+--------+--------+--------+-------+
35-39   | 43     | 73     | 21     | 14     |151    |
        |0.28477 |0.48344 |0.13907 |0.09272 |0.1190 |
        |0.10997 |0.13569 |0.11230 |0.09150 |       |
        |0.00481 |0.00816 |0.00235 |0.00157 |       |
--------+--------+--------+--------+--------+-------+
40-49   | 90     | 98     | 35     | 22     |245    |
        |0.36735 |0.40000 |0.14286 |0.08980 |0.1931 |
        |0.23018 |0.18216 |0.18717 |0.14379 |       |
        |0.01006 |0.01096 |0.00391 |0.00246 |       |
--------+--------+--------+--------+--------+-------+
50-59   | 69     |120     | 42     | 35     |266    |
        |0.25940 |0.45113 |0.15789 |0.13158 |0.2096 |
        |0.17647 |0.22305 |0.22460 |0.22876 |       |
        |0.00772 |0.01342 |0.00470 |0.00391 |       |
--------+--------+--------+--------+--------+-------+
60+     | 64     |100     | 43     | 53     |260    |
        |0.24615 |0.38462 |0.16538 |0.20385 |0.2049 |
        |0.16368 |0.18587 |0.22995 |0.34641 |       |
        |0.00716 |0.01118 |0.00481 |0.00593 |       |
--------+--------+--------+--------+--------+-------+
ColTotl |391     |538     |187     |153     |1269   |
        |0.308   |0.424   |0.147   |0.121   |       |
--------+--------+--------+--------+--------+-------+
```

6.1 CHOOSING SUITABLE DATA SETS

Cross tabulation is a technique for categorical data. You tabulate the number of cases for each combination of factors between your variables. In the claims data set these numbers were already tabulated. However, when looking at data that has been gathered as a count, you must always keep in mind exactly what is being counted—thus we can tell that of the 40–49 year old car owners who submitted insurance claims, 43% owned cars of type `B`, and of the cars of type `B` whose owners submitted insurance claims, 25% were owned by 40-49 year olds.

The data set `guayule` also has a response variable which is a count, while all the predictor variables are factors. Here, the thing being counted is the number of rubber plants that sprouted from seeds of a number of varieties subjected to a number of treatments. However, this experiment was designed so that the same number of seeds were planted for each possible combination of the factors of the controlling variables. Since we know the exact make-up of the larger population from which our counts are taken, we can observe the relative size of counts with complaisance and draw conclusions with great confidence. The difference between `guayule` and `claims` is that with the former we can view the outcome variable as a binomial response variable ("sprouted"/"didn't sprout") for which we have tabulated one of the outcomes ("sprouted"), and in the `claims` data set we can't.

Another data set in which all the controlling variables are factors is `solder`.

```
> summary(solder)
 Opening    Solder       Mask     PadType  Panel       skips
 S:300   Thin :450  A1.5:180  L9   : 90  1:300  Min.   : 0.00
 M:300   Thick:450  A3  :270  W9   : 90  2:300  1st Qu.: 0.00
 L:300              A6  : 90  L8   : 90  3:300  Median : 2.00
                    B3  :180  L7   : 90         Mean   : 5.53
                    B6  :180  D7   : 90         3rd Qu.: 7.00
                              L6   : 90          Max.   :48.00
                         (Other):360
```

The response variable is the number of skips appearing on a finished circuit board. Since any skip on a board renders it unusable, we can easily turn this into a binary response variable:

```
> attach(solder)
> good <- factor(skips==0)
```

Then, when we want to look at the interaction between the variables, `crosstabs` counts up all the cases with like levels among the factors:

```
crosstabs( ~ Opening + Mask + good)
Call:
crosstabs( ~ Opening + Mask + good)
900 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
good=FALSE
Opening|Mask
       |A1.5   |A3     |A6     |B3     |B6     |RowTotl|
-------+-------+-------+-------+-------+-------+-------+
S      |49     |76     |30     |60     |60     |275    |
       |0.1782 |0.2764 |0.1091 |0.2182 |0.2182 |0.447  |
       |0.5326 |0.5033 |0.3371 |0.4444 |0.4054 |       |
       |0.0544 |0.0844 |0.0333 |0.0667 |0.0667 |       |
-------+-------+-------+-------+-------+-------+-------+
M      |22     |35     |59     |39     |51     |206    |
       |0.1068 |0.1699 |0.2864 |0.1893 |0.2476 |0.335  |
       |0.2391 |0.2318 |0.6629 |0.2889 |0.3446 |       |
       |0.0244 |0.0389 |0.0656 |0.0433 |0.0567 |       |
-------+-------+-------+-------+-------+-------+-------+
L      |21     |40     | 0     |36     |37     |134    |
       |0.1567 |0.2985 |0.0000 |0.2687 |0.2761 |0.218  |
       |0.2283 |0.2649 |0.0000 |0.2667 |0.2500 |       |
       |0.0233 |0.0444 |0.0000 |0.0400 |0.0411 |       |
-------+-------+-------+-------+-------+-------+-------+
ColTotl|92     |151    |89     |135    |148    |615    |
       |0.1496 |0.2455 |0.1447 |0.2195 |0.2407 |       |
-------+-------+-------+-------+-------+-------+-------+
```

```
good=TRUE
Opening|Mask
       |A1.5   |A3     |A6     |B3     |B6     |RowTotl|
-------+-------+-------+-------+-------+-------+-------+
S      |11     |14     | 0     | 0     | 0     |25     |
       |0.4400 |0.5600 |0.0000 |0.0000 |0.0000 |0.088  |
       |0.1250 |0.1176 |0.0000 |0.0000 |0.0000 |       |
       |0.0122 |0.0156 |0.0000 |0.0000 |0.0000 |       |
-------+-------+-------+-------+-------+-------+-------+
M      |38     |25     | 1     |21     | 9     |94     |
       |0.4043 |0.2660 |0.0106 |0.2234 |0.0957 |0.330  |
       |0.4318 |0.2101 |1.0000 |0.4667 |0.2812 |       |
       |0.0422 |0.0278 |0.0011 |0.0233 |0.0100 |       |
-------+-------+-------+-------+-------+-------+-------+
L      |39     |80     | 0     |24     |23     |166    |
       |0.2349 |0.4819 |0.0000 |0.1446 |0.1386 |0.582  |
       |0.4432 |0.6723 |0.0000 |0.5333 |0.7188 |       |
       |0.0433 |0.0889 |0.0000 |0.0267 |0.0256 |       |
-------+-------+-------+-------+-------+-------+-------+
ColTotl|88     |119    |1      |45     |32     |285    |
       |0.3088 |0.4175 |0.0035 |0.1579 |0.1123 |       |
-------+-------+-------+-------+-------+-------+-------+
Test for independence of all factors
        Chi^2 = 377.3556 d.f.= 8 (p=0)
        Yates' correction not used
```

In the first example above we specified where to look for the variables `age`, `car.age` and `type` by giving the data frame `claims.src` as the second argument of `crosstabs`. In the second example, we attached the data frame `solder` and let `crosstabs` find the variables in the search list. Both methods work because, when `crosstabs` goes to interpret a term in the formula, it looks first in the data frame specified by the argument data and then in the search list. You can specifiy the data set with the name of a data frame, or a frame number in which to find an attached data frame. Using a frame number gives the advantage of speed that comes from attaching the data frame, while protecting against the possibility of having masked the name of one of the variables with something in your `.Data` directory:

```
> attach(guayule)
> search()
[1] ".Data"
[2] "guayule" . . .
> rubber <- crosstabs(plants~variety+treatment, data=2)
```

If you specify a data frame and do not give a formula, `crosstabs` uses the formula `~ .`, that is, it will cross classify all the variables in the data frame. Any variable names not found in the specified data frame (which is all of them if you don't specify any) are sought in the search list.

6.2 CROSS-TABULATING CONTINUOUS DATA

As was seen in the example of the `solder` data frame above, it is fairly easy to turn a continuous response variable into a binomial response variable. Clearly, we could have used any logical expression that made sense to do so—we could have chosen any cutoff point for acceptable numbers of skips.

A somewhat harder problem is presented by the case where you want a multinomial factor from continuous data. You can make judicious use of the `cut` function to turn the continuous variables into factors, but you need to put care and thought into the points at which to separate the data into ranges. The quartiles given by the function `summary` offer a good starting point. The data frame `kyphosis` represents data on 81 children who have had corrective spinal surgery. The variables here are whether a postoperative deformity (kyphosis) is present, the age of the child in months, the number of vertebrae involved in the operation, and beginning of the range of vertebrae involved.

```
> summary(kyphosis)
  Kyphosis        Age             Number          Start
absent :64  Min.   :  1.00  Min.   : 2.000  Min.   : 1.00
present:17  1st Qu.: 26.00  1st Qu.: 3.000  1st Qu.: 9.00
            Median : 87.00  Median : 4.000  Median :13.00
            Mean   : 83.65  Mean   : 4.049  Mean   :11.49
            3rd Qu.:130.00  3rd Qu.: 5.000  3rd Qu.:16.00
            Max.   :206.00  Max.   :10.000  Max.   :18.00
```

The summary of these variables suggests that two year intervals might be a reasonable division for the age. We use the `cut` function to break the variable `Age` into factors at a sequence of points at 24 month intervals and to label the resulting levels with the appropriate range of years. Since there are at most nine values for `Number` we leave it alone for the moment. Since the mean of the `Start` variable is close to the first quartile, a fairly coarse division of `Start` is probably sufficient. We could require that `cut` simply divide the

data into four segments of equal length with the command `cut(Start, 4)`, but the results of this, while mathematically correct, look a bit bizarre—the first level thus created is `" 0.830+ thru 5.165"`. The `pretty` function divides the range of `Start` into equal intervals with whole number end points, and the `cut` function makes them into levels with reasonable names:

```
> attach(kyphosis)
> kyphosis.fac <- data.frame(Kyphosis=Kyphosis,
+  Age =  cut(Age, c( seq(0, 144, by=24), 206),
+    labels=
+      c("0-2","2-4","4-6","6-8","8-10","10-12","12+")),
+  Number = Number,
+  Start = cut(Start, pretty(Start, 4) ) )
> detach(2)
> summary(kyphosis.fac)
   Kyphosis       Age         Number                Start
 absent :64   0-2  :20   Min.   : 2.000    0+ thru  5:13
 present:17   2-4  : 7   1st Qu.: 3.000    5+ thru 10:14
              4-6  : 8   Median : 4.000   10+ thru 15:32
              6-8  : 9   Mean   : 4.049   15+ thru 20:22
              8-10 :11   3rd Qu.: 5.000
              10-12:14   Max.   :10.000
              12+  :12
> attach(kyphosis.fac)
```

The cross-tabulation of this data can then be easily examined:

```
> crosstabs(~Age+Kyphosis, kyphosis.fac)
Call:
crosstabs( ~ Age + Kyphosis, kyphosis.fac)
81 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
Age    |Kyphosis
       |absent |present|RowTotl|
-------+-------+-------+-------+
0-2    |19     | 1     |20     |
       |0.950  |0.050  |0.247  |
       |0.297  |0.059  |       |
       |0.235  |0.012  |       |
```

```
-------+-------+-------+-------+
2-4    | 6     | 1     |7      |
       |                       |
          .   .   .
       |                       |
-------+-------+-------+-------+
10-12  | 9     | 5     |14     |
       |0.643  |0.357  |0.173  |
       |0.141  |0.294  |       |
       |0.111  |0.062  |       |
-------+-------+-------+-------+
12+    |11     | 1     |12     |
       |0.917  |0.083  |0.148  |
       |0.172  |0.059  |       |
       |0.136  |0.012  |       |
-------+-------+-------+-------+
ColTotl|64     |17     |81     |
       |0.79   |0.21   |       |
-------+-------+-------+-------+
Test for independence of all factors
        Chi^2 = 9.588004 d.f.= 6 (p=0.1431089)
        Yates' correction not used
        Some expected values are less than 5,
             don't trust stated p-value
```

6.3 CROSS-CLASSIFYING SUBSETS OF DATA FRAMES

There are two ways to subset a data frame for cross-classification. First, the `crosstabs` function will cross-tabulate only those variables specified in the formula. If there is one variable in the data frame in which you are not interested, don't mention it. Second, you can choose which rows you want to consider with the subset argument. You can use anything you would normally use to subscript the rows of a data frame. Thus, the `subset` argument can be an expression that evaluates to a logical vector, or a vector of row numbers or row names.

As an example, recall the `solder` data set. You can look at the relation between the variables without turning `skips` explicitly into a binomial variable by using it to subscript the rows of the data frame:

```
> crosstabs(~Solder+Opening, solder, subset= skips<10)
```

```
Call:
crosstabs( ~ Solder+Opening, solder, subset = skips<10)
729 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
Solder |Opening
       |S      |M      |L      |RowTotl|
-------+-------+-------+-------+-------+
Thin   | 50    |133    |140    |323    |
       |0.155  |0.412  |0.433  |0.44   |
       |0.294  |0.494  |0.483  |       |
       |0.069  |0.182  |0.192  |       |
-------+-------+-------+-------+-------+
Thick  |120    |136    |150    |406    |
       |0.296  |0.335  |0.369  |0.56   |
       |0.706  |0.506  |0.517  |       |
       |0.165  |0.187  |0.206  |       |
-------+-------+-------+-------+-------+
ColTotl|170    |269    |290    |729    |
       |0.23   |0.37   |0.40   |       |
-------+-------+-------+-------+-------+
Test for independence of all factors
        Chi^2 = 20.01129 d.f.= 2 (p=4.514445e-05)
        Yates' correction not used
```

A more common use of the subscript is to look at some of the variables while considering only a subset of the levels of another:

```
> crosstabs( ~ Solder+Opening+good, subset = Panel == "1")
Call:
crosstabs( ~ Solder+Opening+good, subset = Panel == "1")
300 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
```

```
good=FALSE
Solder |Opening
       |S      |M      |L      |RowTotl|
-------+-------+-------+-------+-------+
Thin   |49     |33     |31     |113    |
       |0.4336 |0.2920 |0.2743 |0.59   |
       |0.5444 |0.5410 |0.7949 |       |
       |0.1633 |0.1100 |0.1033 |       |
-------+-------+-------+-------+-------+
Thick  |41     |28     | 8     |77     |
       |0.5325 |0.3636 |0.1039 |0.41   |
       |0.4556 |0.4590 |0.2051 |       |
       |0.1367 |0.0933 |0.0267 |       |
-------+-------+-------+-------+-------+
ColTotl|90     |61     |39     |190    |
       |0.474  |0.321  |0.205  |       |
-------+-------+-------+-------+-------+
good=TRUE
Solder |Opening
       |S      |M      |L      |RowTotl|
-------+-------+-------+-------+-------+
Thin   | 1     |17     |19     |37     |
       |0.0270 |0.4595 |0.5135 |0.34   |
       |0.1000 |0.4359 |0.3115 |       |
       |0.0033 |0.0567 |0.0633 |       |
-------+-------+-------+-------+-------+
Thick  | 9     |22     |42     |73     |
       |0.1233 |0.3014 |0.5753 |0.66   |
       |0.9000 |0.5641 |0.6885 |       |
       |0.0300 |0.0733 |0.1400 |       |
-------+-------+-------+-------+-------+
ColTotl|10     |39     |61     |110    |
       |0.091  |0.355  |0.555  |       |
-------+-------+-------+-------+-------+
Test for independence of all factors
        Chi^2 = 82.96651 d.f.= 2 (p=0)
        Yates' correction not used
```

6.4 MANIPULATING AND ANALYZING CROSS-CLASSIFIED DATA

When you apply `crosstabs` to a data frame you get a multi-dimensional array whose elements are the counts and whose dimensions are the variables involved in the cross-tabulations. The first factor variable is the first, or row dimension, the second is the second, or column dimension, the third is the third dimension, etc. If you wish to do more than tabulate data, say compute means or sums of cross-classified data, you can apply functions to the elements of the array with the function `tapply`.

REGRESSION AND SMOOTHING FOR CONTINUOUS RESPONSE DATA

7

A large number of problems can be analyzed using linear regression, and even more with simple transformations.

REGRESSION AND SMOOTHING FOR CONTINUOUS RESPONSE DATA

Regression is a tool for exploring relationships between variables. *Linear regression* explores relationships that are readily described by straight lines, or their generalization to many dimensions. A surprisingly large number of problems can be analyzed using the techniques of linear regression, and even more can be attacked by means of *transformations* of the original variables that result in linear relationships among the transformed variables. In recent years, the techniques themselves have been extended through the addition of robust methods and generalizations of the classical linear regression techniques. These generalizations allow familiar problems in categorical data analysis such as logistic and Poisson regression to be subsumed under the heading of the *generalized linear model* (GLM), while still further generalizations allow a predictor to be replaced by an arbitrary smooth function of the predictor in building a *generalized additive model* (GAM).

This chapter describes regression and smoothing in the case of a univariate, continuous response. We start with simple regression, that is, regression with a single predictor variable: fitting the model, examining the fitted models, and analyzing the residuals. We then examine multiple regression, varying models by adding and dropping terms as appropriate. Again, we examine the fitted models and analyze the residuals. We then consider the special case of *weighted regression*, which underlies many of the robust techniques and generalized regression methods.

One important reason for performing regression analysis is to get a model useful for prediction. The section Prediction with the Model describes how to use S-PLUS to obtain predictions from your fitted model, and the section Confidence Intervals describes how to obtain pointwise and simultaneous confidence intervals.

The classical linear regression techniques make several strong assumptions about the underlying data, and the data can fail to satisfy these assumptions in different ways—for example, the regression line may be thrown off by one or more outliers or the data may not be fitted well by any straight line. In the first case, we can bring *robust regression* methods into play; these minimize the effects of outliers while retaining the basic form of the linear model. Conversely, the robust methods are often useful in identifying outliers. In the second case, we can expand our notion of the linear model, either by adding polynomial terms to our straight line model, or by replacing one or more predictors by an arbitrary smooth function of the predictor, converting the

classical linear model into a generalized additive model (GAM).

Scatterplot smoothers are useful tools for fitting arbitrary smooth functions to a scatter plot of data points. The smoother summarizes the trend of the measured response as a function of the predictor variables. We describe several scatterplot smoothers available in S-PLUS, and describe how the smoothed values they return can be incorporated into additive models.

7.1 SIMPLE LEAST-SQUARES REGRESSION

Simple regression uses the method of least squares to fit a continuous, univariate response as a linear function of a single predictor variable. In the method of least squares, we fit a line to the data so as to minimize the sum of the squared residuals. Given a set of n observations y_i of the response variable corresponding to a set of values x_i of the predictor, and an arbitrary model $\hat{y} = \hat{f}(x)$, the ith *residual* is defined as the difference between the ith observation y_i and the fitted value $\hat{y}_i = \hat{f}(x_i)$, that is $r_i = y_i - \hat{y}_i$.

To do simple regression with S-PLUS, use the function `lm` (for *l*inear *m*odel) with a simple *formula* linking your chosen response variable to the predictor variable. In many cases, both the response and the predictor are components of a single data frame, which can be specified as the `data` argument to `lm`. For example, consider the air pollution data in the built-in data set `air`:

```
> air[,c(1,3)]
      ozone temperature
 1 3.448217          67
 2 3.301927          72
 3 2.289428          74
 4 2.620741          62
 5 2.843867          65
 . . .
```

A scatter plot of the data is shown in figure 7.1. From the scatter plot, we hypothesize a linear relationship between temperature and ozone concentration. We choose `ozone` as the response, and `temperature` as the single predictor. The choice of response and predictor variables is driven by the subject matter in which the data arise, rather than by statistical considerations.

To fit the model, use `lm` as follows:

```
> ozone.lm <- lm(ozone ~ temperature, data = air)
```

The first argument, `ozone ~ temperature`, is the formula specifying that the variable `ozone` is modeled as a function of `temperature`. The second

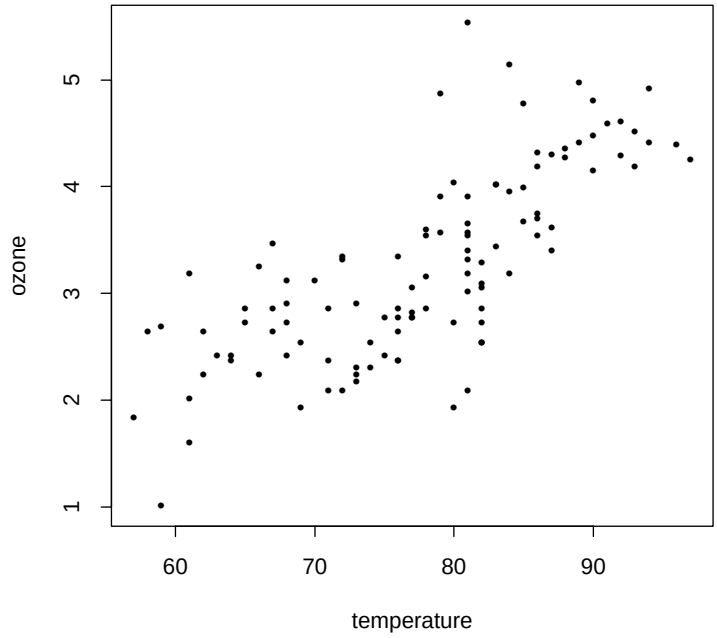

Figure 7.1: *Scatter plot of* `ozone` *against* `temperature`.

argument specifies that the data for the linear model is contained in the data frame `air`.

Use the `summary` function to obtain a summary of the fitted model:

```
> summary(ozone.lm)
Call: lm(formula = ozone ~ temperature)
Residuals:
   Min      1Q  Median      3Q   Max
 -1.49 -0.4258 0.02521 0.3636 2.044

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept)  -2.2260   0.4614    -4.8243   0.0000
temperature   0.0704   0.0059    11.9511   0.0000

Residual standard error: 0.5885 on 109 degrees of freedom
Multiple R-Squared: 0.5672
F-statistic: 142.8 on 1 and 109 degrees of freedom, the p-
value is 0
```

```
Correlation of Coefficients:
            (Intercept)
temperature -0.9926
```

The `Value` column under `Coefficients` gives the coefficients of the linear model, allowing us to read off the estimated regression line as follows:

`ozone` = -2.2260 + 0.0704 × `temperature`

The column headed `Std. Error` gives the estimated standard error for each coefficient. The `Multiple R-Squared` term from the `lm` summary tells us that the model explains about 57% of the variation in `ozone`. The `F-statistic` is the ratio of the mean square of the regression to the estimated variance; if there is no relationship between temperature and ozone, this ratio has an *F* distribution with 1 and 109 degrees of freedom. The ratio here is clearly significant, so the true slope of the regression line is probably not 0.

Diagnostic Plots for Linear Models

Suppose we have the linear model defined as follows:

```
> ozone.lm <- lm(ozone ~ temperature, data=air)
```

How good is the fitted linear regression model? Is temperature an adequate predictor of ozone concentration? Can we do better? Questions such as these are essential any time you try to explain data with a statistical model.

It is not enough to fit a model; you must also assess how well that model fits the data, being ready to modify the model or abandon it altogether if it does not satisfactorily explain the data.

The simplest and most informative method for assessing the fit is to look at the model graphically, using an assortment of plots that, taken together, reveal the strengths and weaknesses of the model. For example, a plot of the response against the fitted values gives a good idea of how well the model has captured the broad outlines of the data, while examining a plot of the residuals against the fitted values often reveals unexplained structure left in the residuals, which in a strong model should appear as nothing but noise. The default plotting method for `lm` objects provides these two plots, along with the following useful plots:

- *Square root of absolute residuals against fitted values.* This plot is useful in identifying outliers and visualizing structure in the residuals.
- *Normal quantile plot of residuals.* This plot provides a visual test of the assumption that the model's errors are normally distributed. If the ordered residuals cluster along the superimposed quantile-quantile line, you have strong evidence that the errors are indeed normal.

- *Residual-Fit spread plot*, or *r-f plot*. This plot compares the spread of the fitted values with the spread of the residuals. Since the model is an attempt to explain the variation in the data, you hope that the spread in the fitted values is *much* greater than that in the residuals.
- *Cook's distance plot*. Cook's distance is a measure of the influence of individual observations on the regression coefficients.

Calling `plot` as follows yields the six plots shown in figure 7.2:

```
> par(mfrow=c(2,3))
> plot(ozone.lm)
```

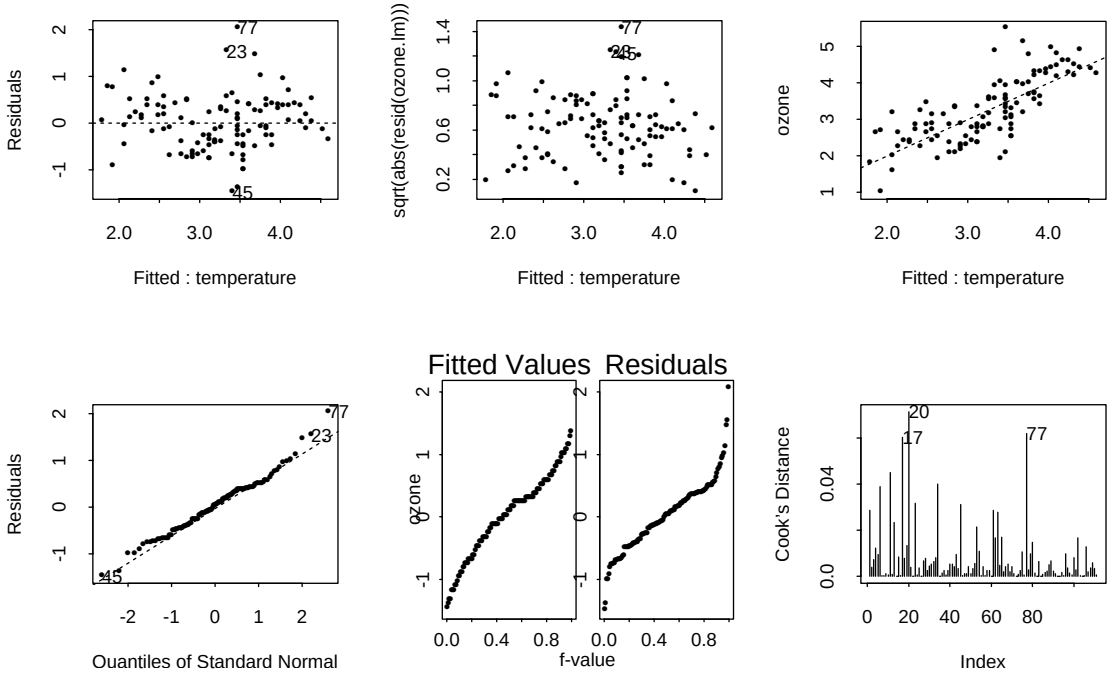

Figure 7.2: *Default plots for* `lm` *objects.*

The line $y = \hat{y}$ is shown as a dashed line in the third plot (far right of top row). In the case of simple regression, this line is visually equivalent to the regression line. The regression line appears to model the trend of the data reasonably well. The residuals plots (left and center, top row) show no obvious pattern, although five observations appear to be outliers. By default, as in figure 7.2, the three most extreme values are identified in each of the residuals plots and the Cook's distance plot. You can request a different number of points by using the `id.n` argument in the call to plot; for this

model, `id.n=5` is a good choice.

Another useful diagnostic plot is the normal plot of residuals (left plot, bottom row). The normal plot gives no reason to doubt that the residuals are normally distributed.

The r-f plot, on the other hand (middle plot, bottom row), shows a weakness in this model; the spread of the residuals is actually greater than the spread in the original data. However, if we ignore the five outlying residuals, the residuals are more tightly bunched than the original data.

The Cook's distance plot shows four or five heavily influential observations. As the regression line fits the data reasonably well, the regression is significant, and the residuals appear normally distributed, we feel justified in using the regression line as a way to estimate the ozone concentration for a given temperature. One important issue remains—the regression line explains only 57% of the variation in the data. We may be able to do somewhat better by considering the effect of other variables on the ozone concentration. See the section Multiple Regression for this further analysis.

At times, you are not interested in all of the plots created by the default plotting method. To view only those plots of interest to you, call `plot` with the argument `ask=T`. This call brings up a menu listing the available plots:

```
> par(mfrow=c(1,1))
> plot(ozone.lm, id.n=5, ask=T)
Make a plot selection (or 0 to exit):

1: plot: All
2: plot: Residuals vs Fitted Values
3: plot: Sqrt of abs(Residuals) vs Fitted Values
4: plot: Response vs Fitted Values
5: plot: Normal QQplot of Residuals
6: plot: r-f spread plot
7: plot: Cook's Distances
Selection:
Enter the number of the desired plot.
```

If you want to view all the plots, but want them all to appear in a full graphics window, do not set `par(mfrow=c(2,3))` before calling `plot`, and do not use the `ask=T` argument. Instead, before calling `plot`, call `par(ask=T)`. This tells S-PLUS to prompt you before displaying each additional plot.

7.2 MULTIPLE REGRESSION

You can construct linear models involving more than one predictor as easily in S-PLUS as models with a single predictor. In general, each predictor contributes a single *term* in the model formula; a single term may contribute more than one coefficient to the fit.

For example, consider the built-in data sets `stack.loss` and `stack.x`. Together, these data sets contain information on ammonia loss in a manufacturing process. The `stack.x` data set is a matrix with three columns representing three predictors: air flow, water temperature, and acid concentration. The `stack.loss` data set is a vector containing the response. To make our computations easier, combine these two data sets into a single data frame, then attach the data frame:

```
> stack.df <- data.frame(stack.loss, stack.x)
> stack.df
   stack.loss Air.Flow Water.Temp Acid.Conc.
 1         42       80         27         89
 2         37       80         27         88
 3         37       75         25         90
 . . .
> attach(stack.df)
```

For multivariate data, it is usually a good idea to view the data as a whole using the pairwise scatter plots generated by the `pairs` function:

```
> pairs(stack.df)
```

The resulting plot is shown in figure 7.3.

Call `lm` as follows to model `stack.loss` as a linear function of the three predictors:

```
> stack.lm <- lm(stack.loss ~ Air.Flow + Water.Temp +
+                Acid.Conc.)
> summary(stack.lm)
Call: lm(formula = stack.loss ~ Air.Flow + Water.Temp +
Acid.Conc.)
Residuals:
   Min     1Q  Median    3Q   Max
-7.238 -1.712 -0.4551 2.361 5.698

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept) -39.9197  11.8960    -3.3557   0.0038
   Air.Flow   0.7156   0.1349     5.3066   0.0001
 Water.Temp   1.2953   0.3680     3.5196   0.0026
```

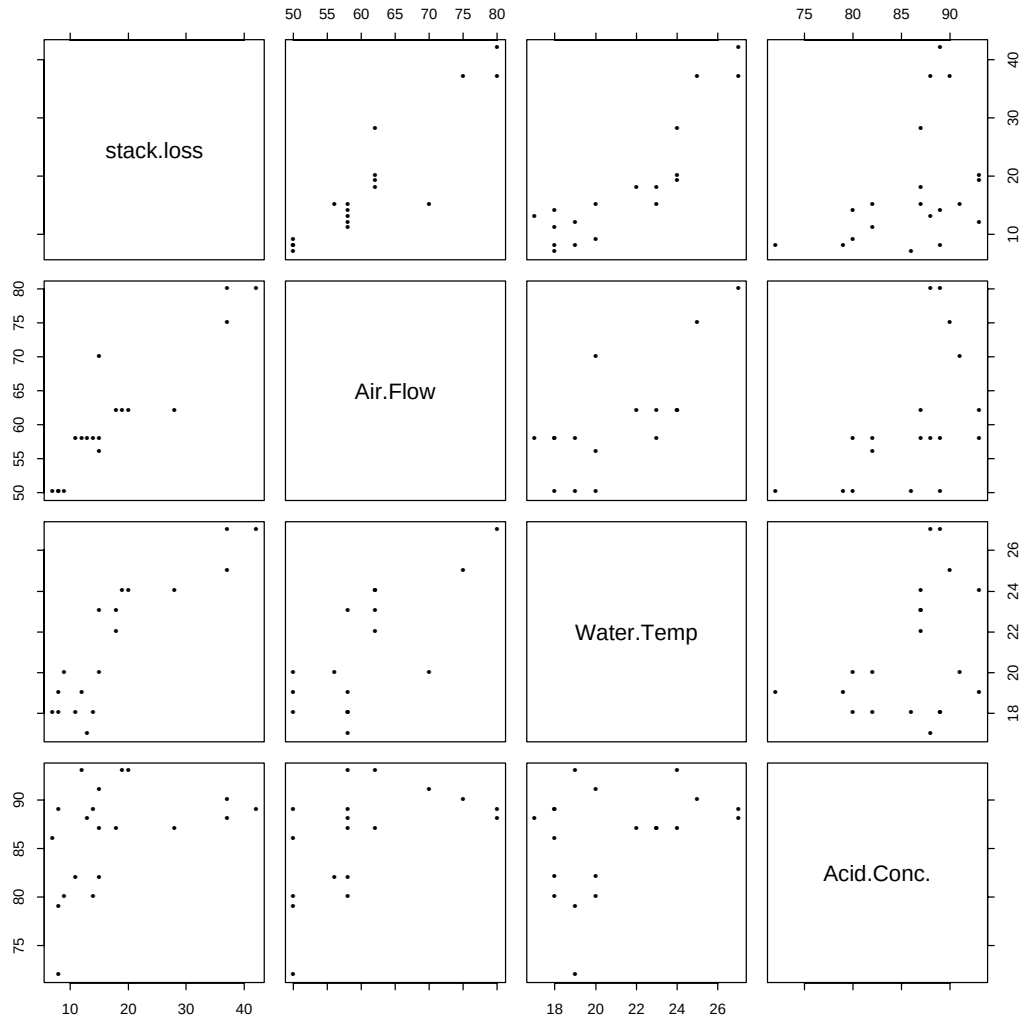

Figure 7.3: *Pairwise scatter plots of stack loss data.*

```
Acid.Conc.   -0.1521    0.1563    -0.9733    0.3440

Residual standard error: 3.243 on 17 degrees of freedom
Multiple R-Squared: 0.9136
F-statistic: 59.9 on 3 and 17 degrees of freedom, the p-
value is 3.016e-09

Correlation of Coefficients:
           (Intercept) Air.Flow Water.Temp
  Air.Flow  0.1793
```

```
Water.Temp -0.1489      -0.7356
Acid.Conc. -0.9016      -0.3389   0.0002
```

When the response is the first variable in the data frame, as in `stack.df`, and the desired model includes all the variables in the data frame, the name of the data frame itself can be supplied in place of the `formula` and `data` arguments:

```
> lm(stack.df)
Call:
lm(formula = stack.df)

Coefficients:
 (Intercept)  Air.Flow Water.Temp Acid.Conc.
   -39.91967 0.7156402   1.295286 -0.1521225

Degrees of freedom: 21 total; 17 residual
Residual standard error: 3.243364
```

We examine the default plots to assess the quality of the model (see figure 7.4):

```
> par(mfrow=c(2,3))
> plot(stack.lm, ask=F)
```

Both the line $y = \hat{y}$ and the residuals plots give support to the model. The multiple R^2 and F statistic also support the model. But would a simpler model suffice?

To find out, let's return to the summary of the `stack.lm` model. From the t values, and the associated p-values, it appears that both `Air.Flow` and `Water.Temp` contribute significantly to the fit. But can we improve the model by dropping the `Acid.Conc.` term? We explore this question further in section 7.3, Adding and Dropping Terms from a Linear Model.

7.3 ADDING AND DROPPING TERMS FROM A LINEAR MODEL

In section 7.2, Multiple Regression, we fitted a linear model with three predictors, of which only two appeared to be significant. Can we improve the model `stack.lm` by dropping one or more terms?

The `drop1` function takes a fitted model and returns an ANOVA table showing the effects of dropping in turn each term in the model:

```
> drop1(stack.lm)
Single term deletions
Model:
stack.loss ~ Air.Flow + Water.Temp + Acid.Conc.
```

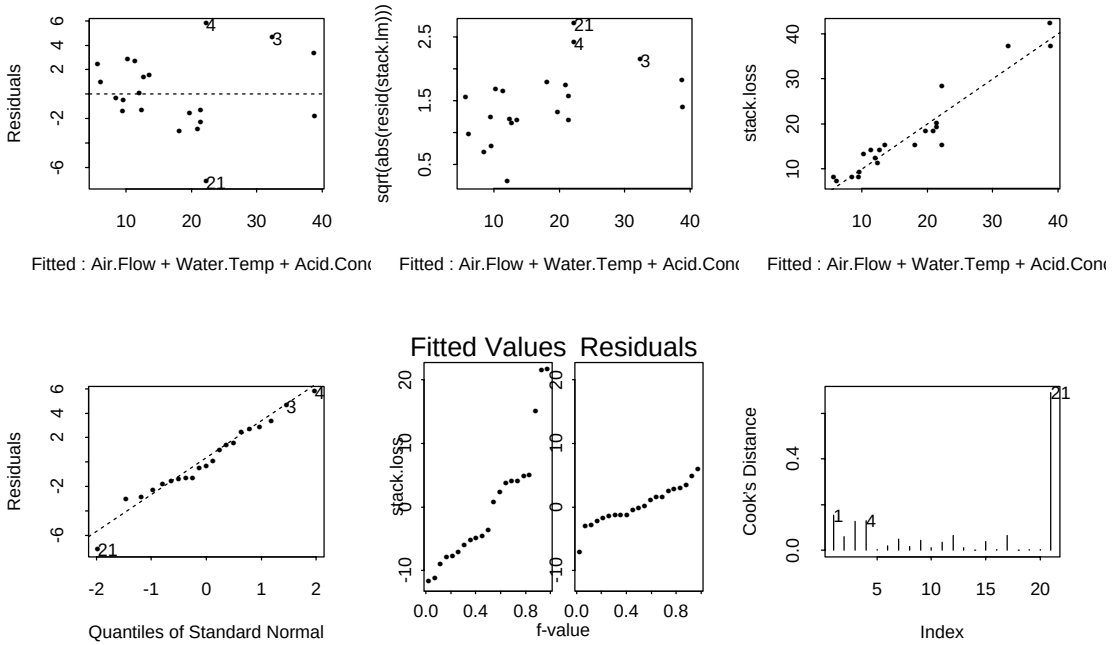

Figure 7.4: *Default plots of fitted model.*

```
            Df Sum of Sq      RSS       Cp
<none>                    178.8300 262.9852
Air.Flow     1  296.2281 475.0580 538.1745
Water.Temp   1  130.3076 309.1376 372.2541
Acid.Conc.   1    9.9654 188.7953 251.9118
```

The columns of the returned value show the degrees of freedom for each deleted term, the sum of squares corresponding to the deleted term, the residual sum of squares from the resulting model, and the C_p statistic for the terms in the reduced model.

The C_p statistic (actually, what is shown is the *AIC* statistic, the likelihood version of the C_p statistic—the two are related by the equation $AIC = \hat{\sigma}^2(C_p + n)$) provides a convenient criterion for determining whether a model is improved by dropping a term. If any term has a C_p statistic lower than that of the current model (shown on the line labeled <none>), the term with the lowest C_p statistic is dropped. If the current model has the lowest C_p statistic, the model is not improved by dropping any term. The regression literature discusses many other criteria for adding and dropping terms. See, for example, chapter 8 of Weisberg (1985).

In our example, the C_p statistic shown for `Acid.Conc.` is lower than that for the current model. So it is probably worthwhile dropping that term from the model:

```
> stack2.lm <- lm(stack.loss ~ Air.Flow + Water.Temp)
> stack2.lm
Call:
lm(formula = stack.loss ~ Air.Flow + Water.Temp)

Coefficients:
 (Intercept)  Air.Flow Water.Temp
   -50.35884 0.6711544   1.295351

Degrees of freedom: 21 total; 18 residual
Residual standard error: 3.238615
```

A look at the summary shows that we have retained virtually all the explanatory power of the more complicated model:

```
> summary(stack2.lm)

Call: lm(formula = stack.loss ~ Air.Flow + Water.Temp)
Residuals:
    Min     1Q Median     3Q    Max
 -7.529 -1.75 0.1894 2.116 5.659

Coefficients:
               Value Std. Error  t value Pr(>|t|)
(Intercept) -50.3588   5.1383    -9.8006   0.0000
   Air.Flow   0.6712   0.1267     5.2976   0.0000
 Water.Temp   1.2954   0.3675     3.5249   0.0024

Residual standard error: 3.239 on 18 degrees of freedom
Multiple R-Squared: 0.9088
F-statistic: 89.64 on 2 and 18 degrees of freedom, the p-
value is 4.382e-10

Correlation of Coefficients:
            (Intercept) Air.Flow
  Air.Flow -0.3104
Water.Temp -0.3438     -0.7819
```

The residual standard error has fallen, from 3.243 to 3.239, while the multiple R^2 has decreased only slightly from 0.9136 to 0.9088.

We create the default set of diagnostic plots as follows:

```
> par(mfrow=c(2,3))
> plot(stack2.lm, ask=F)
```

These plots, shown in figure 7.5, support the simplified model.

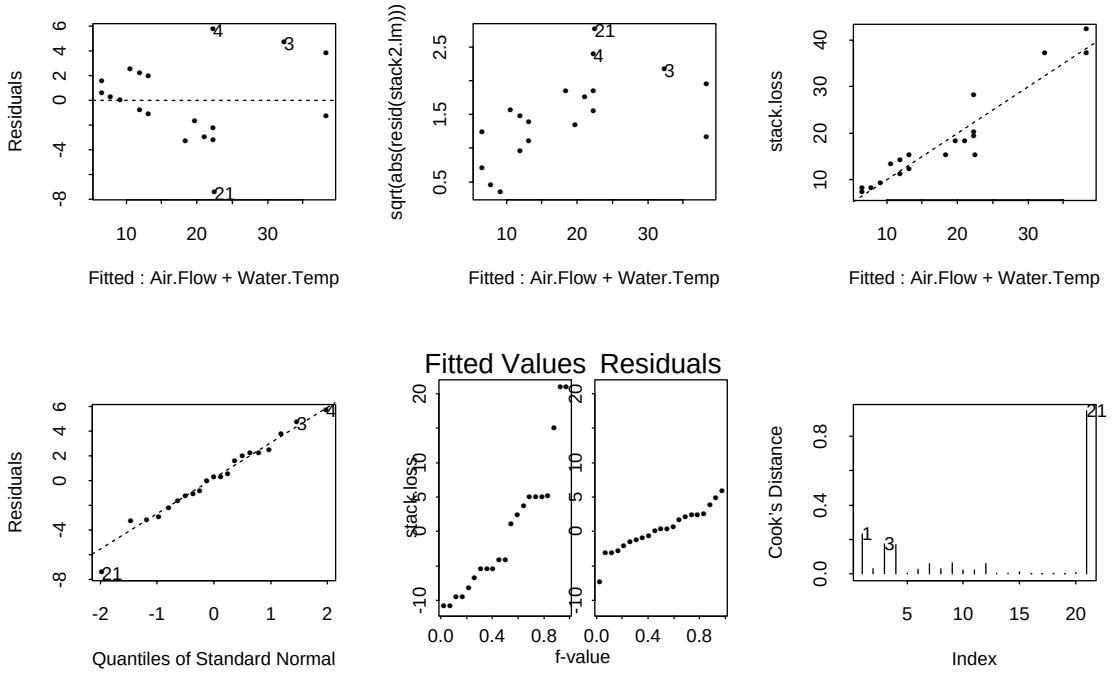

Figure 7.5: *Diagnostic plots for simplified model.*

We turn next to the opposite problem—adding terms to an existing model. Our first linear model hypothesized a relationship between temperature and atmospheric ozone, based on a scatter plot showing an apparent linear relationship between the two variables. The `air` data set containing the two variables `ozone` and `temperature` also includes two other variables, `radiation` and `wind`. Pairwise scatter plots for all the variables can be constructed using `pairs` as follows:

```
> pairs(air)
```

The resulting plot is shown in figure 7.6. The plot in the top row, third column of figure 7.6 corresponds to the scatter plot shown in figure 7.1.

From the pairwise plots, it appears that the `ozone` varies somewhat linearly with each of the variables `radiation`, `temperature`, and `wind`, and the dependence on `wind` has a negative slope.

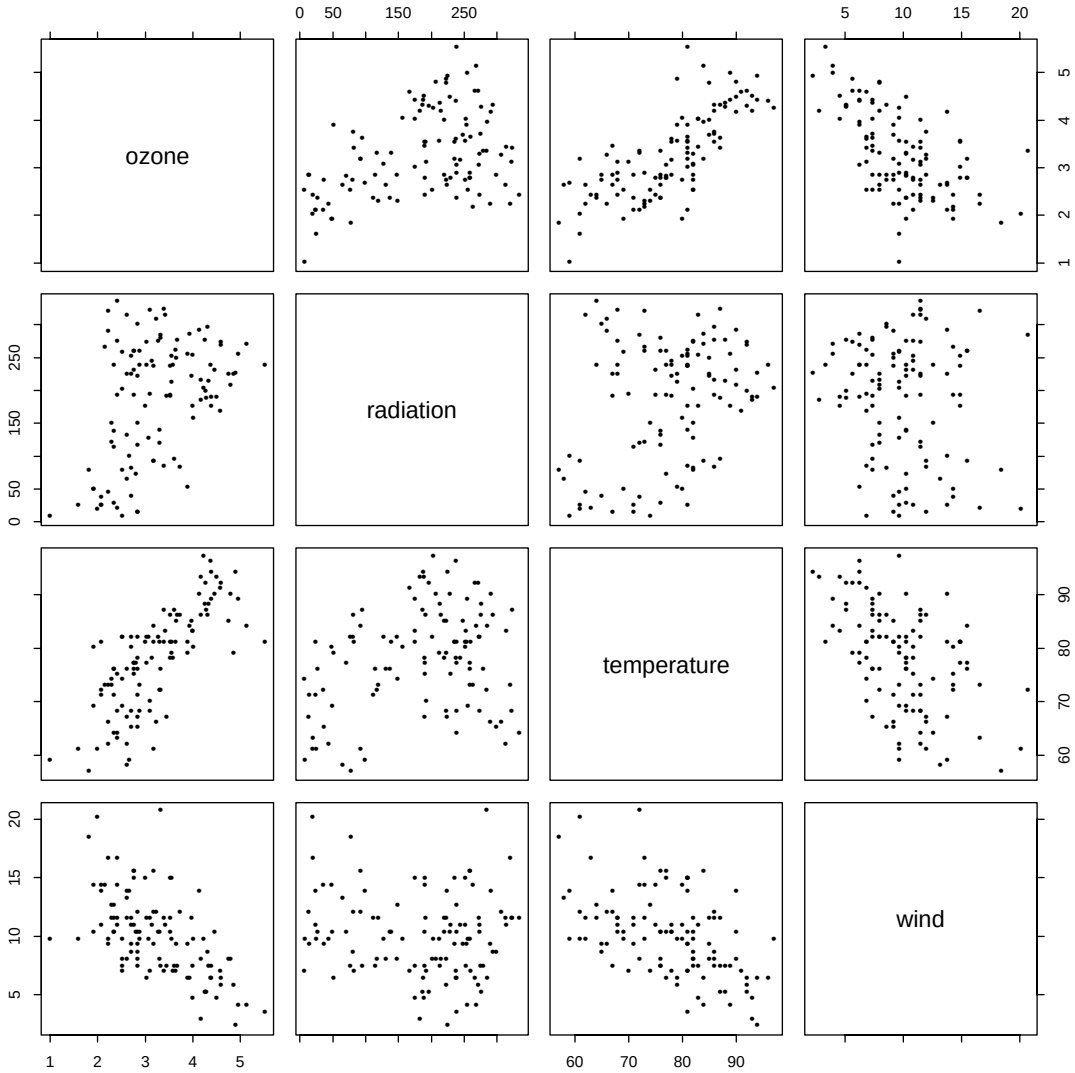

Figure 7.6: *Pairwise scatter plots for* `ozone` *data.*

We can use the `add1` function to add the terms `wind` and `radiation` in turn to our previously fitted model:

```
> ozone.add1 <- add1(ozone.lm, ~ temperature + wind +
radiation)
> ozone.add1
Single term additions
```

```
Model:
ozone ~ temperature
          Df Sum of Sq      RSS       Cp
   <none>               37.74698 39.13219
     wind  1  5.839621 31.90736 33.98517
radiation  1  3.839049 33.90793 35.98575
```

The first argument to `add1` is a fitted model object, the second a formula specifying the *scope*, that is, the possible choices of terms to be added to the model. No response need be specified in the formula supplied; the response must be the same as that in the fitted model. The returned object is an ANOVA table like that returned by `drop1`, showing the sum of squares due to the added term, the residual sum of squares of the new model, and the modified C_p statistic for the terms in the augmented model. Each row of the ANOVA table represents the effects of a single term added to the base model. In general, it is worth adding a term if the C_p statistic for that term is lowest among the rows in the table, including the base model term. In our example, we conclude that it is worthwhile adding the `wind` term.

Our choice of `temperature` as the original predictor in the model, however, was completely arbitrary. We can gain a truer picture of the effects of adding terms by starting from a simple intercept model:

```
> ozone0.lm <- lm(ozone ~ 1, data=air)
> ozone0.add1 <- add1(ozone0.lm, ~ temperature + wind
+ + radiation)
> ozone0.add1
Single term additions

Model:
ozone ~ 1
            Df Sum of Sq      RSS       Cp
     <none>               87.20876 88.79437
temperature  1  49.46178 37.74698 40.91821
       wind  1  31.28305 55.92571 59.09694
  radiation  1  15.53144 71.67732 74.84855
```

The obvious conclusion is that we should start with the `temperature` term, as we did originally.

7.4 CHOOSING THE BEST MODEL—STEPWISE SELECTION

Adding and dropping terms using `add1` and `drop1` is a useful method for selecting a model when only a few terms are involved, but it can quickly become tedious. The `step` function provides an automatic procedure for conducting stepwise model selection. Essentially what `step` does is automate the selection process implied in section 7.3—that is, it calculates the C_p statistics for the current model, as well as those for all reduced and augmented models, then adds or drops the term that reduces C_p the most. The `step` function requires an initial model, often constructed explicitly as an intercept-only model, such as the `ozone0.lm` model constructed in section 7.3. Because `step` calculates augmented models, it requires a `scope` argument, just like `add1`.

For example, suppose we want to find the "best" model involving the stack loss data, we could create an intercept-only model and then call `step` as follows:

```
> stack0.lm <- lm(stack.loss ~ 1, data = stack.df)
> step(stack0.lm, ~ Air.Flow + Water.Temp + Acid.Conc.)
Start:  AIC= 2276.162
 stack.loss ~ 1

Single term additions

Model:
stack.loss ~ 1

scale:  103.4619

            Df Sum of Sq      RSS      Cp
    <none>               2069.238 2276.162
  Air.Flow  1  1750.122  319.116  732.964
Water.Temp  1  1586.087  483.151  896.998
Acid.Conc.  1   330.796 1738.442 2152.290

Step: AIC= 732.9637
 stack.loss ~ Air.Flow

Single term deletions
```

```
Model:
stack.loss ~ Air.Flow

scale:  103.4619

         Df Sum of Sq      RSS       Cp
  <none>               319.116  732.964
Air.Flow  1  1750.122 2069.238 2276.162
Single term additions

Model:
stack.loss ~ Air.Flow

scale:  103.4619

           Df Sum of Sq      RSS       Cp
    <none>              319.1161 732.9637
Water.Temp  1  130.3208 188.7953 809.5668
Acid.Conc.  1    9.9785 309.1376 929.9090
Call:
lm(formula = stack.loss ~ Air.Flow, data = stack.df)

Coefficients:
 (Intercept) Air.Flow
   -44.13202 1.020309

Degrees of freedom: 21 total; 19 residual
Residual standard error (on weighted scale): 4.098242
```

The value returned by `step` is an object of class "`lm`", and the final result appears in exactly the same form as the output of `lm`. However, by default, `step` displays the output of each step of the selection process. You can turn off this display by calling `step` with the `trace=F` argument:

```
> step(stack0.lm, ~ Air.Flow + Water.Temp + Acid.Conc.,
+ trace=F)
Call:
lm(formula = stack.loss ~ Air.Flow, data = stack.df)

Coefficients:
 (Intercept) Air.Flow
   -44.13202 1.020309
```

```
Degrees of freedom: 21 total; 19 residual
Residual standard error (on weighted scale): 4.098242
```

7.5 UPDATING MODELS

We built our alternate model for the stack loss data by explicitly constructing a second call to `lm`. For models involving only one or two predictors, this is not usually too burdensome. However, if you are looking at many different combinations of many different predictors, constructing the full call repeatedly can be tedious.

The `update` function provides a convenient way for you to fit new models from old models, by specifying an *updated* formula or other arguments. For example, we could create the alternate model `stack2.lm` using `update` as follows:

```
> stack2a.lm <- update(stack.lm, .~. - Acid.Conc.,
+ data=stack.df)
> stack2a.lm
Call:
lm(formula = stack.loss ~ Air.Flow + Water.Temp, data
= stack.df)

Coefficients:
 (Intercept)  Air.Flow Water.Temp
   -50.35884 0.6711544   1.295351

Degrees of freedom: 21 total; 18 residual
Residual standard error: 3.238615
```

The first argument to `update` is always a model object, and additional arguments for `lm` are passed as necessary. The `formula` argument typically makes use of the "`.`" notation on either side of the "~". The "`.`" indicates "as in previous model." The "`-`" and "`+`" operators are used to delete or add terms. See chapter 2, Specifying Models in S-PLUS, for more information on formulas with `update`.

7.6 WEIGHTED REGRESSION

You can supply weights in fitting any linear model; this can sometimes improve the fit of models with repeated values in the predictor. Weighted regression is the appropriate method in those cases where it is known *a priori* that not all observations contribute equally to the fit.

For example, a software company with a successful training department wanted to estimate the revenue to be generated by an expanded training

schedule. For previous courses, the company had the data shown in table 7.1 concerning the number of courses per month and revenue generated.

Table 7.1: *Revenue and courses per month.*

Courses per month	Revenue
2	10030
2	7530
2	10801
3	18005
3	15455
3	14986
3	13926
4	16104
4	19166
4	18578
5	27596

We create an S-PLUS data frame from the data in table 7.1 as follows:

```
> ncourse <- c(2,2,2,3,3,3,3,4,4,4,5)
> revenue <- scan()
 [1] 10030  7530 10801 18005 15455 14986 13926 16104
 [9] 19166 18578 27596
> courserev <- data.frame(revenue, ncourse)
> courserev
   revenue ncourse
1   10030       2
2    7530       2
3   10801       2
4   18005       3
5   15455       3
6   14986       3
7   13926       3
8   16104       4
```

```
 9   19166       4
10   18578       4
11   27596       5
```

As a first look at the data, we fit a simple regression model as follows:

```
> course.lm <- lm(courserev)
> course.lm
Call:
lm(formula = courserev)

Coefficients:
 (Intercept)  ncourse
   -650.3113 5123.726

Degrees of freedom: 11 total; 9 residual
Residual standard error: 2131.713
```

We then look at the default plots, shown in figure 7.7:

```
> par(mfrow=c(2,3))
> plot(course.lm)
```

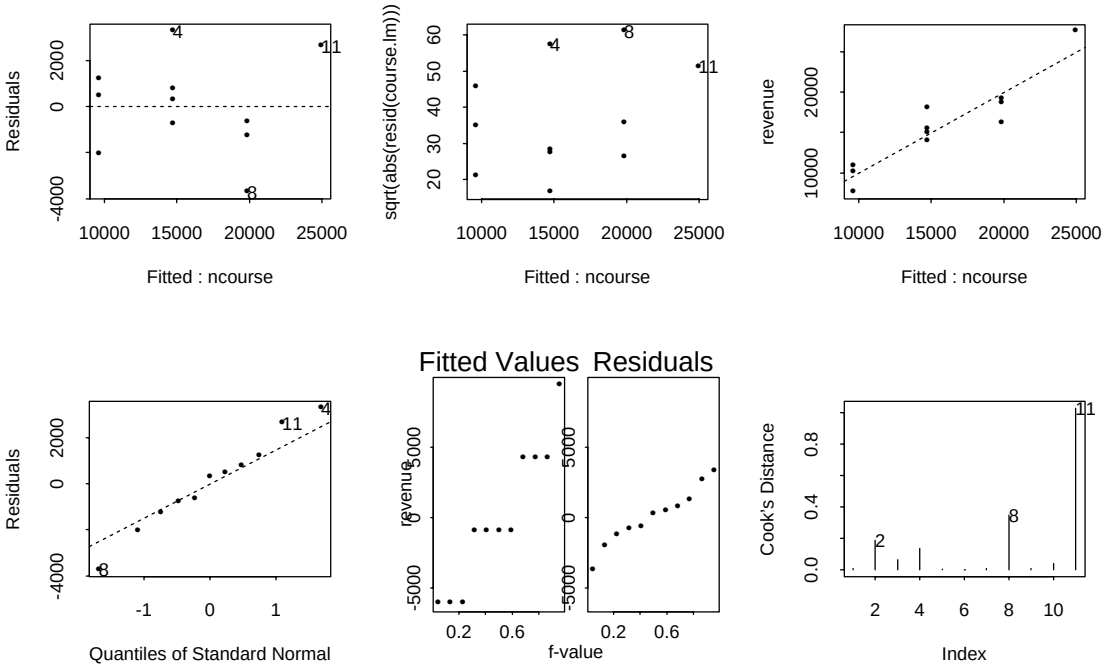

Figure 7.7: *Default diagnostic plots for unweighted course revenue model.*

Overall, the model seems to match the data reasonably well, but the residuals look to be increasing with the predictor. A weighted regression can help in this case.

Weights are specified by supplying a non-negative vector as the `weights` argument to `lm`. We weight with the number of observations for each value of the predictor—this gives a higher weight to the lone observation for 5 courses:

```
> course1.lm <- lm(revenue ~ ncourse, weights=c(1/3, 1/3,
+ 1/3, 1/4, 1/4, 1/4, 1/4, 1/3, 1/3, 1/3, 1))
> course1.lm
Call:
lm(formula = revenue ~ ncourse, weights = c(1/3, 1/3,
1/3, 1/4, 1/4, 1/4, 1/4, 1/3, 1/3, 1/3, 1))

Coefficients:
 (Intercept)  ncourse
   -2226.167 5678.333

Degrees of freedom: 11 total; 9 residual
Residual standard error (on weighted scale): 1297.942
```

The plots of the weighted regression show again a reasonable fit to the data and a less obvious pattern to the residuals.

7.7 PREDICTION WITH THE MODEL

Much of the value of a linear regression model is that, if it accurately models the underlying phenomenon, it can provide reliable *predictions* about the response for a given value of the predictor. The `predict` function takes a fitted model object and a data frame of new data, and returns a vector corresponding to the predicted response. The variable names in the new data must correspond to those of the original predictors; the response may or may not be present, but if present is ignored.

For example, suppose we want to predict the atmospheric ozone concentration from the following vector of temperatures:

```
> newtemp <- c(60, 62, 64, 66, 68, 70, 72)
```

We can obtain the desired predictions using `predict` as follows:

```
> predict(ozone.lm, data.frame(temperature=newtemp))
        1        2        3        4        5        6
1.995822 2.136549 2.277276 2.418002 2.558729 2.699456
        7
2.840183
```

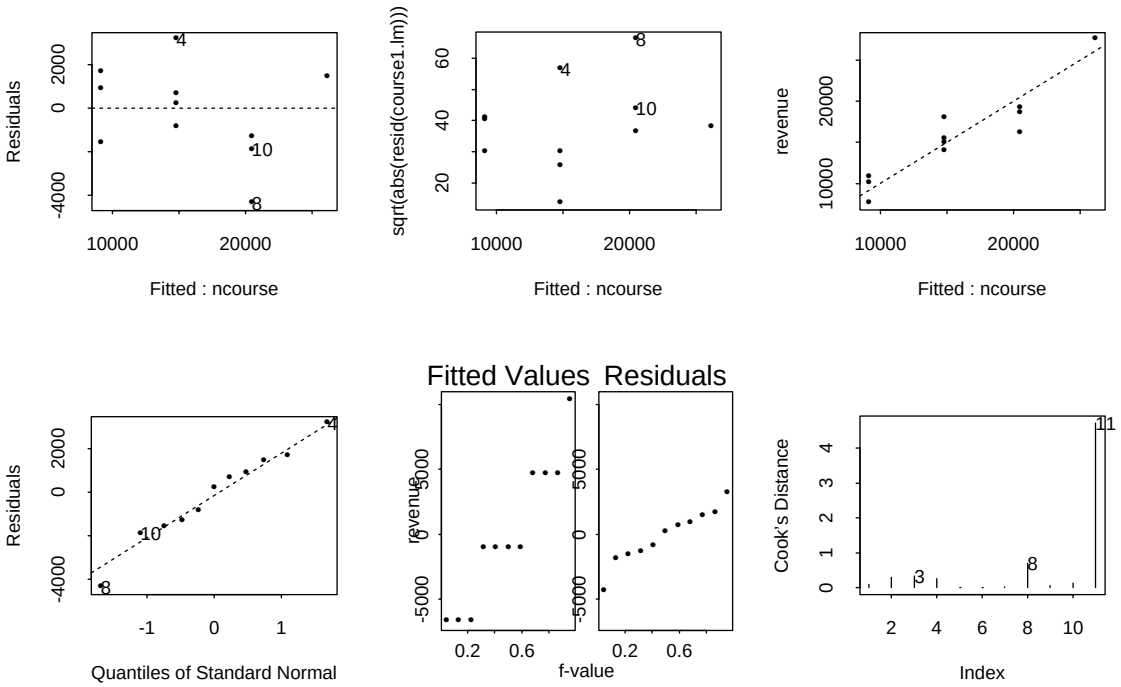

Figure 7.8: *Diagnostic plots for weighted course revenue model.*

The predicted values do not stand apart from the original observations.

You can use the `se.fit` argument to `predict` to obtain the standard error of the fitted value at each of the new data points. When `se.fit=T`, the output of `predict` is a list, with a `fit` component containing the predicted values and an `se.fit` component containing the standard errors:

```
> predict(ozone.lm, data.frame(temperature=newtemp),
se.fit=T)
$fit:
        1        2        3        4        5        6
 1.995822 2.136549 2.277276 2.418002 2.558729 2.699456
        7
 2.840183

$se.fit:
         1         2          3          4          5
 0.1187178 0.1084689 0.09856156 0.08910993 0.08027508
         6          7
 0.07228355 0.06544499
```

```
$residual.scale:
[1] 0.5884748

$df:
[1] 109
```

You can use this output list to compute pointwise and simultaneous confidence intervals for the fitted regression line. See section 7.8, Confidence Intervals, for details. See the `predict` help file for a description of the remaining components of the return list, `$residual.scale` and `$df`, as well as a description of `predict`'s other arguments.

7.8 CONFIDENCE INTERVALS

How reliable is the estimate produced by a simple regression? Provided the standard assumptions hold (that is, normal, identically distributed errors with constant variance σ), we can construct confidence intervals for each point on the fitted regression line based on the *t* distribution, and simultaneous confidence bands for the fitted regression line using the *F* distribution.

In both cases, we need the standard error of the fitted value, `se.fit`, which is computed as follows (Weisberg, 1985, p. 21):

$$\texttt{se.fit} = \hat{\sigma}\left(\frac{1}{n} + \frac{(x-\bar{x})^2}{\sum_i (x_i-\bar{x})^2}\right)^{\frac{1}{2}}$$

For a fitted object of class "`lm`", you can use the `predict` function as follows to calculate `se.fit`:

```
> predict(ozone.lm, se.fit=T)
$se.fit:
          1           2           3         4
 0.08460301 0.06544499 0.06015393 0.1084689
          5
 0.09377002 . . .
```

For any given point *x* in the predictor space, a 1 - α percent confidence interval for the fitted value corresponding to *x* is the set of values *y* such that

$$\hat{y} - t(\alpha, n-2) \times \texttt{se.fit} < y < \hat{y} + t(\alpha, n-2) \times \texttt{se.fit}$$

The `pointwise` function takes the output of `predict` (produced with the `se.fit=T` flag), and returns a list containing three vectors: the vector of lower bounds, the fitted values, and the vector of upper bounds giving the confidence intervals for the fitted values for the predictor:

```
> pointwise(predict(ozone.lm, se.fit=T))
$upper:
        1        2        3       4        5        6
 2.710169 3.011759 3.138615 2.42092 2.593475 2.250401
        7
 2.363895 . . .

$fit:
        1        2       3        4        5        6
 2.488366 2.840183 2.98091 2.136549 2.347639 1.925458
        7
 2.066185 . . .

$lower:
        1        2        3        4        5       6
 2.266563 2.668607 2.823205 1.852177 2.101803 1.600516
        7
 1.768476 . . .
```

The output from `pointwise` is suitable, for example, as input for the `error.bar` function. It is tempting to believe that the curves resulting from connecting all the upper points and all the lower points would give a confidence interval for the entire curve. This, however, is not the case; the resulting curve does not have the desired confidence level across its whole range. What is required instead is a *simultaneous* confidence interval, obtained by replacing the *t* distribution with the *F* distribution. An S-PLUS function for creating such simultaneous confidence intervals (and by default plotting the result) can be defined as follows:

```
"confint.lm"<-
function(object, alpha = 0.05, plot.it = T, ...)
{
        f <- predict(object, se.fit = T)
        p <- length(coef(object))
        fit <- f$fit
        adjust <- (p * qf(1 - alpha, p, length(fit) -
                p))^0.5 * f$se.fit
        lower <- fit - adjust
        upper <- fit + adjust
```

```
        if(plot.it) {
                y <- fit + resid(object)
                plot(fit, y)
                abline(0, 1, lty = 2)
                ord <- order(fit)
                lines(fit[ord], lower[ord])
                lines(fit[ord], upper[ord])
                invisible(list(lower = lower, upper =
                        upper))
        }
        else list(lower = lower, upper = upper)
}
```

To see how it works, let's create a plot of our first model of the ozone data:

```
> confint.lm(ozone.lm)
```

The resulting plot is shown in figure 7.9.

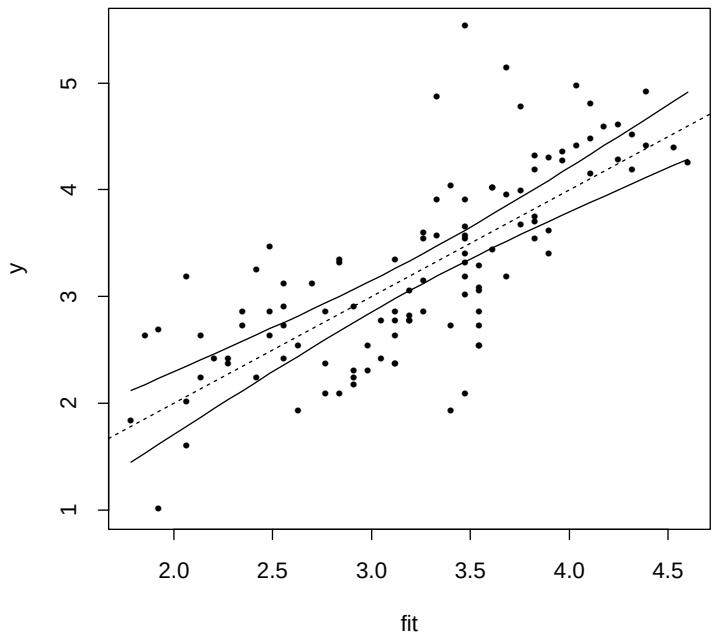

Figure 7.9: *Simultaneous confidence intervals for ozone data.*

7.9 ROBUST REGRESSION

Robust regression techniques are an important complement to the classical least-squares technique in that they provide answers similar to the least-

squares regression when the data are linear with normally distributed errors, but differ significantly from the least-squares fit when the errors don't satisfy the normality conditions or when the data contain significant outliers. S-PLUS includes several robust regression techniques. These robust techniques are not yet seamlessly integrated into the standard model structure, so using them is somewhat different from fitting a classical model with lm. This section describes three of these robust techniques; we recommend the first, least trimmed squares regression, because it is robust not only with respect to outliers but also to high leverage points, which are points in the model matrix with excessive influence on the result.

Least Trimmed Squares Regression

Least trimmed squares regression (LTS) regression, introduced by Rousseeuw, 1984, is a highly robust method for fitting a linear regression model. The LTS estimate $\hat{\boldsymbol{\beta}}_{LTS}$ minimizes the sum of the q smallest squared residuals

$$\sum_{i=1}^{q} r_{(i)}^2(\boldsymbol{\beta}) \tag{7.1}$$

where $r_{(i)}(\boldsymbol{\beta})$ is the ith ordered residual. The value of q is often set to be slightly larger than half of n.

By contrast, the ordinary least squares estimate $\hat{\boldsymbol{\beta}}_{LS}$ minimizes the sum of all of the squared residuals.

$$\sum_{i=1}^{n} r_i^2(\boldsymbol{\beta}) \tag{7.2}$$

The least squares estimator lacks robustness because a single "observation" (y_i, $\mathbf{x}_i^T$) can cause $\hat{\boldsymbol{\beta}}_{LS}$ to take on *any* value. The same is true of M-estimators of regression, which are discussed in section 7.9.

To compute the least trimmed squares regression, use the `ltsreg` function. For example, for the stack loss data, we can compute the LTS estimate as follows:

```
> stack.lts <- ltsreg(stack.x, stack.loss)
> stack.lts
$coefficients:
 (Intercept)  Air Flow Water Temp  Acid Conc.
   -35.66978 0.7301577  0.3566785 0.006565819
```

```
$residuals:
 [1]  9.04248471  4.04905053  8.40006437  9.26849051
 [5] -0.01815256 -0.37483103  0.22909559  1.22909559
 [9] -0.45420019  0.37515288  0.31606051 -0.32069520
[13] -2.63797876 -2.06688124  0.15732218 -0.82298037
[17] -0.08773737 -0.13369810  0.50305761  2.10897972
[21] -8.17232057

$fitted.values:
 [1] 32.957515 32.950949 28.599936 18.731509 18.018153
 [6] 18.374831 18.770904 18.770904 15.454200 13.624847
[11] 13.683939 13.320695 13.637979 14.066881 7.842678
[16]  7.822980  8.087737  8.133698  8.496942 12.891020
[21] 23.172321

$objective:
[1] 0.1436481

$stock:
$stock[[1]]:
[1] 17 10 5 17 5 12 15

$births.n:
[1] 440
```

(You will probably get a slightly different answer each time you call `ltsreg`. Because the objective is hard to compute exactly, `ltsreg` uses a random algorithm.) Comparing the coefficients to those for the least-squares fit, we observe that the LTS values are noticeably different:

```
> coef(stack.lm)
 (Intercept)  Air.Flow Water.Temp Acid.Conc.
   -39.91967 0.7156402   1.295286 -0.1521225
> coef(stack.lts)
 (Intercept)  Air Flow Water Temp  Acid Conc.
   -35.66978 0.7301577  0.3566785 0.006565819
```

A plot of the residuals versus the fitted values for the two fits is also revealing:

```
> graphsheet()
> par(mfrow=c(1,2))
> plot(fitted(stack.lm), resid(stack.lm),
+ ylim=range(resid(stack.lts)))
```

```
> plot(fitted(stack.lts), resid(stack.lts))
```

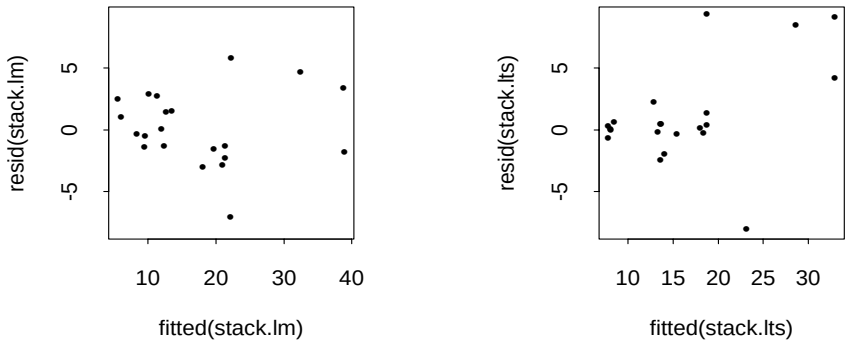

Figure 7.10: Residual plots for least-squares (*left*) and least trimmed squares regression.

The resulting plot is shown in figure 7.10. The plot on the left shows the residuals scattered with no apparent pattern; the plot on the right shows four clear outliers—three at the top and one at the bottom.

The *LTS* estimator has the highly attractive robustness property that its *breakdown point* is approximately 1/2 (if q is the right fraction of n). The *breakdown point* of a regression estimator is the largest fraction of data which may be replaced by arbitrarily large values without making the Euclidean norm $\|\hat{\boldsymbol{\beta}}\|$ of the resulting estimate tend to ∞. The Euclidean norm is defined as follows: $\|\hat{\boldsymbol{\beta}}\|^2 = \sum_{i=1}^{p} \hat{\boldsymbol{\beta}}_i^2$.

To illustrate the concept of breakdown point, consider the simple problem of estimating location, where often the estimator is the sample mean $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. The breakdown point of the mean estimator is 0, since if any single $y_i \to \pm\infty$, then $\bar{y} \to \pm\infty$. On the other hand, the sample median has breakdown point approximately 1/2, since, taking the case of an odd sample size n for convenience, one can move (n - 1)/2 of the observations y_i to ±∞ without taking the median to ±∞.

Any estimator with breakdown point approximately 1/2 is called a *high breakdown point* estimator. Thus, the *LTS* estimator is a high breakdown

point regression estimator.

The high breakdown point of the *LTS* estimator means that the values $\mathbf{x}_i^T\hat{\boldsymbol{\beta}}_{LTS}$, $i = 1, ..., n$, fit the bulk of the data well, even when the bulk of the data may consist of only somewhat more than 50% of the data. Correspondingly, the residuals $r_i(\hat{\boldsymbol{\beta}}_{LTS})=y_i-\mathbf{x}_i^T\hat{\boldsymbol{\beta}}_{LTS}$ will reveal the outliers quite clearly. Least squares residuals $r_i(\hat{\boldsymbol{\beta}}_{LS})=y_i-\mathbf{x}_i^T\hat{\boldsymbol{\beta}}_{LS}$ and M-estimate residuals $r_i(\hat{\boldsymbol{\beta}}_{M})=y_i-\mathbf{x}_i^T\hat{\boldsymbol{\beta}}_{M}$ often fail to reveal problems in the data. This can be illustrated as follows.

First construct an artificial data set with 60 percent of the data scattered about the line $y_i = x_i$ and the remaining 40 percent in an outlying cluster centered at (6, 2).

```
> set.seed(14) #set the seed to reproduce this example
> x30 <- runif(30, .5, 4.5)
> e30 <- rnorm(30, 0, .2); y30 <- 2+x30+e30
> x20 <- rnorm(20, 6, .5); y20 <- rnorm(20, 2, .5)
> x <- c(x30, x20)
> y <- c(y30, y20)
```

Plot the data, then fit and label 3 different regression lines: the ordinary least squares line, an M-estimate line, and the least trimmed squared residuals line.

```
> plot(x, y)
> abline(lm(y ~ x))
> text(5, 3.4, "LS")
> abline(rreg(x, y))
> text(4, 3.2, "M")
> abline(ltsreg(x, y))
> text(4, 6.5, "LTS")
```

The resulting plot is shown in figure 7.11.

The outlying points pull both the ordinary least squares line and the M-estimate away from the bulk of the data. Neither of these two fitting methods is robust to outliers in the *x* direction. (Such outliers are called *leverage points* in the literature.) The LTS line recovers the linear structure in the bulk of the data and essentially ignores the outlying cluster. In higher dimensions such outlying clusters are very hard to identify using classical regression techniques.

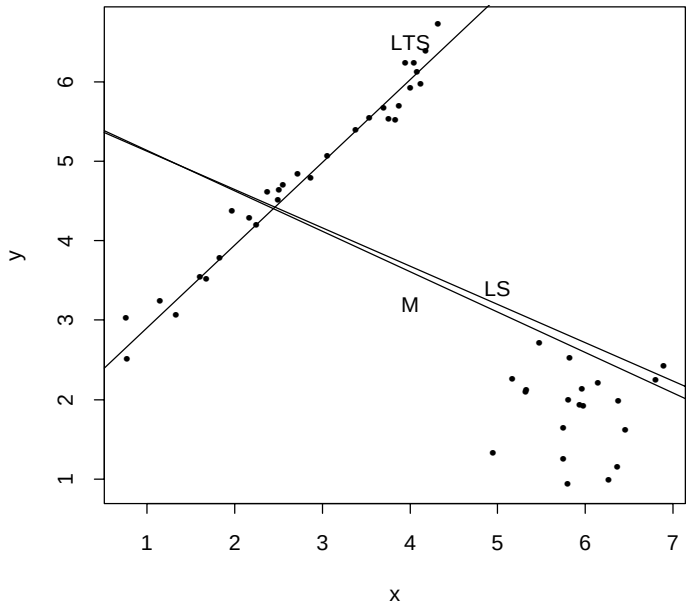

Figure 7.11: *Least trimmed squares regression, compared to least-squares and M-estimates.*

Least Absolute Deviation Regression

The idea of least absolute deviation regression, or L1 regression, is actually older than that of least-squares, but until the development of high-speed computers, it was too cumbersome to have wide applicability. S-PLUS has the function `l1fit` (note that the second character in the function name is the number "1", not the letter "l") for computing least absolute deviation regression. As its name implies, least absolute deviation regression finds the estimate $\hat{\boldsymbol{\beta}}_{L1}$ that minimizes the sum of the absolute values of the residuals

$$\sum_{i=1}^{n} \left| r_i(\boldsymbol{\beta}) \right| .$$

As an example, consider again the stack loss data. We construct the L1 regression using `l1fit` as follows:

```
> stack.l1 <- l1fit(stack.x, stack.loss)
> stack.l1
$coefficients:
 Intercept  Air Flow Water Temp  Acid Conc.
 -39.68984 0.8318844 0.5739114 -0.06086962
```

```
$residuals:
 [1]  5.06086922  0.00000000  5.42898512  7.63478470
 [5] -1.21739304 -1.79130435 -1.00000000  0.00000000
 [9] -1.46376634 -0.02029470  0.52752948  0.04057107
[13] -2.89855528 -1.80290544  1.18260598  0.00000000
[17] -0.42608732  0.00000000  0.48695821  1.61739087
[21] -9.48116493
```

Plotting the residuals against fitted values and comparing the plot to the corresponding plot for the least-squares fit shows the outliers clearly, though not so clearly as for the `ltsreg` plot.

```
> plot(fitted(stack.lm), resid(stack.lm))
> plot(stack.loss - resid(stack.l1), resid(stack.l1))
```

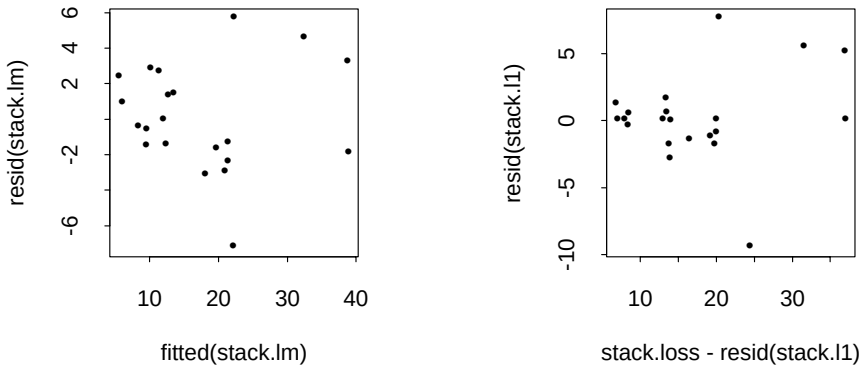

Figure 7.12: Residual plots for least-squares (*left*) and least absolute deviation regression.

The plot is shown in figure 7.12.

M-estimates of Regression

The M-estimator of regression was introduced by Huber, 1973. An M-estimate $\hat{\boldsymbol{\beta}}_M$ of regression is the $\boldsymbol{\beta}$ which minimizes

$$\sum_{i=1}^{n} \rho\left(\frac{r_i(\boldsymbol{\beta})}{\sigma}\right) \tag{7.1}$$

for a given ρ. Least squares corresponds to $\rho(x) = x^2$ and least absolute deviation regression corresponds to $\rho(x) = |x|$. Generally, although not in the two cases mentioned above, the value of $\hat{\boldsymbol{\beta}}_M$ is dependent on the value of σ, which is usually unknown.

Although M-estimates are protected against wild values in the response y, they are susceptible to high leverage points—that is, points which have quite different *x*-values relative to the other observations. In particular, a typographical error in an explanatory variable can have a dramatic affect on an M-estimate, while least trimmed squares regression handles this situation easily. One advantage of M-estimates is that they can be computed in much less time than least trimmed squares or other high-breakdown estimates.

M-estimates in S-PLUS are calculated using the `rreg` function to perform an *iteratively reweighted least-squares* fit. What this means is that S-PLUS calculates an initial fit (using traditional weighted least-squares, by default), then calculates a new set of weights based on the results of the initial fit. The new weights are then used in another weighted least-squares fit, new weights are calculated, and so on, iteratively, until either some convergence criteria are satisfied or a specified maximum number of iterations is reached.

The only required arguments to `rreg` are `x`, the vector or matrix of explanatory variables, and `y`, the vector response:

```
> stack.M1 <- rreg(stack.x, stack.loss)
> stack.M1
$coefficients:
 (Intercept)  Air Flow Water Temp Acid Conc.
   -42.07438 0.8978265   0.731816 -0.1142602

$residuals:
 [1]   2.65838630  -2.45587390   3.72541082   6.78619020
 [5]  -1.75017776  -2.48199378  -1.52824862  -0.52824862
 [9]  -1.89068795  -0.03142924   0.99691253   0.61446835
[13]  -2.80290885  -1.27786270   2.17952419   0.83674360
[17]  -0.49471517   0.30510621   0.68755039   1.52911203
[21] -10.01211661

$w:
 [1] 0.87721539 0.91831885 0.77235329 0.41742415 0.95387576
 [6] 0.90178786 0.95897484 0.99398847 0.93525890 0.99958817
[11] 0.97640677 0.98691782 0.89529949 0.98052477 0.92540436
[16] 0.98897286 0.99387986 0.99933718 0.99574820 0.96320721
[21] 0.07204303
```

You control the choice of ρ in `rreg` by specifying a *weight* function as the `method` argument. There are eight weight functions built into S-PLUS; there is not yet a consensus on which is "best." See the `rreg` help file for details on the weight functions. The default weight function uses Huber's weight

function until convergence, then a bisquare weight function for two more iterations. The following call to `rreg` defines a simple weight function that corresponds to the least-squares choice $\rho = x^2$:

```
> stack.MLS <- rreg(stack.x,stack.loss,
+ method=function(u) 2*abs(u),iter=100)
Warning messages:
failed to converge in 100 steps in:
rreg(stack.x, stack.loss, method = function(u) ....
> stack.MLS$coef
 (Intercept)  Air Flow Water Temp Acid Conc.
   -39.70404 0.7165807   1.298218 -0.1561163
 > coef(stack.lm)
 (Intercept)  Air.Flow Water.Temp Acid.Conc.
   -39.91967 0.7156402   1.295286 -0.1521225
```

7.10 POLYNOMIAL REGRESSION

Thus far in this chapter, we've dealt with data sets for which the graphical evidence clearly indicated a linear relationship between the predictors and the response. For such data, the linear model is a natural and elegant choice, providing a simple and easily analyzed description of the data. But what about data that does *not* exhibit a linear dependence? For example, consider the scatter plot shown in figure 7.13. Clearly, there is *some* functional relationship between the predictor `E` (for Ethanol) and the response `NOx` (for Nitric Oxide), but just as clearly the relationship is not a straight line.

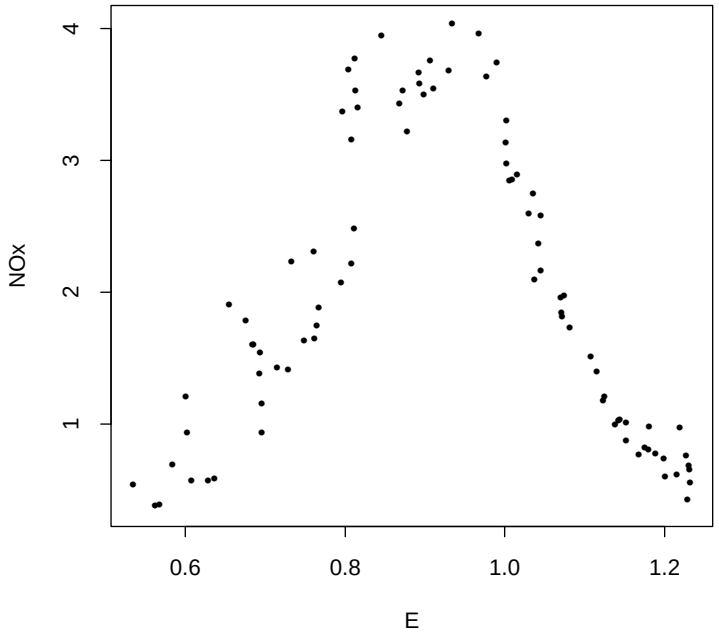

Figure 7.13: *Scatter plot showing nonlinear dependence.*

How should we model such data? One approach is to add polynomial terms to the basic linear model, then use least-squares techniques as before. The classical linear model (with the intercept term represented as the coefficient of a dummy variable X_0 of all 1s) is represented by an equation of the following form:

$$Y = \sum_{k=0}^{n} \beta_k X_k + \varepsilon \tag{7.1}$$

where the predictors X_k enter the equation as linear terms. More generally, classical linear regression techniques apply to any equation of the form

$$Y = \sum_{k=0}^{n} \beta_k Z_k + \varepsilon \tag{7.2}$$

where the Z_k are new variables formed as combinations of the original predictors. For example, consider a cubic polynomial relationship given by the following equation:

$$Y = \sum_{k=0}^{3} \beta_k X^k + \varepsilon \tag{7.3}$$

Taking $X = X_1 = X_2 = X_3$, we can convert this to the desired form by the following assignments:

$$\begin{aligned} X_0 &= Z_0 \\ X_1 &= Z_1 \\ X_2^2 &= Z_2 \\ X_3^2 &= Z_3 \end{aligned}$$

Once these assignments are made, the coefficients β_k can be determined as usual using the classical least-squares techniques.

To perform a polynomial regression in S-PLUS, use `lm` together with the `poly` function. Use `poly` on the right hand side of the `formula` argument to `lm` to specify the independent variable and degree of the polynomial. For example, consider the following made-up data:

```
x <- runif(100, 0, 100)
y <- 50 - 43*x + 31*x^2 - 2*x^3 + rnorm(100)
```

We can fit this as a polynomial regression of degree 3 as follows:

```
> xylm <- lm(y ~ poly(x, 3))
> xylm
Call:
lm(formula = y ~ poly(x, 3))
```

```
Coefficients:
(Intercept) poly(x, 3)1 poly(x, 3)2 poly(x, 3)3
  -329798.8    -3681644    -1738826   -333975.4

Degrees of freedom: 100 total; 96 residual
Residual standard error: 0.9463133
```

The coefficients appearing in the object `xylm` are the coefficients for the *orthogonal form* of the polynomial. To recover the simple polynomial form, use the function `poly.transform`:

```
> poly.transform(poly(x,3), coef(xylm))
     x^0       x^1      x^2       x^3
 49.9119 -43.01118 31.00052 -2.000005
```

These coefficients are very close to the exact values used to create `y`.

If the coefficients returned from a regression involving `poly` are so difficult to interpret, why not simply model the polynomial explicitly? That is, why not use the formula `y ~ x + x^2 + x^3` instead of the formula involving `poly`. In our example, there is little difference. However, in problems involving polynomials of higher degree, severe numerical problems can arise in the model matrix. Using `poly` avoids these numerical problems, because `poly` uses an orthogonal set of basis functions to fit the various "powers" of the polynomial.

As a further example of the use of `poly`, let us consider the `ethanol` data we saw at the beginning of this section. From figure 7.13, we are tempted by a simple quadratic polynomial. However, there is a definite upturn at each end of the data, so we are safer fitting a quartic polynomial, as follows:

```
> ethanol.poly <- lm(NOx ~ poly(E,degree=4))
> summary(ethanol.poly)

Call: lm(formula = NOx ~ poly(E, degree = 4))
Residuals:
     Min      1Q   Median     3Q   Max
 -0.8125 -0.1445 -0.02927 0.1607 1.017

Coefficients:
                       Value Std. Error  t value
         (Intercept)   1.9574   0.0393    49.8407
poly(E, degree = 4)1  -1.0747   0.3684    -2.9170
poly(E, degree = 4)2  -9.2606   0.3684   -25.1367
poly(E, degree = 4)3  -0.4879   0.3684    -1.3243
poly(E, degree = 4)4   3.6341   0.3684     9.8644
```

```
                     Pr(>|t|)
        (Intercept)   0.0000
poly(E, degree = 4)1   0.0045
poly(E, degree = 4)2   0.0000
poly(E, degree = 4)3   0.1890
poly(E, degree = 4)4   0.0000

Residual standard error: 0.3684 on 83 degrees of freedom
Multiple R-Squared: 0.8991
F-statistic: 184.9 on 4 and 83 degrees of freedom, the p-
value is 0

Correlation of Coefficients:
                     (Intercept) poly(E, degree = 4)1
poly(E, degree = 4)1 0
poly(E, degree = 4)2 0           0
poly(E, degree = 4)3 0           0
poly(E, degree = 4)4 0           0
                     poly(E, degree = 4)2
poly(E, degree = 4)1
poly(E, degree = 4)2
poly(E, degree = 4)3 0
poly(E, degree = 4)4 0
                     poly(E, degree = 4)3
poly(E, degree = 4)1
poly(E, degree = 4)2
poly(E, degree = 4)3
poly(E, degree = 4)4 0
> poly.transform(poly(E,4), coef(ethanol.poly))
      x^0       x^1      x^2       x^3     x^4
 174.3601 -872.2071 1576.735 -1211.219 335.356
```

The summary clearly shows the significance of the 4th order term.

7.11 SMOOTHING

Polynomial regression can be useful in many situations. However, the choice of terms is not always obvious, and small effects can be greatly magnified or lost completely by the wrong choice. Another approach to analyzing nonlinear data, attractive because it relies on the data to specify the form of the model, is to fit a curve to the data points *locally*, so that at any point the curve at that point depends only on the observations at that point and some specified neighboring points. Because such a fit produces an estimate of the

response that is less variable than the original observed response, the result is called a *smooth*, and procedures for producing such fits are called *scatterplot smoothers*. S-PLUS offers a variety of scatterplot smoothers:

- `loess.smooth` a *locally weighted regression* smoother.
- `smooth.spline` a cubic smoothing spline, with local behavior similar to that of kernel-type smoothers.
- `ksmooth` a kernel-type scatterplot smoother.
- `supsmu` a very fast variable span bivariate smoother.

Halfway between the global parameterization of a polynomial fit and the local, nonparametric fit provided by smoothers are the parametric fits provided by *regression splines*. Regression splines fit a continuous curve to the data by piecing together polynomials fit to different portions of the data. Thus, like smoothers, they are *local* fits. Like polynomials, they provide a parametric fit. In S-PLUS, regression splines can be used to specify the form of a predictor in a linear or more general model, but are not intended for top-level use.

Locally Weighted Regression Smoothing

In locally weighted regression smoothing, we build the smooth function $s(x)$ pointwise as follows:

1. Take a point, say x_0. Find the k nearest neighbors of x_0, which constitute a neighborhood $N(x_0)$. The number of neighbors k is specified as a percentage of the total number of points. This percentage is called the *span*.
2. Calculate the largest distance between x_0 and another point in the neighborhood:

$$\Delta(x_0) = \max_{N(x_0)} |x_0 - x_1|$$

3. Assign weights to each point in $N(x_0)$ using the tri-cube weight function:

$$W\left(\frac{|x_0 - x_1|}{\Delta(x_0)}\right)$$

where

$$W(u) = \begin{cases} (1-u^3)^3, & \text{for } 0 \le u < 1 \\ 0 & \text{otherwise} \end{cases}$$

4. Calculate the weighted least squares fit of y on the neighborhood $N(x_0)$. Take the fitted value $\hat{y}_0 = s(x_0)$.

5. Repeat for each predictor value.

Use the `loess.smooth` function to calculate a locally weighted regression smooth. For example, suppose we want to smooth the `ethanol` data. The following expressions produce the plot shown in figure 7.14:

```
> plot(E,NOx)
> lines(loess.smooth(E,NOx))
```

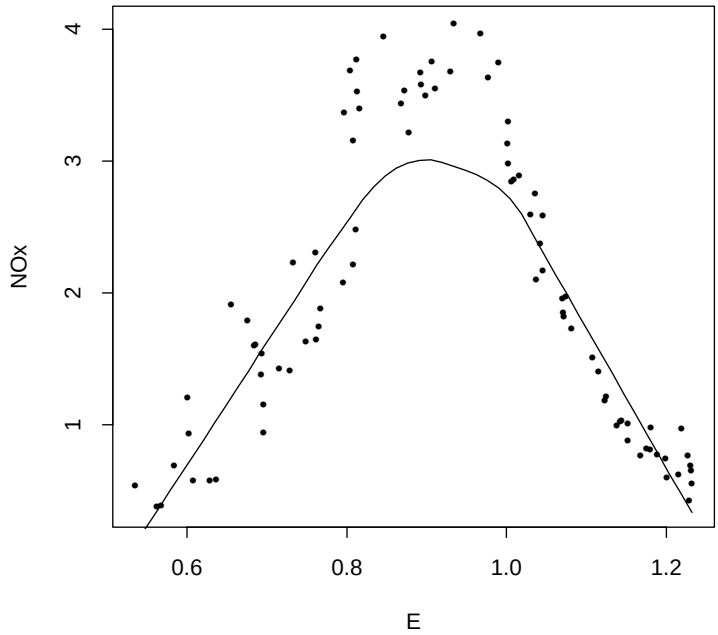

Figure 7.14: *Loess-smoothed ethanol data.*

The plot shown in figure 7.14 shows the default smoothing, which uses a span of 2/3. For most uses, you will want to specify a smaller span, typically in the range of 0.3 to 0.5.

Using the Super Smoother

With `loess`, the span is constant over the entire range of predictor values. However, a constant value will not be optimal if either the error variance or the curvature of the underlying function f varies over the range of x. An increase in the error variance requires an increase in the span whereas an increase in the curvature of f requires a decrease. Local cross-validation avoids this problem by choosing a span for the predictor values x_j based on only the

leave-one-out residuals whose predictor values x_i are in the neighborhood of x_j. The super smoother, `supsmu`, uses local cross-validation to choose the span. Thus, for one-predictor data, it can be a useful adjunct to `loess`.

For example, figure 7.15 shows the result of super smoothing the response `NOx` as a function of `E` in the `ethanol data` (dotted line) superimposed on a `loess` smooth. To create the plot, use the following commands:

```
> scatter.smooth(E,NOx, span=1/4)
> lines(supsmu(E,NOx), lty=2)
```

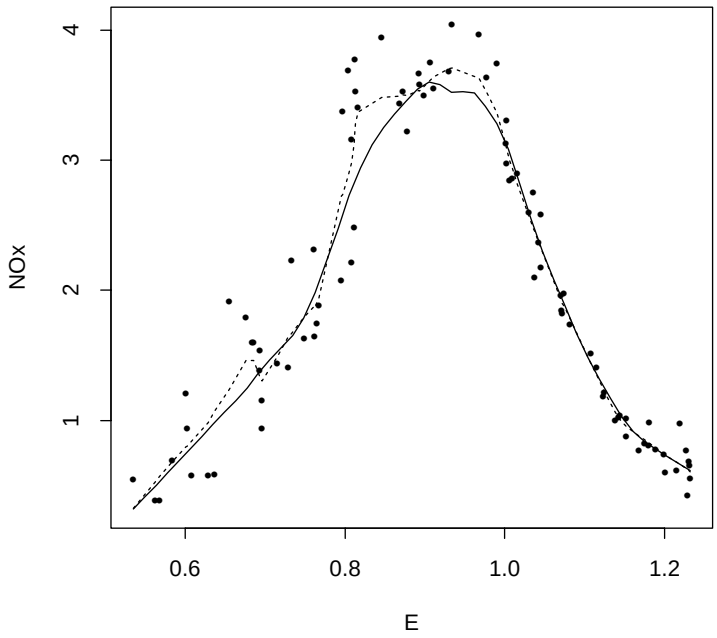

Figure 7.15: *Super Smoothed ethanol data (dotted line).*

Local Cross-Validation

Let $s(x|k)$ denote the linear smoother value at x when span k is used. We wish to choose $k = k(X)$ so as to minimize the mean squared error

$$e^2(k) = E_X Y[Y - s(X|k)]^2$$

where we are considering the joint random variable model for (X, Y). Since

$$E_X Y[Y - s(X|k)]^2 = E_X E_{Y|X}[Y - s(X|k)]^2$$

we would like to choose $k = k(x)$ to minimize

$$e_x^2(k) = E_Y|X = x[Y - s(X|k)]^2$$
$$= E_Y|X = x[Y - s(x|k)]^2 .$$

However, we have only the data (x_i, y_i), $i = 1, ..., n$, and not the true conditional distribution needed to compute $E_Y|X=x$, and so we cannot calculate $e_x^2(k)$. Thus we resort to cross-validation and try to minimize the cross-validation estimate of $e_x^2(k)$:

$$\hat{e}_{CV}^2(k) = \sum_{i=1}^{n} [y_i - s_{(i)}(x_i|k)]^2 .$$

Here $s_{(i)}(x_i|k)$ is the "leave-one-out" smooth at x_i, that is, $s_{(i)}(x_i|k)$ is constructed using all the data (x_j, y_j), $j = 1, ..., n$, *except* for (x_i, y_i), and then the resultant local least squares line is evaluated at x_i thereby giving $s_{(i)}(x|k)$. The *leave-one-out residuals*

$$r_{(i)}(k) = y_i - s_{(i)}(x_i|k)$$

are easily obtained from the ordinary residuals

$$r_i(k) = y_i - s(x_i|k)$$

using the standard regression model relation

$$r_{(i)}(k) = \frac{r_i(k)}{h_{ii}} .$$

Here h_{ii}, $i = 1, ..., n$, are the diagonals of the so-called "hat" matrix, $H = X(X^TX)^{-1}X^T$, where, for the case at hand of local straight-line regression, X is a 2-column matrix.

Using the Kernel Smoother

A kernel-type smoother is a type of local average smoother that, for each *target point* x_i in predictor space, calculates a weighted average y_i of the observations in a neighborhood of the target point:

$$\hat{y}_i = \sum_{j=1}^{n} w_{ij} y_j \qquad (7.4)$$

where

$$w_{ij} = \tilde{K}\left(\frac{x_i - x_j}{b}\right) = \frac{K\left(\frac{x_i - x_j}{b}\right)}{\sum_{k=1}^{n} K\left(\frac{x_i - x_k}{b}\right)}.$$

are weights which sum to one:

$$\sum_{j=1}^{n} w_{ij} = 1.$$

The function K used to calculate the weights is called a *kernel* function, which typically has the following properties:

- (a) $K(t) \geq 0$ for all t
- (b) $\int_{-\infty}^{\infty} K(t)dt = 1$
- (c) $K(-t) = K(t)$ for all t (symmetry)

Note that properties (a) and (b) are those of a probability density function. The parameter b is the *bandwidth* parameter, which determines how large a neighborhood of the target point is used to calculate the local average. A large bandwidth generates a smoother curve, while a small bandwidth generates a wigglier curve. Hastie and Tibshirani (1990), point out that the choice of bandwidth is much more important than the choice of kernel. To perform kernel smoothing in S-PLUS, use the `ksmooth` function. The kernels available in `ksmooth` are shown in table 7.1.

Of the available kernels, the default "`box`" kernel gives the crudest smooth. For most data, the other three kernels yield virtually identical smooths. We recommend "`triangle`" because it is the simplest and fastest to calculate.

The intuitive sense of the kernel estimate y_i is clear: Values of y_j such that x_j is close to x_i get relatively heavy weights, while values of y_j such that x_j is far from x_i get small or zero weight. The bandwidth parameter b determines the width of $K(t/b)$, and hence controls the size of the region around x_i for which y_j receives relatively large weights. Since bias increases and variance decreases with increasing bandwidth b, selection of b is a compromise between bias and variance in order to achieve small mean squared error. In practice this is usually done by trial and error. For example, we can compute a kernel

Table 7.1: *Kernels available for* `ksmooth`.

Kernel	Explicit form
"box"	$K_{\text{box}}(t) = \begin{cases} 1, & \lvert t\rvert \le 0.5 \\ 0, & \lvert t\rvert > 0.5 \end{cases}$
"triangle"[1]	$K_{\text{tri}}(t) = \begin{cases} 1-\lvert t\rvert/C, & \lvert t\rvert \le \frac{1}{C} \\ 0, & \lvert t\rvert > \frac{1}{C} \end{cases}$
"parzen"[2]	$K_{\text{par}}(t) = \begin{cases} (k_1 - t^2)/k_2, & \lvert t\rvert \le C_1 \\ (t^2/k_3) - k_4\lvert t\rvert + k_5, & C_1 < \lvert t\rvert \le C_2 \\ 0, & C_2 < \lvert t\rvert \end{cases}$
"normal"	$K_{\text{nor}}(t) = (1/\sqrt{2\pi}k_6)\ \exp(-t^2/2k_6^2)$
In convolution form, $K_{\text{tri}}(t) = K_{\text{box}} * K_{\text{box}}(t)$ In convolution form, $K_{\text{par}}(t) = K_{\text{tri}} * K_{\text{box}}(t)$ The constants shown in the explicit forms above are used to scale the resulting kernel so that the upper and lower quartiles occur at ±.25. Also, the is taken to be 1 and the dependence of the kernel on the bandwidth is suppressed.	

smooth for the ethanol data as follows:

```
> plot(E,NOx)
> lines(ksmooth(E,NOx, kernel="triangle", bandwidth=.2))
> lines(ksmooth(E,NOx, kernel="triangle", bandwidth=.1),
+ lty=2)
> legend(.54,4.1,c("bandwidth=.2", "bandwidth=.1"),
+ lty=c(1,2))
```

The resulting plot is shown in figure 7.16.

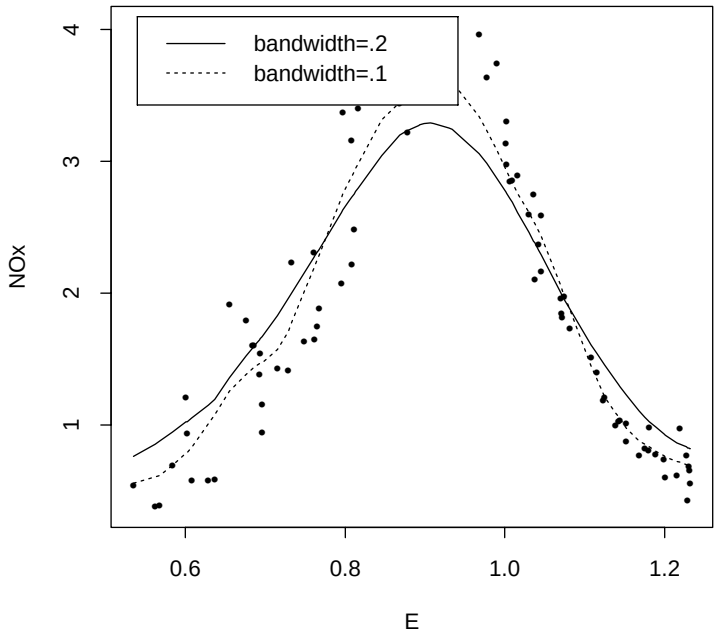

Figure 7.16: *Kernel smooth of ethanol data for two bandwidths.*

Smoothing Splines

A *cubic smoothing spline* behaves approximately like a kernel smoother, but it arises as the function $\hat{f}$ that minimizes the *penalized residual sum of squares* given by

$$PRSS = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \int (f''(t))^2 dt$$

over all functions with continuous first and integrable second derivatives. The parameter λ is the smoothing parameter, corresponding to the span in `loess` or `supsmu` or the bandwidth in `ksmooth`.

To generate a cubic smoothing spline in S-PLUS, use the function `smooth.spline` smooth to the input data:

```
> plot(E,NOx)
> lines(smooth.spline(E,NOx))
```

You can specify a different λ using the `spar` argument, although it is not intuitively obvious what a "good" choice of λ might be. When the data is normalized to have a minimum of 0 and a maximum of 1, and when all weights are equal to 1, λ=`spar`. More generally, the relationship is given by $\lambda=(\max(x)-\min(x))^3 \cdot \text{mean}(w) \cdot$ `spar`. You should either let S-PLUS choose the smoothing parameter, using either ordinary or generalized cross-validation, or supply an alternative argument, `df`, which specifies the *degrees of freedom* for the smooth. For example, to add a smooth with approximately 5 degrees of freedom to our previous plot, use the following:

```
> lines(smooth.spline(E,NOx, df=5), lty=2)
```

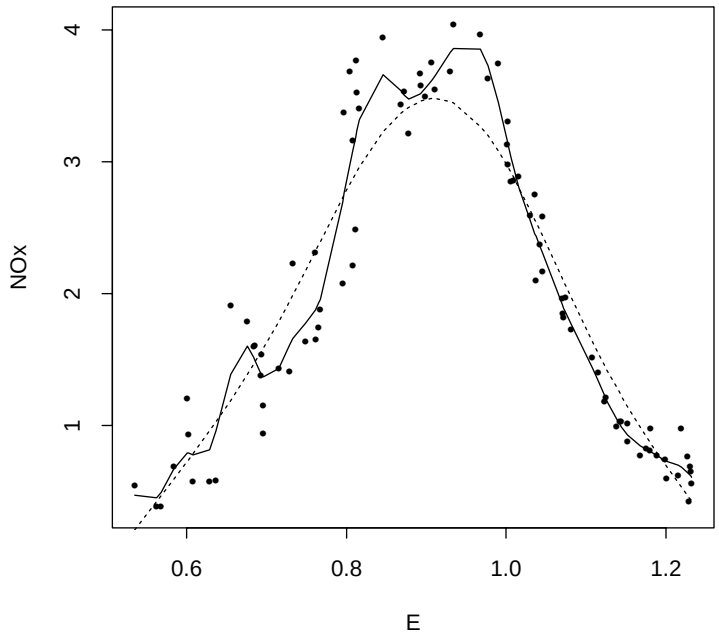

Figure 7.17: *Smoothing spline of ethanol data with cross-validation (solid line) and pre-specified degrees of freedom.*

The resulting plot is shown in figure 7.17.

Comparing Smoothers

The choice of a smoother is somewhat subjective. All the smoothers discussed in this section can generate reasonably good smooths; you might select one or another based on theoretical considerations or the ease with which one or another of the smoothing criteria can be applied. For a direct comparision of these smoothers, consider the artificial data constructed as follows:

```
> set.seed(14) # set the seed to reproduce this example
> e <- rnorm(200)
> x <- runif(200)
> y <- sin(2*pi*(1-x)2)+x*e
```

A "perfect" smooth would recapture the original signal, $f(x) = \sin(2\pi(1\text{-}x)^2)$, exactly. The following commands sort the input and calculate the *exact* smooth:

```
> sx <- sort(x)
> fx <- sin(2*pi*(1-sx)2)
```

The following commands create a scatter plot of the original data, then superimpose the exact smooth and smooths calculated using each of the smoothers described in this chapter:

```
> plot(x,y)
> lines(sx,fx)
> lines(supsmu(x,y),lty=2)
> lines(ksmooth(x,y),lty=3)
> lines(smooth.spline(x,y),lty=4)
> lines(loess.smooth(x,y),lty=5)
> legend(0,2,c("perfect", "supsmu", "ksmooth",
+ "smooth.spline", "loess"), lty=1:5)
```

The resulting plot is shown in figure 7.18. This comparison is crude, at best, because by default each of the smoothers does a different amount of smoothing. A fairer comparison would adjust the smoothing parameters to be roughly equivalent.

7.12 ADDITIVE MODELS

An *additive* model extends the notion of a linear model by allowing some or all linear functions of the predictors to be replaced by arbitrary smooth functions of the predictors. Thus, the standard linear model

$$Y = \sum_{i=0}^{n} \beta_i X_i + \varepsilon$$

is replaced by the additive model

$$Y = \alpha + \sum_{i=1}^{n} f_i(X_i) + \varepsilon \quad .$$

The standard linear regression model is a simple case of an additive model.

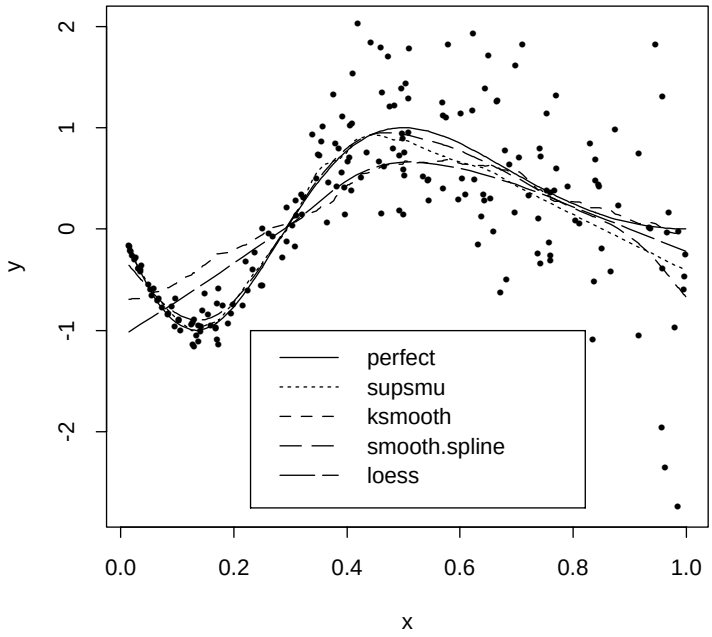

Figure 7.18: *Comparison of* S-PLUS *smoothers.*

Because the forms of the f_i are generally unknown, they are estimated using some form of scatterplot smoother.

To fit an additive model in S-PLUS, use the `gam` function, where `gam` stands for *generalized additive model*. You provide a formula which may contain ordinary linear terms as well as terms fit using any of the following:

- loess smoothers, using the `lo` function;
- smoothing spline smoothers, using the `s` function;
- natural cubic splines, using the `ns` function;
- *B*-splines, using the `bs` function;
- polynomials, using `poly`.

The three functions `ns`, `bs`, and `poly` result in *parametric* fits; additive models involving only such terms can be analyzed in the classical linear model framework. The `lo` and `s` functions introduce *nonparametric* fitting into the model. For example, the following call takes the `ethanol` data and models the response `NOx` as a function of the loess-smoothed predictor `E`:

```
> attach(ethanol)
> ethanol.gam <- gam(NOx ~ lo(E, degree=2))
```

```
> ethanol.gam
Call:
gam(formula = NOx ~ lo(E, degree = 2))

Degrees of Freedom: 88 total; 81.1184 Residual
Residual Deviance: 9.1378
```

In the call to `lo`, we specify that the smooth is to be locally quadratic by using the argument `degree = 2`. For data that is less obviously nonlinear, we would probably be satisfied with the default, which is locally linear fitting. The printed `gam` object closely resembles a printed `lm` object from linear regression—the call producing the model is shown, followed by the degrees of freedom and the *residual deviance* which serves the same role as the residual sum of squares in the linear model. The deviance is a function of the log-likelihood function, which is related to the probability mass function $f(y_i; \mu_i)$ for the observation y_i given μ_i. The log-likelihood for a sample of n observations is defined as follows:

$$l(\boldsymbol{\mu}; \boldsymbol{y}) = \sum_{i=1}^{n} \log f(y_i; \mu_i)$$

The deviance $D(\boldsymbol{y}; \boldsymbol{\mu})$ is then defined as

$$\frac{D(\boldsymbol{y}; \boldsymbol{\mu})}{\phi} = 2l(\boldsymbol{\mu}^*; \boldsymbol{y}) - 2l(\boldsymbol{\mu}; \boldsymbol{y})$$

where $\boldsymbol{\mu}^*$ maximizes the log-likelihood over $\boldsymbol{\mu}$ unconstrained, and ϕ is the *dispersion parameter*. For a continuous response with normal errors, as in the models we've been considering in this chapter, the dispersion parameter is just the variance σ^2, and the deviance reduces to the residual sum of squares. As with the residual sum of squares, the deviance can be made arbitrarily small by choosing an interpolating solution. As in the linear model case, however, we generally have a desire to keep the model as simple as possible. In the linear case, we try to keep the number of *parameters*, that is, the quantities estimated by the model coefficients, to a minimum. Additive models are generally *nonparametric*, but we can define for nonparametric models an *equivalent number of parameters*, which we would also like to keep as small as possible.

The equivalent number of parameters for `gam` models is defined in terms of *degrees of freedom*, or `df`. In fitting a parametric model, one degree of freedom is required to estimate each parameter. For an additive model with parametric terms, one degree of freedom is required for each coefficient the term

contributes to the model. Thus, for example, consider a model with an intercept, one term fit as a cubic polynomial, and one term fit as a quadratic polynomial. The intercept term contributes one coefficient and requires one degree of freedom, the cubic polynomial contributes three coefficients and thus requires three degrees of freedom, and the quadratic polynomial contributes two coefficients and requires two more degrees of freedom. Thus, the entire model has six parameters, and uses six degrees of freedom. A minimum of six observations is required to fit such a model. Models involving smoothed terms use both *parametric* and *nonparametric* degrees of freedom—parametric degrees of freedom result from fitting a linear (parametric) component for each smooth term, while the nonparametric degrees of freedom result from fitting the smooth after the linear part has been removed. The difference between the number of observations and the degrees of freedom required to fit the model is the *residual degrees of freedom*. Conversely, the difference between the number of observations and the residual degrees of freedom is the degrees of freedom required to fit the model, which is the equivalent number of parameters for the model.

The `summary` method for `gam` objects shows the residual degrees of freedom, the parametric and nonparametric degrees of freedom for each term in the model, together with additional information:

```
> summary(ethanol.gam)

Call: gam(formula = NOx ~ lo(E, degree = 2))
Deviance Residuals:
        Min         1Q      Median        3Q      Max
 -0.6814987 -0.1882066 -0.01673293 0.1741648 0.8479226

(Dispersion Parameter for Gaussian family taken to be
0.1126477 )

Null Deviance: 111.6238 on 87 degrees of freedom

Residual Deviance: 9.137801 on 81.1184 degrees of freedom

Number of Local Scoring Iterations: 1

DF for Terms and F-values for Nonparametric Effects

                  Df Npar Df   Npar F        Pr(F)
      (Intercept)  1
lo(E, degree = 2)  2     3.9 35.61398 1.110223e-16
```

The `Deviance Residuals` are, for Gaussian models, just the ordinary residuals $y_i - \hat{\mu}_i$. The `Null Deviance` is the deviance of the model consisting solely of the intercept term.

The `ethanol` data set contains a third variable, `C`, which measures the compression ratio of the engine. Figure 7.19 shows pairwise scatter plots for the three variables. Let's incorporate `C` as a linear term in our additive model:

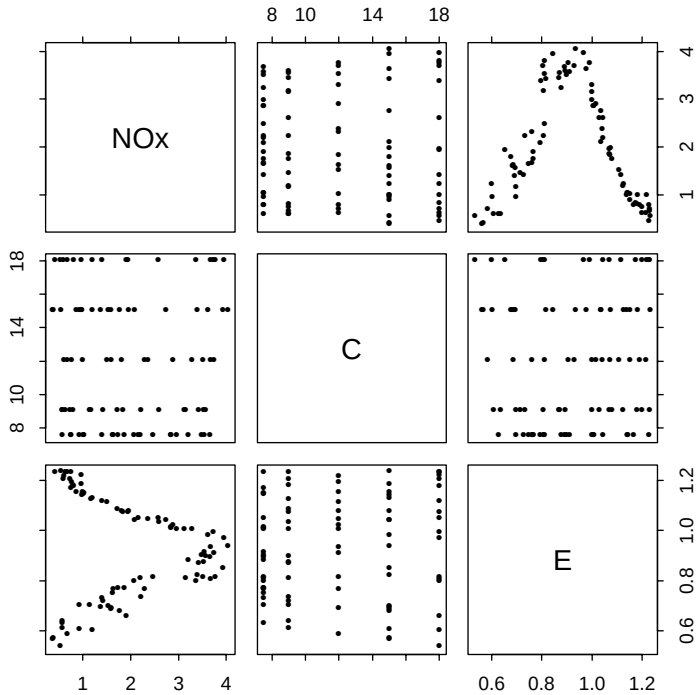

Figure 7.19: *Pairs plot of the* `ethanol` *data.*

```
> attach(ethanol)
> ethanol2.gam <- gam(NOx ~ C + lo(E, degree = 2))
> ethanol2.gam

Call:
gam(formula = NOx ~ C + lo(E, degree = 2))

Degrees of Freedom: 88 total; 80.1184 Residual
Residual Deviance: 5.16751
> summary(ethanol2.gam)
```

```
Call: gam(formula = NOx ~ C + lo(E, degree = 2))
Deviance Residuals:
        Min         1Q     Median         3Q        Max
 -0.6113908 -0.166044 0.0268504 0.1585614 0.4871313

(Dispersion Parameter for Gaussian family taken to be
0.0644985 )

Null Deviance: 111.6238 on 87 degrees of freedom

Residual Deviance: 5.167513 on 80.1184 degrees of freedom

Number of Local Scoring Iterations: 1

DF for Terms and F-values for Nonparametric Effects

                  Df Npar Df   Npar F Pr(F)
      (Intercept)  1
                C  1
lo(E, degree = 2)  2     3.9 57.95895     0
```

We can use the `anova` function to compare this model with the simpler model involving `E` only:

```
> anova(ethanol.gam, ethanol2.gam, test="F")
Analysis of Deviance Table

Response: NOx
                  Terms Resid. Df Resid. Dev Test Df
1     lo(E, degree = 2)   81.1184   9.137801
2 C + lo(E, degree = 2)   80.1184   5.167513   +C  1
  Deviance  F Value        Pr(F)
1
2 3.970288 61.55632 1.607059e-11
```

The model involving `C` is clearly better, since the residual deviance is cut almost in half by expending only one more degree of freedom.

Is the additive model sufficient? Additive models stumble when there are *interactions* among the various terms. In the case of the `ethanol` data, there is a significant interaction between `C` and `E`. In such cases, a full local regression model, fit using the `loess` function, is often more satisfactory. We discuss the `ethanol` data more thoroughly in the chapter Local Regression Models.

7.13 MORE ON NONPARAMETRIC REGRESSION

The additive models fitted by `gam` in the section Additive Models are simple examples of *nonparametric* regression. The machinery of generalized additive models, proposed by Hastie and Tibshirani (1990), is just one approach to such nonparametric models. S-PLUS includes several other functions for performing nonparametric regression, including the `ace` function, which implements the first proposed technique for nonparametric regression—alternating conditional expectations. S-PLUS also includes AVAS (*A*dditive and *VA*riance *S*tabilizing transformations) and projection pursuit regression. This section describes these varieties of nonparametric regression.

Alternating Conditional Expectations

Alternating conditional expectations or `ace`, is an intuitively appealing technique introduced by Breiman and Friedman (1985). The idea is to find nonlinear transformations $\theta(y)$, $\phi_1(x_1)$, $\phi_2(x_2)$,..., $\phi_p(x_p)$ of the response y and predictors $x_1, x_2,\ldots, x_p$, respectively, such that the *additive model*

$$\theta(y) = \phi_1(x_1) + \phi_2(x_2) + \cdots + \phi_p(x_p) + \varepsilon \tag{7.5}$$

is a good approximation for the data $y_i, x_{i1},\ldots, x_{ip}$, $i = 1,\ldots, n$. Let $y_i, x_1, x_2,\ldots, x_p$ be random variables with joint distribution F, and let expectations be taken with respect to F. Consider the *goodness-of-fit* measure

$$e^2 = e^2(\theta, \phi_1, ...,\phi_p) = \frac{E\left[\theta(y) - \sum_{k=1}^{p} \phi_k(x_k)\right]^2}{E\theta^2(y)} \tag{7.6}$$

The measure e^2 is the fraction of variance not explained by regressing $\theta(y)$ on $\phi(x_1),\ldots,\phi(x_p)$. The data-based version of e^2 is

$$\hat{e}^2 = \frac{\sum_{i=1}^{n}\left[\hat{\theta}(y_i) - \sum_{k=1}^{p} \hat{\phi}_k(x_{ik})\right]^2}{\sum_{i=1}^{n} \hat{\theta}^2(y_i)} \tag{7.7}$$

where $\hat{\theta}$ and the $\hat{\phi}_j$, estimates of θ and the ϕ_j, are standardized so that $\hat{\theta}(y_i)$ and the $\hat{\phi}_j(x_{ij})$ have mean zero: $\sum_{i=1}^{n} \hat{\theta}(y_i) = 0$ and $\sum_{i=1}^{n} \hat{\phi}_k(x_{ik}) = 0$, k=1,…,p. For the usual linear regression case, where

$$\hat{\theta}(y_i) = y_i - \bar{y}$$

and

$$\hat{\phi}_1(x_{i1} - \bar{x}_1) = (x_{i1} - \bar{x}_1)\hat{\beta}_1, \ldots, \hat{\phi}_p(x_{ip} - \bar{x}_p) = (x_{ip} - \bar{x}_p)\hat{\beta}_p$$

with $\hat{\beta}_1, \ldots, \hat{\beta}_p$ the least squares regression coefficients, we have

$$\hat{e}^2_{LS} = \frac{RSS}{SSY} \equiv \frac{\sum_{i=1}^{n}\left[(y_i - \bar{y}) - \sum_{k=1}^{p}(x_{ik} - \bar{x}_k)\hat{\beta}_k\right]^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

and the squared multiple correlation coefficient is given by $R^2 = 1 - e^2_{LS}$. The transformations $\hat{\theta}$, $\hat{\phi}_1, \ldots, \hat{\phi}_p$ are chosen to maximize the correlation between $\hat{\theta}(y_i)$ and $\hat{\phi}(x_{i1}) + \cdots + \hat{\phi}(x_{ip})$. Although `ace` is a useful exploratory tool for determining which of the response y and the predictors $x_1,\ldots,x_p$ are in need of nonlinear transformations and what type of transformation is needed, it can produce anomalous results if errors ε and the $\hat{\phi}_1(x_i)$ fail to satisfy the independence and normality assumptions.

To illustrate the use of `ace`, construct an artificial data set with *additive* errors

$$y_i = e^{1 + 2x_i} + \varepsilon_i i \ , \ i=1,\ldots,200$$

with the ε_i's being $N(0, 10)$ random variables (that is, normal random variables with mean 0 and variance 10), independent of the x_i's, with the x_i's being $U(0, 2)$ random variables (that is, random variables uniformly distributed on the interval from 0 to 2).

```
> set.seed(14) # set the seed to reproduce this example
> x <- 2*runif(200)
```

```
> e <- rnorm(200, 0, sqrt(10))
> y <- exp(1+2*x) + e
```

Now use ace:

```
> a <- ace(x,y)
```

Set graphics for 3×2 layout of plots:

```
> par(mfrow=c(3,2))
```

Make plots to do the following:

1. Examine original data;
2. Examine transformation of y ;
3. Examine transformation of x ;
4. Check linearity of the fitted model;
5. Check residuals versus the fit:

The following S-PLUS commands provide the desired plots:

```
> plot(x, y, sub="Original Data")
> plot(x, a$tx, sub="Transformed x vs. x")
> plot(y, a$ty, sub="Transformed y vs. y")
> plot(a$tx, a$ty, sub="Transformed y vs.
+      Continue string: Transformed x")
> plot(a$tx, a$ty - a$tx, xlab="tx",
+ ylab="residuals", sub="Residuals vs. Fit")
```

These plots are displayed in figure 7.20, where the transformed values $\hat{\theta}(y)$ and $\hat{\phi}(y)$ are denoted by *ty* and *tx*, respectively. The estimated transformation $tx = \hat{\phi}(x)$ seems close to exponential, and except for the small bend at the lower left, the estimated transformation $ty = \hat{\theta}(y)$ seems quite linear. The linearity of the plot of *ty* versus *tx* reveals that a good additive model of the type shown in equation (7.5) has been achieved. Furthermore, the error variance appears to be relatively constant, except at the very lefthand end. The plot of residuals, $r_i = \hat{\theta}(y_i) - \hat{\phi}(x_i)$ versus the fit $tx = \hat{\phi}(x_i)$ gives a clearer confirmation of the behavior of the residuals' variance.

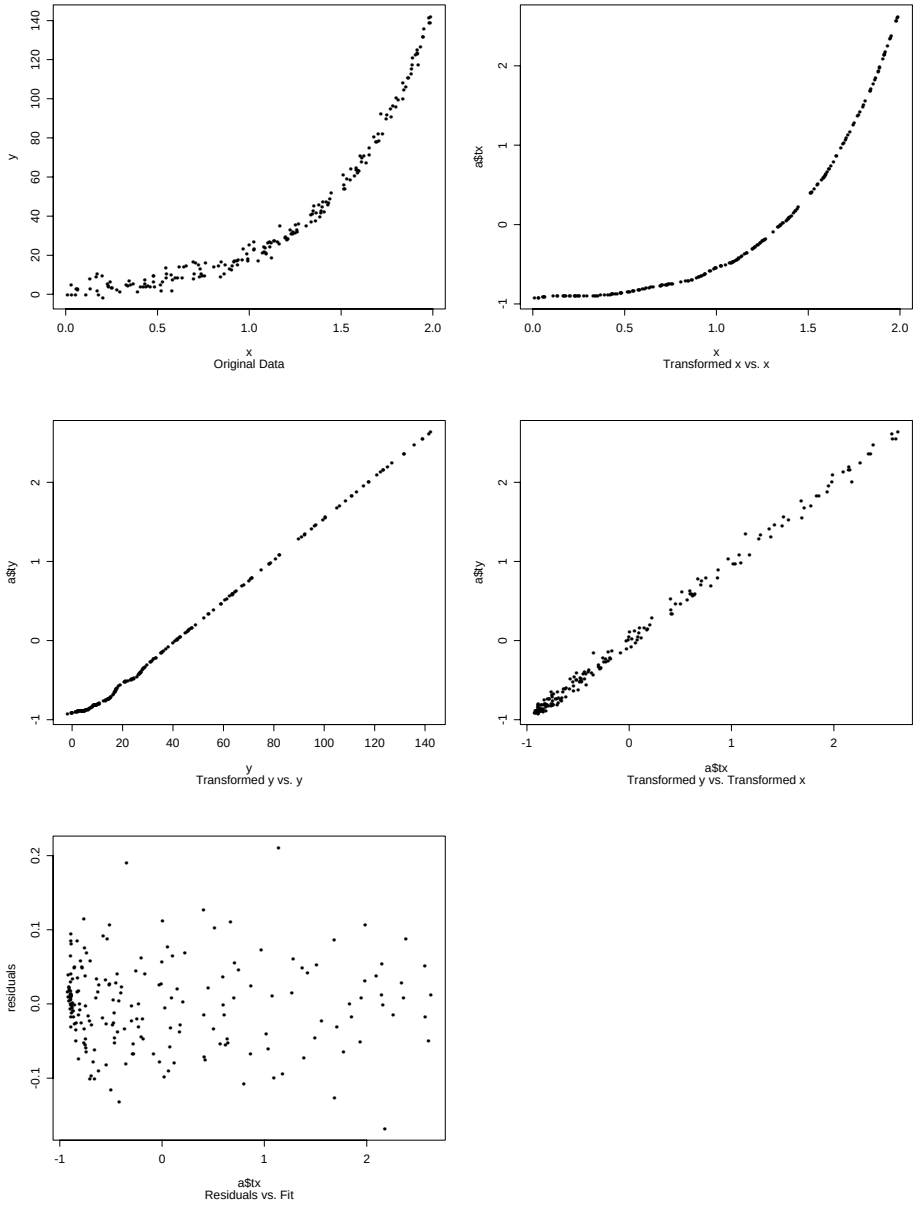

Figure 7.20: `ace` *example with additive errors*

Additive and Variance Stabilizing Transformation

The term "avas" stands for additivity and variance stabilizing transformation. Like ace, avas tries to find transformations $\theta(y)$, $\phi_1(x_1)$,…, $\phi_p(x_p)$ such that

$$\theta(y) = \phi_1(x_1) + \phi_2(x_2) + \cdots + \phi_p(x_p) + \varepsilon \tag{7.8}$$

provides a good *additive* model approximation for the data $y_i, x_{i1},\ldots, x_{ip}$, i=1, 2,…,n. However, avas differs from ace in that it chooses $\theta(y)$ to achieve a special *variance stabilizing* feature. In particular the goal of avas is to estimate transformations $\theta, \phi_1,\ldots, \phi_p$ which have the properties

$$E[\theta(y)|x_1, \ldots, x_p] = \sum_{i=1}^{p} \phi_i(x_i) \tag{7.9}$$

and

$$\text{var}\left[\theta(y)|\sum_{i=1}^{p} \phi_i(x_i)\right] = \text{constant} \tag{7.10}$$

Here $E[z|w]$ is the conditional expectation of z given w. The additivity structure 7.9 is the same as for ace, and correspondingly the ϕ_i's are calculated by the backfitting algorithm

$$\phi_k(x_k) = E\left[\theta(y) - \sum_{i \neq k} \phi_i(x_i)\Big|x_k\right] \tag{7.11}$$

cycling through k = 1, 2,…, p until convergence. The variance stabilizing aspect comes from equation (7.9). As in the case of ace, estimates $\hat{\theta}(y_i)$ and $\phi_j(x_{ik})$, k = 1, 2,…, p are computed to approximately satisfy equations 7.8 through 7.11, with the conditional expectations in 7.8 and 7.11 estimated using the super smoother scatterplot smoother (see supsmu function documentation). The equality 7.9 is approximately achieved by estimating the classic stabilizing transformation.

To illustrate the use of avas, construct an artificial data set with *additive* errors

$$y_i = e^{1+2x_i} + \varepsilon_i \, i \;\; , i = 1, \ldots, 200$$

with the ε_i's being $N(0, 10)$ random variables (that is, normal random variables with mean 0 and variance 10), independent of the x_i's, with the x_i's being $U(0, 2)$ random variables (that is, random variables uniformly distributed on the interval from 0 to 2).

```
> set.seed(14) #set the seed to reproduce this example > x
<- runif(200, 0, 2)
> e <- rnorm(200, 0, sqrt(10))
> y <- exp(1+2*x) + e
```

Now use `avas`:

```
> a <- avas(x, y)
```

Set graphics for a 3×2 layout of plots:

```
> par(mfrow=c(3,2))
```

Make plots to: (1) examine original data; (2) examine transformation of *x*; (3) examine transformation of *y*; (4) check linearity of the fitted model; (5) check residuals versus the fit:

```
> plot(x, y, sub="Original data")
> plot(x, a$tx, sub="Transformed x vs. x")
> plot(y, a$ty, sub="Transformed y vs. y")
> plot(a$tx, a$ty, sub="Transformed y vs. Transformed x")
> plot(a$tx, a$ty - a$tx, ylab="Residuals",
+ sub="Residuals vs. Fit")
```

These plots are displayed in figure 7.9 where the transformed values $\hat{\theta}(y)$ and $\hat{\phi}(x)$ are denoted by *ty* and *tx*, respectively. The estimated transformation $tx = \hat{\phi}(x)$ seems close to exponential, and the estimated transformation $ty = \hat{\theta}(y)$ seems linear. The plot of *ty* versus *tx* reveals that a linear additive model holds; that is, we have achieved a good additive approximation of the type 7.8. In this plot the error variance appears to be relatively constant. The plot of residuals, $r_i = \hat{\theta}(y_i) - \hat{\phi}(x_i)$, versus the fit $tx = \hat{\phi}(x_i)$ gives further confirmation of this.

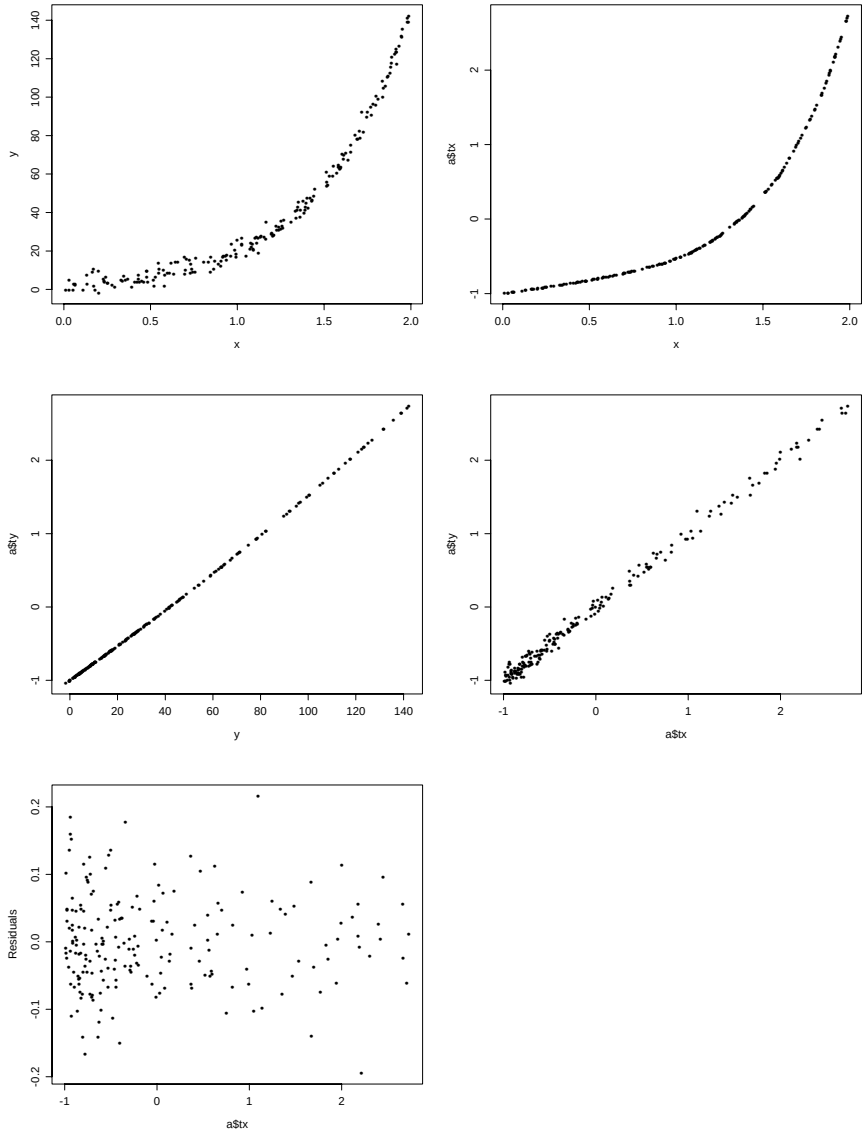

Figure 7.21: `avas` *example with additive errors.*

KEY PROPERTIES

- Suppose that the true additive model is

$$\theta^0(y) = \sum_{i=1}^{p} \phi_i^0(x_i) + \varepsilon \tag{7.12}$$

with ε independent of $x_1, x_2, ..., x_p$, and var(ε) = constant. Then the iterative avas algorithm for 7.9 – 7.11, described below for the data versions of 7.9 – 7.11, yields a sequence of transformations $\theta^{(j)}$, $\phi_1^{(j)}, ..., \phi_p^{(j)}$ which converge to the true transformation θ^0, ϕ_1^0, ..., ϕ_p^0, as the number of iterations j tends to infinity. Correspondingly, the data-based version of this iteration yields a sequence of transformations $\hat{\theta}^{(j)}$, $\hat{\phi}_1^{(j)}$, ..., $\hat{\phi}_p^{(j)}$, which, at convergence, provide estimates $\hat{\theta}$, $\hat{\phi}_1, ..., \hat{\phi}_p$, of the true model transformations θ^0, $\phi_1^0, ..., \phi_p^0$.

- avas appears not to suffer from some of the anomalies of ace, e.g., not finding good estimates of a true additive model (equation 7.12) when normality of ε and joint normality of $\phi_1(x_1), \ldots, \phi_p(x_p)$ fail to hold. See the example below.
- avas is a generalization of the Box and Cox (1964) maximum-likelihood procedure for choosing power transformation y^λ of the response. The function avas also generalizes the Box and Tidwell (1962) procedure for choosing transformations of the carriers $x_1, x_2, ..., x_p$, and is much more convenient than the Box-Tidwell procedure. See also Weisberg (1985).
- $\hat{\theta}(y)$ is a monotone transformation, since it is the integral of a nonnegative function (see Further Details, below). This is important if one wants to predict y by inverting $\hat{\theta}$: monotone transformations are invertible, and hence we can predict y with $\hat{y} = \hat{\theta} G^{-1}\left[\sum_{i=1}^{p} \hat{\phi}_i(x_i)\right]$. This predictor has no particular optimality

property, but is simply one straightforward way to get a prediction of y once an `avas` model has been fit.

FURTHER DETAILS Let

$$v(u) = \mathrm{VAR}\left[\hat{\theta}(y) \Big| \sum_{i=1}^{p} \phi_i(x_i) = u\right] \tag{7.13}$$

where $\ddot{\theta}(y)$ is an arbitrary transformation of y, $\ddot{\theta}(y)$ will be the "previous" estimate of $\theta(y)$ in the overall iterative procedure described below. Given the variance function $v(u)$, it is known that

$$\mathrm{VAR}\left[g(\hat{\theta}(y)) \Big| \sum_{i=1}^{p} \phi_i(x_i) = u\right]$$

will be constant if g is computed according to the rule

$$g(t) = \int_c^t \frac{du}{v^{1/2}(u)} \tag{7.14}$$

for an appropriate constant c. See Box and Cox (1964).

The detailed steps in the population version of the `avas` algorithm are as follows:

1. *Initialize*:
   Set $\hat{\theta}(y) = (y - \mathrm{E}y)/(\mathrm{VAR}^{1/2}y)$ and backfit on $x_1, ..., x_p$ to get $\hat{\phi}_1, ..., \hat{\phi}_p$. (See description of ace for details of "backfitting".)
2. *Get new transformation* of y :
   a) Compute variance function

$$v(u) = \mathrm{VAR}\left[\hat{\theta}(y) \Big| \sum_{i=1}^{p} \hat{\phi}_i(x_i) = u\right]$$

   b) Compute variance stabilizing transformation

$$g(t) = \int_c^t \frac{du}{v^{1/2}(u)}$$

c) Set $\hat{\hat{\theta}}(y) - g(\hat{\hat{\theta}}(y))$ and standardize

$$\hat{\theta}(y) - \frac{\hat{\theta}(y) - \mathrm{E}\hat{\theta}(y)}{\mathrm{VAR}^{1/2}\hat{\theta}(y)}$$

3. *Get new $\hat{\phi}_i$'s:*

   Backfit $\hat{\theta}(y)$ on $x_1, x_2, \ldots, x_p$ to obtain new estimates $\hat{\phi}_1, \ldots, \hat{\phi}_p$.

4. *Iterate steps 2 and 3 until*

$$R^2 = 1 - \hat{e}^2 = 1 - \mathrm{E}\left[\hat{\theta}(y) - \sum_{i=1}^{p} \hat{\phi}_i(x_i)\right]^2 \tag{7.15}$$

   doesn't change.

Of course the above algorithm is actually carried out using the sample of data $y_i, x_{i1}, \ldots, x_{ip}$, $i = 1, \ldots, n$, with expectations replaced by sample averages, conditional expectations replaced by scatterplot smoothing techniques and VAR's replaced by sample variances.

In particular, super smoother is used in the backfitting step to obtain $\hat{\phi}_1(x_{i1}), \ldots, \hat{\phi}_p(x_{ip})$, $i = 1, \ldots, n$. An estimate $v(u)$ of $v(u)$ is obtained as follows: First the scatter plot of $\log r_i^2 = \log\left[\hat{\theta}(y_i) - \sum_{j=1}^{p} \hat{\phi}_j(x_{ij})\right]^2$ versus $u_i = \sum_{j=1}^{p} \hat{\phi}_j(x_{ij})$ is smoothed using a running straight lines smoother. Then the result is exponentiated. This gives an estimate $v(u) \geq 0$, and $v(u)$ is truncated below at 10^{-10} to insure positivity and avoid dividing by zero in the integral 7.14. The integration in equation 7.14 is carried out using a trapezoidal rule.

Projection Pursuit Regression

The basic idea behind projection pursuit regression, `ppreg`, is as follows. Let y and $\mathbf{x} = (x_1, x_2, ..., x_p)^T$ denote the response and explanatory vector, respectively. Suppose you have observations y_i and corresponding predictors $\mathbf{x}_i = (x_{i1}, x_{i2}, ..., x_{ip})^T$, $i = 1, 2, ..., n$. Let $\mathbf{a}_1, \mathbf{a}_2, ...,$ denote p-dimensional unit vectors, as "direction" vectors, and let $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$. The `ppreg` function allows you to find $M = M_0$, direction vectors $\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_{M_0}$ and good nonlinear transformations $\phi_1, \phi_2, ..., \phi_{M_0}$ such that

$$y \approx \bar{y} + \sum_{m=1}^{M_0} \beta_m \phi_m(\mathbf{a}_m^T \mathbf{x}) \tag{7.16}$$

provides a "good" model for the data y_i, $\mathbf{x}_i$, $i = 1, 2, ..., n$. The "projection" part of the term projection pursuit regression indicates that the carrier vector x is *projected* onto the direction vectors $\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_{M_0}$ to get the lengths $\mathbf{a}^T\mathbf{x}$, $i = 1, ..., n$ of the projections, and the "pursuit" part indicates that an optimization technique is used to find "good" direction vectors $\mathbf{a}_1, \mathbf{a}_2, ..., \mathbf{a}_{M_0}$.

More formally, y and $\mathbf{x}$ are presumed to satisfy the conditional expectation model

$$\mathrm{E}[y|x_1, x_2, ..., x_p] = \mu_y + \sum_{m=1}^{M_0} \beta_m \phi_m(\mathbf{a}_m^T \mathbf{x}) \tag{7.17}$$

where $\mu_y = E(y)$, and the ϕ_m have been standardized to have mean zero and unity variance:

$$\mathrm{E}\phi_m(\mathbf{a}_m^T\mathbf{x}) = 0, \quad \mathrm{E}\phi_m^2(\mathbf{a}_m^T\mathbf{x}) = 1, \qquad = 1, ..., M_0. \tag{7.18}$$

The observations y_i, $\mathbf{x}_i = (x_{i1}, ..., x_{ip})^T$, $i = 1, ..., n$, are assumed to be independent and identically distributed random variables like y and $\mathbf{x}$, i.e., they satisfy the model in equation 7.17.

The true model parameters β_m, ϕ_m, $\mathbf{a}_m$, $m = 1, ..., M_0$ in equation 7.17 minimize the mean squared error

$$\mathrm{E}\left[y - \mu_y - \sum_{m=1}^{M_0} \beta_m \phi_m(\mathbf{a}_m^T \mathbf{x})\right]^2 \tag{7.19}$$

over all possible β_m, ϕ_m, and $\mathbf{a}_m$.

Equation 7.17 includes the additive `ace` models under the restriction $\theta(y) = y$. This occurs when $M_0 = p$ and $\mathbf{a}_1 = (1, 0, ..., 0)^T$, $\mathbf{a}_2 = (0, 1, 0, ..., 0)^T$, $\mathbf{a}_p = (0, 0, ..., 0, 1)^T$, and the β_m's are absorbed into the ϕ_m's. Furthermore, the ordinary linear model is obtained when $M_0 = 1$, assuming the predictors $\mathbf{x}$ are independent with mean 0 and variance 1. Then $\mathbf{a}^T = (b_1, ..., b_p)/\sqrt{b_1^2 + \cdots + b_p^2}$, $\phi_1(t) = t$, and $\beta_1 = \sqrt{b_1^2 + \cdots + b_p^2}$, where the b_j are the regression coefficients.

The projection pursuit model in equation 7.17 includes the possibility of having *interactions* between the explanatory variables. For example, suppose that

$$E[y|x_1, x_2] = x_1 x_2. \tag{7.20}$$

This is described by 7.17 with $\mu_y = 0$, $M_0 = 2$, $\beta_1 = \beta_2 = \frac{1}{4}$, $\mathbf{a}_1^T = (1, 1)$, $\mathbf{a}_2^T = (1, -1)$, $\phi_1(t) = t^2$, and $\phi_2(t) = -t^2$. For then

$$\phi_1(\mathbf{a}_1^T \mathbf{x}) = (x_1 + x_2)^2 = x_1^2 + 2x_1x_2 + x_2^2$$

$$\phi_2(\mathbf{a}_2^T \mathbf{x}) = -(x_1 - x_2)^2 = -x_1^2 + 2x_1x_2 - x_2^2$$

so that

$$\sum_{m=1}^{2} \beta_m \phi_m(\mathbf{a}^T \mathbf{x}) = x_1 x_2 .$$

Neither `ace` nor `avas` is able to model interactions. It is this ability to pick up interactions that led to the invention of projection pursuit regression by Friedman and Stuetzle (1981), and it is what makes `ppreg` a useful complement to `ace` and `avas`.

The two variable interactions shown above can be used to illustrate the `ppreg` function. The two predictors, x_1 and x_2 are generated as uniform random variates on the interval - 1 to 1. The response, *y*, is the product of x_1 and x_2 plus a normal error with mean zero and variance 0.04.

```
> set.seed(14)  #set the seed to reproduce this example
> x1 <- runif(400, -1, 1)
> x2 <- runif(400, -1, 1)
> eps <- rnorm(400, 0, .2)
> y <- x1*x2+eps
> x <- cbind(x1, x2)
```

Now run the projection pursuit regression with `max.term` set at 3, `min.term` set at 2 and with the residuals returned in the `ypred` component (the default if `xpred` is omitted).

```
> p <- ppreg(x, y, 2, 3)
```

Make plots (shown in figure 7.22) to examine the results of the regression.

```
> par(mfrow=c(3, 2))
> plot(x1, y, sub="Y vs X1")
> plot(x2, y, sub="Y vs X2")
> plot(p$z[,1], p$zhat[,1], sub="1st Term:
Continue string: Smooth vs Projection Values z1")
> plot(p$z[,2], p$zhat[,2], sub="2nd Term:
Continue string: Smooth vs Projection Values z2")
> plot(y-p$ypred, y, sub="Response vs Fit")
> plot(y-p$ypred, p$ypred, sub="Residuals vs Fit")
```

The first two plots show the response plotted against each of the predictors. It is difficult to hypothesize a function form for the relationship when looking at these plots. The next two plots show the resulting smooth functions from the regression plotted against their respective projection of the carrier variables. Both the plots have a quadratic shape with one being positive and the other negative, the expected result for this type of interaction function. The fifth plot shows clearly a linear relationship between the response and the fitted values. The residuals shown in the last plot do not display any unusual structure.

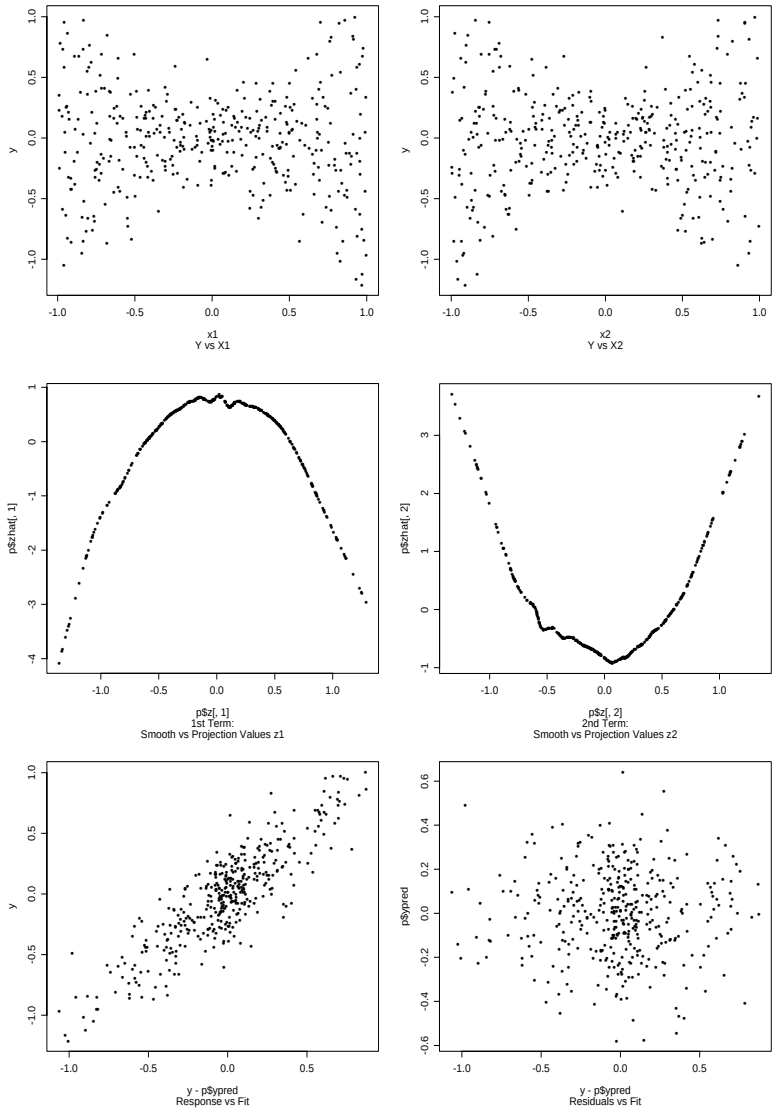

Figure 7.22: *Projection pursuit example.*

FURTHER DETAILS

The Forward Stepwise Procedure

An initial M-term model of the form given by the right-hand side of equation 7.17, with the constraints of equation 7.18 and $M > M_0$, is estimated by a forward stepwise procedure, as described by Friedman and Stuetzle (1981).

First, a trial direction $\mathbf{a}_1$ is used to compute the values $z_{i1} = \mathbf{a}_1^T \mathbf{x}_i$, $i = 1, ..., n$, where $\mathbf{x}_i = (x_{i1}, ..., x_{ip})^T$. Then, with $\tilde{y}_i^1 = y_i - \bar{y}$, you have available a scatter plot of data $(\tilde{y}_i, z_{i1})$, $i = 1, ..., n$, which may be smoothed to obtain an estimate $\hat{\phi}_1(z_{i1})$ of the conditional expectation $E[y|z_1] = E[y_i|z_{i1}]$ for the identically distributed random variables y_i, $z_{i1} = \mathbf{a}_1^T \mathbf{x}_i$. Super Smoother is used for this purpose; see documentation for `supsmu`. This $\hat{\phi}_1$ depends upon the trial direction vector $\mathbf{a}_1$, so we write $\phi_1 = \phi_{1, \mathbf{a}_1}$. Now $\mathbf{a}_1$ is varied to minimize the weighted sum of squares,

$$\sum_{i=1}^{n} w_i [y_i - \hat{\phi}_{1, \mathbf{a}_1}(z_{i1})]^2 \tag{7.21}$$

where for each $\mathbf{a}_1$ in the optimization procedure, a new $\hat{\phi}_{1, \mathbf{a}_1}$ is computed using super smoother. The weights w_i are user-specified, with the default being all weights unitary: $w_i \equiv 1$. The final results of this optimization will be denoted simply $\hat{\mathbf{a}}_1$ and $\hat{\phi}_1$, where $\hat{\phi}_1$ has been standardized according to equation 7.18 and the corresponding value $\hat{\beta}_1$ is computed. We now have the approximation

$$y_i \approx \bar{y} + \hat{\beta}_1 \hat{\phi}_1(\hat{\mathbf{a}}_1^T \mathbf{x}_i) \quad i = 1, ..., n.$$

Next we treat $y_i^{(2)} = y_i - \bar{y} - \hat{\beta}_1 \hat{\phi}_1(z_{i1})$ as the response, where now $z_{i1} = \hat{\mathbf{a}}_1^T \mathbf{x}_i$, and fit a second term $\hat{\beta}_2 \hat{\phi}_2(\mathbf{z}_{i2})$, where $z_{i2} = \hat{\mathbf{a}}_2^T \mathbf{x}_i$, to this modified response, in exactly the same manner that we fitted $\hat{\beta}_1 \hat{\phi}_1(\hat{\mathbf{a}}_1^T \mathbf{x}_i)$ to $\mathbf{y}_i^{(1)}$. This gives the approximation

$$y_i^{(2)} \approx \hat{\beta}_2 \hat{\phi}_2(z_{i2})$$

or

$$y_i \approx \bar{y} + \hat{\beta}_1 \hat{\phi}_1(z_{i1}) + \hat{\beta}_2 \hat{\phi}_2(z_{i2})$$

Continuing in this fashion we arrive at the forward stepwise estimated model

$$y_i \approx \bar{y} + \sum_{m=1}^{M} \hat{\beta}_m \hat{\phi}_m(z_{im}) , \; i = 1, ..., n . \tag{7.22}$$

where $z_{im} = \hat{\mathbf{a}}_m^T \mathbf{x}_i$, $m = 1, ..., M$.

The Backward Stepwise Procedure

Having fit the M term model in equation 7.22 in a forward stepwise manner, `ppreg` fits all models of decreasing order $m = M - 1, M - 2, ..., M_{min}$, where M and M_{min} are user-specified. For each term in the model, the weighted sum of squared residuals

$$\mathrm{SSR}(m) = \sum_{i=1}^{n} w_i \left[y_i - \bar{y} - \sum_{l=1}^{m} \beta_l \phi_l(\mathbf{a}_l^T \mathbf{x}_i) \right]^2 \tag{7.23}$$

is minimized through the choice of β_l, $\mathbf{a}_l$, ϕ_l, $l = 1, ..., m$. The initial values for these parameters, used by the optimization algorithm which minimizes equation 7.23, are the solution values for the *m most important* out of $m + 1$ terms in the previous order $m + 1$ model. Here *importance* is measured by

$$\boldsymbol{I}_l = \left| \hat{\beta}_l \right| \quad l = 1, ..., m + 1 \tag{7.24}$$

where $\hat{\beta}_l$ are the optimal coefficients for the $m + 1$ term model, $m = M$-1, M-2, …, M_{min}.

Model Selection Strategy

In order to determine a “good" number of terms M_0 for the `ppreg` model, proceed as follows. First, run `ppreg` with $M_{min} = 1$ and M set at a value large enough for the data analysis problem at hand. For a relatively small number of variables p, say $p \le 4$, you might well choose $M \ge p$. For large p, you would probably choose $M < p$, hoping for a parsimonious representation.

For each order m, $1 \le m \le M$, `ppreg` will evaluate the fraction of unexplained variance

$$e^2(m) = \frac{\mathrm{SSR}(m)}{\sum_{i=1}^{n} w_i[y_i - \bar{y}]^2}$$

$$= \frac{\sum_{i=1}^{n} w_i \left[y_i - \bar{y} - \sum_{l=1}^{m} \hat{\beta}_l \hat{\phi}_l(\hat{\mathbf{a}}_l^T \mathbf{x}_i) \right]^2}{\sum_{i=1}^{n} w_i[y_i - \bar{y}]^2}$$

A plot of $e^2(m)$ versus m which is decreasing in m may suggest a good choice of $m = M_0$. Often $e^2(m)$ will decrease relatively rapidly when m is smaller than a good model order M_0 (as the (bias)^2 component of prediction mean-squared error is decreasing rapidly), and then tend to flatten out and decrease more slowly for m larger than M_0. You can choose M_0 with this in mind.

The current version of `ppreg` has the feature that when fitting models having $m = M_{min}, M_{min} + 1, ..., M$ terms, *all* of the values $\hat{\beta}_l$, $\hat{\mathbf{a}}_l$, $\hat{\phi}_l(z_{il})$, $z_{il} = \hat{\mathbf{a}}_l^T \mathbf{x}_i$, $i = 1, ..., n$, $l = 1, ..., m$, and $e^2(m)$ are returned for $m = M_{min}$, whereas all of these *except* the smoothed values $\hat{\phi}_l(z_{il})$ and their corresponding arguments z_{il} are returned for all $m = M_{min}, ..., M$. This feature conserves storage requirements. As a consequence, you must run `ppreg` twice for $m = M_{min}, ..., M$, using two different values of M_{min}: The first time $M_{min} = 1$ is used in order to examine $e^2(m)$, $m = 1, ..., M$ (among other things) and choose a good order M_0. The second time $M_{min} = M_0$ is used in order obtain all output, including $\hat{\phi}_l(z_{il})$ and z_{il} values.

Multivariate Response

All of the preceding discussion has been concentrated on the case of a single response y, with observed values $y_1, ..., y_n$. In fact, `ppreg` is designed to handle multivariate responses $y_1, ..., y_q$ with observed values y_{ij}, $i = 1, ..., n$, $j = 1, ..., q$

. For this case, `ppreg` allows you to fit a good model

$$y_j \approx \bar{y}_j + \sum_{m=1}^{M_0} \hat{\beta}_{mj} \hat{\phi}_m(\hat{\mathbf{a}}_m^T \mathbf{x}) \tag{7.25}$$

by minimizing the multivariate response weighted sum of squared residuals

$$\mathrm{SSR}_q(m) = \sum_{j=1}^{q} W_j \sum_{i=1}^{n} w_i \left[y_{ij} - \bar{y}_j - \sum_{l=1}^{m} \hat{\beta}_{lj} \hat{\phi}_l(\mathbf{a}_l^T \mathbf{x}_i) \right]^2 \tag{7.26}$$

and choosing a good value $m = M_0$.

Here the W_j are user-specified *response* weights (with default $W_j \equiv 1$), the w_i are user-specified *observation* weights (with default $w_i \equiv 1$), and $\bar{y}_j = \frac{1}{n}\sum_{i=1}^{n} y_{ij}$. Note that a single set of $\hat{\phi}_m$'s is used for all responses y_{ij}, $j = 1, ..., q$, whereas the different behavior of the different responses is modeled by different linear combinations of the $\hat{\phi}_m$'s by virtue of the different sets of coefficients $\hat{\beta}_j = (\hat{\beta}_{ij}, ..., \hat{\beta}_{mj})^T$, $j = 1, ..., q$.

The `ppreg` procedure for the multivariate response case is similar to the single response case. For given values of M_{min} and M, `ppreg` first does a forward stepwise fitting starting with a single term ($m = 1$), and ending up with M terms, followed by a backward stepwise procedure stopping with an M_{min}-term model. When passing from an $m + 1$ term model to an m-term model in the multivariate response case, the relative importance of a term is given by

$$I_l = \sum_{j=1}^{q} W_j \left| \hat{\beta}_{jl} \right| \quad l = 1, ..., m + 1$$

The most important terms are the ones with the largest I_l, and the corresponding values of $\hat{\beta}_{jl}$, $\hat{\phi}_l$, and $\mathbf{a}_l$ are used as initial conditions in the minimization of $SSR_q(m)$. Good model selection; that is, a good choice $m =$

M_0, can be made just as in the case of a single response, namely, through examination of the multivariate response fraction of unexplained variation

$$e_q^2(m) = \frac{\mathrm{SSR}_q(m)}{\sum_{j=1}^{q} W_j \sum_{i=1}^{n} w_i [y_{ij} - \bar{y}_j]^2} \tag{7.27}$$

by first using `ppreg` with $M_{min} = 1$ and a suitably large M. Then `ppreg` is run again with $M_{min} = M_0$ and the same large M.

7.14 REFERENCES

Box, G.E.P. and Tidwell, P.W. (1962). Transformations of independent variables. *Technometrics*, 4:531–550.

Box, G.E.P. and Cox, D.R. (1964). An analysis of transformations (with discussion). *Journal of the Royal Statistical Society, Series B*, 26:211–246.

Breiman, L. and Friedman, J. H. (1985). Estimating optimal transformations for multiple regression and correlation (with discussion). *Journal of the American Statistical Association*, 80:580–619.

Friedman, J. H. and Stuetzle, W. (1981). Projection pursuit regression. *Journal of the American Statistical Association*, 76:817–823.

Friedman, J.H. (1984). SMART users guide, no. 1. *Laboratory for Computational Statistics, Dept. of Statistics, Stanford University*", Stanford, CA.

Friedman, J.H. (1984). A variable span smoother. Tech. Rept. 5, Laboratory for Computational Statistics, Dept. of Statistics, Stanford University, Stanford, CA.

Friedman, J.H. (1985). Classification and multiple regression through projection pursuit, Tech. Rept. 12, Dept. of Statistics, Stanford University, Stanford, CA.

Hampel, F.R. and Ronchetti, E.M. and Rousseeuw, P.J. and Stahel, W.A. (1986). *Robust Statistics: The Approach Based on Influence Functions*. Wiley, New York.

Hastie, T. and Tibshirani, R. (1990). *Generalized Additive Models*. Chapman and Hall, London.

Heiberger, R. M. and Becker, R. A. (1992). Design of an S function for robust regression using iteratively reweighted least squares. *Journal of Computational and Graphical Statistics*, 1:181–196.

Huber, P.J. (1973). *Robust regression: Asymptotics, conjectures, and Monte Carlo.* Annals of Statistics, 1:799–821.

Huber, P.J. (1981). *Robust Statistics*. Wiley, New York.

Rousseeuw, P. J. (1984). Least median of squares regression. *Journal of the American Statistical Association*, 79:871–888.

Rousseeuw, P. J. and Leroy, A. M. (1987). *Robust Regression and Outlier Detection*. Wiley, New York.

Silverman, B.W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman and Hall, London.

Tibshirani, R. (1988). *Estimating transformations for regression via additivity and variance stabilization*. Journal of the American Statistical Association, 83:394–405.

Watson, G. S. (1966). *Smooth regression analysis*. Sankhya, Series A, 26:359–378.

Weisberg, S. (1985). *Applied Linear Regression*, 2nd edition. John Wiley, New York.

GENERALIZING THE LINEAR MODEL

8

Generalized linear and additive models are powerful tools for sophisticated analyses of categorical responses.

GENERALIZING THE LINEAR MODEL

The use of least squares estimation of regression coefficients for linear models dates back to the early nineteenth century. It met with immediate success as a simple way of mathematically summarizing relationships between observed variables of real phenomena. It quickly became and remains one of the most widely used statistical methods of practicing statisticians and scientific researchers.

Because of the simplicity, elegance, and widespread use of the linear model, researchers and practicing statisticians have tried to adapt its methodology to different data configurations. For example, there is no reason conceptually why a categorical response or some transformation of it could not be related to a set of predictor variables in a similar way to the continuous response of the linear model. Although conceptually plausible, developing regression models for categorical responses lacked solid theoretical foundation until the introduction of the generalized linear model by Nelder and Wedderburn (1972).

This chapter focuses on generalized linear models and their generalization, generalized additive models, as they apply to categorical responses. In particular, we focus on logistic and Poisson regressions and also include a brief discussion of the fitting of models when you can't specify an exact likelihood, using the quasi-likelihood method.

8.1 LOGISTIC REGRESSION

To fit a logistic regression model, use either the `glm` function or the `gam` function with a formula to specify the model and the `family` argument set to `binomial`. In this case the response variable is necessarily binary or two-valued. As an example, consider the built-in data frame `kyphosis`. A summary of the data frame produces the following:

```
> attach(kyphosis)
> summary(kyphosis)
```

```
  Kyphosis         Age              Number          Start
absent :64    Min.   :  1.00   Min.   : 2.000   Min.   : 1.00
present:17    1st Qu.: 26.00   1st Qu.: 3.000   1st Qu.: 9.00
              Median : 87.00   Median : 4.000   Median :13.00
              Mean   : 83.65   Mean   : 4.049   Mean   :11.49
              3rd Qu.:130.00   3rd Qu.: 5.000   3rd Qu.:16.00
              Max.   :206.00   Max.   :10.000   Max.   :18.00
```

The four variables in `kyphosis` are defined as follows:

`Kyphosis` A binary variable indicating the presence/absence of a postoperative spinal deformity called *Kyphosis*.

`Age` The age of the child in months.

`Number` The number of vertebrae involved in the spinal operation.

`Start` The beginning of the range of the vertebrae involved in the operation.

A convenient way of examining the bivariate relationship between each predictor and the binary response, `Kyphosis`, is with a set of boxplots produced by `plot.factor`:

```
> par(mfrow=c(1,3), cex = .7)
> plot.factor(kyphosis)
```

Setting the `mfrow` parameter to `c(1,3)` produces three plots in a row. The

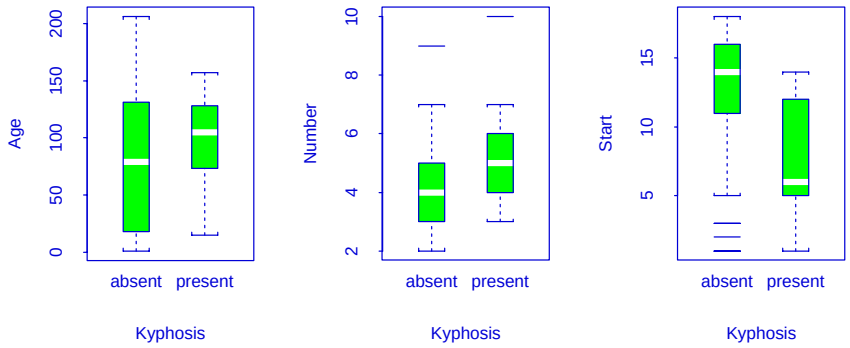

Figure 8.1: *Boxplots of the predictors of kyphosis versus* `Kyphosis`.

character expansion is set to 0.7 times the normal size using the `cex` parameter of the `par` function. Figure 8.1 displays the result.

Both `Start` and `Number` show strong location shifts with respect to the presence or absence of `Kyphosis`. `Age` does not show such a shift in location.

Fitting a Linear Model

The logistic model we start with relates the probability of developing Kyphosis to the three predictor variables, `Age`, `Number` and `Start`. We fit the model using `glm` as follows:

```
> kyph.glm.all <- glm(Kyphosis ~ Age + Number + Start,
+ family = binomial, data = kyphosis)
```

The `summary` function produces a summary of the resulting fit:

```
> summary(kyph.glm.all)
Call: glm(formula = Kyphosis ~ Age + Number + Start,
family = binomial, data = kyphosis)
Deviance Residuals:
      Min         1Q     Median         3Q     Max
-2.312363 -0.5484308 -0.3631876 -0.1658653 2.16133

Coefficients:
                  Value Std. Error   t value
(Intercept) -2.03693225 1.44918287 -1.405573
Age          0.01093048 0.00644419  1.696175
Number       0.41060098 0.22478659  1.826626
Start       -0.20651000 0.06768504 -3.051043

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 61.37993 on 77 degrees of freedom

Number of Fisher Scoring Iterations: 5

Correlation of Coefficients:
       (Intercept)        Age     Number
   Age -0.4633715
Number -0.8480574   0.2321004
 Start -0.3784028  -0.2849547  0.1107516
```

The summary includes:

1. a replica of the call that generated the fit,
2. a summary of the deviance residuals (more on these later),
3. a table of estimated regression coefficients, their standard errors, and the partial *t*-test of their significance,
4. estimates of the null and residual deviances (more on these later), and
5. a correlation matrix of the coefficient estimates.

The partial *t*-tests indicate that `Start` is important even after adjusting for `Age` and `Number`, but they provide little information on the other two variables.

You can produce an analysis of deviance for the sequential addition of each variable by using the `anova` function, specifying the chi-square test to test for differences between models:

```
> anova(kyph.glm.all, test = "Chi")
Analysis of Deviance Table

Binomial model

Response: Kyphosis

Terms added sequentially (first to last)
       Df Deviance Resid. Df Resid. Dev   Pr(Chi)
  NULL                    80   83.23447
   Age  1  1.30198        79   81.93249 0.2538510
Number  1 10.30593        78   71.62656 0.0013260
Start   1 10.24663        77   61.37993 0.0013693
```

Here we see that `Number` is important after adjusting for `Age`. We already know that `Number` loses its importance after adjusting for `Age` and `Start`. `Age` does not appear to be important as a linear predictor.

You can examine the bivariate relationships between the probability of Kyphosis and each of the predictors by fitting a "null" mode and then adding each of the terms, one at a time:

```
> kyph.glm.null <- glm(Kyphosis ~ 1, family = binomial,
+ data = kyphosis)
> add1(kyph.glm.null, ~ . + Age + Number + Start)
Single term additions

Model: Kyphosis ~ 1
             Df Sum of Sq      RSS       Cp
<none>                    81.00000 83.02500
Age           1   1.29546 79.70454 83.75454
Number        1  10.55222 70.44778 74.49778
Start         1  16.10805 64.89195 68.94195
```

The `Cp` statistic is used to compare models that are not nested. A small `Cp` corresponds to a better model in the sense of smaller residual deviance penalized by the number of parameters that are estimated in fitting the model.

Clearly `Start` is the best single variable to use in a linear model. These statistical conclusions, however, should be verified by looking at graphical displays of the fitted values and residuals.

The plot method for generalized linear models produces four plots:

1. a plot of deviance residuals versus the fitted values.
2. a plot of the square root of the absolute deviance residuals versus the linear predictor values.
3. a plot of the response versus the fitted values.
4. a Normal quantile plot of the Pearson residuals.

This set of plots is similar to those produced by the plot method for `lm` objects.

Systematic curvature in the residual plots could be indicative of problems in the choice of link, wrong scale of one of the predictors, or omission of a quadratic term in a predictor. Large residuals can be also be detected with these plots. These may be indicative of the need to remove the corresponding observations and re-fit the model. The plot of the absolute residuals against predicted values gives a visual check on the adequacy of the assumed variance function.

The Normal quantile plot is also generated for `glm` objects. This plot could be useful in the detection of extreme observations deviating from a general trend but one should exercise caution in not over-interpreting its shape, which is not necessarily of interest in the nonlinear context.

Figure 8.2 results from simply plotting the fit. Residual plots are not useful for binary data, such as `Kyphosis`, because all of the points line on one of two curves depending on whether the response is 0 or 1.

```
> par(mfrow=c(2,2))
> plot(kyph.glm.all, ask=F)
```

A more useful diagnostic plot is produced by `plot.gam`. By default, `plot.gam` plots the estimated relationship between the individual fitted terms and each of the corresponding predictors. You can request that partial residuals be added to the plot by specifying the argument `resid=T`. The scale argument can be used to keep all of the plots on the same scale to ease comparison. Figure 8.3 is produced by `plot.gam`:

```
> par(mfrow=c(1,3))
> plot.gam(kyph.glm.all, resid = T, scale = 6)
```

These plots give a quick assessment of how well the model fits the data through examination of the fit of each term in the formula. The plots are of the adjusted relationship for each predictor versus each predictor. When the relationship is specified as *linear* the label on the vertical axis reduces to the variable name. We will see the utility of this plot method and the reason for the labels when we plot additive models produced by `gam`.

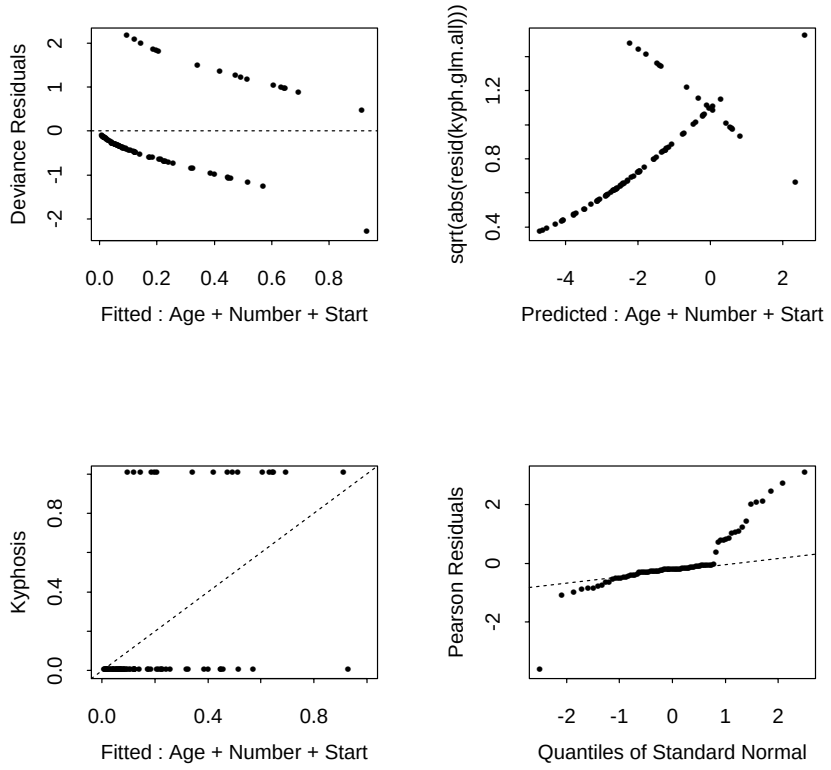

Figure 8.2: *Plots of the generalized linear model of* `Kyphosis` *predicted by* `Age`, `Start`, *and* `Number`.

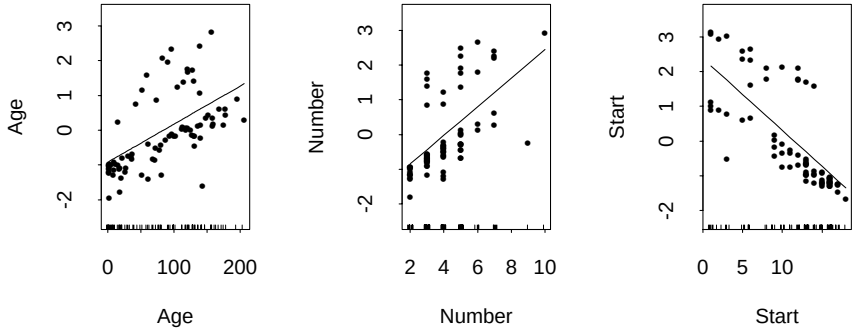

Figure 8.3: *Additional plots of the generalized linear model of* `Kyphosis` *predicted by* `Age`, `Number`, *and* `Start`.

Both `plot.glm` (the underlying plotting method for generalized linear models) and `plot.gam` produce multiple plots; you can, however, choose

which plots you look at by using the argument `ask=T` (the default). This produces a menu of available plots; you then select the number of the desired plot. For example, here is the menu of default GLM plots for the function `kyph.glm.all`:

```
> plot(kyph.glm.all, ask=T)

Make a plot selection (or 0 to exit):

1: plot: All
2: plot: Residuals vs Fitted Values
3: plot: Sqrt of abs(Residuals) vs Predictions
4: plot: Response vs Fitted Values
5: plot: Normal QQplot of Std. Residuals
Selection:
```

Fitting an Additive Model

So far we have examined only *linear* relationships between the predictors and the probability of developing Kyphosis. We can assess the validity of the linear assumption by fitting an *additive* model with relationships estimated by smoothing operations (cubic splines or local regression) and comparing it to the linear fit. We use the `gam` function to fit additive models.

```
> kyph.gam.all <-
+ gam(Kyphosis ~ s(Age) + s(Number) + s(Start),
+ family = binomial, data = kyphosis)
```

Including each variable as an argument to the `s` function instructs `gam` to estimate the "smoothed" relationships with each predictor by using cubic *B*-splines. Alternatively we could have used the `lo` function for local regression smoothing (`loess`). A summary of the fit is:

```
> summary(kyph.gam.all)

Call: gam(formula = Kyphosis ~ s(Age) +s(Number)+ s(Start),
family = binomial, data = kyphosis)
Deviance Residuals:
Min 1Q Median 3Q Max
-1.351358 -0.4439636 -0.1666238 -0.01061843 2.10851

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 40.75732 on 68.1913 degrees of freedom
```

```
Number of Local Scoring Iterations: 7

DF for Terms and Chi-squares for Nonparametric Effects

            Df Npar Df Npar Chisq    P(Chi)
(Intercept)  1
     s(Age)  1     2.9   5.782245 0.1161106
  s(Number)  1     3.0   5.649706 0.1289318
   s(Start)  1     2.9   5.802950 0.1139286
```

The summary of a `gam` fit is similar to the summary of a `glm` fit. One noticeable difference is the analysis of deviance table. For the additive fit the tests correspond to *approximate* partial tests for the importance of the smooth for each term in the model. These tests are typically used for screening variables for inclusion in the model. The approximate nature of these tests is discussed in detail in Hastie and Tibshirani (1990). For a single variable in the model, this is equivalent to testing for a difference between a linear fit and a smooth fit which includes a linear term along with the smooth term.

Now let's fit two additional models, adding a smooth of each of `Age` and `Number` to the base model which has a smooth of `Start`.

```
> kyph.gam.start.age <-
+       gam(Kyphosis ~ s(Start) + s(Age),
+       family = binomial, data = kyphosis)
> kyph.gam.start.number <-
+       gam(Kyphosis ~ s(Start) + s(Number),
+       family = binomial, data = kyphosis)
```

We produce the following analysis of deviance tables:

```
> anova(kyph.gam.start, kyph.gam.start.age, test="Chi")
Analysis of Deviance Table

Response: Kyphosis

              Terms Resid. Df Resid. Dev
1          s(Start)  76.24543   59.11262
2 s(Start) + s(Age)  72.09458   48.41713
     Test        Df  Deviance   Pr(Chi)
1
2 +s(Age)  4.150842  10.69548  0.0336071
```

```
> anova(kyph.gam.start, kyph.gam.start.number,
+ test="Chi")
Analysis of Deviance Table

Response: Kyphosis

                Terms Res.Df Res.Dev
1           s(Start) 76.245 59.1126
2 s(Start)+s(Number) 72.180 54.1790
         Test       Df Deviance   Pr(Chi)
1
2 +s(Number) 4.064954 4.933668 0.3023856
```

The indication is that `Age` is important in the model even with `Start` included whereas `Number` isn't important under the same conditions.

You can plot the fit with a smooth on `Age` and `Start` adding partial residuals while maintaining all figures on the same scale as follows:

```
> par(mfrow = c(2,2))
> plot(kyph.gam.start.age, resid = T, scale = 8)
```

Or you can simply plot the fit adding pointwise confidence intervals for the fit.

```
> plot(kyph.gam.start.age, se = T, scale = 10)
```

Figure 8.4 displays the resulting plots produced by `plot.gam`. Notice the vertical axes labels now. They reflect the smoothing operation included in the modeling.

The summary of the additive fit with smooths of `Age` and `Start` included appears as follows:

```
> summary(kyph.gam.start.age)

Call: gam(formula = Kyphosis ~ s(Start) + s(Age),
family = binomial, data = kyphosis)
Deviance Residuals:
       Min         1Q     Median          3Q      Max
 -1.694389 -0.4212112 -0.1930565 -0.02753535 2.087434

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 48.41713 on 72.09458 degrees of freedom
```

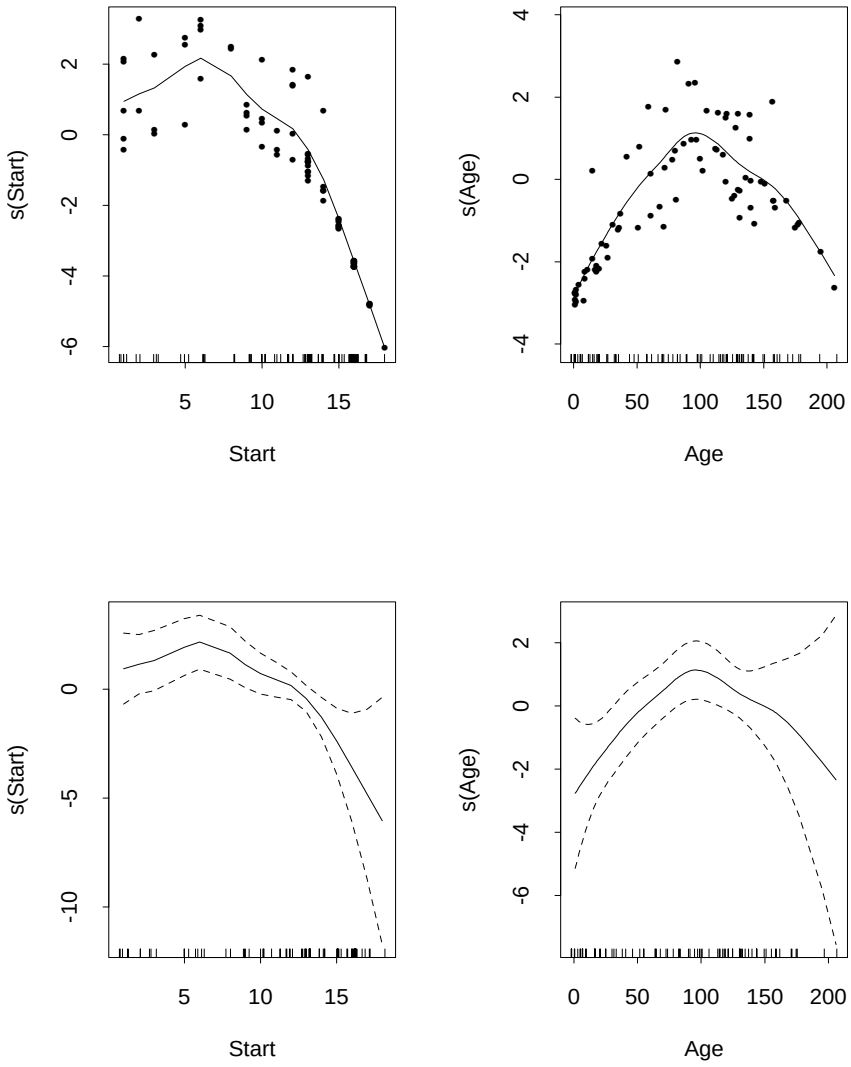

Figure 8.4: *The partial fits for the generalized additive logistic regression model of* `Kyphosis` *with* `Age` *and* `Start` *as predictors.*

```
Number of Local Scoring Iterations: 6

DF for Terms and Chi-squares for Nonparametric Effects
            Df Npar Df Npar Chisq    P(Chi)
(Intercept)  1
   s(Start)  1     2.9   7.729677 0.0497712
     s(Age)  1     3.0   6.100143 0.1039656
```

Returning to the Linear Model

The plots of the fits of the additive model displayed in figure 8.4 suggest a quadratic relationship for `Age` and a piecewise linear relationship for `Start`. It is useful to fit these suggested relationships as a linear model if the model is further simplified without losing too much precision in predicting the response.

For `Age` we fit a second degree polynomial. For `Start`, recall that its values indicate the beginning of the range of the vertebrae involved in the operation. Values less than or equal to 12 correspond to the thoracic region of the spine and values greater than 12 correspond to the lumbar region. Since the relationship for `Start` is fairly flat for values of `Start` approximately less than or equal to 12, and then drops off linearly for values greater than 12, we will try fitting a linear model with the term `I((Start - 12) * (Start > 12))`. The `I` function is used here to prevent the `"*"` from being used for factor expansion in the formula sense.

Figure 8.5 displays the resulting fit along with the partial residuals as well as the fit along with two standard errors bands.

The summary of the fit follows:

```
> summary(kyph.glm.istart.age2)

Call: glm(formula = Kyphosis ~ poly(Age, 2) +
   I((Start - 12) * (Start > 12)), family = binomial,
   data = kyphosis)

Deviance Residuals:
     Min         1Q     Median          3Q      Max
-1.42301 -0.5014355 -0.1328078 -0.01416602 2.116452
Coefficients:
                              Value Std. Error   t value
             (Intercept)  -0.6849607  0.4570976 -1.498500
           poly(Age, 2)1   5.7719269  4.1315471  1.397038
           poly(Age, 2)2 -10.3247767  4.9540479 -2.084109
I((Start-12)*(Start>12))  -1.3510122  0.5072018 -2.663658

(Dispersion Parameter for Binomial family taken to be 1 )

    Null Deviance: 83.23447 on 80 degrees of freedom

Residual Deviance: 51.95327 on 77 degrees of freedom

Number of Fisher Scoring Iterations: 6
```

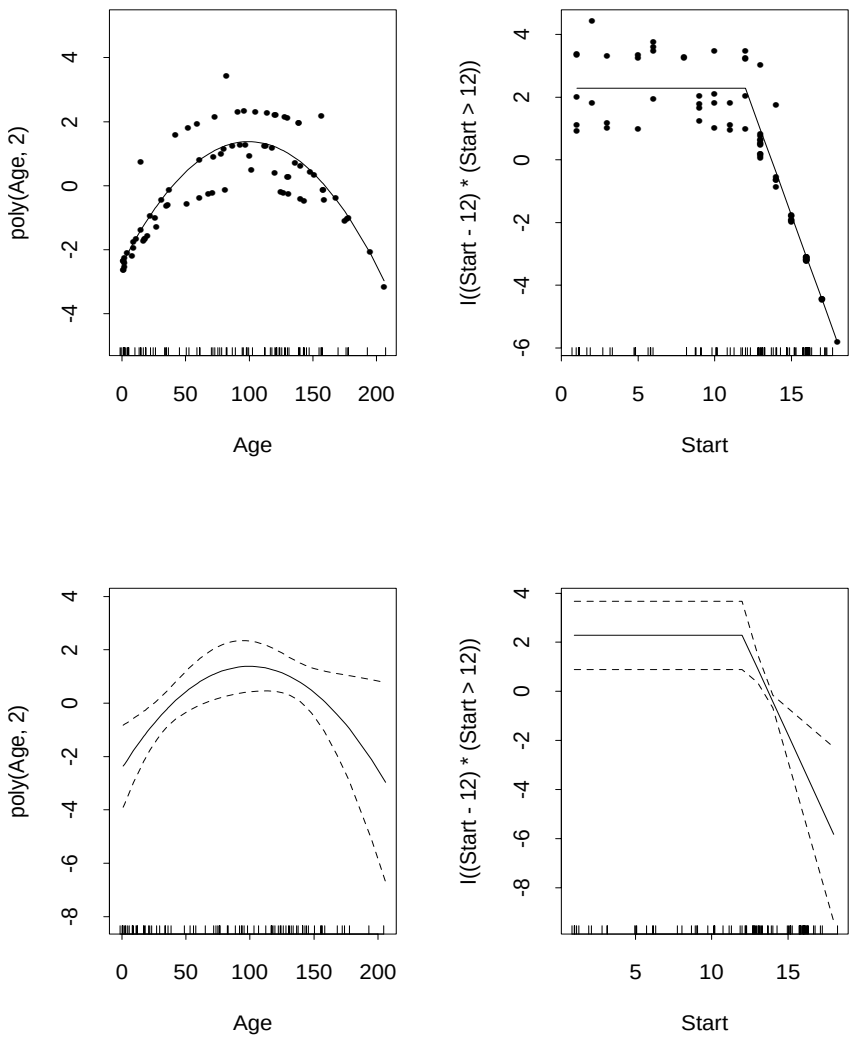

Figure 8.5: *The partial fits for the generalized linear logistic regression model of* `Kyphosis` *with quadratic fit for* `Age` *and piecewise linear fit for* `Start`.

```
Correlation of Coefficients:
                          (Intercept) poly(Age,2)1 poly(Age,2)2
            poly(Age, 2)1 -0.1133772
            poly(Age, 2)2  0.5625194  0.0130579
I((Start-12)*(Start>12)) -0.3261937 -0.1507199 -0.0325155
```

Contrasting the summary of this linear fit (`kyph.glm.istart.age2`) with the additive fit with smooths of `Age` and `Start` (`kyph.gam.start.age`) we can see the following important details:

1. The linear fit is more parsimonious; the effective number of parameters being estimated is approximately 5 less than for the additive model with smooths.
2. The residual deviance has increased by only about 3.5 even with a decrease in the effective number of parameters in fitting the linear model by about five. We use the `anova` function to verify that there is no difference between these models.

```
> anova(kyph.glm.istart.age2, kyph.gam.start.age,
+         test="Chi")

Analysis of Deviance Table

Response: Kyphosis

                                    Terms  Res. Df Res. Dev
1 poly(Age,2)+I((Start-12)*(Start>12)) 77.00000 51.95327
2                    s(Start) + s(Age) 72.09458 48.41713

     Test       Df Deviance   Pr(Chi)
1
2 1 vs. 2 4.905415 3.536134 0.6050618
```

3. Having fit a linear model, we can produce an *analytical* expression for the model, which we can't do for an additive model with smooth fits. This is because for a linear model, coefficients are estimated for a *parametric* relationship whereas for an additive model with smooth fits, the smooths are nonparametric estimates of the relationship. In general, these nonparametric estimates have no analytical form and are based on an iterative computer algorithm. This is an important distinction between linear models and additive models with smooth terms.

8.2 POISSON REGRESSION

To fit a Poisson regression model use either the `glm` function or the `gam` function with a formula to specify the model and the `family` argument set to `poisson`. In this case the response variable is discrete taking on non-negative integer values. Count data is frequently modeled as a Poisson

distribution. As an example, consider the built-in data frame `solder.balance`. A summary of the data frame produces the following:

```
> attach(solder.balance)
> summary(solder.balance)
 Opening    Solder    Mask   PadType  Panel        skips
 S:240   Thin :360 A1.5:180  L9 : 72  1:240  Min.   : 0.000
 M:240   Thick:360 A3  :180  W9 : 72  2:240  1st Qu.: 0.000
 L:240             B3  :180  L8 : 72  3:240  Median : 2.000
                   B6  :180  L7 : 72          Mean : 4.965
                             D7 : 72          3rd Qu.: 6.000
                             L6 : 72            Max. :48.000
                          (Other):288
```

The *solder* experiment, contained in `solder.balance`, was designed and implemented in one of AT&T's factories to investigate alternatives in the "wave-soldering" procedure for mounting electronic components on circuit boards. Five different factors were considered as having an effect on the number of solder skips. A brief description of each of the factors follows. For more details, see the paper by Comizzoli, Landwehr, and Sinclair (1990).

`Opening`	amount of clearance around the mounting pad
`Solder`	amount of solder
`Mask`	type and thickness of the material used for the solder mask
`PadType`	the geometry and size of the mounting pad
`Panel`	each board was divided into three panels, with three runs on a board
`skips`	number of visible solder skips on a circuit board.

Two useful preliminary plots of the data are a histogram of `skips`, the response, and plots of the mean response for each level of the predictor. Figure 8.6 and figure 8.7 display the resulting plots.

```
> par(mfrow=c(1,1))
> hist(skips)
> plot(solder.balance)
```

The histogram of `skips` in figure 8.6 shows the skewness and long-tailedness typical of count data. We will model this using a Poisson distribution.

The plot of the mean skips for different levels of the factors displayed in figure 8.7 shows a very strong effect due to `Opening`. For levels `M` and `L` only about two skips were seen on average, whereas for level `S`, more then 10 skips were seen. Effects almost as strong were seen for different levels of `Mask`.

If we do boxplots of `skips` for each level of the two factors, `Opening` and

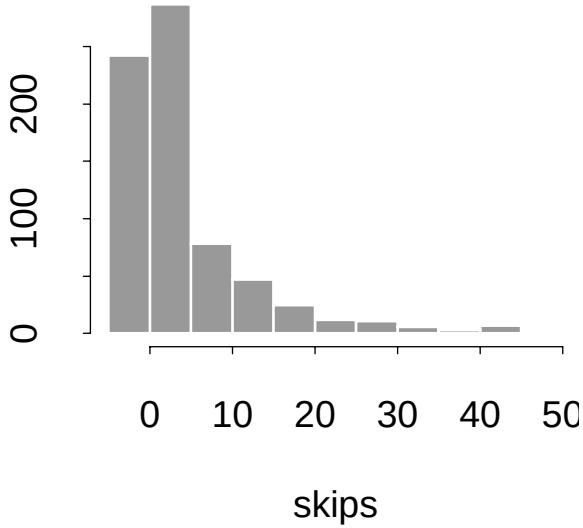

Figure 8.6: *A histogram of* `skips` *for* `solder` *data.*

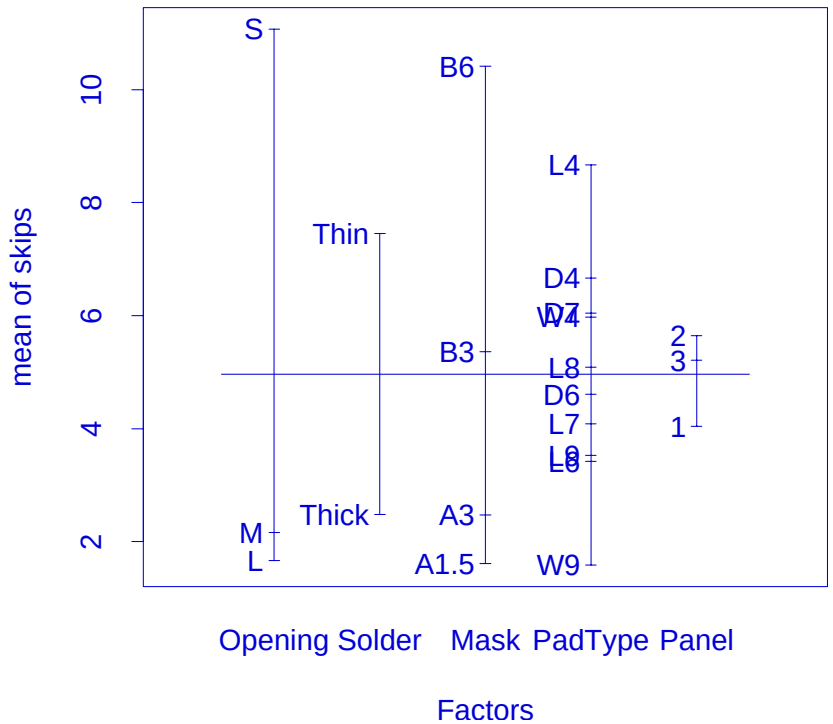

Figure 8.7: *A plot of the mean response for each level of each factor.*

Mask, we get an idea of the distribution of the data across levels of the factors. Figure 8.8 displays the results of doing "factor" plots on these two factors.

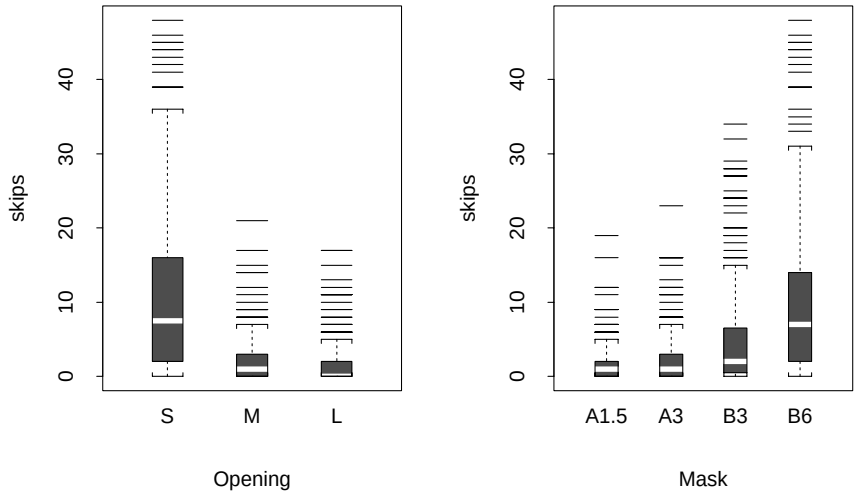

Figure 8.8: *Boxplots for each level of the two factors* Opening *and* Mask.

```
> par(mfrow=c(1,2))
> plot.factor(skips ~ Opening + Mask)
```

On examining figure 8.8, it is clear that the variance of skips increases as its mean increases. This is typical of Poisson distributed data.

We proceed now to model skips, using glm, as a function of the controlled factors in the experiment declaring family = poisson. We start with a simple-effects model for skips as follows:

```
> paov <- glm(skips ~ . , family = poisson,
+             data = solder.balance)
> anova(paov, test = "Chi")
Analysis of Deviance Table

Poisson model

Response: skips

Terms added sequentially (first to last)
        Df Deviance Resid. Df Resid. Dev      Pr(Chi)
   NULL                   719   6855.690
Opening  2 2524.562       717   4331.128 0.000000e+00
```

```
  Solder  1  936.955       716   3394.173 0.000000e+00
    Mask  3 1653.093       713   1741.080 0.000000e+00
 PadType  9  542.463       704   1198.617 0.000000e+00
   Panel  2   68.137       702   1130.480 1.554312e-15
```

The chi-squared test is requested in this case because `glm` assumes $\phi = 1$ (no under- or over-dispersion). We use the quasi-likelihood family, `quasi`, when we want to estimate the scale parameter as part of the model fitting computations for binomial or Poisson families. We could also set the argument `disp` in the `summary` function to 0 to obtain this estimate while summarizing the fitted model.

According to the analysis of deviance, it appears that all of the factors considered have a very significant influence on the number of solder skips. The solder experiment contained in `solder.balance` is balanced, so we need not be concerned with the sequential nature of the analysis of deviance table above; the tests of a sequential analysis are identical to the *partial* tests of a regression analysis when the experiment is balanced.

Now let's fit a second order model. We fit all the simple effects and all the second order terms except those including `Panel` (we have looked ahead and discovered that the interactions with `Panel` are non-significant, marginal, or of less importance than the other interactions). The analysis of deviance table follows:

```
> paov2 <- glm(skips ~ . +
+          (Opening + Solder + Mask + PadType)  2,
+              family = poisson, data = solder.balance)
> anova(paov2, test = "Chi")
Analysis of Deviance Table

Poisson model

Response: skips

Terms added sequentially (first to last)
                Df Deviance Res.Df Resid. Dev      Pr(Chi)
           NULL                719   6855.690
        Opening  2 2524.562    717   4331.128 0.0000000000
         Solder  1  936.955    716   3394.173 0.0000000000
           Mask  3 1653.093    713   1741.080 0.0000000000
        PadType  9  542.463    704   1198.617 0.0000000000
          Panel  2  68.137     702   1130.480 0.0000000000
 Opening:Solder  2  27.978     700   1102.502 0.0000008409
```

```
   Opening:Mask  6  70.984     694   1031.519 0.0000000000
Opening:PadType 18  47.419     676    984.100 0.0001836068
    Solder:Mask  3  59.806     673    924.294 0.0000000000
 Solder:PadType  9  43.431     664    880.863 0.0000017967
   Mask:PadType 27  61.457     637    819.407 0.0001694012
```

All of the interactions estimated in `paov2` are quite significant.

To verify the fit we do several different kinds of plots. The first four result from doing the standard plot for a `"glm"` object.

```
> par(mfrow=c(2,2))
> plot(paov2)
```

The resulting plot is displayed in figure 8.9. The plot of the observations

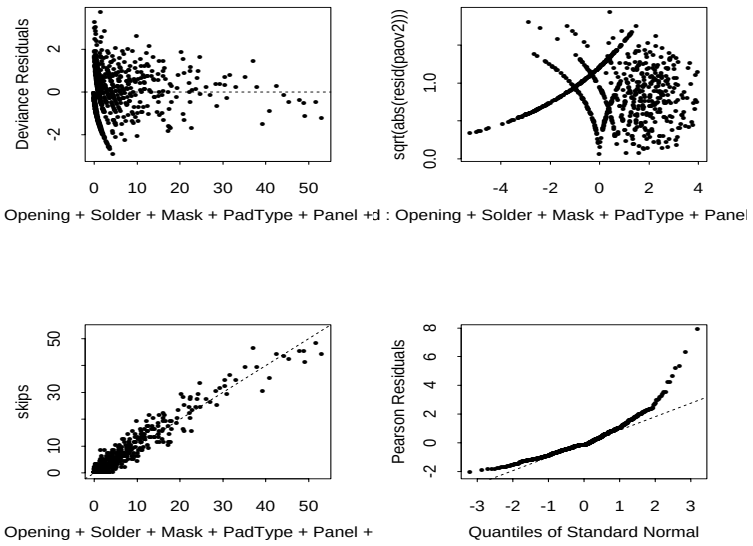

Figure 8.9: *Plots of the second order model of* `skips`.

versus the fitted values shows no great departures from the model. The plot of the absolute deviance residuals shows striations due to the discrete nature of the data. Otherwise the deviance residual plot doesn't reveal anything to make us uneasy about the fit.

The other set of plots useful for examining the fit is produced by `plot.gam`. These are plots of the adjusted fit with partial residuals overlaid for each predictor variable. Since all the variables are factors, the resulting fit is a step function; a constant is fitted for each level of a factor. Figure 8.10 displays the resulting plots.

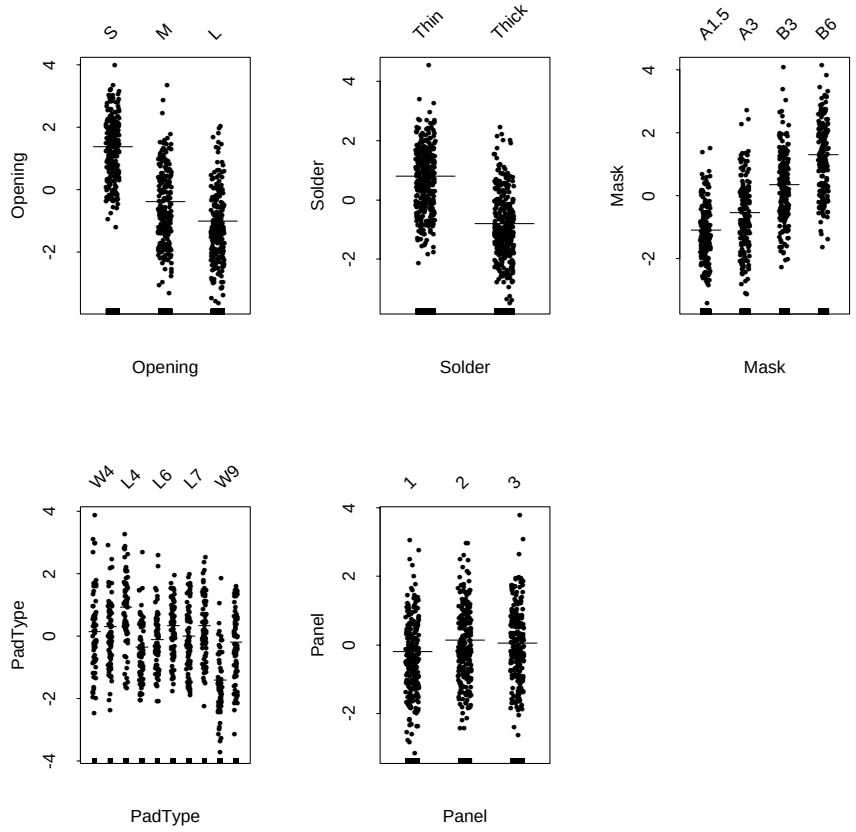

Figure 8.10: *Partial residual plots of the second order model of* `skips`.

```
> par(mfrow=c(2,3))
> plot.gam(paov2, resid = T)
```

The `plot.gam` function adds a bit of random noise to the coded factor levels to spread the plotted points out so that it is easier to see their vertical locations. (Note: the warning message about interaction terms not being saved can be safely ignored here.)

These plots also indicate that the data is modeled reasonably well. Please note, however, that the default plots will show only *glaring* lack of fit.

8.3 GENERALIZED LINEAR MODELS

The linear model discussed in chapter 7, Regression and Smoothing for Continuous Response Data, is a special case of the generalized linear model. A linear model provides a way of estimating the response variable, *Y*,

conditional on a linear function of the values, x_1, x_2, ..., x_p, of some set of predictors variables, X_1, X_2, ..., X_p. Mathematically, we write this as:

$$E(Y|x) = \beta_0 + \sum_{i=1}^{p} \beta_i x_i \tag{8.1}$$

For the linear model the variance of Y is assumed constant and denoted by $\text{var}(Y)=\sigma^2$.

A *generalized* linear model provides a way to estimate a *function* (called the *link* function) of the mean response as a linear function of the values of some set of predictors. This is written as:

$$g(E(Y|\boldsymbol{x})) = g(\mu) = \beta_0 + \sum_{i=1}^{p} \beta_i x_i = \eta(\boldsymbol{x}) \tag{8.2}$$

where g is the link function. The linear function of the predictors, $\eta(\boldsymbol{x})$, is called the *linear predictor*. For the generalized linear model, the variance of Y may be a function of the mean response μ:

$$\text{var}(Y) = \phi V(\mu)$$

The logistic regression and Poisson regression examples we have seen are special cases of the generalized linear model. To do a logistic regression we declare the `binomial` family which uses the *logit* link function defined by

$$g(p) = \text{logit}(p) = log\frac{p}{1-p}$$

and variance function defined by

$$\text{var}(Y) = \phi\frac{p}{1-p}$$

where p is the probability of an event occurring. The parameter p corresponds to the mean response of a binary (0-1) variable. In logistic regression, we model the probability of some event occurring as a linear function of a set of predictors. Usually, for the logistic regression problem ϕ is fixed to be 1 (one).

When we cannot assume that $\phi = 1$ (this is the case of over- or under-dispersion discussed in McCullagh and Nelder (1989)), we must use the `quasi` family for quasi-likelihood estimation. The quasi-likelihood "family" allows us to estimate the parameters in the model without specifying what the distribution function is. In this case the link and variance functions are all that is used for fitting the model. Once these are known, the same iterative procedure that is used for fitting the other families can be used to estimate the model parameters. For more detail, see Chambers and Hastie (1992) and McCullagh and Nelder (1989).

The Poisson regression example declares a `poisson` family with the *log* link function

$$g(\mu) = log(\mu)$$

and the variance defined by

$$\text{var}(Y) = \phi\mu$$

The binomial and Poisson families are for fitting regression models to categorical response data. For the binomial case, the response is a binary variable indicating whether or not some event has occurred. The most common example of using the binomial family is the logistic regression problem where we try to predict the probability of the event occurring as a function of the predictors. Some examples of a binary response are presence/absence of AIDS, presence/absence of a plant species in a vegetation sample, failure/non-failure of a electronic component in a radio.

The Poisson family is useful for modeling counts which typically follow a Poisson distribution. Our earlier example modeled the number of soldering skips as a function of various controlled factors in the solder experiment.

Other families are available for modeling other kinds of data. For example normal (the linear model special case) and inverse normal distributions are modeled with the `gaussian` and `inverse.gaussian` families. Table 8.1 lists the distribution families available for use with either the `glm` or the `gam` function.

Each of these families represents an exponential family of distributions of a particular form. The link function for each family listed in table 8.1 is referred to as the *canonical* link because it relates the canonical parameter of the distribution family to the linear predictor, $\eta(x)$. For more details, on the parameterization of these distributions, see McCullagh and Nelder (1989).

The estimates of the regression parameters in a **glm** are maximum likelihood estimates, produced by iteratively reweighted least-squares (IRLS).

Table 8.1: *Link and variance functions for the generalized linear and generalized additive models.*

Distribution	Family	Link	Variance
Normal/Gaussian	`gaussian`	μ	1
Binomial	`binomial`	$\log(\mu/(1-\mu))$	$\mu(1-\mu)/n$
Poisson	`poisson`	$\log(\mu)$	μ
Gamma	`gamma`	$1/\mu$	μ^2
Inverse Normal/ Gaussian	`inverse.gaussian`	$1/\mu^2$	μ^3
Quasi	`quasi`	$g(\mu)$	$V(\mu)$

Essentially, the log-likelihood, $l(\beta,y)$, is maximized by solving the *score* equations, defined by:

$$\partial l(\beta, y)/\partial\beta = 0 \tag{8.3}$$

Since the score equations are nonlinear in β, they are solved iteratively. This iterative procedure is what is referred to as IRLS. For more details, see Chambers and Hastie (1992) or McCullagh and Nelder (1989).

8.4 GENERALIZED ADDITIVE MODELS

Section 8.3, Generalized Linear Models, discusses an extension of linear models to data with error distributions other than normal or Gaussian. By using **glm**, we can fit data with Gaussian, binomial, Poisson, gamma, or inverse Gaussian errors, which extends dramatically the kind of data for which we can build regression models. The primary restriction of a **glm** is the fact that it is still a *linear* model. The linear predictor is just that, a linear function of the parameters of the model.

The generalized additive model, **gam**, extends the **glm** by fitting nonparametric functions to estimate the relationships between the response and the predictors. The nonparametric functions are estimated from the data using smoothing operations.

The general form of a **gam** is:

$$g(E(Y|\boldsymbol{x})) = g(\mu) = \alpha + \sum_{i=1}^{p} f_i(x_i) = \eta(\boldsymbol{x}) \tag{8.4}$$

where g is the link function, α is a constant intercept term, f_i corresponds to the nonparametric function describing the relationship between the transformed mean response (the link transform function) and the ith predictor. In this context, $\eta(\boldsymbol{x})$ is referred to as the *additive* predictor and is entirely analogous to the *linear* predictor of a **glm** defined in (8.2). As for a **glm**, the variance of Y may be function of the mean response μ:

$$\text{VAR}(Y) = \phi V(\mu)$$

All of the distribution families listed in table 8.1 are available for **gam**s. Thus fully nonparametric, nonlinear additive regression models can be fit to binomial data (logistic regression) and count data (Poisson regression) as presented in sections 8.1 and 8.2 as well as to data with error distributions that are modeled by the other families listed in table 8.1.

Two functions that are useful for fitting a **gam** are `s` and `lo`. Both of these functions are for fitting smooth relationships between the transformed response and the predictors. The `s` function fits cubic B-splines to estimate the smooth and `lo` fits a locally weighted least-squares regression to estimate the smooth. For more detail on using these functions, see their help files.

8.5 QUASI-LIKELIHOOD ESTIMATION

Quasi-likelihood estimation allows you to estimate regression relationships without fully knowing the error distribution of the response variable. Essentially, you provide link and variance functions which are used in the estimation of the regression coefficients. Although the link and variance functions are typically associated with a *theoretical* likelihood, the likelihood need not be specified and fewer assumptions are made in estimation and inference.

As a simple analogy, there is a connection between normal-theory regression models and least-squares regression estimates. Least-squares estimation gives identical parameter estimates to those produced from normal-theory models. However, least-squares estimation assumes far less; only second moment assumptions are made by least-squares compared to full distribution assumptions of normal-theory models.

Quasi-likelihood estimation for the distributions of table 8.1 is analogous to least-squares estimation for the normal distribution. For the Gaussian family, IRLS is equivalent to standard least-squares estimation. Used in this context, quasi-likelihood estimation allows us to estimate the dispersion parameter in under- or over-dispersed regression models. For example, an under- or over-dispersed logistic or Poisson regression model can be estimated by using quasi-likelihood methodology and supplying the appropriate link and variance functions for the binomial and Poisson families, respectively.

However, quasi-likelihood estimation extends beyond the families represented in table 8.1. Any modeling situation for which suitable link and variance functions can be derived can be modeled using the quasi-likelihood methodology. Several good examples of this kind of application are presented in McCullagh and Nelder (1989).

For our example of quasi-likelihood estimation, let's go back to the the Poisson regression example using the `solder.balance` data frame. Recall that we modeled `skips` as a function of all the factors plus all the two-way interactions except those including `Panel`. The modeling call was:

```
> glm(formula = skips ~ . +
+      (Opening + Solder + Mask + PadType)^2,
+            family = poisson, data = solder.balance)
```

When we declare the `family` argument to be either `Poisson` or `binomial`, the dispersion parameter is set to a constant equal to one. In many problems this assumption is not valid. We can use quasi-likelihood estimation to force the *estimation* of the dispersion parameter for these families. For the solder experiment we do it as follows:

```
> paov3 <-glm(formula = skips ~ . +
+   (Opening + Solder + Mask + PadType) ^ 2,
+  family = quasi(link = "log", var = "mu"),
+          data = solder.balance)
```

A summary of the fit reveals that the dispersion parameter is estimated to be 1.4, suggesting over-dispersion. We now recompute the ANOVA table, computing *F*-statistics for testing for effects:

```
> anova(paov3, test = "F")
Analysis of Deviance Table

Quasi-likelihood model

Response: skips
```

```
Terms added sequentially (first to last)
              Df Deviance R.Df Res. Dev  F Value      Pr(F)
           NULL              719 6855.690
       Opening  2 2524.562  717 4331.128 901.1240 0.00000000
        Solder  1  936.955  716 3394.173 668.8786 0.00000000
          Mask  3 1653.093  713 1741.080 393.3729 0.00000000
       PadType  9  542.463  704 1198.617  43.0285 0.00000000
         Panel  2   68.137  702 1130.480  24.3210 0.00000000
Opening:Solder  2   27.978  700 1102.502   9.9864 0.00005365
  Opening:Mask  6   70.984  694 1031.519   8.4457 0.00000001
Opening:PadType 18   47.419  676  984.100   1.8806
0.01494805
   Solder:Mask  3   59.806  673  924.294  14.2316 0.00000001
Solder:PadType  9   43.431  664  880.863   3.4449 0.00036929
  Mask:PadType 27   61.457  637  819.407   1.6249 0.02466031
```

All of the factors and interactions are still significant even when we model the over-dispersion. This gives us more assurance in our previous conclusions.

8.6 RESIDUALS

Residuals are our principal tool for assessing how well a model fits the data. For regression models, residuals are used to assess the importance and relationship of a term in the model as well as to search for anomalous values. For generalized models we have the additional task of assessing and verifying the form of the variance as a function of the mean response.

Generalized models require a generalization of the residual which will be applicable to all the distributions which replace the normal or Gaussian distribution and which can be used in the same way as the normal residuals of the linear model. In fact, four different kinds of residuals are defined for use in assessing how well a model fits, in determining the form of the variance function, and in diagnosing problem observations.

`"deviance"` Deviance residuals are defined as:

$$r_i^D = \text{sign}(y_i - \hat{\mu}_i)\sqrt{d_i}$$

where d_i is the contribution of the ith observation to the deviance.

The deviance itself is $D = \sum_i (r_i^D)^2$. Consequently, these residuals are reasonable for use in detecting observations with unduly large influence in the fitting process, since they reflect the same criterion as used in the fitting.

`"working"` Working residuals are the difference between the *working* response and the linear predictor at the final iteration of the IRLS algorithm. They are defined as:

$$r_i^W = (y_i - \hat{\mu}_i)\frac{\partial \hat{\eta}_i}{\partial \hat{\mu}_i}$$

These residuals are the ones you get when you extract the `residuals` component directly from the `glm` object.

`"pearson"` The Pearson residuals are defined as:

$$r_i^P = \frac{y_i - \hat{\mu}_i}{\sqrt{\hat{V}(\mu_i)}}$$

Their sum-of-squares,

$$X^2 = \sum_{i=1}^{n} \frac{(y_i - \hat{\mu}_i)^2}{\hat{V}(\mu_i)}$$

is the chi-squared statistic. Pearson residuals are a rescaled version of the working residuals. When proper account is taken of the associated weights, $r_i^P = \sqrt{w_i} r_i^W$.

`"response"` The response residuals are simply $y_i - \hat{\mu}_i$.

You compute residuals for `"glm"` and `"gam"` objects with the `residuals` function, abbreviated `resid` (or `resid` for short) function. The `type` argument allows you to specify one of `"deviance"`, `"working"`, `"pearson"`, or `"response"`. By default you get the deviance residuals, so to plot the deviance residuals versus the fitted values of a model you just do:

```
> plot(fitted(glmobj), resid(glmobj))
```

Alternatively, to plot the Pearson residuals versus the fitted values you do:

```
> plot(fitted(glmobj), resid(glmobj, type = "pearson"))
```

Selecting which residual to plot is somewhat a matter of personal preference. The deviance residual is the default because a large deviance residual corresponds to an observation which does not fit the model well in the same sense that a large residual for the linear model doesn't fit well. You can find additional detail on residuals in McCullagh and Nelder (1989).

8.7 PREDICTION FROM THE MODEL

Prediction for generalized linear models, **glm**, and generalized additive models, **gam**, is similar to prediction for linear models. The only important

point to remember is that for either of the generalized models predictions can be on one of two scales. You can predict:

1. on the scale of the linear predictor, which is the *transformed/* scale after applying the link function, or
2. on the scale of the original response variable.

Since prediction is based on the linear predictor, $\eta(\boldsymbol{x})$, computing predicted values on the scale of the original response effectively transforms the linear predictor evaluated at the predictor data back to the scale of the response via the inverse link function.

The `type` argument to either `predict.glm` or `predict.gam` allows you to choose one of three options for predictions:

`"link"`	Computes predictions on the scale of the linear predictor (the *link* scale).
`"response"`	Computes predictions on the scale of the response.
`"terms"`	Computes a matrix of predictions on the scale of the linear predictor, one column for each term in the model.

Specifying `type = "terms"` allows you to compute the component of the prediction for each term separately. Summing the columns of the matrix and adding the constant (intercept) term is equivalent to specifying `type = "link"`.

Predicting the Additive Model of Kyphosis

As an example, consider the generalized additive model of Kyphosis modeled as smooths of `Start` and `Age`. Recall the fit was saved as `kyph.gam.start.age`:

```
> kyph.gam.start.age
Call:
gam(formula = Kyphosis ~ s(Start) + s(Age),
family = binomial, data = kyphosis)

Degrees of Freedom: 81 total; 72.09458 Residual
Residual Deviance: 48.41713
```

If we are interested in plotting the prediction surface over the range of the data we start by generating appropriate sequences of values for each predictor and storing them in a data frame with variable labels that correspond to the variables in the model:

```
> attach(kyphosis)
> kyph.margin <-
+    data.frame(Start = seq(from = min(Start),
```

```
+    to = max(Start), len = 40), Age =
+    seq(from = min(Age), to = max(Age), len = 40) )
```

Since a **gam** is additive, we need to do predictions only at the margins and then sum them together to form the entire prediction surface. We produce the marginal fits by specifying `type = "terms"`.

```
> margin.fit <- predict(kyph.gam.start.age, kyph.margin,
+                type="terms")
```

Now generate the surface for the marginal fits.

```
> kyph.surf <- outer(margin.fit[,1], margin.fit[,2], "+")
> kyph.surf <- kyph.surf + attr(margin.fit, "constant")
> kyph.surf <- binomial()$inverse(kyph.surf)
```

The first line adds the marginal pieces of the predictions together to create a matrix of surface values, the second line adds in the constant intercept term, and the third line applies the inverse link function to transform the predictions back to the scale of the original response. Now we produce the plot using the `persp` function (or `contour` or `image` if we wish):

```
> persp(kyph.margin[,1], kyph.margin[,2], kyph.surf,
+ xlab = "Start", ylab = "Age", zlab = "Kyphosis")
```

Figure 8.11 displays the resulting plot.

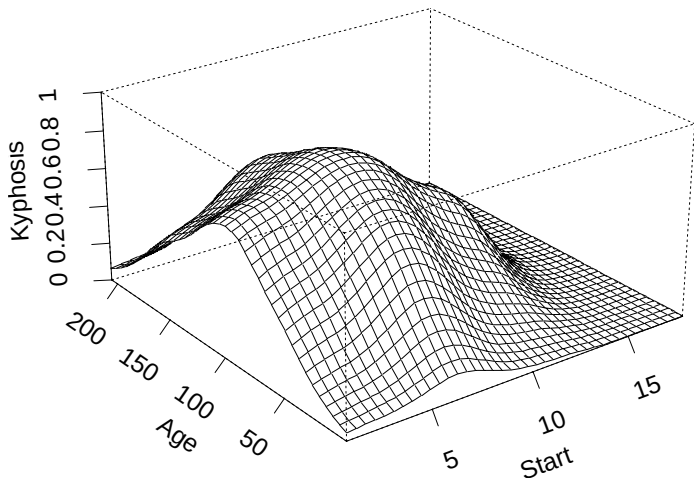

Figure 8.11: *Plot of the probability surface for developing Kyphosis based age in months and start position.*

Safe Prediction

Prediction for linear and generalized linear models is a two-step procedure.

1. Compute a model matrix using the new data where you want predictions.
2. Multiply the model matrix by the coefficients extracted from the fitted model.

This procedure works perfectly fine as long as the model has no *composite* terms which are dependent on some overall summary of a variable such as any of the following:

```
(x - mean(x))/sqrt(var(x))
(x - min(x))/diff(range(x))
poly(x)
bs(x)
ns(x)
```

The reason the procedure doesn't work for such composite terms is that the resulting coefficients are dependent on the summaries used in computing the terms. If the new data are different from the original data used to fit the model (which is more than likely when you provide new data), the coefficients are inappropriate. One way around this problem is to eliminate such dependencies on data summaries. For example, change `mean(x)` and `var(x)` to their numeric values rather than computing them from the data at the time of fitting the model. For the spline functions, `bs` and `ns`, provide the knots explicity in the call to the function rather than letting the function compute them from the overall data. If the removal of dependencies on the overall data is possible, prediction can be made safe for new data. However, when the dependencies cannot be removed (e.g., using `s` or `lo` in a **gam**), there is a function for doing prediction in as safe a way as possible given the need for generality. The function is `predict.gam`, which works as follows when new data is supplied:

1. A new data frame, `both.data`, is constructed by combining the data used to produce the fit, say `old.data`, and the new data in `new.data`.
2. The model frame and model matrix are constructed from the combined data frame `both.data`. The model matrix is separated into two pieces X^0 and X^n corresponding to the old and new data.
3. The parametric part of fit is refit using X^0.

4. The coefficients from this new fit are then applied to X^n to obtain the new predictions.
5. For `"gam"` objects with both parametric and nonparametric components, an additional step is taken to evaluate the fitted nonlinear functions at the new data values.

This procedure works perfectly for terms with `mean` and `var` in them as well as for `poly`. For other kinds of composite therms, it works approximately. For example, for `bs` knots are placed at equally spaced (in terms of percentiles) quantiles of the distribution of the predictor. Because the knots produced by the combined data will, in general, be different from the knots produced by the original data there will be some error in predicting the new data. If the old data and the new data have roughly the same distribution the error in predicting the new data should be small.

8.8 REFERENCES

Chambers, J.M. and Hastie, T.J. (1992). *Statistical Models in S*. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove, CA.

Comizzoli, R. B. and Landwehr, J. M. and Sinclair, J. D. (1990). *Robust Materials and Processes: Key to Reliability,* 6:113–128.

Hastie, T. and Tibshirani, R. (1990). *Generalized Additive Models*. Chapman and Hall, London.

McCullagh, P. and Nelder, J.A. (1989). *Generalized Linear Models*, 2nd edition. Chapman and Hall, London.

Nelder, J.A. and Wedderburn, R.W.M. (1972). *Generalized linear models*. Journal of the Royal Statistical Society, Series A, 135:370–384.

LOCAL REGRESSION MODELS

9

Local regression models provide greater flexibility in fitting a surface to data.

LOCAL REGRESSION MODELS

In both chapter 7, Regression and Smoothing for Continuous Response Data, and chapter 8, Generalizing the Linear Model, we discuss fitting curves or surfaces to data. In both of these earlier chapters, a significant limitation on the surfaces considered was that the effects of the terms in the model were expected to enter the model *additively*, without interactions between terms.

Local regression models provide much greater flexibility in that the model is fitted as a single smooth function of all the predictors. There are no restrictions on the relationships among the predictors.

Local regression models in S-PLUS are created using the `loess` function, which uses locally weighted regression smoothing, as described in section 7.11, Locally Weighted Regression Smoothing. In that section, the focus was on the smoothing function as an estimate of one predictor's contribution to the model. In this chapter, we use locally weighted regression to fit the complete regression surface.

9.1 FITTING A SIMPLE MODEL

As a simple example of a local regression model, we return to the `ethanol` data discussed in chapter 7, Regression and Smoothing for Continuous Response Data. We start by considering only the two variables `NOx` and `E`. We smoothed these data with `loess.smooth` in section 7.11. Now we use `loess` to create a complete local regression model for the data.

We fit an initial model to the ethanol data as follows, using the argument `span=1/2` to specify that each local neighborhood should contain about half of the observations:

```
> ethanol.loess <- loess(NOx ~ E, data = ethanol,
+ span = 1/2)
> ethanol.loess
Call:
loess(formula = NOx ~ E, data = ethanol, span = 1/2)

Number of Observations:          88
Equivalent Number of Parameters: 6.2
Residual Standard Error:         0.3373
Multiple R-squared:              0.92
Residuals:
    min   1st Q   median  3rd Q    max
-0.6656 -0.1805 -0.02148 0.1855 0.8656
```

The *equivalent number of parameters* gives an estimate of the complexity of the model. The number here, 4.3, indicates that the local regression model is somewhere between a cubic polynomial and a quartic polynomial in complexity. The default print method for `"loess"` objects also includes the residual standard error, multiple R^2, and a five number summary of the residuals.

9.2 DIAGNOSTICS: EVALUATING THE FIT

How good is our initial fit? The following function calls plot the `loess` object against a scatter plot of the original data:

```
> attach(ethanol)
> plot(ethanol.loess, xlim=range(E),
+ ylim=range(NOx,fitted(ethanol.loess)))
> points(E, NOx)
```

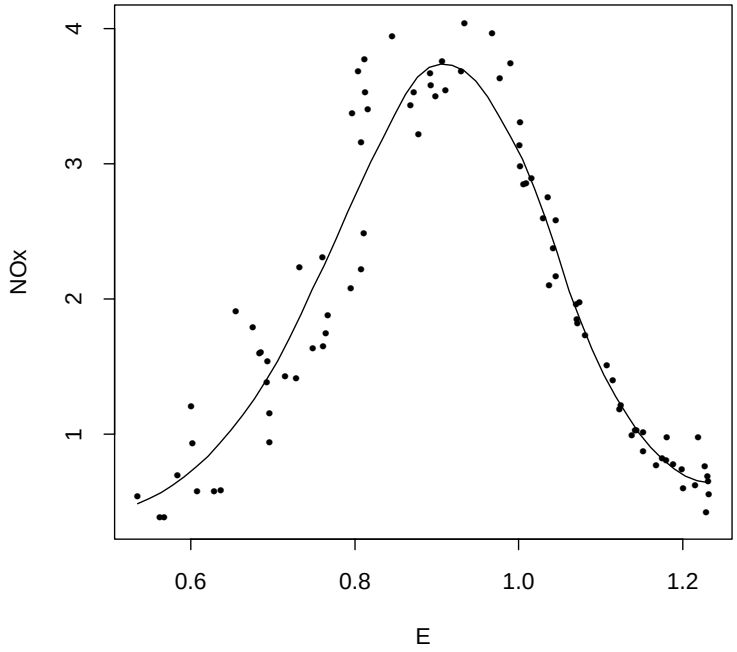

Figure 9.1: *Locally weighted smooth of ethanol data.*

The resulting figure, shown in figure 9.1, captures the trend reasonably well. The following expressions plot the residuals against the predictor `E` to check for lack of fit:

```
> scatter.smooth(E, resid(ethanol.loess), span=1, degree =1)
> abline(h=0)
```

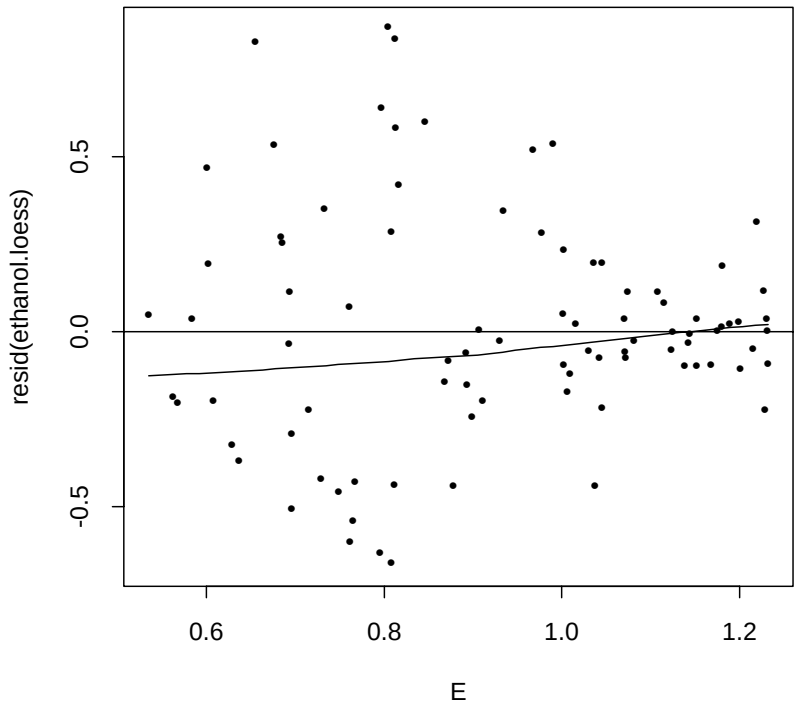

Figure 9.2: *Residual plot for* `loess` *smooth.*

The resulting plot, shown in figure 9.2, indicates no lack of fit.

Is there a surplus of fit? That is, can we increase the span of the data and still get a good fit? To see, let's refit our model, using `update`:

```
> ethanol.loess2 <- update(ethanol.loess, span=1)
> ethanol.loess2
Call:
loess(formula = NOx ~ E, data = ethanol, span = 1)
Number of Observations:          88
Equivalent Number of Parameters: 3.5
Residual Standard Error:         0.5126
Multiple R-squared:              0.81
Residuals:
     min   1st Q median  3rd Q    max
-0.9791 -0.4868 -0.064 0.3471 0.9863
```

By increasing the span, we reduce somewhat the equivalent number of parameters; this model is thus more *parsimonious* than our first model. We do seem to have lost some fit and gained some residual error. The diagnostic plots, shown in figure 9.3, reveal a less satisfying fit in the main plot, and

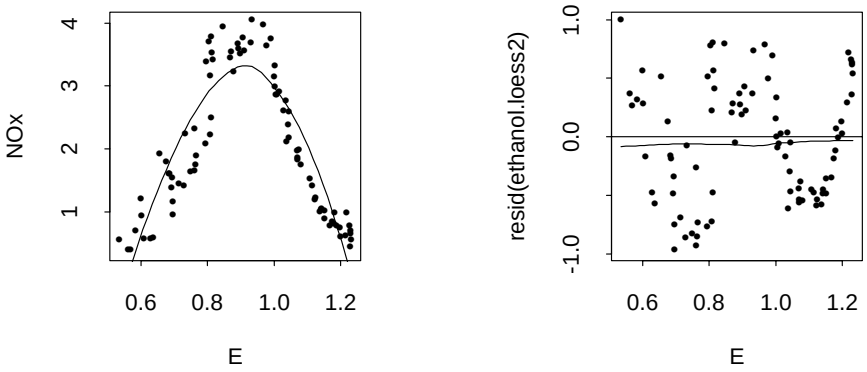

Figure 9.3: *Diagnostic plots for* `loess` *fit with span 1.*

much obvious structure left in the residuals. The residuals are also more broadly spread than those of the first model. We confirm this with a call to `anova` as follows:

```
> anova(ethanol.loess2, ethanol.loess)
Model 1:
loess(formula = NOx ~ E, data = ethanol, span = 1)
Model 2:
loess(formula = NOx ~ E, data = ethanol, span = 1/2)
Analysis of Variance Table
      ENP     RSS     Test     F Value      Pr(F)
1     3.5 22.0840   1 vs 2       32.79 8.2157e-15
2     6.2  9.1685
```

The difference between the models is highly significant, so we stick with our original model.

9.3 EXPLORING DATA WITH MULTIPLE PREDICTORS

Conditioning Plots

The `ethanol` data set actually has three variables, with the compression ratio, `C`, of the engine as another predictor joining the equivalence ratio `E` and the response, nitric oxide emissions, `NOx`. A summary of the data is shown below:

```
> summary(ethanol)
      NOx                C               E
Min.   :0.370   Min.   : 7.500   Min.   :0.5350
1st Qu.:0.953   1st Qu.: 8.625   1st Qu.:0.7618
Median :1.754   Median :12.000   Median :0.9320
Mean   :1.957   Mean   :12.030   Mean   :0.9265
3rd Qu.:3.003   3rd Qu.:15.000   3rd Qu.:1.1100
Max.   :4.028   Max.   :18.000   Max.   :1.2320
```

A good place to start an analysis with two or more predictors is a pairwise scatter plot, as generated by the `pairs` function:

```
> pairs(ethanol)
```

The resulting plot is shown in figure 9.4. The top row shows the nonlinear

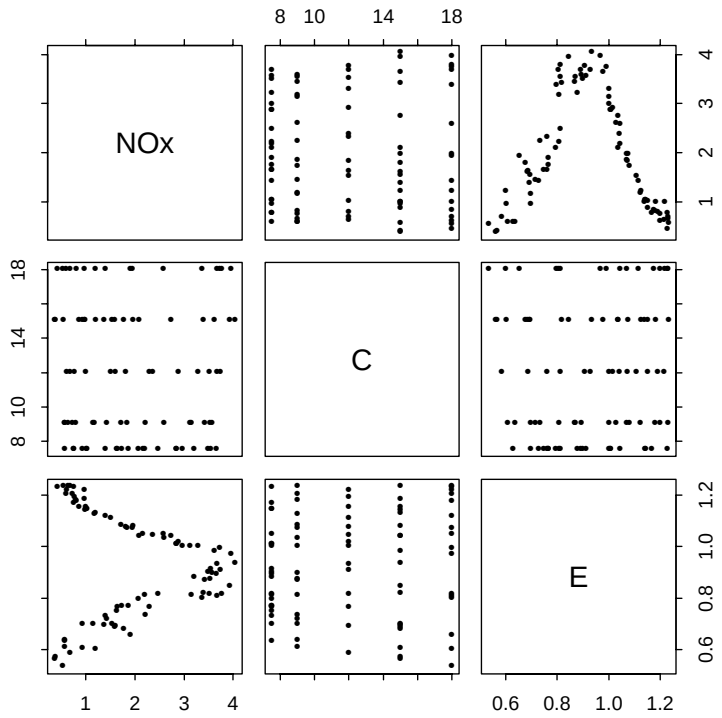

Figure 9.4: *Pairs plot of ethanol data.*

dependence of `NOx` on `E`, and no apparent dependence of `NOx` on `C`. The middle plot in the bottom row shows `E` plotted against `C`—this plot reveals no apparent correlation between the predictors, and shows that the compression ratio `C` takes on only 5 distinct values.

Another useful plot for data with two predictors is the perspective plot. This lets us view the response as a surface over the predictor plane.

```
> persp(interp(E, C, NOx))
```

The resulting plot is shown in figure 9.5.

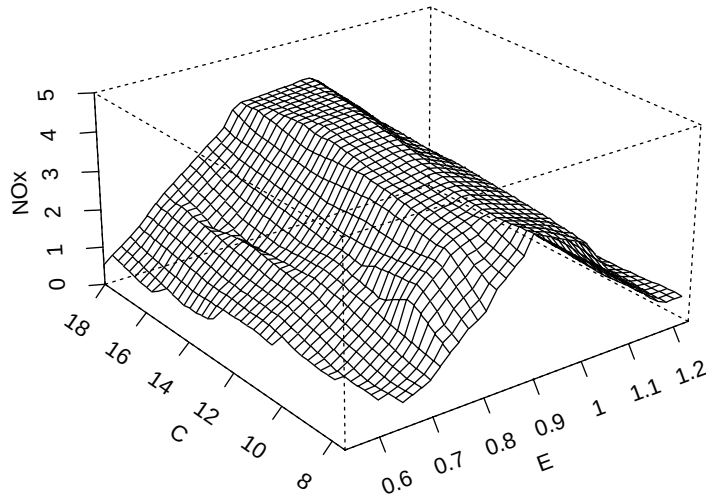

Figure 9.5: *Perspective plot of ethanol data.*

One conclusion we *cannot* draw from the pairwise scatter plot is that there is no effect of `C` on `NOx`. Such an effect might well exist, but be masked by the strong effect of `E`. Another type of plot, the *conditioning plot*, or *coplot*, can reveal such hidden effects.

A coplot shows how a response depends upon a predictor *given* other predictors. Basically, the idea is to create a matrix of *conditioning panels*; each panel graphs the response against the predictor for those observations whose value of the given predictor lie in an interval.

To create a coplot:

1. (*Optional.*) Create the conditioning values. The `coplot` function creates default values if conditioning values are omitted, but they are not usually as good as those created specifically for the data at hand.
2. Use the `coplot` function to create the plot.

We discuss these steps in detail in the following subsections.

Creating Conditioning Values

How you create conditioning values depends on the nature of the values taken on by the predictor, whether continuous or discrete.

For continuous data, the conditioning values are intervals, created using the function `co.intervals`. For example, the following call creates nine intervals for the predictor `E`:

```
> E.intervals <- co.intervals(E, number = 9, overlap = 1/4)
```

For data taking on discrete values, the conditioning values are the sorted, unique values. For example, the following call creates the conditioning values for the predictor `C`:

```
> C.points <- sort(unique(C))
```

Constructing a Conditioning Plot

To construct a conditioning plot, use `coplot` using a formula with the special form `A ~ B | C`, where `A` is the response, `B` is the predictor of interest, and `C` is the given predictor. Thus, to see the effect of `C` on `NOx` given `E`, use the formula `NOx ~ C | E`.

In most cases, you also want to specify one or both of the following arguments:

- `given.values`: the conditioning values created above.
- `panel`: a function of `x` and `y` used to determine the method of plotting in the dependence panels. The default is `points`.

To create the conditioning plot shown in figure 9.6,

```
> coplot(NOx ~ C | E, given.values = E.intervals)
```

Analyzing Conditioning Plots

To read the coplot, move from left to right, bottom to top. The scatter plots on the bottom row show an upward trend, while those on the upper two rows show a flat trend. We can more easily see the trend by using a smoothing function inside the conditioning panels, which we can do by specifying the `panel` argument to `coplot` as follows:

```
> coplot(NOx ~ C | E, given.values = E.intervals,
+ panel = function(x, y) panel.smooth(x, y,
+ degree = 1, span = 1))
```

The resulting plot is shown in figure 9.7. This plot clearly shows that for low values of `E`, `NOx` increases linearly with `C`, while for higher values of `E`, `NOx` remains constant with `C`.

Conversely, the coplot for the effects of `E` on `NOx` given `C` is created with the following call to `coplot`, and shown in figure 9.8:

```
> coplot(NOx ~ E | C, given.values = C.points,
+ panel = function(x, y) panel.smooth(x,y, degree =2,
+ span = 2/3))
```

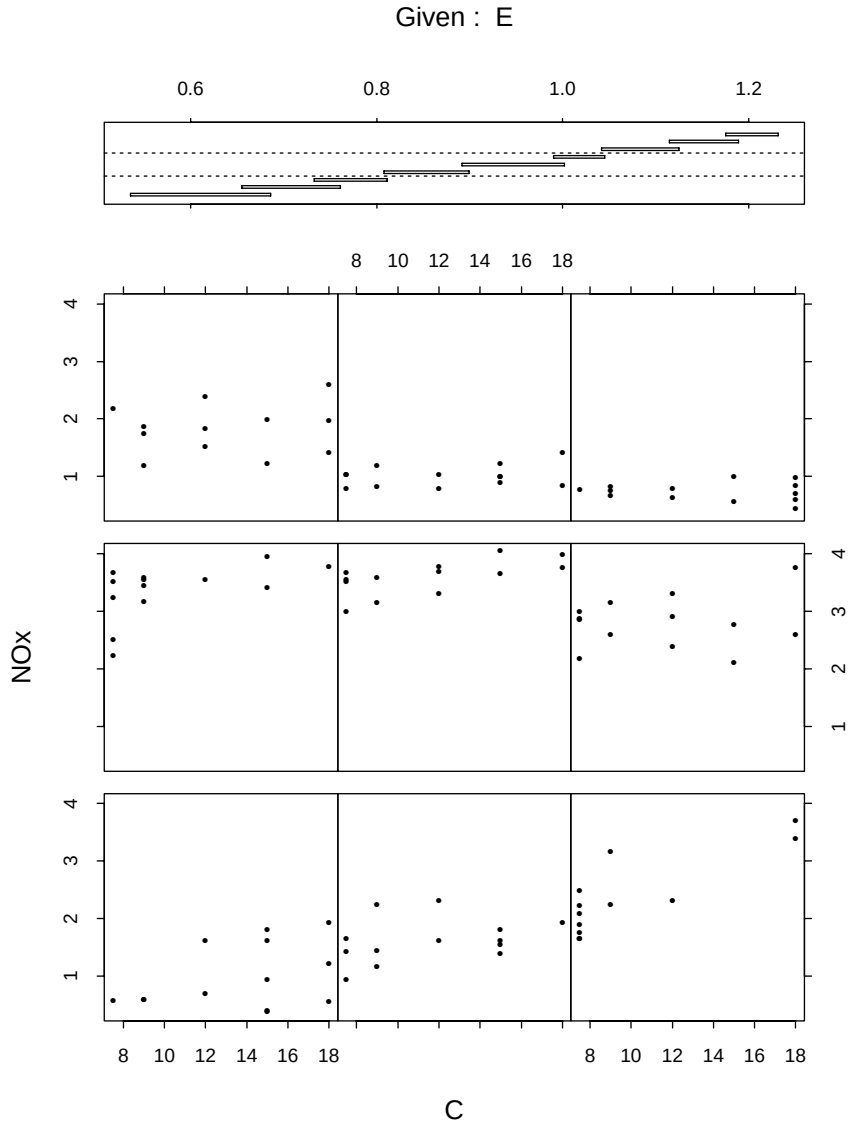

Figure 9.6: *Conditioning plot of ethanol data.*

Comparing the two coplots, we can see that `NOx` changes more rapidly as a function of `E` with `C` fixed than as a function of `C` with `E` fixed. Also, the variability of the residuals is small compared to the effect of `E`, but noticeable compared to the effect of `C`.

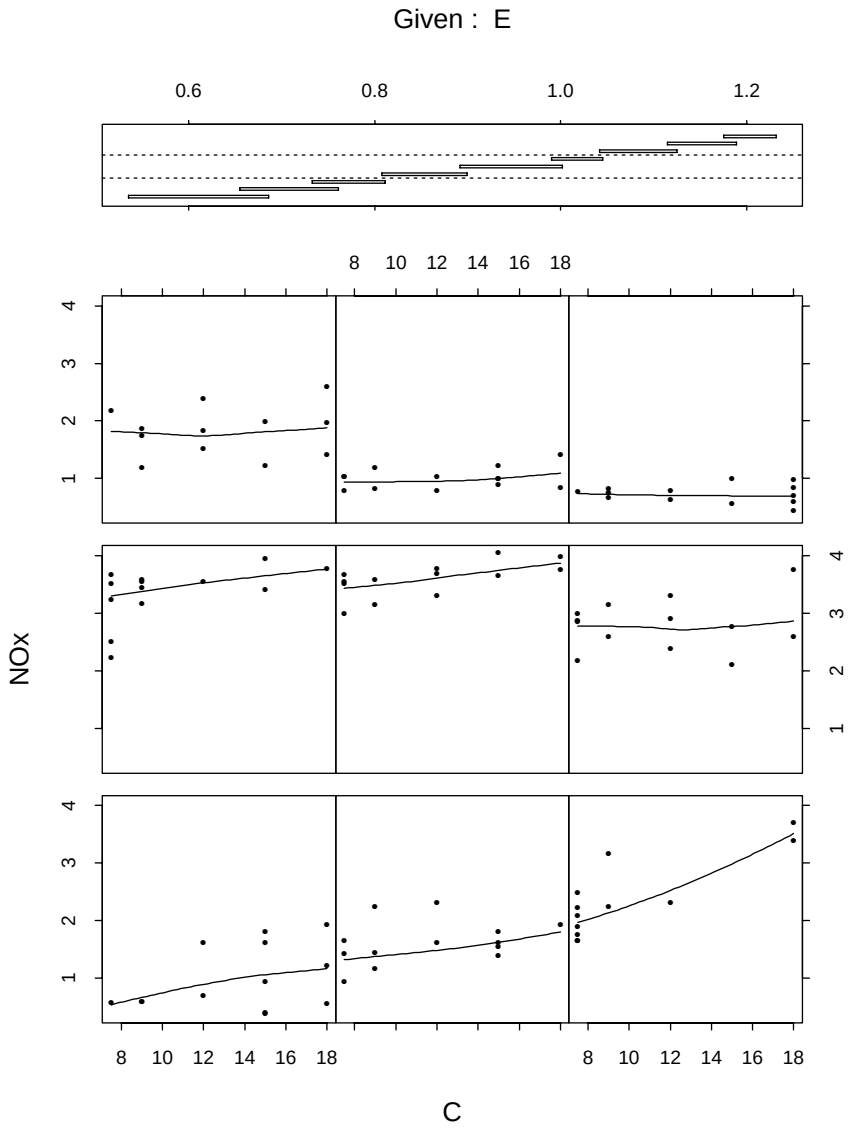

Figure 9.7: *Smooth conditioning plot of ethanol data.*

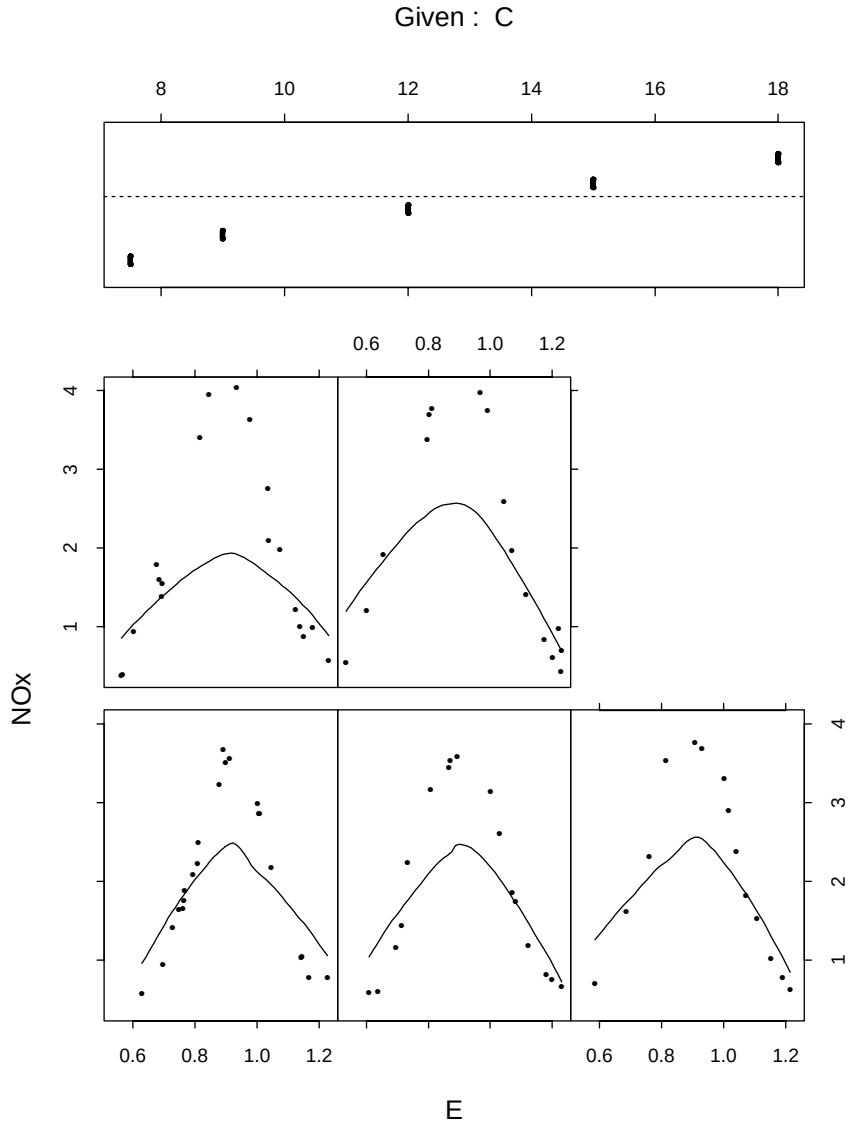

Figure 9.8: *Smooth conditioning plot of ethanol data, conditioned on* `C`.

9.4 FITTING A MULTIVARIATE LOESS MODEL

We have learned quite a bit about the `ethanol` data without fitting a model: there is a strong nonlinear dependence of `NOx` on `E` and there is an interaction between `C` and `E`. We can use this knowledge to shape our initial local regression model. First, we specify a formula that includes as predictors

both `E` and `C`, namely `NOx ~ C * E`. Then, we accept the default of local quadratic fitting to better model the nonlinear dependence. Our

```
> ethanol.m <- loess(NOx ~ C * E, data = ethanol)
> ethanol.m
Call:
loess(formula = NOx ~ C * E, data = ethanol)

Number of Observations:          88
Equivalent Number of Parameters: 9.4
Residual Standard Error:         0.3611
Multiple R-squared:              0.92
Residuals:
     min   1st Q   median 3rd Q    max
-0.7782 -0.3517 -0.05283 0.195 0.6338
```

We search for lack of fit by plotting the residuals against each of the predictors:

```
> par(mfrow=c(1,2))
> scatter.smooth(C, residuals(ethanol.m),span=1, deg=2)
> abline(h=0)
> scatter.smooth(E, residuals(ethanol.m),span=1, deg=2)
> abline(h=0)
```

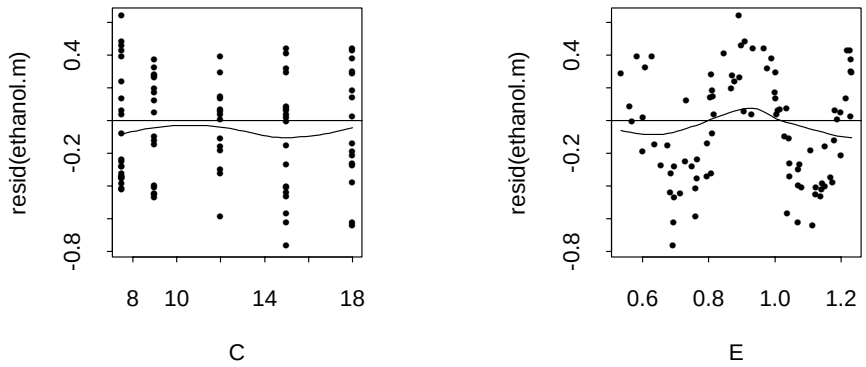

Figure 9.9: *Diagnostic plot for* `loess` *model of ethanol data.*

The resulting plot is shown in figure 9.9. The right-hand plot shows considerable lack of fit, so we reduce the span from the default `0.75` to `0.4`:

```
> ethanol.m2 <- update(ethanol.m, span = .4)
> ethanol.m2
```

```
Call: loess(formula = NOx ~ C * E, data = ethanol, span =
0.4)

Number of Observations:          88
Equivalent Number of Parameters: 15.3
Residual Standard Error:         0.2241
Multiple R-squared:              0.97
Residuals:
    min   1st Q   median  3rd Q    max
-0.4693 -0.1865 -0.03518 0.1027 0.3739
```

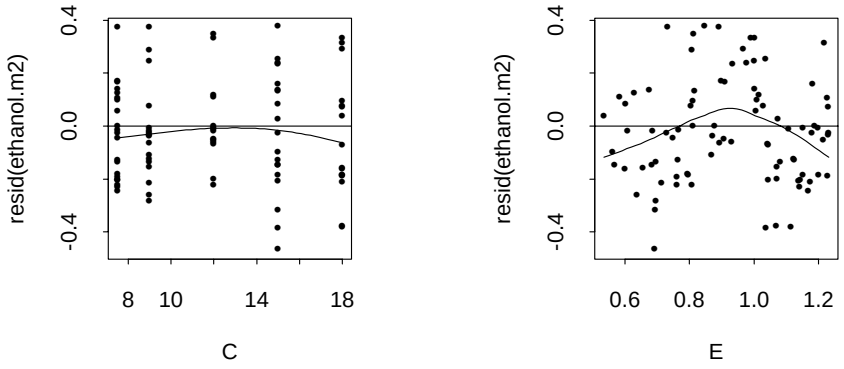

Figure 9.10: *Diagnostic plot for first revised model.*

Repeating the commands for generating the diagnostic plots with `ethanol.m2` replacing `ethanol.m` yields the plot shown in figure 9.10.

The right-hand plot looks better but still has some quadratic structure, so we shrink the span still further, and try again:

```
> ethanol.m3 <- update(ethanol.m, span = .25)
> ethanol.m3
Call:
loess(formula = NOx ~ C * E, data = ethanol, span = 0.25)

Number of Observations:          88
Equivalent Number of Parameters: 21.6
Residual Standard Error:         0.1761
Multiple R-squared:              0.98
Residuals:
    min    1st Q  median   3rd Q    max
-0.3975 -0.09077 0.00862 0.06205 0.3382
```

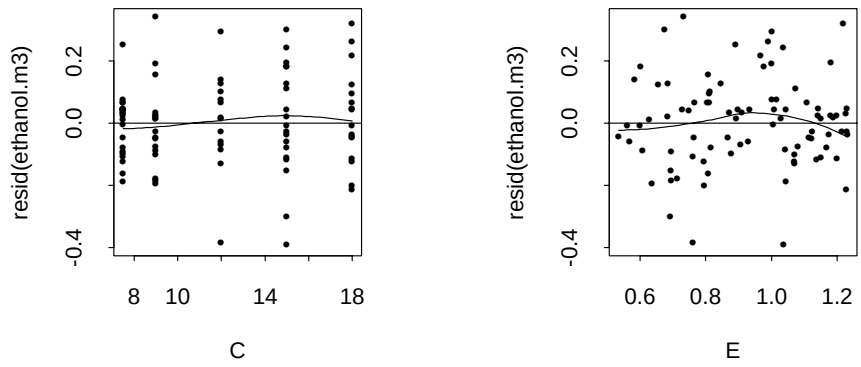

Figure 9.11: *Diagnostic plot for second revised model.*

Again, we create the appropriate residuals plots to check for lack of fit. The result is shown in figure 9.11. This time the fit is much better.

Another check on the fit is provided by coplots using the residuals as the response variable:

```
> coplot(residuals(ethanol.m3) ~ C | E, given = E.intervals,
+ panel= function(x,y)
+ panel.smooth(x,y, degree=1, span=1, zero.line=TRUE))
> coplot(residuals(ethanol.m3) ~ E | C, given = C.points,
+ panel= function(x,y)
+ panel.smooth(x,y, degree=1, span=1, zero.line=TRUE))
```

The resulting plots are shown in figure 9.12 and figure 9.13. The middle row of figure 9.12 shows some anomalies—the residuals are virtually all positive. However, the effect is small, and limited in scope, so it can probably be ignored.

As a final test, we make several more diagnostic plots to check the distribution of the error terms (figure 9.14):

```
> par(mfrow=c(2,2))
> plot(fitted(ethanol.m3), sqrt(abs(resid(ethanol.m3))))
> plot(C, sqrt(abs(resid(ethanol.m3))))
> plot(E, sqrt(abs(resid(ethanol.m3))))
> qqnorm(resid(ethanol.m3))
> qqline(resid(ethanol.m3))
NULL
```

The model passes these checks—the errors appear to be Gaussian, or nearly so.

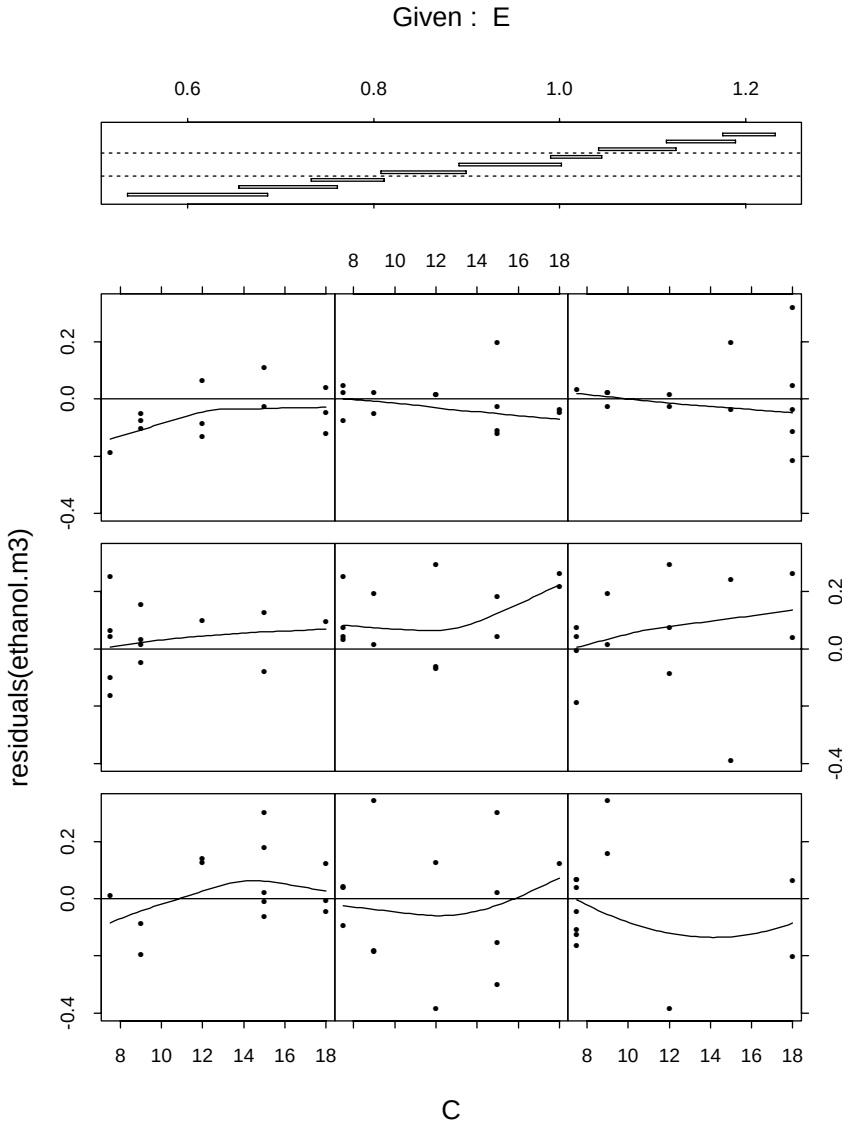

Figure 9.12: *Conditioning plot on* E *for second revised model.*

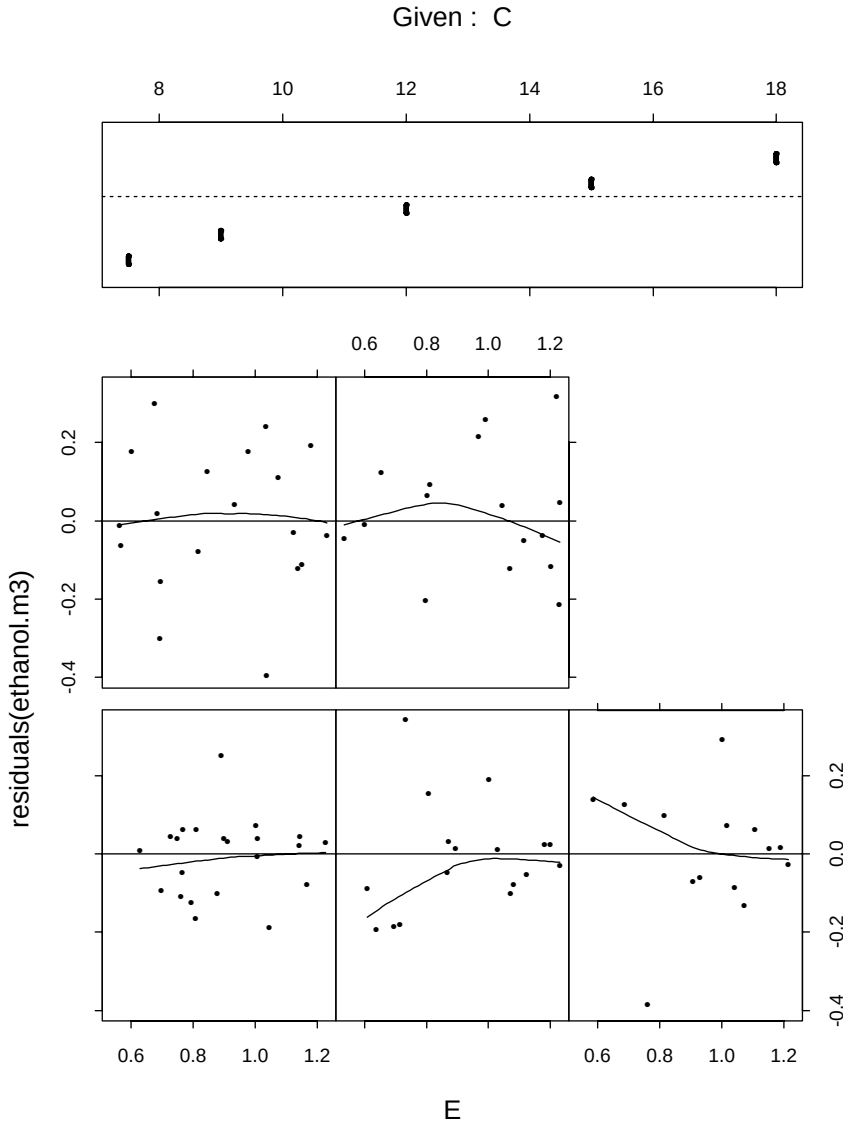

Figure 9.13: *Conditioning plot on* `C` *for second revised model.*

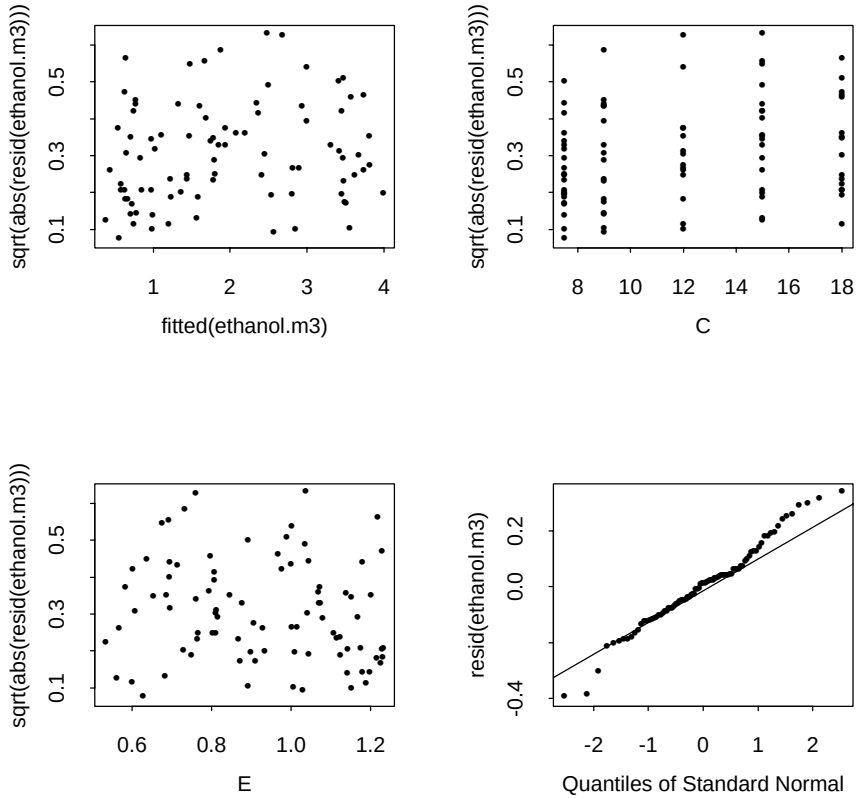

Figure 9.14: *Diagnostic plots for second revised model.*

9.5 LOOKING AT THE FITTED MODEL

Examining the fitted model graphically is no less important than graphically examining the data. One way to test the model is to compare the predicted surface with the data surface shown in figure 9.5 . We can create the corresponding perspective plot for the model as follows. First, define an evenly-spaced grid of points spanning the range of `E` and `C`:

```
> newC <- seq(from = min(C), to = max(C), length = 40)
> newE <- seq(from = min(E), to = max(E), length = 40)
> new.ethanol <- expand.grid(E = newE, C = newC)
```

The `expand.grid` function returns a data frame with 1600 rows and 2 columns, corresponding to all possible combinations of `newC` and `newE`. We can then use `predict` with the fitted model and these new data points to calculate predicted values for each of these grid points:

```
> eth.surf <- predict(ethanol.m3, new.ethanol)
```

The perspective plot of the surface is then created readily as follows:

```
> persp(newE, newC, eth.surf, xlab = "E",
+ ylab = "C")
```

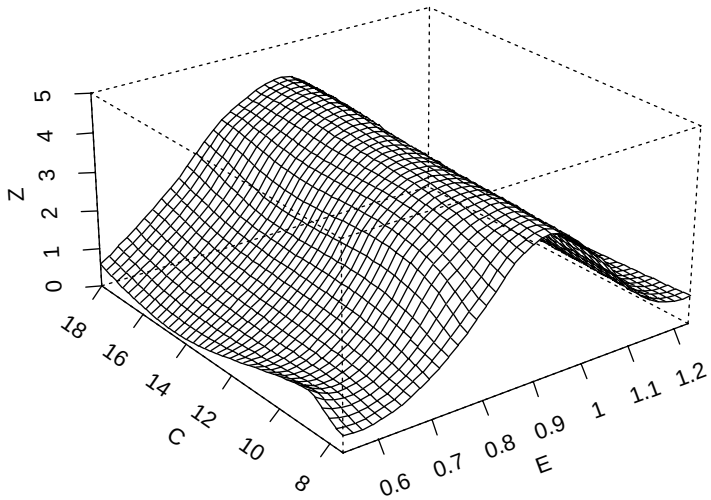

Figure 9.15: *Perspective plot of the model.*

The resulting plot is shown in figure 9.15. Not surprisingly, the surfaces look quite similar, with the model surface somewhat smoother than the data surface. The data surface has a noticeable wrinkle for `E` $\approx$ 0.7, `C` $\approx$ 14. This wrinkle is smoothed out in the model surface. Another graphical view is probably worthwhile.

The default graphical view for `"loess"` objects with multiple predictors is a set of coplots, one per predictor, created using the `plot` function.

```
> par(ask=T)
> plot(ethanol.m3, confidence = 7)
```

The resulting plots are shown in figure 9.16 and figure 9.17. One feature that is immediately apparent, and somewhat puzzling, is the curvy form of the bottom row of figure 9.16. Our preliminary coplots revealed that the dependence of `NOx` on `C` was approximately linear for small values of `E`. Thus, the model as fitted has a noticeable departure from our understanding of the data.

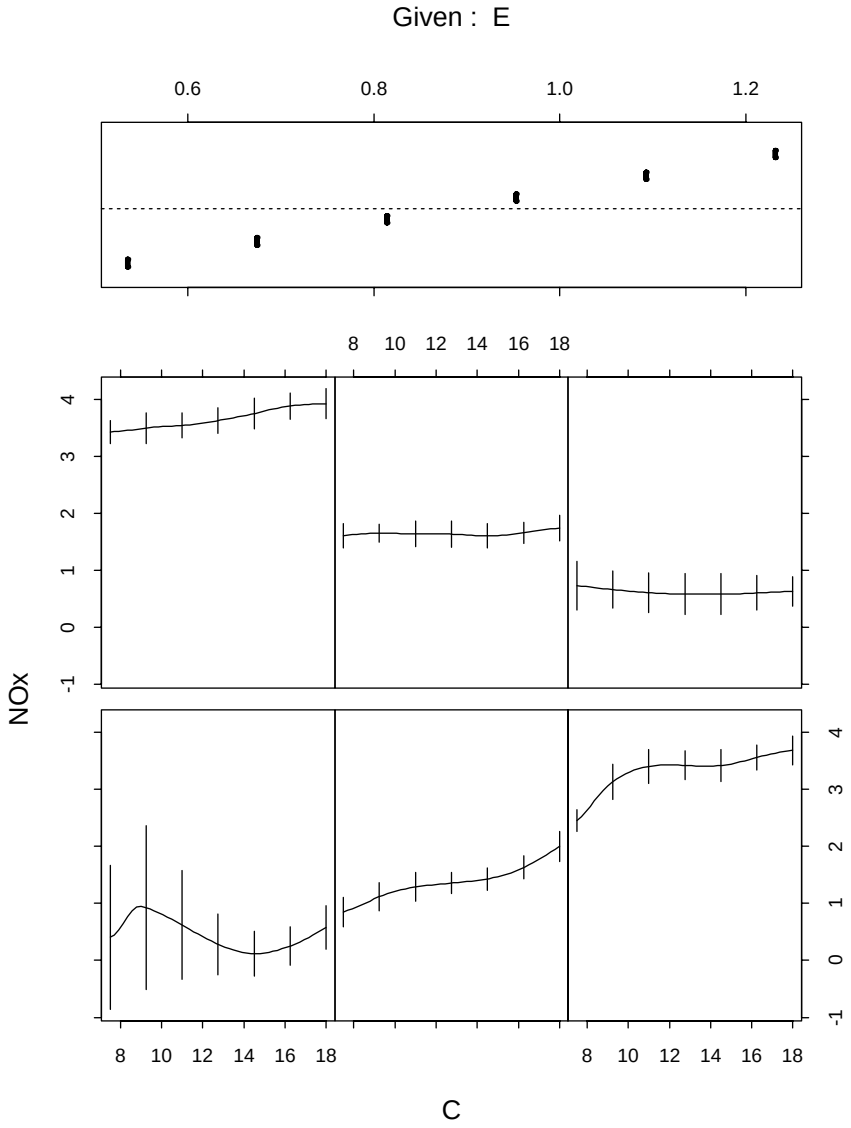

Figure 9.16: *Default conditioning plot of the model, first predictor.*

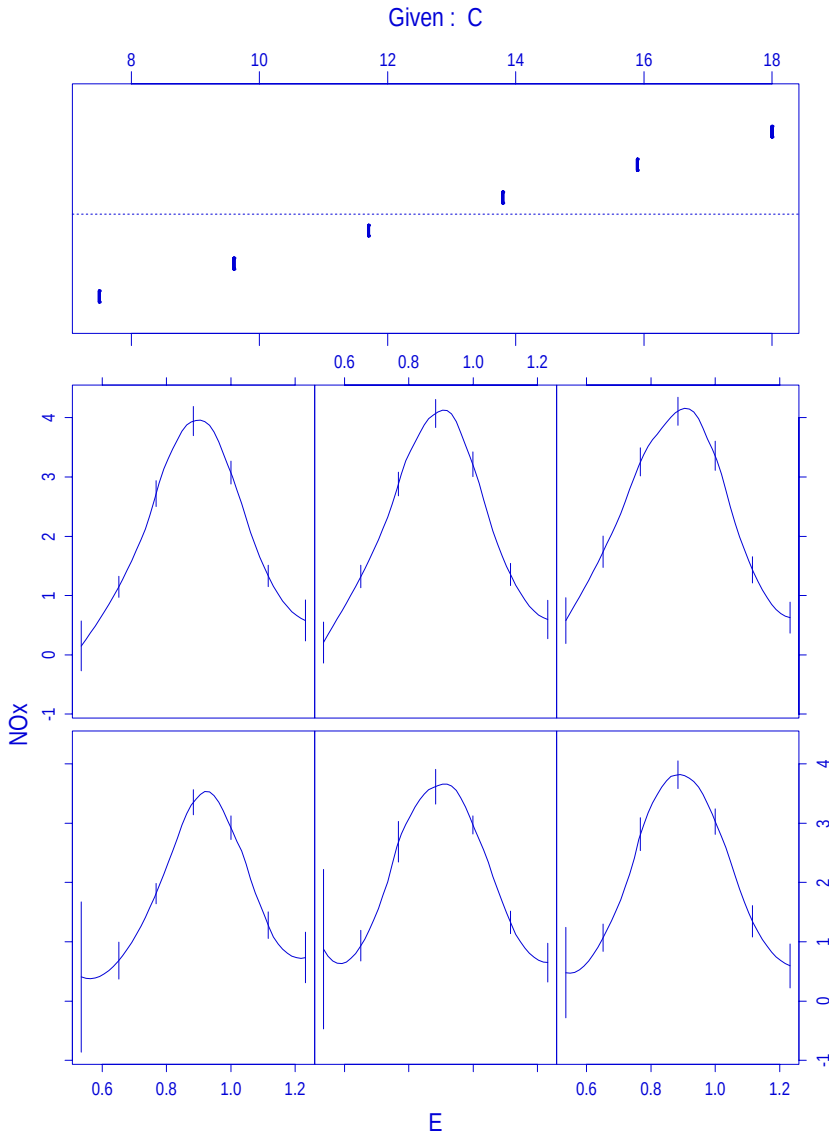

Figure 9.17: *Default conditioning plot of the model, second predictor.*

9.6 IMPROVING THE MODEL

The model in `ethanol.m3` is fit using local quadratic fitting for all terms corresponding to `C*E`. This means that the model contains the following fitting variables: a constant, `E`, `C`, `EC`, C^2, and E^2. However, our original look at the data led us to believe that the effect of `C` was piecewise linear; it thus

makes sense to fit `C` *parametrically*, and drop the quadratic term. We can make these changes using the `update` function as follows:

```
> ethanol.m4 <- update(ethanol.m3, drop.square="C",
+ parametric = "C")
> ethanol.m4
Call:
loess(formula = NOx ~ C * E, span = 0.25, parametric = "C",
drop.square = "C")

Number of Observations:          88
Equivalent Number of Parameters: 18.2
Residual Standard Error:         0.1808
Multiple R-squared:              0.98
Residuals:
    min     1st Q    median   3rd Q    max
-0.4388 -0.07358 -0.009093 0.06616 0.5485
```

The offending coplot, figure 9.18 and figure 9.19, now shows the appropriate linear fit, and we have introduced no lack of fit, as shown by the residuals plots in figure 9.20. In fact, comparing the plot of residuals against `E` for the latest model with that for `ethanol.m3` (figure 9.21) indicates we may be able to increase the span for the latest model and not introduce any lack of fit:

```
> ethanol.m5 <- update(ethanol.m4, span = 1/2)
> ethanol.m5
Call:
loess(formula = NOx ~ C * E, span = 1/2, parametric = "C",
drop.square = "C")

Number of Observations:          88
Equivalent Number of Parameters: 9.2
Residual Standard Error:         0.1842
Multiple R-squared:              0.98
Residuals:
    min   1st Q  median   3rd Q    max
-0.5236 -0.0972 0.01386 0.07326 0.5584
```

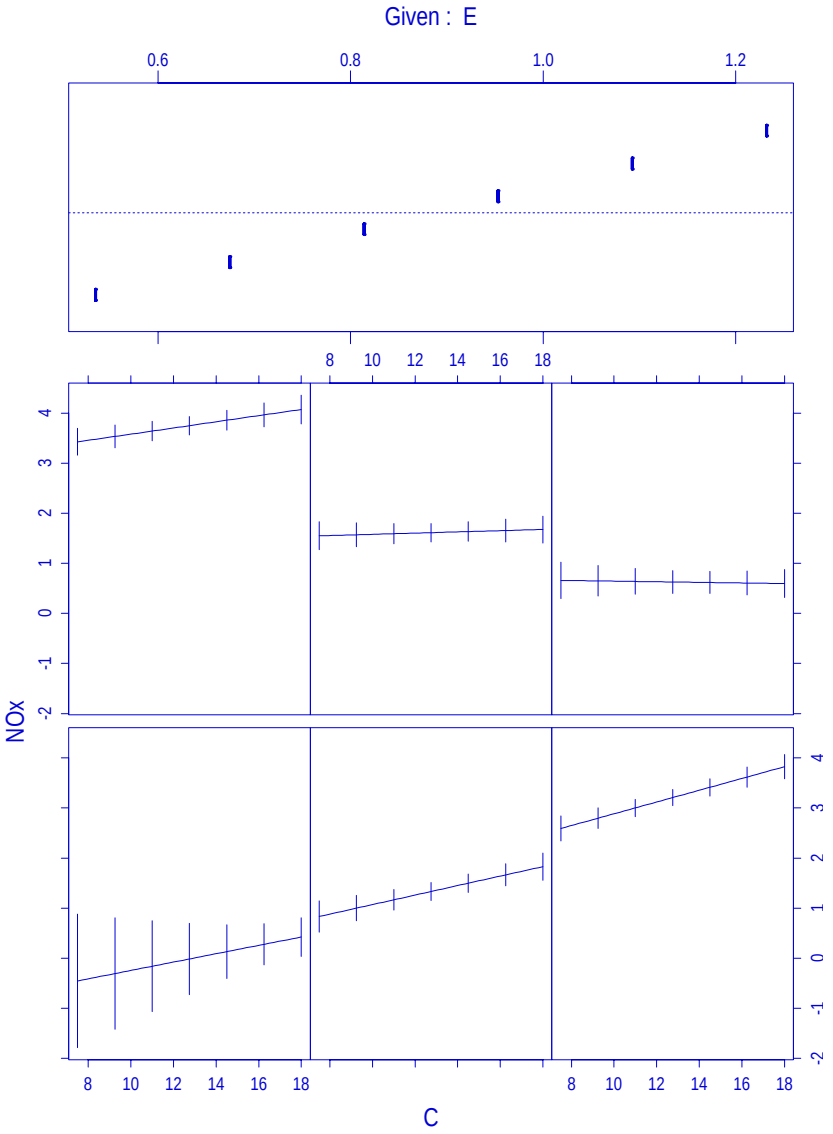

Figure 9.18: *Default conditioning plot of improved model, first predictor.*

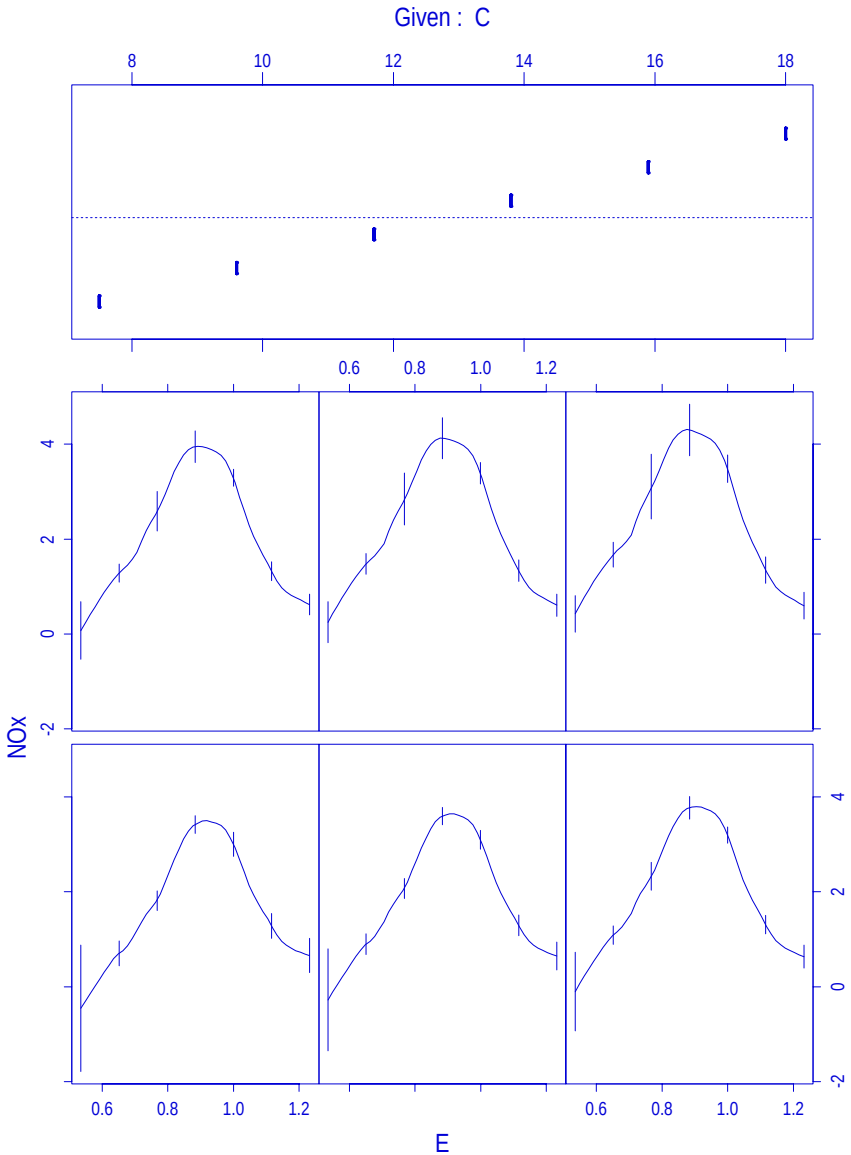

Figure 9.19: *Default conditioning plot of improved model, second predictor.*

We gain a *much* more parsimonious model—the Equivalent Number of Parameters drop from approximately 18 to about 9. An *F*-test using `anova` shows no significant difference between our first acceptable model and the latest, more parsimonious model:

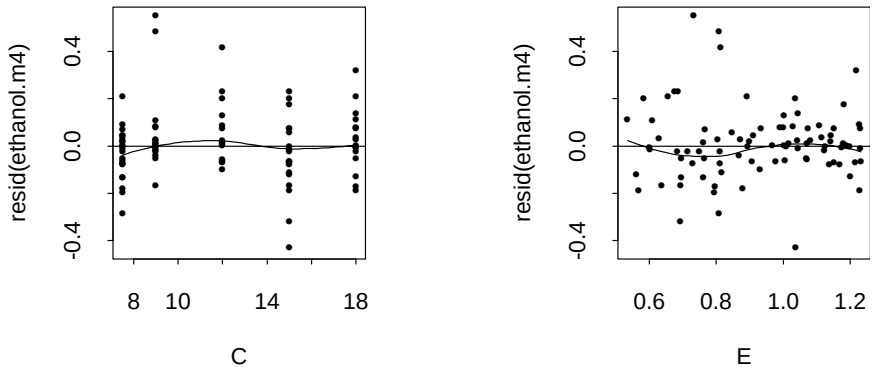

Figure 9.20: *Residual plot of improved model.*

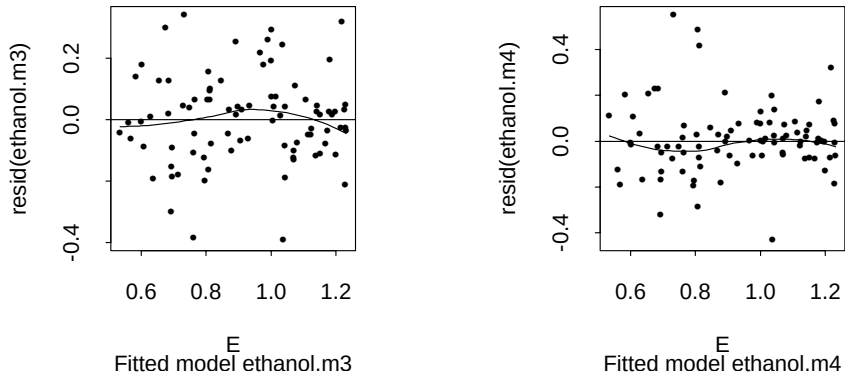

Figure 9.21: *Comparison of residual plots for original and improved models.*

```
> anova(ethanol.m3,ethanol.m5)
Model 1:
loess(formula = NOx ~ C * E, span = 0.25)
Model 2:
loess(formula = NOx ~ C * E, span = 1/2, parametric = "C",
drop.square = "C")
Analysis of Variance Table
      ENP     RSS     Test      F Value     Pr(F)
1    21.6  1.7999   1 vs 2         1.42   0.16486
2     9.2  2.5433
```

CLASSIFICATION AND REGRESSION TREES

10

Tree-based models uncover structure in data.

CLASSIFICATION AND REGRESSION TREES

Tree-based modeling is an exploratory technique for uncovering structure in data, increasingly used for:

- devising prediction rules that can be rapidly and repeatedly evaluated
- screening variables
- assessing the adequacy of linear models
- summarizing large multivariate datasets

Tree-based models are useful for both classification and regression problems. In these problems, there is a set of classification or predictor variables (x), and a single-response variable (y).

If y is a factor, *classification* rules are of the form:

if $(x_1 \leq 2.3)$ *and* $(x_3 \in \{A, B\})$

then y is most likely to be in level 5.

If y is numeric, *regression* rules for description or prediction are of the form:

if $(x_2 \leq 2.3)$ *and* $(x_9 \in \{C, D, F\})$ *and* $(x_5 \leq 3.5)$

then the predicted value of y is 4.75.

A classification or regression tree is the collection of many such rules displayed in the form of a binary tree, hence the name. The rules are determined by a procedure known as *recursive partitioning*. Tree-based models provide an alternative to linear and additive models for regression problems, and to linear and additive logistic models for classification problems.

Compared to linear and additive models, tree-based models have the following advantages:

- Easier to interpret when the predictors are a mix of numeric variables and factors.
- Invariant to monotone re-expressions of predictor variables.
- More satisfactorily treat missing values.
- More adept at capturing nonadditive behavior.
- Allow more general (that is, other than of a particular multiplicative form) interactions between predictor variables.

- Can model factor response variables with more than two levels.

This chapter is organized around the following topics:

- Growing trees
- Displaying tree fits
- Prediction and Residuals
- Missing Values
- Pruning and Shrinking
- Interacting with trees

10.1 GROWING TREES

We describe the tree-growing function `tree` by presenting several examples. The `tree` function generates objects of class `"tree"`. This function automatically decides whether to fit a regression or classification tree, according to whether the response variable is numeric or a factor. We also show two types of displays, generated by generic functions: a tree display produced by `plot` and a table produced by `print`.

In general, the response *y* and predictors *x* may be any combination of numeric or factor types. In fact, the predictors can be a mix of numeric *and* factor. However, no factor predictor can have no more than 32 levels, and no factor response can have more than 128 levels. In both of the examples below, the predictors are all numeric. The numeric response example illustrates a regression tree. The factor response example illustrates a classification tree.

Numeric Response and Predictor

In the first example, we grow a *regression* tree relating the numeric response `Mileage` to the predictor variable `Weight` from the data frame `car.test.frame`. The resulting tree is given the name `auto.tree`, which is then plotted by the generic `plot` function and labeled by the generic `text` function (see figure 10.1).

```
> attach(car.test.frame)
> auto.tree <- tree(Mileage ~ Weight, car.test.frame)
> plot(auto.tree,type="u")
> text(auto.tree)
> title("A Tree-Based Model\nfor Mileage versus Weight")
```

In describing tree-based models, the terminology mimics real trees:

- root The top node of the tree
- leaf A terminal node of the tree

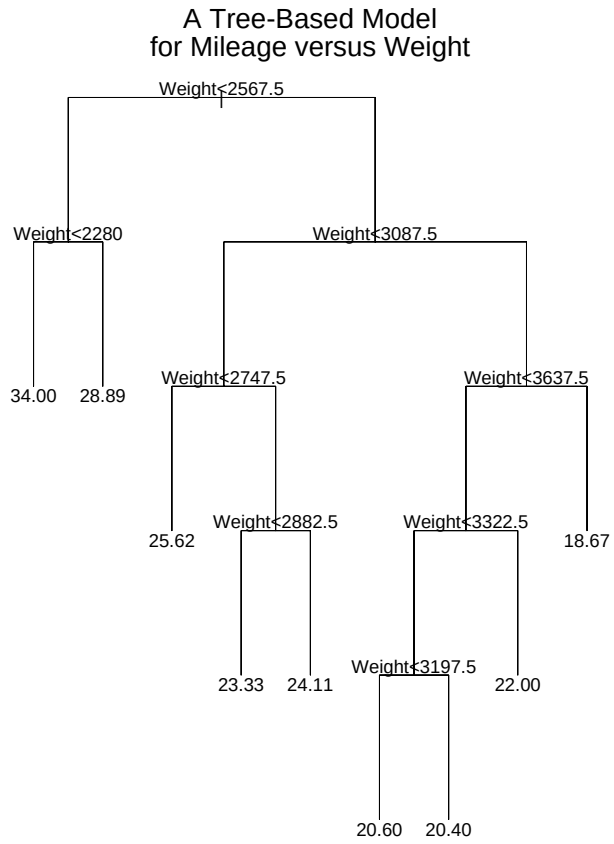

Figure 10.1: *Display of a tree-based model with a numeric response,* `Mileage` *and one numeric predictor,* `Weight`.

- split A rule for creating new branches

In growing a tree, the binary partitioning algorithm recursively splits the data in each node until either the node is homogeneous or the node contains too few observations (≤ 5 , by default).

In order to *predict* mileage from weight, one follows the path from the root, to a leaf, according to the splits at the interior nodes. The tree in figure 10.1 is interpreted in the following way:

- Automobiles are first split according to whether they weigh less than 2567.5 pounds.
- If so, they are again split according to weight being less than 2280 pounds.

- Lighter cars (< 2280 pounds) have a predicted mileage of 34 mpg.
- Heavier cars ($2280 \leq$ `Weight` < 2567.5) have a mileage of 28.9 mpg.
- For those automobiles weighing more than 2567.5 pounds, seven weight classes are formed.
- The predicted mileage ranges from a high of 25.6 mpg to a low of 18.7 mpg.
- Overall, heavier cars get poorer mileage than lighter cars.
- It appears that doubling the weight of an automobile approximately halves its mileage.

Factor Response and Numeric Predictor

In this classification example, we model the probability of developing Kyphosis, using the `kyphosis` data frame with predictors `Age`, `Start`, and `Number`.

First, use boxplots to plot the distributions of the predictor variables as a function of `Kyphosis` in figure 10.2. `Start` appears to be the single best predictor of `Kyphosis` since Kyphosis is more likely to be present among individuals with `Start` ≤ 12 .

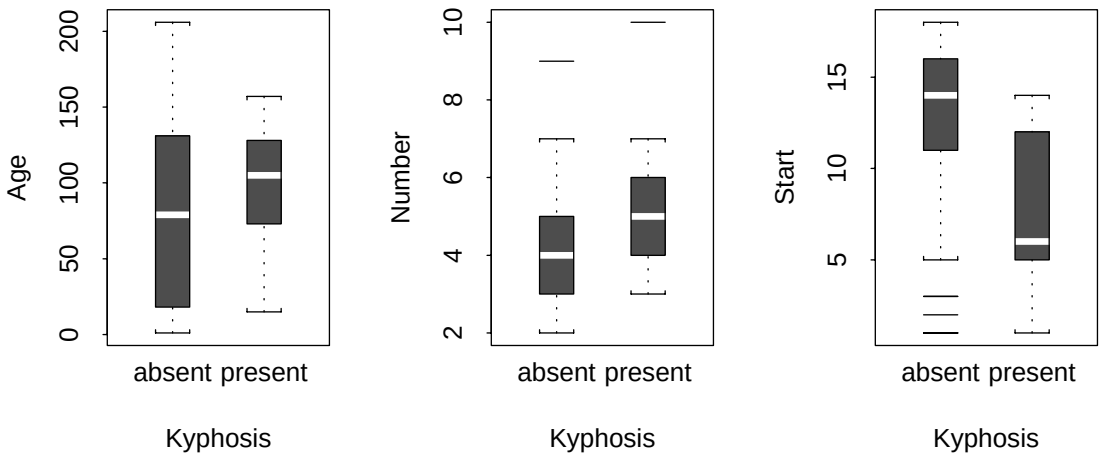

Figure 10.2: *Boxplots of the predictors of* `Kyphosis`.

```
> kyph.tree <- tree(Kyphosis ~ Age + Number + Start,
+ data = kyphosis)
```

Since `Kyphosis` is a factor response, the result `kyph.tree` is a *classification*

tree.

Either the `formula` or `data` arguments to the `tree` function may be missing. Without the `formula` argument, a tree is constructed from the data frame using the first variable as the response. Hence the Kyphosis example could have been constructed as follows:

```
> auto.tree <- tree(car.test.frame)
```

Without the `data` argument, the variables named in `formula` are expected to be in the search list. The Kyphosis tree could also have been grown with

```
> attach(car.test.frame)
> auto.tree <- tree(Mileage ~ Weight)
```

The only meaningful *operator* on the right side of a formula is `"+"`. Since tree-based models are invariant to monotone re-expressions of individual predictor variables, functions like `log`, `I`, and `^` have little use. Also, tree-based models capture interactions without explicit specification.

This time, we display the fitted tree using the generic function, `print`, which is called automatically simply by typing the name of the tree object. This tabular representation is most useful when the details of the fitting procedure are of interest. Indentation is added as a key to the underlying structure.

```
> kyph.tree
node), split, n, deviance, yval, (yprob)
      * denotes terminal node
1) root 81 83.230 absent ( 0.7901 0.20990 )
  2) Start<12.5 35 47.800 absent ( 0.5714 0.42860 )
    4) Age<34.5 10 6.502 absent ( 0.9000 0.10000 )
    8) Age<16 5 5.004 absent ( 0.8000 0.20000 ) *
    9) Age>16 5 0.000 absent ( 1.0000 0.00000 ) *
  5) Age>34.5 25 34.300 present ( 0.4400 0.56000 )
   10) Number<4.5 12 16.300 absent ( 0.5833 0.41670 )
     20) Age<127.5 7 8.376 absent ( 0.7143 0.28570 ) *
     21) Age>127.5 5 6.730 present ( 0.4000 0.60000 ) *
   11) Number>4.5 13 16.050 present ( 0.3077 0.69230 )
     22) Start<8.5 8 6.028 present ( 0.1250 0.87500 ) *
     23) Start>8.5 5 6.730 absent ( 0.6000 0.40000 ) *
3) Start>12.5 46 16.450 absent ( 0.9565 0.04348 )
  6) Start<14.5 17 12.320 absent ( 0.8824 0.11760 )
   12) Age<59 5 0.000 absent ( 1.0000 0.00000 ) *
   13) Age>59 12 10.810 absent ( 0.8333 0.16670 )
     26) Age<157.5 7 8.376 absent ( 0.7143 0.28570 ) *
     27) Age>157.5 5 0.000 absent ( 1.0000 0.00000 ) *
```

```
7) Start>14.5 29 0.000 absent ( 1.0000 0.00000 ) *
```

The first number in each row of the output is a node number. The nodes are numbered to index the tree for quick identification. For a *full* binary tree, the nodes at depth d are integers n, $2^d \le n < 2^{d+1}$. Usually, a tree is *not* full, but the numbers of the nodes that *are* present are the same as they would be in a full tree.

In the `print` output, the nodes are ordered according to a depth-first traversal of the tree, printed output.

Let us first examine one row of the output:

```
2) Start<12.5 35 47.800 absent ( 0.5714 0.42860 )
```

This row is for node 2. Following the node number is the split, `Start<12.5`. This states the the observations in the parent (root) node with `Start<12.5` were put into node 2.

The next number after the split is the number of observations, 35. The number 47.8 is the *deviance*, the measure of node heterogeneity used in the tree-growing algorithm. A perfectly homogeneous node has deviance zero. The fitted value, `yval`, of the node is `absent`. Finally, the numbers in parentheses (0.5714 0.42860), `yprob`, are the estimated probabilities of the observations in that node not having, and having, kyphosis. Therefore the observations with `Start<12.5` have a 0.5714 chance of not having kyphosis under this tree model.

An interpretation of the table follows:

- The split on `Start` partitions the 81 observations into groups of 35 and 46 individuals (nodes 2 and 3) with probability of `Kyphosis` 0.429 and 0.043, respectively.
- The group at node 2 is then partitioned into groups of 10 and 25 individuals (nodes 4 and 5) depending on whether `Age` is less than 34.5 years or not.
- The group at node 4 is divided in half depending on whether `Age` is less than 16 or not. If `Age` >16 none of the individuals have Kyphosis (probability of Kyphosis is 0). These subgoups are divided no further.
- The group at node 5 is subdivided into groups of size 12 and 13 depending on whether or not `Number` is less than 4.5. The respective probabilities of `Kyphosis` for these groups is 0.417 and 0.692.
- The procedure continues, yielding 10 distinct groups with probabilities of `Kyphosis` ranging from 0.0 to 0.875.

- Asterisks signify *terminal* nodes; that is, those that are not split.

10.2 DISPLAYING TREES

The generic functions `print`, `plot`, and `summary` work as expected for `"tree"` objects. We have already encountered the first two functions in the examples above. A further interesting feature of `plot` is that an optional `type` argument controls node placement. The `type` argument can have either of the two values:

- "" Produces nonuniform spacing as the default. The more important the parent split, the further the children node pairs are spaced from their parents.
- "u" Produces uniform spacing.

In the car mileage example, we used uniform spacing in order to label the tree. However, if the goal is tree simplification, we gain insight into the relative importance of the splits by using the default `type`, i.e., nonuniform spacing. This is shown in figure 10.3.

When you first plot the tree using `plot`, the nodes and splits will be displayed without any text labels. The generic `text` function, described in the S-PLUS User's Guide, uses the same arguments to rotate and adjust text in tree plots that it uses with most other types of plots.

The `summary` function has a tree-specific method which indicates the tree type (regression/classification), a record of how the tree was created, the residual mean deviance, and other information.

The residual deviance is the sum, over all the observations, of terms which vary according to type (regression/classification) of tree. The residual mean deviance is then obtained after dividing by the degrees of freedom (number of observations minus the number of terminal nodes).

The following summary is typical for *regression*:

```
> summary(auto.tree)

Regression tree:
tree(formula = Mileage ~ Weight, data = car.test.frame)
Number of terminal nodes: 9
Residual mean deviance: 4.289 = 218.7 / 51
Distribution of residuals:
   Min. 1st Qu. Median Mean 3rd Qu. Max.
 -3.889 -1.111 0 0 1.083 4.375
```

The regression tree has nine terminal nodes. Under a normal (Gaussian) assumption, the terms in the residual mean deviance are the squared

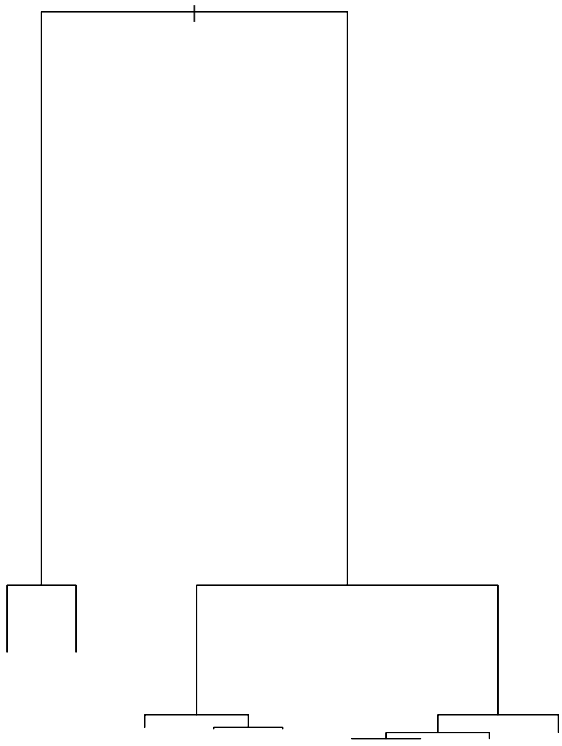

Figure 10.3: *Plot of the car mileage tree with non-uniform node placement.*

differences between the observations and the predicted values. See the section Prediction and Residuals for a discussion of prediction and residuals. The `summary` function also summarizes the distribution of residuals.

The following summary is typical for *classification* trees:

```
> summary(kyph.tree)
Classification tree:
tree(formula = Kyphosis ~ Age + Number + Start)
Number of terminal nodes: 10
Residual mean deviance: 0.5809 = 41.24 / 71
Misclassification error rate: 0.1235 = 10 / 81
```

Note that, for classification trees, the `summary` function gives the misclassification error rate instead of distribution of residuals. First, predicted classifications are obtained as described in the section Prediction and Residuals. The error rate is then obtained by counting the number of misclassified observations, and dividing by the number of observations. The

terms in the residual mean deviance are based on the multinomial distribution (see Chambers and Hastie (1992)).

10.3 PREDICTION AND RESIDUALS

Once a tree is grown, an important use of the fitted tree is to predict the value of the response variable for a set of predictor variables.

For concreteness, consider just one observation *x* on the predictor variables. In prediction, the splits direct *x* through the tree. The *prediction* is taken to be the the `yval` at the deepest node reached. Usually this corresponds to a leaf node. However, in certain situations, a prediction may reside in a nonterminal node (Chambers and Hastie (1992)). In particular this may happen if missing values occur in *x*, and the tree was grown with only complete observations.

The generic function `predict` has a tree-specific method. It takes a tree object and, optionally, a data frame as arguments. If the data frame is not supplied, `predict` returns the fitted values for the data originally used to construct the tree. The function returns predicted values either as a vector (the default) or a tree object (`type="tree"`).

The residuals can then be obtained either by subtracting the fitted values from the response variable, or directly using the function `residuals`. Figure 10.4 presents a plot of the residuals versus the predicted values and a normal probability of the residuals for the `auto.tree` model.

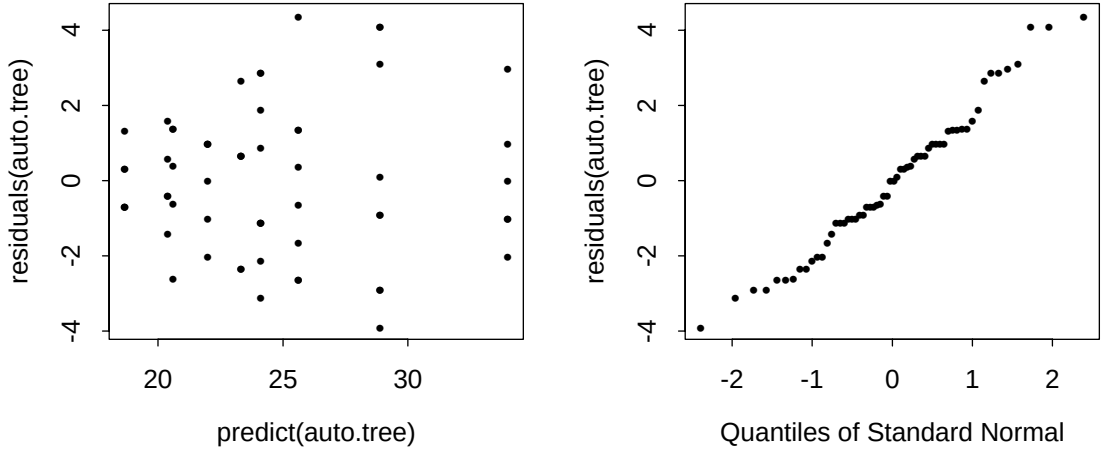

Figure 10.4: *Residuals versus predicted values and a normal probability plot of the residuals for a* `"tree"` *object.*

10.4 MISSING DATA

Missing values, NAs, can occur either in data used to build trees, or in a set of predictors for which the value of the response variable is to be predicted.

For data used to build trees, the `tree` function permits NAs only in predictor variables, but only if the argument `na.action = na.tree.replace`. For any predictor with missing values, the `na.tree.replace` function creates a new factor variable, with an added level named `"NA"` (numeric predictors are first quantized).

In prediction, suppose an observation is missing a value for the variable *V*. Further, suppose there were no missing values for *V* in the training data. The observation follows its path down the tree until it encounters a node whose split is based upon *V*. The prediction is then taken to be the `yval` at that node. If values of several variables are missing, the observation stops at the first such variable split encountered.

To clarify this, let us return to the automobile example, where some of the data are missing values on the variable `Reliability`. We first fit a tree on the data with *no* missing values. The resulting tree is displayed in figure 10.5. Notice the split on the variable `Reliability`.

To create the tree shown in figure 10.5, first create a new data set from `car.test.frame`, omitting those observations which are missing data for `Reliability`:

```
> car.test.no.miss <-
+ car.test.frame[!is.na(car.test.frame[,3]),]
```

Now grow the tree using the cleansed data:

```
> car.tree <- tree(Mileage ~ Weight + Reliability,
+ car.test.no.miss)
```

Next, we predict the data with values missing on `Reliability`, by extracting those observations that were omitted from `car.test.no.miss`, and then calling `predict` on the resulting data set:

```
> car.test.miss <-
+ car.test.frame[is.na(car.test.frame[,3]),]
> pred.miss <-
+ predict(car.tree,car.test.miss,type="tree")
> pred.miss
node), split, n, deviance, yval
      * denotes terminal node
1) root 11 245.300 24.80
  2) Weight<2600 3 65.940 30.92
    4) Weight<2280 1 0.000 34.00 *
```

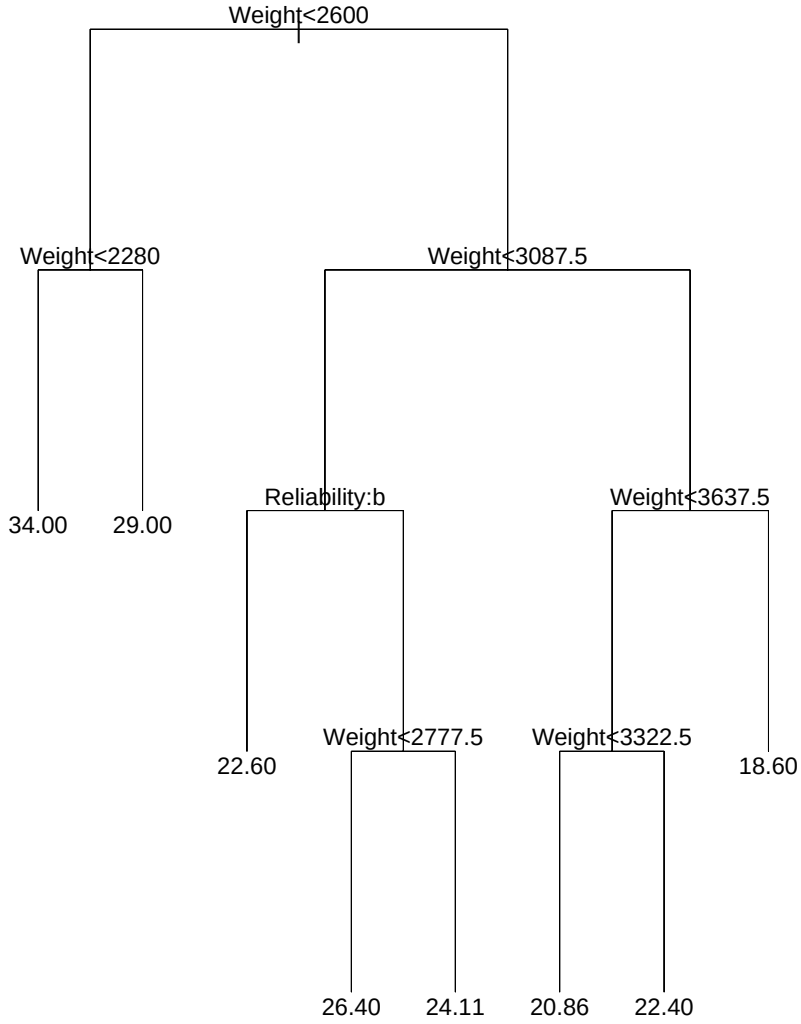

Figure 10.5: *Display of tree relating* `Mileage` *to* `Weight` *and* `Reliability`*. All of the data used to fit the data are complete.*

```
  5) Weight>22figz80 2 26.000 29.00 *
3) Weight>2600 8 81.060 22.58
  6) Weight<3087.5 3 11.770 24.32
   12) Reliability:2 0 0.000 22.60 *
   13) Reliability:1,3,4,5 0 0.000 24.93
     26) Weight<2777.5 0 0.000 26.40 *
     27) Weight>2777.5 0 0.000 24.11 *
  7) Weight>3087.5 5 10.680 20.65
```

```
14) Weight<3637.5 4 17.000 21.50
  28) Weight<3322.5 3 8.918 20.86 *
  29) Weight>3322.5 1 5.760 22.40 *
15) Weight>3637.5 1 0.160 18.60 *
```

Notice that there are no observations in the nodes (12, 13, 26, 27) at or below the split on `Reliability`.

10.5 PRUNING AND SHRINKING

Since tree size is not limited in the growing process, a tree may be more complex than necessary to describe the data. Two functions assess the degree a tree can be simplified without sacrificing goodness-of-fit. The `prune.tree` function achieves parsimonious description by reducing the nodes on a tree, whereas the `shrink.tree` function *shrinks* each node towards its parent.

Both functions take the following arguments:

- tree — Fitted model object of class tree.
- k — cost complexity parameter (for `prune.tree`); shrinkage parameter (for `shrink.tree`).
- newdata — a data frame containing the values at which predictions are required. if missing, the data used to grow the tree are used.

Pruning

Pruning successively snips off the *least important splits*. Importance of a subtree is measured by the cost-complexity measure:

$$D_k(T') = D(T') + k \cdot \text{size}(T')$$

where

$$\begin{aligned} D_k(T') &= \text{the deviance of the subtree } T', \\ \text{size}(T') &= \text{the number of terminal nodes of } T', \\ k &= \text{the cost-complexity parameter.} \end{aligned}$$

Cost-complexity pruning determines the subtree T' that minimizes $D_k(T')$ over all subtrees. The larger the k, the fewer nodes there will be.

The `prune.tree` function takes a cost-complexity parameter argument `k`, which can be either a scalar or a vector. A scalar `k` defines one subtree of `tree` whereas a vector `k` defines a sequence of subtrees minimizing the cost-complexity measure. If the `k` argument is not supplied, a nested sequence of subtrees is created by recursively snipping off the least important splits.

Figure 10.6 shows the deviance decreasing as a function of the number of nodes and the cost-complexity parameter `k`.

```
> plot(prune.tree(kyph.tree))
```

Since over one half of the reduction in deviance is explained by the first three nodes, we limit the tree to three nodes.

```
> plot(prune.tree(kyph.tree, k = 5))
> text(prune.tree(kyph.tree, k = 5))
> summary(prune.tree(kyph.tree, k = 5))
Classification tree:
prune.tree(tree = kyph.tree, k = 5)
Variables actually used in tree construction:
[1] "Start" "Age"
Number of terminal nodes: 3
Residual mean deviance: 0.734 = 57.25 / 78
Misclassification error rate: 0.1728 = 14 / 81
```

By comparing this to the summary of the full tree in the section Displaying Trees, we see that reducing the number of nodes from 10 to 3 simplifies the model, but at the cost of increased misclassification.

Increasing the complexity of the tree to 6 nodes drops the misclassification to a rate comparable to that of the full tree with 10 nodes:

```
> summary(prune.tree(kyph.tree, k = 2))
Classification tree:
prune.tree(tree = kyph.tree, k = 2)
Number of terminal nodes: 6
Residual mean deviance: 0.6383 = 47.88 / 75
Misclassification error rate: 0.1358 = 11 / 81
```

Figure 10.6 shows `kyph.tree` pruned to 3 and 6 nodes.

Shrinking

Shrinking reduces the number of *effective* nodes by shrinking the fitted value of each node towards its parent node. Shrunken fitted values, for a shrinking parameter k, are computed according to the recursion:

$$\hat{y}(\text{node}) = k \cdot \bar{y}(\text{node}) + (1 - k) \cdot \hat{y}(\text{parent})$$

where

$\bar{y}(\text{node})$ = the usual fitted value for a node,

$\hat{y}(\text{parent})$ = the *shrunken* fitted value for the node's parent.

The `shrink.tree` function *optimally* shrinks children nodes to their parent, based on the magnitude of the difference between $\bar{y}(\text{node})$ and $\bar{y}(\text{parent})$.

The shrinkage parameter argument ($0<k<1$) may be a scalar or a vector. A

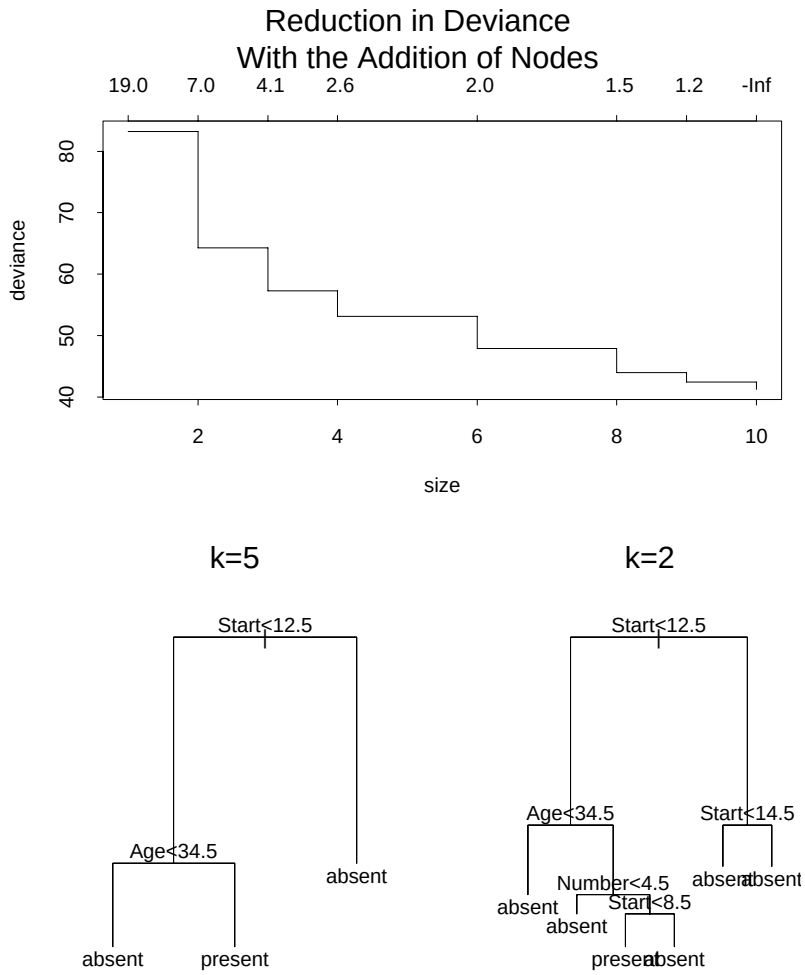

Figure 10.6: *A sequence of plots generated by the* `prune.tree` *function.*

scalar `k` defines one shrunken version of `tree`, whereas a vector `k` defines a sequence of shrunken trees obtained by optimal shrinking for each value of `k`. If the `k` argument is not supplied, a nested sequence of subtrees is created by recursively shrinking the tree for a default sequence of values (roughly .05 to .91) of `k`.

Figure 10.7 shows the deviance decreasing as a function of the number of *effective* nodes and the shrinkage parameter, `k`.

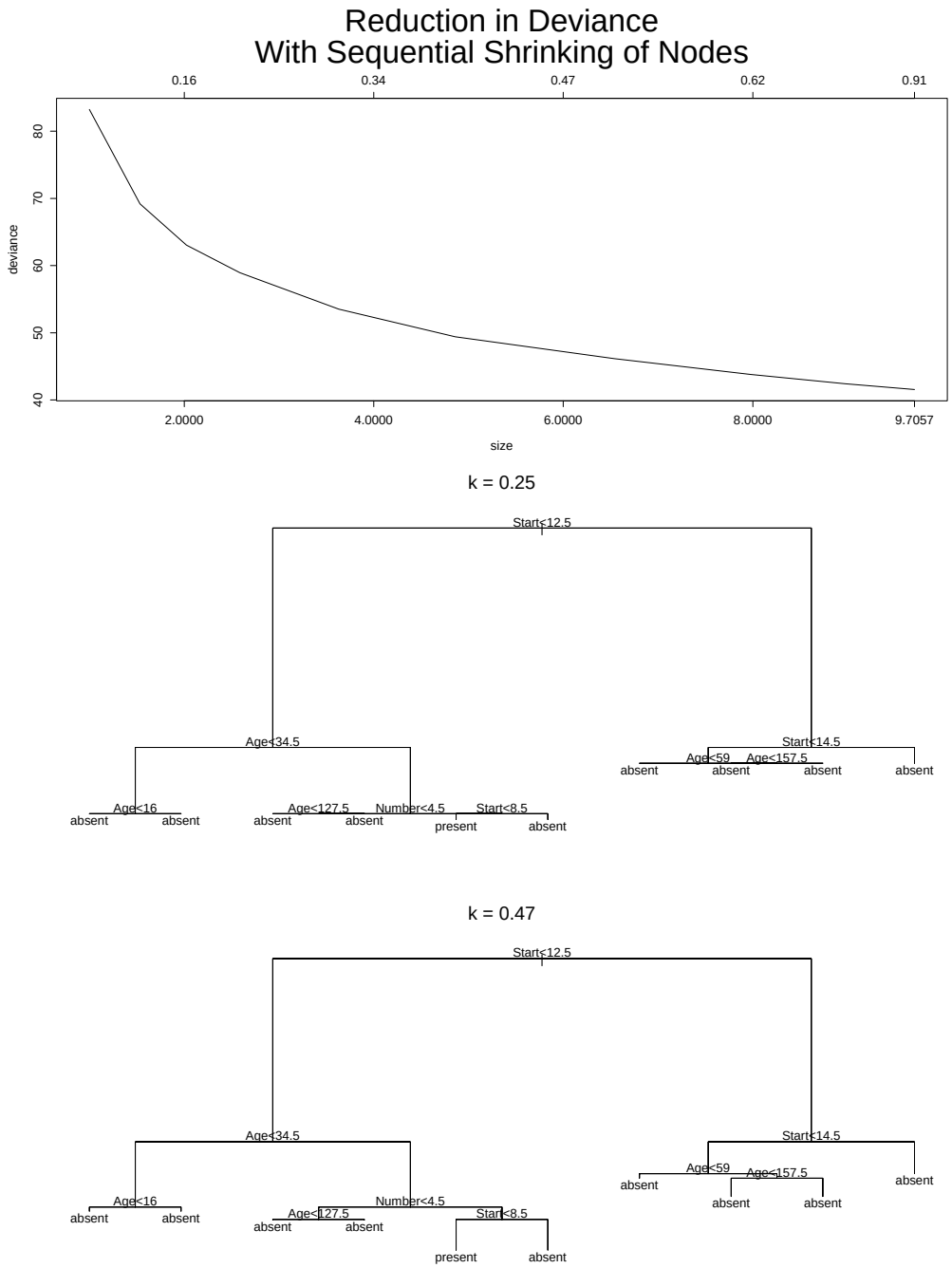

Figure 10.7: *A sequence of plots generated by the* `shrink.tree` *function.*

```
> plot(shrink.tree(kyph.tree))
```

Limit the tree to three *effective nodes* as done with pruning as follows:

```
> kyph.tree.sh.25 <- shrink.tree(kyph.tree, k = 0.25)
> plot(kyph.tree.sh.25)
> text(kyph.tree.sh.25)
> title("k = 0.25")
> summary(kyph.tree.sh.25)

Classification tree:
shrink.tree(tree = kyph.tree, k = 0.25)
Number of terminal nodes: 10
Effective number of terminal nodes: 2.8
Residual mean deviance: 0.7385 = 57.75 / 78.2
Misclassification error rate: 0.1358 = 11 / 81
```

The lower misclassification rate is maintained even with only three effective nodes.

Expand the tree to three *effective nodes* as follows:

```
> kyph.tree.sh.47 <- shrink.tree(kyph.tree, k = 0.47)
> plot(kyph.tree.sh.47)
> text(kyph.tree.sh.47)
> title("k = 0.47")
> summary(kyph.tree.sh.47)
Classification tree:
shrink.tree(tree = kyph.tree, k = 0.47)
Number of terminal nodes: 10
Effective number of terminal nodes: 6
Residual mean deviance: 0.6281 = 47.11 / 75
Misclassification error rate: 0.1358 = 11 / 81
```

Note that no change other than a decrease in the residual mean deviance and an increase in the number of effective nodes.

10.6 GRAPHICALLY INTERACTING WITH TREES

A number of S-PLUS functions use the tree metaphor to diagnose tree-based model fits. The functions are naturally grouped by components of trees: *subtrees*, *nodes*, *splits*, and *leaves*. Except for the leaves functions, these functions allow you to interact graphically with trees, to perform a *what-if* analysis. You can also use these functions noninteractively by including a list of nodes as an argument. The goal is to better understand the fitted model, examine alternatives, and interpret the data in light of the model.

You can select subtrees from a large tree, and apply a common function (such

as a plot) to the *stand* of resulting trees. Similarly, you can snip subtrees from the large tree, in order to gain resolution and label the top of the tree.

You can browse nodes to obtain important information too bulky to be usefully placed on a tree plot. You can obtain the names of observations which occur in a node. By examining the path (that is, the sequence of splits) that lead to a node, you can characterize the observations in that node.

You may compare *optimal* splits (generated by the tree-growing algorithm) to other potential splits. This helps to discover splits on variables that may shed light on the nature of the data. Any split divides the observations in a node into two groups. Therefore, you can compare the distribution of observations of a chosen variable in each of the two groups. This helps characterize the two groups, and also find variables with good discriminating abilities. You may regrow the tree, after designating a different split at a node.

The leaves of the trees represent the most homogeneous partitions of the data. You can investigate the differences across leaves by studying the distribution or summary statistics of chosen variables.

Subtrees

You can select or delete subtrees by *subscripting* the original tree, or by using one of the two functions described below.

The function `snip.tree` function deletes subtrees; that is, it snips branches off a specified tree. One goal may be to gain resolution at the top of the tree so that it can be labeled.

The graphical interface, using a mouse, proceeds as follows:

- first click informs you of the change in tree deviance if that branch is snipped off.
- second click removes the branch from the tree.

Figure 10.8 shows the result of snipping three branches off `kyph.tree`.

```
> par(mfrow=c(3,1))
> plot(kyph.tree)
> plot(kyph.tree)
> kyph.tree.sn <- snip.tree(kyph.tree)
node number: 4
   tree deviance = 41.24
   subtree deviance = 42.74
node number: 10
   tree deviance = 42.74
   subtree deviance = 43.94
node number: 6
   tree deviance = 43.94
   subtree deviance = 47.88
```

```
> plot(kyph.tree.sn)
> text(kyph.tree.sn,cex=1)
```

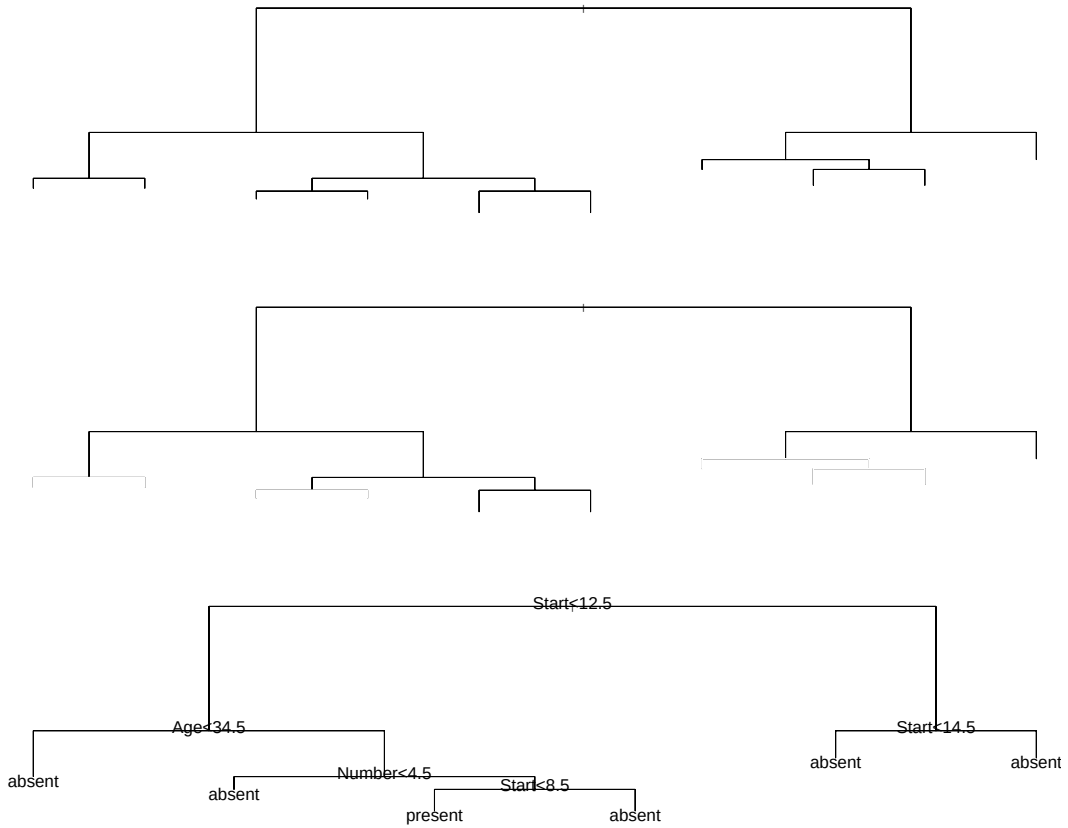

Figure 10.8: *A sequence of plots created by snipping branches from the top tree.*

For noninteractive use, we can equivalently supply the node numbers in `snip.tree(kyph.tree,c(4,10,6))`. Negative subscripting is a convenient shorthand: `kyph.tree[-c(4,10,6)]`.

Similarly, the function `select.tree` function selects subtrees of a specified tree. For each node specified in the argument list or selected interactively, the function returns a tree object rooted at that node. These can in turn be plotted, etc.

Nodes

Several S-PLUS functions encourage the user to obtain more detailed information about nodes. Each of them take a tree object as a required argument, and an optional list of nodes. If the node list is omitted, graphical interaction is expected. The functions return a list, with one component for

each node.

The `browser` function returns a summary of the information contained in a node. Interactively, you obtain information on the second and fifth nodes of `kyph.tree` by:

```
> browser(kyph.tree)
node number: 2
 split: Start<12.5
 n: 35
 dev: 47.800
 yval: absent
        absent present
[1,] 0.5714286 0.4285714
node number: 5
 split: Age>34.5
 n: 25
 dev: 34.300
 yval: present
     absent present
[1,]   0.44    0.56
```

Alternatively, provide a list of nodes as an argument:

```
> browser(kyph.tree,c(2,5))
     var  n      dev   yval splits.cutleft splits.cutright
2    Age 35 47.80357 absent          <34.5           >34.5
5 Number 25 34.29649 present           <4.5            >4.5
  yprob.absen yprob.present
2   0.5714286     0.4285714
5   0.4400000     0.5600000
```

The `identify` function is another generic function with a tree-specific method. The following noninteractive call lists the observations in the eighth and ninth nodes of `kyph.tree`:

```
> identify(kyph.tree,nodes=c(8,9))
$"8":
[1] "4" "14" "26" "29" "39"
$"9":
[1] "13" "21" "41" "68" "71"
```

The function `path.tree` returns the *path* (sequence of splits) from the root to any node of a tree. This is useful in those cases where overplotting results if the tree is labeled indiscriminately. As an example, we interactively look at the path to the rightmost terminal node of the kyphosis tree:

```
> path.tree(kyph.tree)

 node number: 27
   root
   Start>12.5
   Start<14.5
   Age>59
   Age>157.5
```

By examining the path, we can determine that the children in this node are more than 157.5 months old, and the beginnings of the range of vertebrae involved are between 12.5 and 14.5.

Splits

The recursive partitioning algorithm underlying the `tree` function chooses the "best" set of splits that partition the predictor variable space into increasingly homogeneous regions. However, it is important to remember that this is just an algorithm. There may be other splits that also help you understand the data. The functions in this section help to examine alternative splits.

Using the `burl.tree` function, you can select a node either interactively or through the argument list, and observe the *goodness-of-split* for each predictor in the model formula. The goodness-of-split criterion is the difference in deviance between the node and its children (defined by the tentative split). Large differences correspond to important splits. Reduction in deviance is plotted against a quantity which depends upon the form of the predictor:

- numeric — each possible cut-point split.
- factor — a decimal equivalent of the binary representation of each possible subset split. The plotting character is a string labeling the left split.

In the following example and figure 10.9, competing splits are plotted for each of the four predictor variables in the `cu.summary` data frame.

```
> reliab.tree <- tree(Reliability ~
+ Price + Country + Mileage + Type,
+ na.action = na.tree.replace, data = cu.summary)
> tree.screens() #establish plotting regions
[1] 1 2
> plot(reliab.tree,type="u")
> text(reliab.tree)
> burl.tree(reliab.tree) # Now click at the root node
```

The `burl.tree` function returns a list. For each variable there is a component which contains the necessary information for doing each of the plots.

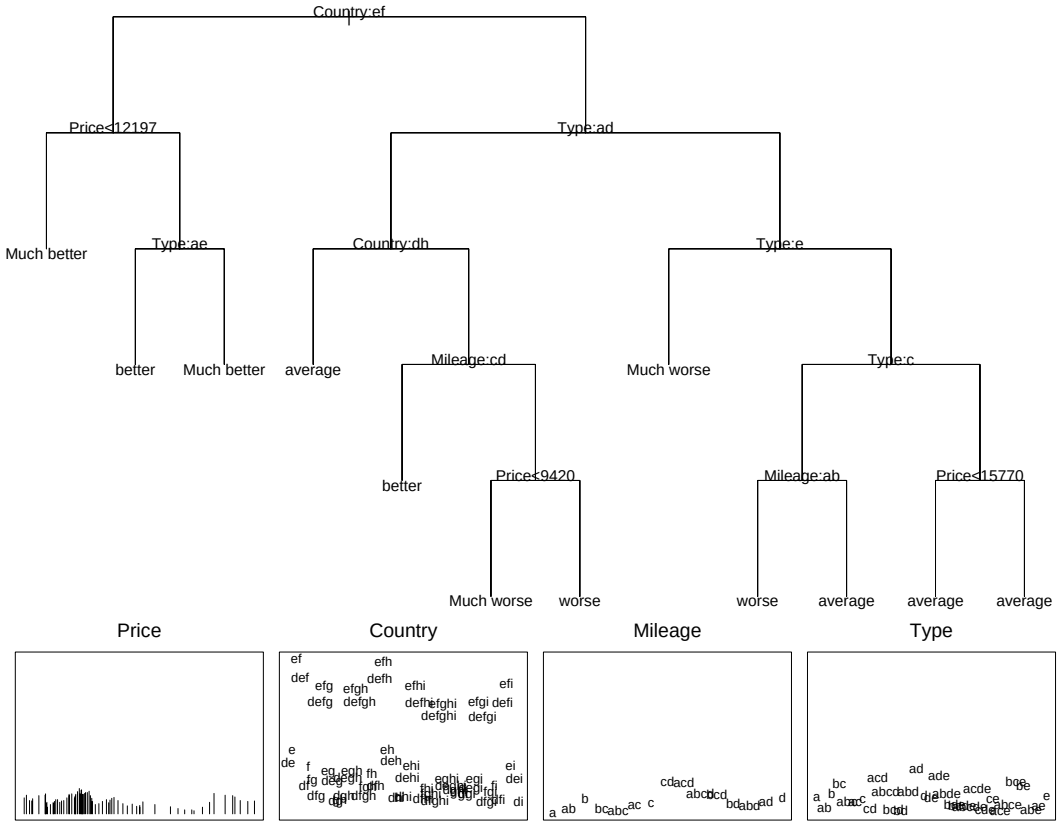

Figure 10.9: *A tree for* `Reliability` *in the* `cu.summary` *data frame with a burl plot of the four predictors for the root node.*

The burl plots show that the most important splits involve the variable `Country`. The candidate splits on this variable divide into two groups; the top group discriminates better than the bottom. The very best split is the one labeled `ef = Japan, Japan/USA`. Moreover, this occurs in all candidate splits in the top group. Therefore, we conclude that this is a meaningful split.

The function `hist.tree` requires a list of variable names, in addition to the tree object (and, optionally, a list of nodes). Unlike `burl.tree`, the variables need not be predictors. For a given node, a side-by-side histogram is plotted for each variable. The histogram on the left displays the distribution of the

observations following the left split; similarly the histogram on the right displays the distribution of the observations following the right split.

Figure 10.10 is produced by the following expression:

```
> reliab.tree.2 <- tree(Reliability ~
+ Country + Mileage + Type,
+ na.action = na.tree.replace, data = cu.summary)
> tree.screens() #establish plotting regions
[1] 1 2
> plot(reliab.tree.2, type="u")
> text(reliab.tree.2)
> hist.tree(reliab.tree.2, Price, Mileage, nodes=1)
```

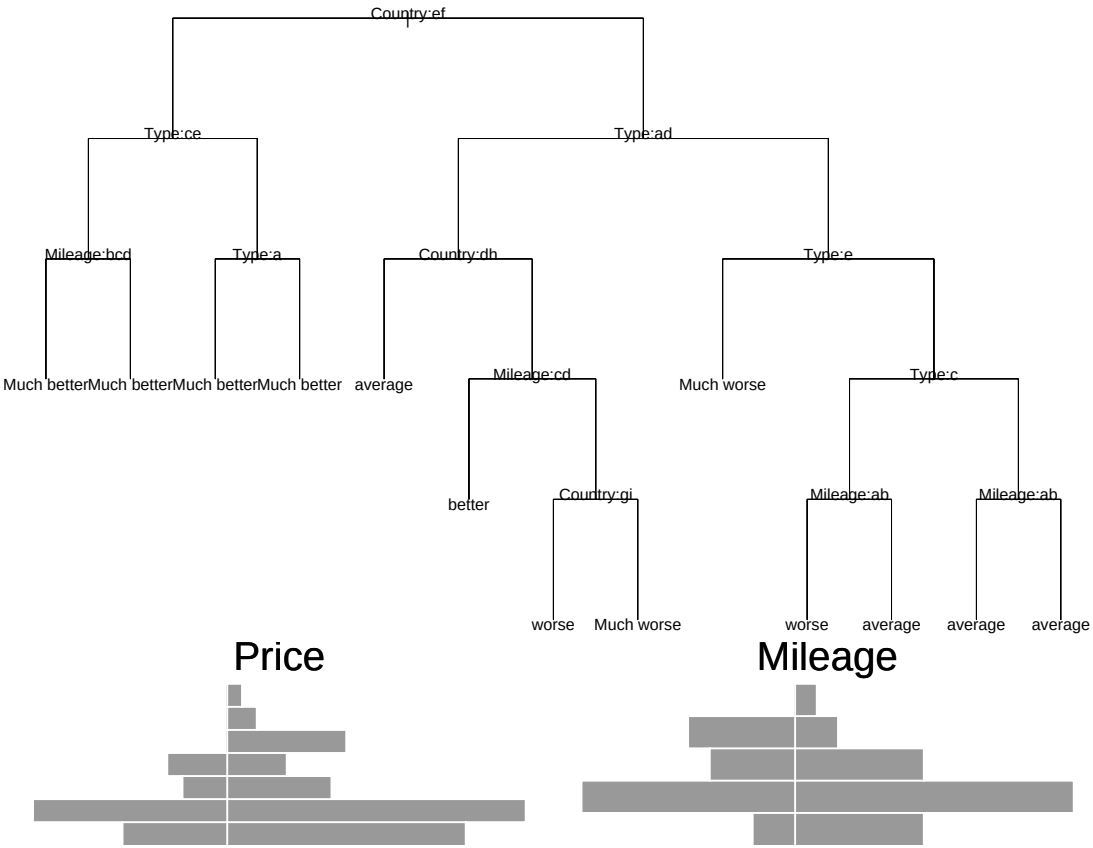

Figure 10.10: *A tree for* `Reliability` *in the* `cu.summary` *data frame.*

The figure shows that Japanese cars manufactured here or abroad tend to be less expensive and more fuel efficient than others. The lower portion of the

plot displays a side-by-side histogram for each of the variables `Price` and `Mileage`. Note that it is possible to get a histogram of this variable even though the formula for this tree does not include `Price`.

Manual Splitting and Regrowing

After examining competitor splits at a node, you may wonder what the tree would look like if the node were split differently. You can achieve this by using the `edit.tree` function.

The arguments to `edit.tree` are:

- `object` fitted model object of class `"tree"`.
- `node` number of the node to edit.
- `var` character string naming variable to split on.
- `splitl` left split. Numeric for continuous variables; character string of levels that go left for a factor.
- `splitr` right split. Character string of levels that go right for a factor.

As an example, look at a burl of `kyph.tree` at the root node for the variable `Start`.

```
> kyph.burl <- burl.tree(kyph.tree, node = 1)
> kyph.burl$Start
   Start       dev numl
 1   1.5  1.001008    5
 2   2.5  1.887080    7
 3   4.0  2.173771   10
 4   5.5  5.098140   13
 5   7.0 11.499747   17
 6   8.5 17.946393   19
 7   9.5 12.812267   23
 8  10.5 12.821041   27
 9  11.5 10.136948   30
10  12.5 18.977175   35
11  13.5 13.927629   47
12  14.5 17.508746   52
13  15.5 12.378558   59
14  16.5  2.441679   76
```

Use `edit.tree` to regrow the tree with a designated split at `Start = 8.5`. The result is shown in figure 10.11.

```
> kyph.tree.edited <- edit.tree(kyph.tree, node = 1,
```

```
+ var = "Start", splitl = 8.5)
```

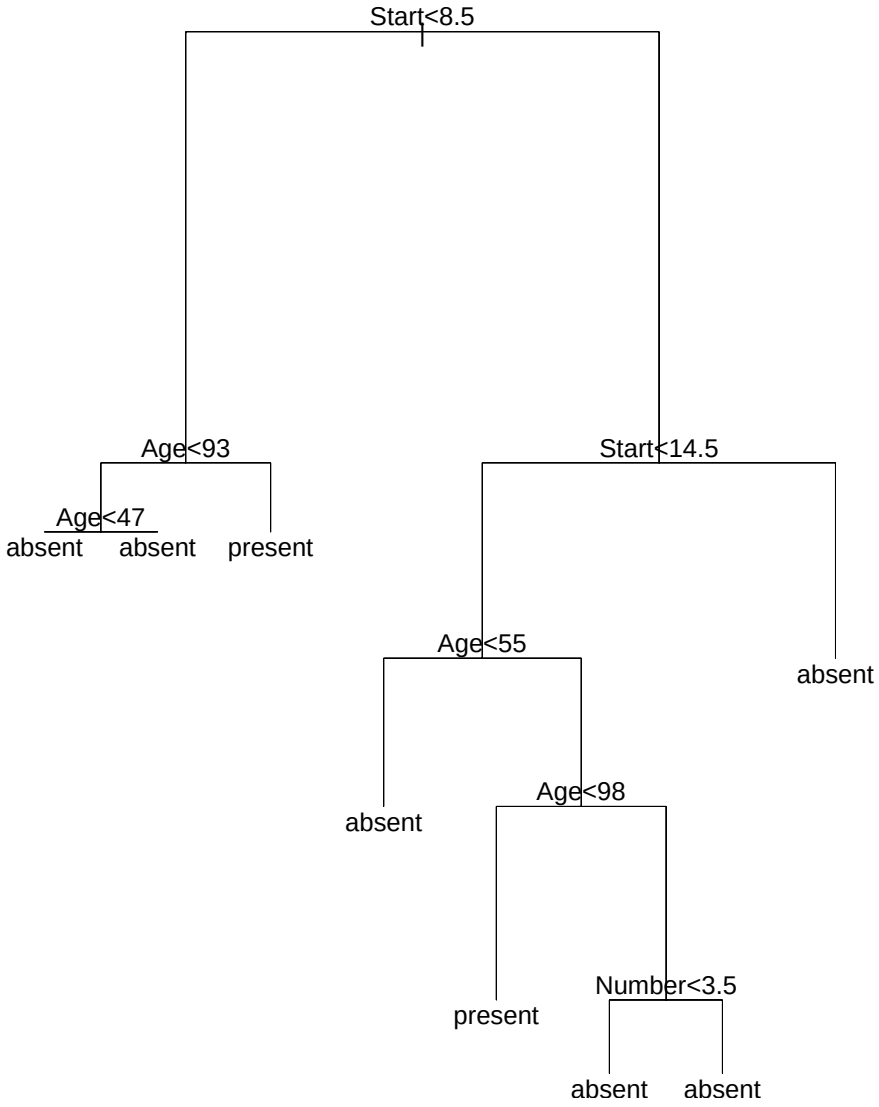

Figure 10.11: `kyph.tree` *regrown at the root node with a split at* `Start = 8.5`.

Leaves

Two noninteractive functions show the distribution of a variable over *all* terminal nodes of a tree.

The function `tile.tree` plots histograms of a specified variable for observations in each leaf. This function can be used, for example, to display

class probabilities across the leaves of a tree. Figure 10.12 shows the distribution across leaves for `Kyphosis`.

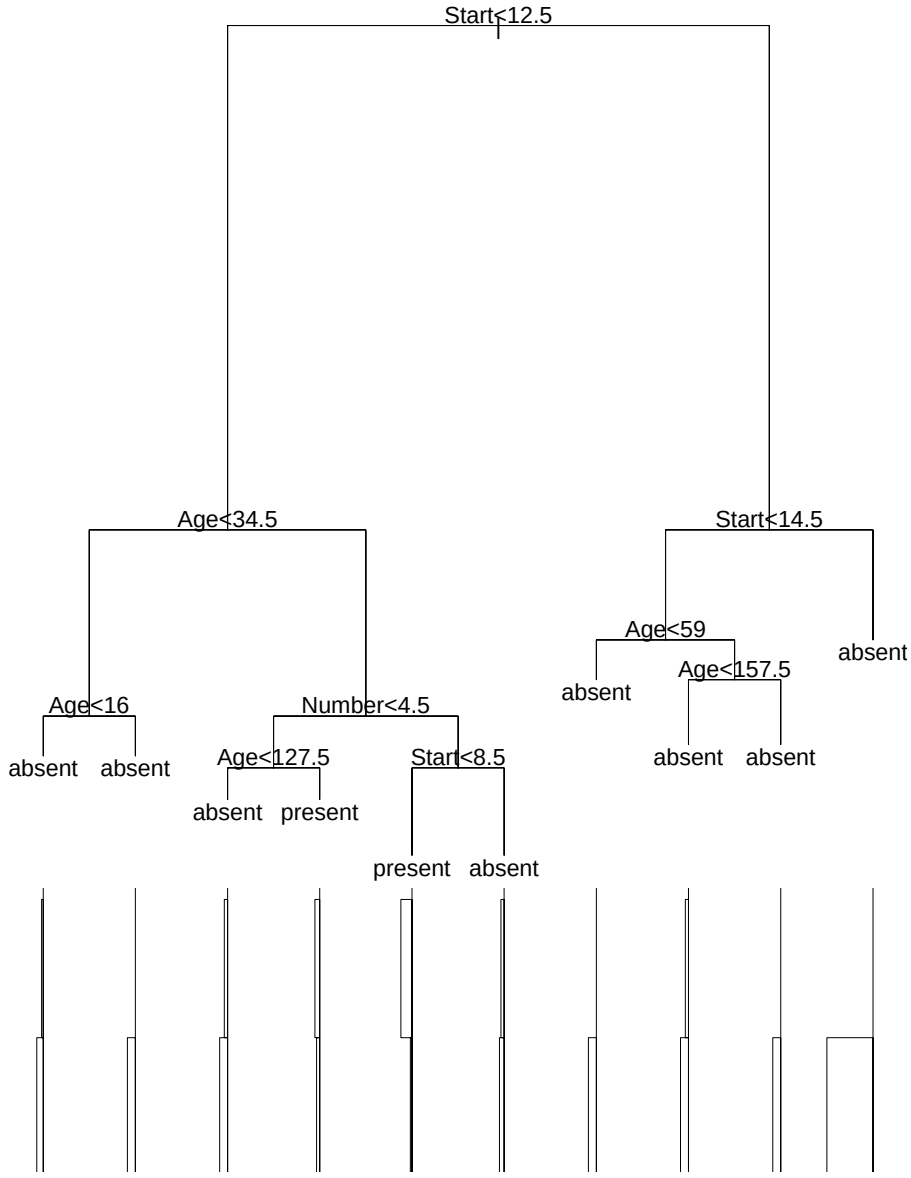

Figure 10.12: *A tree of the* `kyphosis` *data with a tile plot of* `Kyphosis`.

```
> tree.screens() #split plotting screen
> plot(kyph.tree,type="u")
```

```
> text(kyph.tree)
> tile.tree(kyph.tree, Kyphosis)
```

A related function, `rug.tree`, shows the average value of a variable over the leaves of a tree. The optional argument `FUN` allows you to summarize the variable with something other than the mean (for example, trimmed means, medians). Figure 10.13 shows the rug plot of medians for `Start`.

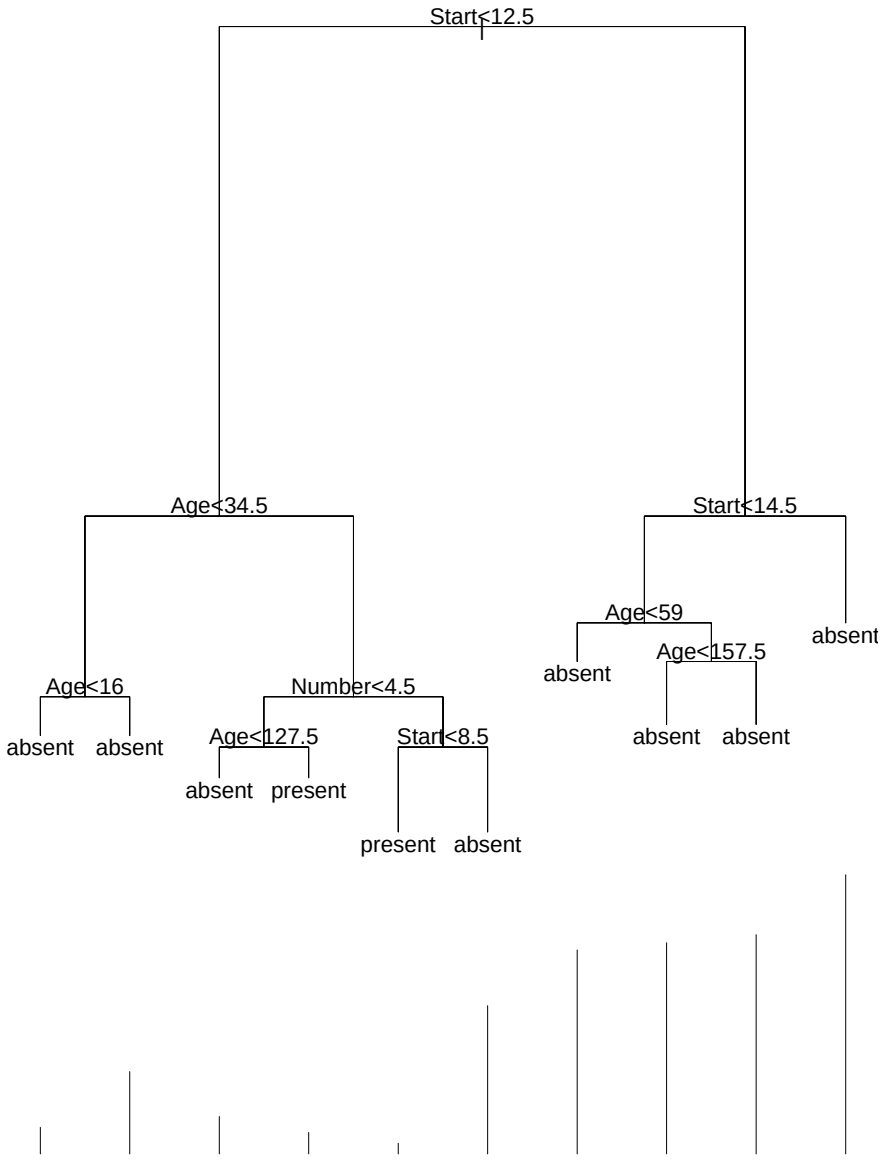

Figure 10.13: *A tree of the* `kyphosis` *data with a rug plot of* `Start`.

```
> rug.tree(kyph.tree, Start, FUN = median)
```

10.7 REFERENCES

Breiman, L. and Friedman, J. H. and Olshen, R. A. and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth and Brooks/Cole, Monterey, CA.

Chambers, J.M. and Hastie, T.J. (1992). *Statistical Models in S*. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove, CA.

LINEAR AND NONLINEAR MIXED-EFFECTS MODELS

11

Mixed effects models analyze data containing both fixed and random effects.

LINEAR AND NONLINEAR MIXED-EFFECTS MODELS

Mixed-effects models provide a powerful and flexible tool for analyzing clustered data encountered in repeated-measures and nested designs. The functions, classes, and methods described here extend the S-PLUS 3.3 linear and nonlinear modeling facilities such as `lm`, `varcomp`, and `nls`. They are applied to six examples of repeated-measures data generated by observing a number of clusters repeatedly under varying experimental conditions.

In the first part of this chapter, analysis of repeated-measures data using linear mixed-effects models is illustrated with the new S-PLUS function `lme`. The major distinction between `lme` and the linear model function `lm` is that `lme` considers random effects and within-cluster (or within-subject) covariance.

Following the general discussion of the `lme` class, specifications for the structured covariance matrix for random-effects parameters and within-cluster errors are described. Beyond the available covariance structures, customized structures can also be designed by the user. Examples of the construction of covariance matrices illustrate the power of `lme` in handling complex problems.

In the second part of this chapter, analysis of repeated-measures data using nonlinear mixed-effects models is illustrated with the new S-PLUS function `nlme`. The `nlme` class inherits from the `lme` class, so many methods designed for `lme` are also incorporated into `nlme`. For example, the methods used to design the covariance matrix for the random-effects parameters in `lme` are applicable to `nlme`.

11.1 LINEAR MIXED-EFFECTS MODELS

In repeated-measures data, multiple measurements are obtained from individuals on different occasions. For each individual, serial correlation usually exists, so a multivariate approach with structured covariance for individuals is very appealing. Although the distribution for the response is the same for each individual, some parameters might vary over individuals. The following examples show the features of such data and illustrate the new S-PLUS function `lme`.

Example: Orthodont Data

These repeated-measures data come from an orthodontic study presented in Potthoff and Roy (1964). They consist of four measurements of the distance in millimeters from the center of the pituitary to the pteryomaxillary fissure made at ages 8, 10, 12, and 14 years on 16 boys and 11 girls. The purpose of the study is to model the distance as a function of age, with consideration of

gender difference. The data are available in a data frame called `Orthodont`, with columns `Subject`, `Sex`, `age`, and `distance` as indicated below.

```
> Orthodont
    Subject Sex age distance
1         1   0   8     26.0
2         1   0  10     25.0
3         1   0  12     29.0
4         1   0  14     31.0
. . .
105      27   1   8     24.5
106      27   1  10     25.0
107      27   1  12     28.0
108      27   1  14     28.0
```

Exploratory Data Analysis

In the `Orthodont` data set, subjects are classified into two groups by `Sex`, an indicator variable assuming the value 0 for boys and 1 for girls. Each subject has four measures of distance, and the 108 total records are grouped into 27 clusters by `Subject`. They are displayed in figure 11.1 using the code found below. You can also use Trellis graphics to visualize these repeated-measures data, as indicated.

Figure 11.1: *The plot of the* `Orthodont` *data suggests a linear model is adequate.*

```
> motif()
> attach(Orthodont)
> plot(age, distance, type="n", xlab="Age (years)",
+   ylab="Distance (mm)")
> points(age[Sex == “0”], distance[Sex == “0”],
+   type="p", pch=0)
> points(age[Sex == “1”], distance[Sex == “1”],
+   type="p", pch=5)
> for (i in unique(Subject)) {
+   lines(age[Subject==i], distanceg[Subject==i],
+   type="l", lty=2)
+   }
> legend(12, 18.5, marks=c(0,5), legend=c("Boys","Girls"))
> detach(Orthodont)

# Trellis graphics examples:
> trellis.device(motif)
> xyplot(distance ~ age | Subject, data = Orthodont)

# The following function can be used too:
> Orthodont.plot()
```

The plot suggests that a linear model is adequate to explain `distance` as a function of `age`, but that the intercept and the slope vary with the individual. To explore this further, one could fit a simple regression to each subject by using the function `lm` and plot the parameter estimates. Alternatively, the new function `lmList` can be used to fit a linear model to each cluster. For example, the cluster variable in `Orthodont` is `Subject`; a simple regression using the covariate `age` for each `Subject` is fitted by `lmList` and displayed by the generic function `coef` as follows.

```
> lmList.fit <- lmList(distance ~ age, cluster = ~ Subject,
+                      data = Orthodont)
> coef(lmList.fit)
   (Intercept)   age
 1       17.30 0.950
 2       14.85 0.775
 3       16.00 0.750
 . . .
25       18.10 0.275
26       13.55 0.450
27       18.95 0.675
> plot(coef(lmList.fit), xlab = "Estimated intercepts",
+      ylab = "Estimated slopes")
```

Figure 11.2 shows that the parameter estimates, with one exception, scatter around 13 to 25 for the intercept and 0.17 to 1.13 for the slope. Our exploratory analysis indicates that a mixed-effects model is suitable. The

Figure 11.2: *The parameter estimates.*

corresponding linear mixed-effects model is

$$d_{ij} = (\beta_0 + b_{i0}) + (\beta_1 + b_{i1}) \cdot \text{age}_j + \varepsilon_{ij} \tag{11.1}$$

where d_{ij} represents the distance for the ith individual at age j, β_0 and β_1 are the population average intercept and the population average slope, b_{i0} and b_{i1} are the random effects in intercept and slope associated with the ith individual, and ε_{ij} is the within-subject error term. It is assumed that the $\boldsymbol{b}_i = (b_{i0}, b_{i1})^T$ are independent and identically distributed with a $N(\mathbf{0}, \sigma^2 D)$ distribution and that the ε_{ij} are independent and identically distributed with a $N(0, \sigma^2)$ distribution, and are independent of the $\boldsymbol{b}_i$.

Of interest for these data is whether the curves in figure 11.1 show significant differences between boys and girls. Model (11.1) can be modified to

$$\begin{aligned} d_{ij} = {} & (\beta_{00} + \beta_{01}\text{Sex}_i + b_{i0}) + \\ & (\beta_{10} + \beta_{11}\text{Sex}_i + b_{i1})\text{age}_i + \varepsilon_{ij} \end{aligned} \tag{11.2}$$

to test for sex-related differences in intercept and slope. In model (11.2), the parameters β_{00} and β_{10} represent the population average intercept and slope for the boys, while β_{01} and β_{11} represent the changes in population average intercept and slope for girls. Differences between boys and girls can be evaluated by testing whether β_{01} and β_{11} are significantly different from zero. The remaining terms in model (11.2) are defined as in model (11.1).

Example: Pixel Data

These data consist of repeated measures of mean pixel values from CT scans, taken from an experiment conducted by Deborah Darien at the Department of Medical Sciences, School of Veterinary Medicine, University of Wisconsin, Madison. CT scans of the right and the left lymphnodes in the axillary region of 10 dogs were measured over a period of 21 days after application of contrast media. The purpose of the experiment was to model the mean pixel value as a function of time, so as to estimate the time when the maximum mean pixel value was attained. The data are available in a data frame called `Pixel` with columns `Dog`, `Side`, `day`, and `pixel` as shown below.

```
> Pixel
    Dog Side day   pixel
1     1    r   0 1045.81
2     1    r   1 1044.54
3     1    r   2 1042.93
4     1    r   4 1050.44
. . .
101   9    l   8 1099.52
102  10    l   4 1132.26
103  10    l   6 1154.51
104  10    l   8 1161.07
```

The 104 total records are grouped into 10 clusters by `Dog`. For the values of `Dog`, up to 14 repeated measures of `Pixel` by `day` are nested in the factor `Side`. The data are displayed in figure 11.3 using the commands below. You can also use the function `Pixel.plot` to display these data with Trellis graphics.

```
> attach(Pixel)
> plot(day, pixel, type="n", ylab="Mean Pixel Value",
+      xlab="Time (days)")
> for (i in 1:10) {
+ points(day[Side == "r" & Dog == i],
+        pixel[Side == "r" & Dog == i],
+        lty=1, pch=letters[i], type="b")
```

Figure 11.3: *The pixel data.*

```
+ points(day[Side == "l" & Dog == i],
+        pixel[Side == "l" & Dog == i],
+        lty=2, pch=letters[i], type="b")
+ }
> legend(16,1147,lty=1:2,legend=c("Right", "Left"))
```

A preliminary analysis indicated that the intercept varies with `Side` within `Dog` and the linear term varies with `Dog`, but not with `Side`. A second-order polynomial seems adequate for these data.

The corresponding linear mixed-effects model is

$$\text{pixel}_{ijk} = (\beta_0 + b_{0ij}) + (\beta_1 + b_{1i}) \cdot \text{day}_{ijk} + \beta_2 \cdot \text{day}_{ijk}^2 + \varepsilon_{ijk} \qquad (11.3)$$

where i refers to the dog number (1 through 10), j to the lymphnode side (1=right, 2=left), and k refers to day. The coefficients β_0, β_1, and β_2 denote respectively the intercept, the linear term, and the quadratic term fixed effects; b_{0ij} denotes the intercept random effect (side-specific, nested within dog) and b_{1i} denotes the linear term random effect (dog-specific); and ε_{ijk} denotes the error term. Assume that the $\boldsymbol{b}_i = (b_{0i1},\ b_{0i2},\ b_{1i})^T$ are independent and identically distributed with common distribution $N(\mathbf{0}, \sigma^2 D)$ and that the ε_{ijk} are independent and identically distributed with

common distribution $N(0,\sigma^2)$ and are independent of the $\boldsymbol{b}_i$.

A further assumption about the structure of D in model (11.3) is that the (b_{0i1}, b_{0i2}) have equal variance and are independent of the b_{1i}. This can be rephrased by saying that the covariance matrix of random effects D can be partitioned into four blocks as follows.

$$D = \begin{bmatrix} D_{11} & D_{12} & 0 \\ D_{12} & D_{11} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \tag{11.4}$$

The (b_{0i1}, b_{0i2}) form a 2×2 block of random effects with compound symmetry covariance matrix and the b_{1i} form another 1×1 block with an unstructured covariance matrix. Actually, because all the structured forms for a block are equivalent when the block contains only one random effect, any of the structures could be used. Entries in the other two blocks are zero due to the assumption of independence between (b_{0i1}, b_{0i2}) and b_{1i}. An analysis of these data is described in the section Design of the Structured Covariance Matrix for Random Effects.

Example: Ovary Data

These data come from an animal study reported in Pierson and Ginther (1987). The data consist of the number of ovarian follicles greater than 10 mm in diameter recorded daily for each of 11 mares. They were recorded daily from three days before ovulation until three days after the following ovulation. The measurement times were scaled so that ovulation for each mare occurs at times 0 and 1. Since the ovulation cycles vary in length, the number of measurements and the times at which they occurred vary among the mares. The data are stored in the data frame called `Ovary`.

```
> Ovary
    Mare         Time follicles
1      1  -0.13636364        20
2      1  -0.09090909        15
3      1  -0.04545455        19
. . .
306   11   1.05                7
307   11   1.10                5
308   11   1.15                5
```

The 308 total records are grouped into 11 clusters by `Mare`. Negative times are scaled times which occurred before the current estrus cycle, `Time`=0

through `Time`=1. A Trellis graphics function is provided to display these data.

```
> trellis.device(motif)
> Ovary.plot()
```

The objective is to model the number of follicles as a function of `Time`. From figure 11.4 and also as suggested in Lindstrom and Bates (1988), a sinusoidal model of the form

$$y_{ij} = (\beta_1 + b_{i1}) + (\beta_2 + b_{i2}) \cdot sin(2\pi x_{ij}) + (\beta_3 + b_{i3}) \cdot cos(2\pi x_{ij}) + \varepsilon_{ij} \quad (11.5)$$

might be appropriate for the data. Here y_{ij} represents the number of follicles of mare i at time x_{ij}. A simple model would assume that the components of $\boldsymbol{b}_i = (b_{i1}, b_{i2})$ are independent and identically distributed as $N(\mathbf{0}, \sigma^2 D)$ and that the ε_{ij} are independent and identically distributed as $N(0, \sigma^2)$.

In repeated-measures data such as `Ovary`, measurements are often correlated. An alternative model would assume that $\varepsilon_i = (\varepsilon_{i1}, \varepsilon_{i2}, \ldots, \varepsilon_{in_i})^T$ has the multivariate normal distribution $N(\mathbf{0}, \sigma^2 \Lambda_i)$, where Λ_i is an $n_i \times n_i$ positive definite matrix parametrized by a fixed number of parameters, and n_i is the number of measurements on the ith subject. As in `Ovary`, the n_i are not the same for different mares. This example shows the serial variations for each animal over unequal measurement periods. An analysis of these data is described in the section The Structured Covariance Matrix for Within-Cluster Errors.

The lme Class and Related Methods

This section demonstrates the basic calls to `lme` and describes in detail its three major arguments, `fixed`, `random`, and `cluster`. Methods for summarizing the fitted model and comparing different models are applied to the `Orthodont` data.

The lme Function

The `lme` function is used to fit a linear mixed-effects model, as described in Laird and Ware (1982), using either maximum likelihood or restricted maximum likelihood. The approximate standard errors for the fixed effects are derived using the asymptotic theory described in Pinheiro (1994). The `lme` function returns an object of class `lme`. Numerous optional arguments can be used with this function, but a typical call can be as brief as `lme(fixed, random, cluster, data)`.

Only the `fixed` and `cluster` arguments are required. The arguments `fixed` and `random` are formulas, as shown below. Any *linear model formula* (see chapter 2) is allowed, giving the model specification considerable

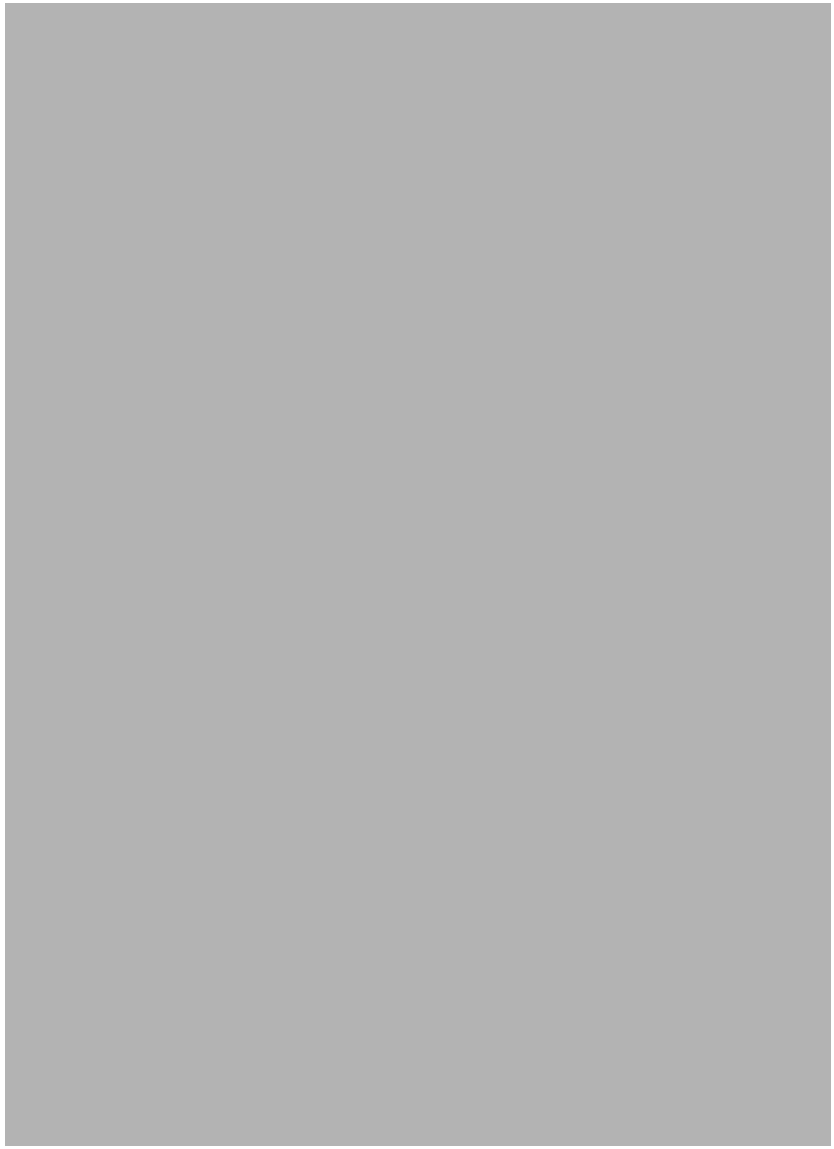

Figure 11.4: *The plot suggests a sinusoidal model might be appropriate.*

flexibility.

For the `Orthodont` data these formulas would be written

```
fixed = distance ~ age, random = ~ age
```

for model (11.1) and

```
fixed = distance ~ age * Sex, random = ~ age
```

for model (11.2). Note that the response variable is given only in the formula for the `fixed` argument. The intercept is automatically included in the `fixed` and `random` arguments. The `cluster` argument is a formula or expression defining the labels for the different subjects in the data. For the `Orthodont` data we would use

```
cluster = ~ Subject
```

for both model (11.1) and model (11.2). The optional argument `data` specifies the data frame containing the variables used in the model. Here is the call to `lme` to fit model (11.1).

```
> Orthodont.fit1 <- lme(fixed = distance ~ age,
+                       random = ~ age, cluster = ~ Subject,
+                       data = Orthodont)
```

The following call fits model (11.2).

```
> Orthodont.fit2 <- lme(fixed = distance ~ age * Sex,
+                       random = ~ age, cluster = ~ Subject,
+                       data = Orthodont)
```

Methods for the print, summary, and anova Functions

The `print` method for the class `lme` gives estimates of the standard errors, correlations of the random effects, the cluster variance, and estimates of the fixed effects. A more complete description of the estimation is returned by `summary`.

```
> Orthodont.fit1
Call:
  Fixed: distance ~ age
 Random:  ~ age
Cluster:  ~ Subject
   Data: Orthodont

Variance/Covariance Components Estimate(s):

  Structure: unstructured
  Parametrization: matrixlog
  Standard Deviation(s) of Random Effect(s)
 (Intercept)       age
    2.327036 0.2264279
 Correlation of Random Effects
    (Intercept)
age -0.6093333
```

```
 Cluster Residual Variance: 1.716204

Fixed Effects Estimate(s):
  (Intercept)       age
    16.76111 0.6601852

Number of Observations: 108
Number of Clusters: 27
> summary(Orthodont.fit2)
. . .
Restricted Loglikelihood: -216.2908
Restricted AIC: 448.5817
Restricted BIC: 470.0387

Variance/Covariance Components Estimate(s):
  Structure: unstructured
  Parametrization: matrixlog
  Standard Deviation(s) of Random Effect(s)
 (Intercept)     age
    2.405606 0.18035
 Correlation of Random Effects
    (Intercept)
age -0.6676482

 Cluster Residual Variance: 1.716195

Fixed Effects Estimate(s):
                   Value Approx. Std.Error  z ratio(C)
(Intercept)  16.3406250        1.01854580  16.0430930
        age   0.7843750        0.08599996   9.1206441
        Sex   1.0321023        1.59575459   0.6467801
    age:Sex  -0.3048295        0.13473604  -2.2624203

 Conditional Correlation(s) of Fixed Effects Estimates
        (Intercept)        age        Sex
    age -0.8801649
    Sex -0.6382847   0.5617958
age:Sex  0.5617958  -0.6382847 -0.8801649
```

The above results show that the `Sex` fixed effect is not significant while the interaction `age:Sex` fixed effect is significant at α-level 0.05 with a p-value of 0.04. These indicate that the average intercept is common to boys and

girls, but the measurement increases faster in boys than in girls.

A likelihood ratio test to evaluate the hypothesis of no sex differences in distance development is returned by the anova method.

```
> anova(Orthodont.fit1, Orthodont.fit2)

 . . .
               Model Df    AIC    BIC  Loglik
Orthodont.fit1     1  6 454.64 470.73 -221.32
Orthodont.fit2     2  8 448.58 470.04 -216.29
                 Test Lik.Ratio   P value
Orthodont.fit1
Orthodont.fit2 1 vs. 2    10.055 0.0065551
```

The likelihood ratio test gives strong evidence against the null hypothesis of no sex differences with a p-value of 0.007. The following uses a likelihood ratio test to test whether the growth rate is only dependent on sex.

```
> Orthodont.fit3 <- lme(fixed = distance ~ age +
+ age:Sex, random = ~ age,
+ cluster = ~ Subject, data = Orthodont)
```

Now use the anova method again.

```
> anova(Orthodont.fit2, Orthodont.fit3)

 . . .
               Model Df    AIC    BIC  Loglik
Orthodont.fit2     1  8 448.58 470.04 -216.29
Orthodont.fit3     2  7 449.77 468.54 -217.88
                 Test Lik.Ratio P value
Orthodont.fit2
Orthodont.fit3 1 vs. 2    3.1843 0.07435
```

As expected, the likelihood ratio test based on α-level = 0.05 indicates that the initial distances do not depend on sex.

The plot Method

Plots of random-effects estimates, residuals, and fitted values can be obtained using the `plot` method for the class `lme`. The following call produces a scatter plot, shown in figure 11.5, of the estimated random effects for intercept and slope in model (11.2).

```
> plot(Orthodont.fit2,pch="o")
```

The outlying point in the upper left corner of figure 11.5 could have a great impact on the correlation and variance estimates.

Residual plots are obtained by specifying `option="r"` in the `plot` function.

```
> par(mfrow = c(2, 2))
> plot(Orthodont.fit3, option="r", pch="o", which=1:2)
> par(fig=c(0,1,0,.5))
```

Figure 11.5: *Scatter plot:* `plot(Orthodont.fit2,pch="0")`.

```
> plot(Orthodont.fit3, option="r", pch="o", which=3)
```

The resulting plots are shown in figure 11.6. The plot of observed versus fitted values indicates that the linear mixed-effects model does a reasonable job of explaining the distance growth. The points fall relatively close to the 45° line, indicating a reasonable agreement between the fitted and the observed values. The residuals versus fitted values plot suggests the presence of three outliers in the data. The remaining residuals appear to be homogeneously scattered around the zero residual line. The boxplots of the residuals by subject suggest that the outliers occurred for subjects 9 and 13. There seems to be considerable variation in the within-subject variability, but it must be remembered that each boxplot represents only four residuals.

Figure 11.7 reproduces the original data plot of figure 11.1, the random-effects estimates scatter plot of figure 11.5, and the residual versus fitted values plot of figure 11.6. Subjects 9 and 13 are distinguished in each plot. These plots show that those extreme values of residuals and estimates of random effects come from subjects 9 and 13. The first plot also shows that the data for subject 9 is probably in error because the measurement decreases substantially between ages 8 and 10 and between ages 12 and 14. The following commands produce the plots in figure 11.7.

```
> par(mfrow=c(2,2))
> plot(age, distance, type="n", xlab="Age (years)",
+                              ylab="Distance (mm)")
```

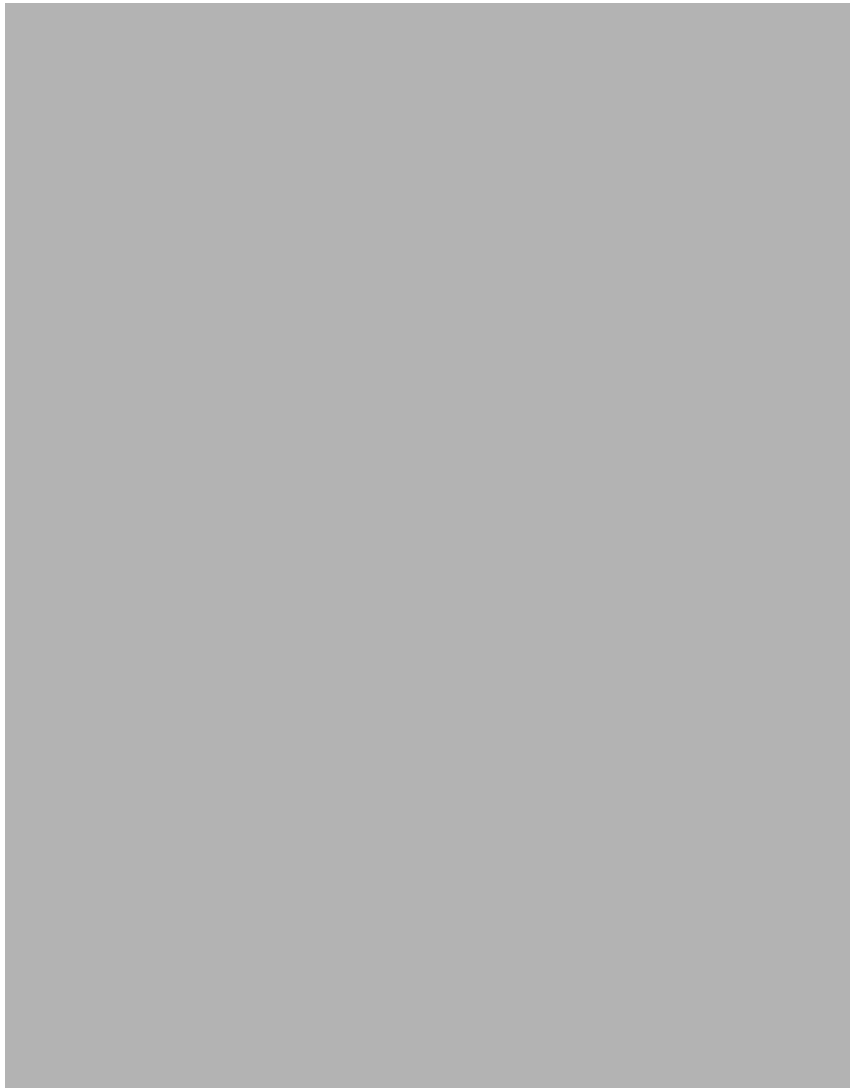

Figure 11.6: *The plot shows that a linear mixed-effects model provides reasonable results.*

```
> for (i in (1:27)[-c(9,13)]) {
+    if (Sex[match(i, Subject)]=="0") {
+    lines(age[Subject==i], distance[Subject==i],
+          type="l", lty=2)}
+    if (Sex[match(i, Subject)]=="1") {
+    lines(age[Subject==i], distance[Subject==i],
```

Figure 11.7: *A plot reproducing the original effects.*

```
+          type="l", lty=2)}}
> text(age[Subject==9], distance[Subject==9], labels="9")
> lines(age[Subject==9], distance[Subject==9],
+       type="l", lty=8)
> text(age[Subject==13], distance[Subject==13], labels="13")
> lines(age[Subject==13], distance[Subject==13],
+       type="l", lty=8)
> # Plot of random effects estimates
> b <- Orthodont.fit3$coefficients$random
> plot(b[, 1], b[, 2],
+      xlab = dimnames(b)[[2]][1],
+      ylab = dimnames(b)[[2]][2], type="n")
> points(b[-c(9,13), 1], b[-c(9,13), 2], pch="o")
> text(b[9,1], b[9,2], labels="9")
> text(b[13,1], b[13,2], labels="13")
> # Plot of residuals versus fitted values
> par(fig=c(.25,.75,0,.5))
> r <- residuals(Orthodont.fit3)$cluster
> p <- fitted(Orthodont.fit3)$cluster
> plot(p, r, ylab="Residuals", xlab="Fitted Values",
```

```
+      type="n")
> abline(0, 0, lty=2)
> points(p[Subject != 9 & Subject != 13],
+        r[Subject != 9 & Subject != 13], pch="o")
> text(p[Subject == 9],r[Subject == 9], labels="9")
> text(p[Subject == 13],r[Subject == 13], labels="13")
```

Other Methods

Standard S-PLUS functions, such as `resid`, `fitted`, and `coef`, for extracting components of fitted objects have methods for the class `lme`. The first two methods return data frames with two columns, `population` and `cluster`, while the last method returns a list with two components, estimates of the fixed-effects and the random-effects.

Estimates of the individual parameters are obtained using the `coef` method.

```
> coef(Orthodont.fit3)
   (Intercept)       age    age:Sex
1     18.18846 0.8421782 -0.2281273
2     15.49069 0.7344114 -0.2281273
3     16.19283 0.7407887 -0.2281273
. . .
27    19.13794 0.8467882 -0.2281273
```

Predicted values are returned by the `predict` method. For example, to predict the average measurement for both boys and girls at ages 14, 15, and 16, as well as for subjects 1 and 20 at age 13, first create a new data frame, say `Orthodont.new`, as follows.

```
> Orthodont.new <-
+   data.frame(Sex = c(0, 0, 0, 1, 1, 1, 0, 1),
+   age = c(14, 15, 16, 14, 15, 16, 13, 13),
+   Subject = c(NA, NA, NA, NA, NA, NA, 1, 20))
```

Then use

```
> predict(Orthodont.fit3, Orthodont.new, ~ Subject)
  cluster fit.cluster fit.population
1      NA          NA       27.30487
2      NA          NA       28.05800
3      NA          NA       28.81113
4      NA          NA       24.11109
```

```
5      NA          NA       24.63609
6      NA          NA       25.16109
7       1    29.13678       26.55175
8      20    22.07238       23.58609
```

to get the cluster and population predictions.

Design of the Structured Covariance Matrix for Random Effects

Structured covariance matrices for the random effects can be specified using the optional argument `re.structure` in the call to `lme`. If there are q random effects, the covariance matrix is a $q \times q$ symmetric matrix. Predefined structures available include

- `unstructured` for a general covariance matrix (requiring $q(q+1)/2$ distinct parameters);
- `diagonal` for independent random effects with possibly different variances (q parameters);
- `identity` for independent random effects with the same variance (1 parameter);
- `compsymm` for a compound symmetry structure where all random effects have the same variance and the same correlation (2 parameters);
- `ar1` for a common variance and AR(1) correlation structure (2 parameters) as in Box, Jenkins, and Reinsel (1994).

The random effects can also be grouped into separate blocks, using the optional argument `re.block`. Random effects belonging to different blocks are assumed to be independent. Different covariance structures can be specified for each block. How to specify a covariance structure is demonstrated in the following analysis of the `Pixel` data.

The Structured Covariance Matrix and Random-Effects Blocks

The `re.block` argument to `lme` is used to specify blocks of random effects in the form of a `list` whose components are vectors or scalars defining the number of the random effects that belong to the block. For model (11.3), apply `lme` as follows.

```
> Pixel.fit <- lme(fixed = pixel ~ day + day^2,
+           random = ~ Side + day - 1,
+           cluster = ~ Dog, data = Pixel,
+           re.block = list(c(1,2),3),
+           re.structure = c("compsymm", "unstructured"))
```

The number assigned to a random effect is established by the way it is extracted from the `random` formula. Alternatively, the names of the random effects can be used in `re.block`. For example, after the following assignments

```
> Side.r <- as.integer(Pixel$Side == "r")
> Side.l <- as.integer(Pixel$Side == "l")
```

either

```
..., random = ~ Side.r + Side.l + day - 1,
re.block = list(c(Side.r, Side.l), day)
```

or

```
..., random = ~ Side + day - 1,
re.block = list(c("Sider","Sidel"), "day")
```

can be used equivalently in the above assignment.

The `re.structure` argument is a vector of character strings with length equal to the number of random-effects blocks defined in `re.block`. When the same structure applies to all random-effects blocks, one specification suffices. Each component of `re.structure` defines the covariance structure to be used for the corresponding block of random effects in the `re.block` list. Partial matching is used on this argument, so that only the first letter of the structure name is required; for example, `re.structure = c("c","u")` would have the same effect in the assignment to `Pixel.fit` above.

The `print` method changes slightly when random-effects blocks are used.

```
> Pixel.fit
Call:
  Fixed: pixel ~ day + day^2
 Random:  ~ Side + day - 1
Cluster:  ~ Dog
   Data: Pixel

Variance/Covariance Components Estimate(s):

 Block: 1
  Structure: compound symmetry
  Standard Deviation(s) of Random Effect(s)
    Sidel    Sider
 31.78052 31.78052
 Correlation of Random Effects
         Sidel
Sider 0.7204526
```

```
 Block: 2
  Structure: unstructured
  Parametrization: matrixlog
  Standard Deviation(s) of Random Effect(s)
      day
 1.782121

 Cluster Residual Variance: 81.46076

Fixed Effects Estimate(s):
  (Intercept)      day   I(day^2)
    1072.681 6.224556 -0.3685574

Number of Observations: 102
Number of Clusters: 10
```

Custom Methods for the Structured Covariance Matrix

Users can define their own covariance structures if desired. The `lme` function uses two generic functions, `lme.re.param` and `lme.re.factor`, when estimating the covariance components for the random effects. The `lme.re.param` function should return a vector of parameters that define the parametrizations of a given covariance matrix D. The `lme.re.factor` function, given a vector of parameters, should return a matrix L such that $L^T L = D$. The structure defines a class for which methods for the generic functions `lme.re.param` and `lme.re.factor` are given. By writing customized methods for these generic functions, you can define special covariance structures.

When writing methods for `lme.re.param` and `lme.re.factor`, observe that the optimization algorithm in `lme` assumes an unrestricted parametrization of the covariance matrices. The following example illustrates this issue. Suppose that b_{0ij} and b_{1i} are not independent as in model (11.3), but rather have a common, possibly nonzero, correlation. This structure is not among the available options, so we would write custom methods for `lme.re.param` and `lme.re.factor`. Let D represent the covariance matrix of $\boldsymbol{b}_i$. Assume that $D_{11} = D_{22}$ and $D_{13} = D_{23}$. There are a total of four parameters in D, and D becomes

$$D = \begin{bmatrix} D_{11} & D_{12} & D_{13} \\ D_{12} & D_{11} & D_{13} \\ D_{13} & D_{13} & D_{33} \end{bmatrix}. \tag{11.6}$$

Note that the positive definiteness of D requires both that $|D_{12}| < D_{11}$ and that $|D_{13}| < \sqrt{D_{11}D_{33}}$. These restrictions can be incorporated, in an unconstrained framework, by defining

$$\begin{aligned}
\theta_1 &= log(D_{11}) \\
\theta_2 &= log[(D_{11} + D_{12})/(D_{11} - D_{12})] \\
\theta_3 &= log(D_{33}) \\
\theta_4 &= log[(\sqrt{D_{11}D_{13}} + D_{13})/(\sqrt{D_{11}D_{13}} - D_{13})]
\end{aligned}$$

and setting

$$\begin{aligned}
D_{11} &= D_{22} = e^{\theta_1} \\
D_{12} &= e^{\theta_1}(e^{\theta_2} - 1)/(e^{\theta_2} + 1) \\
D_{33} &= e^{\theta_3} \\
D_{13} &= D_{23} = (e^{[(\theta_1 + \theta_3)/2]}(e^{\theta_4} - 1)/(e^{\theta_4} + 1)).
\end{aligned}$$

Customized methods using this structure can be defined as follows.

```
lme.re.param.mystruct <- function(D)
{
ax1 <- log(D[1,1])
ax2 <- log(D[3,3])
ax3 <- sqrt(D[1,1] * D[3,3])
c(ax1, log((D[1,1] + D[1,2])/(D[1,1] - D[1,2])),
  ax2, log((D[1,3] + ax3)/(ax3 - D[1,3])))
}
lme.re.factor.mystruct <- function(theta, q)
{
ax1 <- exp(theta[1])
ax2 <- ax1 * (exp(theta[2]) - 1)/(exp(theta[2]) + 1)
ax3 <- exp(theta[3])
ax4 <- sqrt(ax1 * ax3) * (exp(theta[4]) - 1)/
          (exp(theta[4]) + 1)
chol(array(c(ax1, ax2, ax4, ax2, ax1, ax4, ax4, ax4, ax3),
          c(3,3)))
}
```

The following call uses them to fit the desired model.

```
> Pixel.fit2 <- lme(fixed = pixel ~ day + day^2,
+                   random = ~ Side + day - 1,
+                   cluster = ~ Dog, data = Pixel,
+                   re.structure = "mystruct")
```

Here is the resulting fit.

```
> Pixel.fit2
Call:
  Fixed: pixel ~ day + day^2
 Random:  ~ Side + day - 1
Cluster:  ~ Dog
   Data: Pixel

Variance/Covariance Components Estimate(s):

  Structure: mystruct
  Standard Deviation(s) of Random Effect(s)
    Sidel    Sider      day
 33.00453 33.00453 1.842898
 Correlation of Random Effects
           Sidel      Sider
Sider  0.7403167
  day -0.4781016 -0.4781016

 Cluster Residual Variance: 80.91684

Fixed Effects Estimate(s):
  (Intercept)      day   I(day^2)
    1073.343 6.128869 -0.3673046

Number of Observations: 102
Number of Clusters: 10
```

One can use the `anova` method to compare `Pixel.fit` and `Pixel.fit2`. The comparison shows that the assumption of independence between (b_{0i1},b_{0i2}) and b_{1i} is not unreasonable in this case; the p-value is 0.13.

```
> anova(Pixel.fit, Pixel.fit2)
```

```
. . .
           Model Df    AIC    BIC  Loglik    Test
 Pixel.fit     1  7 841.54 859.91 -413.77
Pixel.fit2     2  8 841.29 862.29 -412.65  1 vs. 2
           Lik.Ratio P value
 Pixel.fit
Pixel.fit2    2.2448 0.13406
```

The Structured Covariance Matrix for Within-Cluster Errors

The optional arguments `serial.structure`, `serial.covariate`, `serial.covariate.transformation`, `var.function`, `var.estimate` and `var.covariate` can be used to specify the covariance structure for within-cluster errors. The first three arguments correspond to modeling the correlation structure. The last three arguments correspond to modeling the variance structure. They are applied to the `Ovary` data below.

For the `Ovary` data, you can apply the `lme` calls described in previous sections to model as follows.

```
> Ovary.fit1 <- lme(follicles ~
+                   I(sin(2*pi*Time))+I(cos(2*pi*Time)),
+                   cluster= ~ Mare, data = Ovary)
> Ovary.fit1
Call:
  Fixed: follicles ~ I(sin(2 * pi * Time)) + I(cos(2 * pi *
Time))
 Random: follicles ~ I(sin(2 * pi * Time)) + I(cos(2 * pi *
Time))
Cluster:  ~ Mare
   Data: Ovary

Variance/Covariance Components Estimate(s):

  Structure: unstructured
  Parametrization: matrixlog
  Standard Deviation(s) of Random Effect(s)
 (Intercept) I(sin(2 * pi * Time)) I(cos(2 * pi * Time))
    3.235347             2.095777             1.068676
 Correlation of Random Effects
                     (Intercept) I(sin(2 * pi * Time))
I(sin(2 * pi * Time)) -0.5699137
I(cos(2 * pi * Time)) -0.8011913   0.1786379
```

```
 Cluster Residual Variance: 9.114855

Fixed Effects Estimate(s):
  (Intercept) I(sin(2 * pi * Time)) I(cos(2 * pi * Time))
    12.18591             -3.296668            -0.8731739

Number of Observations: 308
Number of Clusters: 11
```

Model (11.5) assumes that measurements within each cluster are independent. From figure 11.4, it is reasonable to assume the existence of serial correlations. Assume that the correlation matrix is of type AR(1) and that the variances are the same at all occasions. Two types of AR(1) are available for the argument `serial.structure`. They are `ar1` and `ar1.continuous`, corresponding to an integer-valued or a continuous-valued covariate, respectively. In the continuous case, the arguments are set as follows.

```
serial.structure = "ar1.continuous",
serial.covariate = ~ Time,
serial.covariate.transformation = "none"
```

For simplicity, the discrete case `ar1` is applied to the analysis of the `Ovary` data. Hence, for the ith mare, with 3 repeated measures at times x_{ij}, the covariance matrix is $\sigma^2\Lambda_i$, where

$$\Lambda_i = \begin{bmatrix} 1 & \rho_{12} & \rho_{13} \\ \rho_{12} & 1 & \rho_{23} \\ \rho_{13} & \rho_{23} & 1 \end{bmatrix}, \quad \rho_{uv} = \alpha^{d_{uv}}, \quad d_{uv} = |u - v| \quad . \tag{11.7}$$

The following call fits this model.

```
> Ovary.fit2 <- lme(follicles ~
+                   I(sin(2*pi*Time)) + I(cos(2*pi*Time)),
+                   serial.structure="ar1",
+                   cluster= ~ Mare, data = Ovary)
```

```
> Ovary.fit2
Call:
  Fixed: follicles ~ I(sin(2 * pi * Time)) + I(cos(2 * pi *
Time))
 Random: follicles ~ I(sin(2 * pi * Time)) + I(cos(2 * pi *
Time))
Cluster:  ~ Mare
   Data: Ovary

Variance/Covariance Components Estimate(s):

  Structure: unstructured
  Parametrization: matrixlog
  Standard Deviation(s) of Random Effect(s)
 (Intercept) I(sin(2 * pi * Time))
    3.091169             1.408717
 I(cos(2 * pi * Time))
            0.8221724
 Correlation of Random Effects
                      (Intercept)
I(sin(2 * pi * Time)) -0.7399842
I(cos(2 * pi * Time)) -0.9734431
                      I(sin(2 * pi * Time))
I(sin(2 * pi * Time))
I(cos(2 * pi * Time))  0.5663492

 Cluster Residual Variance: 11.46478

 Serial Correlation Structure: ar1
 Serial Correlation Parameter(s): 0.5408387

Fixed Effects Estimate(s):
  (Intercept) I(sin(2 * pi * Time))
    12.18361             -3.01566
 I(cos(2 * pi * Time))
           -0.8703521

Number of Observations: 308
Number of Clusters: 11
```

The above fit gives an estimate of 0.54 for α in the covariance matrix. The estimated correlation between two consecutive measures is 0.54. A likelihood ratio test of these assumptions shows that correlations exist within cluster.

```
> anova(Ovary.fit1,Ovary.fit2)

. . .
       Model Df    AIC    BIC  Loglik    Test Lik.Ratio    P
value
Ovary.fit1 1 10 1630.0 1667.3 -805.02
Ovary.fit2 2 11 1566.1 1607.1 -772.05 1 vs. 2     65.94
4.4409e-16
```

11.2 NONLINEAR MIXED-EFFECTS MODELS

Nonlinear mixed-effects models, which generalize nonlinear models and linear mixed-effects models can be analyzed with the new S-PLUS function `nlme`. Since the `nlme` class inherits from the `lme` class, methods for the `lme` class apply to the `nlme` class.

Structured covariance matrices and random-effects blocks can also be used for nonlinear mixed-effects models. As in the `lme` function, the optional arguments `re.structure` and `re.block` provide this capability in `nlme`. The usage of these arguments is identical to that in `lme`, described in the previous section.

There are many advantages to using nonlinear mixed-effects models. For example, the expectation function is usually based on the theory about the mechanism of the data. Nonlinear models are more flexible than linear models. Further, parameters in the expectation function usually have physical meaning and are of interest to the investigator. The three data analyses below demonstrate the use of `nlme` in a range of applications of nonlinear mixed-effects models.

Example: Soybean Data

These data come from an experiment to compare growth patterns of two genotypes of soybean as described in Davidian and Giltinan (1995). One genotype is a commercial variety, Forrest (F), and the other is an experimental strain, Plant Introduction #416937 (P). The data were collected in the three years from 1988 to 1990. At the beginning of the growing season in each year, 16 plots were planted with seeds; 8 plots with each genotype. Each plot was sampled eight to ten times at approximately weekly intervals. At each sampling time, six plants were randomly selected from each plot, leaves from these plants were weighed, and the average leaf weight per plant was calculated for the plot. Different plots in different sites were used in different years. The data are stored in the data frame `Soybean` shown below.

```
> Soybean
    Plot Variety Year time    weight
1      1       F 1988   14   0.10600
2      1       F 1988   21   0.26100
3      1       F 1988   28   0.66600
. . .
410   48       P 1990   51   6.131667
411   48       P 1990   64  16.411667
412   48       P 1990   79  16.946667
```

Exploratory Data Analysis

The experiment in the soybean study has a 2×3 block design with repeated measures at different measurement times in each year. The 412 total records are grouped into 48 clusters by `Plot`. The objective is to model the growth pattern in terms of average leaf weight. From the plot below and as suggested by Davidian and Giltinan (1995), a logistic function is appropriate.

```
> trellis.device(motif)
> xyplot(weight ~ time | Variety * Year, data = Soybean,
+        panel = function(x, y) {
+                panel.grid()
+                panel.xyplot(x,y)
+                panel.loess(x,y) },
+        xlab = "Days after planting",
+        ylab = "Leaf weight/plant (g)")
```

A logistic model can be written as

$$w_{ij} = \frac{\alpha_i}{1 + e^{-(x_{ij} - \beta_i)/\gamma_i}} + \varepsilon_{ij} \tag{11.8}$$

where w_{ij} is the average leaf weight for plot i at time j. The parameters in the model have special interpretations; α_i is the asymptotic leaf weight as `time`

Figure 11.8: *The objective of the soya bean study is to model growth patterns.*

goes to infinity, β_i is the length of time to reach 50 percent of the asymptotic leaf weight, α_i, and γ_i is a scale parameter. The ε_i are assumed to be independent and identically distributed as $N(\mathbf{0}, \sigma^2 D)$. Without assuming random effects and assuming $D=I$, a nonlinear model can be applied to the above model.

To explore whether the data follow a nonlinear mixed-effects model, we can apply the nonlinear model to each cluster and then plot the fitted parameters. The new S-PLUS function `nlsList` creates a list of fits to each cluster. This function is an extension of the `nls` function. To call `nlsList`, the user must provide either initial estimates or a self-starting function, which will be discussed later in this chapter. Here the same initial estimates of α, β and γ will be used for all clusters. They are derived based on an approximation for α and a simple regression. The mean of the last recorded leaf weights in each plot is used as the initial estimate of α. The initial estimates of β and γ are derived from fitting a simple regression on $og(\alpha/\text{weight} - 1)$. They are generated as follows.

```
> Soybean.new <- Soybean
> Soybean.new$weight.transf <- numeric(412)
> attach(Soybean.new)
> maxwt <- numeric(48)
> for (i in unique(Plot)) {
+ maxwt[i] <- max(weight[Plot == i])}
> mean(maxwt)
[1] 18.22999
> for (i in 1:length(Soybean.new$weight)) {
+   if(Soybean.new$weight[i] < 18)
+     Soybean.new$weight.transf[i] <-
+           log(18.22999/Soybean.new$weight[i]-1)
+   else Soybean.new$weight.transf[i] <- log(.01)}
> Soybean.lm <- lm(weight.transf ~ time, data = Soybean.new)
> Soybean.lm
Call:
lm(formula = weight.transf ~ time, data = Soybean.new)

Coefficients:
 (Intercept)       time
    6.906195 -0.1301867
```

```
Degrees of freedom: 412 total; 410 residual
Residual standard error: 1.002497
```

The initial estimates of $\beta = 53.04839$ and $\gamma = 7.681276$ are computed from $\beta/\gamma = 6.906195$ and $-1/\gamma = -0.1301867$.

The function `nlsList` is called as follows below, where `Asym`=α, `T50`=β, and `scal`=γ. The initial estimates are added to the data frame first and then passed to the `nlsList` call. S-PLUS functions `coef` and `pairs` are used to print and display the results.

```
> param(Soybean.new, "Asym") <- 18.22999
> param(Soybean.new, "T50")  <- 53.04839
> param(Soybean.new, "scal") <- 7.681276
> Soybean.nlsList <- nlsList(weight ~
+                   Asym/(1 + exp((( -(time - T50))/scal))),
+                     data = Soybean.new, cluster = ~ Plot)
Error in nls(formula = formula, data = data, contr..:
step factor reduced below minimum
Dumped
> coef(Soybean.nlsList)
          Asym       T50      scal
 1  20.338845  57.40303  9.605089
 2  19.745594  56.57531  8.406848
. . .
13  17.703854  51.27163  6.809334
14 162.850268 104.85256 17.922592
15  27.485299  61.49776 10.177363
. . .
31  15.472057  46.34357  5.394270
32         NA        NA        NA
33  19.788159  55.68886  9.615056
. . .
45  19.545968  51.15219  7.294325
46  17.688635  50.22827  6.625463
47  19.159893  54.80059 10.846798
48  18.51452   52.44986  8.582159
> pairs(Soybean.nlsList)
```

The error message above refers to cluster 32 which was not fitted.

Figure 11.9 shows the parameter estimates as displayed by the `pairs` function. With the exception of plot 14, the estimates for `Asym` range from 8 to 36, those for `T50` from 46 to 70, and those for `scal` from 6 to 13. From

Figure 11.9: *The parameter estimates displayed by the pairs function.*

the above analysis, we see that plot 32 cannot be fitted by the logistic model with the above initial estimates. Plot 14 has distinctly larger parameter estimates than the other clusters. Figure 11.10 depicts leaf weights of plots

Figure 11.10: *Leaf weights of the selected plots 14, 32, 46 and 48.*

14, 32, 46, and 48. Much variation is discerned in the last three weeks. The above analysis suggests the existence of random effects in the parameters and within-cluster covariance. Hence, a nonlinear mixed-effects model is appealing.

To see the association between random-effects parameters and covariates, we can plot the above fitted parameters versus covariates. Figure 11.11 is the density plot of the estimates for the asymptotic leaf weight `Asym`, excluding the problematic plots 14 and 32. It shows that the estimated values of `Asym` in 1989 are, in general, smaller than those in other years. Plants of the experimental strain have smaller fitted values in 1989 and 1988. This plot suggests that the random-effects parameter `Asym` is affected by `Variety` and `Year`. No patterns are shown in the plots of the estimates for `T50` and `scal`.

```
> # Find the invariants in clusters
> Soybean.inv <- Soybean[match(unique(Soybean$Plot),
+                         Soybean$Plot), 2:3]
> augcoef <- cbind(coef(Soybean.nlsList), Soybean.inv)
> Soybean.nlscoef <- augcoef[c(1:13,15:31,33:48),]
> densityplot( ~ Asym | Variety*Year, data =Soybean.nlscoef,
+              layout = c(1,6), aspect = 0.4)
```

Example: Theoph Data

These data come from a study of the kinetics of the anti-asthmatic agent theophylline reported by Boeckman, Sheiner, and Beal (1992). The objective of the study was to model the kinetics of theophylline following oral administration.

In this experiment, the drug was administered orally to twelve subjects, and then serum concentrations were measured 11 times over the next 25 hours. The data are stored in the data frame `Theoph` as shown below. The column `Wt` gives the subject's weight (kg) and `conc` is the theophylline concentration in the sample (mg/L). The 132 total records are grouped into 12 clusters by `Subject`.

```
> Theoph
    Subject   Wt Dose  time conc
1         1 79.6 4.02  0.00 0.74
2         1 79.6 4.02  0.25 2.84
3         1 79.6 4.02  0.57 6.57
. . .
130      12 60.5 5.30  9.03 6.11
131      12 60.5 5.30 12.05 4.57
132      12 60.5 5.30 24.15 1.17
```

Figure 11.12 shows that the concentration has a peak between one to four

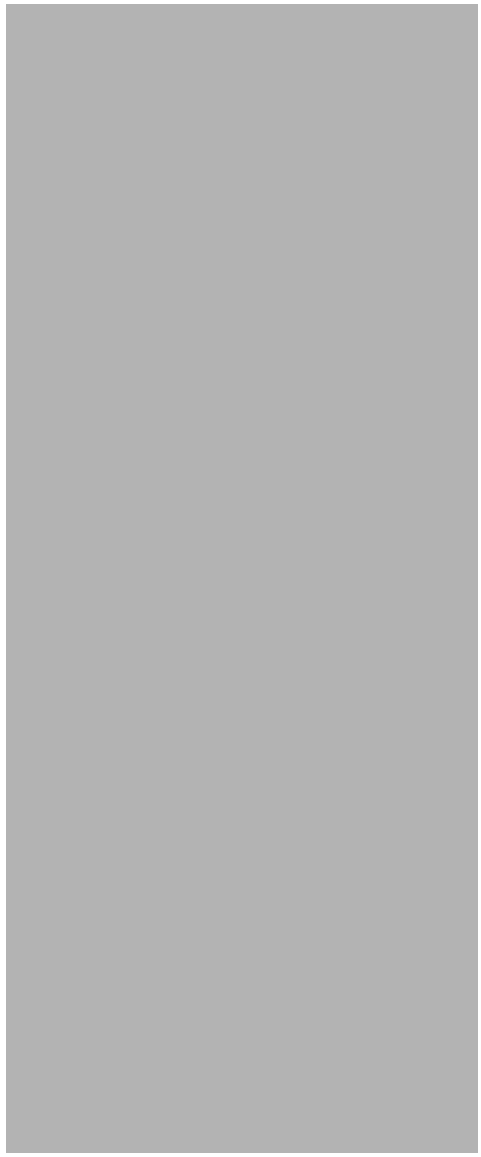

Figure 11.11: *The density plot for the estimates of the asymptotic leaf weight.*

hours following administration.

```
> trellis.device(motif)
> Theoph.plot()
```

Figure 11.12: *The plot shows there is a peak one to four hours after administration.*

A common pharmacokinetic model is the one-compartment open model with first-order absorption and elimination (Davidian and Giltinan, 1995). The model can be written as

Here c_{ij} denotes theophylline concentration for subject i at time t_{ij} (hours) and d_i is the dose administered to the subject. The parametrization used here allows for more stable convergence and also enforces nonnegative estimates. The parameters have special meaning in pharmacokinetics. The expectation function is modeled in terms of the clearance, , the absorption rate

constant, $e^{\phi_{2i}}$, and the elimination rate constant, $e^{\phi_{3i}}$. The error terms e_{ij} are assumed to be independent and identically distributed as $N(0,\sigma^2)$.

Example: CO2 Data

These data, described in Potvin and Lechowicz (1990), come from a study of the cold tolerance of a C_4 grass species, *Echinochloa crus-galli*. A total of twelve four-week-old plants, six from Québec and six from Mississippi, were divided into two groups. Control plants were kept at 26° C, and other plants were subjected to 14 hours of chilling at 7° C. After 10 hours of recovery at 20° C, CO_2 uptake rates (in $\mu mol/m^2 s$) were measured for each plant at seven concentrations (100, 175, 250, 350, 500, 675, 1000 μL/L) of ambient CO_2. Each plant was subjected to the seven concentrations of CO_2 in increasing order. The objective of the experiment was to evaluate the effect of plant type and chilling treatment on the CO_2 uptake. The CO_2 uptake data is held in a data frame called `CO2`, with columns `Plant`, `Type`, `Treatment`, `conc`, and `uptake` as shown below.

```
> CO2
   Plant         Type  Treatment conc uptake
 1     1       Quebec nonchilled   95   16.0
 2     1       Quebec nonchilled  175   30.4
 3     1       Quebec nonchilled  250   34.8
 . . .
83    12 Mississippi    chilled  675   18.9
84    12 Mississippi    chilled 1000   19.9
```

This is an example with repeated measures not on a time-dependent variable. The experiment has a 2×2 factorial design with repeated measures at seven levels of concentration. The 84 total records are grouped into 12 clusters by `Plant`. The data are plotted in figure 11.13.

The model used in Potvin and Lechowicz (1990) is as follows.

$$U_{ij} = \phi_{1i}[1 - e^{-\phi_{2i}(C_j - \phi_{3i})}] + \varepsilon_{ij} \qquad (11.10)$$

Here U_{ij} denotes the CO_2 uptake rate of the ith plant at the jth CO_2 ambient concentration; ϕ_{1i}, ϕ_{2i}, and ϕ_{3i} denote respectively the asymptotic uptake rate, the uptake growth rate, and the maximum ambient CO_2 concentration at which no uptake is verified for the ith plant; C_j denotes the jth ambient CO_2 level; and the ε_{ij} are independent and identically distributed error terms with distribution $N(0,\sigma^2)$.

Figure 11.13: *The CO2 experiment was designed to measure the cold tolerance of certain plants.*

The nlme Class and Related Methods

This section demonstrates the basic call to `nlme`. The five arguments `object`, `fixed`, `random`, `cluster`, and `start` are described in detail. Except for the `predict` method, all methods for the `lme` class are used for the `nlme` class.

The nlme Function

The `nlme` function is used to fit nonlinear mixed-effects models, as defined in Lindstrom and Bates (1990), using either maximum likelihood or restricted maximum likelihood. Many arguments can be used with this function, though a typical call may be as brief as that below. Only `object`, `fixed`, `cluster`, and `start` are required arguments.

```
> nlme(object, fixed, random, cluster, data, start)
```

The required argument `object` consists of a formula specifying the nonlinear model to be fitted. Any S-PLUS nonlinear formula can be used, which gives the function considerable flexibility. Applying equation (11.10), and without incorporating covariates, the parameters become ϕ_1, ϕ_2, ϕ_3 for all plants. The `object` argument would be set to

```
uptake ~ A * (1 - exp(-B * (conc - C)))
```

where $A = \phi_1$, $B = \phi_2$, and $C = \phi_3$.

Alternatively, one can set the `object` argument to a function, say `CO2.fun`,

```
> CO2.fun <- function(conc, A, B, C)
+             A * (1 - exp(-B * (conc-C)))
```

and then set `object` to the following.

```
uptake ~ CO2.fun(conc, A, B, C)
```

The arguments `fixed` and `random` are lists of formulas that define the structures of the fixed and random effects in the model. In these formulas a period "`.`" on the right-hand side of a formula indicates that a single parameter is associated with the effect, but any linear formula in S-PLUS can be used. Again, this gives considerable flexibility to the model. Time-dependent parameters are easily incorporated, for example, when a formula in the `fixed` list involves a covariate that changes with time. Usually every parameter in the model has an associated fixed effect, but it may not have an associated random effect. Since we assumed that all random effects have mean zero, the inclusion of a random effect without a corresponding fixed effect would be unusual. Note that the `fixed` and `random` formulas could be directly incorporated in the model declaration.

To fit a model to the CO_2 uptake data in which all parameters are random and no covariates are included, use the following.

```
fixed = list(A ~ ., B ~ ., C ~ .),
random = list(A ~ ., B ~ ., C ~ .)
```

To estimate the effects of plant type and chilling treatment on the parameters in the model, use the following.

```
fixed = list(A ~ type*treatment, B ~ type*treatment,
             C ~ type*treatment),
random = list(A ~ ., B ~ ., C ~ .)
```

The `cluster` argument is required and defines the cluster label of each observation. Any S-PLUS expression or a formula with no left-hand side can be used here. The optional argument `data` names a data frame and `start` provides a list of starting values for the iterative algorithm. Only the fixed-effects starting estimates are required. The default starting estimates for the random effects are zero. Starting estimates for the scaled covariance matrix D of the random effects and the cluster variance σ^2 are automatically generated using a formula from Laird, Lange, and Stram (1987) if they are not supplied.

Here is a simple call to `nlme` to fit model (11.10) without any covariates and with all parameters as mixed effects. The initial values for the fixed effects were obtained from Potvin and Lechowicz (1990).

```
> CO2.fit1 <-
+   nlme(object = uptake ~ CO2.fun(conc, A, B, C),
+        fixed = list(A ~ ., B ~ ., C ~ .),
+        random = list(A ~ ., B ~ ., C ~ .),
+        cluster = ~ Plant, data = CO2,
+        start = list(fixed = c(30, 0.01, 50)))
```

Since no derivatives are passed to `object` in the above call, numerical derivatives are used in the optimization. An alternative approach is to pass derivatives as the `gradient` attribute of the value returned by `CO2.fun` and to use this in the optimization algorithm. The S-PLUS function `deriv` can be used to create expressions for the derivatives which can then be used by the same `nlme` call.

```
> CO2.fun <- deriv(~ A * (1 - exp(-B * (conc - C))),
+           LETTERS[1:3], function(conc, A, B, C){})
```

For the theophylline data set, assume that all parameters have random effects and no covariates are considered. Here is the call to model (11.9), where `lCl` $= \phi_1$, `lka` $= \phi_2$, and `lke` $= \phi_3$. The initial estimates are derived from a preliminary analysis of the data.

```
> Theoph.func <- deriv( ~ Dose * exp(lke) * exp(lka) *
+    (exp(-exp(lke) * time) - exp(-exp(lka) * time)) /
+    (exp(lCl) * (exp(lka) - exp(lke))),
+    c("lCl", "lka", "lke"),
+    function(Dose, time, lCl, lka, lke){})
> Theoph.fit1 <- nlme(conc ~
+    Theoph.func(Dose, time, lCl, lka, lke),
+    fixed = list(lCl ~ ., lka ~ ., lke ~ .),
+    random = list(lCl ~ ., lka ~ ., lke ~ .),
+    cluster = ~ Subject, data = Theoph,
+    start = list(fixed = c(-2.73, 1.6, -2.3)))
```

Methods for nlme Objects

Objects returned by the `nlme` function are of class `nlme` which inherits from `lme`. All methods described in the sections on `lme` are available for the `nlme` class. In fact, with the exception of the `predict` method, all methods are common to both classes.

The `print` method provides a brief description of the estimation results. It gives estimates of the standard errors and correlations of the random effects, of the cluster variance, and of the fixed effects.

```
> Theoph.fit1
Call:
  Model: conc ~ Theoph.func(Dose, time, lCl, lka, lke)
  Fixed: list(lCl ~ ., lka ~ ., lke ~ .)
 Random: list(lCl ~ ., lka ~ ., lke ~ .)
Cluster:  ~ Subject
   Data: Theoph

Variance/Covariance Components Estimate(s):

  Structure: matrixlog
  Standard Deviation(s) of Random Effect(s)
       lCl       lka       lke
 0.2499743 0.6396112 0.1285968
 Correlation of Random Effects
           lCl        lka
lka 0.05173928
lke 0.99480761 0.12677231

 Cluster Residual Variance: 0.4655799

Fixed Effects Estimate(s):
        lCl       lka       lke
 -3.213992 0.4511411 -2.431956

Number of Observations: 132
Number of Clusters: 12
> CO2.fit1
Call:
  Model: uptake ~ CO2.func(conc, A, B, C)
  Fixed: list(A ~ ., B ~ ., C ~ .)
 Random: list(A ~ ., B ~ ., C ~ .)
Cluster:  ~ Plant
   Data: CO2

Variance/Covariance Components Estimate(s):

  Structure: matrixlog
  Standard Deviation(s) of Random Effect(s)
         A          B       C
 9.519298 0.00115931 11.1271
```

```
 Correlation of Random Effects
           A           B
B -0.09761193
C  0.99956629 -0.09774343

 Cluster Residual Variance: 3.128036

Fixed Effects Estimate(s):
         A           B        C
 32.54721 0.009466799 41.81611

Number of Observations: 84
Number of Clusters: 12
```

In the above fits, some random-effects parameters are strongly correlated and some correlations are close to zero.

In `CO2.fit1`, there is a very strong correlation between the $\phi_1(A)$ and the $\phi_3(C)$ random effects and these are almost uncorrelated with the $\phi_2(B)$ random

Figure 11.14: *The scatter plot matrix with random effects.*

effect. The scatter plot matrix of the random effects obtained using the `plot` function

```
> plot(CO2.fit1, pch = "o")
```

is shown in figure 11.14. It is clear that the ϕ_1 and ϕ_3 random effects are virtually identical. This correlation may be due to the fact that the plant type and the chilling treatment, which were not included in the `CO2.fit1` model, are affecting ϕ_1 and ϕ_3 in the same way.

The powerful integration of analytical and graphical machinery in the S-PLUS environment is shown in the following analysis of the dependence of the individual parameters ϕ_{1i}, ϕ_{2i}, and ϕ_{3i} in model (11.10) on plant type and chilling factor.

First, save the conditional modes of the random effects obtained in the first fit in the data frame `CO2.random`.

```
> CO2.random <- data.frame(CO2.fit1$coef$random)
```

Next, add a column to `CO2.random` with the treatment combinations corresponding to each plant.

```
> CO2.random$type.trt <- as.factor(rep(
+      c("Quebec nonchilled", "Quebec chilled",
+        "Mississippi nonchilled", "Mississippi chilled"),
+      rep(3,4)))
```

Finally, plot the conditional modes of the random effects versus the treatment combinations. The plots are shown in figure 11.15.

```
> plot(A ~ type.trt, data = CO2.random)
> plot(B ~ type.trt, data = CO2.random)
> plot(C ~ type.trt, data = CO2.random)
```

These plots indicate that chilled plants tend to have smaller values of ϕ_1 and ϕ_3, and that the Mississippi plants are much more affected than the Québec plants, which suggests an interaction effect between plant type and chilling treatment. There is no clear pattern of dependence between ϕ_2 and the treatment factors, which suggests that this parameter is not significantly affected either by plant type or by chilling treatment.

The above analysis suggests an alternative model with ϕ_{1i}, ϕ_{2i}, and ϕ_{3i} in model (11.10) reparametrized as follows.
Assuming further that only the intercepts contain random effects, the new model is fitted below, where A=ϕ_{1i}, B=ϕ_{2i}, and C=ϕ_{3i}.

Figure 11.15: *The conditioning modes of the random effects are plotted against the treatment combinations.*

$$\begin{aligned}
\phi_{1i} &= A_0 + A_1 \cdot \text{Type}_i + A_2 \cdot \text{Treatment}_i + A_3 \cdot \text{Type}_i \times \text{Treatment}_i \\
\phi_{2i} &= B \\
\phi_{3i} &= C_0 + C_1 \cdot \text{Type}_i + C_2 \cdot \text{Treatment}_i + C_3 \cdot \text{Type}_i \times \text{Treatment}_i
\end{aligned} \tag{11.11}$$

```
> CO2.fit2 <- nlme(object = uptake ~
+                   CO2.fun(conc, A, B, C),
+                   fixed = list(A ~ Type*Treatment, B ~ .,
+                                C ~ Type*Treatment),
+                   random = list(A ~ ., B ~ ., C ~ .),
+                            cluster = ~ Plant,
+                   data = CO2, start = list(fixed =
+                   c(30, 0, 0, 0, 0.01, 50, 0, 0, 0)))
```

The `summary` method provides detailed information on the new fitted object.

```
> summary(CO2.fit2)

 . . .
Convergence at iteration: 7
Approximate Loglikelihood: -180.7
AIC: 393.4
BIC: 432.3

Variance/Covariance Components Estimate(s):
  Structure: matrixlog
  Standard Deviation(s) of Random Effect(s)
 A.(Intercept)         B C.(Intercept)
          2.28 0.0003264         5.892
 Correlation of Random Effects
              A.(Intercept)        B
            B -0.05615
C.(Intercept)  0.99992      -0.05616

 Cluster Residual Variance: 3.126

Fixed Effects Estimate(s):
                        Value Approx. Std.Error z ratio(C)
   A.(Intercept)  32.451794         0.7236227     44.846
         A.Type  -7.911473         0.7023989    -11.264
     A.Treatment  -4.235456         0.7008491     -6.043
A.Type:Treatment  -2.428173         0.7008658     -3.465
               B   0.009548         0.0005912     16.152
   C.(Intercept)  39.953318         5.6528809      7.068
         C.Type -10.477934         4.2224849     -2.481
     C.Treatment  -7.993575         4.2013654     -1.903
C.Type:Treatment -12.340465         4.2300868     -2.917
 . . .
```

The correlation between the ϕ_1 and the ϕ_3 random effects remains very high, which indicates that the model is probably overparametrized and that fewer random effects are needed. The analysis here does not pursue the model building for the CO_2 uptake data. The goal is just to illustrate the use of the methods for the `nlme` class.

Use `anova` to compare `CO2.fit1` and `CO2.fit2`.

```
> anova(CO2.fit1, CO2.fit2)
```

```
  . . .
         Model Df    AIC    BIC Loglik    Test Lik.Ratio
CO2.fit1     1 10 422.81 447.12 -201.4
CO2.fit2     2 16 393.39 432.28 -180.7 1 vs. 2    41.416
           P value
CO2.fit1
CO2.fit2 2.3975e-07
```

The inclusion of plant type and chilling treatment in the model causes a substantial increase in the log likelihood, indicating that they have a significant effect on ϕ_{1i} and ϕ_{3i}.

Diagnostic plots can be obtained by setting `option="r"` in the call to `plot`.

```
> par(mfrow = c(2,2))
> plot(CO2.fit2, option = "r", pch = "o")
```

The corresponding plots are shown in figure 11.16. The plot of observed versus fitted values indicates that the model fits the data well. Most points lie close to the 45° line. The plot of residuals versus fitted values does not indicate any departures from the assumptions in the model. No outliers are apparent and the residuals are symmetrically scattered around the zero residual line, with similar spread for different levels of the fitted values.

Predictions are returned by the `predict` function. For example, to obtain the population predictions of CO_2 uptake rate for Québec and Mississippi plants under chilling and no chilling, at ambient CO_2 concentrations of 50, 100, 200, and 500 $\mu L/L$, first define

```
> CO2.new <- data.frame(Type = rep(c("Quebec",
+                     "Mississippi"), c(8,8)),
+                     Treatment = rep(rep(c("chilled",
+                     "nonchilled"), c(4,4)), 2),
+                     conc = rep(c(50, 100, 200, 500), 4))
```

and then use the following to obtain the predictions.

```
> predict(CO2.fit2, CO2.new)
```

Figure 11.16: *The plot of observed versus fitted values, top right, indicates a good fit.*

```
    population
1   0.05672987
2  11.88074442
. . .
15 28.92023612
16 38.00663973
```

The `predict` function can also be used for plotting smooth fitted curves by calculating fitted values at closely spaced concentrations. Figure 11.17 presents the individual fitted curves for all twelve plants evaluated at 200 concentrations between 50 and 1000 $\mu L/L$. The code used to produce these plots is shown below.

```
> attach(CO2)
> plot(conc, uptake, type="n", xlab="Ambient CO2",
+      ylab="CO2 uptake rate", ylim=c(-15,48))
> pch <- 0
> levs <- unique(Plant)
> n <- length(levs)
```

Figure 11.17: *The individual fitted curves for all 12 plants.*

```
> for (i in levs) {
+     if (Type[match(i, Plant)]=="Quebec" &
+         Treatment[match(i, Plant)]=="nonchilled") pch <- 1
+     if (Type[match(i, Plant)]=="Quebec" &
+         Treatment[match(i, Plant)]=="chilled") pch <- 2
+     if (Type[match(i, Plant)]=="Mississippi" &
+        Treatment[match(i, Plant)]=="nonchilled") pch <- 16
+     if (Type[match(i, Plant)]=="Mississippi" &
+         Treatment[match(i, Plant)]=="chilled") pch <- 17
+     points(conc[Plant==i], uptake[Plant==i],
+          type="p", pch=pch)
+ }
> legend(400,0,marks=c(1,2,16,17),bty="n",
+    legend=c("Quebec Control","Quebec Chilled",
+             "Mississippi Control","Mississippi Chilled"))
> CO2.new2 <- data.frame(Plant = rep(levs, 200),
+     Treatment = rep(Treatment[match(levs, Plant)], 200),
+          Type = rep(Type[match(levs, Plant)], 200),
+          conc = rep(seq(50,1000,length=200),rep(n,200)))
> mypredict <- predict(CO2.fit2, CO2.new2,
```

```
+                        cluster = ~ Plant)
> for (i in levs) lines(CO2.new2[n*(0:199)+i,"conc"],
+           mypredict$fit$cluster[n*(0:199)+i], lty=i)
```

Self-Starting Functions

In applying `nlsList` and `nlme`, initial estimates of the fixed-effects parameters are required, as demonstrated in the previous sections. The `nlsList` function requires the initial estimates to be included in the data frame. The `nlme` function requires the initial estimates as input to the argument `start`. Alternatively, the approach to derive the initial estimates could be added to the model function as an attribute. With this "`initial`" attribute and derivatives, the model function becomes a self-starting function. When a self-starting function is used in calls to `nlsList` and `nlme`, initial estimates are no longer required. A self-starting function is considered as a class of models, which are useful for some particular applications. Several self-starting functions are provided with S-PLUS. For more information about them, choose Mixed Effects Models under Categories in the S-PLUS help window and select an entry with the extension `.func`. The following four self-starting functions are useful in Biostatistics.

- Biexponential model: `biexp(time, A1, A2, lrc1, lrc2)`

$$\alpha_1 e^{-e^{\beta_1 t}} + \alpha_2 e^{-e^{\beta_2 t}},$$

  where `time` $= t$ is a covariate, and `A1` $= \alpha_1$, `A2` $= \alpha_2$, `lrc1` $= \beta_1$, `lrc2` $= \beta_2$ are parameters.

- First Order Compartment model: `first.order.log(Dose, time, lCl, lka, lke)`

$$\frac{d \cdot e^{\beta} \cdot e^{\gamma} \cdot (e^{-e^{\gamma} t} - e^{-e^{\beta} t})}{e^{\alpha} \cdot (e^{\beta} - e^{\gamma})},$$

  where `Dose` $= d$ is a covariate, and `lCl` $= \alpha$, `lka` $= \beta$, `lke` $= \gamma$ are parameters.

- Four-parameter Logistic model: `fpl(conc, A, B, ld50, scal)`

$$\alpha + \frac{\beta - \alpha}{1 + e^{-(\log x - \gamma)/\theta}},$$

  where `conc` $= x$ is a covariate, and `A` $= \alpha$, `B` $= \beta$, `ld50` $= \gamma$, `scal` $= \theta$ are parameters.

- Logistic model: `logistic(time, Asym, T50, scal)`

$$\frac{\alpha}{1 + e^{-(t-\beta)/\gamma}},$$

where `time` = t is a covariate, and `Asym` = α, `T50` = β, `scal` = γ are parameters.

In our examples, we can apply the self-starting functions `logistic` and `first.order.log` to the `Soybean` data and the `Theoph` data, respectively. Taking the `Soybean` for example, the `nlsList` call is as follows.

```
> Soybean.nlsList2 <- nlsList(weight ~
+                       logistic(time, Asym, T50, scal),
+                       cluster = ~ Plot, data = Soybean)
Error in nls(formula = form, data = d..: singular
        gradient matrix
Dumped
Error in nls(formula = form, data = d..: singular
        gradient matrix
Dumped
> coef(Soybean.nlsList2)
 . . .
31  15.472178  46.34389  5.394573
32         NA        NA        NA
33  19.792019  55.69564  9.617943
 . . .
45  19.546035  51.15231  7.294394
46         NA        NA        NA
47  19.158168  54.79723 10.845646
48 18.51233 52.4458 8.579754
```

The error messages indicate that two clusters could not be fitted by `nls`. The object `Soybean.nlsList2` is still created by `nlsList`.The result shows that plots 32 and 46 do not follow the logistic model if the within-cluster variations are not adjusted.

The `nlme` function can also use the list from `nlsList` as input. In this case, all parameters are, by default, considered as both fixed effects and random effects. The results below show the strength of the `nlsList` and `nlme` functions: population parameters and individual random effects can still be estimated even though `nls` does not converge for clusters 32 and 46.

```
> Soybean.fit1 <- nlme(Soybean.nlsList2)
> summary(Soybean.fit1)
```

```
 . . .
Estimation Method: ML
Convergence at iteration: 6
Approximate Loglikelihood: -739.8383
AIC: 1499.677
BIC: 1539.887

Variance/Covariance Components Estimate(s):
  Structure: matrixlog
  Standard Deviation(s) of Random Effect(s)
     Asym      T50     scal
 5.200114 4.198423 1.406313
 Correlation of Random Effects
         Asym       T50
 T50 0.7203619
scal 0.7111132 0.9576767

 Cluster Residual Variance: 1.261742

Fixed Effects Estimate(s):
         Value Approx. Std.Error z ratio(C)
Asym 19.253091         0.8001025   24.06328
 T50 55.020581         0.7246942   75.92249
scal  8.402632         0.3142551   26.73825

 Conditional Correlation(s) of Fixed Effects Estimates
         Asym       T50
 T50 0.7239821
scal 0.6199830 0.8071952

Random Effects (Conditional Modes):
         Asym         T50        scal
 1   0.3328024  1.36946801  0.46828321
 2   0.5208869  1.33405459  0.36772317
 . . .
13  -1.1091253 -3.01949513 -0.98652484
14  13.5427273  8.18357241  2.71451747
15   6.3224302  4.49687021  1.38122773
 . . .
31  -3.3912435 -6.98830083 -2.24939631
32   2.9788882 -0.15444380 -0.39236193
```

```
33  -0.1304603 -0.18269024  0.04448421
 . . .
45   0.4859144 -3.10940467 -0.95349462
46  -1.2782517 -4.00341391 -1.29091905
47  -1.2594776 -1.55326184 -0.23867527
48  -0.7448815 -2.043285   -0.5467887
 . . .
```

`Soybean.fit1` does not incorporate covariates and within-cluster errors. Comparing the estimated standard deviations and means of `Asym`, `T50`, and `scal`, the asymptotic weight `Asym` has the highest coefficient of variation (0.27). Modeling this random-effects parameter is the focus of the following analyses.

As suggested in the exploratory data analysis, the asymptotic weight `Asym` can be modeled as a function of the variety of the genotype and planting year. To model the within-cluster errors, we will assume the serial correlation is of AR(1). From figure 11.10, the within-cluster variance is assumed to be proportional to some power of the absolute value of the predictions. Hence, for the *i*th plot, the 8×8 covariance matrix $\sigma^2 D$ in model (11.8) is now given by $\text{var}(e_{ij}) = \sigma^2 |\hat{y}_{ij}|^{2\delta}$ at time j and $\text{cor}(e_{ij}, e_{ik}) = \rho^{|k-j|}$ between the *i*th and *j*th measurements.

Two separate `nlme` calls are performed on the `Soybean` data below. The first call uses part of the assumptions in the above model. Only `Variety` is used as a predictor. The serial correlation is assumed to be of AR(1) and the within-cluster variance is assumed to be `identity`, a default for `var.function` in the `nlme` call.

In fitting the full model below with the second `nlme` call, the results from `Soybean.nlsList` are used to derive initial estimates in the parametrization of `Asym`.

```
> Soybean.fit2 <-
+     nlme(weight ~ logistic(time, Asym, T50, scal),
+          fixed = list(Asym ~ Variety, T50 ~ ., scal ~ .),
+          random = list(Asym ~ ., T50 ~ ., scal ~ .),
+          start = list(fixed = c(19.25309, 0,
+                       55.02058, 8.402632)),
+          serial.structure = "ar1",
+          cluster = ~ Plot, data = Soybean)
```

```
> Soybean.fit2
 . . .
  Standard Deviation(s) of Random Effect(s)
 Asym.(Intercept)      T50     scal
        4.520749 4.223438 1.446683
 Correlation of Random Effects
     Asym.(Intercept)       T50
 T50 0.7216902
scal 0.6259055        0.7851180

 Cluster Residual Variance: 1.110669

 Serial Correlation Structure: ar1
 Serial Correlation Parameter(s): -0.6164418

Fixed Effects Estimate(s):
  Asym.(Intercept) Asym.Variety      T50     scal
         19.39241     2.208246 55.31789 8.647944
 . . .
> # Get initial estimates in modeling Asym
> coef(lm(Asym ~ Variety * Year, Soybean.nlscoef))
 (Intercept) Variety     Year1     Year2
    20.08425 2.03699 -3.785161 0.3036094
 VarietyYear1 VarietyYear2
     1.497311    -1.084704
> Soybean.fit3 <-
+   nlme(weight ~ logistic(time, Asym, T50, scal),
+        fixed = list(Asym ~
+                     Variety * Year, T50 ~ ., scal ~ .),
+        random = list(Asym ~ ., T50 ~ ., scal ~ .),
+        start = list(fixed = c(20.08425, 2.03699,
+               -3.785161, 0.3036094, 1.497311,
+               -1.084704, 55.02058, 8.402632)),
+        serial.structure = "ar1",
+        var.function = "power",
+        cluster = ~ Plot, data = Soybean)
```

```
> Soybean.fit3
 . . .
  Standard Deviation(s) of Random Effect(s)
 Asym.(Intercept)      T50      scal
         1.11527 1.531973 0.1430054
 Correlation of Random Effects
     Asym.(Intercept)       T50
 T50 0.9731252
scal 0.9591480        0.9975064
 Cluster Residual Variance: 0.05957224
 Serial Correlation Structure: ar1
 Serial Correlation Parameter(s): 0.1874778
 Variance Function: power
 Variance Function Parameter(s): 0.9225696
Fixed Effects Estimate(s):
  Asym.(Intercept) Asym.Variety Asym.Year1 Asym.Year2
        17.53656     2.504791  -2.771701    1.21122
 Asym.VarietyYear1 Asym.VarietyYear2      T50
         1.264881        -0.3621853 52.22499
     scal
 7.603832
 . . .
> anova(Soybean.fit1,Soybean.fit2,Soybean.fit3)
 . . .
             Model Df    AIC    BIC  Loglik    Test
Soybean.fit1     1 10 1499.7 1539.9 -739.84
Soybean.fit2     2 12 1404.6 1452.9 -690.30 1 vs. 2
Soybean.fit3     3 17  679.9  748.2 -322.93 2 vs. 3
             Lik.Ratio P value
Soybean.fit1
Soybean.fit2     99.08       0
Soybean.fit3    734.74       0
```

The anova function is used to compare these fits. The progress in log likelihood, AIC, and BIC is tremendous. Figure 11.18 shows the residuals plots. No patterns are seen in the standardized residuals, even though the problematic plots 14, 32, and 46 for the nlsList calls are included in these analyses. After adjustment of random effects and within-cluster errors, their random effects are derived. Their standardized residuals are within (-1.5,1.5) except for three with values around 2.5. The variety of the genotype and the year of planting have large impacts on the limiting leaf weight. The

Figure 11.18: *The residuals plots.*

experimental strain gains 2.5 grams in the limit.

```
> par(mfrow = c(2,2))
> plot(Soybean.fit3, option="s")
> summary(Soybean.fit3)
 . . .
Fixed Effects Estimate(s):
                         Value Approx. Std.Error
 Asym.(Intercept)  17.5365613         0.4638525
     Asym.Variety   2.5047905         0.2469177
       Asym.Year1  -2.7717008         0.2829921
       Asym.Year2   1.2112198         0.1864453
Asym.VarietyYear1   1.2648812         0.2776887
Asym.VarietyYear2  -0.3621853         0.1844127
              T50  52.2249943         0.5073733
             scal   7.6038318         0.1027588
```

```
                    z ratio(C)
 Asym.(Intercept)   37.806334
     Asym.Variety   10.144233
       Asym.Year1   -9.794270
       Asym.Year2    6.496381
Asym.VarietyYear1    4.555032
Asym.VarietyYear2   -1.963994
              T50  102.932085
             scal   73.996904

 . . .
Random Effects (Conditional Modes):
   Asym.(Intercept)          T50          scal
 . . .
14      0.049307561  0.018802336  0.0008801504
 . . .
32      0.746219856  1.028182060  0.0951769261
 . . .
46     -0.581105585 -0.801146841 -0.0740665473
47     -0.214035251 -0.252438575 -0.0225885040
48     -0.5266585   -0.7154175   -0.06597178


Standardized Population-Average Residuals:
       Min         Q1        Med        Q3     Max
 -2.462553 -0.6578548 -0.1209506 0.5526576 4.05955
```

NONLINEAR MODELS 12

Nonlinear models utilize more general formulas and starting values to fit a model to experimental data.

NONLINEAR MODELS

This chapter covers the fitting of nonlinear models such as in nonlinear regression, likelihood models, and Bayesian estimation. Nonlinear models are more general than the linear models usually discussed. Specifying nonlinear models typically requires one or more of the following: more general formulas, extended data frames, starting values and derivatives.

The two most common fitting criteria for nonlinear models considered are Minimum sum and Minimum sum-of-squares. Minimum sum minimizes the sum of contributions from observations (the maximum likelihood problem). Minimum sum-of-squares minimizes the sum of squared residuals (the nonlinear least-squares regression problem).

The first sections of this chapter summarizes the use of the nonlinear optimization functions. Starting with the section on Examples of Nonlinear Models, the use of the `ms` and `nls` functions are examined, along with corresponding examples and theory, in much more detail.

12.1 OPTIMIZATION FUNCTIONS

S-PLUS has several functions for finding roots of equations and local maxima and minima of functions, as shown in table figure 12.1.

Finding Roots

The function `polyroot` finds the roots (zeros) of the complex-valued polynomial equation: $a_k z^k + \ldots + a_1 z + a_0 = 0$.

The input to `polyroot` is the vector of coefficients $c(a_0, \ldots, a_k)$. For example, to solve the equation `z^2 + 5z + 6 = 0`, use `polyroot` as follows:

```
> polyroot(c(6,5,1))
[1] -2+2.584939e-26i -3-2.584939e-26i
```

The function `uniroot` finds a zero of a continuous, univariate, real-valued function within a user-specified interval for which the function has opposite signs at the endpoints. The input to `uniroot` includes the function, the lower and upper endpoints of the interval, and any additional arguments to the function. For example, suppose you have the function:

```
> my.fcn
function(x, amp=1, per=2*pi, horshft=0, vershft=0)
{
  amp * sin(((2*pi)/per) * (x-horshft)) + vershft
}
```

Table 12.1: *The range of S-PLUS functions for finding roots, maxima and minima.*

Function	Description
`polyroot`	Finds the roots of a complex polynomial equation.
`uniroot`	Finds the root of a univariate real-valued function in a user-supplied interval.
`peaks`	Finds local maxima in a set of discrete points.
`optimize`	Approximates a local optimum of a continuous univariate function within a given interval.
`ms`	Finds a local minimum of a multivariate function.
`nlmin`	Finds a local minimum of a nonlinear function using a general quasi-Newton optimizer.
`nlminb`	Local minimizer for smooth nonlinear functions subject to bound-constrained parameters.
`nls`	Finds a local minimum of the sums of squares of one or more multivariate functions.
`nlregb`	Local minimizer for sums of squares of nonlinear functions subject to bound-constrained parameters.
`nnls`	Finds least-squares solution subject to the constraint that the coefficients be nonnegative.

This is the sine function with amplitude `abs(amp)`, period `abs(per)`, horizontal (phase) shift `horshft` and vertical shift `vershft`. To find a root of the function `my.fcn` in the interval $[\pi/2, 3\pi/2]$ using its default arguments, type:

```
> uniroot(my.fcn, interval = c(pi/2, 3*pi/2))
$root
[1] 3.141593
. . .
```

To find a root of `my.fcn` in the interval $[\pi/4, 3\pi/4]$ with the period set to π, type:

```
> uniroot(my.fcn, interval = c(pi/4, 3*pi/4), per=pi)
$root:
[1] 1.570796
. . .
> pi/2
[1] 1.570796
```

See the help file for `uniroot` for information on other arguments to this function.

Finding Local Maxima and Minima of Univariate Functions

The `peaks` function takes a data object `x` and returns an object of the same type with logical values: `T` if a point is a local maximum, otherwise `F`:

```
> peaks(corn.rain)
1890: F T F F F F T F T F T F T F F F F T F F F F T F F T F
1917: T F F F T F F T F T F
```

Use `peaks` on the data object `-x` to find local minima:

```
> peaks(-corn.rain)
1890: F F F F T F F F F F F T F F F T F F F F F T F T F F T
1917: F T F F F T F F T F F
```

To find a local optimum (maximum or minimum) of a continuous univariate function within a particular interval, use the `optimize` function. The input to `optimize` includes the function to optimize, the lower and upper endpoints of the interval, which optimum to look for (maximum versus minimum) and any additional arguments to the function.

```
> optimize(my.fcn, c(0, pi), maximum=T)
$maximum:
[1] 1.570799

$objective:
[1] -1

$nf:
[1] 10

$interval:
[1] 1.570759 1.570840
. . .
```

```
> pi/2
[1] 1.570799

> optimize(my.fcn, c(0, pi), maximum=F, per = pi)
$minimum:
[1] 2.356196

$objective:
[1] -1

$nf:
[1] 9

$interval:
[1] 2.356155 2.356236
. . .
> 3*pi/4
[1] 2.356194
```

See the help file for `optimize` for information on other arguments to this function.

Finding Maxima and Minima of Multivariate Functions

S-PLUS has two functions to find the local minimum of a multivariate function: `nlminb` (Nonlinear Minimization with Box Constraints) and `ms` (Minimize Sums).

The two required arguments to `nlminb` are `objective` (the function f to minimize) and `start` (a vector of starting values for the minimization). The function f must take as its first argument a vector of parameters over which the minimization is carried out. By default, there are no boundary constraints on the parameters. The `nlminb` function, however, also takes the optional arguments `lower` and `upper` that specify the bounds on the parameters. (Other arguments to *f* can be passed in the call to `nlminb`.)

1. Example: Using `nlminb` to find a local minimum.

```
> my.multvar.fcn
function(xvec, ctr = rep(0, length(xvec)))
{
  if(length(xvec) != length(ctr))
     stop("lengths of xvec and ctr do not match")
  sum((xvec - ctr)^2)
}
> nlminb(start = c(0,0), objective = my.multvar.fcn,
```

```
+ ctr = c(1,2))
$parameters:
[1] 1 2

$objective:
[1] 3.019858e-30

$message:
[1] "ABSOLUTE FUNCTION CONVERGENCE"
. . .
```

To find a local maximum of *f*, use `nlminb` on *-f*. Since unary minus cannot be performed on a function, you must define a new function that returns -1 times the value of the function you want to maximize:

2. Example: Using `nlminb` to find a local maximum.

```
> fcn.to.maximize
function(xvec)
{
  - xvec[1]^2 + 2 * xvec[1] - xvec[2]^2 + 20 * xvec[2] + 40
}

> fcn.to.minimize
function(xvec)
{
  - fcn.to.maximize(xvec)
}

> nlminb(start = c(0, 0), objective = fcn.to.minimize)
$parameters:
[1] 1 10

$objective:
[1] -141

$message:
[1] "RELATIVE FUNCTION CONVERGENCE"
. . .
```

See the help file for `nlminb` for information on other arguments to this function. To find the local minimum of a multivariate function subject to constraints, use `nlminb` with the `lower` and/or `upper` arguments.

3. Example: Using `nlminb` to find a constrained minimum.

As an example of using `nlminb` to find a constrained minimum, consider the following function `norm.neg.2.ll`, which is (minus a constant) `-2` times the log-likelihood function of a normal (Gaussian) distribution:

```
> norm.neg.2.ll <-
+ function(theta, y)
+ {
+ length(y) * log(theta[2]) +
+ (1/theta[2]) * sum((y - theta[1])^2)
+ }
```

This function assumes that observations from a normal distribution are stored in the vector `y`. The vector `theta` contains the mean (`theta[1]`) and variance (`theta[2]`) of this distribution. To find the maximum likelihood estimates of the mean and variance, we need to find the values of `theta[1]` and `theta[2]` that minimize `norm.neg.2.ll` for a given set of observations stored in `y`. We must use the lower argument to `nlminb` because the estimate of variance (`theta[2]`) must be greater than zero:

```
> set.seed(12)
> my.obs <- rnorm(100, mean = 10, sd = 2)
> nlminb(start = c(0,1), objective = norm.neg.2.ll,
+ lower = c(-Inf, 0), y = my.obs)
$parameters:
[1] 9.863812 3.477773

$objective:
[1] 224.6392

$message:
[1] "RELATIVE FUNCTION CONVERGENCE"
. . .

> mean(my.obs)
[1] 9.863812
> (99/100) * var(my.obs)
[1] 3.477774
```

The Minimum Sums function `ms` also minimizes a multivariate function, but in the context of the modeling paradigm, so it expects a formula rather than a function as its main argument. Here is the last example redone with `ms` (`mu` is the estimate of the population mean μ, and `ss` is the estimate of the population variance σ^2):

4. Example: Using `ms`.

```
> ms( ~length(y) * log(ss) + (1/ss) * sum((y - mu)^2),
+ data = data.frame(y = my.obs),
+ start = list(mu = 0, ss = 1))
value: 224.6392
parameters:
      mu       ss
9.863813 3.477776

formula:  ~length(y) * log(ss) + (1/ss) * sum((y-mu)^2)

1 observations

call: ms(formula = ~length(y) * log(ss) + (1/ss) *
        sum((y - mu)^2),
data = data.frame(y=my.obs), start=list(mu=0, ss=1))
```

Hint: the `ms` function does not do minimization subject to constraints on the parameters.

If there are multiple solutions to your minimization problem, you may *not* get the answer you want using `ms`. In the above example, the `ms` function tells us we have "1 observations" because the whole vector `y` was used at once in the formula. The Minimum Sum function minimizes the *sum* of contributions to the formula, so we could have gotten the same estimates `mu` and `ss` with the formula shown in example 5.

5. Example: Using `ms` with several observations.

```
> ms( ~log(ss) + (1/ss) * (y - mu)^2,
+ data = data.frame(y = my.obs),
+ start = list(mu = 0, ss = 1))
value: 224.6392

parameters:
      mu       ss
9.863813 3.477776

formula:  ~log(ss) + (1/ss) * (y - mu)^2

100 observations
call: ms(formula =  ~log(ss) + (1/ss) * (y - mu)^2,
data = data.frame(y=my.obs), start=list(mu=0,ss=1))
```

If the function you are trying to minimize is fairly complicated, then it is usually easier to write a function to supply as the formula:

6. Example: Using `ms` with a formula function.

```
> ms( ~norm.neg.2.ll(theta,y), data=data.frame(y=my.obs),
+ start = list(theta = c(0,1)))

value: 224.6392

parameters:
  theta1   theta2
9.863813 3.477776

formula:  ~norm.neg.2.ll(theta, y)

1 observations

call: ms(formula = ~norm.neg.2.ll(theta, y), data =
  data.frame(y = my.obs),
start = list(theta = c(0, 1)))
```

Solving Nonnegative Least Squares Problems

Given an $m \times n$ matrix A and a vector $\boldsymbol{b}$ of length m, the linear nonnegative least squares problem is to find the vector $\boldsymbol{x}$ of length n that minimizes $\|Ax - b\|$, subject to the constraint that $x_i \geq 0$ for i in 1, ..., n.

To solve nonnegative least squares problems in S-PLUS, use the `nnls.fit` function. For example, consider the following fit using the `stack` data:

```
$coefficients
  Air Flow Water Temp Acid Conc.
0.2858057  0.05715152          0

$residuals:
[1]  17.59245246 12.59245246 14.13578403
[4]   8.90840973 -0.97728723 -1.03443875
[7]  -0.09159027  0.90840973 -2.89121593
[10] -3.60545832 -3.60545832 -4.54830680
[13] -6.60545832 -5.66260984 -7.31901267
[16] -8.31901267 -7.37616419 -7.37616419
[19] -6.43331572 -2.14814995 -6.14942983

$dual:
    Air Flow   Water Temp Acid Conc.
3.637979e-12 5.400125e-13 -1438.359
```

```
$rkappa:
     final    minimum
0.02488167 0.02488167

$call:
nnls.fit(x = stack.x, y = stack.loss)
```

You can also use `nlregb` to solve the nonnegative least squares problem, since the nonnegativity constraint is just a simple box constraint. To pose the problem to `nlregb`, define two functions (`lin.res` and `lin.jac`) of the form `f(x,params)`, to represent the residual function and the Jacobian of the residual function, respectively:

```
> lin.res <- function(x, b, A) A%*% x - b
> lin.jac <- function(x, A) A
> nlregb(n = length(stack.loss), start = rnorm(3),
+ res = lin.res, jac = lin.jac, lower = 0,
+ A = stack.x, b = stack.loss)

$parameters:
[1] 0.28580571 0.05715152 0.00000000

$objective:
[1] 1196.252
. . .
```

Generally, `nnls.fit` should be preferred to `nlregb` for reasons of efficiency, since `nlregb` is primarily designed for nonlinear problems. However, `nlregb` can solve degenerate problems that can not be handled by `nnls.fit`. You may also want to compare the results of `nnls.fit` with those of `lm`. Remember that `lm` requires a formula, and also that it fits an intercept term by default (which `nnls.fit` does not). Keeping this in mind, you can construct the comparable call to `lm` as follows:

```
> lm(stack.loss ~ stack.x - 1)
Call:
lm(formula = stack.loss ~ stack.x - 1)
Coefficients:
stack.xAir Flow stack.xWater Temp
      0.7967652          1.111422 -0.6249933

Degrees of freedom: 21 total; 18 residual
Residual standard error: 4.063987
```

For the stack loss data, the results of the constrained optimization methods `nnls.fit` and `nlregb` agree completely. The linear model produced by `lm` includes a negative coefficient.

You can use `nnls.fit` to solve the weighted nonnegative least squares problem by providing a vector of weights as the `weights` argument. The weights used by `lm` are the square roots of the weights used by `nnls.fit`; you must keep this in mind if you are trying to solve a problem using both functions.

Solving Nonlinear Least Squares Problems

Two functions, `nls` and `nlregb`, are available for solving the special minimization problem of nonlinear least squares. The function `nls` is used in the context of the modeling paradigm, so it expects a formula rather than a function as its main argument. The function `nlregb` expects a function rather than a formula (the argument name is `residuals`), and, unlike `nls`, it can perform the minimization subject to constraints on the parameters.

1. Example: Using `nls`.

In this example, we create 100 observations where the underlying signal is a sine function with an amplitude of 4 and a horizontal (phase) shift of π. Noise is added in the form of normal (Gaussian) random numbers. We then use the `nls` function to estimate the true values of amplitude and horizontal shift.

```
> set.seed(20)
> noise <- rnorm(100, sd = 0.5)
> x <- seq(0, 2*pi, length = 100)
> my.nl.obs <- 4 * sin(x - pi) + noise
> plot(x, my.nl.obs)
> nls(y ~ amp * sin(x - horshft),
+ data = data.frame(y = my.nl.obs, x = x),
+ start = list(amp = 1, horshft = 0))
Residual sum of squares : 20.25668
parameters:
       amp    horshft
 -4.112227 0.01059317
formula: y ~ amp * sin(x - horshft)
100 observations
```

The above example illustrates the importance of finding appropriate starting values. The `nls` function returns an estimate of `amp` close to `-4` and an estimate of `horshft` close to 0 because of the cyclical nature of the sine function: `sin(x - pi) = -sin(x)`. If we start with initial estimates of `amp` and `horshft` closer to their true values, `nls` gives us the estimates we want.

2. Example: Using `nls` with better starting values.

```
> nls(y ~ amp * sin(x - horshft),
+ data = data.frame(y = my.nl.obs, x = x),
+ start = list(amp = 3, horshft = pi/2))
Residual sum of squares : 20.25668
parameters:
      amp horshft
 4.112227  -3.131
formula: y ~ amp * sin(x - horshft)
100 observations
```

We could use the `nlregb` function to redo the above example, and specify that the value of `amp` must be greater than 0:

3. Example: Creating `my.new.func` and using `nlregb`.

```
> my.new.fcn
function(param, x, y)
{
  amp <- param[1]
  horshft <- param[2]
  y - amp * sin(x - horshft)
}
> nlregb(n = 100, start = c(3,pi/2),
+ residuals = my.new.fcn,
+ lower = c(0, -Inf), x = x, y = my.nl.obs)
$parameters:
[1] 4.112227 3.152186

$objective:
[1] 20.25668

$message:
[1] "BOTH X AND RELATIVE FUNCTION CONVERGENCE"

$grad.norm:
[1] 5.960581e-09
```

12.2 EXAMPLES OF NONLINEAR MODELS

Maximum Likelihood Estimation

Parameters are estimated by maximizing the likelihood function. Suppose n independent observations are distributed with probability densities $p_i(\theta) = p(y_i;\theta)$ where θ is a vector of parameters. The likelihood function is defined as:

$$L(y;\theta) = \prod_{i=1}^{n} p_i(\theta) \tag{12.1}$$

The problem is to find the estimate $\tilde{\theta}$ of that maximizes the likelihood function for the observed data. Maximizing the likelihood is equivalent to minimizing the negative of the log-likelihood:

$$l(\theta) = -log(L(y;\theta)) = \sum_{i=1}^{n} (-log(p_i(\theta))) \tag{12.2}$$

Example One: Ping-Pong

Each member of the U.S. Table Tennis Association is assigned a rating based on the member's performance in tournaments. Winning a match boosts the winner's rating and lowers the loser's rating some number of points depending on the current ratings of the two players. Using this data, two questions we might like to ask are:

1. Do players with a higher rating tend to win over players with a lower rating?
2. Does a larger difference in rating imply that the higher-rated player is more likely to win?

Assuming a logistic distribution in which `log(p/(1-p))` is proportional to the difference in rating between the winner and loser and the average rating of the two players:

$$p_i = \frac{e^{D_i\alpha + R_i\beta}}{1 + e^{D_i\alpha + R_i\beta}} \tag{12.3}$$

where `Di = Wi - Li` is the difference in rating between the winner and loser and `Ri = 1/2(Wi + Li)` is the average rating for the two players.

To fit the model, we need to find α and β which minimize the negative log-

likelihood:

$$\sum(-log(p_i)) = \sum\left\{-D_i\alpha - R_i\beta + log(1 + e^{D_i\alpha + R_i\beta})\right\} \quad (12.4)$$

Example Two: Wave-Soldering Skips

In a 1988 AT&T wave-soldering experiment several factors were varied:

Factor	Description
`opening`	amount of clearance around the mounting pad
`solder`	amount of solder
`mask`	type and thickness of the material used for the solder mask
`padtype`	the geometry and size of the mounting pad
`panel`	each board was divided into three panels, with three runs on a board

The results of the experiment gave the number of visible soldering skips (faults) on a board. Physical theory and intuition suggest a model in which the process is in one of two states:

1. a "perfect" state where no defects occur.
2. an "imperfect" state where there may or may not be defects.

Both the probability of being in the imperfect state and the distribution of skips in that state depend on the factors in the experiment. Assume that some "stress", S, induces the process to be in the imperfect state and also increases the tendency to generate skips when in the imperfect state.

Assume S depends linearly on the levels of the factors, x_j, j=1,...,p:

$$S_i = \sum_{j=1}^{p} x_{ij}\beta_j \quad (12.5)$$

where β is the vector of parameters to be estimated.

Assume the probability P_i of being in the imperfect state is monotonically

related to the stress by a logistic distribution:

$$P_i = \frac{1}{1 + e^{(-\tau)S_i}} \tag{12.6}$$

As the stress increases, the above function approaches 1.

Given that the process is in an imperfect state, assume the probability of k_i skips is modeled by the Poisson distribution with mean λ_i:

$$P(k_i) = e^{-\lambda_i} \cdot \frac{\lambda_i^{k_i}}{k_i!} \tag{12.7}$$

The probability of zero skips is the probability of being in the perfect state plus the probability of being in the imperfect state and having zero skips. The probability of one or more skips is the probability of being in the imperfect state and having one or more skips. Mathematically the probabilities may be written as:

$$P(y = y_i) = \begin{cases} \dfrac{e^{(-\tau)S_i}}{1 + e^{(-\tau)S_i}} + \dfrac{e^{-\lambda_i}}{1 + e^{(-\tau)S_i}} & if \quad y_i = 0 \\ \dfrac{1}{1 + e^{(-\tau)S_i}} e^{-\lambda_i} \dfrac{\lambda_i^{y_i}}{y_i!} & if \quad y_i > 0 \end{cases} \tag{12.8}$$

The mean skips in the imperfect state is always positive and modeled in terms of the stress by: $\lambda_i = e^{S_i}$. The parameters, τ and β, can be estimated by minimizing the negative log-likelihood. The *i*th element of the negative log-likelihood can be written to within constants as:

$$l_i(\beta, \tau) = log(1 + e^{(-\tau)S_i}) - \begin{cases} log\left(e^{(-\tau)S_i} + e^{-e^{S_i}}\right) & if \quad y_i = 0 \\ y_i S_i - e^{S_i} & if \quad y_i > 0 \end{cases} \tag{12.9}$$

The model depicted above does not reduce to any simple linear model.

Nonlinear Regression

Parameters are estimated by minimizing the sum of squared residuals. Suppose n independent observations $\boldsymbol{y}$ can be modeled as a nonlinear parametric function f of a vector $\boldsymbol{x}$ of predictor variables and a vector of parameters, β. $y = f(x;\beta) + \varepsilon$

where the errors, ε, are assumed normally distributed. The nonlinear least-squares problem finds parameter estimates $\tilde{\beta}$ which minimize:

$$\sum_{i=1}^{n} (y_i - f(x;\beta))^2 \tag{12.10}$$

Example Three: Puromycin

A biochemical experiment measured reaction velocity in cells with and without treatment by Puromycin. There are three variables in the `Puromycin` data frame.

Variable	Description
`conc`	the substrate concentration
`vel`	the reaction velocity
`state`	indicator of treated or untreated

Assume a Michaelis-Menten relationship between velocity and concentration:

$$V = \frac{V_{max} c}{K + c} + \varepsilon \tag{12.11}$$

where V is the velocity, c is the enzyme concentration, V_{max} is a parameter representing the asymptotic velocity as $c \rightarrow \infty$, K is the Michaelis parameter, and ε is experimental error. Assuming the treatment with the drug would change V_{max} but not K, the optimization function is:

$$S(V_{max}, K) = \sum \left(V_i - \frac{(V_{max} + \Delta V_{max} I_{\{treated\}}(state)) c_i}{K + c_i} \right)^2 \tag{12.12}$$

where $I_{\{treated\}}$ is the function indicating if the cell was treated with Puromycin.

12.3 INFERENCE FOR NONLINEAR MODELS

Likelihood Models

With likelihood models distributional results are asymptotic. Maximum likelihood estimates tend toward a normal distribution with a mean equal to the true parameter, and a variance matrix given by the inverse of the information matrix, the negative of the second derivatives of the log-likelihood.

Least-Squares Models

In least-squares models approximations to quantities such as standard errors or correlations of parameter estimates are used. The approximation proceeds as follows:

1. Replace the nonlinear model with its linear Taylor series approximation at the parameter estimates.
2. Use the methods for *linear* statistical inference on the approximation.

Consequently, the nonlinear inference results are called linear approximation results.

The Fitting Algorithms

Minimum-Sum Algorithm

This section deals with the general optimization of an objective function modeled as a sum. The algorithm is a version of Newton's method based on a quadratic approximation of the objective function. If both first and second derivatives are supplied, the approximation is a local one using the derivatives. If no derivatives or only the first derivative are supplied, the algorithm approximates the second derivative information. It does this in a way specifically designed for minimization.

The algorithm actually used is taken from the PORT subroutine library which evolved from the published algorithm by Gay (1983). Two key features of this algorithm are:

1. A quasi-Newton approximation for second derivatives.
2. A "trust region" approach controlling the size of the region in which the quadratic approximation is believed to be accurate.

The algorithm is capable of working with user models specifying 0, 1, or 2 orders of derivatives.

Nonlinear Least-Squares Algorithm

The Gauss-Newton algorithm is used with a step factor to ensure that the sum of squares decreases at each iteration. A line-search method is used, as opposed to the trust region employed in the minimum-sum algorithm. The step direction is determined by a quadratic model. The algorithm proceeds as follows:

1. The residuals are calculated, and the gradient is calculated or approximated (depending on the data), at the current parameter values.
2. A linear least-squares fit of the residual on the gradient gives the parameter increment.
3. If applying the full parameter increment increases the sum-of-squares rather than decreasing it, the length of the increment is successively halved until the sum-of-squares is decreased.
4. The step factor is retained between iterations and started at min{2*(*previous step factor*), 1}

If the gradient is not specified analytically, it is calculated using finite differences with forward differencing. For partially linear models, the increment is calculated using the Golub-Pereyra method (Golub and Pereyra, 1973) as implemented by Bates and Lindstrom (1986).

Specifying Models

Nonlinear models typically require specifying more details than models of other types. The information typically required to fit a nonlinear model, using the S-PLUS functions `ms` or `nls` is:

1. a formula
2. data
3. starting values

Formulas

For nonlinear models a formula is an S-PLUS expression involving data, parameters in the model, and any other relevant quantities. The parameters must be specified in the formula because there is no assumption about where they are to be placed (as in linear models, for example). Formulas are typically specified differently depending on whether you have a minimum-sum problem or nonlinear least-squares problem.

In the puromycin example you would specify a formula for the simple model (described in equation 12.11) by: `vel ~ Vm*conc / (K + conc)`

The parameters `Vm` and `K` are specified along with the data `vel` and `conc`. Since there is no explicit response for minimum-sum models (e.g. likelihood models), it is left off in the formula.

In the ping-pong example (ignoring the average rating effect), the formula for (equation 12.4) is:

```
~- DV * alpha + log( 1 + exp( DV * alpha ) )
```

where `DV` is a variable in the data and `alpha` is the parameter to fit. Note that the model here is based only on the difference in ratings, ignoring for the moment the average rating.

Simplifying Formulas

Some models can be organized as a simple expression involving one or more S-PLUS functions that do all the work. Note that `DV*alpha` occurs twice in the formula for the ping-pong model. You can write a general function for the log-likelihood in terms of `DV*alpha`.

```
> lprob <- function(lp) log(1 + exp(lp)) - lp
```

Recall that `lp` is the linear predictor for the GLM. A simpler expression for the model is now:

```
~ lprob( DV * alpha )
```

Having `lprob` now makes it easy to add additional terms or parameters.

Implications of the Formulas

For nonlinear least-squares formulas the response on the left of ~ and the predictor on the right must evaluate to numeric vectors of the same length. The fitting algorithm tries to estimate parameters to minimize the sum of squared differences between response and prediction. If the response is left out the formula is interpreted as a residual vector.

For Minimum-Sum formula, the right of ~ must evaluate to a numeric vector. The fitting algorithm tries to estimate parameters to minimize the sum of this "predictor" vector. The concept here is linked to maximum-likelihood models. The computational form does not depend on an MLE concept. The elements of the vector may be anything and there need not be more than one.

The evaluated formulas can include derivatives with respect to the parameters. The derivatives are supplied as attributes to the vector that results when the predictor side of the formula is evaluated. When explicit derivatives are not supplied, the algorithms use numeric approximations.

Parametrized Data Frames

Relevant data for nonlinear modeling includes:

- variables
- initial estimates of parameters
- fixed values occurring in a model formula

Parametrized data frames allow you to "attach" relevant data to a data frame when the data doesn't occupy an entire column. Information is attached as a

"parameter" attribute of the data frame. The `parameter` function returns or modifies the entire list of parameters and is analogous to the `attributes` function. Similarly the `param` function returns or modifies one parameter at a time and is analogous to the `attr` function. You could supply values for `Vm` and `K` to the `Puromycin` data frame with:

```
> parameters(Puromycin) <- list(Vm = 200, K = 0.1)
```

The parameter values can be retrieved with

```
> parameters(Puromycin)
$Vm:
[1] 200

$K:
[1] 0.1
```

The class of `Puromycin` is now:

```
> class(Puromycin)
[1] "pframe" "data.frame"
```

Now, when `Puromycin` is attached, the parameters `Vm` and `K` are available when referred to in formulas.

Starting Values; Identifying Parameters

Before the formulas can be evaluated, the fitting functions must know which names in the formula are parameters to be estimated and must have starting values for these parameters. The fitting functions determine this in the following way:

1. If the `start` argument is supplied, its names are the names of the parameters to be estimated, and its values are the corresponding starting values.
2. If `start` is missing, the parameters attribute of the `data` argument defines the parameter names and values

Hint: explicitly use the start argument to name and initialize parameters.
You can easily see what the starting values are in the `call` component of the fit and you can arrange to keep particular parameters constant when that makes sense.

Derivatives

Supplying derivatives of the predictor side of the formula with respect to the parameters along with the formula can reduce the number of iterations (so speed up the computations), increase numerical accuracy and improve the chance of convergence. In general derivatives should be used whenever possible.

The fitting algorithms can use both first (the gradient) and second derivatives (the Hessian). The derivatives are supplied to the fitting functions as attributes to the formula. Recall that evaluating the formula gives a vector of n values. Evaluating the first derivative expression should give n values for each of the p parameters, that is an $n \times p$ matrix. Evaluating the second derivative expression should give n values for each of the $p \times p$ partial derivatives, that is an $n \times p \times p$ array.

First Derivatives

The negative log-likelihood for the simple ping-pong model is:

$$l(\alpha) = \sum [log(1 + e^{D_i \alpha}) - D_i \alpha] \tag{12.13}$$

Differentiating with respect to α and simplifying gives the gradient:

$$\frac{\partial l}{\partial \alpha} = \sum \left[\frac{-D_i}{(1 + e^{D_i \alpha})} \right] \tag{12.14}$$

The gradient is supplied to the fitting function as the `"gradient"` attribute of the formula:

```
> form.pp <-  ~log(1 + exp( DV*alpha ) ) - DV*alpha
> attr(form.pp, "gradient") <-
+  ~ -DV / ( 1 + exp( DV*alpha ) )
> form.pp
 ~ log(1 + exp(DV * alpha)) - DV * alpha
> attr(form.pp,"gradient")
 ~ - DV/(1 + exp(DV * alpha))
```

When a function is used to simplify a formula, build the gradient into the function. The `lprob` function is used to simplify the formula expression to `~lprob(DV*alpha)`.

```
> lprob
function(lp)
log(1 + exp(lp)) - lp
```

An improved version of `lprob` adds the gradient.

```
> lprob2
function(lp, X)
{
  elp <- exp(lp)
  z <- 1 + elp
  value <- log(z) - lp
```

```
  attr(value, "gradient") <- -X/z
  value
}
```

Note `lp` is again the linear predictor and `X` is the data in the linear predictor. With the gradient built into the function you don't need to add it as an attribute to the formula; it is already an attribute to the object hence used in the formula.

Second Derivatives

The second derivatives may be added as the `"hessian"` attribute of the formula. In the ping-pong example the second derivative of the negative log-likelihood with respect to α is:

$$\frac{\partial^2 l}{\partial \alpha^2} = \sum \frac{D_i^2 e^{D_i \alpha}}{(1 + e^{D_i \alpha})^2} \tag{12.15}$$

The `lprob2` function is now modified to add the Hessian as follows. The Hessian is added in a general enough form to allow for multiple predictors.

```
> lprob3
function(lp, X)
{
   elp <- exp(lp)
   z <- 1 + elp
   value <- log(z) - lp
   attr(value, "gradient") <- -X/z
   if(length(dx <- dim(X)) == 2)
   {
      n <- dx[1]; p <- dx[2]
   } else
   {
      n <- length(X); p <- 1
   }
   xx <- array(X, c(n, p, p))
   attr(value, "hessian") <- (xx * aperm(xx, c(1, 3, 2)) *
         elp)/z^2
   value
}
```

Interesting points of the added code are:

- The second derivative computations are performed at the time of the assignment of the `"hessian"` attribute.
- The rest of the code (starting with `if(length(...))`) is to make

the Hessian general enough for multiple predictors.

- The `aperm` function does the equivalent of a transpose on the 2nd and 3rd dimensions to produce the proper cross products when multiple predictors are in the model.

Symbolic Differentiation

A symbolic differentiation function, `D`, is available to aid in taking derivatives.

Table 12.2: *Arguments to* `D`

Argument	Purpose
`expr`	Expression to be differentiated.
`name`	Which parameters to differentiate with respect to.

The function `D` is used primarily as a support routine to `deriv`.

Again referring to the ping-pong example, `form` contains the expression of the negative log-likelihood:

```
> form
expression(log((1 + exp(DV * alpha))) - DV * alpha)
```

The first derivative is computed as:

```
> D(form, "alpha")
(exp(DV * alpha) * DV)/(1 + exp(DV * alpha)) - DV
```

And the second derivative is computed as:

```
> D( D(form, "alpha"), "alpha")
(exp(DV * alpha) * DV * DV)/(1 + exp(DV * alpha))
   - (exp(DV * alpha) * DV * (exp(DV * alpha) * DV))
   /(1 + exp(DV * alpha))^2
```

Improved Derivatives

The `deriv` function takes an expression, computes a derivative, simplifies the result then returns an expression or function for computing the original

expression along with its derivative(s).

Table 12.3: *Arguments to* `deriv`

Argument	Purpose
`expr`	Expression to be differentiated, typically a formula, in which case the expression returned computes the right side of the ~ *and* its derivatives.
`namevec`	Character vector of names of parameters.
`function.arg`	Optional argument vector or prototype for a function.
`tag`	Base of the names to be given to intermediate results. Default is `".expr"`.

Periods are used in front of created object names to avoid conflict with user chosen names. The `deriv` function returns an expression in the form expected for nonlinear models.

```
> deriv(form,"alpha")
expression(
{
   .expr1 <- DV * alpha
   .expr2 <- exp(.expr1)
   .expr3 <- 1 + .expr2
   .value <- (log(.expr3)) - .expr1
   .grad <- array(0, c(length(.value), 1), list(NULL,
"alpha"))
   .grad[, "alpha"] <- ((.expr2 * DV)/.expr3) - DV
   attr(.value, "gradient") <- .grad
   .value
})
```

If the `function.arg` argument is supplied, a function is returned.

```
> deriv(form,"alpha",c("DV","alpha"))
function(DV, alpha)
{  .expr1 <- DV * alpha
   .expr2 <- exp(.expr1)
   .expr3 <- 1 + .expr2
   .value <- (log(.expr3)) - .expr1
   .grad <- array(0, c(length(.value), 1), list(NULL,
"alpha"))
```

```
    .grad[, "alpha"] <- ((.expr2 * DV)/.expr3) - DV
    attr(.value, "gradient") <- .grad
    .value
}
```

The `namevec` argument can be a vector.

```
> deriv(vel ~ Vm * (conc/(K + conc)), c("Vm","K"))
expression(
{  .expr1 <- K + conc
   .expr2 <- conc/.expr1
   .value <- Vm * .expr2
   .grad <- array(0, c(length(.value), 2), list(NULL,
c("Vm","K")))
   .grad[, "Vm"] <- .expr2
   .grad[, "K"] <- - (Vm * (conc/(.expr1^2)))
   attr(.value, "gradient") <- .grad
   .value
})
```

The symbolic differentiation interprets each parameter as a scalar. Generalization from scalar to vector parameters (for example, `lprob2`) must be done by hand. Use parentheses to help `deriv` find relevant subexpressions. Without the redundant parentheses around `conc/(K + conc)` the expression `deriv` returns is not as simple as possible.

Fitting Models

There are two different fitting functions for nonlinear models:

- `ms` minimizes the sum of the vector supplied as the right side of the formula.
- `nls` minimizes the sum of squared differences between the left and right sides of the formula.

Table 12.4: *Arguments to* `ms`

Argument	Purpose
`formula`	The nonlinear model formula (without a left side).
`data`	A data frame in which to do the computations.
`start`	Numeric vector of initial parameter values for the iteration.
`scale`	Parameter scaling.
`control`	List of control values to be used in the iteration.
`trace`	Indicates whether or not intermediate estimates should be printed.

Table 12.5: *Arguments to* `nls`

Argument	Purpose
`formula`	The nonlinear regression model as a formula.
`data`	A data frame in which to do the computations.
`start`	Numeric vector of initial parameter values for the iteration.
`control`	List of control values to be used in the iteration.
`algorithm`	Which algorithm to use. The default algorithm is a Gauss-Newton algorithm. If `algorithm = "plinear"`, the Golub-Pereyra algorithm for partially linear least-squares models is used.
`trace`	Indicates whether or not intermediate estimates should be printed.

Fitting a Model to the Puromycin Data

Before fitting a model, take a look at the data displayed in figure 12.1.

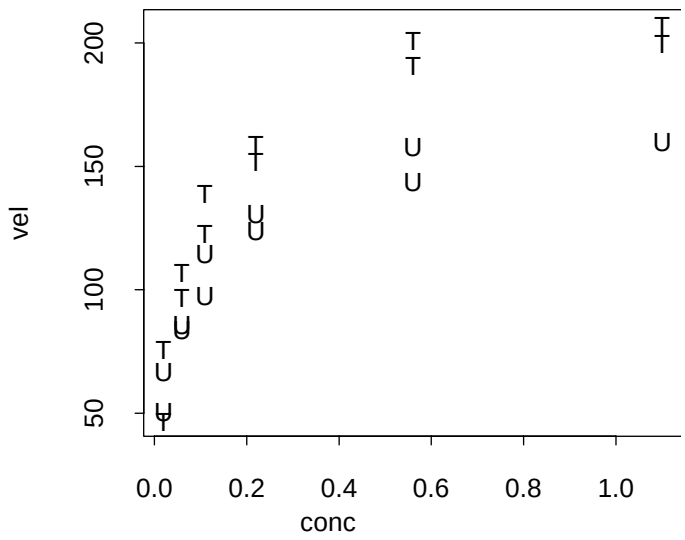

Figure 12.1: `vel` *versus* `conc` *for treated (T) and untreated (U) groups.*

```
> attach(Puromycin)
> plot(conc,vel,type="n")
> text(conc,vel,ifelse(state == "treated", "T","U"))
```

1. Estimating Starting Values

Obtain an estimate of V_{max} for each group as the maximum value each group attains.

- The treated group has a maximum of about 200.
- The untreated group has a maximum of about 160.

The value of K is the concentration at which V reaches $V_{max}/2$, roughly 0.1 for each group.

2. A Simple Model

Start by fitting a simple model for the treated group only.

```
> Treated <- Puromycin[Puromycin$state == "treated",]
> Purfit.1 <- nls(vel ~ Vm*conc/(K + conc), data = Treated,
+ start = list(Vm = 200, K = 0.1))
> Purfit.1
residual sum of squares: 1195.449
```

```
parameters:
      Vm          K
212.6836 0.06412111

12 observations

gradient.norm:
[1] 0.2781043

RELATIVE FUNCTION CONVERGENCE

formula: vel~(Vm * conc)/(K + conc)
```

Fit a model for the untreated group similarly but with `Vm = 160`.

```
> Purfit.2
residual sum of squares: 859.6043
parameters:
    Vm          K
160.28 0.04770808

11 observations

gradient.norm:
[1] 0.1389438

RELATIVE FUNCTION CONVERGENCE

formula: vel ~ (Vm * conc)/(K + conc)
```

3. A More Complicated Model

Obtain summaries of the fits with the `summary` function:

```
> summary(Purfit.1)
parameters:
          Value   Std.Error   t value
Vm 212.68362994 6.947148850 30.614520
 K   0.06412111 0.008280931  7.743225
. . .
sigma: 10.93366
df:
[1] 2 10
. . .
```

```
correlation:
          Vm         K
Vm 1.0000000 0.7650835
 K 0.7650835 1.0000000

formula:
vel ~ (Vm * conc)/(K + conc)
> summary(Purfit.2)
parameters:
          Value   Std.Error   t value
Vm 160.27997949 6.480240801 24.733646
 K   0.04770808 0.007781862  6.130677
. . .
sigma: 9.773003
df:
[1] 2 9
. . .
correlation:
          Vm         K
Vm 1.0000000 0.7768269
 K 0.7768269 1.0000000

formula:
vel ~ (Vm * conc)/(K + conc)
```

An approximate t-test for the difference in *K* between the two models suggests there is no difference:

```
> (.06412 - .04771)/sqrt(.00828^2 + .007782^2)
[1] 1.44416
```

The correct test of whether the *K*s should be different, and is as follows:

```
> Purboth <- nls(vel ~ (Vm + delV*(state=="treated"))*conc/
+ (K + conc), data=Puromycin,
+ start=list(Vm=160, delV=40, K=0.05))
> summary(Purboth)
parameters:
            Value   Std.Error   t value
  Vm 166.60396617 5.807422147 28.688110
delV  42.02590886 6.272136003  6.700414
   K   0.05797157 0.005910154  9.808809
. . .
sigma: 10.58511
```

```
df:
[1] 3 20
. . .
             Vm        delV          K
  Vm  1.0000000 -0.54055820 0.61128147
delV -0.5405582  1.00000000 0.06440645
   K  0.6112815  0.06440645 1.00000000

formula:
vel  ~ ((Vm + delV * (state == "treated")) * conc)/(K + conc)
> combinedSS <- sum(Purfit.1$res^2) + sum(Purfit.2$res^2)
> Fval <- (sum(Purboth$res^2) - combinedSS)/(combinedSS/19)
> Fval
[1] 1.718169
> 1 - pf(Fval, 1, 19)
[1] 0.2055523
```

Using a single K appears to be reasonable.

Fitting a Model to the Ping-Pong Data

The example here develops a model based only on the difference in ratings, ignoring, for the moment, the average rating. The model to fit is:

```
~ DV * alpha + log( 1 + exp(DV * alpha) )
```

There are four stages to the development of the model.

1. Estimating Starting Values

A very crude initial estimate for `alpha` can be found as follows:

- Replace all the differences in ratings by $\pm\bar{d}$, where $\bar{d}$ is the mean difference.
- For each match, the probability from the model that the winner had a higher rating satisfies: $\bar{d}$ `* a = log(p/(1-p))`
- Solve for α by substituting for p the observed frequency with which the player with the higher rating wins.

The computations in S-PLUS proceed as follows:

```
> param(pingpong, "p") <- 0 # make pingpong a "pframe"
> attach(pingpong,1)
> DV <- winner - loser
> p <- sum(winner > loser) /length(winner)
> p
[1] 0.8223401
```

```
> alpha <- log(p/(1-p))/mean(DV)
> alpha
[1] 0.007660995
> detach(1, save = "pingpong")
```

2. A Simple Model

Recall the `lprob` function which calculates the log-likelihood for the ping-pong problem.

```
> lprob
function(lp)
log(1 + exp(lp)) - lp
```

The model is fitted as follows:

```
> attach(pingpong)
> fit.alpha <- ms( ~ lprob( DV * alpha ),
+ start = list(alpha=0.0077))
> fit.alpha
objective: 1127.635

parameters:
     alpha
0.01114251

gradient:
[1] -0.0004497159

RELATIVE FUNCTION CONVERGENCE.
formula:  ~ lprob(DV * alpha)
```

3. Adding The Gradient

To fit the model with the gradient added to the formula use `lprob2`.

```
> fit.alpha.2 <- ms( ~ lprob2( DV * alpha, DV),
+ start = list(alpha=0.0077))
> fit.alpha.2
objective: 1127.635

parameters:
     alpha
0.01114251

gradient:
       alpha
```

```
2.938858e-07

RELATIVE FUNCTION CONVERGENCE.
formula:  ~ lprob2(DV * alpha, DV)
```

Even for this simple problem, providing the derivative has decreased the computation time by 20%.

4. Adding The Hessian

To fit the model with the gradient and the Hessian added to the formula use `lprob3`.

```
> fit.alpha.3 <- ms( ~ lprob3(DV*alpha, DV),
+ pingpong, start = list(alpha = .0077))
> fit.alpha.3
objective: 1127.635

parameters:
     alpha
0.01114251

gradient:
       alpha
-0.000218718

BOTH X- AND RELATIVE FUNCTION CONVERGENCE
formula:  ~ lprob3(DV * alpha, DV)
```

Profiling the Objective Function

Profiling provides a more accurate picture of the uncertainty in the parameter estimates than simple standard errors. When there are only two parameters, contours of the objective function can be plotted by generating a grid of values. When there are more than two parameters, examination of the objective function is usually done in one of two ways:

- *slices* Fixing all but two of the parameters at their *estimated* values and creating a grid of the objective function by varying the remaining two parameters of interest.
- *projections* Vary two parameters of interest over fixed values, optimizing the objective function over the other parameters.

Two-dimensional projections are often too time consuming to compute. One-dimensional projections are called profiles. Profiles are plots of a t-statistic equivalent called the profile t function for a parameter of interest against a range of values for the parameter.

The Profile t Function

For `nls`, the profile t function for a given parameter, θ_p, is denoted by $\tau(\theta_p)$ and is computed as follows:

$$\tau(\theta_p) = sign(\theta_p - \tilde{\theta}_p)\sqrt{\frac{\tilde{S}(\theta_p) - S(\tilde{\theta})}{s}} \tag{12.16}$$

where: $\tilde{\theta}_p$ is the model estimate of θ_p, $\tilde{S}(\theta_p)$ is the sum of squares based on optimizing all parameters except θ_p, which is fixed, and $S(\tilde{\theta})$ is the sum of squares based on optimizing all parameters.

The profile t function is directly related to confidence intervals for the corresponding parameter. $\tau(\theta_p)$ can be shown to be equivalent to the studentized parameter

$$\delta(\theta_p) = \frac{\theta_p - \tilde{\theta}_p}{se(\tilde{\theta}_p)} \tag{12.17}$$

for which a $1-\alpha$ confidence interval can be constructed as follows:

$$-t\left(N-P;\frac{a}{2}\right) \le \delta(\theta_p) \le t\left(N-P;\frac{a}{2}\right). \tag{12.18}$$

The profile Function in S-PLUS

The profile function produces profiles for "`nls`" and "`ms`" objects. Profiles show confidence intervals for parameters as well as the nonlinearity of the objective function. If the model were linear the profile would be a straight line through the origin with a slope of one. You can produce the profile plots for the Puromycin fit `Purboth` as follows:

```
> Purboth.prof <- profile(Purboth)
> plot(Purboth.prof)
```

The "`profile`" object returned by `profile` has a component for each parameter containing the evaluations of the profile t function plus some additional attributes. The component for the `Vm` parameter is:

```
> Purboth.prof$Vm
          tau par.vals.Vm par.vals.delV par.vals.K
 1 -3.9021051    144.6497      54.60190 0.04501306
 2 -3.1186052    148.8994      52.07216 0.04725929
 3 -2.3346358    153.2273      49.54358 0.04967189
 4 -1.5501820    157.6376      47.01846 0.05226722
```

```
 5 -0.7654516    162.1334      44.50315 0.05506789
 6  0.0000000    166.6040      42.02591 0.05797157
 7  0.7548910    171.0998      39.57446 0.06103225
 8  1.5094670    175.6845      37.12565 0.06431820
 9  2.2635410    180.3616      34.67194 0.06783693
10  3.0171065    185.1362      32.20981 0.07160305
11  3.7701349    190.0136      29.73812 0.07563630
12  4.5225948    194.9997      27.25599 0.07995897
```

Figure 12.2 shows profile plots for the three-parameter Puromycin fit. Each plot shows the profile t function, (τ), when the parameter on the x-axis ranges over the values shown, and the other parameters are optimized. The surface is quite linear with respect to these three parameters.

Computing Confidence Intervals

An example of a simple function to compute the confidence intervals from the output of `profile` follows:

```
> conf.int <- function(profile.obj, variable.name,
+ confidence.level = 0.95)
+ {if(is.na(match(variable.name, names(profile.obj))))
+ stop(paste("Variable", variable.name, "not in the model"))
+ resid.df <- attr(profile.obj, "summary")[["df"]][2]
+ tstat <- qt(1 - (1 - confidence.level)/2, resid.df)
+ prof <- profile.obj[[variable.name]]
+ approx(prof[, "tau"], prof[, "par.vals"][, variable.name],
+ c(-tstat, tstat))[[2]]
+ }
```

The tricky line in `conf.int` is the last one which calls `approx`. `Purboth.prof$Vm` is a data frame with two components (columns). The first component is the vector of τ values which we pick off using `prof[, "tau"]`. The second component named `par.vals` contains a matrix with as many columns as there are parameters in the model. This results in the strange looking subscripting given by `prof[, "par.vals"][, variable.name]`. The first subscript removes the matrix from the `par.vals` component, and the second subscript removes the appropriate column. Three examples using `conf.int` and the profile object `Purboth.prof` follow:

```
> conf.int(Purboth.prof, "delV", conf = .99)
[1] 24.20945 60.03857
> conf.int(Purboth.prof, "Vm", conf = .99)
[1] 150.4079 184.0479
```

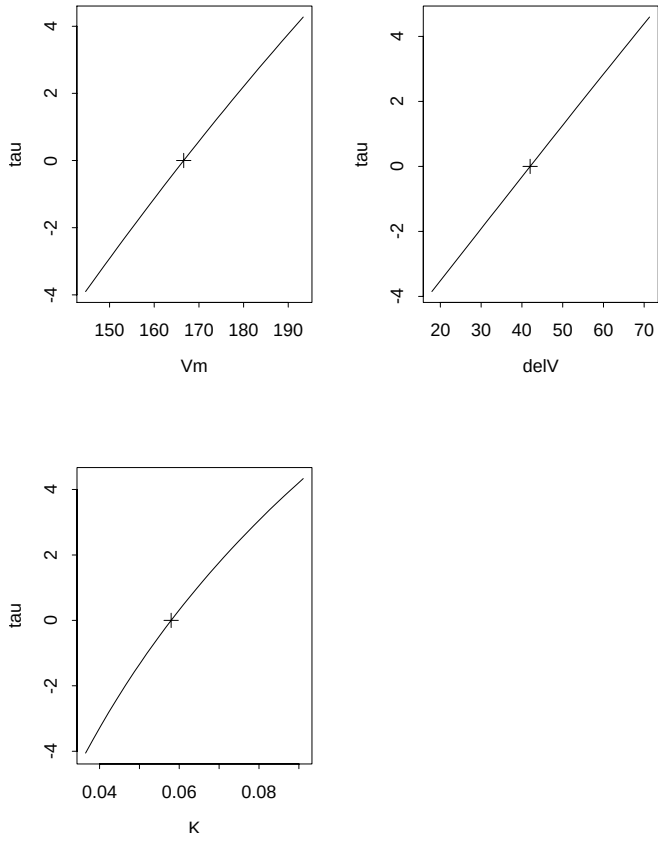

Figure 12.2: *The profile plots for the Puromycin fit.*

```
> conf.int(Purboth.prof, "K", conf = .99)
[1] 0.04217613 0.07826822
```

The `conf.int` function can be improved by, for example, doing a cubic spline interpolation rather than the linear interpolation that `approx` does. A marginal confidence interval computed from the profile t function is exact, disregarding any approximations due to interpolation, whereas the marginal confidence intervals produced by using the coefficient and its standard error from the summary of the fit is only a linear approximation.

DESIGNED EXPERIMENTS AND ANALYSIS OF VARIANCE 13

Use S-PLUS to analyse data from experiments with one, two, or more factors.

DESIGNED EXPERIMENTS AND ANALYSIS OF VARIANCE

This chapter discusses how to analyze designed experiments. Typically, the data have a numeric response and one or more categorical variables (*factors*) that are under the control of the experimenter. For example, an engineer may measure the yield of some process using each combination of four catalysts and three specific temperatures. This experiment has two factors, catalyst and temperature, and the response is the yield.

Traditionally, the analysis of experiments has centered on the performance of an Analysis of Variance (ANOVA). In more recent years graphics have played an increasingly important role. There is a large literature on the design and analysis of experiments—Box, Hunter, and Hunter (1978) is an example.

This chapter consists of sections which show you how to use S-PLUS to analyze experimental data for each of the following situations:

- Experiments with one factor (section 13.1)
- Experiments with two factors and a single replicate (section 13.2)
- Experiments with two factors and two or more replicates (section 13.3)
- Experiments with many factors at two levels: 2^k designs (section 13.4)

Each of these sections stands alone. You can read whichever section is appropriate to your problem, and get the analysis done without having to read the other sections. The examples used in sections 13.1–13.4 are from Box, Hunter, and Hunter (1978). Thus, this chapter is a useful supplement in a course which covers the material of Chapters 6, 7, 9, 10, 11 of Box, Hunter, and Hunter.

Setting Up the Data Frame

In analyzing experimental data using S-PLUS, the first thing you do is set up an appropriate *data frame* for your experimental data. You may think of the data frame as a matrix, with the columns containing values of the *variables*. Each row of the data frame contains an observed value of the response (or responses), and the corresponding values of the experimental factors.

A First Look at the Data

Use the functions `plot.design`, `plot.factor` and possibly `interaction.plot` to graphically explore your data.

The Model and Analysis of Variance

It is important that you have a clear understanding of exactly what model is being considered when you carry out the analysis of variance. Use `aov` to carry out the analysis of variance, and use `summary` to display the results.

In using `aov`, you use *formulas* to specify your *model*. The examples in this chapter introduce you to simple uses of formulas. You may supplement your understanding of how to use formulas in S-PLUS by reading chapter 2, Specifying Models in S-PLUS, or chapter 2, Statistical Models, and chapter 5, Analysis of Variance; Designed Experiments, in Chambers and Hastie (1992).

Diagnostic Plots

For each analysis, you should make the following minimal set of plots to convince yourself that the model being entertained is adequate:

- Histogram of residuals (using `hist`)
- Normal qqplot of residuals (using `qqnorm`)
- Plot of residuals versus fit (using `plot`)

When you know the time order of the observations, you should also make plots of the original data and the residuals in the time order in which the data were collected.

The diagnostic plots may indicate inadequacies in the model from one or more of the following sources: existence of interactions, existence of outliers, and existence of inhomogeneous error variance.

13.1 EXPERIMENTS WITH ONE FACTOR

The simplest kind of experiments are those in which a single continuous *response* variable is measured a number of times for each of several *levels* of some experimental *factor*.

For example, consider the data in table 13.1 (from Box, Hunter, and Hunter (1978)), which consists of numerical values of "blood coagulation times" for each of four diets. Coagulation time is the continuous response variable, and diet is a *qualitative* variable, or *factor*, having four levels: A, B, C, and D. The diets corresponding to the levels A, B, C, and D were determined by the experimenter.

Your main interest is to see whether or not the factor "diet" has any effect on the mean value of blood coagulation time. The experimental factor, "diet" in this case, is often called the *treatment*.

Formal statistical testing for whether or not the factor level affects the mean is carried out using the method of analysis of variance (ANOVA). This needs to be complemented by exploratory graphics to provide confirmation that the model assumptions are sufficiently correct to validate the formal ANOVA

Table 13.1: *Blood coagulation times for four diets.*

Diet			
A	B	C	D
62	63	68	56
60	67	66	62
63	71	71	60
59	64	67	61
	65	68	63
	66	68	64
			63
			59

conclusion. S-PLUS provides tools for you to do both the data exploration and formal ANOVA.

Setting Up the Data Frame

In order to analyze the data, you need to get it into a form that S-PLUS can use for the analysis of variance. You do this by setting up a *data frame*. First create a numeric vector `coag`:

```
> coag <- scan()
1: 62 60 63 59
5: 63 67 71 64 65 66
11: 68 66 71 67 68 68
17: 56 62 60 61 63 64 63 59
25:
```

Next, create a factor called `diet`, that corresponds to `coag`:

```
> diet <- factor(rep(LETTERS[1:4],c(4,6,6,8)))
> diet
[1] A A A A B B B B B B C C C C C C D D D D D D D D
```

Now create a data frame with columns `diet` and `coag`:

```
> coag.df <- data.frame(diet,coag)
```

The data frame object `coag.df` is a matrix-like object, so it looks like a matrix when you display it on your screen:

```
> coag.df
   diet coag
 1    A   62
 2    A   60
 3    A   63
     .
     .
     .
23    D   63
24    D   59
```

A First Look at the Data

For each level of the treatment factor, you make an initial graphical exploration of the response data y_{ij} by using the functions `plot.design` and `plot.factor`.

You can make plots of the treatment means and treatment medians for each level of the experimental factor `diet` by using the function `plot.design` twice, as follows:

```
> par(mfrow=c(1,2))
> plot.design(coag.df)
> plot.design(coag.df, fun= median)
> par(mfrow=c(1,1))
```

The results are shown in the two plots of figure 13.1. In the left hand plot, the tick marks on the vertical line are located at the treatment means for the diets A, B, C, and D, respectively. The mean values of coagulation time for diets A and D happen to have the same value, 61, and so the labels A and D are overlaid. The horizontal line, located at 64, indicates the overall mean of all the data.

In the right hand plot of figure 13.1, medians rather than means are indicated. There is not much difference between the treatment means and the treatment medians, so you should not be too concerned about adverse effects due to outliers.

The function `plot.factor` produces a boxplot of the response data for each level of the experimental factor:

```
> plot.factor(coag.df)
```

The resulting plot is shown in figure 13.2. This plot indicates that the responses for diets A and D are quite similar, while the median responses for diets B and C are considerably larger relative to the variability reflected by the heights of the boxes. Thus, you suspect that diet has an effect on blood coagulation time.

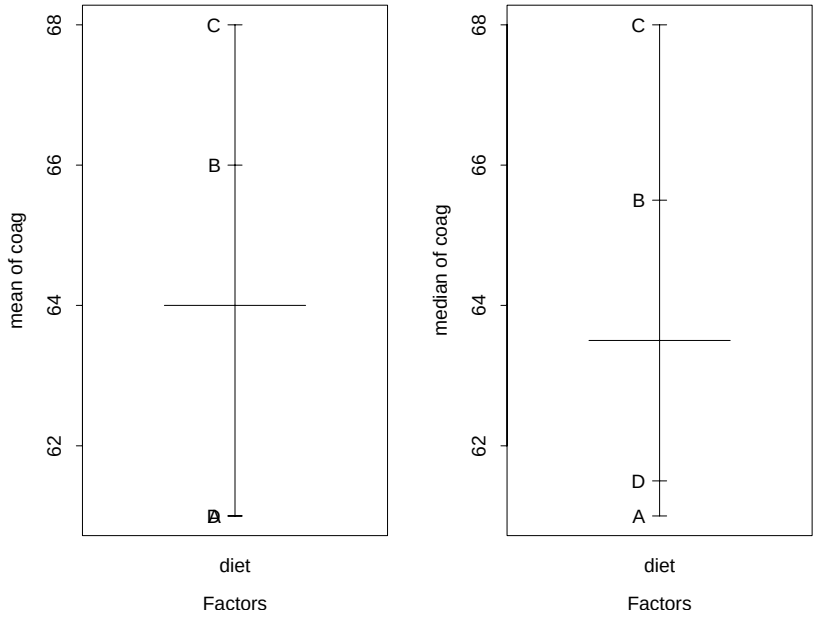

Figure 13.1: *Treatment means and medians.*

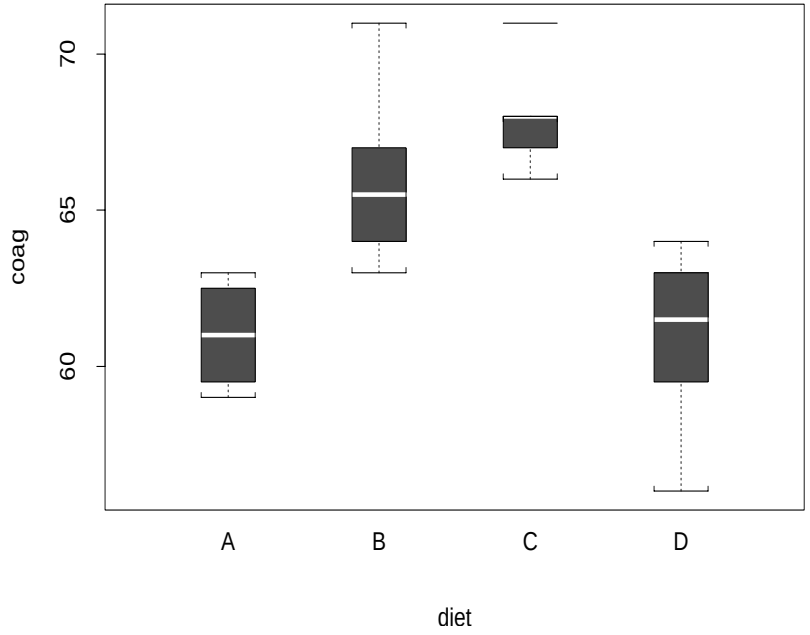

Figure 13.2: *Boxplots for each treatment.*

If the exploratory graphical display of the response using `plot.factor` indicates that the interquartile distance of the boxplots depends upon the median, then a transformation to make the error variance constant is called for. The transformation may be selected with a "spread versus level" plot. See, for example, section 13.3, or Hoaglin, Mosteller, and Tukey (1983).

The One-Way Layout Model and Analysis of Variance

The classical model for experiments with a single factor is

$$y_{ij} = \mu_i + \varepsilon_{ij} \qquad \begin{aligned} j &= 1, \ldots, J_i \\ i &= 1, \ldots, I \end{aligned}$$

where μ_i is the mean value of the response for the ith level of the experimental factor. There are I levels of the experimental factor, and J_i measurements $y_{i1}, y_{i2}, \ldots, y_{iJ_i}$ are taken on the response variable for level i of the experimental factor.

Using the treatment terminology, there are I treatments, and μ_i is called the ith treatment mean. The above model is often called the *one-way layout* model. For the blood coagulation experiment there are I=4 diets, and the means μ_1, μ_2, μ_3, and μ_4 correspond to diets A, B, C, and D, respectively. The numbers of observations are $J_A = 4$, $J_B = J_C = 6$, and $J_D = 8$.

You carry out the analysis of variance with the function `aov`:

```
> aov.coag <- aov(coag ~ diet, coag.df)
```

The first argument to `aov` above is the *formula* `coag ~ diet`. This formula is a symbolic representation of the one-way layout model equation; the formula excludes the error term ε_{ij}. The second argument to `aov` is the data frame you created, `coag.df`, which provides the data needed to carry out the ANOVA. The names `diet` and `coag`, used in the formula `coag ~ diet`, need to match the names of the variables in the data frame `coag.df`.

To display the ANOVA table, use `summary`:

```
> summary(aov.coag)
            Df Sum of Sq Mean Sq F Value       Pr(F)
      diet  3       228    76.0 13.5714 4.65847e-05
 Residuals 20       112     5.6
```

The p-value is equal to .000047, which is highly significant.

Diagnostic Plots

You obtain the fitted values and residuals using the `fitted.values` and `residuals` functions on the result of `aov`. Thus, for example, you get the fitted values with the following:

```
> fitted.values(aov.coag)
  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17
 61 61 61 61 66 66 66 66 66 66 68 68 68 68 68 68 61

 18 19 20 21 22 23 24
 61 61 61 61 61 61 61
```

The `resid` and `fitted` functions are shorter names for `residuals` and `fitted.values`, respectively.

You can check the residuals for distributional shape and outliers by using `hist` and `qqnorm`, with the residuals component of `aov.coag` as argument:

```
> hist(resid(aov.coag))
> qqnorm(resid(aov.coag))
```

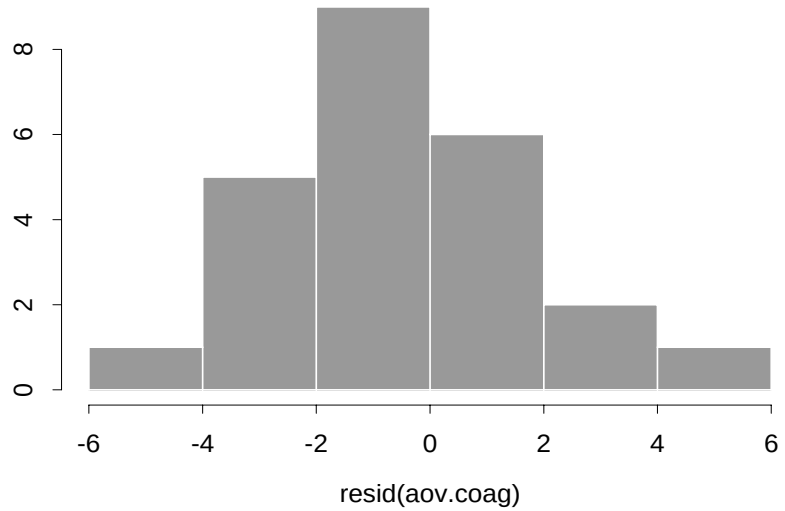

Figure 13.3: *Histogram of residuals.*

Figure 13.3 shows the resulting histogram and figure 13.4 shows the resulting quantile-quantile plot.

The shape of the histogram, and the linearity of the normal QQ-plot, both indicate that the error distribution is quite Gaussian. (The flat sections in the QQ-plot are a consequence of tied values in the data.)

You can check for inhomogeneity of error variance and possible outliers by plotting the residuals versus the fit:

```
> plot(fitted(aov.coag), resid(aov.coag))
```

This plot reveals no unusual features, and is not shown.

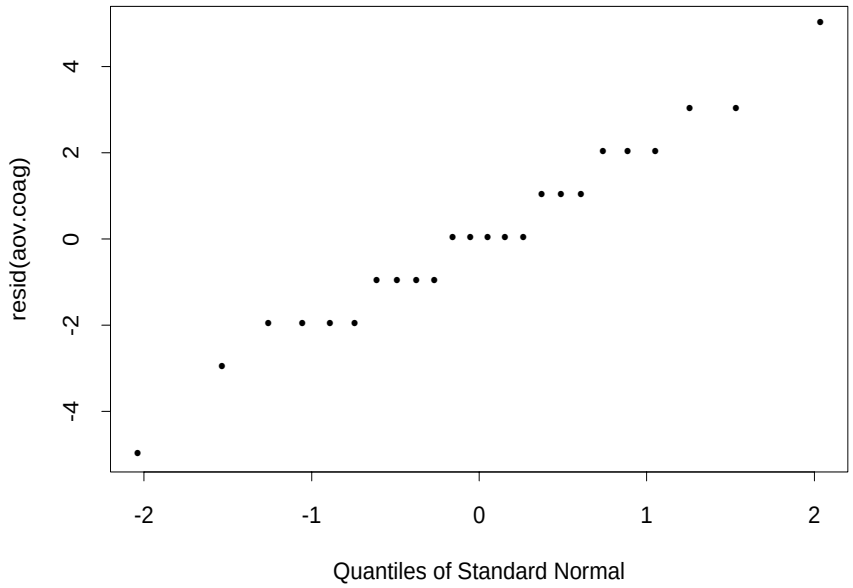

Figure 13.4: *Normal QQ-plot of residuals.*

Details

An alternate form of the one-way layout model is the *overall mean* plus *effects* form

$$y_{ij} = \mu + \alpha_i + \varepsilon_{ij}$$

where μ is the overall mean, and α_i is the effect for level (or treatment) i. The mean μ_i for level (or treatment) i in the first form of the model is related to μ and α_i by

$$\mu_i = \mu + \alpha_i$$

and the effects a$_i$ satisfy the constraint

$$\alpha_1 + \alpha_2 + \cdots + \alpha_I = 0\,.$$

The function `aov` fits the one-way model in the "overall mean plus effects" form $y_{ij} = \hat{\mu} + \hat{\alpha}_i + r_{ij}$. See section 14.1 for more on this.

To obtain the effects, use `model.tables` as follows:

```
> model.tables(aov.coag)
Refitting model to allow projection
Tables of effects
```

```
 diet
     A B C  D
    -3 2 4 -3
 rep 4 6 6 8
```

You can get the treatment means as follows:

```
> model.tables(aov.coag, type="means")
Refitting model to allow projection
Tables of means

Grand mean

 64

 diet
     A  B  C  D
    61 66 68 61
rep  4  6  6  8
```

13.2 THE UNREPLICATED TWO-WAY LAYOUT

The data in table 13.2 (used by Box, Hunter, and Hunter (1978)) were collected to determine the effect of treatments A, B, C, and D on the yield of penicillin in a penicillin manufacturing process.

Table 13.2: *Effect of four treatments on penicillin yield.*

	Treatment			
Block	A	B	C	D
Blend 1	89	88	97	94
Blend 2	84	77	92	79
Blend 3	81	87	87	85
Blend 4	87	92	89	84
Blend 5	79	81	80	88

The values of the response variable “yield” are the numbers in the table, and the columns of the table correspond to the levels A, B, C, and D of the treatment factor. There was a second factor, namely the blend factor, since a

separate blend of the corn-steep liquor had to be made for each application of the treatments.

Your main interest is in determining whether the treatment factor affects yield. The blend factor is of only secondary interest; it is a "blocking" variable introduced to increase the sensitivity of the inference for treatments. The order of the treatments within blocks was chosen at random. Hence, this is a *randomized blocks* experiment.

The methods we use in this section applies equally well to two-factor experiments in which both factors are experimentally controlled and of equal interest.

Setting Up the Data Frame

Table 13.2 is *balanced*—each entry or *cell* of the table (i.e., each row and column combination) has the same number of observations (one observation per cell, in the present example)—so you can use `fac.design` to create the data frame.

First, create a list `fnames` with two components named `blend` and `treatment`, where `blend` contains the level names of the blend factor and `treatment` contains the level names of the treatment factor:

```
> fnames <- list(blend=paste("Blend ", 1:5),
+           treatment=LETTERS[1:4])
```

Then use `fac.design` to create the *design* data frame `pen.design`

```
> pen.design <- fac.design(c(5,4), fnames)
```

The first argument, `c(5,4)`, to `fac.design` specifies the design as having two factors because its length is two. The `5` specifies five levels for the first factor, blend, and the `4` specifies four levels for the second factor, treatment. The second argument, `fnames`, specifies the factor names and the labels for their levels.

The design data frame `pen.design` that you just created contains the factors `blend` and `treatment` as its first and second columns, respectively.

Now create `yield` to match `pen.design`:

```
> yield <- scan()
1: 89 84 81 87 79
6: 88 77 87 92 81
11: 97 92 87 89 80
16: 94 79 85 84 88
21:
```

You can now use `data.frame` to combine the design data frame `pen.design` and the response `yield` into the data frame `pen.df`:

```
> pen.df <- data.frame(pen.design,yield)
```

Now look at `pen.df`:

```
> pen.df
      blend treatment yield
 1 Blend 1         A    89
 2 Blend 2         A    84
 3 Blend 3         A    81
 4 Blend 4         A    87
 5 Blend 5         A    79
 6 Blend 1         B    88
               .
               .
               .
19 Blend 4         D    84
20 Blend 5         D    88
```

Alternatively, you could build the model data frame directly from `pen.design` as follows:

```
> pen.design[,"yield"] <- yield
```

When you plot the object `pen.design`, S-PLUS uses the method `plot.design`, because the object `pen.design` is of class `design`. Thus, you obtain the same results as if you called `plot.design` explicitly on the object `pen.df`.

A First Look at the Data

You can look at the (comparative) values of the sample means of the data for each level of each factor using `plot.design`:

```
> plot.design(pen.df)
```

This function produces the plot shown in figure 13.5. For the blend factor, each tick mark is located at the mean of the corresponding row of table 13.2. For the treatment factor, each tick mark is located at the mean of the corresponding column of table 13.2. The horizontal line is located at the sample mean of all the data. Figure 13.5 suggests that the blend has a greater effect on yield than does the treatment.

Since sample medians are insensitive to outliers, and sample means are not, you may want to make a plot similar to figure 13.5 using sample medians instead of sample means. You can do this with `plot.design`, using the second argument `fun=median`:

```
> plot.design(pen.df, fun=median)
```

In this case, the plot does not indicate great differences between sample means and sample medians.

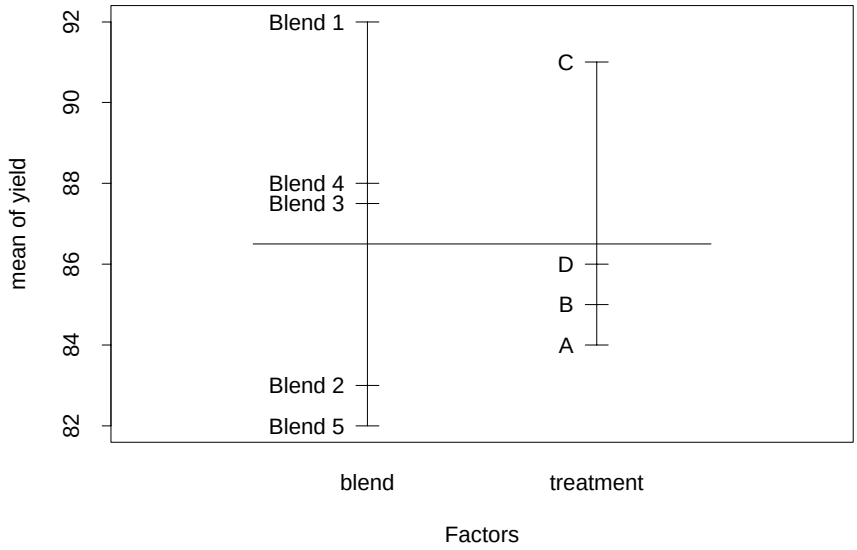

Figure 13.5: *Sample means in penicillin yield experiment.*

Use `plot.factor` to get a more complete exploratory look at the data. But first use `par` to get a one row by two column layout for two plots:

```
> par(mfrow=c(1,2))
> plot.factor(pen.df)
> par(mfrow=c(1,1))
```

This command produces the plot shown in figure 13.6 .

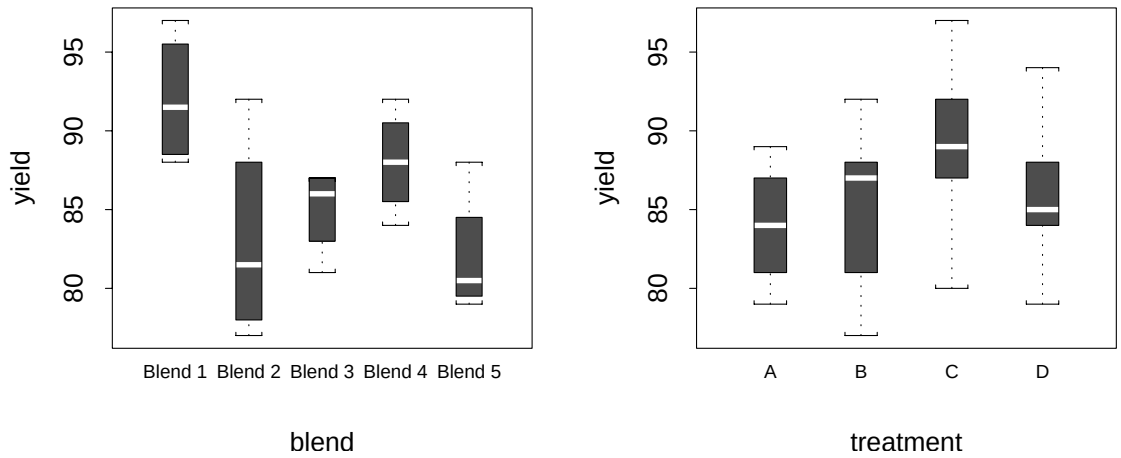

Figure 13.6: *Factor plot for penicillin yield experiment.*

The boxplots for factors, produced by `plot.factor`, give additional information about the data besides the location given by `plot.design`. The boxplots indicate variability, skewness, and outliers in the response, for each fixed level of each factor. For this particular data, the boxplots for both blends and treatments indicate rather constant variability, relatively little overall skewness, and no evidence of outliers.

For two-factor experiments, you should use `interaction.plot` to check for possible interactions (i.e., non-additivity). The `interaction.plot` function does not accept a data frame as an argument. Instead, you must supply appropriate factor names and the response name. To make these factor and response data objects available to `interaction.plot`, you must first "attach" the data frame `pen.df`:

```
> attach(pen.df)
> interaction.plot(treatment,blend,yield)
```

These commands produce the plot shown in figure 13.7.

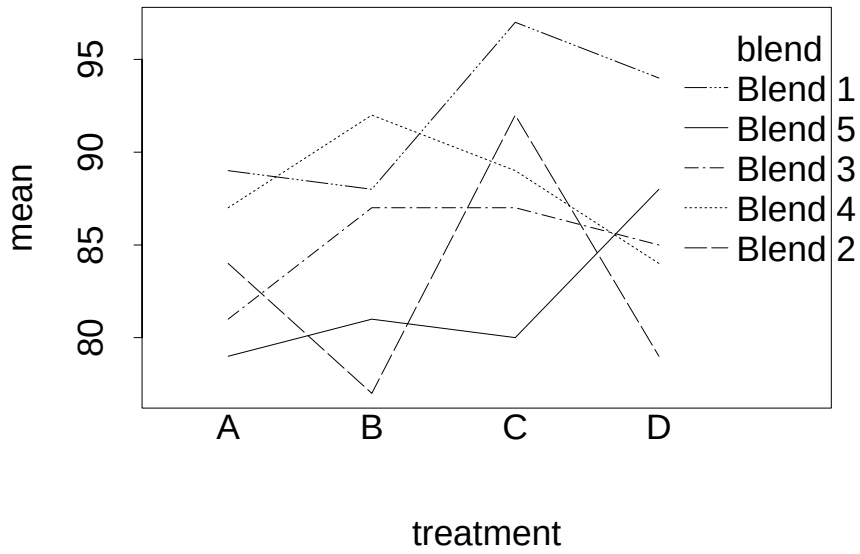

Figure 13.7: *Interaction plot of penicillin experiment.*

The first argument to `interaction.plot` specifies which factor appears along the *x*-axis (in this case, `treatment`). The second argument specifies which factor is associated with each line plot, or "trace" (in this case, `blend`). The third argument is the response variable (in this case, `yield`).

Without replication it is often difficult to interpret an interaction plot since random error tends to dominate. There is nothing striking in this plot.

The Two-Way Model and ANOVA (One Observation per Cell)

The *additive* model for experiments with two factors, A and B, and one observation per cell is:

$$y_{ij} = \mu + \alpha_i^A + \alpha_i^B + \varepsilon_{ij} \qquad \begin{aligned} i &= 1, \ldots, I \\ j &= 1, \ldots, J \end{aligned}$$

where μ is the overall mean, α_i^A is the effect of the ith level of factor A and α_j^B is the effect of the jth level of factor B.

For the penicillin data above, factor A is "blend" and factor B is "treatment." Blend has $I = 5$ levels and treatment has $J = 4$ levels.

To estimate the *additive* model, use `aov`:

```
> aov.pen <- aov(yield ~ blend + treatment, pen.df)
```

The formula `yield ~ blend + treatment` specifies that a two factor additive model is fit, with `yield` the response, and `blend` and `treatment` the factors.

Display the analysis of variance table with `summary`:

```
> summary(aov.pen)
            Df Sum of Sq Mean Sq F Value    Pr(F)
     blend   4       264 66.0000 3.50442 0.040746
 treatment   3        70 23.3333 1.23894 0.338658
 Residuals  12       226 18.8333
```

The p-value for blend is moderately significant, while the p-value for treatment is insignificant.

Diagnostic Plots

Make a histogram of the residuals

```
> hist(resid(aov.pen))
```

The resulting histogram is shown in figure 13.8. Now make a normal QQ-plot of residuals:

```
> qqnorm(resid(aov.pen))
```

The resulting plot is shown in figure 13.9. The central four cells of the histogram in figure 13.8 are consistent with a fairly normal distribution in the middle. The linearity of the normal QQ-plot in figure 13.9, except near the ends, also suggests that the distribution is normal in the middle. The relatively larger values of the outer two cells of the histogram, and the flattening of the normal QQ-plot near the ends, both suggest that the error distribution is slightly more short-tailed than a normal distribution. This is not a matter of great concern for the ANOVA F tests.

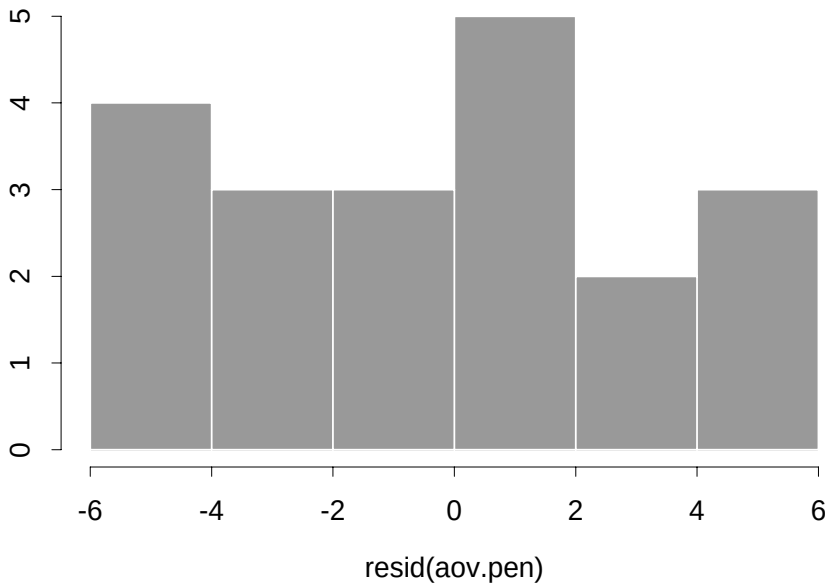

Figure 13.8: *Histogram of residuals for penicillin yield experiment.*

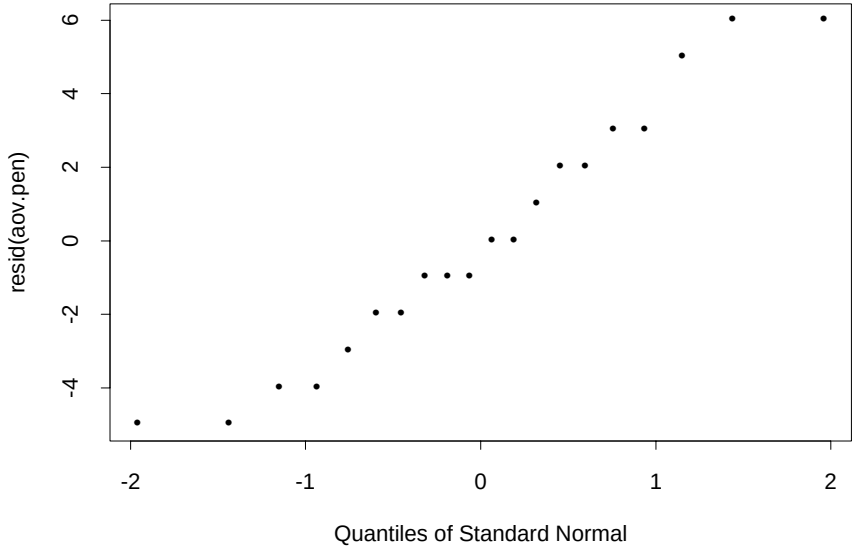

Figure 13.9: *Quantile-Quantile plot of residuals for penicillin yield experiment.*

Make a plot of residuals versus the fit:

```
> plot(fitted(aov.pen), resid(aov.pen))
```

The resulting plot is shown in figure 13.10.

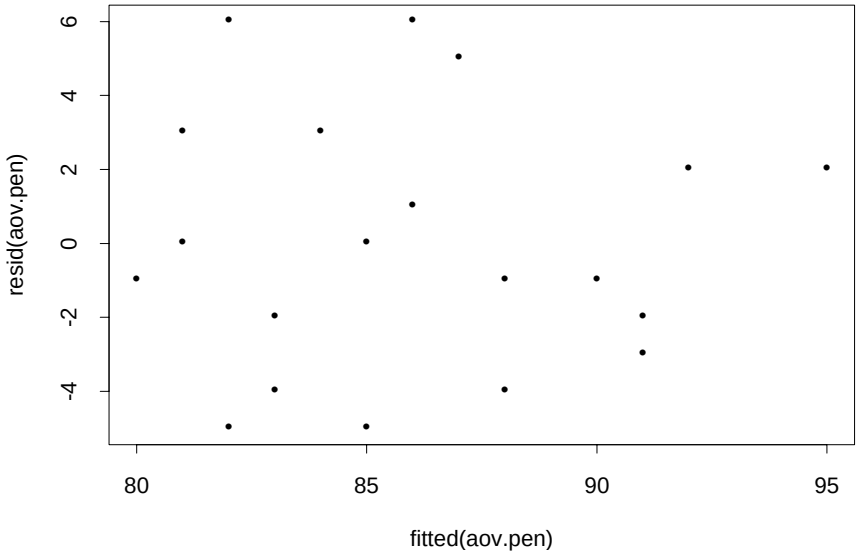

Figure 13.10: *Residuals vs. fitted values for penicillin yield experiment.*

The plot of residuals versus fit gives some slight indication that smaller error variance is associated with larger values of the fit.

Guidance

Since there is some indication of inhomogeneity of error variance, we now consider transforming the response, `yield`.

You may want to test for the existence of a multiplicative interaction, specified by the model

$$y_{ij} = \mu + \alpha_i^A + \alpha_j^B + \theta\alpha_i^A\alpha_j^B + \varepsilon_{ij}.$$

When the unknown parameter θ is not zero, multiplicative interaction exists. A test for the null hypothesis of no interaction may be carried out using the test statistic T_{1df} for Tukey's one degree of freedom for non-additivity.

An S-PLUS function, `tukey.1`, is provided in the Details section. You can use it to compute T_{1df} and the *p*-value. For the penicillin data:

```
> tukey.1(aov.pen, pen.df)
$T.1df:
[1] 0.09826791
```

```
$p.value:
[1] 0.7597822
```

The statistic $T_{1df} = .098$ has a p-value of $p = .76$, which is not significant. Therefore there is no indication of a multiplicative interaction.

Assuming that the response values are positive, you can find out whether or not the data suggest a specific transformation to remove multiplicative interaction as follows: Plot the residuals r_{ij} for the additive fit versus the *comparison values*

$$c_{ij} = \frac{\hat{\alpha}_i^A \hat{\alpha}_j^B}{\hat{\mu}}.$$

If this plot reveals a linear relationship with estimated slope $\hat{\theta}$, then you should analyze the data again, using as new response values the power transformation y_{ij}^{λ} of the original response variables y_{ij}, with exponent

$$\lambda = 1 - \hat{\theta}.$$

(If $\lambda = 0$, use $\log(y_{ij})$.) See Hoaglin, Mosteller, and Tukey (1983) for details.

An S-PLUS function called `comp.plot`, for computing the comparison values c_{ij}, plotting r_{ij} versus c_{ij}, and computing $\hat{\theta}$, is provided in the Details section below. Applying `comp.plot` to the penicillin data gives the following result:

```
> comp.plot(aov.pen, pen.df)
$theta.hat:
[1] 4.002165

$std.error:
[1] 9.980428

$R.squared:
          R2
 0.008854346
```

In this case the estimated slope is $\hat{\theta} = 4$, which gives $\lambda = -3$. However, this is not a very sensible exponent for a power transformation. The standard deviation of $\hat{\theta}$ is nearly 10 and the R^2 is only .009, which indicates that θ may be zero. Thus we do not recommend using a power transformation.

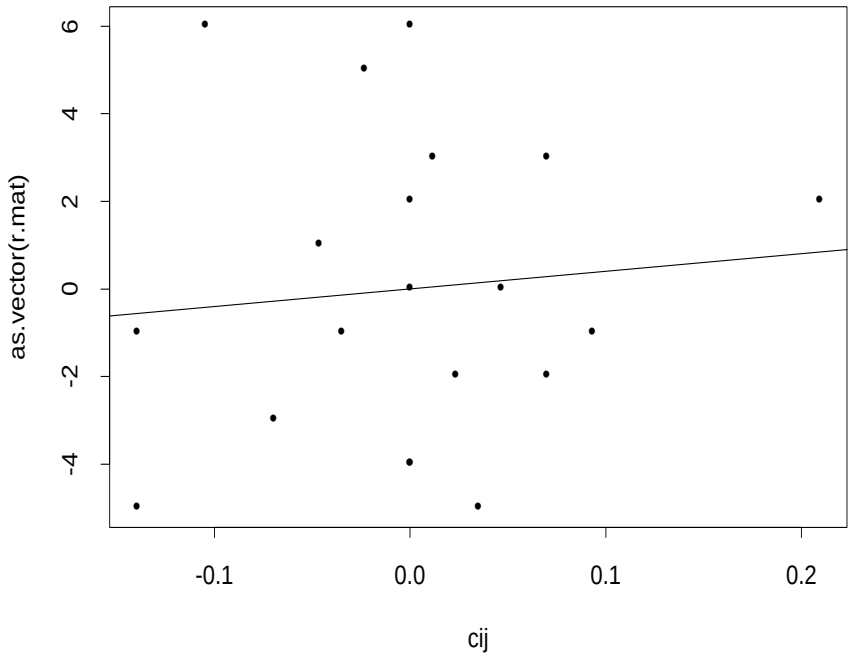

Figure 13.11: *Display from* `comp.plot`.

Details

The test statistic T_{1df} for Tukey's one degree of freedom is given by:

$$T_{1df} = (IJ - I - J)\frac{SS_{\theta}}{SS_{res.1}}$$

where

$$SS_{\theta} = \frac{\left(\sum_{i=1}^{I}\sum_{j=1}^{J}\hat{\alpha}_i^A \hat{\alpha}_j^B y_{ij}\right)^2}{\sum_{i=1}^{I}(\hat{\alpha}_i^A)^2 \sum_{j=1}^{J}(\hat{\alpha}_j^B)^2}$$

$$SS_{res.1} = SS_{res} - SS_{\theta}$$

$$SS_{res} = \sum_{i=1}^{I}\sum_{j=1}^{J} r_{ij}^2$$

with the $\hat{\alpha}_i^A$, $\hat{\alpha}_j^B$ the additive model estimates of the α_i^A and α_j^B, and r_{ij} the residuals from the additive model fit. The statistic T_{1df} has an $F_{1,IJ-I-J}$ distribution.

Here is a function `tukey.1` to compute the Tukey one-degree of freedom for non-additivity test. You can create your own version of this function by typing `tukey.1 <-` and then the definition of the function.

```
> tukey.1
function(aov.obj, data)
{
        vnames <- names(aov.obj$contrasts)
        if(length(vnames) != 2)
                stop("the model must be two-way")
        vara <- data[, vnames[1]]
        varb <- data[, vnames[2]]
        na <- length(levels(vara))
        nb <- length(levels(varb))
        resp <- data[, as.character(attr(aov.obj$terms,
          "variables")[attr(aov.obj$terms, "response" )])]
        cfs <- coef(aov.obj)
        alpha.A <- aov.obj$contrasts[[vnames[1]]] %*% cfs[
                aov.obj$assign[[vnames[1]]]]
        alpha.B <- aov.obj$contrasts[[vnames[2]]] %*% cfs[
                aov.obj$assign[[vnames[2]]]]
        r.mat <- matrix(0, nb, na)
        r.mat[cbind(as.vector(unclass(varb)), as.vector(
                unclass(vara)))] <- resp
        SS.theta.num <- sum((alpha.B %*% t(alpha.A)) *
                r.mat)^2
        SS.theta.den <- sum(alpha.A^2) * sum(alpha.B^2)
        SS.theta <- SS.theta.num/SS.theta.den
        SS.res <- sum(resid(aov.obj)^2)
        SS.res.1 <- SS.res - SS.theta
        T.1df <- ((na * nb - na - nb) * SS.theta)/SS.res.1
        p.value <- 1 - pf(T.1df, 1, na * nb - na - nb)
        list(T.1df = T.1df, p.value = p.value)
}
```

Here is a function `comp.plot` for computing a least-squares fit to the plot of residuals versus comparison values:

```
> comp.plot
function(aov.obj, data)
{
        vnames <- names(aov.obj$contrasts)
        if(length(vnames) != 2)
               stop("the model must be two-way")
        vara <- data[, vnames[1]]
        varb <- data[, vnames[2]]
        cfs <- coef(aov.obj)
        alpha.A <- aov.obj$contrasts[[vnames[1]]] %*% cfs[
               aov.obj$assign[[vnames[1]]]]
        alpha.B <- aov.obj$contrasts[[vnames[2]]] %*% cfs[
               aov.obj$assign[[vnames[2]]]]
        cij <- alpha.B %*% t(alpha.A)
        cij <- c(cij)/cfs[aov.obj$assign$"(Intercept)"]
        na <- length(levels(vara))
        nb <- length(levels(varb))
        r.mat <- matrix(NA, nb, na)
        r.mat[cbind(as.vector(unclass(varb)), as.vector(
               unclass(vara)))] <- resid(aov.obj)
        plot(cij, as.vector(r.mat))
        ls.fit <- lsfit(as.vector(cij), as.vector(r.mat))
        abline(ls.fit)
        output <- ls.print(ls.fit, print.it = F)
        list(theta.hat = output$coef.table[2, 1], std.error
                 = output$coef.table[2, 2], R.squared =
                 output$summary[2])
}
```

13.3 THE TWO-WAY LAYOUT WITH REPLICATES

The data in table 13.3 (used by Box, Hunter, and Hunter (1978)) displays the survival times, in units of 10 hours, of animals in a 3×4 *replicated* factorial experiment. In this experiment, each animal was given one of three poisons, labeled I, II, and III, and one of four treatments, labeled A, B, C, and D. Four animals were used for each combination of poison and treatment, making four *replicates*.

Setting Up the Data Frame

To set up the data frame, first make a list, `fnames`, with components `treatment` and `poison`, containing the level names of these two factors:

```
> fnames <- list(treatment=LETTERS[1:4],
+ poison=c("I","II","III"))
```

Table 13.3: *A replicated factorial experiment.*

	treatment			
poison	A	B	C	D
I	0.31	0.82	0.43	0.45
	0.45	1.10	0.45	0.71
	0.46	0.88	0.63	0.66
	0.43	0.72	0.76	0.62
II	0.36	0.92	0.44	0.56
	0.29	0.61	0.35	1.02
	0.40	0.49	0.31	0.71
	0.23	1.24	0.40	0.38
III	0.22	0.30	0.23	0.30
	0.21	0.37	0.25	0.36
	0.18	0.38	0.24	0.31
	0.23	0.29	0.22	0.33

Use `fac.design`, with optional argument `rep=4`, to create the design data frame `poisons.design`:

```
> poisons.design <- fac.design(c(4,3), fnames, rep=4)
```

Note that since `treatments` is the first factor in the `fnames` list, and treatments has 4 levels, 4 is the *first* argument of `c(4,3)`.

You now need to create the vector `surv.time` to match `poisons.design`. Each replicate of the experiment consists of data in three rows of table 13.3. Rows 1, 5, and 9 make up the first replicate, and so on. The command to get what we want is:

```
> surv.time <- scan()
1: .31 .82 .43 .45
5: .36 .92 .44 .56
9: .22 .30 .23 .30
13: .45 1.10 .45 .71
17: .29 .61 .35 1.02
21: .21 .37 .25 .36
25: .46 .88 .63 .66
29: .40 .49 .31 .71
33: .18 .38 .24 .31
37: .43 .72 .76 .62
41: .23 1.24 .40 .38
45: .23 .29 .22 .33
49:
```

Finally, make the data frame `poisons.df`:

```
> poisons.df <- data.frame(poisons.design, surv.time)
```

A First Look at the Data

Use `plot.design`, `plot.factor`, and `interaction.plot` to get a first look at the data through summary statistics.

Set `par(mfrow=c(3,2))` and use the above three functions to get the three row and two column layout of plots displayed in figure 13.12:

```
> par(mfrow=c(3,2))
```

To obtain the design plot of sample means shown in the upper left plot of figure 13.12, use `plot.design` as follows:

```
> plot.design(poisons.df)
```

To obtain the design plot of sample medians shown in the upper right hand plot of figure 13.12, use `plot.design` again:

```
> plot.design(poisons.df, fun=median)
```

The two sets of boxplots shown in the middle row of figure 13.12 are obtained with:

```
> plot.factor(poisons.df)
```

To obtain the bottom row of figure 13.12, use `interaction.plot`:

```
> attach(poisons.df)
> interaction.plot(treatment,poison,surv.time)
> interaction.plot(treatment,poison,surv.time,fun=median)
```

The main differences between the plots obtained with `plot.design` using means and medians are:

- the difference between the horizontal lines which represents the mean and median, respectively, for all the data,
- the difference between the tick marks for the poison factor at level II.

The boxplots resulting from the use of `plot.factor` indicate a clear tendency for variability to increase with the (median) level of response.

The plots made with `interaction.plot` show stronger treatment effects for the two poisons with large levels than for the lowest level poison—an indication of an interaction.

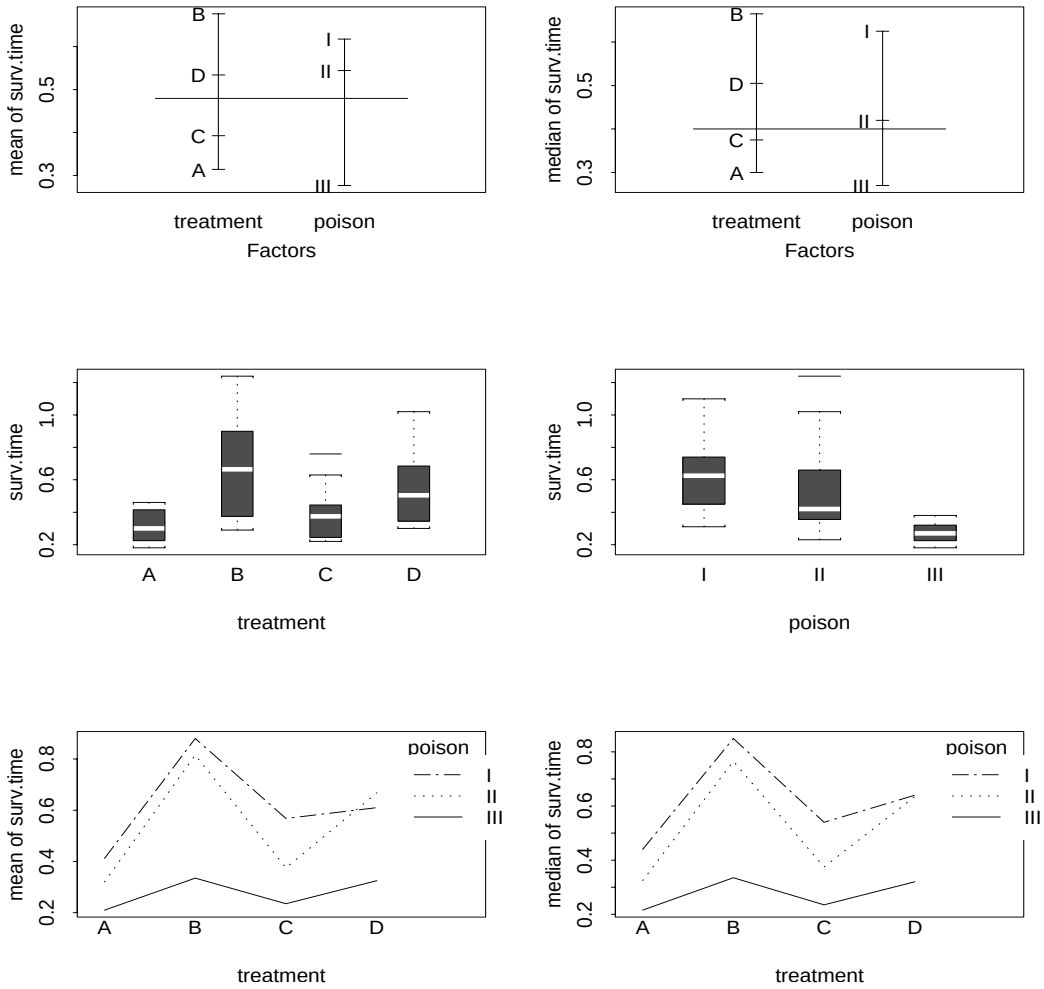

Figure 13.12: *Initial plots of the data.*

The Two-Way Model and ANOVA (With Replicates)

When you have replicates, you can consider a model which includes an interaction term α_{ij}^{AB}:

$$y_{ijk} = \mu + \alpha_i^A + \alpha_j^B + \alpha_{ij}^{AB} + \varepsilon_{ijk} \qquad \begin{aligned} i &= 1, \ldots, I \\ j &= 1, \ldots, J \\ k &= 1, \ldots, K \end{aligned}$$

You can now carry out an ANOVA for the above model using `aov` as follows:

```
> aov.poisons <- aov(surv.time ~ poison*treatment,
+                    poisons.df)
```

The expression `poison*treatment` on the right-hand side of the formula specifies that `aov` fit the above model with interaction. This contrasts with the formula `surv.time ~ poison + treatment`, which tells `aov` to fit an additive model for which α_{ij}^{AB} is assumed to be zero for all levels i, j.

You now display the ANOVA table with `summary`:

```
> summary(aov.poisons)
                 Df Sum of Sq   Mean Sq  F Value     Pr(F)
poison            2  1.033013 0.5165063 23.22174 0.0000003
treatment         3  0.921206 0.3070688 13.80558 0.0000038
poison:treatment  6  0.250138 0.0416896  1.87433 0.1122506
Residuals        36  0.800725 0.0222424
```

The p-values for both poisons and treatment are highly significant, while the p-value for interaction is insignificant.

The colon in `poison:treatment` denotes an interaction, in this case the poison-treatment interaction.

Diagnostic Plots

Make a histogram and a normal QQ-plot of residuals, arranging the plots side by side in a single figure with `par(mfrow=c(1,2))` before using `hist` and `qqnorm`:

```
> par(mfrow=c(1,2))
> hist(resid(aov.poisons))
> qqnorm(resid(aov.poisons))
> par(mfrow=c(1,1))
```

The call `par(mfrow=c(1,1))`, resets the plot layout to a single plot per figure.

The histogram in the left-hand plot of figure 13.13 reveals a marked asymmetry, which is reflected in the normal QQ-plot in the right-hand side of figure 13.13. The latter shows a curved departure from linearity toward

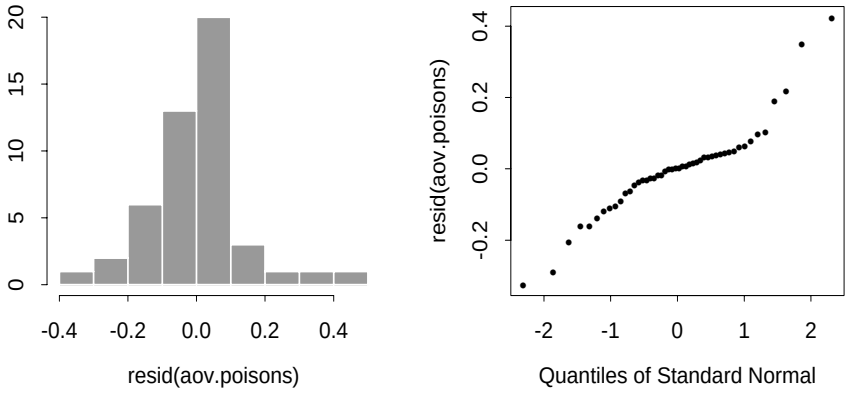

Figure 13.13: *Histogram and normal QQ-plot of residuals.*

the lower left part of the plot, and a break in linearity in the upper right part of the plot. Evidently, all is not well (see the discussion on transforming the data in the Guidance section below).

Make a plot of residuals versus fit:

```
plot(fitted(aov.poisons), resid(aov.poisons))
```

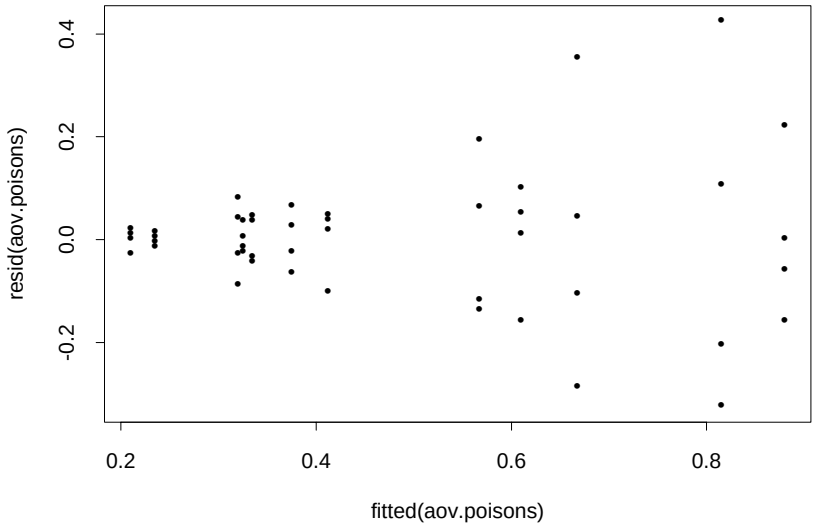

Figure 13.14: *Plot of residuals versus fit.*

The result, displayed in figure 13.14, clearly reveals a strong relationship between the residuals and the fitted values. The variability of the residuals increases with increasing fitted values. This is another indication that

transformation would be useful.

Guidance

When the error variance for an experiment varies with the expected value of the observations, a *variance stabilizing* transformation will often reduce or eliminate such behavior.

We shall show two methods for determining an appropriate variance stabilizing transformation, one which requires replicates and one which does not.

Method for Two-Factor Experiments with Replicates

For two-factor experiments with replicates, you can gain insight into an appropriate variance stabilizing transformation by carrying out the following informal procedure. First, calculate the within-cell standard deviations $\hat{\sigma}_{ij}$ and means $\bar{y}_{ij}$:

```
> std.poison <- tapply(poisons.df$surv.time,
+              list(poisons.df$treatment,
+                   poisons.df$poison),var)^.5
> std.poison <- as.vector(std.poison)
> means.poison <- tapply(poisons.df$surv.time,
+                list(poisons.df$treatment,
+                     poisons.df$poison),mean)
> means.poison <- as.vector(means.poison)
```

Then plot $\log(\hat{\sigma}_{ij})$ versus $\log(\bar{y}_{ij})$, and use the slope of the regression line to estimate the variance stabilizing transform:

```
> plot(log(means.poison),log(std.poison))
> var.fit <- lsfit(log(means.poison),
+ log(std.poison))
> abline(var.fit)
> theta <- var.fit$coef[2]
> theta
       X
 1.97704
```

Now let $\hat{\lambda} = 1 - \hat{\theta}$, and choose λ to be that value among the set of values $\left\{-1, -\frac{1}{2}, 0, \frac{1}{2}, 1\right\}$ which is closest to $\hat{\lambda}$. If $\lambda = 0$, then make the transformation $\tilde{y}_{ij} = \log y_{ij}$. Otherwise, make the power transformation $\tilde{y}_{ijk} = y_{ijk}^{\lambda}$. Now you should repeat the complete analysis described in the

previous subsections, using the response $\tilde{y}_{ijk}$ in place of y_{ijk}.

Since for the poisons experiment you get $\hat{\theta} \approx 2$, you choose λ = -1. This gives a reciprocal transformation $\tilde{y}_{ijk} = y_{ijk}^{-1}$, where y_{ijk} are the values you used in the response with surv.time. You can think of the new response $\tilde{y}_{ijk}$ as representing the rate of dying.

The model can be refit using the transformed response:

```
> summary(aov(1/surv.time ~ poison*treatment, poisons.df))
                 Df Sum of Sq  Mean Sq  F Value     Pr(F)
poison            2  34.87712 17.43856 72.63475 0.0000000
treatment         3  20.41429  6.80476 28.34307 0.0000000
poison:treatment  6   1.57077  0.26180  1.09042 0.3867329
Residuals        36   8.64308  0.24009
```

With the transformation the *p*-values for the main effects have decreased while the *p*-value for the interaction has increased—a more satisfactory fit. The diagnostic plots with the new response are much improved also.

Method for Unreplicated Two-Factor Experiments

An alternative simple method for estimating the variance stabilizing transformation is based on the relationship between the log of the absolute residuals and the log of the fitted values. This method has the advantage that it can be used for unreplicated designs. This method is also often preferred to that of plotting log $\hat{\sigma}_{ij}$ against $\bar{y}_{ij}$ even for cases with replication, because $\bar{y}_{ij}$ and $\hat{\sigma}_{ij}$ are not always adequately good estimates of the mean and standard deviation for small values of K (K<8).

This method consists of plotting log of absolute residuals versus log of fitted values, and computing the slope $\hat{\theta}$ of the regression line. You then set $\hat{\lambda} = 1 - \hat{\theta}$. Residuals with very small absolute values should usually be omitted before applying this method. Here is some sample code.

```
> plot(log(abs(fitted(aov.poisons)[
+ abs(resid(aov.poisons))>exp(-10)])),
+ log(abs(resid(aov.poisons)[
+ abs(resid(aov.poisons))>exp(-10)])))
> logrij.fit <- lsfit(
+ log(abs(fitted(aov.poisons)[
+ abs(resid(aov.poisons))>exp(-10)])),
+ log(abs(resid(aov.poisons)[
+ abs(resid(aov.poisons))>exp(-10)])))
```

```
> abline(logrij.fit)
> theta <- logrij.fit$coef[2]
> theta
        X
1.930791
```

You get $\hat{\lambda} = 1 - \hat{\theta} \approx -1$.

Note that the two simple methods described above both lead to nearly identical choices of power transformation to stabilize variance.

Details

You will find that a non-constant standard deviation for observations y_i (y_{ijk} for the two-factor experiment with replicates) is well-explained by a power law relationship in many datasets. In particular, for some constant B and some exponent θ, we have

$$\sigma_y \approx B\eta^{\theta}$$

where σ_y is the standard deviation of the y_i and η is the mean of the y_i. If you then use a power law transformation

$$\tilde{y}_i = y_i^{\lambda}$$

for some fixed exponent λ, it can be shown that the standard deviation $\sigma_{\tilde{y}}$ for the transformed data $\tilde{y}_i$, is given by

$$\sigma_{\tilde{y}} = K\lambda\eta^{\lambda-(1-\theta)}.$$

You can therefore make $\sigma_{\tilde{y}}$ have a constant value, independent of the mean η of the original data y_i (and independent of the approximate mean η^{λ} of the transformed data $\tilde{y}_i$), by choosing

$$\lambda = 1 - \theta.$$

Note that

$$\log \sigma_y \approx \log K + \theta \log \eta$$

Suppose you plot $\log \hat{\sigma}_{ij}$ versus $\log \hat{y}_{ij}$ for a two factor experiment with replicates and find that this plot results in a fairly good straight line fit with slope $\hat{\theta}$, where $\hat{\sigma}_{ij}$ is an estimate of σ_y and $\hat{y}_{ij}$ is an estimate of η. Then the slope $\hat{\theta}$ provides an estimate of θ, and so you set $\hat{\lambda} = 1 - \hat{\theta}$. Since a fractional exponent $\hat{\lambda}$ is not very natural, one often chooses the closest value

$\hat{\lambda}$ in the "natural" set:

-1	Reciprocal
$-\frac{1}{2}$	Reciprocal square root
0	Log
$\frac{1}{2}$	Square root
1	No transformation

Alternative Formal Methods

There are two alternative formal approaches to stabilizing the variance. One approach is to select the power transformation that minimizes the residual squared error. This is equivalent to maximizing the log-likelihood function, and is sometimes referred to as a Box-Cox analysis (see for example, Weisberg (1985); Box (1988); Haaland (1989))

The second approach seeks to stabilize the variance without the use of a transformation, by including the variance function directly in the model. This approach is called generalized least squares/variance function estimation (see for example, Carroll and Ruppert (1988); Davidian and Haaland (1990)).

Transformations are easy to use and may provide a simpler, more parsimonious model (Box (1988)). On the other hand, modeling the variance function directly allows the analysis to proceed on the original scale and allows more direct insight into the nature of the variance function. In cases when the stability of the variance is critical, either of these methods have better statistical properties than the simple informal graphical methods described above.

13.4 MANY FACTORS AT TWO LEVELS: 2^K DESIGNS

The data in table 13.4 come from an industrial product development experiment in which a response variable called *conversion* is measured (in percent) for each possible combination of two levels of four factors:

K *catalyst charge* (10 or 15 pounds),

Te *temperature* (220 or $240^\circ C$),

P *pressure* (50 or 80 pounds per square inch),

C *concentration* (10% or 12%).

The levels are labeled "-" and "+" in the table. All the factors in the experiment are quantitative, so the "-" indicates the "low" level and the "+" indicates the "high" level for each factor. This data set was used by Box,

Hunter, and Hunter (1978).

The design for this experiment is called a 2^4 design, because there are $2^4 = 16$ possible combinations of two levels for four factors.

Setting Up the Data Frame

To set up the data frame first create a list of the four factor names with the corresponding pairs of levels labels:

Table 13.4: *Data from product development experiment.*

	Factor					
observation number	**K**	**Te**	**P**	**C**	conversion (%)	run order
1	–	–	–	–	71	(8)
2	+	–	–	–	61	(2)
3	–	+	–	–	90	(10)
4	+	+	–	–	82	(4)
5	–	–	+	–	68	(15)
6	+	–	+	–	61	(9)
7	–	+	+	–	87	(1)
8	+	+	+	–	80	(13)
9	–	–	–	+	61	(16)
10	+	–	–	+	50	(5)
11	–	+	–	+	89	(11)
12	+	+	–	+	83	(14)
13	–	–	+	+	59	(3)
14	+	–	+	+	51	(12)
15	–	+	+	+	85	(6)
16	+	+	+	+	78	(7)

```
> fnames <- list(K=c("10","15"), Te=c("220","240"),
+ P=c("50","80"), C=c("10","12"))
```

Now use `fac.design` to create the 2^k design data frame `devel.design`:

```
> devel.design <- fac.design(rep(2,4), fnames)
```

The first argument to `fac.design` is a vector of length four, which specifies that there are four factors. Each entry of the vector is a 2, which specifies that there are two levels for each factor.

Since `devel.design` matches table 13.4, you can simply scan in the coversion data:

```
> conversion <- scan()
1: 71 61 90 82 68 61 87 80
9: 61 50 89 83 59 51 85 78
17:
```

Finally, create the data frame `devel.df`:

```
> devel.df <- data.frame(devel.design, conversion)
> devel.df
    K  Te  P  C conversion
 1 10 220 50 10         71
 2 15 220 50 10         61
 3 10 240 50 10         90
            .
            .
            .
15 10 240 80 12         85
16 15 240 80 12         78
```

A First Look at the Data

Use `plot.design` and `plot.factor` to make an initial graphical exploration of the data. To see the design plot with sample means, use the following command, which yields the plot shown in figure 13.15:

```
> plot.design(devel.df)
```

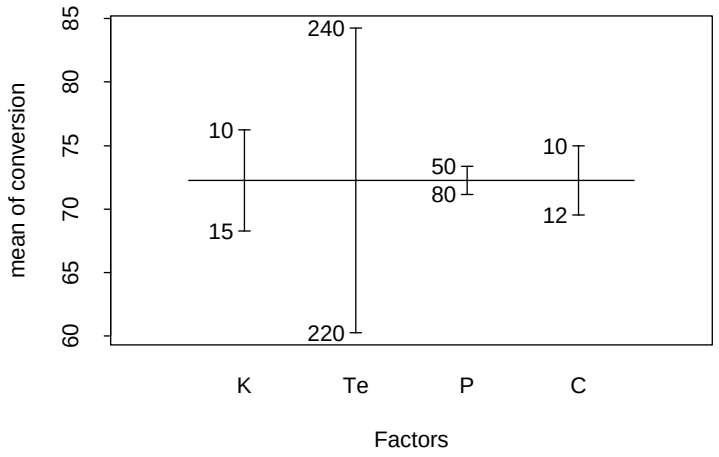

Figure 13.15: *Sample means for product development experiment.*

To see the design plot with sample medians, use:

```
> plot.design(devel.df, fun=median)
```

To see boxplots of the factors, use the following commands, which yield the plots shown in figure 13.16:

```
> par(mfrow=c(2,2))
> plot.factor(devel.df)
> par(mfrow=c(1,1))
```

Estimating All Effects in the 2^k Model

You can use `aov` to estimate *all* effects (main effects and all interactions), and carry out the analysis of variance. Let's do so, and store the results in `aov.devel`:

```
> aov.devel <- aov(conversion ~ K*Te*P*C, devel.df)
```

The product form `K*Te*P*C` on the right-hand side of the formula tells S-PLUS to fit the above 2^4 design model with *all* main effects and *all* interactions included. You can accomplish the same thing by using the power function `^` to raise the expression `K+Te+P+C` to the 4th power:

```
> aov.devel <- aov(conversion ~ (K+Te+P+C)^4, devel.df)
```

This second method is useful when you want to specify only main effects plus certain low-order interactions. For example, replacing 4 by 2 above results in a model with all main effects and all second-order interactions.

You can obtain the estimated coefficients using the `coef` function on the `aov` output:

```
> coef(aov.devel)
 (Intercept)  K Te      P     C K:Te   K:P   Te:P           K:C
       72.25 -4 12 -1.125 -2.75  0.5 0.375 -0.625 -5.464379e-17
 Te:C    P:C K:Te:P K:Te:C  K:P:C Te:P:C K:Te:P:C
 2.25 -0.125 -0.375   0.25 -0.125 -0.375   -0.125
```

Notice that colons are used to connect factor names to represent interactions, e.g., `K:P:C` is the three factor interaction between the factors `K`, `P`, and `C`.

For more on the relationship between coefficients, contrasts and effects, see sections 13.1 and 13.2.

You can get the analysis of variance table with the `summary` command :

```
> summary(aov.devel)
          Df Sum of Sq Mean Sq
K          1    256.00  256.00
Te         1   2304.00 2304.00
P          1     20.25   20.25
C          1    121.00  121.00
```

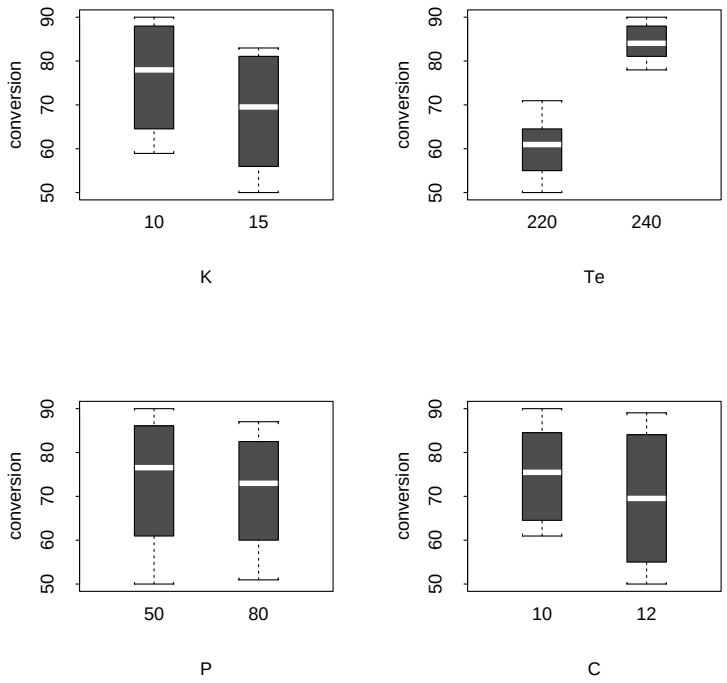

Figure 13.16: *Factor plot for product development experiment.*

```
K:Te       1       4.00     4.00
```

```
K:P       1      2.25    2.25
Te:P      1      6.25    6.25
K:C       1      0.00    0.00
Te:C      1     81.00   81.00
P:C       1      0.25    0.25
K:Te:P    1      2.25    2.25
K:Te:C    1      1.00    1.00
K:P:C     1      0.25    0.25
Te:P:C    1      2.25    2.25
K:Te:P:C  1      0.25    0.25
```

The ANOVA table does not provide any *F* statistics. This is because you have estimated 16 parameters with 16 observations. There are no degrees of freedom left for estimating the error variance, and hence there is no error mean square to use as the denominator of the *F* statistics. However, the ANOVA table can give you some idea of which effects are the main contributors to the response variation.

Estimating All Effects in the 2^k Model with Replicates

On some occasions, you may have replicates of a 2^k design. In this case you can estimate the error variance σ^2, as well as all effects. For example, the data in table 13.5 is from a replicated 2^3 pilot plant example used by Box, Hunter, and Hunter (1978). The three factors are *temperature* (**Te**), *concentration* (**C**) and *catalyst* (**K**), and the response is *yield*.

Table 13.5: *Replicated pilot plant experiment.*

Te	**C**	**K**	rep 1	rep 2
–	–	–	59	61
+	–	–	74	70
–	+	–	50	58
+	+	–	69	67
–	–	+	50	54
+	–	+	81	85
–	+	+	46	44
+	+	+	79	81

To set up the data frame, first make the factor names list:

```
> fnames <- list(Te=c("Tl","Th"), C=c("Cl","Ch"),
+ K=c("Kl","Kh"))
```

Because `T` is a constant in S-PLUS which stands for the logical value "true," you can not use `T` as a factor name for temperature. Instead, use `Te`, or some such alternative abbreviation. Then make the design data frame, `pilot.design`, with $M = 2$ replicates, by using `fac.design` with the optional argument `rep=2`:

```
> pilot.design <- fac.design(c(2,2,2), fnames, rep=2)
```

Now, create the response vector `pilot.yield` as a vector of length 16, with the second replicate values following the first replicate values:

```
> pilot.yield <- scan()
1: 59 74 50 69 50 81 46 79
9: 61 70 58 67 54 85 44 81
17:
```

Finally, use `data.frame`:

```
> pilot.df <- data.frame(pilot.design, pilot.yield)
```

You can now carry out the ANOVA, and because the observations are replicated, the ANOVA table has an error variance estimate, i.e., mean square for error, and F statistics:

```
> aov.pilot <- aov(pilot.yield ~ (Te + C + K)^3, pilot.df)
> summary(aov.pilot)
          Df Sum of Sq Mean Sq F Value    Pr(F)
       Te  1      2116    2116 264.500 0.000000
        C  1       100     100  12.500 0.007670
        K  1         9       9   1.125 0.319813
     Te:C  1         9       9   1.125 0.319813
     Te:K  1       400     400  50.000 0.000105
      C:K  1         0       0   0.000 1.000000
   Te:C:K  1         1       1   0.125 0.732810
Residuals  8        64       8
```

Temperature is clearly highly significant, as is the temperature-catalyst interaction, and concentration is quite significant.

Estimating All Small Order Interactions

In cases where you are confident that high-order interactions are unlikely, you can fit a model which includes interactions only up to a fixed order, through the use of the power function `^` with an appropriate exponent. For example, in the product development experiment of table 13.4, you may wish to estimate only the main effects and all second-order interactions. In

this case, use :

```
> aov.devel.2 <- aov(conversion ~ (K+Te+P+C)^2,devel.df)
```

Now you are using 16 observations to estimate 11 parameters: the mean, the four main effects, and the six two-factor interactions. Since you only use 11 degrees of freedom for the parameters, out of a total of 16, you still have 5 degrees of freedom to estimate the error variance. So the command:

```
> summary(aov.devel.2)
```

will produce an ANOVA table with an error variance estimate and *F* statistics.

Using Half-Normal Plots to Choose a Model

You are usually treading on thin ice if you assume that higher-order interactions are zero, unless you have extensive first-hand knowledge of the process you are studying with a 2^k design. When you are not sure whether or not higher-order interactions are zero, you should use a half-normal quantile-quantile plot to judge which effects, including interactions of any order, are significant. Use the function `qqnorm` as follows to produce a half-normal plot on which you can identify points:

```
> qqnorm(aov.devel, label=6)
```

The resulting figure, with six points labeled, is shown in figure 13.17.

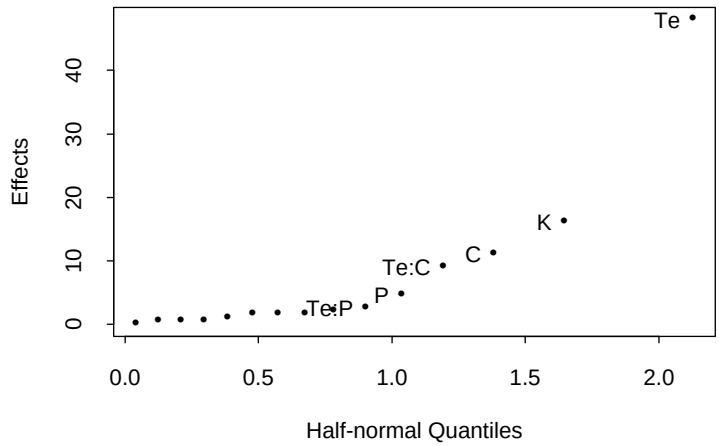

Figure 13.17: *Half-normal plot for product development experiment.*

In general, there are 2^k - 1 points in the half-normal plot, since there are 2^k effects and the estimate of the overall mean is not included in this plot. The *y*-axis positions of the labeled points are the absolute values of the estimated effects. The messages you get from this plot are: You judge the effects for

temperature, catalyst, concentration, and temperature by concentration to be *clearly* non-zero. The effect for Pressure is also very likely non-zero. You can examine the marginal effects better by creating a plot with a smaller *y*-range:

```
> qqnorm(aov.devel, label=6, ylim=c(0,20))
```

A full QQ-plot of the effects can give you somewhat more information. To get this type of plot, use:

```
> qqnorm(aov.devel, full=T, label=6)
```

Having determined from the half-normal plot which effects are non-zero, now fit a model having terms for the main effects plus the interaction between temperature and concentration:

```
> aov.devel.small <- aov(conversion ~ K+P+Te*C, devel.df)
```

You can now get an ANOVA summary, including an error variance estimate:

```
> summary(aov.devel.small)
          Df Sum of Sq  Mean Sq  F Value       Pr(F)
K          1    256.00  256.000  136.533 0.000000375
P          1     20.25   20.250   10.800 0.008200654
Te         1   2304.00 2304.000 1228.800 0.000000000
C          1    121.00  121.000   64.533 0.000011354
Te:C       1     81.00   81.000   43.200 0.000062906
Residuals 10     18.75    1.875
```

Diagnostic Plots

Once you have tentatively identified a model for a 2^k experiment, you should make the usual graphical checks based on the residuals and fitted values. In the product development example, you should examine the following plots:

```
> hist(resid(aov.devel.small))
> qqnorm(resid(aov.devel.small))
> plot(fitted(aov.devel.small),resid(aov.devel.small))
```

The latter two plots are shown in figures 13.18 and 13.19.

You should also make plots using the time order of the runs:

```
> run.ord <- scan()
1: 8 2 10 4 15 9 1 13 16 5 11 14 3 12 6 7
17:
> plot(run.ord, resid(aov.devel.small))
> plot(run.ord, fitted(aov.devel.small))
```

This gives a slight hint that the first runs were more variable than the latter runs.

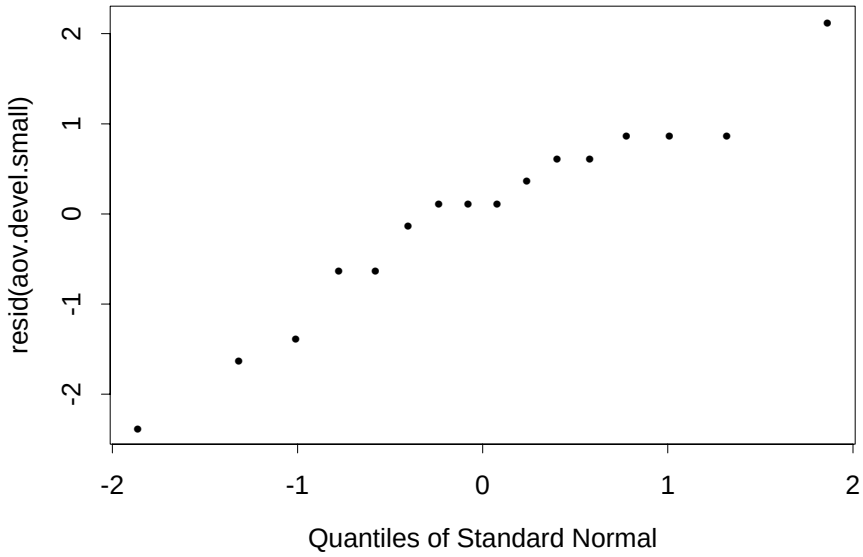

Figure 13.18: *Quantile-quantile plot of residuals, product development example.*

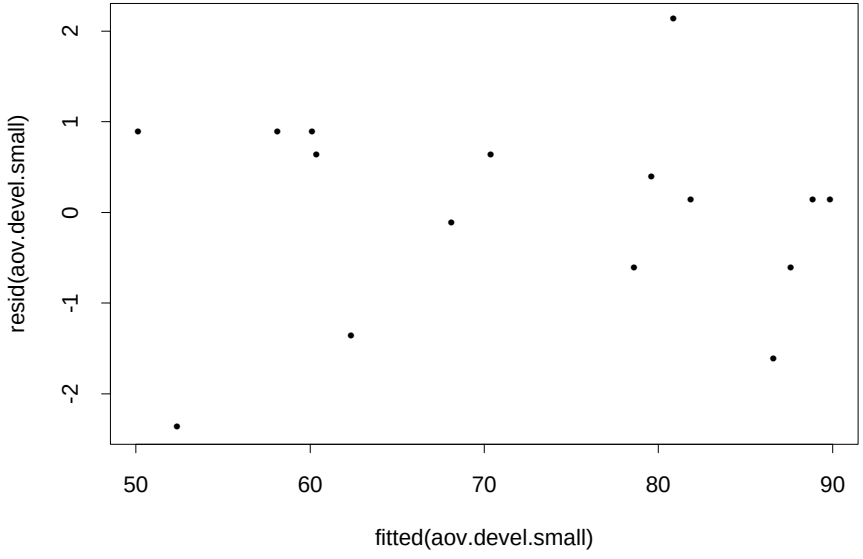

Figure 13.19: *Fitted values vs. residuals, product development example.*

Details

The function aov returns, by default, coefficients corresponding to the following "usual" ANOVA form for the η_i:

$$\begin{aligned}\eta_i = \eta_{i_1 \ldots i_k} = \mu &+ \alpha^1_{i_1} + \alpha^2_{i_2} + \cdots + \alpha^k_{i_k} \\ &+ \alpha^{12}_{i_1 i_2} + \alpha^{13}_{i_1 i_3} + \cdots + \alpha^{k-1,\,k}_{i_{k-1} i_k} \\ &+ \cdots \\ &+ \alpha^{123\ldots k}_{i_1 i_2 \ldots i_k}\end{aligned}$$

In this form of the 2^k model, each i_m takes on just two values, 1 and 2. There are 2^k values of the k-tuple index $i_1 i_2 \ldots i_k$. The parameter μ is the overall mean. The parameters $\alpha^m_{i_m}$, $m = 1, \ldots, k$ correspond to the *main effects*. The parameters $\alpha^{mn}_{i_m i_n}$ correspond to the *two-factor interactions*, the parameters $\alpha^{lmn}_{i_l i_m i_n}$ correspond to the *three-factor interactions*, and the remaining coefficients are the higher-order interactions. The coefficients for the main effects satisfy the constraint $\alpha^i_1 + \alpha^i_2 = 0, i = 1, \ldots, k$. All higher-order interactions satisfy the constraint that the sum over any individual subscript index is zero, e.g. $\alpha^{12}_{i_1 1} + \alpha^{12}_{i_1 2} = 0$, $\alpha^{124}_{1 i_2 i_4} + \alpha^{124}_{2 i_2 i_4} = 0$, etc.

Because of the constraints on the parameters in this form of the model, it suffices to specify one of the two values for each effect. The function aov returns estimates for the "high" levels, e.g., $\hat{\alpha}^i_2$, $\hat{\alpha}^{12}_2$.

An estimated effect (in the sense usually used in 2^k models) is equal to the difference between the estimate at the high level minus the estimate at the low level for the ANOVA model form given above:

$$\hat{\alpha}^1 = \hat{\alpha}^1_2 - \hat{\alpha}^1_1$$

and since

$$\hat{\alpha}^1_1 + \hat{\alpha}^1_2 = 0,$$

we have

$$\hat{\alpha}^1 = 2\hat{\alpha}^1_2.$$

13.5 REFERENCES

Box, G.E.P. and Hunter, W.G. and Hunter, J.S. (1978). *Statistics for Experimenters: An Introduction to Design, Data Analysis*. John Wiley, New York.

Box, G.E.P. (1988). *Signal-to-noise ratios, performance criteria, and transformations*. Technometrics, 30:1–17.

Carroll, R.J. and Ruppert, D. (1988). *Transformation and Weighting in Regression*. Chapman and Hall, New York.

Chambers, J.M. and Hastie, T.J. (1992). *Statistical Models in S*. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove, CA.

Davidian, M. and Haaland, P.D. (1990). *Regression and calibration with non-constant error variance*. Chemometrics and Intelligent Laboratory Systems; 9:231–248.

Haaland, P. (1989). *Experimental Design in Biotechnology.* Marcel Dekker, New York.

Hoaglin, D.C. and Mosteller, F. and Tukey, J.W. (1983). *Understanding Robust and Exploratory Data Analysis*. John Wiley, New York.

Weisberg, S. (1985). *Applied Linear Regression*, 2nd edition. John Wiley, New York.

FURTHER TOPICS IN ANALYSIS OF VARIANCE

14

ANOVA models can be expanded, for example, into multivariate analysis, split-plot designs, and repeated measures.

FURTHER TOPICS IN ANALYSIS OF VARIANCE

The previous chapter, Designed Experiments and Analysis of Variance, describes the basic techniques for using S-PLUS for analysis of variance. This chapter extends the concepts to several related topics:

- Multivariate analysis of variance (MANOVA), discussed in the section Multivariate Analysis of Variance, page 432.
- Split-plot designs (the section Split-plot Designs, page 433).
- Repeated measures (the section Repeated-Measures Designs, page 435).
- Nonparametric tests for one-way and blocked two-way designs (the section Rank Tests for One-Way and Two-Way Layouts, page 438)
- Variance components models (the section Variance Components Models, page 439)

These topics are preceded by a discussion of model coefficients and contrasts (the section Model Coefficients and Contrasts, page 419); this information is important in interpreting the available ANOVA summaries, described in the section Summarizing ANOVA Results, page 423.

14.1 MODEL COEFFICIENTS AND CONTRASTS

This section explains what the coefficients mean in ANOVA models, and how to get more meaningful coefficients for particular cases.

Suppose we have 5 measurements of a response variable `scores` for each of three treatments, `"A"`, `"B"`, and `"C"`, as shown below:

```
> scores
[1] 4 5 4 5 4 10 7 7 7 7 7 7 8 7 6
> scores.treat
[1] A A A A A B B B B B C C C C C
```

In solving the basic ANOVA problem, we are trying to solve the following simple system of equations:

$$\hat{\mu}_A = \hat{\mu} + \hat{\alpha}_A$$
$$\hat{\mu}_B = \hat{\mu} + \hat{\alpha}_B$$
$$\hat{\mu}_C = \hat{\mu} + \hat{\alpha}_C$$

The sample means $\hat{\mu}_A$, $\hat{\mu}_B$, and $\hat{\mu}_C$ can be calculated directly from the data:

$$\hat{\mu}_A = (4 + 5 + 4 + 5 + 4)/5 = 4.4$$
$$\hat{\mu}_B = (10 + 7 + 7 + 7 + 7)/5 = 7.6$$
$$\hat{\mu}_C = (7 + 7 + 8 + 7 + 6)/5 = 7.0 \quad .$$

This leaves the following three equations in four unknowns:

$$4.4 = \hat{\mu} + \hat{\alpha}_A$$
$$7.6 = \hat{\mu} + \hat{\alpha}_B$$
$$7.0 = \hat{\mu} + \hat{\alpha}_C .$$

Like all ANOVA models, this system is *overparametrized*, meaning there are more coefficients than can be estimated. We can, however, replace the three variables $\hat{\alpha}_A$, $\hat{\alpha}_B$, and $\hat{\alpha}_C$ with a pair of variables $\hat{\beta}_1$ and $\hat{\beta}_2$ that are functionally independent linear combinations of the original variables, and also independent of $\hat{\mu}$. Such a replacement can be done in more than one way. For unordered factors such as `scores.treat`, the default choice in S-PLUS is the set of *Helmert contrasts* :

$$\hat{\beta}_1 = -\hat{\alpha}_A + \hat{\alpha}_B$$
$$\hat{\beta}_2 = 2\hat{\alpha}_C - (\hat{\alpha}_A + \hat{\alpha}_B) .$$

These contrasts, in effect, contrast the *i*th level with the average of the preceding levels.

More generally, if you have variables α_i, i=1, …, k, you can reparametrize with the k - 1 variables β_j as follows:

$$\beta_j = j\alpha_{j+1} + \sum_{i=1}^{j} \alpha_i . \qquad (14.1)$$

The transpose of the matrix of coefficients for equation 14.1 is the following

$k \times (k\text{-}1)$ *contrast matrix*:

$$A = \begin{bmatrix} -1 & -1 & -1 & \dots & -1 \\ 1 & -1 & -1 & \dots & -1 \\ 0 & 2 & -1 & \dots & -1 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & k-1 \end{bmatrix}. \tag{14.2}$$

You can recover the original treatment effects $\boldsymbol{\alpha}$ from the reparametrized variables $\boldsymbol{\beta}$ by matrix multiplication as follows:

$$A\boldsymbol{\beta} = \boldsymbol{\alpha}.$$

Returning to our simple example, we can rewrite our original variables in terms of $\hat{\beta}_1$ and $\hat{\beta}_2$ as follows:

$$\begin{aligned} \hat{\alpha}_A &= -\hat{\beta}_1 - \hat{\beta}_2 \\ \hat{\alpha}_B &= \hat{\beta}_1 - \hat{\beta}_2 \\ \hat{\alpha}_C &= 2\hat{\beta}_2. \end{aligned}$$

S-PLUS now solves the following system of equations:

$$\begin{aligned} 4.4 &= \hat{\mu} - \hat{\beta}_1 - \hat{\beta}_2 \\ 7.6 &= \hat{\mu} + \hat{\beta}_1 - \hat{\beta}_2 \\ 7.0 &= \hat{\mu} + 2\hat{\beta}_2. \end{aligned}$$

If we use `aov` as usual to create the `aov` object `scores.aov`, we can use the `coef` function to look at the solved values $\hat{\mu}$, $\hat{\beta}_1$, and $\hat{\beta}_2$:

```
> scores.aov <- aov(scores ~ scores.treat)
> coef(scores.aov)
(Intercept) scores.treat1 scores.treat2
   6.333333           1.6     0.3333333
```

In our example, the contrast matrix is as follows:

$$\begin{pmatrix} -1 & -1 \\ 1 & -1 \\ 0 & 2 \end{pmatrix}.$$

You can obtain the contrast matrix for any factor object using the `contrasts` function. For unordered factors such as `scores.treat`, `contrasts` returns the Helmert contrast matrix of the appropriate size:

```
> contrasts(scores.treat)
  [,1] [,2]
A   -1   -1
B    1   -1
C    0    2
```

The contrast matrix, together with the treatment coefficients returned by `coef`, provides an alternative to using `model.tables` to calculate effects:

```
> contrasts(scores.treat) %*% coef(scores.aov)[-1]
        [,1]
A -1.9333333
B  1.2666667
C  0.6666667
```

For *ordered* factors, the Helmert contrasts are replaced, by default, with *polynomial contrasts* that model the response as a polynomial through equally spaced points. For example, suppose we define an ordered factor `water.temp` as follows:

```
> water.temp <- ordered(c(65, 95, 120))
> water.temp
[1] 65  95  120
 65 < 95 < 120
```

The contrast matrix for `water.temp` uses polynomial contrasts:

```
> contrasts(water.temp)
             .L         .Q
 65 -0.7071068  0.4082483
 95  0.0000000 -0.8164966
120  0.7071068  0.4082483
```

For the polynomial contrasts, $\hat{\beta}_1$ represents the linear component of the response, $\hat{\beta}_2$ represents the quadratic component, and so on. When examining ANOVA summaries, you can split a factor's effects into contrast terms to examine each component's contribution to the model. See the section Splitting Treatment Sums of Squares into Contrast Terms, page 424, for complete details.

At times it is desirable to give particular contrasts to some of the coefficients. In our example, you might be interested in a contrast that has A equal to a weighted average of B and C. This might occur, for instance, if the treatments were really doses. You can add a contrast attribute to the factor using the assignment form of the `contrasts` function:

```
> contrasts(scores.treat) <- c(4,-1,-3)
> contrasts(scores.treat)
  [,1]       [,2]
A    4  0.2264554
B   -1 -0.7925939
C   -3  0.5661385
```

Note that a second contrast was automatically added.

Refitting the model, we now get different coefficients (but the fit remains the same).

```
> scores.aov2 <- aov(scores ~ scores.treat)
> coef(scores.aov2)
 (Intercept) scores.treat1 scores.treat2
    6.333333    -0.4230769      -1.06434
```

More details on working with contrasts can be found in the section Contrasts: The Coding of Factors, in chapter 2.

14.2 SUMMARIZING ANOVA RESULTS

Results from an analysis of variance are typically displayed in an *analysis of variance table*, which shows a decomposition of the variation in the response: the total sum of squares of the response is split into sums of squares for each treatment and interaction and a residual sum of squares. You can obtain the ANOVA table, as we have throughout this chapter, by using `summary` on the result of a call to `aov`, such as this overly simple model for the `wafer` data:

```
> attach(wafer)
> wafer.aov <- aov( pre.mean ~ visc.tem + devtime
+                 + etchtime)
> summary(wafer.aov)
           Df Sum of Sq   Mean Sq  F Value     Pr(F)
visc.tem    2  1.343361 0.6716807 3.678485 0.0598073
devtime     2  0.280239 0.1401194 0.767369 0.4875574
etchtime    2  0.103323 0.0516617 0.282927 0.7588959
Residuals 11  2.008568 0.1825971
```

Splitting Treatment Sums of Squares into Contrast Terms

Each treatment sum of squares in the ANOVA table can be further split into terms corresponding to the treatment contrasts. By default, the Helmert contrasts are used for unordered factors and polynomial contrasts for ordered factors. For instance, with ordered factors you can assess whether the response is fairly linear in the factor by listing the polynomial contrasts separately. In the dataset `wafer`, you can examine the linear and quadratic contrasts of `devtime` and `etchtime` by using the `split` argument to the `summary` function:

```
> summary(wafer.aov, split = list(etchtime =
+     list(L = 1, Q = 2),
+     devtime = list(L = 1, Q = 2)))
               Df Sum of Sq   Mean Sq  F Value     Pr(F)
visc.tem        2  1.343361 0.6716807 3.678485 0.0598073
devtime         2  0.280239 0.1401194 0.767369 0.4875574
  devtime: L    1  0.220865 0.2208653 1.209577 0.2949025
  devtime: Q    1  0.059373 0.0593734 0.325161 0.5799830
etchtime        2  0.103323 0.0516617 0.282927 0.7588959
  etchtime: L   1  0.094519 0.0945188 0.517636 0.4868567
  etchtime: Q   1  0.008805 0.0088047 0.048219 0.8302131
Residuals      11  2.008568 0.1825971
```

Each of the (indented) split terms sum to their overall sum of squares.

The `split` argument can evaluate only the effects of the contrasts used to specify the ANOVA model: if you wish to test a specific contrast, you need to set it explicitly before fitting the model. Thus, if you want to test a polynomial contrast for an unordered factor, you must specify polynomial contrasts for the factor before fitting the model. The same is true for other non-default contrasts. For instance, the variable `visc.tem` in the `wafer` dataset is a three-level factor constructed by combining two levels of viscosity (204 and 206) with two levels of temperature (90 and 105).

```
> levels(visc.tem)
[1] "204,90"  "206,90"  "204,105"
```

To assess viscosity, supposing temperature has no effect, we define a contrast that takes the difference of the middle and the sum of the first and third levels of `visc.tem`; the contrast matrix is automatically completed:

```
> contrasts(visc.tem) <- c(-1,2,-1)
> contrasts(visc.tem)
        [,1]          [,2]
204,90    -1 -7.071068e-01
206,90     2 -1.110223e-16
204,105   -1  7.071068e-01
```

```
> wafer.aov <- aov( pre.mean ~ visc.tem + devtime
+                 + etchtime)
```

In this fitted model, the first contrast for `visc.aov` reflects the effect of viscosity:

```
> summary(wafer.aov, split = list(visc.tem = list(visc = 1)))
                  Df Sum of Sq  Mean Sq  F Value     Pr(F)
visc.tem           2  1.343361 0.671681 3.678485 0.0598073
  visc.tem: visc   1  1.326336 1.326336 7.263730 0.0208372
devtime            2  0.280239 0.140119 0.767369 0.4875574
etchtime           2  0.103323 0.051662 0.282927 0.7588959
Residuals         11  2.008568 0.182597
```

Treatment Means and Standard Errors

Commonly the ANOVA model is written in the form "grand mean plus treatment effects,"

$$y_{ijk} = \mu + \alpha_i + \beta_j + (\alpha\beta)_{ij} + \varepsilon_{ijk}$$

The treatment effects, α_i, β_j, $(\alpha\beta)_{ij}$ reflect changes in the response due to that combination of treatments. In this parameterization, the effects are constrained, usually to sum to zero.

Unfortunately, the use of the term "effect" in ANOVA is not standardized: in factorial experiments an effect is the difference between treatment levels, in balanced designs it is the difference from the grand mean, and in unbalanced designs there are (at least) two different standardizations that make sense.

The coefficients of an `aov` object returned by `coef(aov.object)` are coefficients for the contrast variables derived by the `aov` function, rather than the grand-mean-plus-effects decomposition. The functions `dummy.coef` and `model.tables` translate the internal coefficients into the more natural treatment effects.

Balanced Designs

In a balanced design, both computing and interpreting effects are straightforward:

```
> gun.aov <- aov(Rounds ~ Method + Physique/Team, gun)
> coef(gun.aov)
 (Intercept)    Method Physique.L  Physique.Q
    19.33333 -4.255556  -1.154941 -0.06123724
 PhysiqueSTeam1 PhysiqueATeam1 PhysiqueHTeam1
         1.9375           0.45          -0.45
 PhysiqueSTeam2 PhysiqueATeam2 PhysiqueHTeam2
        -0.4875    0.008333333     -0.1083333
```

The `dummy.coef` function translates the coefficients into the more natural effects:

```
> dummy.coef(gun.aov)
$"(Intercept)":
 (Intercept)
    19.33333

$Method:
       M1        M2
 4.255556 -4.255556

$Physique:
[1]  0.7916667  0.0500000 -0.8416667

$"Team %in% Physique":
   1T1        2T1       3T1   1T2       2T2
 -1.45 -0.4583333 0.5583333 2.425 0.4416667
        3T2    1T3        2T3        3T3
 -0.3416667 -0.975 0.01666667 -0.2166667
```

For the default contrasts, these effects always sum to zero.

The same information is returned in a tabulated form by `model.tables` Note that `model.tables` calls `proj`, hence it is helpful to use `qr = T` in the call to aov.

```
> model.tables(gun.aov, se = T)
Refitting model to allow projection
Tables of effects


 Method
    M1     M2
 4.256 -4.256

 Physique
      S    A       H
 0.7917 0.05 -0.8417

 Team %in% Physique
Dim 1 : Physique
Dim 2 : Team
```

```
      T1     T2     T3
S -1.450  2.425 -0.975
A -0.458  0.442  0.017
H  0.558 -0.342 -0.217

Standard errors of effects
     Method Physique Team %in% Physique
     0.3381   0.4141             0.7172
rep 18.0000  12.0000             4.0000
```

Using the first method, the gunners fired on average 4.26 more rounds than the overall mean. The standard errors for the effects are simply the residual standard error scaled by the replication factor, `rep`, the number of observations at each level of the treatment. For instance, the standard error for the `Method` effect is:

$$\text{se(Method)} = \frac{\text{se(Residual)}}{\sqrt{r\text{eplication(Method)}}} = \frac{1.434}{\sqrt{18}} = 0.3381$$

The `model.tables` function also computes cell means for each of the treatments. This provides a useful summary of the analysis that is more easily related to the original data.

```
> model.tables(gun.aov, type = "means", se = T)
Tables of means

Grand mean
 19.33

 Method
    M1    M2
 23.59 15.08

 Physique
     S     A     H
 20.13 19.38 18.49

 Team %in% Physique
Dim 1 : Physique
Dim 2 : Team
     T1    T2    T3
S 18.68 22.55 19.15
A 18.93 19.83 19.40
H 19.05 18.15 18.28
```

```
Standard errors for differences of means
     Method Physique Team %in% Physique
     0.4782   0.5856              1.014
rep 18.0000  12.0000              4.000
```

The first method had an average firing rate of 23.6 rounds. For the tables of means, standard errors of *differences* between means are given, as these are usually of most interest to the experimenter. For instance the standard error of differences for `Team %in% Physique` is

$$\text{SED} = \sqrt{2 \times \frac{2.0576}{4}} = 1.014$$

To gauge the statistical significance of the difference between the first and second small physique teams, we can compute the "least significant difference (LSD)" for the `Team %in% Physique` interaction. The validity of the statistical significance is based on the assumption that the model is correct and the residuals are Gaussian. The plots of the residuals indicate these are not unreasonable assumptions for this dataset—you can verify this by creating a histogram and normal QQ-plot of the residuals as follows:

```
> hist(resid(gun.aov))
> qqnorm(resid(gun.aov))
```

The LSD at the 95% level is

$$t(0.975, 26) \times \text{SED}(\texttt{Team \%*\% Physique}).$$

We use the *t*-distribution with 26 degrees of freedom because the residual sum of squares has 26 degrees of freedom. In S-PLUS, we type the following:

```
> qt(0.975, 26) * 1.014
[1] 2.084307
```

Since the means of the two teams differ by more than 2.08, the teams are different at the 95% level of significance. From an interaction plot it is clear that the results for teams of small physique are unusually high.

2^k Factorial Designs

In factorial experiments, where each experimental treatment has only two levels, a treatment *effect* is, by convention, the difference between the high and low levels. Interaction effects are half the average difference between paired levels of an interaction. These *factorial effects* are computed when `type = "feffects"` is used in the `model.tables` function:

```
> catalyst.aov <- aov( Yield ~ ., catalyst, qr = T)
> model.tables(catalyst.aov, type = "feffects", se = T)
```

```
Table of factorial effects
     Effects    se
Temp    23.0 5.062
Conc    -5.0 5.062
Cat      1.5 5.062
```

Unbalanced Designs

When designs are unbalanced (there are unequal numbers of observations in some cells of the experiment), the effects associated with different treatment levels can be standardized in different ways. For instance, suppose we use only the first 35 observations of the `gun` data set:

```
> gunsmall.aov <- aov(Rounds ~ Method
+               + Physique/Team, gun,
+               + subset=1:35, qr = T)
```

The `dummy.coef` function standardizes treatment effects to sum to zero:

```
> dummy.coef(gunsmall.aov)
$"(Intercept)":
 (Intercept)
    19.29177

$Method:
       M1        M2
 4.297115 -4.297115

$Physique:
[1]  0.83322650  0.09155983 -0.92478632

$"Team %in% Physique":
   1T1        2T1       3T1   1T2       2T2
 -1.45 -0.4583333 0.6830128 2.425 0.4416667

        3T2    1T3        2T3        3T3
 -0.2169872 -0.975 0.01666667  -0.466025
```

The `model.tables` function computes effects that are standardized so the weighted effects sum to zero: $\sum_{i=1}^{T} n_i \tau_i = 0$, where n_i is the replication of level i and τ_i the effect. The `model.tables` effects are identical to the values of the projection vectors computed by `proj(gunsmall.aov)`:

```
> model.tables(gunsmall.aov)
```

```
Tables of effects

 Method
        M1      M2
     4.135  -4.378
rep 18.000  17.000

 Physique
          S        A        H
     0.7923  0.05065  -0.9196
rep 12.0000 12.00000  11.0000

 Team %in% Physique
Dim 1 : Physique
Dim 2 : Team
         T1     T2     T3
S    -1.450  2.425 -0.975
rep   4.000  4.000  4.000
A    -0.458  0.442  0.017
rep   4.000  4.000  4.000
H     0.639 -0.261 -0.505
rep   4.000  4.000  3.000
```

With this standardization, treatment effects are orthogonal: consequently cell means can be computed by simply adding effects to the grand mean; standard errors are also more readily computed.

```
> model.tables(gunsmall.aov, type="means", se=T)
Standard error information not returned as design
   is unbalanced.
Standard errors can be obtained through se.contrast.
Tables of means
Grand mean

 19.45

 Method
       M1    M2
    23.59 15.08
rep 18.00 17.00
```

```
 Physique
        S    A     H
    20.25 19.5 18.53
rep 12.00 12.0 11.00

 Team %in% Physique
Dim 1 : Physique
Dim 2 : Team
       T1    T2    T3
S   18.80 22.67 19.27
rep  4.00  4.00  4.00
A   19.05 19.95 19.52
rep  4.00  4.00  4.00
H   19.17 18.27 18.04
rep  4.00  4.00  3.00
```

Note that the `(Intercept)` value returned by `dummy.coef` is not the grand mean of the data, and the coefficients returned are not a decomposition of the cell means. This is a difference that occurs only with unbalanced designs: in balanced designs the functions `dummy.coef` and `model.tables` return identical values for the effects.

In the unbalanced case, the standard errors for comparing two means depend on the replication factors, hence it could be very complex to tabulate all combinations. Instead, they can be computed directly with `se.contrast`. For instance, to compare the first and third teams of heavy physique:

```
> se.contrast(gunsmall.aov, contrast = list(
+         Physique=="S"  Team == "T1",
+         Physique=="S"  Team == "T3"),
+         data = gun[1:35,])
[1] 1.018648
```

By default, the standard error of the difference of the means specified by `contrast` is computed. Other contrasts are specified by the argument `coef`. For instance, to compute the standard error of the contrast tested in the section Splitting Treatment Sums of Squares into Contrast Terms, page 424, for the variable `visc.tem`:

```
> attach(wafer)
> se.contrast(wafer.aov, contrast = list(
+         visc.tem ==levels(visc.tem)[1],
+         visc.tem == levels(visc.tem)[2],
+         visc.tem == levels(visc.tem)[3]),
+         coef = c(-1,2,-1), data = wafer)
```

```
Refitting model to allow projection
[1] 0.07793052
```

The value of the contrast can be computed from `model.tables(wafer.aov)`. The effects for `visc.tem` are:

```
visc.tem
204,90  206,90 204,105
0.1543 -0.3839  0.2296
```

The contrast is `-0.3839 - mean(c(0.1543, 0.2296)) = -0.5758`. The standard error for testing whether the contrast is zero is 0.0779; clearly the contrast is nonzero.

14.3 MULTIVARIATE ANALYSIS OF VARIANCE

Multivariate analysis of variance, known as MANOVA, is the extension of analysis of variance techniques to multiple responses. The responses for an observation are considered as one multivariate observation, rather than as a collection of univariate responses.

If the responses are independent, then it is sensible to just perform univariate analyses. However, if the responses are correlated, then MANOVA can be more informative than the univariate analyses as well as less repetitive.

In S-PLUS the `manova` function is used to estimate the model. The formula needs to have a matrix as the response:

```
> wafer.manova <- manova(cbind(pre.mean, post.mean) ~ .,
+                        wafer[,c(1:9, 11)])
```

The `manova` function creates an object of class `"manova"`. This class of object has methods specific to it for a few generic functions. The most important function is the `"manova"` method for `summary`, which produces a MANOVA table:

```
> summary(wafer.manova)
           Df Pillai Trace approx. F num df   den df   P-value
  maskdim 1  0.9863       36.00761  2        1        0.11703
 visc.tem 2  1.00879      1.01773   4        4        0.49341
   spinsp 2  1.30002      1.85724   4        4        0.28173
 baketime 2  0.80133      0.66851   4        4        0.64704
 aperture 2  0.96765      0.93733   4        4        0.52425
  exptime 2  1.63457      4.47305   4        4        0.08795
  devtime 2  0.99023      0.98065   4        4        0.50733
 etchtime 2  1.26094      1.70614   4        4        0.30874
Residuals 2
```

There are four common types of test in MANOVA. The example above

shows the Pillai-Bartlett trace test, which is the default test in S-PLUS. The last four columns show an approximate *F* test (since the distributions of the four test statistics are not implemented). The other available tests are Wilks' Lambda, Hotelling-Lawley trace, and Roy's maximum eigenvalue. (By the way, a model with this few residual degrees of freedom is not likely to produce informative tests.)

You can view the results of another test by using the `test` argument. The following command shows you Wilks' lambda test:

```
> summary(wafer.manova, test="wilk")
```

Below is an example of how to see the results of all four of the multivariate tests:

```
> wafer.manova2 <- manova(cbind(pre.mean, post.mean,
+    log(pre.dev), log(post.dev)) ~ maskdim + visc.tem
+    + spinsp, wafer)
> wafer.ms2 <- summary(wafer.manova2)
> for(i in c("p","w","h","r")) print(wafer.ms2, test=i)
```

You can also look at the univariate ANOVA tables for each response with a command like:

```
> summary(wafer.manova, univariate=T)
```

Hand and Taylor (1987) provide a nice introduction to MANOVA. Many books on multivariate statistics contain a chapter on MANOVA. Examples include Mardia, Kent and Bibby (1979), and Seber (1984).

14.4 SPLIT-PLOT DESIGNS

A split-plot design contains more than one source of error. This can arise because factors are applied at different scales, as in the guayule example below.

Split-plots are also encountered because of restrictions on the randomization. For example, an experiment involving oven temperature and baking time will probably not randomize the oven temperature totally, but rather only change the temperature after all of the runs for that temperature have been made. This type of design is often mistakenly analyzed as if there were no restrictions on the randomization (an indication of this can be *p*-values that are close to 1). See Hicks (1973) and Daniel (1976).

S-PLUS includes the `guayule` data frame which is also discussed in Chambers and Hastie (1992). This experiment was on eight varieties of guayule (a rubber producing shrub) and four treatments on the seeds. Since a flat (a shallow box for starting seedlings) was not large enough to contain all 32 combinations of variety and treatment, the design was to use only a single

variety in each flat and to apply each treatment within each flat. Thus the flats each consist of four sub-plots. This is a split-plot design since flats are the experimental unit for varieties, but the sub-plots are the experimental unit for the treatments. The response is the number of plants that germinated in each sub-plot.

To analyze a split-plot design like this, put the variable that corresponds to the whole plot in an `Error` term in the formula of the `aov` call:

```
> gua.aov1 <- aov(plants ~ variety*treatment
+               + Error(flats), guayule)
```

As usual, you can get an ANOVA table with `summary`:

```
> summary(gua.aov1)
Error: flats
          Df Sum of Sq  Mean Sq  F Value     Pr(F)
variety    7   763.156 109.0223 1.232036 0.3420697
Residuals 16  1415.833  88.4896

Error: Within
                  Df Sum of Sq  Mean Sq  F Value       Pr(F)
treatment          3  30774.28 10258.09 423.4386 0.00000e+00
variety:treatment 21   2620.14   124.77   5.1502 1.32674e-06
Residuals         48   1162.83    24.23
```

This shows varieties tested with the error from flats, while treatment and its interaction with variety are tested with the within-flat error (which is substantially smaller).

The `guayule` data actually represent an experiment in which the flats were grouped into replicates—making three sources of error, or a split-split-plot design. To model this we put more than one term inside the `Error` term:

```
> gau.aov2 <- aov(plants ~ variety*treatment
+                  + Error(reps/flats), guayule)
> summary(gau.aov2)
Error: reps
          Df Sum of Sq  Mean Sq F Value Pr(F)
Residuals  2  38.58333 19.29167

Error: flats %in% reps
          Df Sum of Sq  Mean Sq  F Value     Pr(F)
variety    7   763.156 109.0223 1.108232 0.4099625
Residuals 14  1377.250  98.3750
```

```
Error: Within
                  Df Sum of Sq  Mean Sq  F Value       Pr(F)
treatment          3  30774.28 10258.09 423.4386 0.00000e+00
variety:treatment 21   2620.14   124.77   5.1502 1.32674e-06
Residuals         48   1162.83    24.23
```

The `Error` term could also have been specified as `Error(reps + flats)`. However, the specification `Error(flats + reps)` would not give the desired result (the sequence within the `Error` term is significant); explicitly stating the nesting is preferred. Note that only one `Error` term is allowed.

14.5 REPEATED-MEASURES DESIGNS

Repeated-measures designs are those that contain a sequence of observations on each subject—for example, a medical experiment in which each patient is given a drug, and observations are taken at zero, one, two and three weeks after taking the drug. (The above description is a little too simplistic to encompass all repeated-measures designs, but it captures the spirit.)

Repeated-measures designs are similar to split-plot designs in that there is more than one source of error (between subjects and within subjects), but there is correlation in the within-subjects observations. In the example we expect that the observations in week three will be more similar to week two observations than to week zero observations. Because of this, the split-plot analysis (referred to as the univariate approach) is valid only under certain restrictive conditions.

We will use the artificial dataset `drug.mult`, which has the following form:

```
> drug.mult
  subject gender  Y.1  Y.2  Y.3  Y.4
1      S1      F 75.9 74.3 80.0 78.9
2      S2      F 78.3 75.5 79.6 79.2
3      S3      F 80.3 78.2 80.4 76.2
4      S4      M 80.7 77.2 82.0 83.8
5      S5      M 80.3 78.6 81.4 81.5
6      S6      M 80.1 81.1 81.9 86.4
```

The dataset consists of the two factors `subject` and `gender`, and the matrix `Y` which contains 4 columns. The first thing to do is stretch this out into a form suitable for the univariate analysis:

```
> drug.uni <- drug.mult[rep(1:6, rep(4,6)), 1:2]
> ymat <- data.matrix(drug.mult[,paste("Y.",1:4, sep="")])
> drug.uni <- cbind(drug.uni, time=ordered(rep(paste("Week",
+         0:3,sep=""),6)), y=as.vector(t(ymat)))
```

The univariate analysis treats the data as a split-plot design:

```
> summary(aov(y ~ gender*time + Error(subject), drug.uni))
Error: subject
          Df Sum of Sq  Mean Sq  F Value   Pr(F)
gender     1  60.80167 60.80167 19.32256 0.01173
Residuals  4  12.58667  3.14667

Error: Within
            Df Sum of Sq  Mean Sq  F Value     Pr(F)
time         3  49.10833 16.36944 6.316184 0.0081378
gender:time  3  14.80167  4.93389 1.903751 0.1828514
Residuals   12  31.10000  2.59167
```

Tests in the "Within" stratum are valid only if the data satisfy the "circularity" property, in addition to the usual conditions. Circularity means that the variance of the difference of measures at different times is constant; for example, the variance of the difference between the measures at week 0 and week 3 should be the same as the variance of the difference between week 2 and week 3. We also need the assumption that actual contrasts are used; for example, the `contr.treatment` function should not be used. When circularity does not hold, then the *p*-values for the tests will be too small.

One approach is to perform tests which are as conservative as possible. Conservative tests are formed by dividing the degrees of freedom in both the numerator and denominator of the F test by the number of repeated measures minus one. In our example there are four repeated measures on each subject, so we divide by 3. The split-plot and the conservative tests are:

```
> 1 - pf(6.316184, 3, 12) # usual univariate test
[1] 0.008137789
> 1 - pf(6.316184, 1, 4) # conservative test
[1] 0.06583211
```

These two tests are telling fairly different tales, so the data analyst would probably move on to one of two alternatives. A Huynh-Feldt adjustment of the degrees of freedom provides a middle ground between the tests above—see Winer, Brown and Michels (1991), for instance. The multivariate approach, discussed below, substantially relaxes the assumptions.

The univariate test for "time" was really a test on three contrasts. In the multivariate setting we want to do the same thing, so we need to use contrasts in the response:

```
> drug.man <- manova(ymat %*% contr.poly(4) ~ gender,
+                    drug.mult)
> summary(drug.man, intercept=T)
```

```
            Df Pillai Trace approx. F num df   den df    P-value
(Intercept) 1   0.832005     3.301706  3         2
0.241092
     gender 1   0.694097     1.512671  3         2
0.421731
  Residuals 4
```

The line marked "(Intercept)" corresponds to "time" in the univariate approach, and similarly the "gender" line here corresponds to "gender:time". The *p*-value of .24 is larger than either of the univariate tests—the price of the multivariate analysis being more generally valid is that quite a lot of power is lost. Although the multivariate approach is preferred when the data do not conform to the required conditions, the univariate approach is preferred when they do (the trick, of course, is knowing which is which).

Let's look at the univariate summaries that this MANOVA produces:

```
> summary(drug.man, intercept=T, univar=T)
Response: .L
            Df Sum of Sq Mean Sq  F Value     Pr(F)
(Intercept)  1    22.188 22.1880 4.327255 0.1059983
gender       1     6.912  6.9120 1.348025 0.3101900
Residuals    4    20.510  5.1275

Response: .Q
            Df Sum of Sq  Mean Sq F Value     Pr(F)
(Intercept)  1  5.415000 5.415000 5.30449 0.0826524
gender       1  4.001667 4.001667 3.92000 0.1188153
Residuals    4  4.083333 1.020833

Response: .C
            Df Sum of Sq  Mean Sq  F Value     Pr(F)
(Intercept)  1  21.50533 21.50533 13.22049 0.0220425
gender       1   3.88800  3.88800  2.39016 0.1969986
Residuals    4   6.50667  1.62667
```

If you add up the respective degrees of freedom and sums of squares, you will find that the result is the same as the univariate "Within" stratum. For this reason, the univariate test is sometimes referred to as the "average *F* test".

The above discussion has focused on classical inference, which should not be done before graphical exploration of the data.

Many books discuss repeated measures. Some examples are Hand and Taylor (1987), Milliken and Johnson (1984), Crowder and Hand (1990), Winer, Brown and Michels (1991).

14.6 RANK TESTS FOR ONE-WAY AND TWO-WAY LAYOUTS

This section briefly describes how to use two non-parametric rank tests for ANOVA: the *Kruskal-Wallis* rank sum test for a one-way layout and the *Friedman* test for unreplicated two-way layout with (randomized) blocks.

Since these tests are based on ranks, they are *robust* with regard to the presence of outliers in the data; that is, they are not affected very much by outliers. This is not the case for the classical *F* tests.

You can find detailed discussions of the Kruskal-Wallis and Friedman rank-based tests in a number of books on nonparametric tests; for example, Lehmann (1975) and Hettmansperger (1984).

The Kruskal-Wallis Rank Sum Test

When you have a one-way layout, as in the section Experiments with One Factor in the previous chapter, you can use the *Kruskal-Wallis rank sum* test `kruskal.test` to test the null hypothesis that all group means are equal.

We illustrate how to use `kruskal.test` for the blood coagulation data of 13.1. First you set up your data as for a one-factor experiment (or one-way layout). You create a vector object `coag`, arranged by factor level (or treatment), and you create a factor object `diet` whose levels correspond to the factor levels of vector object `coag`. Then use `kruskal.test`:

```
> kruskal.test(coag,diet)

        Kruskal-Wallis rank sum test

data:  coag and diet
Kruskal-Wallis chi-square = 17.0154, df = 3,
 p-value = 7e-04
alternative hypothesis: two.sided
```

The *p*-value of $p = .0007$ is highly significant. This *p*-value is computed using an asymptotic chi-squared approximation. See the help file for more details.

You may find it helpful to note that `kruskal.test` and `friedman.test` return the results of its computations, and associated information, in the same style as the functions in chapter 3, Statistical Inference for One and Two Sample Problems.

The Friedman Rank Sum Test

When you have a two-way layout with one blocking variable and one treatment variable, you can use the *Friedman rank sum* test `friedman.test` to test the null hypothesis that there is no treatment effect.

We illustrate how you use `friedman.test` for the penicillin yield data described in 13.2 of the previous chapter. The general form of the usage is

```
friedman.test(y,groups,blocks)
```

where `y` is a numeric vector, `groups` contains the levels of the treatment factor and `block` contains the levels of the blocking factor. Thus you can do:

```
> attach(pen.df) # make treatment and blend available
> friedman.test(yield, treatment, blend)

        Friedman rank sum test

data:  yield and pen.design[, 2] and pen.design[, 1]
Friedman chi-square = 3.4898, df = 3, p-value = 0.3221
alternative hypothesis: two.sided
```

The *p*-value is *p*=.32, which is not significant. This *p*-value is computed using an asymptotic chi-squared approximation. For further details on `friedman.test`, see the help file.

14.7 VARIANCE COMPONENTS MODELS

Variance components models are used when there is interest in the variability of one or more variables other than the residual error. For example, manufacturers often run experiments to see which parts of the manufacturing process contribute most to the variability of the final product. In this situation variability is undesirable, and attention is focused on improving those parts of the process that are most variable. Animal breeding is another area in which variance components models are routinely used. Some data, from surveys for example, that have traditionally been analyzed using regression can more profitably be analyzed using variance component models.

Estimating the Model

To estimate a variance component model, you first need to use `is.random` to state which factors in your data are random. A variable that is marked as being random will have a variance component in any models that contain it. Only variables that inherit from class `"factor"` can be declared random. Although `is.random` works on individual factors, it is often more practical to use it on the columns of a data frame. You can see if variables are declared random by using `is.random` on the data frame:

```
> is.random(pigment)
Batch Sample  Test
    F      F    F
```

Declare variables to be random by using the assignment form of `is.random`:

```
> is.random(pigment) <- c(T, T, T)
> is.random(pigment)
Batch Sample  Test
    T      T    T
```

Because we want all of the factors to be random, we could have simply done the following:

```
> is.random(pigment) <- T
```

The value on the right is replicated to be the length of the number of factors in the data frame.

Once you have declared your random variables, you are ready to estimate the model using the `varcomp` function. This function takes a formula and other arguments very much like `lm` or `aov`. Because the pigment data are from a nested design, the call has the following form:

```
> pigment.vc <- varcomp(Moisture ~ Batch/Sample, pigment)
> pigment.vc
Variances:
    Batch Sample %in% Batch Residuals
 7.127976          28.53333 0.9166667
Call:
varcomp(formula = Moisture ~ Batch/Sample, data = pigment)
```

The result of `varcomp` is an object of class `"varcomp"`. You can use `summary` on `"varcomp"` objects to get more details about the fit, and you can use `plot` to get QQ-plots for the normal distribution on the estimated effects for each random term in the model.

Estimation Methods

The `method` argument to `varcomp` allows you to choose the type of variance component estimator. Maximum likelihood and REML (restricted maximum likelihood) are two of the choices. REML is very similar to maximum likelihood but takes the number of fixed effects into account (the usual unbiased estimate of variance in the one-sample model is a REML estimate). See Harville (1977) for more details on these estimators.

The default method is a MINQUE (minimum norm quadratic unbiased estimate); this class of estimator is locally best at a particular spot in the parameter space. The MINQUE option in S-PLUS is locally best if all of the variance components (except that for the residuals) are zero. The MINQUE estimate agrees with REML for balanced data. See Rao (1971) for details. This method was made the default because it is less computationally intense than the other methods, however, it can do significantly worse for severely unbalanced data (Swallow and Monahan (1984)).

You can get robust estimates by using `"method=winsor"`. This method creates new data by moving outlying points or groups of points toward the rest of the data. One of the standard estimators is then applied to this possibly revised data. Burns (1992) gives details of the algorithm along with simulation results. This method uses much larger amounts of memory than

the other methods if there are a large number of random levels, such as in a deeply nested design.

Random Slope Example

We now produce a more complicated example in which there are random slopes and intercepts. The data consist of several pairs of observations on each of several individuals in the study. An example might be that the *y* values represent the score on a test and the *x* values are the time at which the test was taken.

Let's start by creating simulated data of this form. We create data for 30 subjects and 10 observations per subject:

```
> subject <- factor(rep(1:30, rep(10,30)))
> set.seed(357) # makes these numbers reproducible
> trueslope <- rnorm(30, mean=1)
> trueint <- rnorm(30, sd=.5)
> times <- rchisq(300, 3)
> scores <- rep(trueint, rep(10,30)) + times *
+     rep(trueslope, rep(10,30)) + rnorm(300)
> test.df <- data.frame(subject, times, scores)
> is.random(test.df) <- T
> is.random(test.df) subject T
```

Even though we want to estimate random slopes and random intercepts, the only variable that is declared random is `subject`. Our model for the data has two coefficients: the mean slope (averaged over subjects) and the mean intercept. It also has three variances: the variance for the slope, the variance for the intercept, and the residual variance.

The following command estimates this model using Maximum Likelihood (the default MINQUE is not recommended for this type of model):

```
> test.vc <- varcomp(scores ~ times * subject,
+     data=test.df, method="ml")
```

This seems very simple. We can see how it works by looking at how the formula get expanded. The right side of the formula is expanded into four terms:

```
scores ~ 1 + times + subject + times:subject
```

The intercept term in the formula, represented by `1`, gives the mean intercept. The variable `times` is fixed and produces the mean slope. The `subject` variable is random and produces the variance component for the random intercept. Since any interaction containing a random variable is considered random, the last term, `times:subject`, is also random; this term gives the variance component for the random slope. Finally, there is always a residual variance.

Now we can look at the estimates:

```
> test.vc
Variances:
   subject times:subject Residuals
 0.3162704      1.161243 0.8801149
Message:
[1] "RELATIVE FUNCTION CONVERGENCE"
Call:
varcomp(formula = scores ~ times*subject, data=test.df,
  method = "ml")
```

This shows the three variance components. The variance of the intercept, which has true value .25, is estimated as .32. Next, labeled `times:subject` is the variance of the slope, and finally the residual variance. We can also view the estimates for the coefficients of the model, which have true values of 0 and 1.

```
> coef(test.vc)
 (Intercept)   times
   0.1447211 1.02713
```

14.8 REFERENCES

Burns, P. J. (1992). *Winsorized REML estimates of variance components*. Submitted.

Chambers, J.M. and Hastie, T.J. (1992). *Statistical Models in S*. Wadsworth and Brooks Cole Advanced Books and Software, Pacific Grove, CA.

Crowder, M. J. and Hand, D. J. (1990). *Analysis of Repeated Measures*. Chapman and Hall, London.

Daniel, C. (1976). *Applications of Statistics to Industrial Experimentation*. Wiley, New York.

Hand, D. J. and Taylor, C. C. (1987). *Multivariate Analysis of Variance and Repeated Measures*. Chapman and Hall, London.

Harville, D. A. (1977). *Maximum likelihood approaches to variance component estimation and to related problems (with discussion)*. Journal of the American Statistical Association, 72:320–340.

Hettmansperger, T.P. (1984). *Statistical Inference Based on Ranks*. John Wiley, New York.

Hicks, C. R. (1973). *Fundamental Concepts in the Design of Experiments*. Holt, Rinehart and Winston, New York.

Lehmann, E.L. (1975). *Nonparametrics: Statistical Methods Based on Ranks*.

Holden-Day, San Francisco.

Mardia, K. V. and Kent, J. T. and Bibby, J. M. (1979). *Multivariate Analysis*. Academic Press, London.

Milliken, G. A. and Johnson, D. E. (1984). *Analysis of Messy Data Volume 1: Designed Experiments*. Van Nostrand Reinhold, New York.

Rao, C. R. (1971). *Estimation of variance and covariance components—MINQUE theory*. Journal of Multivariate Analysis, 1:257–275.

Seber, G. A. F. (1984). *Multivariate Observations*. Wiley, New York.

Swallow, W. H. and Monahan, J. F. (1984). *Monte Carlo comparison of ANOVA, MIVQUE, REML, and ML estimators of variance components*. Technometrics, 26:47–57.

Winer, B. J. and Brown, D. R. and Michels, K. M. (1991). *Statistical Principles in Experimental Design*. McGraw-Hill, New York.

MULTIPLE COMPARISONS 15

Multiple comparisons are used to compare treatment levels or combinations, following an ANOVA.

MULTIPLE COMPARISONS

This chapter describes the use of the function `multicomp` in the analysis of multiple comparisons. Section 15.1 describes simple calls to `multicomp` for standard comparisons in one-way layouts. Section 15.2 tells how to use `multicomp` for nonstandard designs and comparisons. In section 15.3, the capabilities and limitations of this function are summarized.

15.1 OVERVIEW

When an experiment has been carried out in order to compare effects of several treatments, a classical analytical approach is to begin with a test for equality of those effects. Regardless of whether one embraces this classical strategy, and regardless of the outcome of this test, one is usually not finished with the analysis until determining where any differences exist, and how large the differences are (or might be); that is, until one does multiple comparisons of the treatment effects.

As a simple start, consider the built-in S-PLUS data frame on fuel consumption of vehicles, `fuel.frame`. Each row provides the fuel consumption (`Fuel`) in 100*gallons/mile for a vehicle model, as well as the `Type` group of the model: Compact, Large, Medium, Small, Sporty, or Van. There is also information available on the `Weight` and `Displacement` of the vehicle. Figure 15.1 shows a boxplot of fuel consumption, the result of the following commands.

```
> attach(fuel.frame)
> boxplot(split(Fuel,Type))
```

Not surprisingly, the plot suggests that there are differences between vehicle types in terms of mean fuel consumption. This is confirmed by a one-factor analysis of variance test of equality obtained by a call to `aov`.

```
> aovout.fuel <- aov( Fuel ~ Type, data = fuel.frame)
> anova(aovout.fuel)
Analysis of Variance Table
Response: Fuel
Terms added sequentially (first to last)

          Df  Sum of Sq    Mean Sq    F Value          Pr(F)
Type       5   24.23960   4.847921   27.22058   1.220135e-13
Residuals 54    9.61727   0.178098
```

The boxplots show some surprising patterns, and inspire some questions. Do Small cars really have lower mean fuel consumption than Compact cars? If so, by what amount? What about Small versus Sporty cars? Vans versus Large

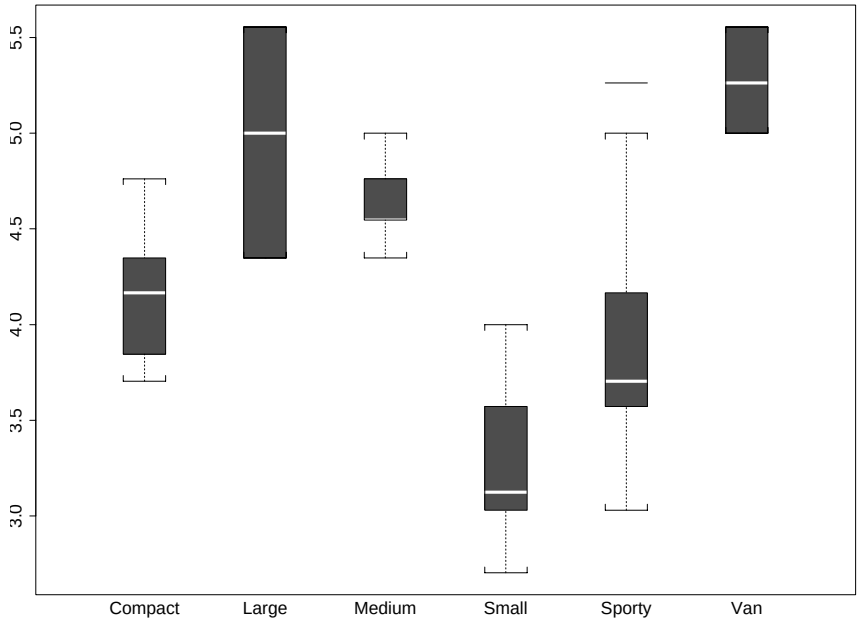

Figure 15.1: *Fuel consumption boxplot.*

cars? Answers to these questions are offered by an analysis of all pairwise differences in mean fuel consumption, which can be obtained from a call to `multicomp`:

```
> mca.fuel <- multicomp(aovout.fuel, focus = "Type")
> plot(mca.fuel)
> mca.fuel
95 % simultaneous confidence intervals for specified
linear combinations, by the Tukey method
critical point: 2.9545
response variable: Fuel
intervals excluding 0 are flagged by '****'

                Estimate      Std.     Lower     Upper
                            Error     Bound     Bound
Compact-Large     -0.800     0.267    -1.590   -0.0116  ****
Compact-Medium    -0.434     0.160    -0.906    0.0387
Compact-Small      0.894     0.160     0.422    1.3700  ****
Compact-Sporty     0.210     0.178    -0.316    0.7360
Compact-Van       -1.150     0.193    -1.720   -0.5750  ****
Large-Medium       0.366     0.270    -0.432    1.1600
Large-Small        1.690     0.270     0.896    2.4900  ****
```

```
Large-Sporty        1.010     0.281     0.179    1.8400  ****
Large-Van          -0.345     0.291    -1.210    0.5150
Medium-Small        1.330     0.166     0.839    1.8200  ****
Medium-Sporty       0.644     0.183     0.103    1.1800  ****
Medium-Van         -0.712     0.198    -1.300   -0.1270  ****
Small-Sporty       -0.684     0.183    -1.220   -0.1440  ****
Small-Van          -2.040     0.198    -2.620   -1.4600  ****
Sporty-Van         -1.360     0.213    -1.980   -0.7270  ****
```

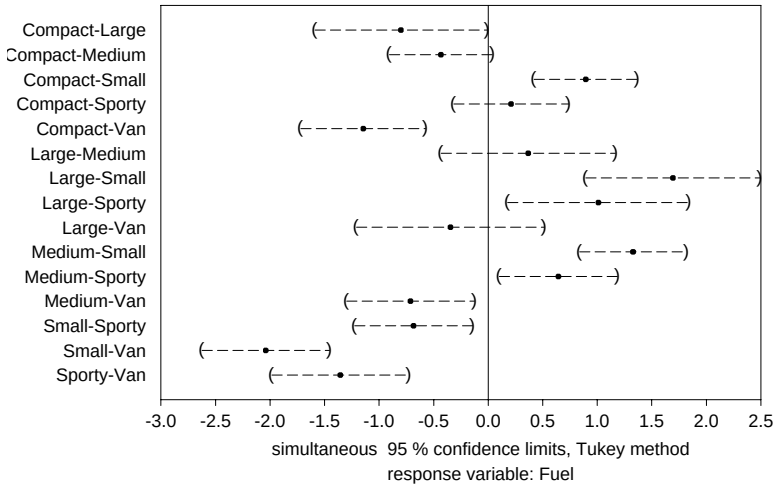

Figure 15.2: *Fuel consumption ANOVA.*

As the output and plot in figure 15.2 indicate, this default call to `multicomp` has resulted in the calculation of simultaneous 95% confidence intervals for all pairwise differences between vehicle `Fuel` means, based on the levels of `Type`, sometimes referred to as MCA comparisons (Hsu, 1996). The labeling states that Tukey's method (Tukey, unpublished report, Princeton University, 1953) has been used; since group sample sizes are unequal, this is actually equivalent to what is commonly known as the Tukey-Kramer (Kramer, 1956) multiple comparison method.

Honestly Significant Differences

The output indicates via asterisks the confidence intervals which exclude zero; in the plot, these can be identified by noting intervals that do not intersect the vertical reference line at zero. These identified statistically significant comparisons correspond to pairs of (long run) means which can be declared different by Tukey's "HSD" (honestly significant difference) method. Not surprisingly, we can assert that most of the vehicle types have different mean fuel consumption rates. If we require 95% confidence in all of

our statements, we cannot claim different mean fuel consumption rates between the Compact and Medium types, the Compact and Sporty types, the Large and Medium types, and the Large and Van types. Note we should *not* assert that these pairs have *equal* mean consumption rates; for example, the interval for Compact-Medium states that this particular difference in mean fuel consumption is between -0.906 and 0.0387 units. Hence, the Medium vehicle type may have larger mean fuel consumption than the Compact, by as much as 0.9 units. Only an engineer can judge the importance of a difference of this size; if it is considered trivial, then using these intervals we can claim that for all *practical* purposes these two types have equal mean consumption rates; if not, there may still be an important difference between these types, and we would need more data to resolve the question.

The point to the above discussion is that there is more information in these simultaneous intervals than is provided by a collection of significance tests for differences. This is true whether the tests are reported via conclusions "Reject"/"Do not reject", or via *p*-values or adjusted *p*-values. This superior level of information using confidence intervals has been acknowledged by virtually all modern texts on multiple comparisons (Hsu, 1996; Bechhofer *et al*, 1996; Hochberg and Tamhane, 1987; Toothaker, 1993). All multiple comparison analyses using `multicomp` are represented by using confidence intervals or bounds.

Rat Growth Hormone Treatments

If all the intervals are to hold simultaneously with a given confidence level, it is important to calculate intervals only for those comparisons which are truly of interest. For example, consider the summary data in table 15.1 from Hsu (Hsu, 1996) concerning a study by Juskevich and Guyer (1990) in which rat growth was studied under several growth-hormone treatments.

In this setting, it may only be necessary to compare each hormone treatment's mean growth with that of the placebo (that is, the oral administration with zero dose). These all-to-one comparisons are usually referred to as multiple comparisons with a control (MCC) (Dunnett, 1955). Suppose that the raw data for each rat were available in a data frame `hormone.dfr` with variables `growth` (numeric) and `treatment` (a factor object) for each rat. Then the following statements would calculate, print, and plot Dunnett's intervals:

```
> aovout.growth <- aov(growth~treatment, data=hormone.dfr)
> multicomp(aovout.growth, focus = "treatment",
+ comparisons = "mcc", control = 1, plot = T)
```

The results are shown graphically in figure 15.3. The intervals clearly show that only the injection method is distinguishable from the placebo in terms of long run mean weight gain.

Table 15.1: *Mean weight gain in rats under hormone treatments.*

method /dose	mean growth (g)	std.dev.	sample size
oral, 0	324	39.2	30
inject,1.0	432	60.3	30
oral,0.1	327	39.1	30
oral,0.5	318	53.0	30
oral,5	325	46.3	30
oral,50	328	43.0	30

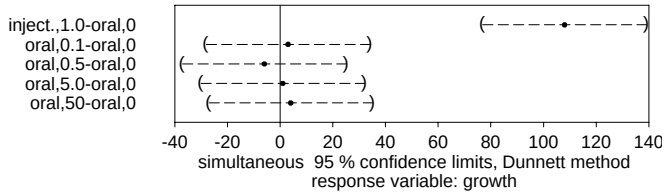

Figure 15.3: *MCC for rat hormone treatments.*

More detail on multicomp

The first and only required argument to `multicomp` is an `aov` object (or equivalent), the results of a fixed-effects linear model fit by `aov` or a similar model-fitting function. The `focus` argument, when specified, names a factor (a main effect) in the fitted `aov` model. Comparisons will then be calculated on (adjusted) means for levels of the `focus` factor. The `comparisons` argument is an optional argument which can specify a standard family of comparisons for the levels of the `focus` factor. The default is `comparisons = "mca"`, which creates all pairwise comparisons. Setting `comparisons = "mcc"` creates all-to-one comparisons relative to the level specified by the `control` argument. The only other `comparisons` option available is `"none"`, which states that the adjusted means themselves are of interest (with no differencing), in which case the default method for interval calculation is known as the studentized maximum modulus method. Other kinds of comparisons and different varieties of adjusted means can be specified through the `lmat` and `adjust` options discussed below.

Upper and Lower Bounds

Confidence intervals provide both upper and lower bounds for each difference or adjusted mean of interest. In some instances, only the lower bounds, or only the upper bounds, may be of interest. For example, in the

fuel consumption example earlier, we may only be interested in determining which types of vehicle clearly have greater fuel consumption than compacts, and in calculating lower bounds for the difference. This can be accomplished through lower mcc bounds:

```
> aovout.fuel<-aov(Fuel~Type, data=fuel.frame)
> multicomp(aovout.fuel, focus="Type",comparison="mcc",
+           bounds="lower", control=1, plot=T)
95 % simultaneous confidence bounds for specified
linear combinations, by the Dunnett method

critical point: 2.3332000000000002
response variable: Fuel

bounds excluding 0 are flagged by '****'

               Estimate Std.Error Lower Bound
 Large-Compact    0.800     0.267      0.1770 ****
Medium-Compact    0.434     0.160      0.0606 ****
 Small-Compact   -0.894     0.160     -1.2700
Sporty-Compact   -0.210     0.178     -0.6250
   Van-Compact    1.150     0.193      0.6950 ****
```

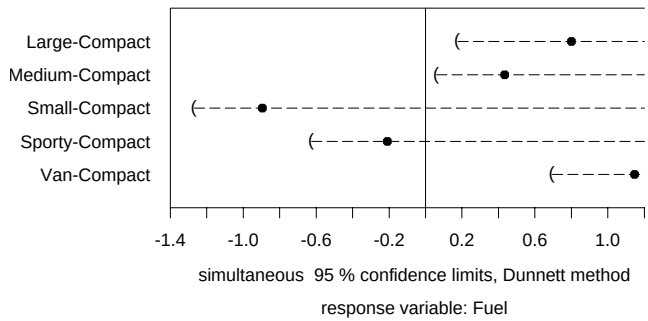

Figure 15.4: *Lower mcc bounds for rat hormone treatments.*

The intervals or bounds computed by `multicomp` are always of the form

$$(\text{estimate}) \pm (\text{critical point}) \times (\text{standard error of estimate})$$

The reader has probably already noticed that the estimates and standard errors are supplied in the output table. The critical point used depends on the specified or implied multiple comparison `method`.

Calculation of Critical Points

The `multicomp` function can calculate critical points for simultaneous intervals or bounds by the following methods:

- Tukey (`method = "tukey"`),
- Dunnett (`method = "dunnett"`),
- Sidak (`method = "sidak"`),
- Bonferroni (`method = "bon"`),
- Scheffé (`method = "scheffe"`)
- Simulation-based (`method = "sim"`).

Non-simultaneous intervals use the ordinary Student's-t critical point, `method = "lsd"`. If the user specifies a `method`, the function will check its validity in view of the model fit and the types of comparisons requested. For example, `method = "dunnett"` will be invalid if `comparisons = "mca"`. If the specified method does not satisfy the validity criterion, the function terminates with a message to that effect. This safety feature can be disabled by specifying the optional argument `valid.check = F`. If no `method` is specified, the function uses the smallest critical point among the valid non-simulation-based methods. If the user specifies `method = "best"`, the function uses the smallest critical point among all valid methods including simulation; this latter method may take a few moments of computer time.

The simulation-based method generates a near-exact critical point via Monte Carlo simulation, as discussed by Edwards and Berry (1987). For nonstandard families of comparisons or unbalanced designs, this method will often be substantially more efficient than other valid methods. The simulation size is set by default to provide a critical point whose actual error rate is within 10% of the nominal α (with 99% confidence). This amounts to simulation sizes in the tens of thousands for most choices of α. The user may directly specify a simulation size via the `simsize` argument to `multicomp`, but smaller simulation sizes than the default are not advisable. It is important to note that if the simulation-based method is used, the critical point (and hence the intervals) will vary slightly over repeated calls; recalculating the intervals repeatedly searching for some desirable outcome will usually be fruitless, and will result in intervals which do not provide the desired confidence level.

Error Rates for Confidence Intervals

Other `multicomp` arguments of interest are the `alpha` argument which specifies the error rate for the intervals or bounds, with default `alpha = .05`. By default, `alpha` is a familywise error rate, that is, the user may be (1 - `alpha`) x 100% confident that *every* calculated bound holds. If the user

desires confidence intervals or bounds without simultaneous coverage, specify `error.type = "cwe"`, meaning comparisonwise error rate protection; in this case the user must also specify `method = "lsd"`. Finally, for users familiar with the Scheffé (1953) method, the critical point is of the form:

```
sqrt(Srank*qf(1-alpha, Srank, df.residual))
```

The numerator degrees of freedom `Srank` may be directly specified as an option. If omitted, it is computed based on the specified comparisons and `aov` object.

15.2 ADVANCED APPLICATIONS

In the first example, the `Fuel` consumption differences found between vehicle types are almost surely attributable to differences in `Weight` and/or `Displacement`. Figure 15.5 shows a plot of `Fuel` versus `Weight` with plotting symbols identifying the various model types:

```
> plot(Weight,Fuel,type = 'n')
> text(Weight,Fuel,abbreviate(as.character(Type)))
```

This plot shows a strong, roughly linear relationship between `Fuel`

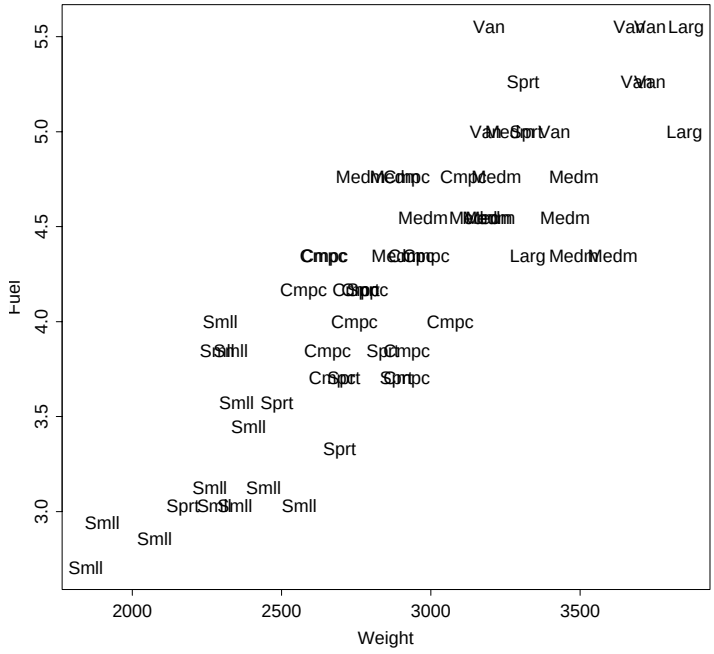

Figure 15.5: *Fuel consumption verses Weight.*

consumption and `Weight`, suggesting the addition of `Weight` as a covariate in the model. Though it may be inappropriate to compare adjusted means for all six vehicle types (see below), for the sake of example the following calls fit this model and calculates simultaneous confidence intervals for all pairwise differences of adjusted means, requesting the best valid method:

```
> lmout.fuel.ancova <- lm(Fuel ~ Type+Weight,
+ data = fuel.frame)
> multicomp(lmout.fuel.ancova, focus = "Type",
+ method = "best", plot = T)
```

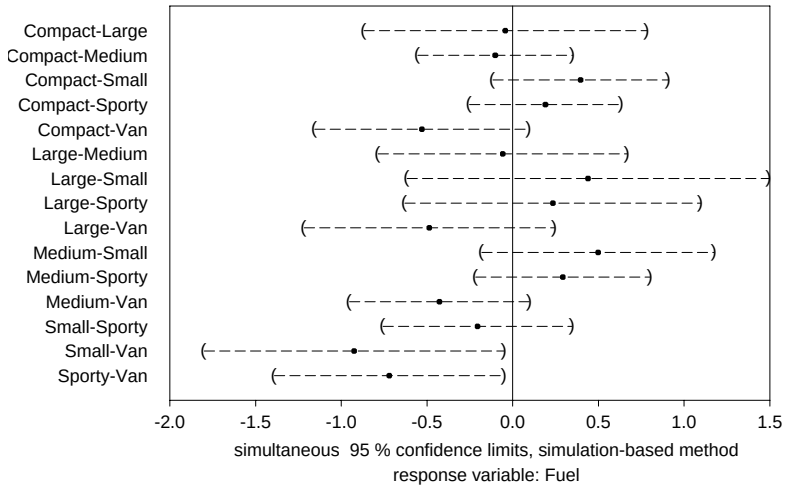

Figure 15.6: *Fuel consumption ANCOVA (adj. for Weight).*

The "best" valid method for this particular setting is the simulation-based method; Tukey's method has not been shown to be valid in the presence of covariates when there are more than three treatments. The intervals show that, adjusting for weight, the mean fuel consumption of the various vehicle types are in most cases within one unit of each other. The most notable exception is the Van type, which is showing higher mean fuel consumption than the Small and Sporty types, and most likely higher than the Compact, Medium and Large types.

Adjustment Schemes

When there is more than one term in the `lm` model, `multicomp` calculates standard adjusted means for levels of the `focus` factor and then takes differences as specified by the `comparisons` argument. Covariates are adjusted to their grand mean value. If there are other factors in the model, the standard adjusted means for levels of the `focus` factor use the average effect over the levels of any other (non-nested) factors. This adjustment

scheme can be changed using the `adjust` argument, which specifies a list of adjustment levels for non-`focus` terms in the model. Any terms excluded from the `adjust` list are adjusted in the standard way. The `adjust` list may include multiple adjustment values for each term; a full set of adjusted means for the `focus` factor is calculated for each combination of values specified by the `adjust` list. Differences (if any) specified by the `comparisons` argument are then calculated for each combination of values specified by the `adjust` list.

Toothaker's Two Factor Design

Besides allowing the user to specify covariate values for adjustment, the `adjust` argument can be used to calculate"simple effects" comparisons when factors interact, or (analogously) when covariate slopes are different. This is probably best illustrated by an example: Toothaker (1993) discusses a two-factor design, using the data collected by Frank (1984). Subjects are undergraduate females, with response the score on a 20-item multiple choice test over a taped lecture. Factors are cognitive style (*cogstyle*, levels FI = *Field independent* and FD = *Field dependent*) and study technique (*studytech*: NN = no notes, SN = student notes, PO = partial outline supplied, CO = complete outline). The following code fits the model and performs a standard two-factor analysis of variance.

```
> score <- c(13, 13, 10, 16, 14, 11, 13, 13, 11, 16, 15, 16,
+ 10, 15, 19, 19, 17, 19, 17, 20, 17, 18, 17, 18, 18, 19, 19,
+ 18, 17, 19, 17, 19, 17, 19, 17, 15, 18, 17, 15, 15, 19, 16,
+ 17, 19, 15, 20, 16, 19, 16, 19, 19, 18, 11, 14, 11, 10, 15,
+ 10, 16, 16, 17, 11, 16, 11, 10, 12, 16, 16, 17, 16, 16, 16,
+ 14, 14, 16, 15, 15, 15, 18, 15, 15, 14, 15, 18, 19, 18, 18,
+ 16, 16, 18, 16, 18, 19, 15, 16, 19, 18, 19, 19, 18, 17, 16,
+ 17, 15)
> cogstyle <- factor(c(rep("FI",52), rep("FD",52)))
> studytec <- factor(c(rep("NN",13), rep("SN", 13),
+   rep("PO",13), rep("CO",13), rep("NN",13), rep("SN", 13),
+   rep("PO",13), rep("CO",13)))
> interaction.plot(cogstyle,studytec,score)
> aovout.students <- aov( score ~ cogstyle*studytec)
> anova(lmout.students)
Analysis of Variance Table
Response: score
Terms added sequentially (first to last)
              Df   Sum of Sq   Mean Sq   F Value     Pr(F)
cogstyle       1      25.0096   25.0096   7.78354 0.00635967
studytec       3     320.1827  106.7276  33.21596 0.00000000
cogstyle:studytec 3   27.2596    9.0865   2.82793 0.04259714
Residuals      96     308.4615    3.2131
```

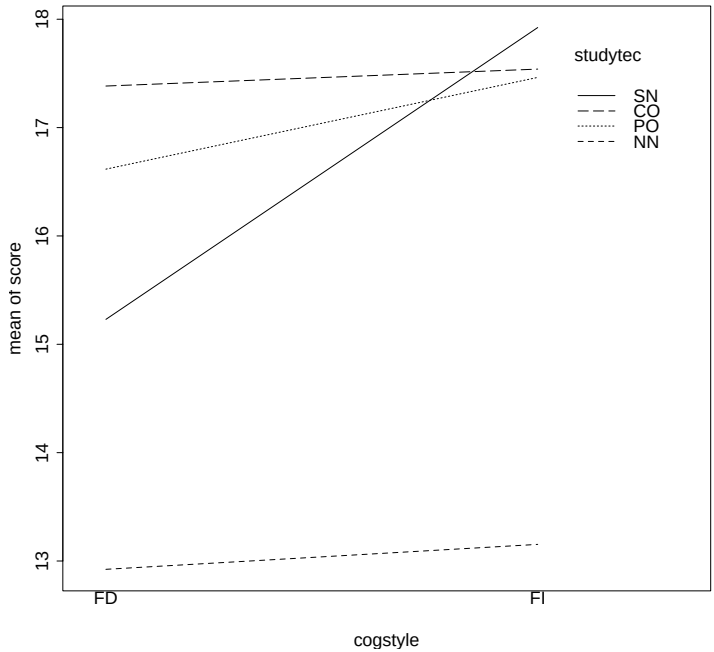

Figure 15.7: *Two-factor design test scores.*

It is apparent from the test for interaction and the profile plot that there is non-negligible interaction between these factors. In such cases it will often be of interest to follow the tests with an analysis of "simple effects," in this case a comparison of the four study techniques performed separately for each cognitive style group. The following call calculates simultaneous 95% intervals for these differences by the best valid method, which is again simulation.

```
> mcout.students <- multicomp(aovout.students,
+   focus = "studytech", adjust = list(cogstyle =
+   c("FI","FD") ), method = "best")
> plot(mcout.students)
> mcout.students
95 % simultaneous confidence intervals for specified
linear combinations, by the simulation-based method
critical point: 2.8774
response variable: score
simulation size= 12616
```

	Estimate	Std. Error	Lower Bound	Upper Bound	
CO-NN.adj1	4.3800	0.703	2.360	6.410	****
CO-PO.adj1	0.0769	0.703	-1.950	2.100	
CO-SN.adj1	-0.3850	0.703	-2.410	1.640	
NN-PO.adj1	-4.3100	0.703	-6.330	-2.280	****
NN-SN.adj1	-4.7700	0.703	-6.790	-2.750	****
PO-SN.adj1	-0.4620	0.703	-2.480	1.560	****
CO-NN.adj2	4.4600	0.703	2.440	6.480	****
CO-PO.adj2	0.7690	0.703	-1.250	2.790	
CO-SN.adj2	-2.3100	0.703	-4.330	-0.285	****
NN-PO.adj2	-3.6900	0.703	-5.720	-1.670	****
NN-SN.adj2	-2.3100	0.703	-4.330	-0.285	****
PO-SN.adj2	1.3800	0.703	-0.638	3.410	

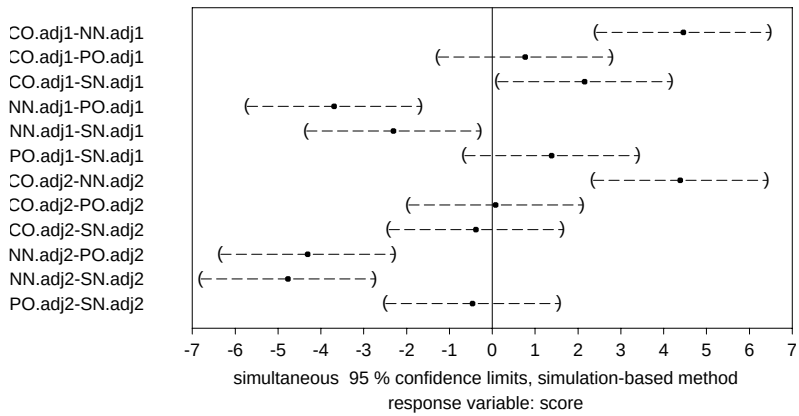

Figure 15.8: *Simple effects for study techniques.*

Setting Linear Combinations of Effects

In many situations, the setting calls for inference on a collection of comparisons or linear combinations other than those available through specifications of the `focus`, `adjust`, and `comparisons` arguments. The `lmat` argument to `multicomp` allows the user to directly specify any collection of linear combinations of the model effects for inference. `lmat` is a matrix (or an expression evaluating to a matrix) whose columns specify linear combinations of the model effects for which confidence intervals or bounds are desired. Specified linear combinations are checked for estimability; if inestimable, the function terminates with a message to that effect. The user may disable this safety feature by specifying the optional argument `est.check = F`. Specification of `lmat` overrides any `focus` or `adjust` arguments; at least one of `lmat` or `focus` must be specified. Differences requested or implied by the `comparisons` argument are taken over the

columns of `lmat`. In many instances no such further differencing would be desired, in which case the user should specify `comparisons = "none"`.

Textbook Parameterization

Linear combinations in `lmat` use the "textbook parameterization" of the model. For example, the fuel consumption analysis of covariance model parameterization has eight parameters: an `Intercept`, six coefficients for the factor `Type (Compact, Large, Medium, Small, Sporty, Van)` and a coefficient for the covariate `Weight`. Note that the levels of the factor object `Type` are listed in alphabetical order in the parameter vector.

In the Fuel consumption problem, many would argue that it is not appropriate to compare, for example, adjusted means of Small vehicles and Large vehicles, since these two groups' weights do not overlap. Inspection of figure 15.5 shows that, under this consideration, comparisons are probably only appropriate within two weight groups: Small, Sporty, and Compact as a small weight group; Medium, Large, and Van as a large weight group. We can accomplish comparisons within the two `Weight` groups using the following matrix, which is assumed to be pre-typed in a text file `"lmat.fuel"`. Note the column labels, which will be used to identify the intervals in the created figure and plot:

Table 15.2: *The Weight comparison matrix in the file* `lmat.fuel`.

	Com-Sma	Com-Spo	Sma-Spo	Lar-Med	Lar-Van	Med-Van
Intercept	0	0	0	0	0	0
Compact	1	1	0	0	0	0
Large	0	0	0	1	1	0
Medium	0	0	0	-1	0	1
Small	-1	0	1	0	0	0
Sporty	0	-1	-1	0	0	0
Van	0	0	0	0	-1	-1
Weight	0	0	0	0	0	0

The code below creates the intervals. If we restrict attention to these comparisons only, we cannot assert any differences in adjusted mean fuel consumption.

```
> multicomp.lm( lmout.fuel.ancova, lmat = lmat.fuel,
+ comparisons = "none", method = "best", plot = T)
```

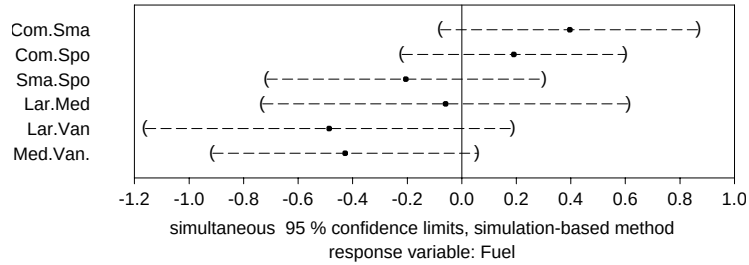

Figure 15.9: *Using lmat for specialized contrasts.*

The textbook parameterizations for linear models are created according to the following algorithm:

1. An Intercept parameter is included first, if the model contains one.
2. For each "main effect" term in the model (terms of order one), groups of parameters are included in the order the terms are listed in the model specification. If the term is a factor, a parameter is included for each level. If the term is numeric, a parameter is included for each column of its matrix representation.
3. Parameters for terms of order 2 (e.g. A:B) are created by "multiplying" the parameters of each main effect in the term, in left-to-right order. For example, if A has levels A1, A2 and B has levels B1, B2, B3, the parameters for A:B are A1B1 A1B2 A1B3 A2B1 A2B2 A2B3.
4. Parameters for higher level terms are created by multiplying the parameterizations of lower level terms two at a time, left to right. For example, the parameters for A:B:C are those of A:B multiplied by C.

Over-parameterized Models

The textbook parameterization will often be awkwardly overparameterized. For example, the 2 x 4 factorial model specified in the student study techniques example has the following parameters, in order; note the alphabetical rearrangement of the factor levels:

- Intercept
- FD FI
- CO NN PO SN
- FDCO FDNN FDPO FDSN FICO FINN FIPO FISN

Clearly, care must be taken in creating an `lmat` for factorial designs, especially with crossed and/or nested terms. The flexibility `lmat` provides for

creating study-specific linear combinations can be extremely valuable, though. If you are in doubt about the actual "textbook parameterization" of a given linear model, it may help to run a standard analysis and inspect the `lmat` created, which is part of the output list of `multicomp`. For example, for the simple effects analysis of the student test scores of figure 15.8, the implied `lmat` can be seen using the command:

```
> mcout.students$lmat
```

Multicomp Methods Compared

The function `multicomp.lm`, after checking estimability of specified linear combinations and creating a vector of estimates, a covariance matrix, and degrees of freedom, calls the "base" function `multicomp.default`. The function `multicomp.default` will be directly valuable in many settings. It uses a vector of estimates `bvec` and associated covariance matrix `vmat` as required arguments, with optional degrees of freedom `df.residual` (possibly `Inf`, the default) to calculate confidence intervals on linear combinations of `bvec`. These linear combinations can be specified through an optional `lmat` argument and/or `comparisons` argument; there is neither a `focus` nor an `adjust` argument. Linear combinations of `bvec` defined by columns of `lmat` (if any; the default `lmat` is an identity matrix) are calculated, followed by any differences specified or implied by the `comparisons` argument. The `multicomp.lm` options `method`, `bounds`, `alpha`, `error.type`, `crit.point`, `sim.size`, `Srank`, `valid.check`, and `plot` are also available in `multicomp.default`.

The function `multicomp.default` can be very useful as a means of calculating intervals based on summary data, or using the results of some model-fitting program other than `lm`; `bvec` must be considered as a realization of a multivariate normal vector. If the matrix `vmat` incorporates any estimate of variance considered to be a realized chi-square variable, the degrees of freedom `df.residual` must be specified.

The rat growth data discussed earlier (Table 15.1) provides a simple example of the use of `multicomp.default`. Here, the first few statements create the vector of estimates `bvec` and covariance matrix `vmat` assuming that a single factor analysis of variance model is appropriate for the data, followed by the statement that produced the lower mcc bounds of figure 15.4:

```
> growth <- c(324, 432, 327, 318, 325, 328)
> stddev <- c(39.2, 60.3, 39.1, 53.0, 46.3, 43.0)
> samp.size <- rep(30,6)
> names(growth) <- c( "oral,0", "inject,1.0", "oral,0.1",
+   "oral,0.5", "oral,5", "oral,50")
> mse <- mean(stddev^2)
```

```
> vmat <-mse*diag(1/samp.size)
> multicomp.default(growth, vmat, df.residual =
+   sum(samp.size-1), comparisons = “mcc”, bounds = “lower”,
+   control = 1, plot = T)
```

15.3 CAPABILITIES AND LIMITS

In summary, the function `multicomp` uses the information in a linear model; that is, a fitted fixed effects linear model. Through some combination of the `focus`, `adjust`, `comparisons` and `lmat` arguments, any collection of estimable linear combinations of the fixed effects may be estimated, and simultaneous or non-simultaneous intervals or bounds computed by any of the applicable methods mentioned above. Specified linear combinations are checked for estimability unless the user specifies `est.check = F`. Specified methods are checked for validity unless the user specifies `valid.check = F`.

The function `multicomp.default` uses a specified vector of parameter estimates `bvec` and a covariance matrix `vmat`, which will usually have some associated degrees of freedom `df.residual` specified. Possibly through some combination of the `comparisons` or `lmat` arguments, any collection of linear combinations of the parameters may be estimated, and simultaneous or non-simultaneous intervals or bounds computed by any of the applicable methods discussed above. Specified methods are checked for validity unless the user specifies `valid.check = F`.

The output from either procedure is an object of class "`multicomp`", a list containing elements `$table` (a matrix of calculated linear combination estimates, standard errors, and lower and/or upper bounds), `$alpha`, `$error.type`, `$method`, `$crit.point`, `$lmat` (the final matrix of linear combinations specified or implied), and other ancillary information pertaining to the intervals. If the argument `plot = T` is specified, the intervals/bounds are plotted on the active device. If not, the created `multicomp` object can be used as an argument to `plot` (see `plot.multicomp`).

The critical points for the methods of Tukey and Dunnett are calculated by numerically using the S-PLUS quantile functions `qtukey`, `qdunnett`, `qmvt`, and `qmvt.sim`, which may be directly useful to advanced users for their own applications.

What the function `multicomp` does *not* do:

1. Any stagewise or multiple range test. The simultaneous testing procedures attributed to Fisher, Tukey, Scheffé, Sidak and Bonferroni are implied by the use of the corresponding `method` and noting which of the calculated intervals excludes zero. The multiple range tests of Duncan(1955) and Newman(1959)-Keuls(1952) do not provide familywise error protection, and are not very efficient for comparisonwise error protection; modern texts on multiple comparisons recommend uniformly against these two multiple range tests (Hsu, 1996; Hochberg and Tamhane, 1987; Bechofer *et al*, 1996; Toothaker 1993).

2. Multiple comparisons with the "best" treatment (MCB; Hsu, 1996, chapter 4), or any ranking and selection procedure (Bechofer *et al*, 1996) other than selection of treatments better than a control implied by Dunnett's one-sided methods. Users familiar with these methods and reasonably proficient at S-PLUS programming will be able to code many of these procedures through creative use of `multicomp` with the `comparisons = "mcc"` option.

15.4 REFERENCES

Bechhofer, Robert E., Thomas J. Santner, and David M. Goldsman (1995). *Design and Analysis of Experiments for Statistical Selection, Screening, and Multiple Comparisons.* New York: Wiley.

Duncan, D.B. (1955), "Multiple range and multiple F tests," *Biometrics* 11, 1-42.

Dunnett, C.W. (1955), "A multiple comparison procedure for comparing several treatments with a control," *J.Amer.Stat.Assoc.* 50, 1096-1121.

Edwards, Don and Berry, Jack J. (1987), "The efficiency of simulation-based multiple comparisons," *Biometrics* 43, 913-928.

Frank, B.M. (1984). "Effect of field independence-dependence and study technique on learning from a lecture," *Amer.Educ.Res.J.* 21, 669-678.

Hsu, Jason C. (1996). *Multiple Comparisons: Theory and Methods*. London: Chapman and Hall.

Hochberg, Y. and Tamhane, A.C. (1987). *Multiple Comparison Procedures*. New York: Wiley.

Juskevich, J.C. and Guyer, C.G. (1990). "Bovine growth hormone: human food safety evaluation," *Science* 249, 875-884.

Kramer, C.Y. (1956). "Extension of multiple range tests to group means with unequal numbers of replications," *Biometrics* 12, 309-310.

Keuls, M. (1952). "The use of the 'studentized range' in connection with an analysis of variance," *Euphytica* 1, 112-122.

Newman, D. (1939). "The distribution of the range in samples from a normal population, expressed in terms of an independent estimate of standard deviation," *Biometrika* 35, 16-31.

Scheffé, H. (1953). "A method for judging all contrasts in the analysis of variance," *Biometrika* 40, 87-104.

Sidak, A. (1967). "Rectangular confidence regions for the means of multivariate normal distributions," *J.Amer.Stat.Assoc*. 62, 626-633.

Toothaker, Larry E. (1993). *Multiple comparison procedures*. London: Sage publications.

PRINCIPAL COMPONENTS ANALYSIS

16

For a large number of variables it is often easier to consider only a smaller number of combinations of the original data.

PRINCIPAL COMPONENTS ANALYSIS

For investigations involving a large number of observed variables, it is often useful to simplify the analysis by considering a smaller number of *linear combinations* of the original variables. For example, scholastic achievement tests typically consist of a number of examinations in different subject areas. In attempting to rate students applying for admission, college administrators frequently attempt to reduce the scores from all subject areas to a single, overall score. If the reduction can be done with minimal information loss, all the better.

One obvious choice for the overall score is the mean over all subject areas. For three subject areas s_1, s_2, and s_3, the mean corresponds to the linear combination $\frac{1}{3}s_1 + \frac{1}{3}s_2 + \frac{1}{3}s_3$, or equivalently $\boldsymbol{l}'\boldsymbol{s}$, where $\boldsymbol{l}$ is the vector of coefficients $\left(\frac{1}{3}, \frac{1}{3}, \frac{1}{3}\right)'$. A linear combination with $\sum l_i^2 = 1$ is called a *standardized linear combination*, or SLC. By restricting attention to SLCs, you can make meaningful comparisons between various choices of linear combinations. For example, with the test scores, you can seek the combination with the greatest variance as a way of ranking the students and separating them.

Principal components analysis finds a set of SLCs, called the principal components, which are orthogonal and taken together explain all the variance of the original data. The principal components are defined as follows (from Mardia, Kent, and Bibby (1979)):

> If $\boldsymbol{x}$ is a random vector with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, then the *principal component transformation* is the transformation
>
> $$\boldsymbol{x} \rightarrow \boldsymbol{y} = \boldsymbol{\Gamma}(\boldsymbol{x} - \boldsymbol{\mu}),$$
>
> where Γ is orthogonal, $\boldsymbol{\Gamma}\boldsymbol{\Sigma}\boldsymbol{\Gamma} = \boldsymbol{\Lambda}$ is diagonal, and $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_p \geq 0 \ldots$. The *ith principal component* of $\boldsymbol{x}$ may be defined as the *i*th element of the vector $\boldsymbol{y}$, namely as
>
> $$y_i = \gamma'_{(i)}(\boldsymbol{x} - \boldsymbol{\mu}).$$
>
> Here $\boldsymbol{\gamma}_{(i)}$ is the *i*th column of $\boldsymbol{\Gamma}$, and may be called the *i*th vector of *principal component loadings.*

Note: definition of loadings.

Some authors define the loadings somewhat differently, as the covariances of the principal components with the original variables. S-PLUS follows Mardia, Kent, and Bibby (1979).

The first principal component has the largest variance among all SLCs of $\boldsymbol{x}$. Similarly, the second principal component has the largest variance among all SLCs of $\boldsymbol{x}$ uncorrelated with the first principal component, and so on.

In general, there are as many principal components as variables. However, because of the way they are calculated, it is usually possible to consider only a few of the principal components, which together explain *most* of the original variation.

16.1 CALCULATING PRINCIPAL COMPONENTS

To calculate principal components, use the `princomp` function. In general, the first argument to `princomp` is a numeric matrix or a data frame consisting solely of numeric variables. For example, table 16.1 shows the results of qualifying examinations for 25 graduate students in mathematics at a fictional university. The students sat for examinations in each of five subject areas—differential geometry, complex analysis, algebra, real analysis, and statistics. The differential geometry and complex analysis examinations were closed book, while the remaining three exams were open book.

You can use `matrix` together with `scan` to create an S-PLUS matrix from the data in table 16.1:

```
> testscores <- matrix(scan(), ncol=5, byrow=T)
1: 36 58 43 36 37
2:  . . .
76:
> dimnames(testscores) <- list(1:25, c("diffgeom",
+      "complex", "algebra", "reals", "statistics"))
> testscores
   diffgeom complex algebra reals statistics
 1       36      58      43    36         37
 2    . . .
```

You can then use `princomp` to perform a principal components analysis as follows:

```
> testscores.prc <- princomp(testscores)
```

Table 16.1: *Examination scores for graduate students in mathematics.*

	diffgeom	complex	algebra	reals	statistics
1	36	58	43	36	37
2	62	54	50	46	52
3	31	42	41	40	29
4	76	78	69	66	81
5	46	56	52	56	40
6	12	42	38	38	28
7	39	46	51	54	41
8	30	51	54	52	32
9	22	32	43	28	22
10	9	40	47	30	24
11	32	49	54	37	52
12	40	62	51	40	49
13	64	75	70	66	63
14	36	38	58	62	62
15	24	46	44	55	49
16	50	50	54	52	51
17	42	42	52	38	50
18	2	35	32	22	16
19	56	53	42	40	32
20	59	72	70	66	62
21	28	50	50	42	63
22	19	46	49	40	30
23	36	56	56	54	52
24	54	57	59	62	58
25	14	35	38	29	20

```
> testscores.prc
Standard deviations:
  Comp. 1  Comp. 2  Comp. 3  Comp. 4  Comp. 5
 28.48968 9.035471 6.600955 6.133582 3.723358

The number of variables is 5
        and the number of observations is 25

Component names:
 "sdev" "loadings" "correlations" "scores" "center"
 "scale" "n.obs" "call" "factor.sdev" "coef"

Call:
princomp(x = testscores)
```

The `princomp` function returns an object of mode `"princomp"`, and the printing method for objects of this class shows the standard deviations of the resulting principal components, together with information on the size of the original data set, the names of the components making up the object, and the original call.

By default, `princomp` uses a weighted covariance estimation function, `cov.wt`, to perform the principal components analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function, which is described in section 16.4, Estimating the Model Using a Covariance or Correlation Matrix.

Use `summary` to produce a summary showing the importance of the calculated principal components:

```
> summary(testscores.prc)
Importance of components:
                          Comp. 1    Comp. 2
    Standard deviation 28.4896795 9.03547104
Proportion of Variance  0.8212222 0.08260135
 Cumulative Proportion  0.8212222 0.90382353
                          Comp. 3    Comp. 4
    Standard deviation 6.60095491 6.13358179
Proportion of Variance 0.04408584 0.03806395
 Cumulative Proportion 0.94790936 0.98597332
                          Comp. 5
    Standard deviation 3.72335754
Proportion of Variance 0.01402668
 Cumulative Proportion 1.00000000
```

In our example, the first principal component explains 82% of the variance, and the first two principal components together explain 90% of the variance.

16.2 PRINCIPAL COMPONENT LOADINGS

The *principal component loadings* are the coefficients of the principal components transformation. They provide a convenient summary of the influence of the original variables on the principal components, and thus a useful basis for interpretation. A large coefficient (in absolute value) corresponds to a *high* loading, while a coefficient near zero has a *low* loading.

You can view the loadings for a principal components object in either of two ways. First, you can print them as part of the object summary by using the `loadings=T` argument to `summary`:

```
> summary(testscores.prc, loadings=T)
Importance of components:
                          Comp. 1    Comp. 2
    Standard deviation 28.4896795 9.03547104
Proportion of Variance  0.8212222 0.08260135
 Cumulative Proportion  0.8212222 0.90382353
                          Comp. 3    Comp. 4
    Standard deviation 6.60095491 6.13358179
Proportion of Variance 0.04408584 0.03806395
 Cumulative Proportion 0.94790936 0.98597332
                          Comp. 5
    Standard deviation 3.72335754
Proportion of Variance 0.01402668
 Cumulative Proportion 1.00000000

Loadings:
           Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
  diffgeom  0.598  -0.675  -0.185  -0.386
   complex  0.361  -0.245   0.249   0.829  -0.247
   algebra  0.302   0.214   0.211   0.135   0.894
     reals  0.389   0.338   0.700  -0.375  -0.321
statistics  0.519   0.570  -0.607          -0.179
```

To see the loadings alone, use the `loadings` function:

```
> loadings(testscores.prc)
           Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
  diffgeom  0.598  -0.675  -0.185  -0.386
   complex  0.361  -0.245   0.249   0.829  -0.247
   algebra  0.302   0.214   0.211   0.135   0.894
```

```
     reals  0.389   0.338   0.700  -0.375  -0.321
statistics  0.519   0.570  -0.607          -0.179
```

The `loadings` function returns an object of class `"loadings"`. This class has methods for printing and plotting; a plot of the loadings lets you see at a glance which variables are best explained by each component. For example, consider the loadings plot created by the following call to `plot` (and shown in figure 16.1:

```
> plot(loadings(testscores.prc))
```

The loadings for the first principal component are all of the same sign, and of moderate size. A reasonable interpretation is that this component represents an "average" score for the five qualifying examinations. The second component contrasts the two closed book exams with the three open book exams, with the first and last exams weighted most heavily.

16.3 PRINCIPAL COMPONENTS ANALYSIS USING CORRELATION

The principal components decomposition is not scale-invariant, so that you will obtain different decompositions depending on whether you calculate them for the (unscaled) covariance matrix or the (scaled) correlation matrix. In general, you use the covariance matrix when the original observations are on a common scale (as, for example, our test scores example), but use the correlation matrix when you have observations of different types (such as those of the variables in `state.x77`). Use the `cor=T` argument to `princomp` to calculate principal components for scaled data:

```
> state.prc <- princomp(state.x77, cor=T)
> state.prc
Standard deviations:
  Comp. 1  Comp. 2  Comp. 3   Comp. 4   Comp. 5   Comp. 6
 1.897076 1.277466 1.054486 0.8411327 0.6201949 0.5544923
   Comp. 7   Comp. 8
 0.3800642 0.3364338

The number of variables is 8
        and the number of observations is 50

Component names:

 "sdev" "loadings" "correlations" "scores" "center"
 "scale" "n.obs" "call"

Call:
princomp(x = state.x77, cor = T)
```

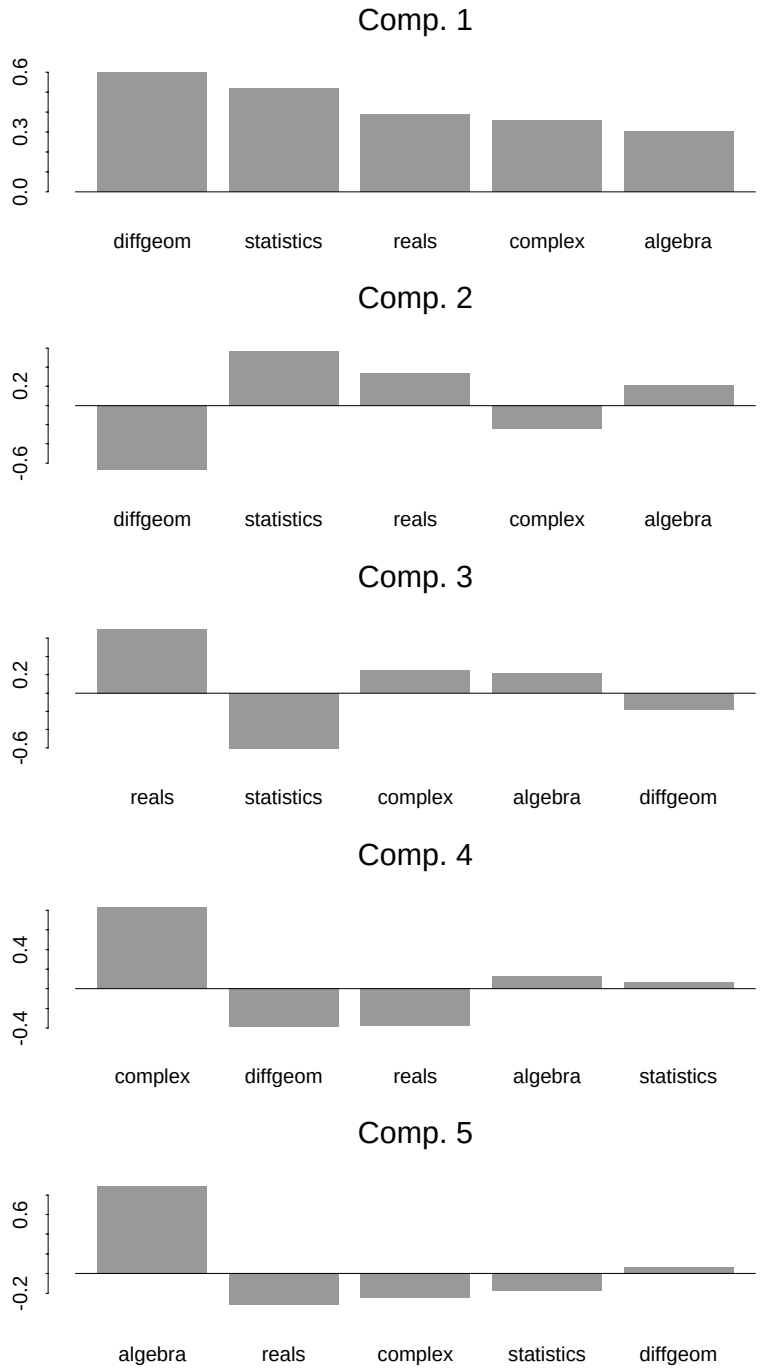

Figure 16.1: *Loadings plot for the test scores data.*

```
> summary(state.prc, loadings=T)
Importance of components:
                          Comp. 1   Comp. 2   Comp. 3
    Standard deviation 1.8970755 1.2774659 1.0544862
Proportion of Variance 0.4498619 0.2039899 0.1389926
 Cumulative Proportion 0.4498619 0.6538519 0.7928445
                          Comp. 4    Comp. 5
    Standard deviation 0.84113269 0.62019488
Proportion of Variance 0.08843803 0.04808021
 Cumulative Proportion 0.88128252 0.92936273
                          Comp. 6   Comp. 7
    Standard deviation 0.55449226 0.3800642
Proportion of Variance 0.03843271 0.0180561
 Cumulative Proportion 0.96779544 0.9858515
                          Comp. 8
    Standard deviation 0.33643379
Proportion of Variance 0.01414846
 Cumulative Proportion 1.00000000

Loadings:
           Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
Population -0.126   0.411   0.656   0.409   0.406
    Income  0.299   0.519   0.100          -0.638
Illiteracy -0.468                  -0.353
  Life Exp  0.412           0.360  -0.443   0.327
    Murder -0.444   0.307  -0.108   0.166  -0.128
   HS Grad  0.425   0.299          -0.232
     Frost  0.357  -0.154  -0.387   0.619   0.217
      Area          0.588  -0.510  -0.201   0.499
           Comp. 6 Comp. 7 Comp. 8
Population                  0.219
    Income -0.462
Illiteracy -0.387  -0.620   0.339
  Life Exp -0.219  -0.256  -0.527
    Murder  0.325  -0.295  -0.678
   HS Grad  0.645  -0.393   0.307
     Frost -0.213  -0.472
      Area -0.148   0.286
```

From the loadings for this decomposition, we see that the first principal component contrasts “good” variables such as income and life expectancy with “bad” variables such as murder and illiteracy. It is tempting to interpret

this component as a real measure of some nebulous quantity labeled, for example, "Quality of Life." From the importance-of-components summary, however, we see that this component explains only about 45% of the total variance. If we give this "obvious" interpretation to the first principal component, what natural interpretation can we give to the second principal component, which seems to contrast the proportion of frosty days with virtually all of the other variables, and explains another 20% of the variance? This example shows that, while calculating principal components is straightforward, interpreting the resulting components in physical or social terms is not always so.

16.4 ESTIMATING THE MODEL USING A COVARIANCE OR CORRELATION MATRIX

If you do not have raw data, but either a covariance or correlation matrix derived from the original data, you can use the `covlist` argument of the `princomp` function to perform a principal components analysis. The data object that is passed to `princomp` must be a list object with two components, `cov` and `center`.

For example, suppose you have a data object `covmatrix` containing the following covariance matrix:

```
           diffgeom  complex  algebra    reals statistics
  diffgeom 334.8224  174.424 132.0432 169.8096    224.312
   complex 174.4240  139.920  87.6320 104.1360    136.800
   algebra 132.0432   87.632  91.5776 101.8928    129.776
     reals 169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760
```

Convert `covmatrix` into a list object containing the `cov` and `center` components as follows:

```
> cov.obj <- list(cov = covmatrix, center = c(0,0,0,0,0))
> cov.obj
$cov:
           diffgeom  complex  algebra    reals statistics
  diffgeom 334.8224  174.424 132.0432 169.8096    224.312
   complex 174.4240  139.920  87.6320 104.1360    136.800
   algebra 132.0432   87.632  91.5776 101.8928    129.776
     reals 169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760

$center:
[1] 0 0 0 0 0
```

To perform the principal components analysis, pass the `cov.obj` object to the `princomp` function by using the `covlist` argument, as follows:

```
> princov <- princomp(covlist = cov.obj)
> princov
Standard deviations:
  Comp. 1  Comp. 2  Comp. 3  Comp. 4  Comp. 5
 28.48968 9.035471 6.600955 6.133582 3.723358

The number of variables is 5 and the number of
    observations is unknown.

Component names:

 "sdev" "loadings" "correlations" "center" "scale" "call"

Call:
princomp(covlist = cov.obj)
```

If you have a correlation matrix, you can use the `covlist` argument in the same way. For example, suppose you have a data object `cormatrix` containing the following correlation matrix:

```
            diffgeom   complex   algebra     reals statistics
   diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
    complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
    algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
      reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
 statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999
```

Convert `cormatrix` into a list object containing the `cov` and `center` components as follows:

```
> cor.obj <- list(cov = cormatrix, center = c(0,0,0,0,0))
> cor.obj
$cov:
            diffgeom   complex   algebra     reals statistics
   diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
    complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
    algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
      reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
 statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999

$center:
[1] 0 0 0 0 0
```

To perform the principal components analysis, pass the `cor.obj` object to the `princomp` function by using the `covlist` argument, as follows:

```
> princor <- princomp(covlist = cor.obj)
> princor
Standard deviations:
  Comp. 1   Comp. 2   Comp. 3   Comp. 4   Comp. 5
 2.020188 0.6114408 0.4653519 0.4525298 0.3516317

The number of variables is 5 and the number of observations
    is unknown.

Component names:
 "sdev" "loadings" "correlations" "center" "scale" "call"

Call:
princomp(covlist = cor.obj)
```

By default, `princomp` uses a weighted covariance estimation function, `cov.wt`, to perform the principal components analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function by performing the following steps:

1. Use the `cov.mve` function with the raw data, in this example, the `rawdataobj` object, as follows:

```
> mve.object <- cov.mve(rawdataobj)
```

   The returned object is a list containing the `cov` and `center` components.

2. Pass the raw data and `mve.object` to `princomp` by using the `covlist` argument as follows:

```
> prin.obj <- princomp(rawdataobj, covlist=mve.object)
```

16.5 EXCLUDING PRINCIPAL COMPONENTS

The purpose of principal components analysis is to reduce the complexity of multivariate data by transforming the data into the principal components space, and then choosing the first *n* principal components that explain "most" of the variation in the original variables. Many criteria have been suggested for deciding how many principal components to retain, including the following:

- (Cattell) Plot the eigenvalues λ_j against j. The resulting plot, called a *screeplot* because it resembles a mountainside with a jumble of

boulders at its base, often provides a convenient visual method of separating the important components from the less-important components.

- Include just enough components to explain some arbitrary amount (typically, 90%) of the variance.
- (Kaiser) Exclude those principal components with eigenvalues below the average. For principal components calculated from a correlation matrix, this criterion excludes components with eigenvalues less than 1.

Mardia *et al.* point out that using Cattell's criterion typically results in too many included components, while Kaiser's criterion typically includes too few. The 90% criterion is often a useful compromise.

Creating a Screeplot

A screeplot plots the eigenvalues against their indices, and generally breaks visually into a steady downward slope (the mountainside) and a gradual tailing away (the scree). The break from the steady downward slope indicates the break between the "important" principal components and the remaining components which make up the scree. The screeplot is the default plot for objects of class `"princomp"`. Thus, to create a screeplot for a principal components object, simply use the `plot` function:

```
> plot(state.prc)
[1] 0.700000 1.900000 3.100000 4.300000 5.500000
[6] 6.700000 7.900000 9.099999
```

By default, the screeplot takes the form of a barplot, and the call to plot returns the *x*-coordinates of the centers of the bars. The resulting plot is shown in figure 16.2. Looking for an obvious break between mountainside and scree, you would probably conclude that four or six components should be retained. The 90% criterion retains five components.

You can also create a screeplot as a line graph, using the argument `style="lines"`:

```
> plot(testscores.prc, style="lines")
[1] 1 2 3 4 5
```

The screeplot for the test scores is shown in figure 16.3. Only the first and second components appear important here, in agreement with the 90% criterion.

The plot method objects of class `"princomp"` simply calls the `screeplot` function. You can call `screeplot` directly to create the plots in figures 16.2 and 16.3. Using `screeplot` is particularly useful when writing functions or S-PLUS scripts; it clearly indicates what type of plot is being created.

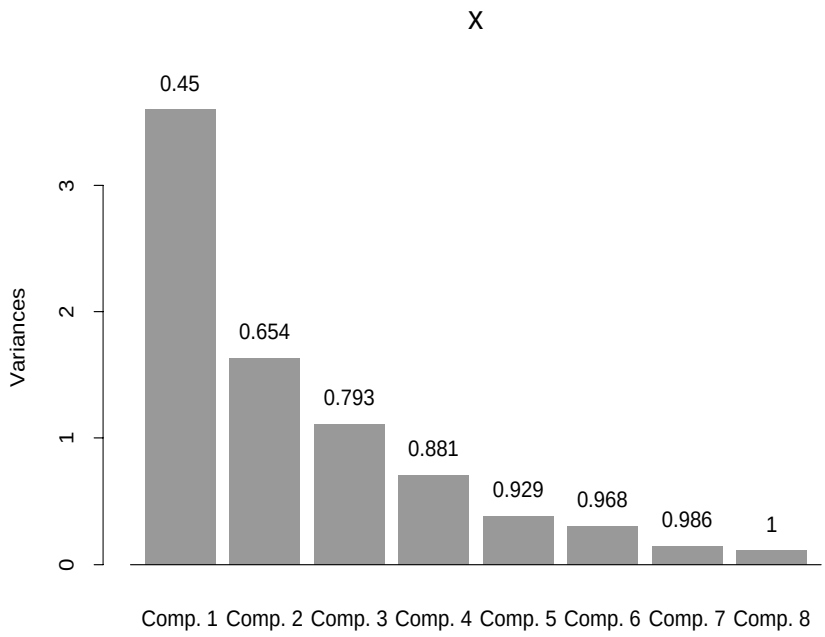

Figure 16.2: *Screeplot for the* `state.x77` *data.*

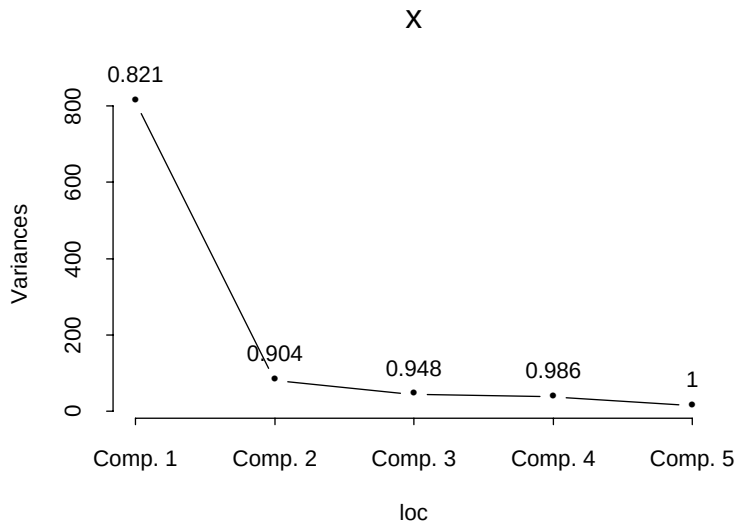

Figure 16.3: *Screeplot for the* `state.x77` *data, using* `style="lines"`.

Evaluating Eigenvalues

To apply Kaiser's criterion for excluding eigenvalues:

1. Square the `sdev` component of the principal components object to obtain the vector of eigenvalues.
2. Take the mean of the vector of eigenvalues.
3. Exclude those components with eigenvalues less than the mean.

For example, for the `testscores` data:

```
> testscores.eigen <- testscores.prc$sdev^2
> testscores.eigen
 Comp. 1 Comp. 2 Comp. 3 Comp. 4 Comp. 5
 811.662 81.6397 43.5726 37.6208 13.8634
> mean(testscores.eigen)
[1] 197.672
```

Using Kaiser's criterion, we exclude all components except the first. The 90% criterion suggests keeping the first two.

For principal components objects created from correlation matrices, such as our state.prc example, the mean of the eigenvalues is 1, so we can simply look at the eigenvalues to determine which components to exclude:

```
> state.prc$sdev^2
 Comp. 1 Comp. 2 Comp. 3  Comp. 4  Comp. 5  Comp. 6
  3.5989 1.63192 1.11194 0.707504 0.384642 0.307462
  Comp. 7  Comp. 8
0.144449 0.113188
```

Kaiser's criterion suggests including only the first three principal components. The 90% criterion suggests including the first five.

16.6 PREDICTION: PRINCIPAL COMPONENT SCORES

One important use of principal components is interpreting the original data in terms of the principal components. For example, the first principal component of the test scores data seems to reflect a weighted average of the test scores. Evaluating this average for each student provides a simple criterion for ranking the students. The images of the original data under the principal components transformation are referred to as *principal component scores*. By default, `princomp` calculates the scores and stores them in the `scores` component of the returned object:

```
> testscores.prc$scores
      Comp. 1    Comp. 2    Comp. 3    Comp. 4
 1  -7.540322 -10.216765  -2.537471   8.670900
 2  20.361037       . . .
```

You can force `princomp` to omit the scores by giving the argument `scores=F`.

Alternatively, if you view the principal components as estimates of interpretable quantities (for example, interpreting the first principal component of the test scores as an estimate of overall ability), it is perhaps more natural to view the principal component scores as predictions from the principal components model. In this case, it is most natural to obtain the scores using the generic `predict` function:

```
> predict(testscores.prc)
      Comp. 1    Comp. 2    Comp. 3    Comp. 4
 1  -7.540322 -10.216765  -2.537471   8.670900
 2  20.361037       . . .
```

You can use `predict` to obtain estimated scores for new data, as well. The new data must be in the same form as the original data. For example, suppose you obtained test scores for five additional students and stored them in the matrix `newscores`:

```
> newscores
  diffgeom complex algebra reals statistics
1       22      50      70    54         30
2       22      46      38    52         62
3       22      42      50    40         62
4       42      49      70    42         50
5       32      35      44    66         32
```

You can obtain the predicted scores for this new data using `predict` as follows:

```
> predict(testscores.prc, newdata=newscores)
     Comp. 1   Comp. 2    Comp. 3     Comp. 4
1  -7.273022  9.070945  20.624141   3.8263656
2  -2.559011 20.754755  -7.975341  -0.7556388
3  -5.044379 20.243279 -14.834342   2.0521791
4  10.041295  3.158848  -3.878835   1.2183456
5  -8.851869  5.635621  16.724818 -20.3311596
       Comp. 5
1  16.4349148
2 -16.2811592
3  -0.7045226
4  18.1853226
5  -6.7149242
```

16.7 ANALYZING PRINCIPAL COMPONENTS GRAPHICALLY

The Biplot

We have already seen several graphical views of some portions of the principal components analysis, namely the screeplot and the loadings plot. However, neither of these plots gives a comprehensive view of both the principal components and the original data. The *biplot* (Gabriel (1971)) allows you to represent both the original variables and the transformed observations on the principal components axes. By showing the transformed observations, you can easily interpret the original data in terms of the principal components. By showing the original variables, you can view graphically the relationships between those variables and the principal components.

To create a biplot in S-PLUS, use the `biplot` function, giving an object of class `"princomp"` as its first argument. For example, to create a biplot for the test scores data, use `biplot` as follows:

```
> biplot(testscores.prc)
```

The resulting plot is shown in figure 16.4.

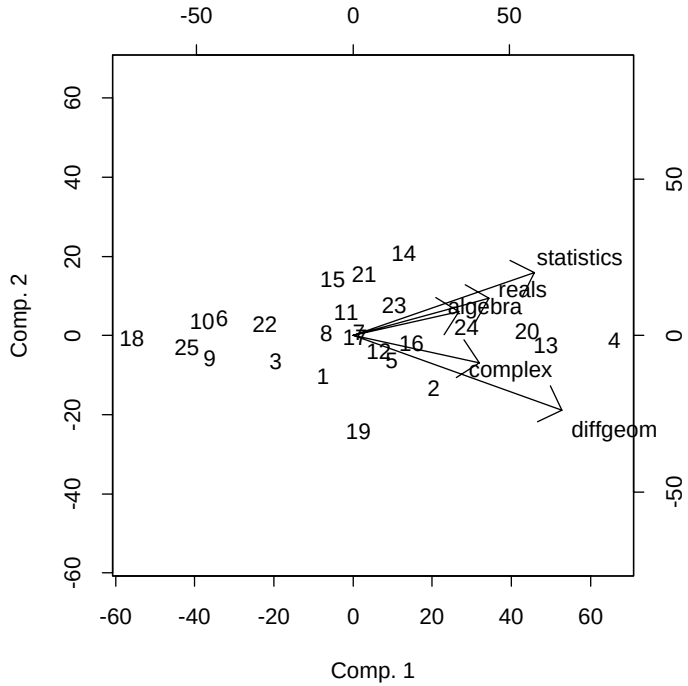

Figure 16.4: *Biplot of test scores data.*

Interpreting the biplot is straightforward: the x-axis represents the scores for the first principal component, the y-axis the scores for the second principal component. The original variables are represented by arrows which graphically indicate the proportion of the original variance explained by the first two principal components. The direction of the arrows indicates the relative loadings on the first and second principal components. For example, the variable `diffgeom` has the largest loadings in absolute value for both the first and second components, and the loading on the second component has negative sign. Thus `diffgeom` is represented by a longish, downward sloping arrow. The variable `algebra` has the smallest loadings on the first two components, and both loadings have the same sign. Thus `algebra` is represented by a short, slightly upward-pointing arrow.

16.8 REFERENCES

Mardia, K. V. and Kent, J. T. and Bibby, J. M. (1979). *Multivariate Analysis*. Academic Press, London.

Gabriel, K. R. (1971). *The biplot graphical display of matrices with applications to principal component analysis*. Biometrika, 58:453–467.

FACTOR ANALYSIS 17

Factor analysis attempts to provide usable numeric values to quantities that are not directly measurable.

FACTOR ANALYSIS

In many scientific fields, notably psychology and other social sciences, you are often interested in quantities, such as intelligence or social status, that are not directly measurable. However, it is often possible to measure other quantities which reflect the underlying variable of interest. *Factor analysis* is an attempt to explain the correlations between observable variables in terms of underlying *factors*, which are themselves not directly observable. For example, measurable quantities such as performance on a series of tests can be explained in terms of an underlying factor such as intelligence.

Note: Different uses of the word "factor"
The use of the word "factor" in factor analysis has nothing to do with the usual S-PLUS sense of a factor as a categorical data object. In this chapter, we reserve the phrase "S-PLUS factor" for this usual sense; the word "factor" alone refers to the traditional meaning in factor analysis, that is, an underlying variable that is not directly observable.

At first glance, factor analysis closely resembles principal components analysis. Both use linear combinations of variables to explain sets of observations of many variables. In principal components analysis, the observed variables are themselves the quantities of interest. The combination of these variables in the principal components is primarily a tool for simplifying the interpretation of the observed variables. In factor analysis, by contrast, the observed variables are of relatively little intrinsic interest—the underlying factors are the quantity of interest.

Formally, if $\boldsymbol{x}$ is a $p \times 1$ random vector with mean $\boldsymbol{\mu}$ and covariance matrix $\boldsymbol{\Sigma}$, then the k-factor model holds for $\boldsymbol{x}$ if $\boldsymbol{x}$ can be written in the form

$$\boldsymbol{x} = \boldsymbol{\mu} + \boldsymbol{\Lambda f} + \boldsymbol{u} \tag{17.1}$$

where $\boldsymbol{\Lambda} = \{\lambda_{ij}\}$ is a $p \times k$ matrix of constants called the *matrix of factor loadings* and $\boldsymbol{f}$ and $\boldsymbol{u}$ are random vectors representing, respectively, the k underlying *common* factors and p *unique* factors associated with the original observed variables. Equivalently, the covariance matrix $\boldsymbol{\Sigma}$ can be decomposed into a *factor covariance matrix* and an *error covariance matrix*:

$$\boldsymbol{\Sigma} = \boldsymbol{\Lambda\Lambda}' + \boldsymbol{\Psi} \tag{17.2}$$

where $\boldsymbol{\Psi} = \mathrm{VAR}(\boldsymbol{u})$. The diagonal of the factor covariance matrix is called

the vector of *communalities* h_i^2, where

$$h_i^2 = \sum_{j=1}^{k} \lambda_{ij}^2 .$$

The communalities represent the common variation in the factors, while the ψ_{ii}, called the *uniquenesses*, represent the variation in the x_i not shared with the other variables.

The *k*-factor model makes sense only if the degrees of freedom $s \geq 0$, where *s* is given by the equation

$$s = \frac{1}{2}(p-k)^2 - \frac{1}{2}(p+k) .$$

For example, if $p = 5$, $s > 0$ for $k = 1$ and $k = 2$, but $s < 0$ for $k = 3$, $k = 4$, and $k = 5$. Thus, if a factor model is appropriate for a set of five variables, it will have no more than two factors.

17.1 ESTIMATING THE MODEL

To perform factor analysis in S-PLUS, use the `factanal` function. There are two main techniques for estimating the factors in factor analysis: the *principal factor estimate* and the *maximum likelihood estimate*. For a description of these techniques, see Harman (1976) or Mardia, Kent, and Bibby (1979). The principal factor estimate (`method="principal"`) is the default.

For example, consider again the test scores data of table 16.1. We suppose a two-factor model, one factor representing the overall ability of each student and the second factor representing the relative effects of open vs.closed book exams. We perform the factor analysis as follows, giving `factanal` the raw data `testscores` and specifying the number of factors with the `factors` argument:

```
> testscores.fa <- factanal(testscores, factors=2)
```

The `factanal` function returns an object of class `"factanal"`. As always, you can look at the object by typing its name. The print method for objects of class `"factanal"` shows the sum of squares of the factor loadings, the size of the data, the names of the components in the returned object, and the call that created the object:

```
> testscores.fa
Sums of squares of loadings:
  Factor1  Factor2
 2.219645 1.866672
```

```
The number of variables is 5
   and the number of observations is 25

Component names:

 "loadings" "uniquenesses" "correlation" "criteria"
 "factors" "dof" "method" "center" "scale" "n.obs"
 "scores" "call"

Call:
factanal(x = testscores, factors = 2)
```

By default, `factanal` uses a weighted covariance estimation function, `cov.wt`, to perform the factor analysis. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function, which is described in section 17.3, Estimating the Model Using a Covariance or Correlation Matrix.

To see a numeric summary of the factor solution, use the `summary` function:

```
> summary(testscores.fa)
Importance of factors:
                Factor1   Factor2
   SS loadings 2.219645 1.8666722
Proportion Var 0.443929 0.3733344
Cumulative Var 0.443929 0.8172634

The degrees of freedom for the model is 1.

Uniquenesses:
  diffgeom   complex   algebra     reals statistics
 0.1970121 0.1879035 0.1201226 0.1984058  0.2102388

Loadings:
           Factor1 Factor2
  diffgeom 0.506   0.739
   complex 0.457   0.777
   algebra 0.787   0.510
     reals 0.775   0.448
statistics 0.730   0.507
attr(, "names"):
 [1] Factor1 Factor1 Factor1 Factor1 Factor1 Factor2
 [7] Factor2 Factor2 Factor2 Factor2
```

The table at the top of the summary, labeled "Importance of Factors," shows the sum of squares of the loadings on each factor, along with the proportion of the total variance explained by each factor, and the cumulative proportion explained after each factor is included. Thus, the two-factor model for the test scores data explains about 80% of the variation in the original data, with the first factor accounting for about 45%.

The summary also shows the number of degrees of freedom in the model, the uniquenesses, and the factor loadings. The factor loadings can also be seen by themselves, using the `loadings` function:

```
> loadings(testscores.fa)

           Factor1 Factor2
  diffgeom 0.506   0.739
   complex 0.457   0.777
   algebra 0.787   0.510
     reals 0.775   0.448
statistics 0.730   0.507
attr(, "names"):
 [1] Factor1 Factor1 Factor1 Factor1 Factor1 Factor2
 [7] Factor2 Factor2 Factor2 Factor2
```

Since the uniquenesses and communalities sum to 1 for each variable, you can calculate the communalities h_i^2 from the uniquenesses as follows:

```
> 1 - testscores.fa$uniquenesses
  diffgeom   complex   algebra     reals statistics
 0.8029879 0.8120965 0.8798774 0.8015942  0.7897612
```

17.2 ESTIMATING THE MODEL USING MAXIMUM LIKELIHOOD

To use the maximum likelihood factor estimate, specify `method="mle"` in the call to `factanal`:

```
> testscores.fa2 <- factanal(testscores, factors=2,
+  method="mle")
> testscores.fa2
Sums of squares of loadings:
 Factor1  Factor2
 2.48222 1.726735

The number of variables is 5
  and the number of observations is 25
```

```
Test of the hypothesis that 2 factors are sufficient
versus the alternative that more are required:
The chi square statistic is 0.78 on 1 degree of freedom.
The p-value is 0.378

Component names:

 "loadings" "uniquenesses" "correlation" "criteria"
 "factors" "dof" "method" "center" "scale" "n.obs"
 "scores" "call"

Call:
factanal(x = testscores, factors = 2, method = "mle")
```

With the maximum likelihood method, it is possible to perform a test of the hypothesis that the specified number of factors is adequate to explain the model, and the print method for objects of class `"factanal"` gives the results of this test. In this case, there is no evidence that more factors should be added.

17.3 ESTIMATING THE MODEL USING A COVARIANCE OR CORRELATION MATRIX

If you do not have raw data, but either a covariance or correlation matrix derived from the original data, you can use the `covlist` argument of the `factanal` function to estimate the factors. The data object that is passed to `factanal` must be a list object with two components, `cov` and `center`.

For example, suppose you have a data object `covmatrix` containing the following covariance matrix:

```
           diffgeom  complex  algebra    reals statistics
diffgeom 334.8224  174.424 132.0432 169.8096    224.312
 complex 174.4240  139.920  87.6320 104.1360    136.800
 algebra 132.0432   87.632  91.5776 101.8928    129.776
   reals 169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760
```

Convert `covmatrix` into a list object containing the `cov` and `center` components as follows:

```
> cov.obj <- list(cov = covmatrix, center = c(0,0,0,0,0))
> cov.obj
```

```
$cov:
            diffgeom  complex  algebra    reals statistics
  diffgeom 334.8224  174.424 132.0432 169.8096    224.312
   complex 174.4240  139.920  87.6320 104.1360    136.800
   algebra 132.0432   87.632  91.5776 101.8928    129.776
     reals 169.8096  104.136 101.8928 160.2784    160.848
statistics 224.3120  136.800 129.7760 160.8480    261.760

$center:
[1] 0 0 0 0 0
```

To perform the factor analysis, pass the `cov.obj` object to the `factanal` function by using the `covlist` argument, as follows:

```
> factcov <- factanal(covlist = cov.obj)
> factcov
Sums of squares of loadings:
  Factor1
 3.854577

The number of variables is 5
   and the number of observations is unknown.

Component names:

 "loadings" "uniquenesses" "correlation"
   "criteria" "factors" "dof"

 "method" "center" "scale" "call"

Call:
factanal(covlist = cov.obj)
```

If you have a correlation matrix, you can use the `covlist` argument in the same way. For example, suppose you have a data object `cormatrix` containing the following correlation matrix:

```
            diffgeom   complex   algebra     reals statistics
  diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
   complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
   algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
     reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999
```

Convert `cormatrix` into a list object containing the `cov` and `center` components as follows:

```
> cor.obj <- list(cov = cormatrix, center = c(0,0,0,0,0))
> cor.obj
$cov:
             diffgeom   complex   algebra     reals statistics
   diffgeom 1.0000000 0.8058590 0.7540744 0.7330229  0.7576935
    complex 0.8058590 0.9999999 0.7741556 0.6953821  0.7148164
    algebra 0.7540744 0.7741556 1.0000000 0.8410298  0.8382009
      reals 0.7330229 0.6953821 0.8410298 1.0000000  0.7852836
 statistics 0.7576935 0.7148164 0.8382009 0.7852836  0.9999999

$center:
[1] 0 0 0 0 0
```

To perform the factor analysis, pass the `cor.obj` object to the `factanal` function by using the `covlist` argument, as follows:

```
> factcor <- factanal(covlist = cor.obj)
> factcor
Sums of squares of loadings:
  Factor1
 3.854577

The number of variables is 5
   and the number of observations is unknown.

Component names:

 "loadings" "uniquenesses" "correlation"
   "criteria" "factors" "dof"

 "method" "center" "scale" "call"

Call:
factanal(covlist = cor.obj)
```

By default, `factanal` uses a weighted covariance estimation function, `cov.wt`, to estimate the factors. If you want to use a minimum volume ellipsoid covariance estimate, use the `cov.mve` function by performing the following steps:

1. Use the `cov.mve` function with the raw data, in this example, the `rawdataobj` object, as follows:

   ```
   > mve.object <- cov.mve(rawdataobj)
   ```

   The returned object is a list containing the `cov` and `center` components.

2. Pass the raw data and `mve.object` to `factanal` by using the `covlist` argument as follows:

```
> fact.obj <- factanal(rawdataobj, covlist=mve.object)
```

17.4 ROTATING FACTORS

The solution to the equation (17.2) is not unique (unless the number of factors k is 1); if $\boldsymbol{G}$ is a $k \times k$ orthogonal matrix, then

$$\boldsymbol{\Sigma} = (\boldsymbol{\Lambda G}')(\boldsymbol{G}'\boldsymbol{\Lambda}') + \boldsymbol{\Psi} \tag{17.3}$$

which has the form of equation (17.2) with $\boldsymbol{\Delta} = \boldsymbol{\Lambda G}$ being the matrix of *rotated factor loadings*. Thus, the factor loadings are inherently indeterminate. Any solution can be rotated arbitrarily to arrive at a new solution. In practice, this indeterminancy is used to arrive at a factor solution that has what Thurstone (1935) named *simple structure*. Loosely, the factor solution has simple structure if each variable is loaded highly on one factor, and all factor loadings are either large (in absolute value) or near zero.

Factor analysts have developed many different criteria for choosing the appropriate rotation. By default, S-PLUS uses the "varimax" method. You can specify a different rotation with the `rotation` argument to `factanal`. For example, to compute the factor solution to the test scores data using the `"oblimin"` rotation, call `factanal` as follows:

```
> testscores.fao <- factanal(testscores, factors=2,
+       rotation="oblimin")
> summary(testscores.fao)
Importance of factors:
                 Factor1   Factor2
   SS loadings 404.22436 400.63064
Proportion Var  80.84487  80.12613
Cumulative Var  80.84487 160.97100

The degrees of freedom for the model is 1.

Uniquenesses:
  diffgeom   complex   algebra     reals statistics
 0.1970121 0.1879035 0.1201226 0.1984058  0.2102388
```

```
Loadings:
           Factor1 Factor2
  diffgeom 8.875   9.040
   complex 8.759   8.985
   algebra 9.425   9.229
     reals 8.911   8.680
statistics 8.972   8.814
attr(, "names"):
 [1] Factor1 Factor1 Factor1 Factor1 Factor1 Factor2
 [7] Factor2 Factor2 Factor2 Factor2
```

You can rotate any object of class `"factanal"` using the `rotate` function:

```
> rotate(testscores.fa, rotation="biquartimin")
Sums of squares of loadings:
  Factor1  Factor2
 3.943076 2.836276

The number of variables is 5
   and the number of observations is 25

Component names:

 "loadings" "uniquenesses" "correlation" "criteria"

 "factors" "dof" "method" "center" "scale" "n.obs"

 "scores" "call"

Call:
rotate.factanal(x = factanal(x = testscores, factors
        = 2), rotation = "biquartimin")
> loadings(.Last.value)
           Factor1 Factor2
  diffgeom -0.153   1.042
   complex -0.372   1.252
   algebra  1.137  -0.210
     reals  1.235  -0.359
statistics  0.981
attr(, "names"):
 [1] Factor1 Factor1 Factor1 Factor1 Factor1 Factor2
 [7] Factor2 Factor2 Factor2 Factor2
```

S-PLUS recognizes the following character strings as valid `rotation` arguments:

```
"varimax"       "quartimax"  "equamax"
"parsimax"      "orthomax"   "covarimin"
"biquartimin"   "quartimin"  "oblimin"
"procrustes"    "promax"     "none"
"crawford.ferguson"
```

See Harman (1976) for descriptions of the various rotations. See the `rotate` help file for additional information on using the various rotations in S-PLUS.

17.5 VISUALIZING THE FACTOR SOLUTION

The loadings matrix provides a precise, numeric answer to the question of which variables are loaded most strongly on each factor. However, you can get a much more intuitive feel for the answer if you look at the loadings visually. You obtain a loadings plot by calling `plot` on the factor loadings:

```
> plot(loadings(testscores.fa))
```

The resulting plot is shown in figure 17.1.

To see the relation of the factors to both the original variables and the original data, use `biplot`:

```
> biplot(testscores.fa)
```

The resulting plot is shown in figure 17.2.

17.6 PREDICTION: FACTOR ANALYSIS SCORES

An important use of factor analysis is to translate the original data into the planes of the factors. You view the factors as estimates of interpretable quantities (for example, interpreting the first factor of the test scores as an estimate of overall ability). The images of the original data under the factor analysis transformation are referred to as *factor analysis scores*. By default, `factanal` calculates the scores and stores them in the `scores` component of the returned object:

```
> testscores.fa$scores
       Factor1   Factor2
 1  -1.1778029 0.7612478
 2  -0.2755734     . . .
```

You can force `factanal` to omit the scores by giving the argument `scores=F`.

It is perhaps more natural to view the factor scores as predictions from the factor analysis model. In this case, it is most natural to obtain the scores using the generic `predict` function:

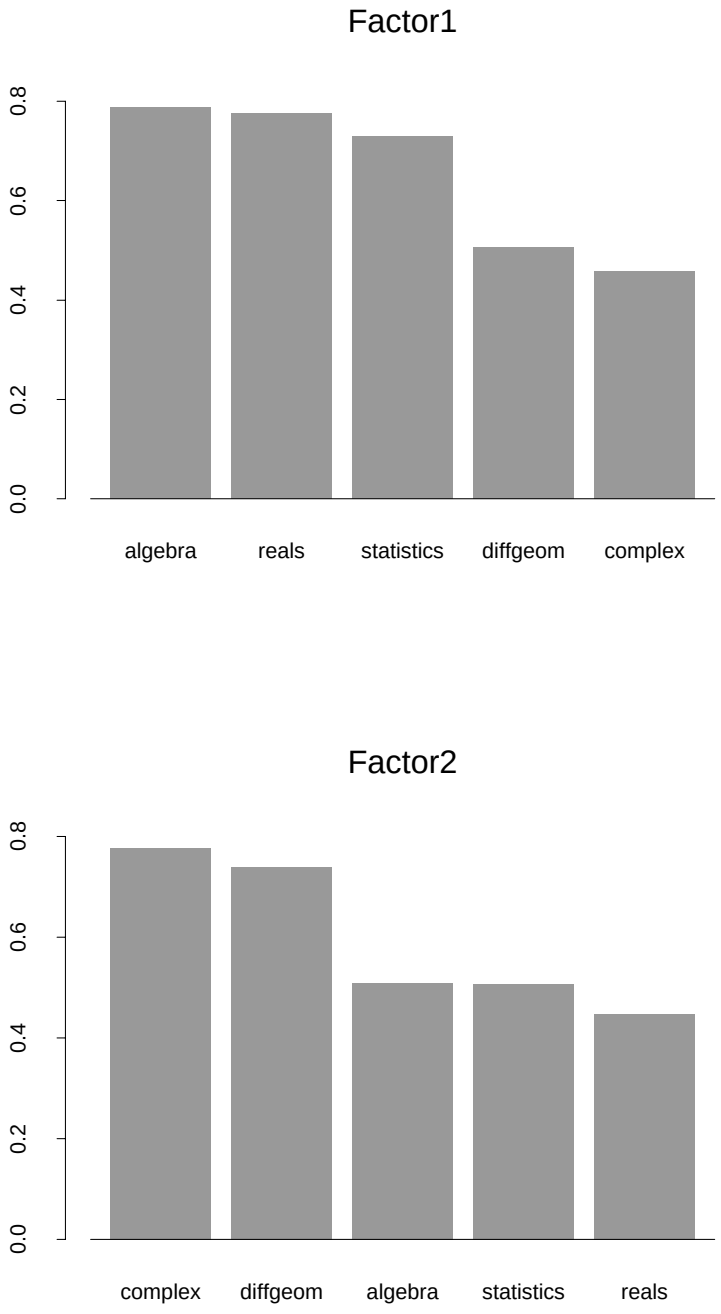

Figure 17.1: *Loadings for the test scores principal factor solution.*

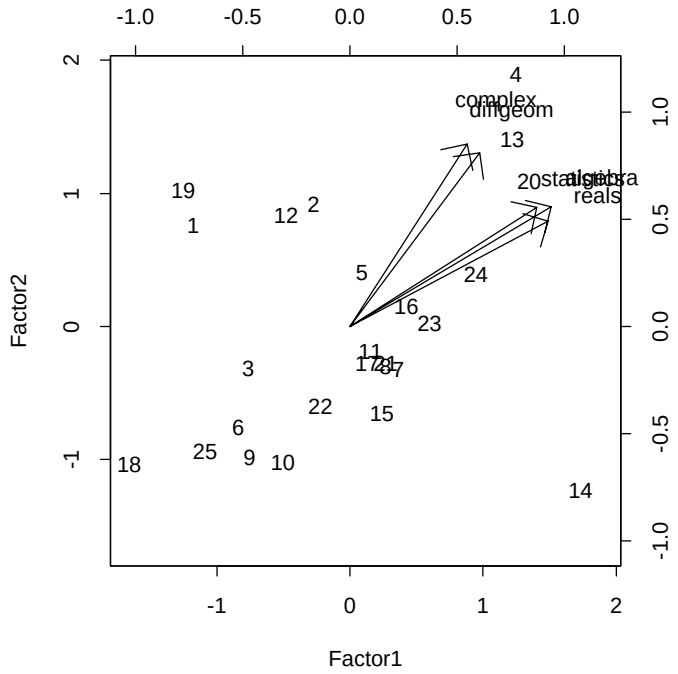

Figure 17.2: *Biplot for the test scores principal factor solution.*

```
> predict(testscores.fa)
      Factor1    Factor2
 1  -1.1778029  0.7612478
 2  -0.2755734      . . .
```

You can use `predict` to obtain estimated scores for new data, as well. The new data must be in the same form as the original data. For example, suppose you obtained test scores for five additional students and stored them in the matrix `newscores`:

```
> newscores
  diffgeom complex algebra reals statistics
1       22      50      70    54         30
2       22      46      38    52         62
3       22      42      50    40         62
4       42      49      70    42         50
5       32      35      44    66         32
```

You can obtain the predicted scores for this new data using `predict` as follows:

```
> predict(testscores.fa, newdata=newscores)
          Factor1    Factor2
[1,]  1.454873272 -0.9626068
[2,] -0.001166622 -0.5764937
[3,]  0.493414880 -0.8808624
[4,]  1.216808651 -0.3201456
[5,]  0.570954434 -1.1814138
attr(, "type"):
[1] "regression"
```

17.7 REFERENCES

Harman, H. H. (1976). *Modern Factor Analysis*. University of Chicago Press, Chicago.

Mardia, K. V. and Kent, J. T. and Bibby, J. M. (1979). *Multivariate Analysis*. Academic Press, London.

Thurstone, L. L. (1935). *The Vectors of Mind*. University of Chicago Press, Chicago.

CLUSTER ANALYSIS 18

The aim of cluster analysis is to classify data into groups that are cohesive but separate.

CLUSTER ANALYSIS

This chapter describes the S-PLUS cluster analysis functions. The chapter is in two parts. The first, sections 18.1–18.4, describes clustering functions built into S-PLUS, and the second, sections 18.5–18.7, describes a library of different functions that can be attached and used accordingly.

The aim of cluster analysis is to classify objects into groups that are cohesive but separate. Examples include the taxonomy of plants and animals, the grouping of pixels in digitized satellite images into types of terrain, the grouping of documents for information retrieval, and the classification of archeological artifacts. Good readable introductions to cluster analysis are given by Gordon (1981) and Murtagh (1985). Data that are suitable for cluster analysis are generally multivariate measurements on objects, and matrices of distances or dissimilarities between objects.

18.1 CLUSTERING FUNCTIONS BUILT INTO S-PLUS

Many clustering algorithms are based on *hierarchical agglomeration*, which starts with each object forming a separate group and in which objects or groups close to one another are successively merged. Other algorithms use *iterative relocation*, which starts with an initial classification and attempts to improve it iteratively by moving objects from one group to another. It is often advantageous to combine these two approaches, first using a hierarchical agglomerative algorithm and then refining the result using iterative relocation.

Many clustering algorithms have been based on heuristic or "reasonable seeming" measures of closeness between groups. Recently, however, it has been realized that basing cluster analysis on an explicit probability model provides a theoretical justification for some of the older heuristic criteria and suggests when they are likely to work best. It also leads to extensions of these criteria that work better in particular circumstances; see Banfield and Raftery (1992). Model-based clustering also leads to a way of choosing the number of clusters that is based on standard statistical ideas, and allows one to take account of "noise", or outliers, thus yielding robust clustering methods.

S-PLUS features clustering via hierarchical agglomeration with both model-based and heuristic criteria; for the model-based methods this includes the criterion for choosing the number of clusters, and a robust clustering option. Iterative relocation for a given initial classification is available relative to any one of the model-based criteria. S-PLUS also provides ways of plotting and manipulating the classification tree, or dendogram.

In section 18.2, the main ideas of clustering are laid out, together with the main kinds of algorithm used and the classification tree. In section 18.3, model-based clustering is outlined, while in section 18.4 the implementation of these methods in S-PLUS is described.

The S-PLUS cluster analysis functions are listed in table 18.1.

Table 18.1: *Cluster analysis functions in S-PLUS.*

Function	Use
`clorder`	Re-Order Leaves of a Classification Tree
`cutree`	Create Groups from Hierarchical Agglomerative Clustering
`dist`	Distance Matrix Calculation
`hclust`	Hierarchical Clustering (three heuristic criteria)
`kmeans`	Iterative Relocation (sum of squares criterion only)
`labclust`	Label the Leaves of a Classification Tree
`mclass`	Classify Objects (uses output of `mclust`)
`mclust`	Model-Based and Heuristic Hierarchical Agglomerative Clustering; Determination of the Number of Clusters; Robust Clustering
`mreloc`	Model-Based Iterative Relocation
`plclust`	Plot a Classification Tree
`subtree`	Extract Part of a Classification Tree

18.2 CLUSTER ANALYSIS

The aim of cluster analysis is to classify a data set into groups that are internally cohesive and externally isolated. Clustering algorithms differ in the measures of cohesion and isolation that they use, in the weighting of these that defines the overall criterion to be optimized by the classification, and in the algorithm used to find the best classification.

If the objects to be classified can be represented by points in Euclidean space, the sum of squares is a popular criterion. For a given classification, this is simply the sum of the within-group sums of squares. To find the global

minimum of this criterion for a specified number of groups could be very expensive computationally; the hierarchical agglomeration and iterative relocation algorithms are ways of finding good, but possibly sub-optimal, classifications.

Hierarchical Agglomeration

The hierarchical agglomeration algorithm starts with each object in a group of its own. At each iteration it merges two groups to form a new group; the merger chosen is the one that leads to the smallest increase in the sum of within-group sums of squares. The number of iterations is equal to the number of objects minus one, and at the end all the objects are together in a single group. This is known variously as Ward's method, the sum of squares method, or the trace method.

One important output from the hierarchical agglomeration algorithm is the classification tree, or dendogram. This represents the entire process graphically, and enables one to see which objects and groups are merged at what stages of the algorithm.

The hierarchical agglomeration algorithm can be used with criteria other than the sum of squares criterion. For example in the single link (or nearest neighbor) method, the distance between two groups is defined to be the smallest distance between any two members from different groups, and at each iteration the two closest groups are merged. The complete link method (also known as the compact or farthest neighbor method) is similar except that the distance between any two groups is defined to be the largest distance between any two members from different groups, while the centroid method defines the distance between two groups to be the distance between their centroids. The average weighted link method uses the average of the distances between the objects in one group and the objects in the other group. These are all heuristic criteria.

Iterative Relocation

In an iterative relocation algorithm, an initial classification is modified by moving objects from one group to another if this will reduce the sum of squares. One such algorithm is the *k*-means algorithm of Hartigan (1975). Iterative relocation, like hierarchical agglomeration, can also be used with criteria other than the sum of squares.

The iterative relocation method is limited in that it requires one to specify the number of clusters in advance, and it also requires an initial classification. A strategy that often yields good results is to use hierarchical agglomeration to determine the number of clusters and to find an initial classification, and then use iterative relocation to improve the classification.

18.3 MODEL-BASED CLUSTERING

Clusters of Different Orientations, Shapes, and Sizes

Model-based clustering is based on the assumption that the data are generated by a mixture of underlying probability distributions. Specifically, it is assumed that the population of interest consists of G different subpopulations, and that the density of an observation x from the kth subpopulation is $f_k(x;\theta)$ for some unknown vector of parameters θ. Given data $D = (x_1, \ldots, x_n)$, we let $\gamma = (\gamma_1, \ldots, \gamma_n)$ denote the identifying labels, where $\gamma_i = k$ if x^i comes from the kth subpopulation. In the classification maximum likelihood procedure, θ and γ are chosen so as to maximize the likelihood.

$$L(D;\theta,\gamma) = \prod_{i=1}^{n} f_{\gamma_i}(x_i;\theta) \tag{18.1}$$

We consider mainly the situation where $f_k(x;\theta)$ is a multivariate normal density with mean μ_k and variance matrix Σ_k. If $\Sigma_k = \sigma^2 I$ for each k, where I is the identity matrix, then maximizing the likelihood (18.1) is the same as minimizing the sum of within-group sums of squares that underlies Ward's method, discussed in section 18.2. Thus Ward's method corresponds to the situation where clusters are hyperspherical with the same variance. If clusters are not of this kind, for example if they are thin and elongated, Ward's method will tend to break them up into hyperspherical blobs.

Other forms of Σ_k yield clustering methods that are appropriate in different situations; see Banfield and Raftery (1992). The key to specifying this is the eigenvalue decomposition of Σ_k. The eigenvectors of Σ_k specify the orientation of the kth cluster, the biggest eigenvalue specifies its variance or size, and the ratios of the other eigenvalues to the largest one specify its shape. We can constrain some but not all of these features (orientation, size and shape) to be the same across clusters. For example, if we let $\Sigma_k = \sigma_k^2 I$, the criterion corresponds to hyperspherical clusters of different sizes; this is the "Spherical" criterion.

A criterion that appears to work well in a variety of situations results from constraining only the shape to be the same across clusters; this is denoted by S^*. Here the user must specify the shape, represented by the eigenvalue ratios $\alpha_j = \lambda_j/\lambda_1$ $(j = 2,\ldots,p)$, where $\{\lambda_1,\ldots,\lambda_p\}$ are the eigenvalues ordered from

largest to smallest. Specifying each $\alpha_j = 0.2$ leads to elliptical clusters that are moderately concentrated about a line in p-space, while choosing each $\alpha_j = 0.01$ yields very concentrated and linear clusters. Setting each $\alpha_j = 1$ just gives the Spherical criterion as a special case. The user's choice will be determined by the kind of data that he or she is working with, but we have found setting each $\alpha_j = 0.2$ often to be a good first guess.

Table 18.2 shows the different model-based clustering criteria and the assumptions that they embody.

Table 18.2: *Model-based clustering criteria with corresponding assumptions.*

Criterion	Reference	Distribution	Orientation	Size	Shape
Sum of Squares	Ward (1963)	Spherical	None	Same	Same
Spherical	Banfield and Raftery (1992)	Spherical	None	Different	Same
Determinant	Friedman and Rubin (1967)	Ellipsoidal	Same	Same	Same
S	Murtagh and Raftery (1984)	Ellipsoidal	Different	Same	Same
S^*	Banfield and Raftery (1992)	Ellipsoidal	Different	Different	Same
Unconstrained	Scott and Symons (1971)	Ellipsoidal	Different	Different	Different

Choosing the Number of Clusters

In model-based clustering, choosing the number of clusters is the same as choosing a model for the data. A standard approach to this is to calculate the Bayes factor, B_k, for the model defined by k clusters against the model defined by a single cluster (that is, all the objects belong to the same group). The Bayes factor is the odds for one model against another given the data (provided that one has no initial preference for either model). Thus the larger B_k, the more evidence there is for the existence of k clusters.

The approximate weight of evidence for k clusters (AWE_k) is an approximation to $2 \log B_k$; see Banfield and Raftery (1992). This is calculated by mclust. The larger AWE_k, the more evidence there is for the existence of k clusters. by definition, $\text{AWE}_1 = 0$, so if all the AWE_k ($k = 2,\ldots,n$) are negative,

there is no evidence for any clustering.

The value of k which maximizes AWE_k is the number of clusters for which there is the most evidence. However, we do not recommend using the AWE criterion to choose a single number of clusters unless the evidence is overwhelming. Rather, we suggest that the plot of AWE_k be inspected with a view to picking several plausible possibilities to be further investigated. The change in the approximate weight of evidence, AWE_k - AWE_{k-1}, is often large and positive for the first few values of k, $k = 2,...,K$, say, and small or negative thereafter. If that is the case, ideas of parsimony suggest considering the classification into K groups, as well as the value of k which maximizes AWE_k, and any intervening values.

Robust Clustering

So far, it has been assumed that each object belongs to a cluster. However, even when a data set is made up mainly of clusters of the prescribed type, there may be other data points that do not follow this pattern. This possibility can be allowed for by extending the model (18.1) to include such isolated observations, or outliers, assumed to occur according to a Poisson process with an intensity which is constant over the region from which the data have been drawn. The likelihood (18.1) is modified accordingly. This yields a class of clustering algorithms designed to be robust to outliers; see Banfield and Raftery (1992).

18.4 PERFORMING CLUSTER ANALYSIS IN S-PLUS

The function `mclust` does most of the analyses described in this chapter. It carries out hierarchical agglomerative clustering using the six model-based criteria shown in table 18.2, and also the five heuristic criteria discussed in section 18.2. For the model-based criteria, it returns the AWE statistic for each number of clusters k; this is used to determine the number of clusters.

If `noise=T` is specified in `mclust`, it will do robust clustering (available for the model-based criteria only). If the existence of outliers is suspected, it may be a good idea to run `mclust` with `noise=F` and `noise=T` and to compare the results. Important differences between the resulting classifications would suggest that there are outliers that are contaminating the results, in which case either these outliers could be removed from the data sets and studied separately, or the robust clustering results (with `noise=T`) could be used. Note that the *number* of clusters indicated by the AWE in the non-robust case (`noise=F`) will tend to be larger than in the robust case (`noise=T`), because in the non-robust case some of the outliers may be classified as single-member groups.

Iterative relocation for any of the eleven criteria listed can be done using the function `mreloc`. The function `mclass` takes the output of `mclust` or `mreloc` and produces a classification of the data objects.

The output of `mclust` and `mreloc` can be used to plot and manipulate classification trees. The function `plclust` plots the tree, `subtree` extracts part of the tree, `clorder` reorders the leaves of the tree, `labclust` labels the leaves of the tree, and `cutree` creates groups using the tree.

The function `hclust` also does hierarchical agglomerative clustering, but only for three of the heuristic criteria included in `mclust`. `mclust` is much more general and is to be preferred for many purposes. However, `hclust` has two features which can be advantages in certain situations. It takes as argument a distance matrix rather than a data matrix, and it is applicable even when the data cannot be represented by points in Euclidean space; it accepts a dissimilarity matrix which need not be a distance matrix in the strict sense. A distance matrix can be calculated from a data matrix using the function `dist`. Also, unlike `mclust`, `hclust` returns the height at which each merger was made; this can yield more informative plots of the classification tree. The function `kmeans` does iterative relocation, but only for the sum of squares criterion.

Example of Simple Use

```
> elect.years <- c( "1960", "1964", "1968", "1972", "1976")
> votes.S <- mclust( votes.repub[,elect.years], method="S",
+    noise=T)
> motif()
> # display dendrogram
> plclust( votes.S$tree, label = state.abb)
> # plot the awe
> plot( x = 1:length(votes.S$awe), y = votes.S$awe)
> # 9-cluster classication
> votes.9 <- mclass( votes.S, 9)
> # 3-cluster classification
> votes.3 <- mclass( votes.S, 3, votes.9)
> votes.3 <- mreloc( votes.3, votes.repub[,elect.years],
+    method="S", noise = T)
```

18.5 CLUSTERING FUNCTIONS FOUND IN THE CLUSTER LIBRARY

This section describes a collection of clustering methods based on original work by L. Kaufman and P. J. Rousseeuw. These methods are available to S-PLUS users in the form of a library.

Review of Cluster Analysis

Cluster Analysis is the searching for groups (*clusters*) in the data, in such a way that objects belonging to the same cluster resemble each other, whereas objects in different clusters are dissimilar.

In two or three dimensions, clusters can be visualized. With more than three dimensions, or in the case of *dissimilarity* data (see below), we need some kind of analytical assistance.

Generally speaking, clustering algorithms fall into two categories:

1. *Partitioning Algorithms*.
   A partitioning algorithm describes a method that divides the data set into k clusters, where the integer k needs to specified by the user. Typically, the user runs the algorithm for a range of k-values. For each k, the algorithm carries out the clustering and also yields a 'quality index', which allows the user to select the 'best' value of k afterwards. Algorithms of this type described in this chapter are used by the functions `pam`, `clara` and `fanny`.

2. *Hierarchical Algorithms*.
   A hierarchical algorithm describes a method yielding an entire hierarchy of clusterings for the given data set. *Agglomerative* methods start with the situation where each object in the data set forms its own little cluster, and then successively merges clusters until only one large cluster remains which is the whole data set. The function `agnes` uses an agglomerative method. *Divisive* methods start by considering the whole data set as one cluster, and then splits up clusters until each object is separate. Algorithms of this type are used in the functions `diana` and `mona`.

Attaching the cluster Library

To access the clustering functions described in this chapter you must attach the cluster library to your S-PLUS session type

```
>library(cluster)
```

The associated help files will also be attached at this point, and become immediately available for use.

Input Structures for Clustering Algorithms

Data sets for clustering can have either of the following structures:

1. $n \times p$ data matrix:

$$\begin{bmatrix} x_{11} & \cdots & x_{1p} \\ \vdots & & \vdots \\ x_{n1} & \cdots & x_{np} \end{bmatrix}$$

where rows stand for objects and columns stand for variables.

2. $n \times n$ dissimilarity matrix:

$$\begin{bmatrix} 0 & & & \\ d(2,1) & 0 & & \\ d(3,1) & d(3,2) & 0 & \\ \vdots & \vdots & \vdots & \\ d(n,1) & d(n,2) & \ldots & \ldots\ 0 \end{bmatrix}$$

where $d(i,j) = d(j,i)$ measures the "difference" or *dissimilarity* between the objects i and j. This kind of data occurs frequently in the social sciences and in marketing.

Most of the clustering algorithms considered here operate on a dissimilarity matrix. If the data consist of an $n \times p$ data matrix, the algorithm first constructs the corresponding dissimilarity matrix.

Dissimilarity Matrices: the Function `daisy`

The function `daisy` constructs a dissimilarity matrix. This is described in full in Kaufman and Rousseeuw (1990, chapter 1). Compared to the S-PLUS function `dist` whose input must be numeric variables, the main feature of `daisy` is its ability to handle other variable types (for example, nominal, ordinal, asymmetric binary) even when the different types occur in the same data set.

Dissimilarities

The dissimilarity between two objects measures "how different" they are. Sometimes we can use an actual metric (distance function) between objects, but a dissimilarity function is not necessarily a metric. Often only the following three axioms of a metric are satisfied:

1. $d(i,i) = 0$
2. $d(i,j) \geq 0$
3. $d(i,j) = d(j,i)$

Computation

How we compute the dissimilarity between two objects depends on the type of the original variables.

1. Interval-Scaled Variables

These are continuous measurements on a (roughly) linear scale. Typical examples are temperature, height, weight, and energy.

If all variables are interval-scaled, we can use an actual metric such as:

$$d(i,j) = \sqrt{\sum_{f=1}^{p} (x_{if} - x_{jf})^2} \qquad \text{(Euclidean distance)} \tag{18.2}$$

or

$$d(i,j) = \sum_{f=1}^{p} |x_{if} - x_{jf}| \qquad \text{(Manhattan distance)} \tag{18.3}$$

Note that the choice of measurement units strongly affects the resulting clustering. The variable with the largest dispersion will have the largest impact on the clustering. If all variables are considered equally important, the data need to be standardized first.

Put $m_f = \frac{1}{n}\sum_{i=1}^{n} x_{if}$ and $s_f = \frac{1}{n}\sum_{i=1}^{n} |x_{if} - m_f|$; then the standardized measurements are defined as follows:

$$z_{if} = \frac{x_{if} - m_f}{s_f} \tag{18.4}$$

Here we have used s_f, the *mean absolute deviation* instead of the usual standard deviation, because the former is more robust: since the deviations are not squared, the effect of outliers is somewhat reduced. Of course, there are more robust measures of dispersion, such as the median absolute deviation (the function `mad` in S-PLUS). The advantage of using a robust measure of dispersion is that the *z*-scores of outliers do not become too small, hence the outliers remain detectable (and hence visible in the clustering).

2. Continuous Ordinal Variables

These are continuous measurements on an unknown scale, or where only the ordering is known but not the actual magnitude. Then the dissimilarities are computed as follows:

1. Replace the x_{if} by their rank $r_{if} \in \{1,...,M_f\}$.

2. Transform the scale to [0,1] as follows: $z_{if} = \frac{r_{if} - 1}{M_f - 1}$.

3. Compute the dissimilarities as for interval-scaled variables.

3. Ratio-Scaled Variables

These are positive continuous measurements on a nonlinear scale, such as an exponential scale. One example would be the growth of a bacterial population (say, with a growth function Ae^{Bt}). With this model, equal time intervals multiply the population by the same ratio.

There are different ways to compute dissimilarities for ratio-scaled variables:

1. Simply as interval-scaled variables, though this is not recommended as it can distort the measurement scale.

2. As continuous ordinal data.

3. By first transforming the data (perhaps by taking logarithms), and then treating the results as interval-scaled variables.

4. Discrete Ordinal Variables

A variable of this type has *M* possible values (scores) which are ordered. The dissimilarities are computed in the same way as for continuous ordinal variables.

5. Nominal Variables

Such a variable has *M* possible values, which are not ordered. The dissimilarity between objects *i* and *j* is usually defined as:

$$d(i,j) = \frac{\text{\# variables taking different values for } i \text{ and } j}{\text{total number of variables}}$$

This is called the *simple matching coefficient*.

6. Symmetric Binary Variables

These have two possible values, coded 0 and 1, which are *equally* important (such as male and female, or vertebrate and invertebrate).

Symmetric binary variables are nominal variables, hence we again use the simple matching coefficient given above for *Nominal Variables*. Let us also consider the contingency table of the objects *i* and *j*:

i\j	1	0
1	a	b
0	c	d

We can then rewrite the simple matching coefficient as

$$d(i,j) = \frac{b+c}{a+b+c+d}. \tag{18.5}$$

7. Asymmetric Binary Variables

Here the variable has two possible values, one of which carries *more* importance than the other. The most meaningful outcome is coded as 1, and the less meaningful outcome as 0. Typically, 1 stands for the presence of a certain attribute (for example, a particular disease), and 0 for its absence.

The dissimilarity between i and j is then defined as:

$$d(i,j) = \frac{\#\text{ variables taking different values for } i \text{ and } j}{\text{total number of meaningful comparisons}}$$

Using the contingency table again, this becomes $d(i,j) = \frac{b+c}{a+b+c}$, which is called the *Jaccard coefficient*.

8. Variables of Mixed Types

The above formulas hold when all variables in the data set are of the same type. However, many data sets contain variables of different types. Therefore, we want a method to compute dissimilarities between objects when the data set contains p variables that may be of different types. For this the function `daisy` uses the formula

$$d(i,j) = \frac{\sum_{f=1}^{p} \delta_{ij}^{(f)} \, d_{ij}^{(f)}}{\sum_{f=1}^{p} \delta_{ij}^{(f)}} \in [0,1] \tag{18.6}$$

where $\delta_{ij}^{(f)} = 0$ if x_{if} or x_{jf} is missing, $\delta_{ij}^{(f)} = 0$ if $x_{if} = x_{jf} = 0$ and variable f is asymmetric binary $\delta_{ij}^{(f)} = 1$ otherwise. And $d_{ij}^{(f)}$ = the contribution of variable f, which depends on its type:

1. f binary or nominal: $d_{ij}^{(f)} = 0$ if $x_{if} = x_{jf}$, and $d_{ij}^{(f)} = 1$ otherwise.

2. f interval-scaled: $d_{ij}^{(f)} = \frac{|x_{if} - x_{jf}|}{max_h x_{hf} - min_h x_{hf}}$

3. ordinal and ratio-scaled variables: compute ranks r_{if} and $z_{if} = \frac{r_{if} - 1}{M_f - 1}$ and treat these z_{if} as interval-scaled.

Partitioning Around Medoids: the Function `pam`

The method `pam` is fully described in chapter 2 of Kaufman and Rousseeuw (1990). Compared to the function `kmeans` (refer to the on-line help for more information) the function `pam` has the following features: (a) it accepts a dissimilarity matrix; (b) it is more robust because it minimizes a sum of dissimilarities instead of a sum of squared euclidean distances; (c) it provides a novel graphical display, the *silhouette plot* (see below). It also allows the user to select the number of clusters.

Algorithm

The function `pam` operates on the dissimilarity matrix of the given data set. When it is presented with an $n \times p$ data matrix, `pam` will first compute a dissimilarity matrix.

The algorithm computes k *representative objects*, called *medoids*, which together determine a clustering. The number k of clusters is an argument of the function.

Each object is then assigned to the cluster corresponding to the nearest medoid. That is, object i is put into cluster v_i when medoid m_{v_i} is nearer than any other medoid m_w:

$d(i, m_{v_i}) \leq d(i, m_w)$ for all w=1,...,k.

The k representative objects should minimize the sum of the dissimilarities of all objects to their nearest medoid:

$$\text{objective function} = \sum_{i=1}^{n} d(i, m_{v_i})$$

The algorithm proceeds in two steps:

1. *Build-step*
   This step sequentially selects k "centrally located" objects, to be used as initial medoids.

2. *Swap-step*
   If the objective function can be reduced by interchanging

(swapping) a selected object with an unselected object, then the swap is carried out. This is continued until the objective function no longer decreases.

Graphical Display: the Silhouette Plot

A partition of the data, such as the clustering found by `pam`, can be displayed by means of the *silhouette plot* (Rousseeuw 1987).

For each object i the silhouette value $s(i)$ is computed, and then represented in the plot as a bar of length $s(i)$. In order to define $s(i)$, A denotes the cluster to which object i belongs, and the calculation proceeds as

$a(i)$ = average dissimilarity of i to all other objects of A

Now consider any cluster C different from A and define

$d(i,C)$ = average dissimilarity of i to all objects of C

After computing $d(i,C)$ for all clusters C *not equal to* A we take the smallest of those: $b(i) = \min_{C \neq A} d(i, C)$

The cluster B which attains this minimum, namely $d(i,B)=b(i)$, is called the *neighbor* of object i. This is the second-best cluster for object i.

The value $s(i)$ can now be defined:

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}} \tag{18.7}$$

We see that $s(i)$ always lies between -1 and 1. The value $s(i)$ may be interpreted as follows:

$s(i) \approx 1 \Rightarrow$ object i is well classified

$s(i) \approx 0 \Rightarrow$ object i lies between two clusters

$s(i) \approx -1 \Rightarrow$ object i is badly classified

The silhouette of a cluster is a plot of the $s(i)$, ranked in decreasing order, of all its objects i. The entire silhouette plot shows the silhouettes of all clusters next to each other, so the "quality" of the clusters can be compared. The *overall average silhouette width* of the silhouette plot is the average of the $s(i)$ over all objects i in the data set.

It is possible to run `pam` several times, each time for a different k, and to compare the resulting silhouette plots. The user can then select that value of k yielding the highest average silhouette width. If even that highest width is below (say) 0.25, one may conclude that no substantial structure has been found.

See figure 18.2 and figure 18.3 for examples of silhouette plots.

Clustering Large Applications: the Function `clara`

The method `clara` is fully described in chapter 3 of Kaufman and Rousseeuw (1990). Compared to other partitioning methods such as `pam`, `clara` can deal with much larger data sets. Internally, this is achieved by considering data subsets of fixed size, so that the overall time and storage requirements become linear in the total number of objects, rather than quadratic.

The function `pam` needs to store the dissimilarity matrix of the entire data set (which has $O(n^2)$ entries) in central memory, while its computation time goes up accordingly. For larger data sets (say, with more than 250 objects) this becomes less convenient.

To avoid this problem, the function `clara` does not compute the entire dissimilarity matrix at once. Therefore, this function only accepts input of an $n \times p$ data matrix.

Algorithm

The algorithm takes a data subset, and then applies the `pam` algorithm to it. This divides the data subset into k clusters. The remaining objects of the original data set are then assigned to the nearest medoid. In this way, all n objects are assigned. The objective function is then computed for the entire data set, namely by summing all n terms $d(i, m_{v_i})$.

This procedure is repeated for several data subsets, and the clustering with the lowest overall objective function is retained. In this way, we only need to compute and store the dissimilarity matrix of one data subset at any one time, which makes the overall order of complexity linear in n.

The first data subset is drawn randomly. Each of the following data subsets is forced to contain the currently best medoids, supplanted with randomly drawn objects.

Graphical Display

The clustering obtained by `clara` can also be represented by means of a silhouette plot, described in the previous section on `pam`. Due to the potential sizes of the data sets, the silhouette plot is given only for the best data subset.

Fuzzy Analysis: the Function `fanny`

The method `fanny` is fully described in chapter 4 of Kaufman and Rousseeuw (1990). Compared to other fuzzy clustering methods, `fanny` has the following features: (a) it accepts a dissimilarity matrix; (b) it is more robust to the 'spherical cluster' assumption (see Kaufman and Rousseuw); (c) graphical display is in the form of a silhouette plot.

The functions `pam` and `clara` are *crisp* clustering methods. This means that each object of the data set is assigned to exactly one cluster. For instance, an object lying between two clusters must be assigned to one of them.

Fuzzy methods like `fanny` will spread each object over the various clusters. For each object i and each cluster v there will be a *membership* u_{iv} which indicates how strongly object i belongs to cluster v.

Memberships have to satisfy the following conditions:

1. $u_{iv} \geq 0$ for all i=1,...,n and all v=1,...,k.
2. $\sum_{v=1}^{k} u_{iv} = 1 = 100\%$ for all i=1,...,n.

Algorithm

The memberships are defined through minimization of:

$$\text{objective function} = \sum_{v=1}^{k} \frac{\sum_{i,j=1}^{n} u_{iv}^2 u_{jv}^2 d(i,j)}{2\sum_{j=1}^{n} u_{jv}^2} \tag{18.8}$$

In this expression, the dissimilarities $d(i,j)$ are known and the memberships u_{iv} are unknown. The minimization is carried out numerically by means of an iterative algorithm, taking into account the above conditions that memberships need to obey. To have an idea of "how fuzzy" the resulting clustering is, *Dunn's partition coefficient* is computed:

$$F_k = \sum_{i=1}^{n} \sum_{v=1}^{k} \frac{u_{iv}^2}{n} \tag{18.9}$$

which always lies in the range $\left[\frac{1}{k}, 1\right]$.

This coefficient attains its extreme values in the following situations:

1. *entirely fuzzy clustering;* all $u_{iv} = \frac{1}{k} \Rightarrow F_k = nk\frac{1}{nk^2} = \frac{1}{k}$
2. *crisp clustering*; all $u_{iv} = 0$ or . $\Rightarrow F_k = \frac{n}{n} = 1$

The normalized version of this coefficient is

$$F_k' = \frac{F_k - \frac{1}{k}}{1 - \frac{1}{k}} = \frac{kF_k - 1}{k - 1} \tag{18.10}$$

which always lies in the range [0,1].

Graphical Display

For any fuzzy clustering, such as the one produced by `fanny`, the *nearest crisp clustering* method should be considered for graphical output. It assigns each object i to the cluster v in which it has the highest membership u_{iv}. This crisp clustering is then represented graphically by means of a silhouette plot.

Agglomerative Nesting: the Function `agnes`

The method `agnes` is fully described in chapter 5 of Kaufman and Rousseeuw (1990). Compared to other agglomerative clustering methods such as `hclust` (see the help files for more information), `agnes` has the following features: (a) it yields the *agglomerative coefficient* which measures the amount of clustering structure found; (b) apart from the usual clustering tree it also utilizes the *banner* plot.

As the function `agnes` is an agglomerative hierarchical clustering method, it yields a sequence of clusterings. In the first clustering each of the n objects forms its own separate cluster. In subsequent steps clusters are merged, until (after n–1 steps) only one large cluster remains, consisting of all the objects.

Algorithm

The algorithm is based on dissimilarities only. If a data matrix is input, the function starts by computing the dissimilarity matrix.

Initially (at step 0), each object is considered as a separate cluster. All the other steps have the following form:

1. Merge the two clusters with smallest between-cluster dissimilarity;
2. Compute the dissimilarity between the new cluster and all remaining clusters.

The between-cluster dissimilarity can be defined in various ways, notably:

1. Group average method

$$d(R, Q) = \frac{1}{|R||Q|} \sum_{i \in R,\ j \in Q} d(i, j)$$

2. Nearest neighbor method = single linkage method

$$d(R, Q) = \min_{i \in R,\ j \in Q} d(i, j)$$

3. Furthest neighbor method = complete linkage method

$$d(R, Q) = \max_{i \in R,\ j \in Q} d(i, j)$$

The *group average method* is taken as the default, based on arguments of robustness and consistency.

The function `agnes` also provides the *agglomerative coefficient* (Rousseeuw 1986), which measures the clustering structure of the data set.

For each object i, $d(i)$ denotes its dissimilarity to the first cluster it is merged with, divided by the dissimilarity of the merger in the last step of the algorithm. The agglomerative coefficient (AC) is defined as the average of all $1-d(i)$.

Because the AC grows with the number of objects, this measure should not be used to compare data sets of very different sizes.

Graphical Display: the Clustering Tree and Banner

The hierarchy obtained from `agnes` can be graphically displayed in two ways, by means of a *clustering tree* or by a *banner.*

1. *Clustering tree.*
   This is a tree in which the leaves represent objects. The vertical coordinate of the place where two branches join equals the dissimilarity between the corresponding clusters.

2. *Banner.*
   The banner shows the successive mergers from left to right. (Imagine the ragged flag parts at the left, and the flagstaff at the right.) The objects are listed from top to bottom. The mergers (which commence at the between-cluster dissimilarity) are represented by horizontal bars of the correct length. The banner thus contains the same information as the clustering tree.

Note that the agglomerative coefficient (AC) defined above can also be defined as the average width (or the percentage filled) of the banner plot.

Divisive Analysis: the Function `diana`

The method `diana` is fully described in chapter 6 of Kaufman and Rousseeuw (1990). It is probably unique in computing a divisive hierarchy, because most other software for hierarchical clustering is agglomerative. Moreover, `diana` provides (a) the divisive coefficient which measures the amount of clustering structure found; and (b) the banner plot.

The function `diana` is a divisive hierarchical method. The initial clustering (at step 0) consists of one large cluster containing all n objects. In each subsequent step, the largest available cluster is split into two smaller clusters, until finally all clusters contain but a single object.

In the first step of an agglomerative method, there are $\binom{n}{2} = \frac{n(n-1)}{2}$ possible ways to merge two clusters. But in the first step of a divisive method, we are faced with `2n-1-1` possibilities to split up the data set into two clusters. The latter number is much larger than the first, and in practice it is not feasible to try all possible splits.

Algorithm

To avoid considering all possible splits, `diana` divides the data set in the following way (based on dissimilarities only).

1. Find the most disparate object, which is the one with the highest average dissimilarity to the other objects. This object initiates the *splinter group*, analogous to a dissenting fraction of a political party.
2. For each object *i* outside the splinter group, compute $V_i = \text{average}_{j \notin \text{splinter group}} d(i,j) - \text{average}_{j \in \text{splinter group}} d(i,j)$. To find the object *h* for which this difference is largest; if $V_h > 0$, then *h* is on average closer to the splinter group than to the remainder, so add object *h* to the splinter group.
3. Repeat step 2 until all differences V_h are negative. The data set is then split into two clusters.
4. Select the cluster with the largest diameter. (The diameter of a cluster is the largest dissimilarity between any two of its objects.) Then divide this cluster as in steps 1 to 3.
5. Repeat step 4 until all clusters contain only a single object.

The function `diana` also provides the *divisive coefficient* (Rousseeuw 1986), which measures the clustering structure of the data set.

For each object *i*, $d(i)$ denotes the diameter of the last cluster to which it belongs (before being split off as a single object), divided by the diameter of the whole data set.

The divisive coefficient (DC) is then defined as the average of all $d(i)$.

Like the AC in the previous section on `agnes`, the DC also grows with the number of objects. Therefore, the DC should not be used to compare data sets of very different sizes.

Graphical Display

The hierarchy obtained from `diana` can again be graphically displayed either as a clustering tree or as a banner.

Note that the divisive coefficient (DC) defined above can also be defined as the average width (or the percentage filled) of the banner plot.

Monothetic Analysis: the Function mona

The method mona is fully described in chapter 7 of Kaufman and Rousseeuw (1990). It is a different type of divisive hierarchical method. Contrary to diana, which can process a dissimilarity matrix as well as a data matrix with interval-scaled variables, mona operates on a data matrix with binary variables. For each split mona uses a single (well-chosen) variable, which is why it is called a *monothetic* method. Most other hierarchical methods (including agnes and diana) are *polythetic* (that is, they use all variables simultaneously).

Algorithm

First all missing values in the binary data matrix (all those values *not* = 0 or 1) are replaced by estimated values, obtained as follows. Suppose that x_{if} is missing. Then we consider any other variable g, and construct the contingency table

f\g	1	0
1	a_{fg}	b_{fg}
0	c_{fg}	d_{fg}

The association between f and g is then defined as $A_{fg} = | a_{fg}\, d_{fg} - b_{fg}\, c_{fg} |$

The variable t for which $A_{ft} = \max_g A_{fg}$ is the most correlated with variable f. The missing values of f are then estimated by means of variable t in the following way:

put $x_{if} = x_{it}$ when $a_{ft}\, d_{ft} - b_{ft}\, c_{ft} > 0$

put $x_{if} = 1 - x_{it}$ when $a_{ft}\, d_{ft} - b_{ft}\, c_{ft} < 0$

When all missing values have been replaced, the actual splitting can begin. (If the data matrix cannot be filled in completely, due to too many missing values in the original data, the method stops with a warning message.)

The mona algorithm constructs a hierarchy of clusterings, starting with one large cluster. In each step, each available cluster is divided according to one variable. The cluster is divided into two: one cluster with all objects having value 1 for that variable, and another cluster with all objects having value 0 for that variable.

The variable used for splitting a cluster is the variable with the largest total association to the other variables. The association between variables f and g is given by the expression A_{fg} above, but now the contingency table only uses the objects of the cluster to be split. The total association of a variable f is

then defined as:

$$A_f = \sum_{g \neq f} A_{fg} \tag{18.11}$$

The variable t which satisfies $A_t = \max_f A_f$ is selected for splitting the cluster. We continue to divide clusters in this way, until each cluster consists of objects having identical values for all variables. Such clusters cannot be split any more. A final cluster is thus a singleton or an indivisible cluster.

Graphical Display

The clustering hierarchy constructed by `mona` can be represented by means of a banner. This is again a divisive banner, however the length of a bar is now given by the number of divisive steps needed to make that split. Inside the bar, the variable is listed which was responsible for the split.

18.6 FUNCTION IMPLEMENTATION ISSUES

Object-Oriented Structure

The algorithms of Kaufman and Rousseeuw (1990), summarized above, have been implemented in S-PLUS as a library of functions, which generate objects of seven different classes. For each class of objects, methods for textual or graphical output are available. Most of the objects are named after the function that generates them. In this way, classes `pam`, `clara`, `fanny`, `agnes`, `diana`, and `mona` exist. The seventh class, class `dissimilarity`, is generated by the function `daisy`, but will also be part of the objects of classes `pam`, `clara` and `fanny`.

Some of these classes are grouped together, and inherit from the same superclass. The created hierarchy of classes is as follows:

1. Class `dissimilarity`
2. Class `partition`
   (a) Class `pam`
   (b) Class `clara`
   (c) Class `fanny`
3. Class `agnes`
4. Class `diana`
5. Class `mona`

These classes have methods for the following functions:

1. `print`
   For classes `dissimilarity`, `pam`, `clara`, `fanny`, `agnes`, `diana`, and `mona`.
2. `summary`
   For classes `pam`, `clara`, `fanny`, `agnes`, `diana`, and `mona`. These `summary` methods return new objects of class `summary.<old-class>`. For each of those new summary classes, a `print` method is available.
3. `plot`
   For classes `partition`, `agnes`, `diana`, and `mona`.
4. `pltree`
   For classes `agnes`, and `diana`.

The `partition` class has a method for the generic `plot` function that is common to all its subclasses.

Calling the Functions

The `daisy` function, for calculating dissimilarities, is similar to the `dist` function in S-PLUS. One advantage of `daisy` it that it accepts data sets with different types of variables. The function's header is

```
daisy(x, metric = "euclidean", stand = F, type = list())
```

When all variables are interval scaled, this specifies the metric to be used for calculating dissimilarities, and whether or not to standardize first. When other variable types occur, a list of types can be given. The output of `daisy` can be used as input for several of the new clustering functions.

The input arguments of the six clustering functions are similar. The calls to the six functions are the following.

```
pam(x, k, diss = F, metric = "euclidean", stand = F)
clara(x, k, metric = "euclidean", stand = F, samples = 5,
sampsize = 40+2*k)
fanny(x, k, diss = F, metric = "euclidean", stand = F)
agnes(x, diss = F, metric = "euclidean", stand = F,
method = "average")
diana(x, diss = F, metric = "euclidean", stand = F)
mona(x)
```

All functions, except for `clara` and `mona`, accept two possible input structures: a dissimilarity matrix or a data matrix. The logical argument `diss` tells the algorithm how `x` should be interpreted, the default being a data

matrix of observations by variables. When a dissimilarity matrix is given as input, it is preferably an object of class `dissimilarity`. However, the functions will also accept dissimilarities produced with `dist`, or a vector that can be interpreted as a dissimilarity matrix.

The algorithms of `clara` and `mona` don't accept dissimilarities as input, but only accept the second input form: a matrix of observations by variables.

If a function has to compute dissimilarities from a given data matrix, the function needs to know which metric to use and whether or not to standardize first. These arguments are similar to the corresponding arguments of `daisy`. Since `mona` doesn't compute dissimilarities, it does not have the arguments `metric` and `stand`.

The function `clara` has two additional arguments, specifying the number of samples and the size of each sample. Also `agnes` has a special argument defining the method to be used for calculating dissimilarities between clusters.

Sometimes the output of the functions is rather extensive, especially when the `summary` method is invoked for an object of one of the `partition` classes. In those cases, the output scrolls off the screen. Therefore, all available components of the output are listed on the last output lines. Those components can be extracted from the result like a component from a list: `object$component`.

Objects resulting from the clustering functions can be given as input to high level graphics functions:

- a silhouette plot of an object of class `partition` is made on the current graphics device with the `plot` method.
- a banner of an object of class `agnes`, `diana` or `mona` is also constructed with the `plot` method.
- a clustering tree of an object of class `agnes` or `diana` is plotted on the current graphics device with the `pltree` method.

One partitioning method, `kmeans`, already exists in S-PLUS. The new function `pam` has additional possibilities: it accepts a dissimilarity matrix as input, and silhouette plots are supported.

The function `hclust`, an agglomerative hierarchical method, also exists in S-PLUS. In comparison with `hclust`, `agnes` has two new features: it computes the agglomerative coefficient and allows for banner plots.

More information and details about the input arguments and the structure of

the output can be found in the help files.

Table 18.3: *Summary of Clustering functions*

Function	Description and example function call
`daisy`	Computes a dissimilarity matrix from a data matrix. `daisy(x, metric = "euclidean", stand = F, type = list())`
`pam`	A crisp partitioning method for smaller data sets. `pam(x, k, diss = F, metric = "euclidean", stand = F)`
`clara`	A method for larger data sets (more than 250 objects) utilizing the function `pam`. `clara(x, k, metric = "euclidean", stand = F, samples = 5, sampsize = 40+2*k)`
`fanny`	A fuzzy partitioning method, employing the concept of memberships. `fanny(x, k, diss = F, metric = "euclidean", stand = F)`
`agnes`	An agglomerative hierarchical method, computes a measure of the clustering found. `agnes(x, diss = F, metric = "euclidean", stand = F, method = "average")`
`diana`	A divisive hierarchical method, computing a measure of the divisive clustering found. `diana(x, diss = F, metric = "euclidean", stand = F)`
`mona`	A divisive hierarchical method that works on binary data. `mona(x)`

18.7 CLUSTERING EXAMPLES

Partitioning Methods: ruspini Data

The Ruspini data were originally used by Ruspini (1970) in order to illustrate fuzzy clustering techniques. The data set consists of 75 points; see figure 18.1.

We first use `pam` to partition the data into four clusters. After that, a partition into five clusters is constructed. The four medoids resulting from the first call are points in the centers of the four clusters (see the plot). The second call to `pam` produces the same four medoids, and takes an intermediate object as the fifth medoid. The minimal value reached for the objective function is a little smaller when five clusters are formed. However, that does not necessarily

imply that the second clustering is better. From the clustering vector, and the numerical output per cluster, it can be seen that both clusterings are similar. The second partition places the three most outlying points of the third cluster in a separate cluster. This new cluster is an isolated one.

On the other hand, the clusters resulting from the second call are not as well-separated as those from the first call. Looking at the silhouette plots (see figure 18.2), the conclusion is similar. With the first clustering, all $s(i)$ are above 0.4. The second clustering yields very large silhouette widths for the new cluster with three objects. But some of the silhouette widths of the second and third cluster have decreased. That is, those objects lie somewhere between two clusters. According to the overall average silhouette width both clustering structures are approximately of the same quality, k=4 slightly preferable over k=5.

When we call `fanny` with the same data and k=4, nearly all objects have a large membership to one of the clusters (see figure 18.1). The three objects that were placed in a separate cluster when calling `pam` for k=5 now are classified in a fuzzy way, since none of their memberships is much higher than the other memberships. We conclude that the majority of the data can be divided into four clusters, but some objects are situated between the clusters. The nearest crisp clustering is the same as that from `pam` with k=4. Hence, the silhouette plots are identical. But this is not always the case. When we call `fanny` for k=5, the nearest crisp clustering is different from that produced by `pam`. The second cluster has been split instead of the third one. Because the average silhouette width is smaller than before, the clustering structure is less clear.

```
> summary(pam(ruspini,4))
Medoids:
    x   y
10 19  65
32 44 149
52 99 119
70 69  21
Clustering vector:
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
24 25
 1 1 1 1 1 1 1 1  1  1  1  1  1  1  1  1  1  1  2  2  2
2  2
 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45
46 47
  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  2  3  3
3  3
```

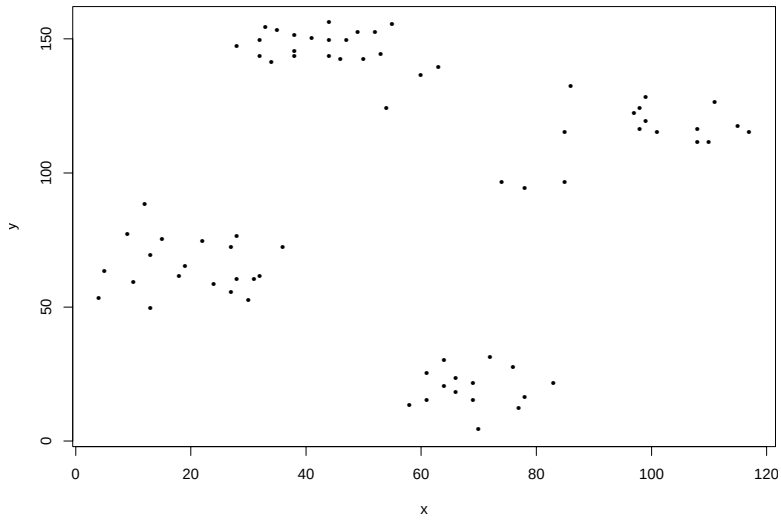

Figure 18.1: *The ruspini data.*

```
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67
68 69
  3  3  3  3  3  3  3  3  3  3  3  3  3  4  4  4  4  4  4  4
4  4
 70 71 72 73 74 75
  4  4  4  4  4  4
Objective function:
    build     swap
 17.22898 11.48637
Numerical information per cluster:

     size max_diss  av_diss diameter separation
[1,]   20 24.04163 12.55362 40.24922   40.49691
[2,]   23 26.92582 10.44238 36.61967   24.04163
[3,]   17 33.97058 13.84800 47.63402   24.04163
[4,]   15 17.02939  8.98767 27.07397   40.49691
```

```
Isolated clusters:
L-clusters: NULL
L*-clusters: [1] 1 4
Silhouette plot information:
   cluster neighbor sil_width
10       1        4 0.8056096
 6       1        4 0.7954977
 9       1        4 0.7923048
11       1        4 0.7831672
 8       1        2 0.7811793
        ...
75       4        1 0.7425538
Average silhouette width per cluster:
[1] 0.7262347 0.7548344 0.6691154 0.8042285
Average silhouette width of total dataset:
[1] 0.737657

Dissimilarities :
 [1] 10.049876  8.485281 24.515301  9.848858 18.357560
35.902646
 [7] 24.596748 16.124515 19.209373 27.658633 29.832868
33.241540
[13] 20.615528 23.086793 25.000000 26.019224 27.892651
29.120440
...
[2767] 10.770330 10.000000  7.280110 15.132746 12.529964
5.830952
[2773]  5.656854 12.369317  8.062258

Metric :  euclidean
Number of objects :  75

Available arguments:
[1] "medoids"    "clustering" "objective"  "isolation"
"clusinfo"
[6] "silinfo"    "diss"
```

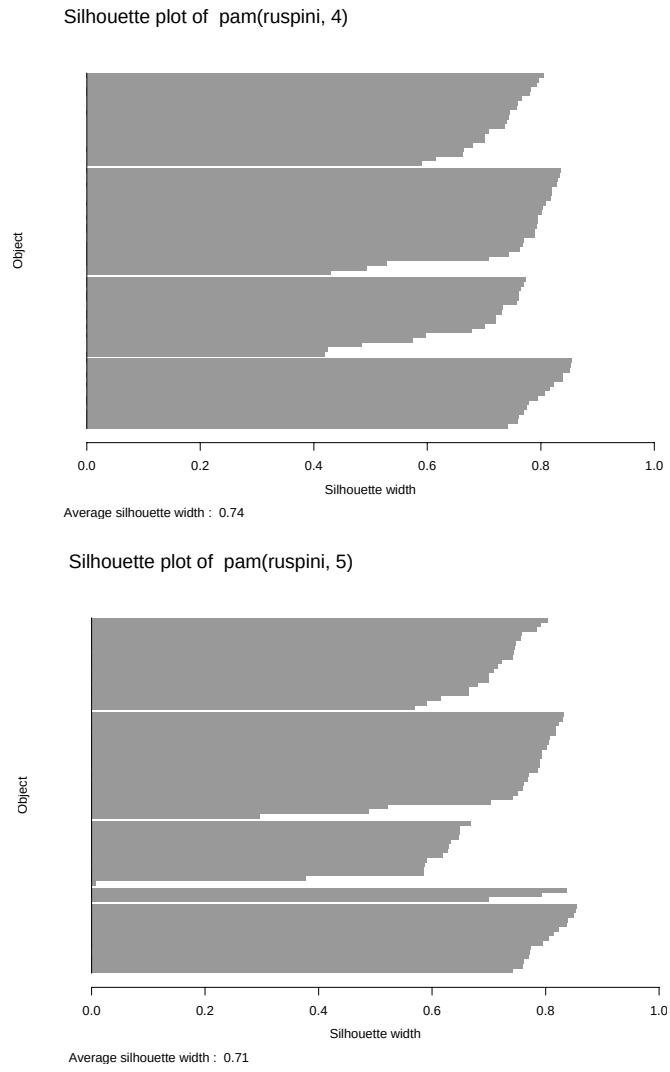

Figure 18.2: *Silhouette plots generated by* `pam(ruspini,4)` *and* `pam(ruspini,5)`.

A Larger Data set of 500 Objects

This data set, consisting of 500 two-dimensional points, is generated in S-PLUS using the following command:

```
x <- rbind( cbind(rnorm(200,0,8),rnorm(200,0,8)),
            cbind(rnorm(300,50,8),rnorm(300,50,8)))
```

A plot of the points is shown in figure 18.4.

The objects in the data set are clearly divided into two clusters. If `pam` had

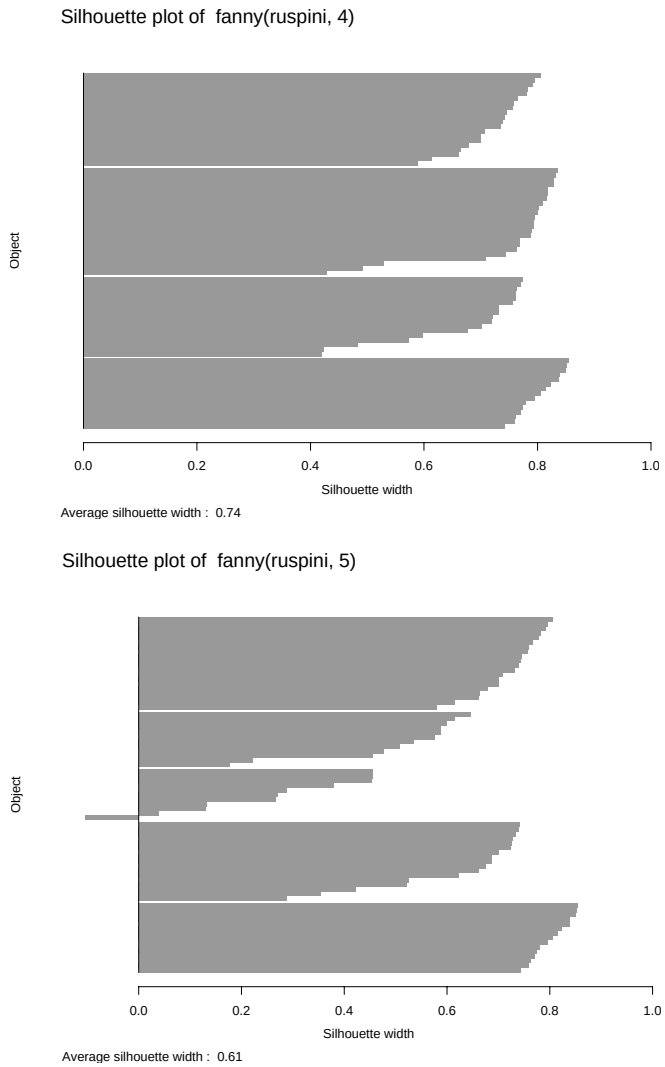

Figure 18.3: *Silhouette plots generated by* `fanny(ruspini,4)` *and* `fanny(ruspini,5)`.

been used with this data set, `124750 (= 500*499/2)` dissimilarities would have been considered. The function `clara` uses only `946 (= 44*43/2)` dissimilarities, since the default sample size is `40 + 2.k = 40 + 2.2 = 44`. `clara` still finds the correct clustering. The average silhouette width, `0.79`, indicates a good clustering structure.

```
> clara(x,2)
```

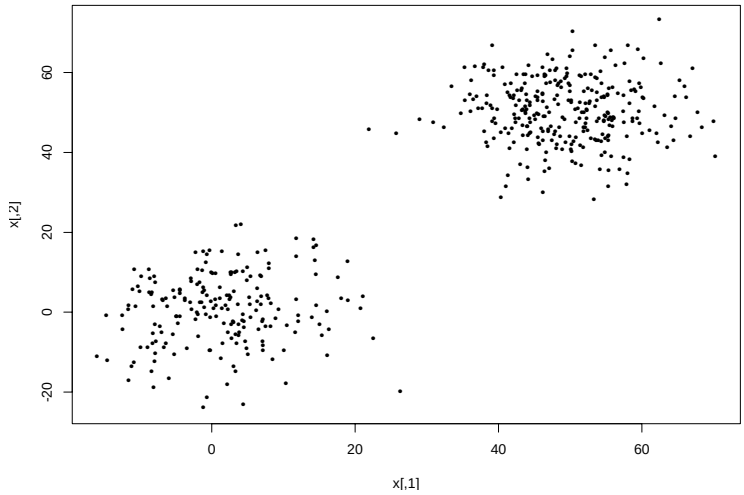

Figure 18.4: *A large data set of 50 points.*

Hierarchical Methods: "Republican Votes" Data

The `votes.repub` data set is standard in S-PLUS. This matrix contains the percentage of people in the 50 states of the USA that voted republican in the 31 presidential elections between 1856 and 1956. If a state did not yet belong to the USA in 1856, an `NA` value is given.

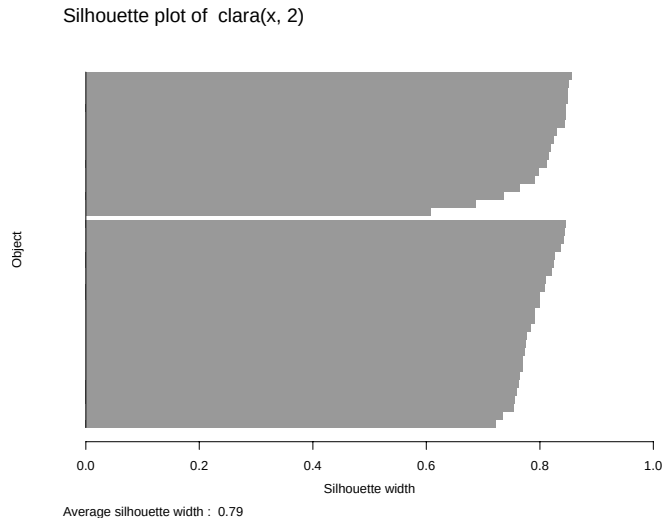

Figure 18.5: *Silhouette plot of* `clara(x,2)`*, where* `x` *is the large data set.*

When `agnes` is applied to this data set, the clustering tree indicates a division

of the data into two well-separated clusters. A cluster containing eight of the southern states is merged with the other states in the last step. The dissimilarity between the two clusters is large in comparison with the dissimilarities of the mergers at the other stages. When the complete linkage method is used, the same clustering structure is found. The clustering tree obtained by the single linkage method looks very different. Upon closer scrutiny, one sees that the states which are merged in the final steps are exactly those states that the other methods considered as a separate cluster. The single linkage method has a tendency towards chains of clusters, which causes the differences between the trees in this example. The `diana` function finds the same main clustering structure: the eight southern states are already split off at the first stage.

Since all these hierarchical methods seem to agree on the division of the data set into two clusters, the conclusion might be that the voting behavior in the southern states of the USA is rather different from that in the other states. The further division of the clusters is not so clear-cut: different methods yield more or less different structures.

```
> agnes(votes.repub)
Merge:
      [,1] [,2]
 [1,]  -12  -50
 [2,]   -7  -32
 [3,]  -14  -35
 [4,]  -13  -30
 [5,]  -25  -31
 [6,]  -37  -47
 [7,]  -21  -39
 [8,]   -3  -28
 [9,]    4  -38
[10,]  -16  -27
[11,]  -15  -41
[12,]    2  -29
[13,]    8  -26
[14,]   -2  -22
[15,]    9    3
[16,]  -33  -42
[17,]   16  -46
[18,]   -6    1
[19,]    5  -48
[20,]   12   15
[21,]   -5    6
```

```
[22,]  -11  -19
[23,]  -17  -20
[24,]  -34  -49
[25,]   18  -44
[26,]   11   10
[27,]   14   20
[28,]   -8   19
[29,]  -23   24
[30,]   28   23
[31,]   13   25
[32,]   31  -36
[33,]   27   21
[34,]   26   29
[35,]   33   34
[36,]   -1  -10
[37,]   30   17
[38,]   22    7
[39,]   32   37
[40,]   -4   -9
[41,]   35   38
[42,]   36  -43
[43,]   42  -18
[44,]  -24  -40
[45,]   43   40
[46,]   41   39
[47,]   45   44
[48,]   46  -45
[49,]   47   48
Order of objects:
 [1] Alabama        Georgia        Texas          Louisiana
 [5] Arkansas       Florida        Mississippi    South
Carolina
 [9] Alaska         Michigan       Connecticut    New York
[13] New Hampshire  Illinois       New Jersey
Pennsylvania
[17] Indiana        Ohio           California     Oregon
[21] Washington     Iowa           South Dakota   Kansas
[25] Nebraska       Minnesota      North Dakota   Wisconsin
[29] Hawaii        Maine         Massachusetts  Rhode Island
[33] Arizona        Nevada         Montana        Colorado
[37] Idaho          Wyoming        Utah           Oklahoma
```

```
[41] Delaware       Missouri       New Mexico     West
Virginia
[45] Kentucky       Maryland       North Carolina Tennessee
[49] Virginia       Vermont
Height:
 [1]  48.23970  57.99085  61.44242  63.72070  56.13635
96.95981
 [7]  63.09507 144.13654  28.14370  36.41734  19.42184
26.73269
[13]  31.03100  20.87924  25.06279  28.75456  20.22583
43.81834
[19]  31.62366  22.18307  47.44331  26.15473  35.80800
25.92214
[25]  45.21792  38.34994  33.69779  57.20589  32.76106
52.10821
[31]  22.63338  65.24701  23.42057  26.89488  40.46388
30.11337
[37]  17.19925  34.74599  43.05386  54.24960  38.14008
21.14125
[43]  30.25411  40.24728  33.47438  50.07572  29.50990
30.01351
[49] 103.01080
Agglomerative coefficient:
[1] 0.7688431
Available arguments:
[1] "order"     "height"    "ac"        "merge"
"ordertree"
```

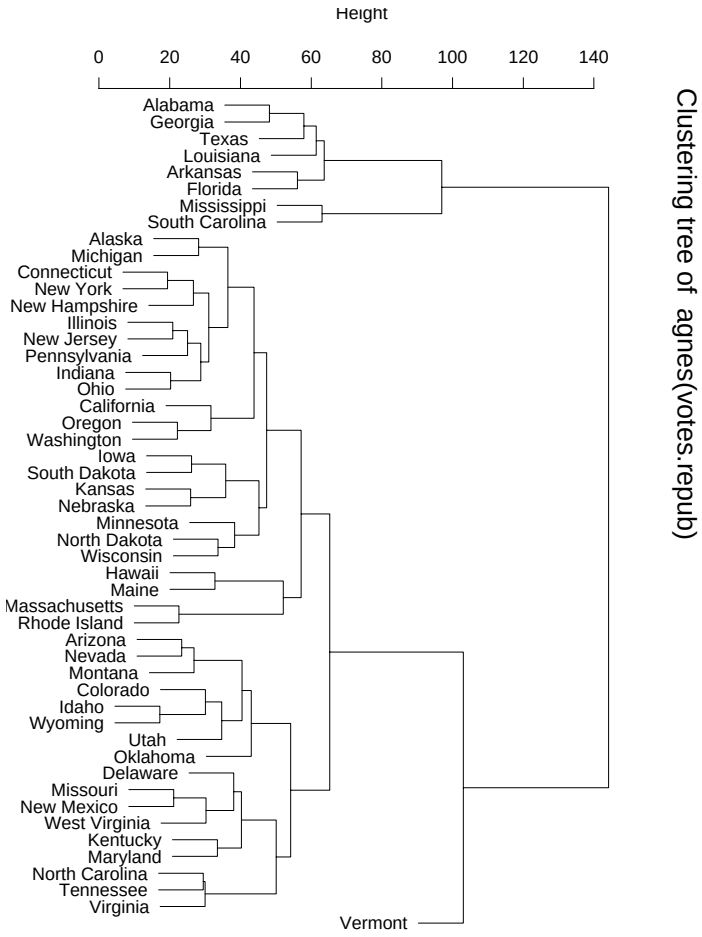

Figure 18.6: *Clustering tree of* `agnes(votes.repub)`.

"European Countries" Data

This data set is an extract from the brochure "Cijfers en feiten: Een statistisch portret van de Europese Unie" (1994) published by Eurostat, the European agency for statistics. For each country belonging to the European Union during 1994, it gives the gross national product (bbp) in 1992, and the percentage of the gross national product due to agriculture (landbouw).

Here, both partitioning and hierarchical methods yield the same division of the European countries into two clusters; with one cluster consisting of four countries that are more oriented towards agriculture and whose gross national product is relatively low..

Table 18.4: *Countries of the European Union*

Code	Country	Code	Country
B	Belgium	**I**	Italy
D	Germany	**IRL**	Ireland
DK	Denmark	**L**	Luxembourg
E	Spain	**NL**	Netherlands
F	France	**P**	Portugal
GR	Greece	**UK**	United Kingom

```
> euro
    landbouw  bbp
  B      2.7 16.8
 DK      5.7 21.3
  D      3.5 18.7
 GR     22.2  5.9
  E     10.9 11.4
  F      6.0 17.8
IRL     14.0 10.9
  I      8.5 16.6
  L      3.5 21.0
 NL      4.3 16.4
  P     17.4  7.8
 UK      2.3 14.0
> pam(euro,2)
Medoids:
  landbouw  bbp
D      3.5 18.7
P     17.4  7.8
Clustering vector:
 B DK D GR E F IRL I L NL P UK
 1  1 1  2 2 1   2 1 1  1 2  1
Objective function:
    build    swap
 3.429317 3.36061
Available arguments:
[1] "medoids"    "clustering" "objective"  "isolation"
[5] "clusinfo"   "silinfo"    "diss"
```

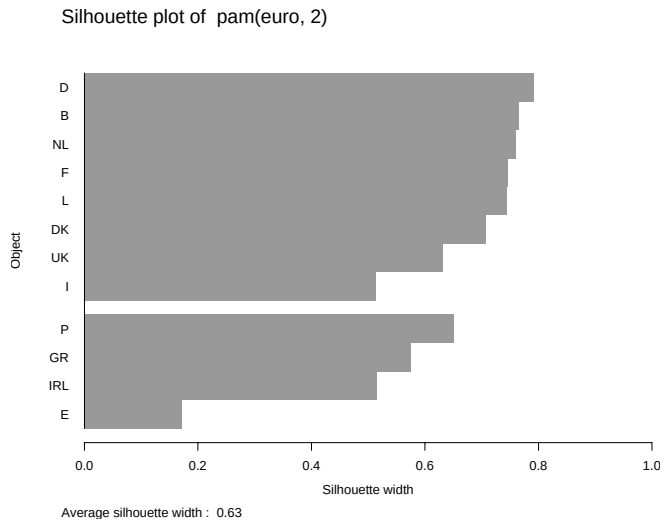

Figure 18.7: *Silhouette plot of* `pam(euro,2)`.

"Animals" Data: an Example of a Binary Data set

Six binary attributes are considered for twenty animals.

Table 18.5: *Animal attributes*

Abbreviation	Attribute
`war`	warm or cold blooded
`fly`	flying or non-flying
`var`	vertebrate or invertebrate
`end`	endangered or not
`gro`	lives in social groups, or not
`hai`	hairy or not hairy

This example illustrates the use of `mona`. The banner shows that `mona` classifies the animals according to the six attributes. In the first step, cold- and warm-blooded animals are put in separate clusters. The first cluster is then split into vertebrate and invertebrate animals, and the second cluster into flying and non-flying animals. Finally, after the fifth step, animals belonging to the same group have the same value for all six variables (on the banner, no bar is drawn between these animals).

If we wished to apply `agnes` or `diana` to this data set, we would have to compute the dissimilarities with `daisy`, because the variables are not numeric. The instruction is: `agnes(daisy(animals),diss=T)`. When we consider variable two (flying or not flying), and six (hairy or not hairy) as asymmetric binary, the call becomes:

```
agnes(daisy(animals,type=list( asymm = c(2,6))), diss=T).
```

The resulting clusterings will differ from the previous clustering since `agnes` and `diana` operate on the dissimilarities only, they do not use the individual variables any more. The function `mona` is probably more suitable for this example, where the animals have been classified nicely according to their attributes.

Table 18.6: *The animals and the three letter abbreviations used in the data.*

ant	caterpillar	frog	man
bee	duck	hermit crab	rabbit
cat	eagle	lion	salamander
chimpanzee	elephant	lizard	spider
cow	fly	lobster	whale

```
> animals
    war fly ver end gro hai
ant   1   1   1   1   2   1
bee   1   2   1   1   2   2
cat   2   1   2   1   1   2
cpl   1   1   1   1   1   2
chi   2   1   2   2   2   2
cow   2   1   2   1   2   2
duc   2   2   2   1   2   1
eag   2   2   2   2   1   1
ele   2   1   2   2   2   1
fly   1   2   1   1   1   1
fro   1   1   2   2  NA   1
her   1   1   2   1   2   1
lio   2   1   2  NA   2   2
liz   1   1   2   1   1   1
lob   1   1   1   1  NA   1
```

```
man   2   1   2   2   2   2
rab   2   1   2   1   2   2
sal   1   1   2   1  NA   1
spi   1   1   1  NA   1   2
wha   2   1   2   2   2   1
> mona(animals)
```

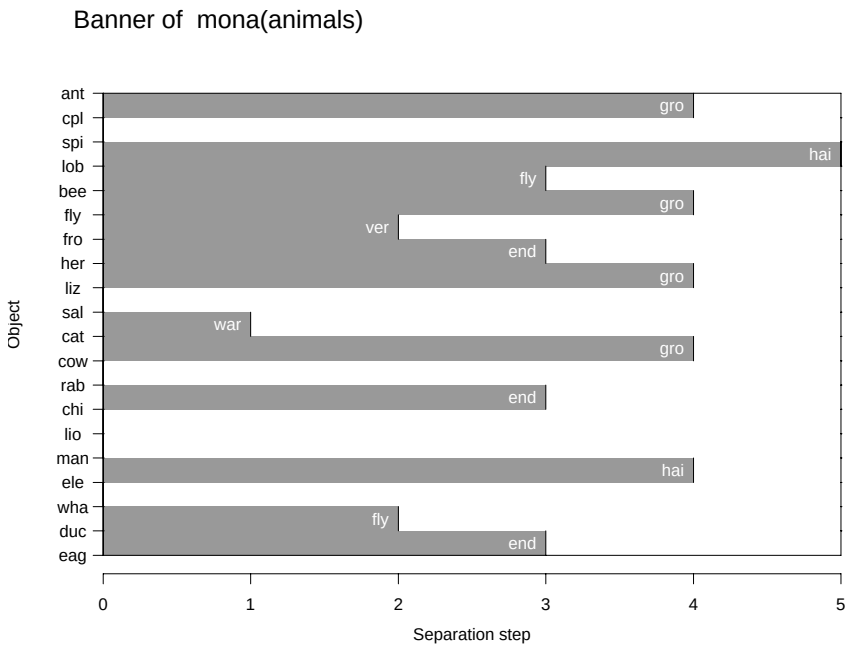

Figure 18.8: *Banner of* `mona(animals)`.

18.8 REFERENCES

Banfield, J.D. and Raftery, A.E. (1992). *Model-based Gaussian and non-Gaussian clustering*. Biometrics, **49**:803–822.

Friedman, H. P. and Rubin, J. (1967). *On some invariant criteria for grouping data*. Journal of the American Statistical Association, **62**:1159–1178.

Gordon, A.E. (1981). *Classification: Methods for the Exploratory Analysis of Multivariate Data*. Chapman and Hall, New York.

Hartigan, J.A. (1975). *Clustering Algorithms*. Wiley, New York.

Kaufman, L. and Rousseeuw, P.J. (1990). *Finding Groups in Data: An Introduction to Cluster Analysis.* Wiley, New York.

Murtagh, F. and Raftery, A.E. (1984). *Fitting straight lines to point patterns*. Pattern Recognition, **17**:479–483.

Murtagh, F. (1985). *Multidimensional Clustering Algorithms*. CompStat Lectures, 4. Physica-Verlag, Heidelberg.

Rousseeuw, P. J. (1986). A Visual display for hierarchical classification. In E. Diday, Y. Escoufier, L. Lebart, J. Pages, Y. Schektman, R. Tomassone (Eds.) *Data Analysis and Informatics*, vol. 4, North-Holland, Amsterdam, **7**43=48.

Rousseeuw, P. J. (1987). Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, **20**:53=65.

Scott, A.J. and Symons, M.J. (1971). *Clustering methods based on likelihood ratio criteria*. Biometrics, **27**:387–397.

Struyf, A., Hubert, M., and Rousseeuw, P. J. (1997). Integrating Robust Clustering Techniques in S-PLUS, *Computational Statistics and Data Analysis*, **26**:17=37.

Ward, J.H. (1963). *Hierarchical groupings to optimize an objective function*. Journal of the American Statistical Association, **58**:236–244.

HEXAGONAL BINNING

19

Hexbinning is used to graphically display spatial data, the classic example being earthquake epicenters and strengths.

HEXAGONAL BINNING

This chapter describes the use of the `hexbin` function to graphically display spatial data. The S+SPATIALSTATS module, available for both UNIX and Windows, provides a more extensive set of tools for analyzing spatial data in the form of geostatistical data, lattice data, and spatial point patterns.

19.1 THE APPEAL OF HEXAGONAL BINNING

Hexagonal binning is a data grouping or reduction method typically employed on large data sets to clarify spatial structure. It can be thought of as partitioning a scatter plot into larger units to reduce dimensionality, while maintaining a measure of data density. The groups or bins are used to make hexagon mosaic maps colored or sized according to density. Rectangular or square grids are often used in this context for image-processing applications, for example, in grayscale, contour, and perspective maps. However, hexagons are preferable for visual appeal and representational accuracy (Carr, Olsen, and White, 1992). Hexagonal binning can also be used to group geostatistical data into a lattice for use in spatial regression modeling.

The data frame `quakes.bay` contains the locations of earthquakes in the San Francisco Bay Area for 1962–1981. Hexagonal bins are maintained in an object of class `hexbin`. Use the function `hexbin` to create the `hexbin` object for the earthquake data as follows.

```
> quakes.bin <- hexbin(quakes.bay$longitude,
+ quakes.bay$latitude)
> summary(quakes.bin)

Call:
hexbin(x = quakes.bay$longitude, y = quakes.bay$latitude)
Total Grid Extent: 36 by 31
      cell            count            xcenter
Min.   :  17.0   Min.   :  1.000   Min.   :-123.3
1st Qu.: 239.0   1st Qu.:  1.000   1st Qu.:-122.0
Median : 419.0   Median :  3.000   Median :-121.6
Mean   : 467.9   Mean   :  7.505   Mean   :-121.5
3rd Qu.: 696.0   3rd Qu.:  5.000   3rd Qu.:-121.0
Max.   :1091.0   Max.   :144.000   Max.   :-119.8
```

```
     ycenter
Min.   :36.01
1st Qu.:36.51
Median :36.94
Mean   :37.06
3rd Qu.:37.59
Max.   :38.50
```

The `summary` function shows the four components of the `hexbin` object and their distributions. The hexagon identified by `cell` contains `count` observations, and has center of mass at (`xcenter`, `ycenter`). The default settings for `hexbin` partition the range of x values into approximately 30 equal-sided hexagonal bins. The most useful bin size depends on the number of observations, and is best chosen iteratively. Plot the hexagonal bins as follows.

```
> trellis.device(color=F)
> at.quakes <- c(0,10,20,30,40,50,150)
> plot(quakes.bin,border=T, col.regions=80:15, at=at.quakes)
```

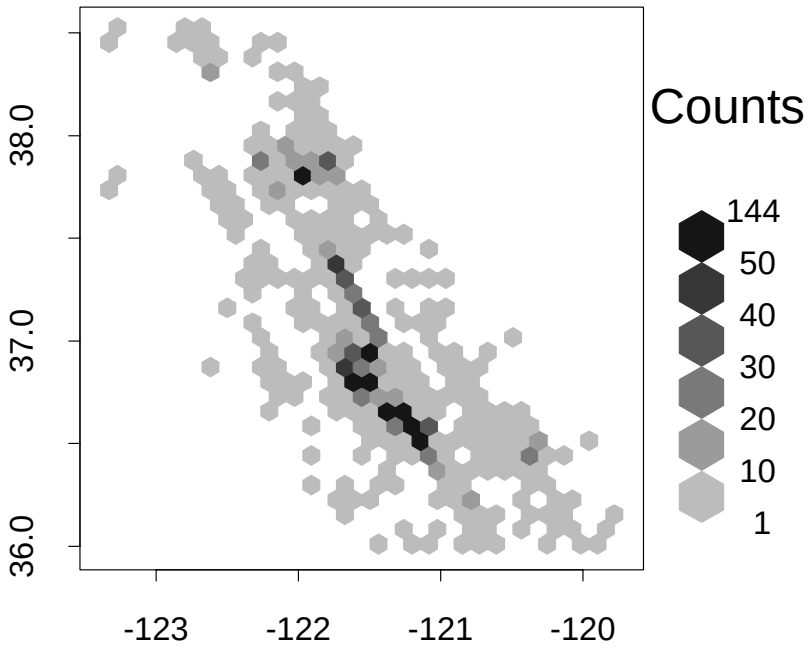

Figure 19.1: *The San Andreas Fault has a clear ridge of frequent earthquakes.*

The Trellis graphics device produces the best color and grayscale images for hexagonal binning. The default settings for `plot.hexbin` plot the hexagonal bins as a full tessellation, containing equally sized hexagons with color corresponding to grouped bin counts. By default, the groups are equal in range. Since the distribution of `quakes.bin$count` (shown by the `summary` output above) is skewed, we have chosen the groups formed by `at.quakes`. The plot in figure 19.1 shows the ridge of frequent earthquakes along the San Andreas Fault.

Hexagonal Bin Plot Styles

Besides the default grayscale style used here, there are four other plot styles available which plot the hexagons in varying sizes depending on cell density. Plot the earthquake `hexbin` object with differing sizes of hexagons as follows:

```
> plot(quakes.bin, style="centroids", cuts=6)
```

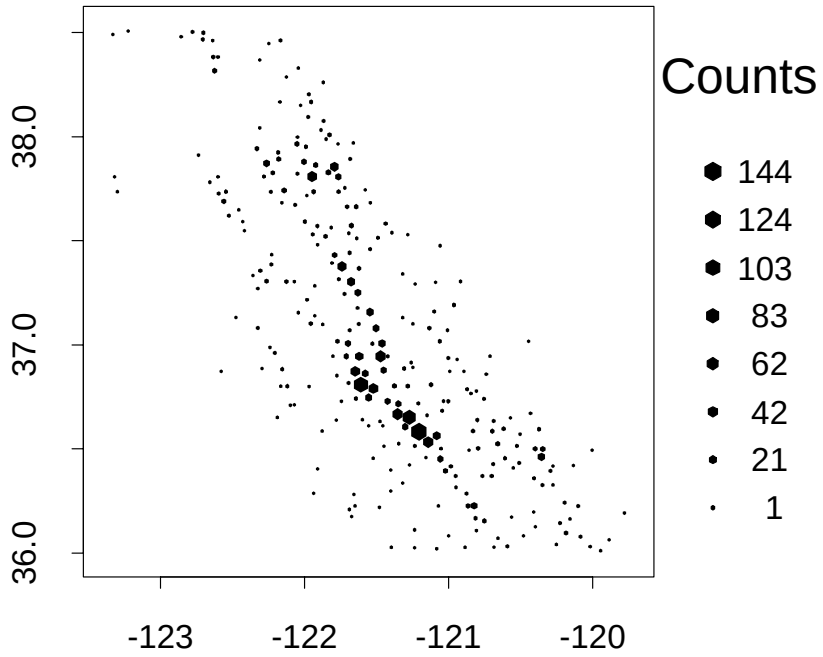

Figure 19.2: *As an alternative to using different grayscales in a hex plot, the hexagons can be drawn to a range of sizes. The range is determined by the* `cuts=` *argument.*

The `"centroids"` style shown in the figure scales the hexagon sizes by cell count, and plots them at the center of mass determined by `xcenter` and `ycenter`. The `cuts = 6` argument yields six different hexagon sizes. There

are two nested plot styles (`nested.lattice` and `nested.centroids`, not shown) which provide depth when plotted on a color screen.

Examining Individual Bins

There are several large bins in the plot which we may want to examine more closely. The generic `identify` function can be used to interactively identify points on a hexagonal bin plot. The two largest bins can be identified as follows.

```
> quake.par <- plot(quakes.bin, style="centroids", cuts=6)
> oldpar <- par(quake.par)
> identify(quakes.bin, use.pars = quake.par, offset=1)
[1] 114 79
> par(oldpar)
```

First it is necessary to save the graphical parameters used to plot the hexagonal bin. After entering the `identify` command, use the cross-hairs to locate the point of interest on the graphics screen, and click the left mouse button. The count in the closest cell will appear on the graphics screen. We have used the optional argument `offset` to make the count easier to read. When you have identified both points, click the center or right mouse button, while keeping your pointer within the graphics window. The index of the points you have identified will appear on your command line, as above. Then use the `par` function to reset the graphics parameters.

Directional Rays

The `rayplot` function can be used to display the magnitudes of a variable of interest at spatial locations using directional rays. For smaller data sets, these rays or other types of symbols can be plotted at each data location. However, when the number of sites is large, the magnitudes and trends are easier to visualize if the locations are first binned using `hexbin`. The following example uses the `ozone` data set:

1. Create a `hexbin` object for the `ozone` data, using eight bins in the *x* direction.

```
> ozone.bin <- hexbin(ozone.xy$x, ozone.xy$y, xbins=8)
```

2. Map each (*x*,*y*) pair in the original data to a hexagonal cell using the function `xy2cell`.

```
> ozone.cells <- xy2cell(ozone.xy$x, ozone.xy$y, xbins=8)
```

3. Use the function `tapply` to calculate the median for each cell, and use these values as angles for the rayplot.

```
> ozone.angle <- tapply(ozone.median, ozone.cells, median)
> library(maps)
```

```
Warning messages:
   The functions and datasets in library section maps are
   not supported by MathSoft. in: library(maps)
> map(region=c("new york","new jersey","conn","mass"),lty=2)
> rayplot(ozone.bin$xcenter,ozone.bin$ycenter,ozone.angle)
```

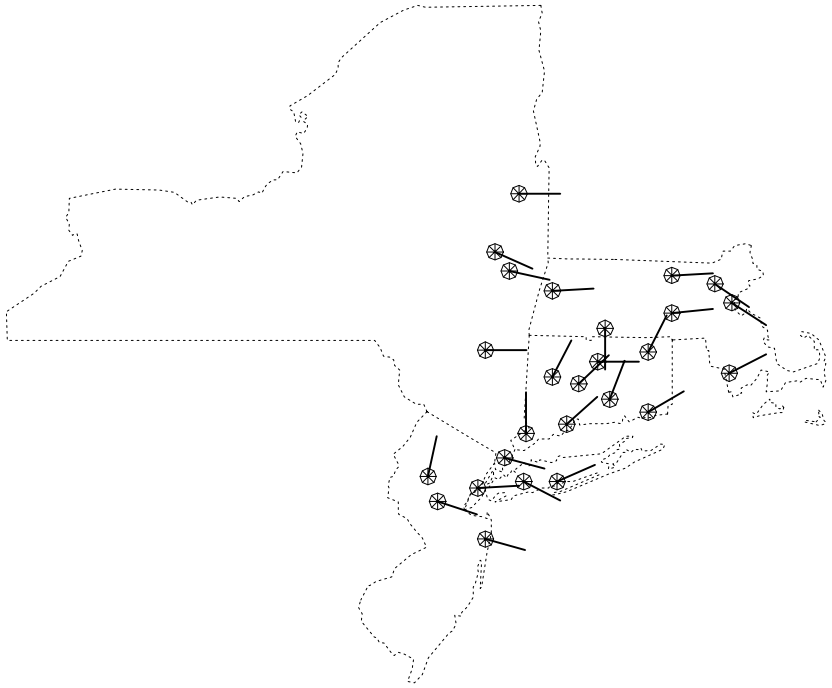

Figure 19.3: *Rayplots add direction as well as density. This plot shows median ozone emissions.*

The plot shows the median ozone emissions for the group of sites within each hexagonal bin. The ray is plotted at the center of each bin, and the medians are scaled so the rays follow an arc from $-\pi/2$ (lowest median) to $\pi/2$ (highest median). It appears that the highest emissions for the time period covered are in Connecticut. Additional attributes can be used with `rayplot` to add confidence intervals and a second variable to the plot. Also, the lengths and widths of the rays and the size of the base octagon can be changed. See the on-line help file for more information on `rayplot`.

19.2 REFERENCES

Carr, D. B., Olsen, A. T., and White, D. (1992). *Hexagon mosaic maps for display of univariate and bivariate geographical data*. Cartography and Geographical Information Systems, 19:228–236.

CREATING AND VIEWING TIME SERIES

20

Time series arise where the exact time of the data is critical in the required analysis.

CREATING AND VIEWING TIME SERIES

Time series arise in situations where the timing of the data acquisition is an important feature of the values and their analysis. For example, weekly or monthly measurements of sunspot activity can be used to study cycles in sunspot activity. Old records from the Hudson's Bay Company on annual trappings of the Canadian lynx can be used to study yearly fluctuations in population numbers of the lynx. Yearly measures of the U.S. corn crop yield can be used to study factors that might influence corn production.

This chapter describes how to create, manipulate, and view time series in S-PLUS. Section 20.1, Creating and Modifying Time Series, describes how to create time series in S-PLUS how to combine two series, and various ways of subsetting time series. Section 20.2, Visualizing Correlation in Time Series Data, explores some methods for plotting and visually analyzing time series.

20.1 CREATING AND MODIFYING TIME SERIES

A *time series* is a collection of observations made sequentially in time. If the observations are multidimensional, then we have a *multivariate time series*. Three classes of time series are recognized in S-PLUS:

- Regularly spaced time series, which are series sampled at equal intervals, make up the class `"rts"`
- Calendar time series, in which the regularly spaced observations are associated with a calendar date, make up the class `"cts"`
- Irregularly spaced time series, in which the observations may be sampled at irregular intervals and which may have calendar or non-calendar time domains, make up the class `"its"`.

A time series can have the form of a vector, a factor, a matrix, or a data frame. A univariate time series has the form of a vector or factor; a multivariate time series has the form of a matrix or data frame. A univariate S-PLUS time series object is just a special case of the general multivariate S-PLUS time series object. The columns, or channels, of an S-PLUS multivariate time series represent univariate time series with simultaneous observations across the rows.

Creating Regular Time Series

A regular time series (`rts`) is a sequence of observations obtained at regular intervals. A regular time series is characterized by four time parameters which together give a summary of the sequence of observation times:

1. the time of the first observation,
2. the interval between observation times, Δt,

3. the sampling rate, which is the reciprocal of the interval Δt, and
4. the time of the last observation.

Use the function `rts` to create a regular time series data object from your time series data. Use the arguments to `rts` (`start` `deltat` (Δt), `frequency`, and `end`) to specify the time parameters. The data generally supply the length of the time sequence. You can, however create an empty time series, with `NA` for all the values, by supplying both a beginning and an ending time, and either `frequency` or `deltat`.

```
> empty.rts <- rts(start=0, end=8.8, deltat=0.2)
> empty.rts
    1  2  3  4  5
0: NA NA NA NA NA
1: NA NA NA NA NA
2: NA NA NA NA NA
3: NA NA NA NA NA
4: NA NA NA NA NA
5: NA NA NA NA NA
6: NA NA NA NA NA
7: NA NA NA NA NA
8: NA NA NA NA NA
 start deltat frequency
     0    0.2         5
```

Since `frequency` and `deltat` are reciprocals, you can define either one and the other is determined automatically. For instance, suppose you want to make a time series of the outcomes of presidential elections, which are held every four years. In this case it is easier to define `deltat`. The matrix `votes.repub` shows the percent of votes in each state given to the Republican candidate in presidential elections starting in the year 1856. The rows of this matrix are the states, so you transpose the matrix to make each column a univariate time series.

```
> votes.rts <- rts(t(votes.repub), start = 1856, deltat = 4,
+ units = "years")
```

When the observation intervals occur in regular cycles it is often easier to define the `frequency`. For example, `frequency=12, units="months"` or `frequency=1000, units="kHz"`. Note that `"units"` always refers to the interval between observations (`deltat`), never to the larger period `deltat` $\times$ `frequency`.

You can define the `start` argument, the `end` argument, or both. If you define both, they must agree with the length of the data. The `end` must be

later than the `start`. The `start` argument may be a single numeric value giving the starting time (e.g., for `votes.rts` the starting year was `start=1856`), or a pair of values giving the base time and an integer offset (e.g., `start=c(1962,2)`). The offset gives the position in the cycle of the first observation. Thus to indicate a starting time of the second quarter of 1962, use the `start` argument as follows:

```
> freeny.rts <- rts(freeny.y, start=c(1962, 2), freq=4,
+ units="quarters")
> freeny.rts
           1Q      2Q      3Q      4Q
1962:         8.79236 8.79137 8.81486
1963: 8.81301 8.90751 8.93673 8.96161
1964: 8.96044 9.00868 9.03049 9.06906
1965: 9.05871 9.10698 9.12685 9.17096
1966: 9.18665 9.23823 9.26487 9.28436
1967: 9.31378 9.35025 9.35835 9.39767
1968: 9.42150 9.44223 9.48721 9.52374
1969: 9.53980 9.58123 9.60048 9.64496
1970: 9.64390 9.69405 9.69958 9.68683
1971: 9.71774 9.74924 9.77536 9.79424
   start deltat frequency
 1962.25   0.25         4
Time units :  quarters
```

The `end` argument is used similarly.

You can name the component series (columns) of a time series directly with the `names` argument to `rts`, or allow the creating function to use the dimnames of the matrix or data frame which contains the data. If neither of these are given, the series are named `"Series 1"`, `"Series 2"`, etc. The rows are named with the times that correspond to the observations.

Suppose you want to create a bivariate white noise series of length 100 and sampling interval $\Delta t = 1/5$, starting at 1. Since 1 is the default starting time for time series, you don't need to give the starting time explicitly in this case:

```
> whitenoise2 <- rts(matrix(rnorm(200), ncol=2), deltat=1/5)
> whitenoise2
        Series 1   Series 2
 1.0 -0.40333165  0.3468278
 1.2  1.32106086 -0.7209995
 1.4 -1.21063699  1.6346167
 1.6 -0.06814786  3.2141895
 1.8 -0.65618203 -1.9486379
```

```
 2.0 -0.20831037 -0.2666580
 2.2  0.03356625 -0.7492557
 2.4 -1.92188396 -1.1880001
 2.6  1.00097830  0.9222979
 2.8  0.96451061 -0.2713598
 . . .
 start deltat frequency
     1    0.2         5
```

To view information about a time series without printing the entire object, use the `summary` function. This function gives the type of time series (regular, calendar, or irregular), the number of component series (channels) and the number of observations in each, a vector summary of each channel (range, quartiles, and median), and the time parameters `start`, `deltat`, `frequency`, and `units`:

```
> rain.rts <- rts(cbind(rain.nyc1, rain.nyc2),
+ start=1869, names=c("nyc1", "nyc2") )
> summary(rain.rts)

Regular Time Series:
Observations: 89  on 2 channels

       nyc1            nyc2
 Min.   :32.70   Min.   :32.6
 1st Qu.:37.80   1st Qu.:38.8
 Median :40.80   Median :42.1
 Mean   :42.31   Mean   :42.9
 3rd Qu.:46.00   3rd Qu.:46.7
 Max.   :56.10   Max.   :58.7

Time Parameters :
 start deltat frequency
 1869     1      1
```

The functions `start` and `end` return the starting and ending times of the series, respectively.

```
> start(rain.rts)
[1] 1869
> end(rain.rts)
[1] 1957
```

Manipulating Dates

Dates in S-PLUS can be represented and manipulated in very natural ways. Use the `dates` function to create a `dates` object from a character string or a vector of character strings.

```
> holiday93 <- dates(c("01/01/93", "01/18/93",
+  "02/15/93", "05/31/93", "07/04/93", "09/06/93",
+  "10/11/93", "11/11/93", "11/25/93", "12/25/93"))
> holiday93
[1] 01/01/93 01/18/93 02/15/93 05/31/93 07/04/93
[6] 09/06/93 10/11/93 11/11/93 11/25/93 12/25/93
```

You can specify dates in a variety of formats; use the `format` argument to specify the format you are using for the input, and the `out.format` argument for the format of the output. The strings that control the way a date object is interpreted and printed consist of the letters `"y"`, `"m"`, and `"d"` in any order, with or without a separator.

```
> election <- dates("931102", format="ymd",
+ out.format="month day year")
> election
[1] November 02 1993
> attr(election,"format")
[1] "month day year"
```

The formats `"d-m-y"`, `"m/d/y"`, and `"ymd"` cause `election` to be printed as `02-11-93`, `11/02/93`, and `931102`, respectively. Spelling out `"month"` and `"year"` causes them to print out fully, as in the example above, while abbreviating month as `"mon"` causes the month to print as a three-letter abbreviation.

```
> dates(election, out.format="day mon y")
[1] 02 Nov 1993
```

The default input and output format for `"dates"` is `"m/d/y"`.

Sequences and Dates

You can create a sequence of dates with the `seq` function much the same way you create a sequence of integers by using a date as the first (`from`) argument. The other necessary arguments are the interval between the elements, `by`, and either an ending date, `to`, or an integer length, `length`. You can specify `by` as one of `"days"`, `"weeks"`, `"months"`, `"quarters"` or `"years"`, or as an integer number of days.

```
> start.dates <- seq(dates("09/27/93"), length=5,
+ by ="weeks")
> start.dates
[1] 09/27/93 10/04/93 10/11/93 10/18/93 10/25/93
```

```
> seq(dates("09/27/93"), length = 5, by = 7 )
[1] 09/27/93 10/04/93 10/11/93 10/18/93 10/25/93
```

You must supply a starting date and an increment (`by`). You may supply an ending date (`to`) instead of a desired length.

```
> seq(dates("09/27/93"), dates("10/30/93"), by="weeks")
[1] 09/27/93 10/04/93 10/11/93 10/18/93 10/25/93
```

Unlike the case when using the `seq` function with numbers, you cannot give `seq` a starting and ending date and ask for a vector of a specific length.

Operations on Dates

You can perform certain types of arithmetic operations on dates. The operations that work on dates are addition or subtraction of a scalar number of days, subtraction of one date from another to get the number of days between them, and logical comparison of dates:

```
> end.dates <- start.dates + 10
> preview.dates <- start.dates - 30
> max(start.dates) - min(start.dates)
Time in days:
[1] 28
> max(start.dates) > min(end.dates)
[1] T
```

You cannot do arithmetic calculations with dates that make no sense—for example, you cannot multiply or divide a date by a scalar, nor can you add two dates.

All of the usual tools for examining and manipulating vectors are available for use on date objects, so, for example to obtain a vector of differences between elements in a dates vector, use `diff`:

```
> diff(holiday93)
Time in days:
[1]  17  28 105  34  64  35  31  14  30
```

Julian Dates

Dates in S-PLUS are represented internally as Julian dates, that is, the number of days from an arbitrary day of origin. The default origin date in S-PLUS is January 1, 1960. The `dates` function interprets integers as Julian dates, and so does any function that is expecting a date as an argument. You can specify a different origin when you create a date. For example, to create a vector of five random days in August 1993 use the `origin` argument as follows:

```
> random.days <- dates(sample(0:30, size=5),
+ origin=c(8, 1, 1993))
> random.days
[1] 08/12/93 08/10/93 08/22/93 08/19/93 08/07/93
```

View the origin of a date with the `origin` function:

```
> origin(start.dates)
 month day year
     1   1 1960
> origin(random.days)
[1]    8    1 1993
```

You can convert a `dates` object to an ordinary integer with `as.integer`.

Calendar Time Series

A calendar time series is a sequence of observations taken at regular intervals, in which each observation is associated with a calendar date. The time parameters that define a calendar time series are the start date, the units of the observation interval, and a multiplier which indicates how many units of time in each interval.

To create a calendar time series (`cts` object) use the `cts` function. You can present the `start` argument with a date created by the `dates` function, a string of the appropriate format, or a Julian date (an integer).

The `units` argument of `cts`, which defaults to `"years"`, must be one of `"days"`, `"weeks"`, `"months"`, `"quarters"`, or `"years"`. Each of these has a default sampling frequency, as is shown in table 20.1. Suppose you have

Table 20.1: *Units and frequencies in calendar time series.*

Units	Frequency
"days"	365
"weeks"	52
"months"	12
"quarters"	4
"years"	1

monthly temperature records for three different weather stations for the years 1985–1987, stored as S-PLUS data sets “`temp1`”, “`temp2`”, and “`temp3`”. To create a 3-dimensional time series with these records as the component series, and name each series, use the commands:

```
> temp1 <- scan(file="Aberdeen.temp")
> temp2 <- scan(file="Forks.temp")
> temp3 <- scan(file="Quinault.temp")
```

```
> temp.cts <- cts(cbind(temp1,temp2,temp3),
+       start="01/01/85", units="months",
+       names=c("Aberdeen","Forks","Quinault"))
```

Sometimes you have data that are sampled at regular intervals that are not one of the five shown in table 20.1, for instance bi-weekly or every ten days. The multiplier, set by the argument `k.units`, allows sampling intervals that are whole numbers of units apart. Thus, to specify observations taken every ten days, set the `units` to `"days"` and `k.units` to 10; to specify bi-weekly data, set the `units` to `"weeks"` and `k.units` to 2; to specify semi-annual data, set the `units` to `"months"` and `k.units` to 6, or set `units` to `"quarters"` and `k.units` to 2. Other intervals can be defined similarly.

The sampling frequency is determined by the units of the time interval and the multiplier `k.units`. It does not need to be set directly. However, to create a sampling cycle within a time series, you can set the `frequency` to the length of the desired cycle. In the example below the starting time is the full moon on October 30, and the weekly observations are whether the moon is approximately full, half, or new.

```
>  moon <- cts(rep(c(1, 1/2, 0, 1/2), 4), start="10/30/93",
+ units="weeks", freq=4)
> moon
    week 1 week 2 week 3 week 4
  :                       1.0
93: 0.5    0.0    0.5    1.0
93: 0.5    0.0    0.5    1.0
93: 0.5    0.0    0.5    1.0
94: 0.5    0.0    0.5
    start units k.units frequency
 10/30/93 weeks 1       4
> end(moon)
 02/12/94
```

See the section Manipulating Time Series and the functions `time` and `cycle` for more about the use of frequency and sampling cycles.

Irregular Time Series

An irregular time series is a set of observations taken over time at unequal intervals. Each observation of an irregular time series (`its` object) is associated with an observation time. To create an irregular time series, use the `its` function and supply as the `times` argument a vector containing the times of each successive observation. This vector may be either numeric or of class `"dates"`. The observation times must be unique and in ascending order. You can supply the time units with the argument `units`, and the names of the channels (columns) with `names`, in the case of a multivariate time series.

```
> votes.its <- its(t(votes.repub), times = votes.year,
+ names = state.abb)
> votes.its[,1:8]      #print only the first 8 states
        AL    AK    AZ    AR    CA    CO    CT    DE
1856    NA    NA    NA    NA 18.77    NA 53.18  2.11
1860    NA    NA    NA    NA 32.96    NA 53.86 23.71
1864    NA    NA    NA    NA 58.63    NA 51.38 48.20
1868 51.44    NA    NA 53.73 50.24    NA 51.54 40.98
1872 53.19    NA    NA 52.17 56.38    NA 52.25 50.99
1876 40.02    NA    NA 39.88 50.88    NA 48.34 44.55
...
> faithful.its <- its(geyser$duration,
+ cumsum(geyser$waiting), units="minutes")
> number.of.deaths
 [1] 156  89  40  71  84  47  84  57 118  88
> vehicular <- its(death, holiday93)
> ts.plot(vehicular)
```

You can retrieve the observation times of an irregular time series with the `time` function; the result is a vector, not a time series.

```
> time(faithful.its)[1:10]
 [1]  80 151 208 288 363 440 500 586 663 719
```

You can plot, subset, and summarize irregular time series in the same way as regular or calendar time series.

```
> summary(faithful.its)
Irregular Time Series:
Observations: 299

   Min. 1st Qu. Median  Mean 3rd Qu. Max.
 0.8333       2      4 3.461   4.383 5.45

Time Parameters :
 start   end
 80    21622
```

There are as yet no methods in S-PLUS for analyzing irregular data sets.

Updating Old Time Series Objects

Time series created in S-PLUS versions 3.1 and earlier were classless objects with a `"tsp"` attribute. Since the `"tsp"` attribute supplies values for `start` and `frequency`, old `ts` objects can be easily coerced to `rts` objects with `as.rts`:

```
> sunspots.rts <- as.rts(sunspots)
> tspar(sunspots.rts)
```

```
  start     deltat frequency
   1749 0.08333333        12
```

All the time series functions that were available for S-PLUS versions 3.1 and before still accept `"ts"` objects, but newer ones may not, and `ts` will eventually be deprecated.

Binding Time Series Together

To bind two or more time series together into a single multivariate time series, use `ts.intersect` and `ts.union`. For example, if you have a weekly and a bi-weekly time series with different starting and ending times, you can use `ts.intersect` to create a matrix of two bi-weekly series. The starting time of the new series is the later of the starting times of the input series, and the ending time of the new series is the earlier of the ending times of the input series. The following example shows how to create a bivariate irregular time series from two subscripted time series.

```
> rain.low <- rts(corn.rain,
+ start=1890)[corn.rain < mean(corn.rain)]
> yield.low <- rts(corn.yield,
+ start=1890)[corn.yield < mean(corn.yield)]
> ts.intersect(rain.low, yield.low)
```

You can use `ts.union` to create a multivariate series retaining all the data of the component series. The starting time of the new series is the earlier of the starting times of the input series, and the ending time of the new series is the later of the ending times of the input series. `NA`'s fill the places of the missing data.

```
> lynx.lag <- ts.union(lynx.rts, lag(lynx.rts, k=10))
> ts.plot(lynx.lag)
```

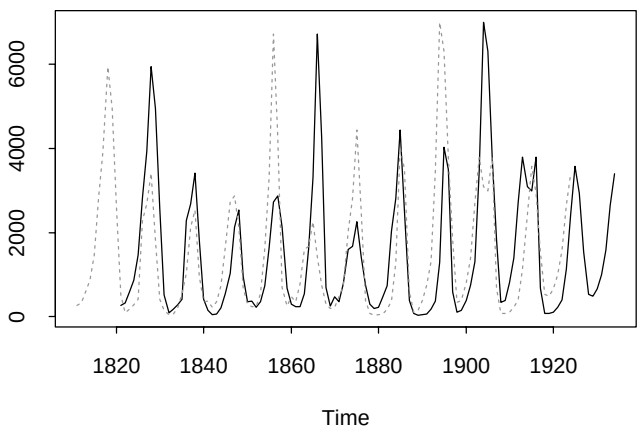

Figure 20.1: *Time series plot of lynx data and lagged lynx data.*

Manipulating Time Series

To get the time of each observation of a time series, use the function `time`. In this example the number of lynx trappings is plotted versus the year, and then specific years are identified interactively on the plot.

```
> lynx.rts <- as.rts(lynx)
> lynxtime <- time(lynx.rts)
> ts.plot(lynx.rts)
> # interactively identify points on the plot
> identify(lynxtime, lynx.rts, lynxtime)
```

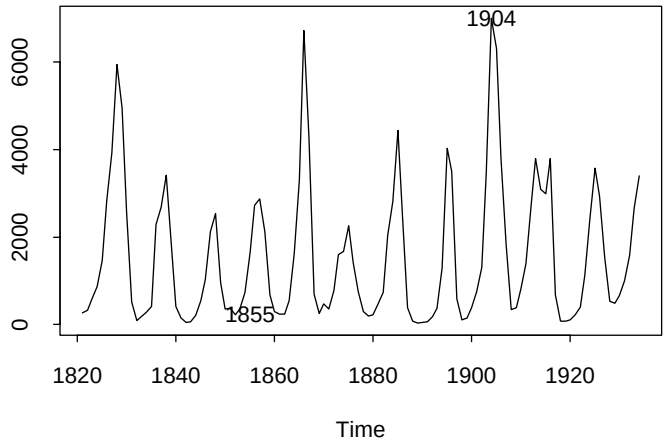

Figure 20.2: *Time series plot of lynx data with high and low points identified.*

Whenever the `frequency` of a time series is greater than one, there is an implied sampling cycle of length `frequency`. The layout of a univariate series such as `freeny.rts` shows this clearly—all the observations occurring at the same point in the cycle are in columns labeled with their position in the cycle while the sampling period increases by one at each row.

```
> freeny.rts
           1Q      2Q      3Q      4Q
1962:         8.79236 8.79137 8.81486
1963: 8.81301 8.90751 8.93673 8.96161
1964: 8.96044 9.00868 9.03049 9.06906
1965: 9.05871 9.10698 9.12685 9.17096
1966: 9.18665 9.23823 9.26487 9.28436
1967: 9.31378 9.35025 9.35835 9.39767
1968: 9.42150 9.44223 9.48721 9.52374
1969: 9.53980 9.58123 9.60048 9.64496
```

```
1970: 9.64390 9.69405 9.69958 9.68683
1971: 9.71774 9.74924 9.77536 9.79424

   start deltat frequency
 1962.25   0.25         4
 Time units :  quarters
```

To obtain a time series giving the position of each observation in the sampling cycle, use the `cycle` function.

```
> cycle(freeny.rts)
      1Q 2Q 3Q 4Q
1962:     2  3  4
1963:  1  2  3  4
       .  .  .
```

You can use the result of `cycle` to get subsets of the time series. For example, to get all the fourth quarter revenue from `freeny.rts`, use the results of `cycle` to subset the time series:

```
>  freeny.rts[cycle(freeny.rts)==4]
1961.75:         8.81486 8.96161 9.06906
1965.75: 9.17096 9.28436 9.39767 9.52374
1969.75: 9.64496 9.68683 9.79424
   start deltat frequency
 1962.75      1         1
```

As you can see from the example above, when you subscript a univariate regular time series using the `cycle` function, you get another regular time series. In fact, subscripting a univariate time series or the rows (time dimension) of multivariate time series yields a time series of the same type as long as the observations in the resulting time series are still at equal intervals. For instance, in the following example the monthly housing starts of the data set `hstart` are sampled quarterly:

```
> hstart.cts <- cts(hstart, start="01/31/66",units="months")
> qtrs <- rep(c(F,F,T), length=length(hstart.cts))
> hstart.qtrs <- hstart.cts[qtrs]
> hstart.qtrs
            1     2     3     4
Mar 66: 122.4 123.5  91.9  62.3
Mar 67:  92.9 131.6 125.8  83.1
Mar 68: 128.6 142.5 139.5  99.3
Mar 69: 135.6 150.5 132.9  85.3
Mar 70: 117.8 141.9 133.8 124.1
Mar 71: 169.3 196.8 175.6 155.3
```

```
Mar 72: 205.8 226.2 204.4 152.7
Mar 73: 201.1 203.4 148.9  90.6
Mar 74: 127.2 149.5  99.6  54.9
    start  units k.units frequency
 03/31/66 months 3       4
```

and in the next the monthly sunspot data of `sunspots.rts` is sampled at two year intervals:

```
> twoyear<- seq(from=13,  to=end(sunspots.rts), by=24)
> ts.plot(sunspots.rts[twoyear], sunspots.rts[twoyear -6],
+ sunspots.rts[twoyear + 12])
```

Whenever the observations of the resulting time series are *not* at equal intervals, it is returned as an irregular time series (class `"its"`). This often happens when you subscript on the values of the observations, as in the following:

```
> hi <- rain.rts[,1] > 46 & rain.rts[,2] > 46
> rain.hi <- rain.rts[hi,] #46 = 3rd Quartile for nyc1
> rain.hi
     nyc1 nyc2
1871 49.2 48.8
1884 49.7 55.3
1888 51.0 53.0
1889 54.4 58.7
1893 46.6 53.0
1901 47.0 47.1
1902 50.3 47.1
1903 55.5 48.6
1919 50.8 48.4
1920 53.2 48.8
1926 47.8 49.7
1927 56.1 49.9
1933 53.5 49.7
1936 49.8 46.3
1937 53.0 48.1
1938 48.5 46.5
1942 48.5 49.6
1948 46.9 54.2
 start      end
 1871  ... 1948
```

Of course, when you subscript the columns of a multivariate time series, the result is a time series with the same class and time parameters as the original,

as demonstrated below:

```
> freeny2.rts <- rts(freeny.x, start=c(1962, 2), freq=4,
+ units="quarters")
> price.rts <- freeny2.rts[ ,2:3]
> price.rts
        price index income level
1962.25     4.70997      5.82110
1962.50     4.70217      5.82558
1962.75     4.68944      5.83112
1963.00     4.68558      5.84046
 . . .

1971.00     4.30552      6.18231
1971.25     4.29627      6.18768
1971.50     4.27839      6.19377
1971.75     4.27789      6.20030
   start deltat frequency
 1962.25   0.25         4
 Time units :  quarters
```

To obtain a segment of a time series with only portion of the time domain, use the `window` function. Suppose you want to look more closely at two shorter segments of `sunspots.rts`, one from a time with relatively few sunspots, and one from a time with relatively many.

```
> ts.plot(sunspots.rts)
> suntime<- time(sunspots.rts)
> identify(suntime, sunspots.rts, suntime)
```

By selecting various points on the plot with the mouse, you determine that the years 1790 to 1840 were a relatively quiet time for sunspots, while between 1925 and 1975 there were high peaks of sunspot activity. You can then use `window` to extract the intervals of interest:

```
> quiet.rts <- window(sunspots.rts,start = 1790, end = 1840)
> noisy.rts <- window(sunspots.rts,start = 1925, end = 1975)
```

A *lagged* time series is a new time series with the same data as a given time series shifted in time by a specified amount. You can create a lagged series in S-PLUS with the function `lag`. A positive lag shifts the series earlier in time; a negative lag shifts it later.

```
> hstart.rts <- as.rts(hstart)
> lag.yr <- lag(hstart.rts, 12)
> adv.yr <- lag(hstart.rts, -12)
```

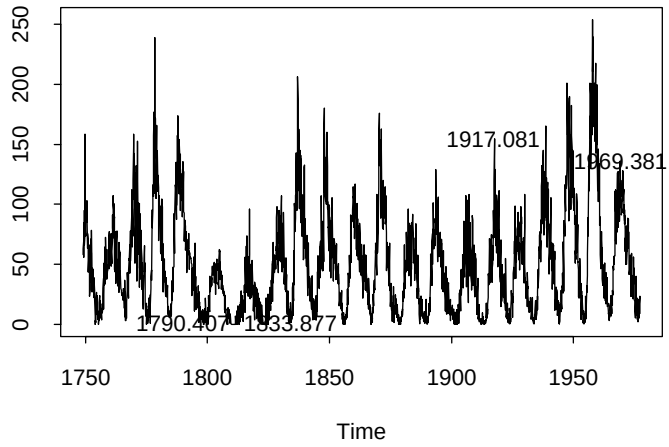

Figure 20.3: *Ranges in sunspot data identified interactively.*

The three commands above create three time series with the same data but different starting dates:

```
> c(start(lag.yr), start(hstart.rts), start(adv.yr))
[1] 1965 1966 1967
```

You can look at them all side by side. In the example below you can see that the data in `lag.yr` at 1965.883 and 1965.917 (November and December of 1965) are the same as the data in `hstart.rts` as 1966.883 and 1966.917, twelve months later.

```
> hstart.lag <- ts.union(lag.yr, hstart.rts, adv.yr)
> window(hstart.lag, 1965.8, 1967)
         lag.yr hstart.rts adv.yr
1965.833   75.1         NA     NA
1965.917   62.3         NA     NA
1966.000   61.7       81.9     NA
1966.083   63.2       79.0     NA
1966.167   92.9      122.4     NA
1966.250  115.9      143.0     NA
1966.333  134.2      133.9     NA
1966.417  131.6      123.5     NA
1966.500  126.1      100.0     NA
1966.583  130.2      103.7     NA
1966.667  125.8       91.9     NA
1966.750  137.0       79.1     NA
1966.833  120.2       75.1     NA
```

```
1966.917   83.1       62.3     NA
1967.000   82.7       61.7   81.9
    start     deltat frequency
 1965.833 0.08333333        12
time units :  months
> ts.plot(hstart.lag, lty=c(2,1,3) )
```

To find the difference between numeric observations at fixed intervals, use the `diff` function. The `lag` argument gives the numbers of intervals apart to take the differences; the default is 1. The `diff` function creates a time series whose channels are `lag` shorter than the original, with a starting time `lag` intervals later than the starting time of the original. A `differences` argument greater than one repeats the process, so that `diff(x,1,4)` is the same as `diff(diff(diff(diff(x))))` The following example shows how to take the yearly difference in the values of `freeny.rts` by quarter:

```
> diff(freeny.rts, lag=4)
            1       2       3       4
1963:         0.11515 0.14536 0.14675
1964: 0.14743 0.10117 0.09376 0.10745
1965: 0.09827 0.09830 0.09636 0.10190
1966: 0.12794 0.13125 0.13802 0.11340
1967: 0.12713 0.11202 0.09348 0.11331
1968: 0.10772 0.09198 0.12886 0.12607
1969: 0.11830 0.13900 0.11327 0.12122
1970: 0.10410 0.11282 0.09910 0.04187
1971: 0.07384 0.05519 0.07578 0.10741
   start deltat frequency
 1963.25   0.25         4
 Time units :  quarters
```

The `diff` function also works for vectors and matrices. See the section Integrals, Differences, and Derivatives in chapter 28 for more uses of the `diff` function.

20.2 VISUALIZING CORRELATION IN TIME SERIES DATA

If data are collected over time, there may be correlation between successive observations. You can visually explore whether or not your data is *serially correlated* by using S-PLUS functions to make three kinds of plots:

- *simple time series plots*, which you have already seen in section entitled “Getting Started with Simple Plots” in the S-PLUS User’s Guide.

- *lagged scatter plots*, which are scatter plots of pairs of values (y_t, y_{t+m}) of a time series separated by a *lag* of one or more time units.
- *autocorrelation function plots*, which provide an estimate of the correlation between observations separated by a lag of zero, one, or more time units.

To illustrate the use of these functions, we use the function `rnorm` to create *uncorrelated* normal random numbers, and from these numbers create a *correlated* series `x.cor`:

```
> r.norm <- rnorm(100)
> x.cor <- r.norm[1:98] + r.norm[2:99] + r.norm[3:100]
```

The series `x.cor` is serially correlated at `lags` 1 and 2; that is, `x.cor[i]` is correlated with `x.cor[i+1]` and `x.cor[i+2]`. But `x.cor` is serially uncorrelated at lags greater than 2; that is, `x.cor[i]` and `x.cor[i+k]` are uncorrelated for $k>2$.

Basic Time Series Plots

The basic time series plot shows each observation plotted against its observation time. For example, our time series `x.cor` can be plotted as follows:

```
> ts.plot(x.cor,type="b",pch=16)
```

This expression yields the plot of figure 20.4.

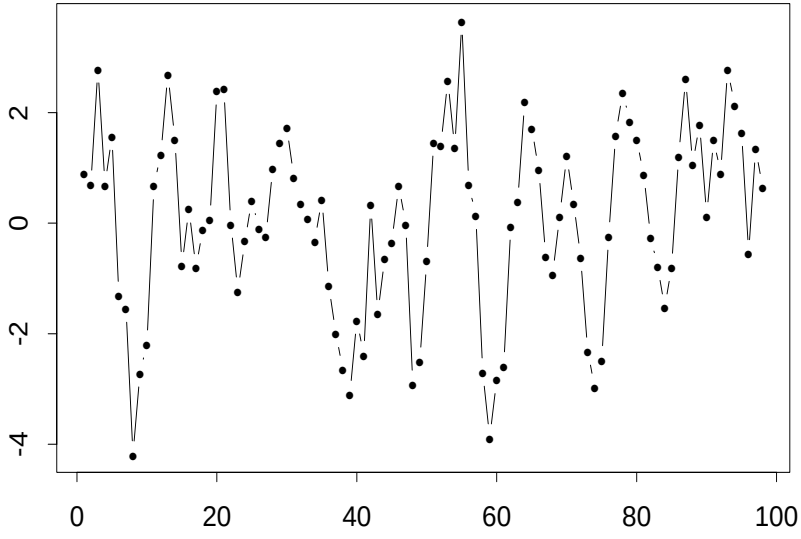

Figure 20.4: *Time series plot for a correlated series.*

The values of successive observations tend to be close together, so you suspect some serial correlation. You can see this more clearly with `lag.plot` and `acf`, as described in the following sections.

Lagged Scatter Plots

The lagged scatter plots in figure 20.5 consist of scatter plots of pairs of values (y_t, y_{t+m}) of a time series separated by m time units for $m = 1, 2, ..., M$. The figure is generated with the following expression:

```
> lag.plot(x.cor,lags=4,layout=c(2,2))
```

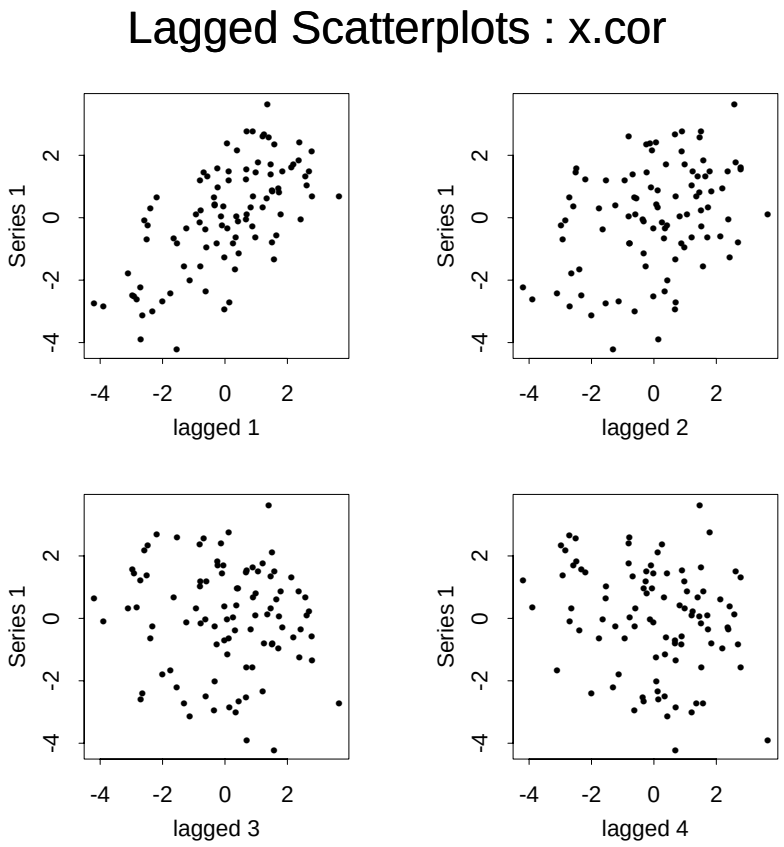

Figure 20.5: *Lagged scatter plots for a correlated series.*

The maximum lag M is specified by the `lags=` argument to `lag.plot`. For the above example, the choice `lags=4` results in M =4, and so there are four plots. The argument `layout=` specifies the way the M lagged scatter plots are arranged in a single figure, just as you use the function `par` to specify multiple figure layout.

A circular shape for a lagged scatter plot at a specific lag *m* indicates that there is little correlation at that lag. On the other hand, an elliptical shape for a lag *m* scatter plot in the 45 degree direction indicates positive correlation at lag *m*. An elliptical shape in the 135 degree direction indicates negative correlation. In the above example using `x.cor`, the lag 1 plot shows clear evidence of positive correlation, and the lag 2 plot shows some indication of positive correlation.

Autocorrelation Plots

An autocorrelation function (acf) plot provides an estimate of the correlation between observations separated by a lag of *m* time units, for *m* = 0, 1, 2, ..., *M*. Use the following expressions to obtain the plots shown in figure 20.6:

```
> ts.plot(x.cor)
> acf(x.cor)
```

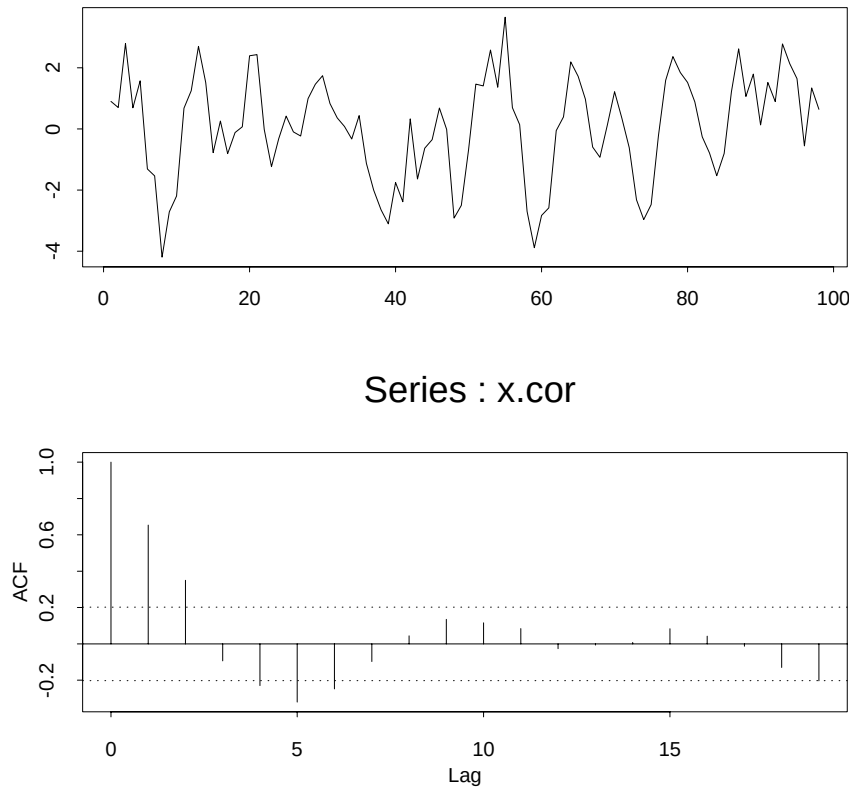

Figure 20.6: *Time series plot and ACF plot for a correlated series.*

You can specify the number of lags *M* for which you want autocorrelations by using the optional argument `lag.max=`.

The value of the autocorrelation function at lag 0 is always 1. The horizontal dotted lines provide an approximate 95% confidence interval for the autocorrelation estimate at each lag. If no autocorrelation estimate (given by the vertical lines for positive lags) falls outside the strip defined by the two dotted lines (and the data contain no outliers!), you may safely assume that there is no serial correlation. Otherwise, you should be concerned about the presence of serial correlation. In our example, the acf plot indicates serial correlation at lags 1 and 2.

ANALYZING TIME SERIES 21

The two general approaches to time series analysis are to use the data directly, or to use a frequency domain method.

ANALYZING TIME SERIES

There are two general approaches to analyzing time series. One is to use time domain methods in which the values of the process are used directly, the other is to use frequency domain methods. Frequency methods investigate the periodic properties of the process. The books by Chatfield (1984) and Shumway (1988) provide readable introductions to time series analysis which cover both time domain and frequency domain methods.

Fields of study tend to focus on analyzing data in one domain or the other. For example, economists use the time domain extensively while electrical engineers often use the frequency domain. To a large extent, this division arises from the types of questions that are being asked of the data. However, combining the approaches can at times give a more thorough understanding of the data.

Robust methods are necessary for both domains because the failure of model assumptions (such as Gaussian errors) can cause misleading results when classical techniques are applied.

Time domain methods are covered in sections 21.1, 21.2, and 21.3. Section 21.5 treats univariate and multivariate frequency domain spectral analysis methods, including complex demodulation and least squares low-pass filtering. General recursive and non-recursive filtering algorithms are provided in section 21.6. Section 21.7 explains robust methods.

21.1 COVARIANCE, CORRELATION, AND PARTIAL CORRELATION

UNIVARIATE SERIES

The autocovariance function is an important tool for describing the serial (or temporal) dependence structure of a univariate time series. Let x_t be a stationary time series with mean μ and variance σ_x^2, and assume for ease of notation that t takes on integer values $t = 0, \pm 1, \pm 2, \ldots$. The autocovariance function of x_t at lag k is defined as

$$\gamma(k) = E(x_t - \mu)(x_{t+k} - \mu). \tag{21.1}$$

Since x_t is stationary, this does not depend on t. The autocorrelation function at lag k is defined as

$$\rho(k) = \frac{\gamma(k)}{\gamma(0)} = \frac{\gamma(k)}{\sigma_x^2} \tag{21.2}$$

and is simply a standardized version of the autocovariance function. Both the

autocovariance function and the autocorrelation function are even functions; that is, $\gamma(k) = \gamma(-k)$ and $\rho(k) = \rho(-k)$. In addition, the autocorrelation function satisfies

$$|\rho(k)| \le 1 \quad \text{for all } k = 0, \pm 1, \pm 2, \ldots \tag{21.3}$$

Example 1. White Noise. A stationary time series for which x_t and x_{t+k} are uncorrelated, i.e., $\gamma(k) = E(x_t - \mu)(x_{t+k} - \mu) = 0$ for all integers $k \neq 0$, is called *white noise*. Such a process is sometimes loosely termed a "purely random process." Since $\gamma(0) = \sigma_x^2$, a white noise process has autocovariance function

$$\gamma(k) = \begin{cases} \sigma_x^2 & k=0 \\ 0 & k \neq 0 \end{cases} \tag{21.4}$$

and autocorrelation function

$$\rho(k) = \begin{cases} 1 & k=0 \\ 0 & k \neq 0 \end{cases} . \tag{21.5}$$

Example 2. Moving Average Process. A moving average process of order q, denoted MA(q), is defined by the equation

$$x_t = \mu + \beta_0 \epsilon_t + \beta_1 \epsilon_{t-1} + \cdots + \beta_q \epsilon_{t-q} \tag{21.6}$$

where ϵ_t is a white noise process. It is easy to show that the autocovariance function for this process is given by

$$\gamma(k) = \begin{cases} \sum_{t=0}^{q-|k|} \beta_t \beta_{t+|k|} & |k| \le q \\ 0 & |k| > q \end{cases} \tag{21.7}$$

and the autocorrelation function is given by

$$\rho(\tau) = \begin{cases} \sum_{t=0}^{q-|\tau|} \beta_t \beta_{t+|\tau|} & |\tau| \le q \\ 0 & |\tau| > q \end{cases} \tag{21.8}$$

The *autocovariance* function estimate at lag *k* is:

$$\hat{\gamma}(k) = \frac{1}{n} \sum_{t=1}^{n-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{21.9}$$

where

$$\bar{x} = \frac{1}{n} \sum_{t=1}^{n} x_t$$

is the mean of the series and *n* is the length of the observed series. Notice that the divisor *n* is used, even though there are only *n-k* terms. As a result, $\hat{\gamma}(k)$ is a biased estimate, even if $\bar{x}$ is replaced by the true mean μ. However, $\hat{\gamma}(k)$ has some other properties which make up for a small amount of bias. In particular, use of the divisor *n* ensures positive semi-definiteness of the function $\hat{\gamma}(k)$, and the mean squared error of this estimate is often smaller than that obtained when n^{-1} is replaced by $(n-k)^{-1}$. See Priestley (1981) for details.

The *autocorrelation* function estimate at lag *k* is.

$$\hat{\rho}(k) = \frac{\hat{\gamma}(k)}{\hat{\gamma}(0)} \tag{21.10}$$

Multivariate Series

The autocovariance and autocorrelation functions for multivariate series are defined analogously to those of univariate series. In addition, one is interested in *crosscovariance* and *crosscorrelation* functions. Suppose that x_t is an *m*-variate stationary time series, and $x_{it} = (x_t)_i$ is the *i*th time series $i = 1, \ldots, m$ with mean values $\mu_i = Ex_{it}$, $i=1, \ldots, m$.

The covariance function matrix for $x_t = (x_{1t}, \ldots, x_{mt})$ at lag k is defined as

$$\boldsymbol{\Gamma}(k) = E(\boldsymbol{x}_t - \boldsymbol{\mu})(\boldsymbol{x}_{t+k} - \boldsymbol{\mu})^T \tag{21.11}$$

where $\boldsymbol{a}^T$ is the transpose of $\boldsymbol{a}$ and $\mu^T = (\mu_1, \ldots, \mu_m)$. $\boldsymbol{\Gamma}(k)$ is an $m \times m$ matrix with the property that $\boldsymbol{\Gamma}^T(k)=\boldsymbol{\Gamma}(-k)$. The ith main diagonal element of $\boldsymbol{\Gamma}(k)$ is the *autocovariance* function

$$\gamma_{ii}(k) = E(x_{it} - \mu_i)(x_{i(t+k)} - \mu_i) \tag{21.12}$$

for the ith time series x_{it}, $i = 1, \ldots, m$. The ijth off-diagonal element of $\boldsymbol{\Gamma}(k)$ is the *cross*covariance

$$\gamma_{ij}(k) = E(x_{it} - \mu_i)(x_{j(t+k)} - \mu_j) \tag{21.13}$$

for the ith and jth series x_{it} and x_{jt}; $i, j = 1, \ldots, m$, $i \neq j$. Note carefully that a crosscovariance function $\gamma_{ij}(k)$, $i \neq j$, is *not* generally symmetric in k; i.e., in general $\gamma_{ij}(k) \neq \gamma_{ij}(-k)$. The estimate of either an autocovariance or crosscovariance at lag k is given by

$$\hat{\gamma}_{ij}(k) = \frac{1}{n}\sum_{t=1}^{n-k}(x_{it} - \bar{x}_i)(x_{j(t+k)} - \bar{x}_j) \tag{21.14}$$

where

$$\bar{x}_i = \frac{1}{n}\sum_{t=1}^{n} x_{it}$$

Note that for $i = j$, the autocovariance estimate $\hat{\gamma}_{ij}(k)$ in equation 21.14 has the same form as in 21.9. The autocorrelation and crosscorrelation estimates at lag k are

$$\rho_{ij}(k) = \frac{\hat{\gamma}_{ij}(k)}{\sqrt{\hat{\gamma}_{ii}(0)\hat{\gamma}_{jj}(0)}}. \tag{21.15}$$

Partial Autocorrelation

Another useful diagnostic tool for the analysis of the serial dependence is the partial autocorrelation function. Background on this will be deferred to the next section, after introducing autoregressive processes.

Examples of Simple Use

The function `acf` can be used to compute the sample autocovariance, autocorrelation or partial correlation functions for a specified number k of lags.

To compute an estimate of the autocorrelation function $\gamma(k)$ for lags k=0, 1, …, 40 of the univariate series `log.lynx` we can use the command:

```
> llynx.acr <- acf(log(lynx), 40, "correlation")
```

The result is plotted automatically (figure 21.1). The horizontal band about zero represents the approximate 95% confidence limits for $H_0 : \rho = 0$.

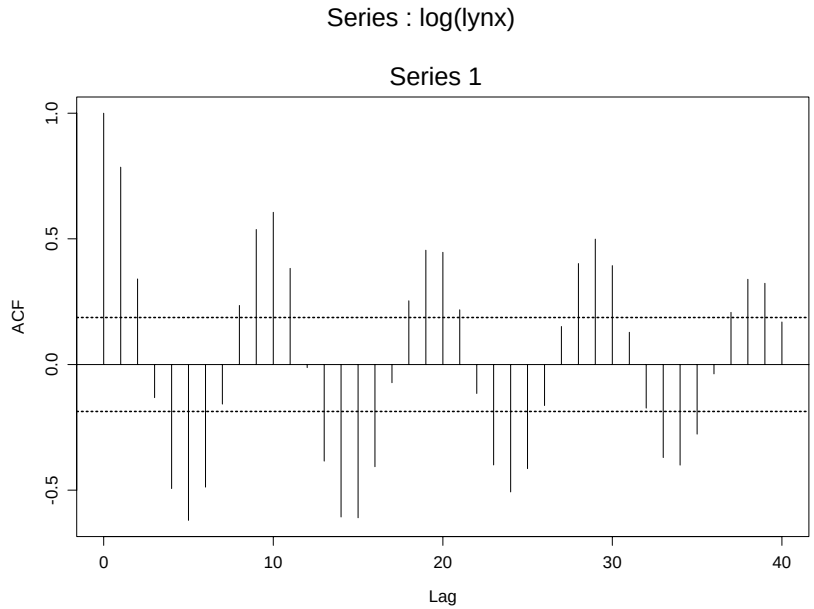

Figure 21.1: *Autocorrelation for the logarithm of the lynx data.*

The function `acf.plot` can be used to plot the results from `acf`. This function will take the list returned by the function `acf` and use its components in calculating approximate limits and deciding layout and appropriate labeling.

21.2 AUTOREGRESSION METHODS

Univariate Autoregressions

Consider a time series x_t that satisfies the difference equation (recursion)

$$x_t = \alpha_1 x_{t-1} + \alpha_2 x_{t-2} + \cdots + \alpha_p x_{t-p} + \varepsilon_t \tag{21.16}$$

where ϵ_t is a white noise process with zero mean and finite variance σ_ε^2. The time series x_t is called an autoregressive process of order p and is denoted AR(p). The x_t in equation 21.16 has zero mean, a fact which can be easily verified. An AR(p) process with nonzero mean μ is generated by the equation

$$x_t - \mu = \alpha_1(x_{t-1} - \mu) + \cdots + \alpha_p(x_{t-p} - \mu) + \varepsilon_t \tag{21.17}$$

It is worth noting that an AR(p) process is a pth-order Markov process.

Not all values of the autoregression coefficients $\alpha_1, \ldots, \alpha_p$ result in a stationary process. In particular, in an AR(1) process

$$x_t = \alpha x_{t-1} + \varepsilon_t, \tag{21.18}$$

it is fairly easy to show that the condition for stationarity is that $|\alpha| < 1$. For $\alpha = 1$, the AR(1) process becomes a discrete time random walk, which is well known to be non-stationary. For an AR(p) process, the condition for stationarity is that the (complex) roots of

$$\phi(z) = 1 - \alpha_1 z - \alpha_2 z^2 - \cdots - \alpha_p z^p \tag{21.19}$$

lie outside the unit circle. An interpretation of AR(2) models from a physical point of view is given by Priestley (1981).

Autoregressive models have seen a wide range of uses in statistics (for example, for forecasting and autoregression-type spectral density function estimation) and engineering (for example, in speech analysis and recognition systems) where autoregression modeling is referred to as linear prediction modeling. For many applications autoregression provides a good approximate (linear) model which has the virtue of extreme simplicity. In particular, the equations used to estimate the unknown coefficients $\alpha_1, \ldots, \alpha_p$ are linear, as we point out below. Of course one should be careful not to insist on using an autoregression model where another type of model may be appropriate (for example, a moving average component is needed, non-stationarity must be

dealt with, or a nonlinear model is needed). When in doubt consult an experienced statistician with a time series background.

The Yule-Walker Equations.

Let $\gamma(k)$ be the autocovariance function for the AR(p) process x_t. Then it may be shown that the AR(p) coefficients $\alpha_1, \ldots, \alpha_p$ satisfy the Yule-Walker equations

$$\sum_{k=1}^{p} \gamma(k-i)\,\alpha_k = \gamma(i) \qquad i = 1, 2, \ldots, p\,. \tag{21.20}$$

In addition, one can show that.

$$\sigma_x^2 = \sum_{k=1}^{p} \gamma(k)\,\alpha_k + \sigma_\varepsilon^2 \tag{21.21}$$

Given that the AR(p) coefficients satisfy the Yule-Walker equations 21.20, there is a very natural way to obtain estimates $\hat{\alpha}_1, \hat{\alpha}_2, \ldots, \hat{\alpha}_p$ based on a finite sample $x_1, x_2, \ldots, x_n$ of the time series. Namely, replace the $\gamma(k)$ in 21.20 by the estimates

$$\hat{\gamma}(k) = \frac{1}{n} \sum_{t=1}^{n-|k|} (x_t - \bar{x})(x_{t+|k|} - \bar{x}) \tag{21.22}$$

where

$$\bar{x} = \sum_{t=1}^{n} x_t\,, \tag{21.23}$$

and solve the resulting equations for $\hat{\alpha}_1, \ldots, \hat{\alpha}_p$. Since $\hat{\gamma}(-k) = \hat{\gamma}(k)$, we

can write the equations as

$$\begin{aligned}
\hat{\gamma}(1) &= \alpha_1\hat{\gamma}(0) + \alpha_2\hat{\gamma}(1) + \alpha_3\hat{\gamma}(2) + \cdots + \alpha_p\hat{\gamma}(p-1) \\
\hat{\gamma}(2) &= \alpha_1\hat{\gamma}(1) + \alpha_2\hat{\gamma}(0) + \alpha_3\hat{\gamma}(1) + \cdots + \alpha_p\hat{\gamma}(p-2) \\
\hat{\gamma}(3) &= \alpha_1\hat{\gamma}(2) + \alpha_2\hat{\gamma}(1) + \alpha_3\hat{\gamma}(0) + \cdots + \alpha_p\hat{\gamma}(p-3) \\
&\cdots \\
\hat{\gamma}(p) &= \alpha_1\hat{\gamma}(p-1) + \alpha_2\hat{\gamma}(p-2) + \alpha_3\hat{\gamma}(p-3) + \cdots + \alpha_p\hat{\gamma}(0).
\end{aligned} \tag{21.24}$$

We call these equations the sample-based Yule-Walker equations. Once the $\hat{\alpha}_j$'s are obtained by solving equation 21.24, we can use them along with the $\hat{\gamma}(k)$ in equation 21.21,

$$\hat{\gamma}(0) = \alpha_1\hat{\gamma}(1) + \alpha_2\hat{\gamma}(2) + \cdots + \alpha_p\hat{\gamma}(p) + \hat{\sigma}_\varepsilon^2, \tag{21.25}$$

to solve for σ_ε^2.

In practice, the order of the autoregression is not known and often it is desired to compare solutions of various orders. Hence we will wish to solve the equation set 21.24 for a variety of values of p from 1 up through p_{max}, where p_{max} is sometimes 10 or 15 or even larger.

The Levinson-Durbin Recursion.

Because the matrix of coefficients for the equation set 21.24 is a Toeplitz matrix (i.e., the elements on each diagonal are all the same), there is a recursive method which allows you to obtain estimates for a *k*th-order model from the estimates of the *k*-1 model in a fast and accurate manner. The method is referred to as the Levinson or Levinson-Durbin algorithm. Let $a_{i,k}$ denote the estimate of the *i*th autoregression coefficient (α_i) in an AR(*k*) model. If we have the estimates $a_{i,k}, \ldots, a_{k-1,k-1}$ and the estimated error variance σ_{k-1}^2 assuming an AR(*k-1*) model, then estimates for an AR(*k*) model are

$$a_{k,k} = \frac{\hat{\gamma}(k) - \sum_{j=1}^{k-1} a_{j,k-1}\hat{\gamma}(j-k)}{\sigma_{k-1}^2} \tag{21.26}$$

where

$$a_{j,k} = a_{j,k-1} - a_{k,k}a_{k-j,k-1} \quad \text{for } 1 \le j \le k-1 \tag{21.27}$$

and

$$\sigma_k^2 = \sigma_{k-1}^2(1 - a_{k,k}^2). \tag{21.28}$$

From equation 21.28 it may be seen that the squares of the $a_{k,k}$ can be interpreted as a measure of the usefulness of increasing the order of the AR process from k-1 to k. The $a_{k,k}$ sequence is called the *partial autocorrelation function* or "reflection coefficients," depending on the field of study. This sequence is useful in diagnosing whether the series is in fact an AR process. If the process is an AR(p), then all $a_{k,k}$ should be close to zero for $k > p$. A common approximation for the standard error of the $a_{k,k}$ for $k > p$ is $(1/n)^{1/2}$. See Box and Jenkins (1976).

AIC Order Selection

A way of selecting the order of the AR process is to find an order that balances the reduction of estimated error variance with the number of parameters being fit. One such measure is Akaike's Information Criterion (AIC). For the present case of an order k model, this criterion can be written as

$$AIC(k) = n\,log(\hat{\sigma}_{\varepsilon,k}^2) + 2k. \tag{21.29}$$

If the series is an AR process, then the value of k which minimizes $AIC(k)$ is an estimate of the order of the autoregression.

Multivariate Autoregressions

If the scalar quantities x_t, ϵ_t, and μ in equation 21.17 are replaced by m-dimensional vectors $\boldsymbol{x}_t$, $\boldsymbol{\epsilon}_t$ and $\boldsymbol{\mu}$, and the scalars α_t are replaced by $m \times m$ matrices A_t, we obtain the multivariate pth-order autoregression.

$$x_t - \mu = \boldsymbol{A}_1(\boldsymbol{x}_{t-1} - \boldsymbol{\mu}) + \cdots + \boldsymbol{A}_p(\boldsymbol{x}_{t-p} - \boldsymbol{\mu}) + \boldsymbol{\epsilon}_t \tag{21.30}$$

Here $\boldsymbol{\epsilon}_t$ is an m-dimensional white noise series with mean zero and covariance matrix $\boldsymbol{Q}$. This covariance matrix is sometimes loosely referred to as the "prediction variance."

The vector autoregression $\boldsymbol{x}_t$ satisfies a vector analogue of the Yule-Walker

equations 21.20. Namely, with $\boldsymbol{\Gamma}(i) = cov\{\boldsymbol{x}_t, \boldsymbol{x}_{t+i}\}$ we have

$$\sum_{k=1}^{p} \boldsymbol{\Gamma}(k-i)\boldsymbol{A}_k = \boldsymbol{\Gamma}(i), \qquad i = 1, 2, \ldots, p. \tag{21.31}$$

We also have the vector autoregression analogue of 21.21, namely

$$\boldsymbol{\Gamma}(0) = \sum_{k=1}^{p} \boldsymbol{\Gamma}(k)\boldsymbol{A}_k + \boldsymbol{Q}. \tag{21.32}$$

Sample Yule-Walker equations for this vector case are obtained by replacing the $\hat{\gamma}(k)$'s in 21.22 by

$$\hat{\boldsymbol{\Gamma}}(k) = \frac{1}{n}\sum_{t=1}^{n-|k|} (\boldsymbol{x}_t - \bar{\boldsymbol{x}})(\boldsymbol{x}_{t+|k|} - \bar{\boldsymbol{x}})^T \tag{21.33}$$

where

$$\bar{\boldsymbol{x}} = \frac{1}{n}\sum_{t=1}^{n} \boldsymbol{x}_t, \tag{21.34}$$

and solving the equations

$$\sum_{k=1}^{p} \hat{\boldsymbol{\Gamma}}(k-i)\hat{\boldsymbol{A}}_k = \hat{\boldsymbol{\Gamma}}(i), \qquad i = 1, 2, \ldots, p \tag{21.35}$$

for the estimates $\hat{A}_k$, $k = 1, \ldots, p$. The multivariate version of 21.25 is then

$$\hat{\boldsymbol{\Gamma}}(0) = \sum_{k=1}^{p} \hat{\boldsymbol{\Gamma}}(k)\hat{\boldsymbol{A}}_k + \hat{\boldsymbol{Q}}, \tag{21.36}$$

which may be solved for $\hat{Q}$.

There is also an analogue of the Levinson-Durbin algorithm (equations 21.26-21.28), which may be used to obtain estimates $\hat{\boldsymbol{A}}_{i,k}$, $i = 1, ...,k$ and $\hat{\boldsymbol{Q}}_k$ for a kth-order vector autoregression given estimates $\hat{\boldsymbol{A}}_{i,k}$, $i = 1, \ldots, k\text{-}1$, and $\hat{\boldsymbol{Q}}_{k-1}$ for an order k-1 vector autoregression. This method is referred to as "Whittle's recursion."

Autoregression Estimation via Yule-Walker Equations

The S-PLUS function `ar.yw` fits autoregressive models to multivariate time series using Whittle's extension to the Levinson-Durbin recursion.

Examples of Simple Use

The following S-PLUS commands fit an autoregression model to the log of the lynx time series.

```
> llynx.ar <- ar.yw(log(lynx))
> llynx.ar$order.max
  [1] 20
> llynx.ar$order
  [1] 11
> acf.plot(llynx.ar)

> ts.plot(llynx.ar$aic, main=
+      "Akaike Information Criteria for log(lynx)")
```

The result of the `acf.plot` command is shown in figure 21.2; the output from the `ts.plot` command is shown in figure 21.3. The maximum order fit defaults to 20 in this case, and the AIC picks a model of order 11. Figure 21.3 shows the minimum AIC at 12; this plot starts indexing at 1, but the first element of the `aic` component is for order 0.

A plot is also made of the residuals:

```
> ts.plot(llynx.ar$resid, main=
+ "Residuals after fitting an AR(11) to log(lynx)")
> abline(h=0, lty=2)
```

The resulting plot is shown in figure 21.4.

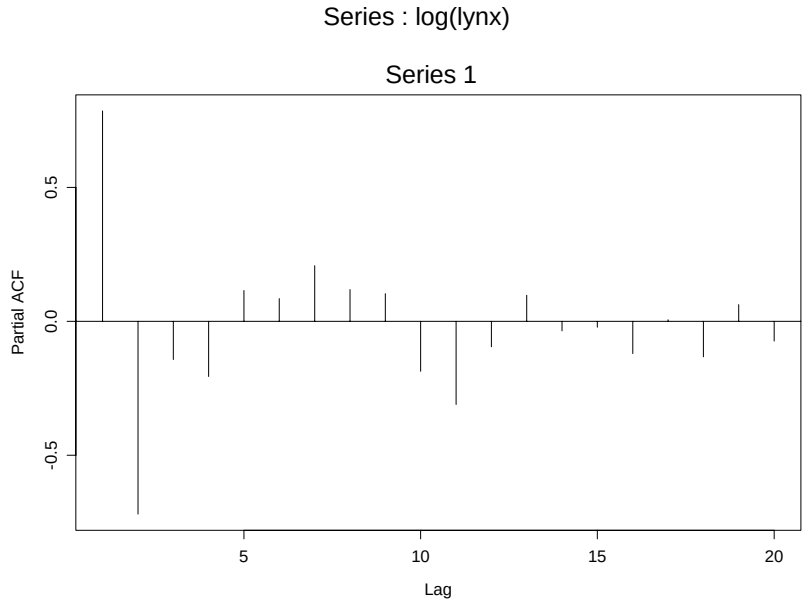

Figure 21.2: *Partial autocorrelation for the lynx data.*

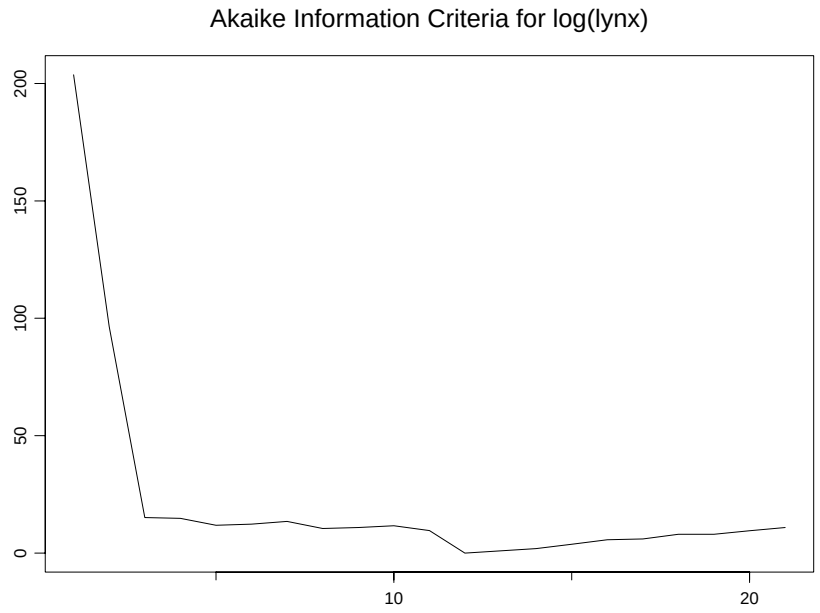

Figure 21.3: *AIC for the lynx data.*

Autoregression Estimation with Burg's Algorithm

This section presents Burg's algorithm, an alternative to using Yule-Walker equations for fitting autoregressive models. Burg's approach is based on estimating the kth partial correlation coefficient by minimizing the sum of forward and backward prediction errors.

$$SS(a_{k,k}) = \sum_{t=k+1}^{n} \{ [x_t - a_{1,k}x_{t-1} - \cdots - a_{k,k}x_{t-k}]^2 + [x_{t-k} - a_{1,k}x_{t-k+1} - \cdots - a_{k,k}x_t]^2 \}. \quad (21.37)$$

Given all of the coefficients for the order k-1 model, this is a function only of $a_{k,k}$. Equation 21.37 essentially measures how well the order k model predicts forwards and backwards. The algorithm is optimal in the sense of maximizing a measure of entropy. See Burg (1967).

Examples of Simple Use

The following S-PLUS commands fit an AR(2) model to the log of the lynx time series using Burg's algorithm.

```
> llynx.arb <- ar.burg(log(lynx),  F, 2)
> llynx.ar <- ar(log(lynx), aic=F, order.max=2)
> llynx.arb$ar
  , , 1
            [,1]
  [1,]  1.5595934
  [2,] -0.5711427
> llynx.ar$ar
  , , 1
            [,1]
  [1,]  1.3504381
  [2,] -0.7200314
```

Finding the Roots of a Polynomial Equation

The function `polyroot` finds the zeroes of the complex-valued polynomial equation:

$$a_k z^k + \cdots + a_1 z + a_0 = 0.$$

Use this function to find the roots of an autoregression or moving average operator with user-specified coefficients. For example, if one has estimated pth-order autoregressive coefficients, $\hat{\phi}_1, \hat{\phi}_2, \ldots, \hat{\phi}_p$, then the autoregression polynomial is $1 - \hat{\phi}_1 z - \hat{\phi}_2 z^2 - \ldots - \hat{\phi}_p z^p$, and one would choose $\boldsymbol{a} = (a_0, \ldots, a_k)$ with $k = p$, $a_0 = 1$, and $a_i = -\hat{\phi}_i$, $i = 1, \ldots, p$.

Examples of Simple Use

To solve the equation $z^2 + 5z + 6 = 0$:

```
> polyroot(c(6,5,1))
  [1] -2+0i -3+0i
```

21.3 UNIVARIATE ARIMA MODELING

S-PLUS provides several functions for fitting autoregressive integrated moving-average (ARIMA) models to univariate time series data. ARIMA models are useful for a wide variety of problems including forecasting, quality control, seasonal adjustment, spectral estimation, as well as providing a summary of the data. Box and Jenkins (1976) give a comprehensive account of ARIMA modeling, and discussions of ARIMA models can be found in many recent standard textbooks for time series.

ARMA Models

A stationary autoregressive moving-average process is obtained by combining the equations for an MA process given by 21.6 and an AR process given by 21.16. A zero mean ARMA(p, q) process x_t can be written in the form

$$x_t - \phi_1 x_{t-1} - \cdots - \phi_p x_{t-p} = \varepsilon_t - \theta_1 \varepsilon_{t-1} - \cdots - \theta_q \varepsilon_{t-q} \qquad (21.38)$$

where ϵ_t is a white noise process; that is, the ϵ_t are uncorrelated, and have zero mean and variance σ^2. The process ϵ_t is sometimes called the *innovations* process. The parameters $\phi_1, \ldots, \phi_p$ are the autoregressive coefficients, and the parameters $\theta_1, \ldots, \theta_q$ are the moving-average coefficients.

If the innovations ϵ_t are Gaussian (the process x_t is Gaussian) and the ϵ_t are uncorrelated, then they are also independent. This is a frequently used assumption.

The ARMA model of 21.38 is often written in the form $\phi(B)x_t = \theta(B)\epsilon_t$, where B is a *backshift* operator (that is, $B(x_t) = x_{t-1}$) and

$$\begin{aligned} \phi(B) &= 1 - \phi_1 B - \cdots - \phi_p B^p \\ \theta(B) &= 1 - \theta_1 B - \cdots - \theta_q B^q. \end{aligned} \qquad (21.39)$$

ARIMA Models

Many time series encountered in practice are *non-stationary*. For these series, simple ARMA models are typically inadequate. However, the *differenced* series may be stationary. Box and Jenkins (1976) developed a methodology for fitting ARMA models to differenced data. These are known as autoregressive integrated moving-average (ARIMA) models. An

ARIMA(p,d,q) process x_t can be defined by

$$\phi(B)\nabla^2 x_t = \theta(B)\varepsilon_t \tag{21.40}$$

where ϵ_t, $\phi(B)$, and $\theta(B)$ are as above, $\nabla = 1\text{-}B$ is the first-difference operator and $\nabla^d = (1\text{-}B)^d$ is the d-fold differencing operator. For example, with d=1, the differenced series $w_t = \nabla x_t = x_t - x_{t\text{-}1}$ is assumed to follow an ARMA(p,q) process: $\phi(B)w_t = \theta(B)\epsilon_t$. When $d = 2$, the twice differenced series w_t is an ARMA(p,q) process:

$$w_t = \nabla^2 x_t = \nabla(x_t - x_{t-1}) = x_t - 2x_{t-1} + x_{t-2}$$

Seasonal Models

Time series data frequently exhibit seasonal cycles or periodicities. For example, data collected on a monthly basis may have a period of length s = 12 months, reflecting the *seasonal* behavior of the process. The framework for ARIMA models can be extended to handle periodicities as well (see Box and Jenkins (1976), chapter 9). The seasonal behavior is modeled by using seasonal autoregressive, moving average, and differencing operators. For a period of length s, these operators are of the form

$$\begin{aligned} \Phi(B^s) &= 1 - \Phi_1 B^s - \cdots - \Phi_P B^{sP} \\ \Theta(B^s) &= 1 - \Theta_1 B^s - \cdots - \Theta_Q B^{sQ} \\ \nabla_s^D &= (1 - B^s)^D \end{aligned} \tag{21.41}$$

The parameters $\Phi_1, \ldots, \Phi_p$ are the seasonal autogressive coefficients and the parameters $\Theta_1, \ldots, \Theta_Q$ are the seasonal moving average coefficients. ∇_s^d is the seasonal d-fold differences operator. Typically, $\Phi(B^s)$, $\Theta(B^s)$, and ∇_s^d are combined with the ordinary operators $\Phi(B)$, $\Theta(B)$, and ∇^d in a multiplicative fashion.

The multiplicative seasonal ARIMA$(p,d,q)\times(P,D,Q)_s$ process can be represented by

$$\Phi(B^s)\phi(B)\nabla_s^D \nabla^d x_t = \Theta(B^s)\theta(B)\varepsilon_t \tag{21.42}$$

In general, S-PLUS allows for any number of multiplicative operators with arbitrary periods. However, 21.42 should be sufficiently general for most problems.

ARIMA Models with Regression Variables

In addition to using past values to model a series, it is often desirable to use explanatory or regression variables. The regression variables may simply be a constant (intercept) term, a deterministic function of time, dummy variables to model outliers, or lagged values of another time series.

Let $\boldsymbol{z}_t$ be a vector of m elements. An ARIMA process y_t with (known) regression variables is defined by

$$y_t = \boldsymbol{z}_t'\beta + x_t \tag{21.43}$$

where β is an (unknown) parameter vector and x_t is an ARIMA process. For example, setting $\boldsymbol{z}_t' = (1, t)$ would result in a straight line regression model component $\boldsymbol{z}_t'\ \beta = \beta_1 + \beta_2 t$ with slope β_2 and intercept β_1.

Identifying and Fitting ARIMA Models

Box and Jenkins (1976) give a paradigm for fitting ARIMA models, which is to iterate through the following steps:

1. *Model Identification*: Determination of the ARIMA model orders (p, d, q) and (P, D, Q).
2. *Estimation of Model Parameters*: The unknown parameters in 21.42 and 21.43 are estimated.
3. *Diagnostics and Model Criticism*: The residuals are used to validate the model and to suggest potential alternative models which may be better.

These steps are repeated until a satisfactory model is found.

Model Identification

Initial model identification is done using the autocorrelation and partial autocorrelation functions. These can be computed using the S-PLUS function `acf`. See chapter 6 of Box and Jenkins (1976) for a complete discussion on the identification of ARIMA models.

An alternative procedure for selecting the model order is use of a penalized log-likelihood measure. One such measure is Akaike's Information Criterion (AIC). For autoregressive models, AIC is given by 21.29. For general ARIMA models, AIC is defined below in 21.46.

Estimation of Model Parameters

ARMA Models

The log likelihood for an ARMA model 21.40 can be computed using the

prediction error decomposition (see Harvey (1981)). Consider an ARMA process x_t as in 21.38, and assume the innovations ϵ_t are independent Gaussian random variables. Let

$$\hat{x}_t^{t-1} = E(x_t | x_1, \ldots, x_{t-1}, \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q)$$

denote the conditional mean one-step-ahead prediction of x_t based on the data $x_1, x_2, \ldots, x_{t-1}$, and let

$$\sigma^2 f_t = \text{var}(x_1, \ldots, x_{t-1}, \phi_1, \ldots, \phi_p, \theta_1, \ldots, \theta_q) \qquad (21.44)$$

denote the conditional variance of $\hat{x}_t^{t-1}$. The parameter σ^2 is the variance of the innovations process ϵ_t. Defining the *prediction errors* by $e_t = x_t - \hat{x}_t$, and letting $L = L(x_1, \ldots, x_n)$ denote the likelihood, one can show that.

$$-2 \log L(x_1, \ldots, x_n) = n \log(2\pi\sigma^2) + \sum_{t=1}^{n} f_t + \frac{1}{\sigma^2} + \sum_{t=1}^{n} e_t^2 / f_t. \qquad (21.45)$$

Fitting an ARMA(p,q) model by Gaussian maximum likelihood involves finding the estimates $\hat{\phi}_1, \ldots, \hat{\phi}_p$ and $\hat{\theta}_1, \ldots, \hat{\theta}_q$ which yield a minimum in 21.45. The parameters $\phi_1, \ldots, \phi_p$ and $\theta_1, \ldots, \theta_q$ enter into 21.45 through 21.44. The estimate of σ^2 is $\sum_{t=1}^{n} e_t^2 / f_t$, which can be concentrated out of the likelihood. The likelihood is, in general, nonlinear in $\phi_1, \ldots, \phi_p$ and $\theta_1, \ldots, \theta_q$ and so a nonlinear optimizer must be used.

The likelihood for an ARMA model 21.43 with regression variables can be computed in a similar fashion. In this case replace x_t's by y_t's in 21.45. The regression coefficients can be concentrated out of the likelihood (see Kohn and Ansley, (1985)).

A so-called conditional log-likelihood approximation to 21.45 can be obtained by *conditioning* on the first p values of the series, where p is the order of the autoregressive operator.

This conditional log-likelihood function is given by.

$$-2 \log L(x_{p+1}, \ldots, x_n | x_1, \ldots, x_p) = (n-p) \log(2\pi\sigma^2) + \sum_{t=p+1}^{n} \log f_t + \frac{1}{\sigma^2} \sum_{t=p+1}^{n} e_t^2 / f_t. \quad (21.46)$$

Bell and Hillmer (1987) give several arguments in favor of using 21.46. The main advantage with 21.46 is that the AR parameters $\phi_1, \ldots, \phi_p$ can be concentrated out of the likelihood, reducing the computational complexity of the nonlinear optimization. Usually, little information is lost in using 21.46 instead of 21.45.

The prediction errors e_t and their variances f_t can be computed in a number of ways. Ansley (1979) gives an efficient algorithm based on the Choleski decomposition of the covariance of the process x_t. However, if missing values are present, this algorithm no longer applies. Alternative algorithms are based on applying the Kalman filter to a state space representation of an ARMA process. See Jones (1980), Harvey (1981), and Kohn and Ansley (1986) for various methods based on the Kalman filter approach. All of these methods handle missing values, although the Kohn and Ansley approach is the most general.

Multiplicative ARIMA Models

Estimating multiplicative ARIMA models by Gaussian maximum likelihood is a straightforward extension from estimating ARMA models. With no missing data present, the likelihood for a non-stationary series is obtained by differencing the data and computing the likelihood for the differenced process.

With missing values present, the likelihood can be computed using the Kalman filter: see Kohn and Ansley (1986) and Bell and Hillmer (1987). The simplest approach is to condition on the first $p^* + d^*$ observations, where p^* and d^* are the orders of the expanded autoregressive and differencing operators obtained by multiplying the regular and seasonal AR and the regular and seasonal difference operators in 21.42. Specifically, $p^* = p + sP$ is the order of the polynomials $\Phi(B^s)\phi(B)$, and $d^* = d + sD$ is the order of $\nabla_s^D \nabla^d$. This gives the general ARIMA analog to the ARMA log-likelihood 21.46 and is equivalent to the differencing approach in the case of no missing values.

Missing Values in the Beginning of the Series

If a missing value occurs in the first $p^* + d^*$ observations, then conditioning on the first $p^* + d^*$ observations is not possible. In this case, the series can be reversed, and the likelihood function can be computed for the reversed series. The likelihood is invariant to reversing the order of the data. If there are missing values at both the beginning and the end of the series, then the exact likelihood must be computed using a modification of the Kalman filter, derived by Kohn and Ansley (1986). However, an approximate likelihood can be obtained by including a dummy regression variable for each missing value and replacing the missing value by an arbitrary number (see Bruce and Martin (1989)). The dummy regression variable is zero at all time points except for the time of the missing value.

Starting Values for the Optimizer

The likelihood is maximized using a general quasi-Newton optimizer (see the `nlmin` help file for a discussion of the optimizer). It is necessary to provide starting values for the ARIMA parameters. Poor starting values can lead to slow convergence to the maximum, or even worse, convergence to a local maximum. To avoid this, it is advisable to use a stepwise fitting procedure, starting with relatively simple ARIMA models and adding one coefficient at a time. Several tuning constants can be adjusted to provide better performance (see the `nlmin` help file). However, these usually do not need to be adjusted.

Transformation to Ensure Stationarity and Invertibility

The ARIMA coefficients can be transformed to ensure stationarity and invertibility of the model (see Jones, (1980)). If the solution lies on the boundary of stationarity or invertibility, then the optimizer may take many steps to converge. For this reason it may be desirable *not* to constrain the model to be invertible.

Warning

If printed output from the optimizer is requested, the printed coefficients are the *transformed* coefficients, and not the original ARIMA coefficients.

AIC and Model Selection

One method of model selection is based on Akaike's information criterion (AIC). The best model is given by the model with the lowest AIC value. AIC is a penalized version of the log-likelihood function 21.46, and is defined by

$$AIC = -2 \log L(x_{m+1}, \ldots, x_n | x_1, \ldots, x_m) + 2r \qquad (21.47)$$

where r is the total number of parameters estimated. Specifically, r is the number of AR, MA, and regression coefficients. For example, for an ARIMA $(1,1,1)$ model, r=2.

When comparing the AIC values for different models, it is important to *condition* the likelihood on the same number of observations. In other words, m should be the same in 21.47 for all models. This allows one to compare models with different numbers of AR or differencing coefficients using AIC.

Computational Notes

The S-PLUS function `arima.mle` fits ARIMA models to univariate time series data through Gaussian maximum likelihood. The conditional form of the likelihood 21.46 is used.

The regression parameters are concentrated out of the likelihood, as in Kohn and Ansley (1985). With no missing data, an algorithm similar to that of Ansley (1979) is used to compute the likelihood. With missing data, the Kalman filter is used with the state space representation of Kohn and Ansley (1986). However, missing values are not permitted in the beginning of the series; see the above discussion on missing values.

By default, the moving average parameters are transformed to ensure invertibility. However, if the solution lies on the boundary of invertibility, better performance by the optimizer can be obtained by *not* transforming the parameters. In certain circumstances, it might be useful to fit models in which lower order AR or MA parameters are constrained to be zero. In this case, the coefficients cannot be transformed to ensure stationarity or invertibility.

Examples of Simple Use

Simulate an MA(2) series and fit it using a Gaussian maximum likelihood.

```
> ma <- arima.sim(100, model=list(ma=c(-.5, -.25))
> ma.fit <- arima.mle(ma, model=list(ma=c(-.5, -.25))
```

Fit a Box-Jenkins (0,1,1) × (0,1,1) Airline model to the ship data. Use zeroes as the starting values for the optimizer.

```
> model <- list(list(order=c(0,1,1)), list(order=
+ c(0,1,1), period=12))
> fit <- arima.mle(ship, model=model)
```

Diagnostics and Model Criticism

The third stage in fitting ARIMA models consists of validating the model through examination of the one-step prediction residuals e_t. See chapter 8 of Box and Jenkins (1976) for a more complete discussion of ARIMA model diagnostics. The single most important diagnostic is a plot of the *standardized residuals* $\tilde{e}_t = e_t / \sqrt{f_t}$. If the correct ARIMA model is fit and the data are Gaussian, then $\tilde{e}_t$ should behave approximately like a Gaussian white noise process with zero mean and unit variance. Problems to look for in the plot of $\tilde{e}_t$ include outliers, non-homogeneity of variance, and obvious structure in time.

Another basic technique is to examine the autocorrelation function of the residuals e_t. Let $\hat{\gamma}_k$ be the autocorrelations of the residuals e_t. If the model is adequate, then $\hat{\gamma}_k$ should be uncorrelated and approximately Gaussian

random variables with mean zero and variance n^{-1}. Hence, the presence of large autocorrelations $\hat{\gamma}_k$ indicates that the model may be inadequate. The nature of the autocorrelations $\hat{\gamma}_k$ may suggest how to improve the model. However, some caution should be exercised in the use of $\hat{\gamma}_k$ to evaluate the model. For example, the variance times n^{-1} can be a serious overestimate of the true variance for small lags, leading to an underestimate of the significance for lack of fit.

In addition to examining the $\hat{\gamma}_k$'s individually, it is useful to base a diagnostic on the auto-correlations taken as a whole. Define the *portmanteau* test statistic Q by

$$Q = n \sum_{k=1}^{K} \hat{\gamma}_k^2$$

where K is a fixed maximum number of lags and n is the number of observations used to compute the likelihood. Typically, K should be between 10 and 20. If the correct ARIMA model is fit, and the data are Gaussian, then Q is approximately distributed as a χ^2 random variable on K-r degrees of freedom, where r is the number of parameters fit to the model.

The S-PLUS function `arima.diag` computes these diagnostics for an ARIMA model fit to a univariate time series.

Examples of Simple Use

Compute diagnostics for simulated AR(1) series.

```
> x <- arima.sim(model=list(ar=.9))
> fit <- arima.mle(x, model=list(ar=.9))
> diag <- arima.diag(fit)
```

Since, by default, `plot` = `TRUE` in `arima.diag`, the diagnostics will be plotted using the function `arima.diag.plot`.

Forecasting Using ARIMA Models

An important application of ARIMA models is to forecast beyond the end of a series. Under the assumption that the model order and parameters are known, the forecast means and confidence intervals are easily produced using the Kalman filter (see Harvey (1981)). Typically, one would first fit an ARIMA model using the techniques described in sections on pages 590–594. The resulting model can then be used to produce forecasts for the series.

The S-PLUS function `arima.forecast` produces forecasts given an ARIMA model for a univariate time series.

Predicted and Filtered Values for ARIMA Models

The S-PLUS function `arima.filt` produces one-step predicted values and their variances f_t, defined in 21.44. The primary application of `arima.filt` is for use in other S-PLUS functions: `arima.diag` (to compute the residuals) and `arima.forecast` (to compute the forecasts).

If autoregressive or differencing operators are present in the model, then predicted values are not produced for the first $p^* + d^*$ time points (p^* and d^* are the orders of the expanded autoregressive and differencing polynomials).

Computational Note

The function `arima.filt` also returns filtered values and their variances. Let y_t be a process which behaves according to a signal plus noise model

$$y_t = x_t + v_t$$

where x_t is the signal and v_t is the noise. A common problem is to extract the signal by filtering the observed process y_t. The filtered values and their variances are $E(x_t \mid y_1, \ldots, y_t)$ and $var(x_t \mid y_1, \ldots, y_t)$.

For a pure signal (v_t is 0 for all t), the filtered values are simply the observations themselves. The current version of S-PLUS does not support signal plus noise models. Hence, the filtered values are the same as the input series. However, the filtered values are returned for compatibility with future releases.

Simulating ARIMA Processes

The S-PLUS function `arima.sim` generates a simulated ARIMA process of the form 21.42 or 21.43 given an ARIMA model structure, regression variables and a vector of innovations or a random generator. The innovations vector corresponds to ϵ_t of 21.40, and can be input directly. Alternatively, a random generator may be supplied, and the innovations are generated accordingly.

For stationary ARMA processes, the series can be initialized by generating an initial random state vector according to a state space form of the model. The initial state vector is computed through transforming a white noise vector by the Choleski decomposition of the unconditional covariance matrix of the state vector.

For non-stationary ARIMA processes, the unconditional covariance matrix of the state vector doesn't exist. Hence, the simulated series is initialized by assuming that the initial state vector is zero. This is equivalent to assuming past innovations and simulated values are zero. To avoid the effects of the initialization, a series longer than the one needed is generated, and the simulated series is taken from the end of the generated series.

Examples of Simple Use

Simulate an ARMA(1,1):

```
> x <- arima.sim(model=list(ar=.5, ma=-.6), n=100)
```

Simulate an ARIMA(0,1,1) with contaminated innovations:

```
> rand.gen <- function(n) ifelse(runif(n)>.90, rnorm(n),
+ rcauchy(n))
> x.wild <- arima.sim(100, model=list(ndiff=1, ma=.6),
+ start.innov=50, rand.gen=rand.gen)
```

Modeling Effects of Trading Days

In many monthly or quarterly economic time series, the data are affected by the number of trading days in that month. For example, if a given month has more weekdays and fewer weekends than other months, then one might expect a higher level of economic activity during that month. One approach to handling the trading day effect is to include regression variables reflecting the number of Mondays, Tuesdays, etc., in each month (or quarter).

The function `arima.td` returns a multivariate time series which is suitable for use as a regression variable. The first column gives the number of days in the month (quarter). The following six columns give the number of Saturdays, Sundays, Mondays, Tuesdays, Wednesdays, and Thursdays *minus* the number of Fridays in the month (quarter). See Hillmer, Bell, and Tiao (1983) for use of trading day variables in ARIMA modeling of time series data.

Examples of Simple Use

```
> td.ship <- arima.td(ship)
> mle.td <- arima.mle(ship, model=list(order=c(0,1,1)),
+ xreg=td.ship)
```

21.4 LONG MEMORY TIME SERIES MODELING

Long memory is a common feature of time series in a wide variety of areas. It has enormous effects on standard statistical quantities such as standard errors and tests and hence on the conclusions drawn, but it is hard to detect. One major application has been to time series of wind speeds (Haslett and Raftery, (1989)), and there long memory means intuitively that there is a tendency to observe not just windy weeks and months, but windy years and decades and presumably also windy centuries and millennia; we often say that there is variation at all temporal scales.

Long memory time series have autocorrelations that decay slowly as lag increases; typically the autocorrelations tend to zero hyperbolically (that is, $\rho(k) \sim k^{-\alpha}$, with $\alpha > 0$) so that the sum of the autocorrelations is infinite (that is, $\sum_{k=0}^{\infty} \rho(k) = \infty$). Thus the autocorrelations between observations

far away from one another in time while small, are not negligible. The spectrum of a long memory time series goes to infinity as the frequency goes to zero at the rate $f(\omega) \sim \omega^{-(1-\alpha)}$.

One important property is that the variance of the sample mean declines not at the usual rate of $O(n^{-1})$, but at a slower rate. If $\rho(k) \sim k^{-\alpha}$, then $var(\bar{X}) = O(n^{-\alpha})$. (Note that a long memory time series is stationary only if $0 < \alpha \leq 1$.) This can have huge consequences. For example, in the wind data a was estimated to be 0.34, and this implied that for estimating the mean wind speed at a given location, twenty years of actual data were worth only about the same as one month of independent daily observations.

The ARMA models (with no differencing) discussed in sections 21.2 and 21.3 above are, by contrast, short memory models. For them, the autocorrelations decay exponentially, the sum of the autocorrelations is finite, the spectrum is finite at zero, and the variance of the sample mean is the usual $O(n^{-1})$. Fitting a (short memory) ARMA model to data can give very misleading results if the long memory property holds, even if the fitted model matches the lower-lag autocorrelations well. In the wind example, a fitted short memory ARMA model underestimated the variance of the sample mean by a factor of more than ten in many cases.

The long memory property in time series was discussed by Mandelbrot (1977) who called it the "Joseph effect" because of the sequence of seven years of plenty followed by seven lean years recounted in the Book of Genesis story of Joseph. Mandelbrot pointed out that long memory time series tend to be asymptotically approximately self-similar and hence to be, at least approximately, equivalent to fractals.

Fractionally Differenced ARIMA Modeling

Fractionally Differenced ARIMA Models.

The fractionally differenced ARIMA (p,d,q) model has been found to represent long memory time series quite well. It is defined by equation 21.40, namely

$$\phi(B)\nabla^d x_t = \theta(B)\varepsilon_t,$$

except that now d may take any value in the unit interval $[0, 1]$ instead of being restricted to being either 0 or 1, and $\nabla^d = (1\text{-}B)^d$ is defined by the

binomial expansion $(1\text{-}B)^d = \sum_{j=0}^{\infty} C(d,j)(-1)^j B^j$, where $C(d,j)$ are the binomial coefficients. When the series has a nonzero mean μ, the model is better written as .

$$x_t = \mu + \nabla^{-d}\phi(B)^{-1}\theta(B)\varepsilon_t . \tag{21.48}$$

For model 21.48, $\rho(k) = k^{-(1-2d)}$, so that $\alpha = 1\text{-}2d$, where α was defined in section 21.4. This model is stationary only for $0 \leq d < 1/2$, and reduces to the usual short memory ARMA(p,q) model when $d = 0$.

Estimation of Model Parameters

The log-likelihood for the fractionally differenced ARIMA(p,d,q) model of equation 21.48 can be computed exactly using the prediction error decomposition given by equation 21.45, where $\hat{x}_t^{t-1}$ and f_t are given by equations 4.3 and 4.4 of Haslett and Raftery (1989). A major practical problem with maximum likelihood estimation based on this likelihood is that the required CPU time is $O(n^2)$ and this can be enormous for the long series that are typical of application areas where long memory is known to arise often; for example in the wind data set, $n = 6574$.

We therefore use an approximation described in section 4.3 of Haslett and Raftery (1989) that essentially approximates the dependence of x_t on x_{t-j} for $j > M$ by asymptotic values. This reduces the order of the required CPU time from $O(n^2)$ to $O(n)$ and, in practice, for the wind data it reduced the actual computer time by a factor of 70. It is extremely accurate. We have found $M = 100$ to be a good choice; the exact maximum likelihood estimator can be recovered by setting $M = n$.

Computational Notes

The S-PLUS function `arima.fracdiff` estimates the parameters of the fractionally differenced ARIMA(p,d,q) model and returns exact or approximate maximum likelihood estimators, standard errors, the covariance and correlation matrices of the parameter estimates, and the log-likelihood. The degree of approximation is determined by M; we recommend $M = 100$. The exact maximum likelihood estimator can be found by setting $M = n$, but if the series is long it can require a lot of CPU time. The log-likelihood is useful for comparing models; that is, for choosing the number of AR and MA parameters. An approximate test of the long memory property can be carried out by dividing the estimate of d by its standard error and comparing the result with a standard normal distribution.

Simulating Fractionally Differenced ARIMA Processes

The S-PLUS function `arima.fracdiff.sim` generates a simulated fractionally differenced ARIMA(p,d,q) series of the form in equation 21.48 given the values of d, the AR and MA parameters, and the mean μ.

This uses the prediction error decomposition to generate x_t from its conditional distribution given the previous values.

Examples of Simple Use

Simulate a fractionally differenced ARIMA(2,.33,0):

```
> x.sim <- arima.fracdiff.sim( model = list( d=.33,
+ ar=c(.01,-.06), mu=3.1))
> arima.fracdiff( x.sim, model = list( ar=rep(2,NA)))
```

21.5 SPECTRAL ANALYSIS

Let x_t be a stationary time series with sampling interval Δt. A major theorem for time series states that any series with zero mean ($\mu = Ex_t = 0$) and finite variance $\sigma^2 = varx_t$ may be well approximated by a truncated Fourier series

$$x_t \approx \sum_{j=1}^{J} A_j \cos(2\pi f_j t) + B_j \sin(2\pi f_j t) \tag{21.49}$$

where A_j and B_j are *random* Fourier (series) coefficients, the f_j are well-chosen frequencies, and J is sufficiently large. This approximation of x_t as a Fourier series may be re-expressed in complex exponential form

$$x_t \approx \sum_{j=-J}^{J} C_j e^{i2\pi f_j t} \tag{21.50}$$

where the C_j are *complex random* Fourier coefficients which have zero mean, $EC_j = 0$ and are uncorrelated:.

$$cov(C_j, C_k) = EC_j\bar{C}_k = 0 \quad \text{for } j \neq k. \tag{21.51}$$

The notation $\bar{a}$ denotes the complex conjugate of a.

Sometimes the set of real coefficients A_j, B_j, or complex coefficients C_j are referred to as the (discrete time) Fourier transform of x_t.

Time series with a nonzero mean may be approximated by adding the mean μ to the right hand side of equation (21.50):

$$x_t \approx \mu + \sum_{j=1}^{J} C_j e^{i2\pi f_j t}. \tag{21.52}$$

The exact version of approximation 21.50 is an integral known as the *spectral representation* of x_t. The *spectrum* or *spectral density* $S(f)$ for the series x_t can be described in terms of the coefficients C_j defined in 21.50 as:.

$$S(f_j) = E|C_j|^2 \tag{21.53}$$

Thus the value of the spectrum at frequency f_j is the second moment of the random amplitude at frequency f_j. The spectrum $S(f)$ at an arbitrary frequency f may also be expressed exactly in terms of the autocovariance sequence .

$$R(l) = EX_t X_{t+l}, \quad l = 0, \pm 1, \pm 2, \ldots . \tag{21.54}$$

Namely, $S(f)$ has the exact Fourier series representation

$$S(f) = \sum_{l=-\infty}^{\infty} R(l) e^{-il2\pi f} \tag{21.55}$$

and the autocovariances are the Fourier coefficients of $S(f)$:.

$$R(l) = \int_{-\frac{1}{2}}^{\frac{1}{2}} S(f) e^{il2\pi f} df \tag{21.56}$$

Again, we often refer to $S(f)$ as the (discrete time) Fourier transform of $R(l)$, and refer to $R(l)$ as the inverse Fourier transform of $S(f)$.

Estimating the Spectrum From the Periodogram

Suppose that we have a time series $x_1, \ldots, x_n$ observed at a sampling interval Δ. The spectrum of this series may be estimated from the periodogram by using the function `spec.pgram`. The steps involved in this computation are described below.

1. *Detrending and De-meaning*

   The first step in estimating the spectrum is to ensure that the mean is zero for the time series. If it is thought that the original series may contain a linear trend, then this is accomplished by subtracting a least squares regression line from the series (that is, by replacing x_t with $x_t - \hat{\gamma} - \hat{\beta}t$ where $\hat{\gamma} + \hat{\beta}t$ is the least squares linear fit to the data). If it is thought that there is no trend in the data, it will suffice to subtract the mean from the series (that is, x_t is replaced by $x_t - \bar{x}$ where $\bar{x}$ is the sample mean of $x_1, \ldots, x_N$). By default the `spec.pgram` function removes the least squares line.

2. *Tapering*

   A data taper is often applied to the (detrended or de-meaned) series. A taper sequence w_t multiplies each value in a series by a number between 0 and 1. Tapering reduces "leakage" of power. See Bloomfield (1976) and Priestley (1981) for discussions of tapering. The `spec.pgram` function includes a default split cosine taper of ten percent on each end of the series. See page 608 for further details.

3. *Padding*

   Padding consists of increasing the length of the series x_t from n to n' by adding $n' - n$ zero values $x_{n+1} = \ldots = x_{n'} = 0$. Padding may generally be ignored for the spectrum function—see discussions of the fast Fourier transform (FFT) in the references for explanation.

4. *The Periodogram*

   To avoid extra notation, let n be the length of the series with or without padding. Let Δ be the sampling interval; that is, $\Delta = 1/freq$ where *freq* is the frequency sampling rate component of the `tspar` attribute, and where following the above operations, an estimate of the power spectrum at discrete Fourier frequencies $f_k = k/\Delta n$ is found by forming the periodogram

$$\begin{aligned} \hat{S}(f_k) &= \frac{1}{n}\left|\sum_{t=1}^{n} \tilde{x}_t \, exp(-2\pi i f_k t)\right|^2 \\ &= \frac{n}{4}(A_k^2 + B_k^2), \quad k = 0, 1, \ldots, n/2 \end{aligned} \tag{21.57}$$

   where $\tilde{x}_t = w_t(x_t - \hat{\gamma} - \hat{\beta}t)$ is the tapered, detrended series. Note that $\hat{\beta} = 0$ and $\hat{\gamma} = \bar{x}$ if only a mean was removed from the series. The discrete Fourier transform (DFT) sum in equation 21.57 is computed using a mixed radix fast Fourier transform (FFT) algorithm.

5. *Smoothing*

   The periodogram is smoothed to reduce variability in the spectrum estimate (the estimates in equation 21.57 do not become less variable as the length of the series increases). However, smoothing also introduces bias in the estimates. There is a trade-off between the variability of the estimates and the bias. A thorough analysis might include inspecting the periodogram with several levels of smoothing. The smoothing that is performed on the periodogram is a sequence of running averages. The user can specify the lengths of modified Daniell windows to be run sequentially over the periodogram for the `spec.pgram` function. The `spec.pgram` function yields the smoothed estimate $\tilde{S}(f_k)$, expressed in decibels; that is, $10 \times log10\,\tilde{S}(f_k)$.

6. *Degrees of Freedom and Bandwidth*

   The degrees of freedom for a χ^2 approximation of the spectral density estimate at each Fourier frequency is also computed by `spec.pgram`. When there is no smoothing, tapering or padding, there are $n = 2$ degrees of freedom. The degrees of freedom n increase with the amount of smoothing.

   Bandwidth is a measure of the amount of smoothing. The formula for bandwidth used by the `spec.pgram` is

$$b_w = \frac{I}{\Delta n}\sum_{j=0}^{2k}\left[\left(\frac{1}{12} + (j-k)^2\right)a_j\right]^{1/2} \tag{21.58}$$

   where a_j, $j = 0, \ldots, 2k$ are the values of the smoothing filter (which is returned in the `filter` component with the index starting at zero) and $1/\Delta n$ is the interval between discrete Fourier frequencies. See Bloomfield (1976) for details.

Readers with no interest in multivariate time series may skip to the Example of Simple Use.

Cross Spectra Coherency and Phase

The *cross-spectrum* $S_{xy}(f_j)$ between two time series x_t and y_t at frequency f_j is approximately $EC_{xj}\overline{C}_{yj}$, where the C_{xj} and C_{yj} are given by the approximation 21.50, with an extra subscript (x or y) to distinguish coefficients for the two different series. One may think of this complex quantity as the complex covariance between C_{xj} and C_{yj}. The *phase* of x_t and y_t at frequency f_j is the argument (angle) of the cross spectrum $S_{xy}(f_j)$.

The *squared-coherency* $K(f_j)$ between x_t and y_t at frequency f_j is the squared modulus of the cross spectrum at f_j, normalized by the product of the two spectral densities $S_x(f_j)$ and $S_y(f_j)$:

$$K(f_j) = \frac{|S_{xy}(f_j)|^2}{S_x(f_j)S_y(f_j)}.$$

In view of 21.53, we have

$$K(f_j) \approx \frac{\left|EC_{xj}\overline{C}_{yj}\right|^2}{E|C_{xj}|^2 E|C_{yj}|^2} = |corr(C_{xj}, C_{yj})|^2$$

which provides the most natural interpretation of squared coherency as the square of the correlation between the random coefficients C_{xj} and C_{yj} of the

series x_t and y_t at frequency f_j.

Smoothing of the spectral estimates is mandatory for the estimation of coherency—if no smoothing is performed, the estimate is identically 1. See Priestley (1981). Similarly, the estimation of phase will be basically meaningless unless smoothing is done.

The `spec.pgram` function gives estimates of the squared-coherency and the phase for multivariate series. This output is in the form of matrices with each column being identified with a particular pair of univariate components of `x`. If j is less than k, then the column associated with the pair (j,k) is $(k - 1)(k - 2)/2 + j$.

Example of Simple Use

A spectral estimate of the square root of the sunspots data may be obtained with:

```
> srsun.sp <- spec.pgram(sqrt(sunspots),
+ spans=c(3, 5, 7, 9), detrend=F, demean=T)
```

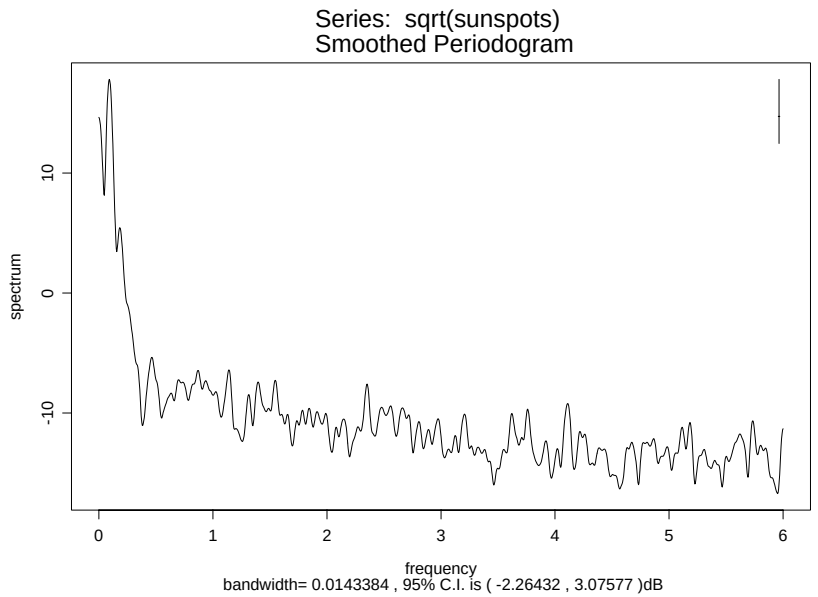

Figure 21.4: *Smoothed periodogram of the sunspot data*

The result is shown in figure 21.4. This subtracts the mean from the series but assumes that there is no trend. The spectrum is smoothed with a series of 4 running averages. By default ten percent on each end of the series has been tapered with a split cosine bell. The length of the series was automatically

padded from 2739 to 2744. A plot of the spectrum is automatically produced as a side effect (see function `spec.plot` for details).

Another simple example of the use of this function is:

```
> llynx <- log(lynx)
> ll.sp <- spec.pgram(llynx, taper=0)
```

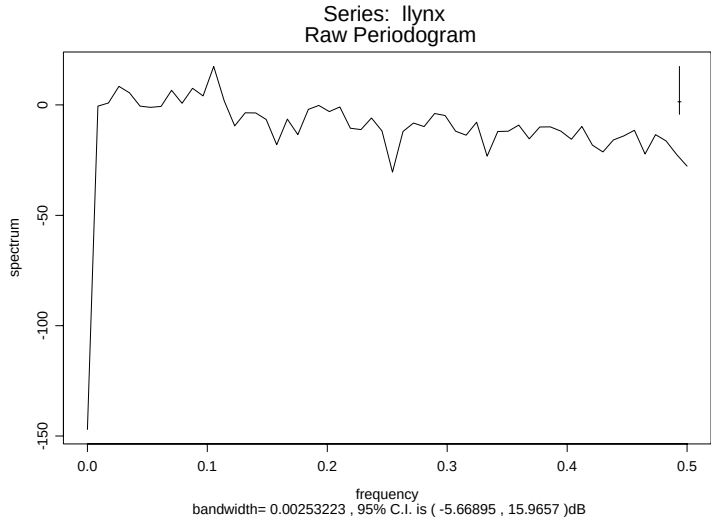

Figure 21.5: *Periodogram of the lynx data*

The result is shown in figure 21.5. This spectral estimate uses no tapering, and since it uses no smoothing it is the raw periodogram estimate. The data are detrended—allowing for the possibility that there is a linear trend in the data. Note that this is probably a poor spectral estimate for this dataset.

Below we analyze monthly CO_2 concentrations at Mauna Loa, Hawaii from January 1958 to December 1975. A `ts.plot` of the data reveals a strong linear trend and obvious cyclic behavior. Not surprisingly the cycles appear to be yearly. The analysis is shown in figure 21.6.

```
> par(mfrow=c(3,1)) # put three plots in the figure
> co.sp1 <- spec.pgram(co2)
> co.sp2 <- spec.pgram(co2, spans=c(9, 9))
> co.sp3 <- spec.pgram(co2, spans=c(3, 3, 3))
```

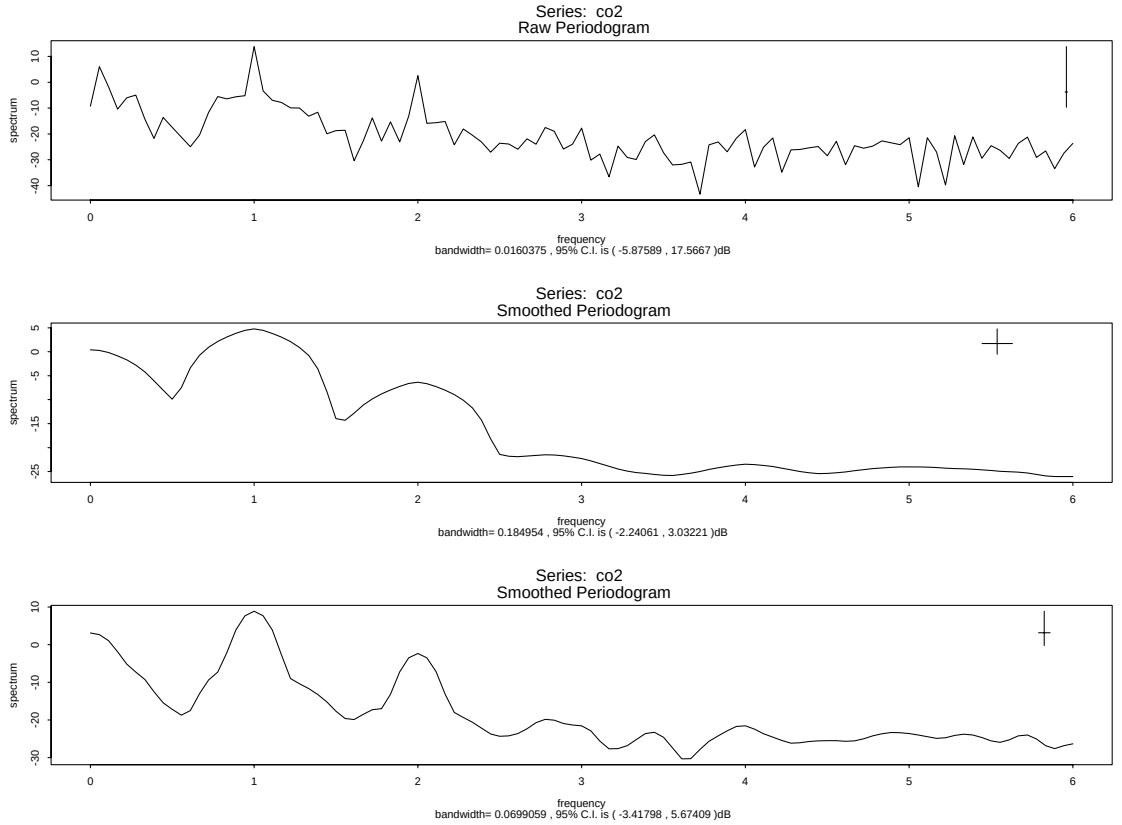

Figure 21.6: *Spectral estimates for the CO_2 data.*

Autoregressive Spectrum Estimation

An alternative spectral estimate to the smoothing of the periodogram is to estimate an autoregressive (or some other) model and use the spectrum of the estimated model as the spectral estimate.

The spectrum $S(f)$ of an autoregressive process with coefficients $\alpha_1, \ldots, \alpha_p$ is

$$S(f) = \frac{\sigma_\varepsilon^2}{\left|1 - \alpha_1 exp(-2\pi i f) - \cdots - \alpha_p exp(-2\pi i p f)\right|^2} \tag{21.59}$$

where f is the frequency in cycles per unit time and σ_ε^2 is the variance of the innovation process ϵ_t.

Phase and coherency may also be estimated for multivariate series. The S-PLUS function `spec.ar` computes the autoregressive spectrum of a time series.

Examples of Simple Use

```
> lynx.ar <- ar(log(lynx))
> lynx.spar <- spec.ar(lynx.ar, plot=T)
```

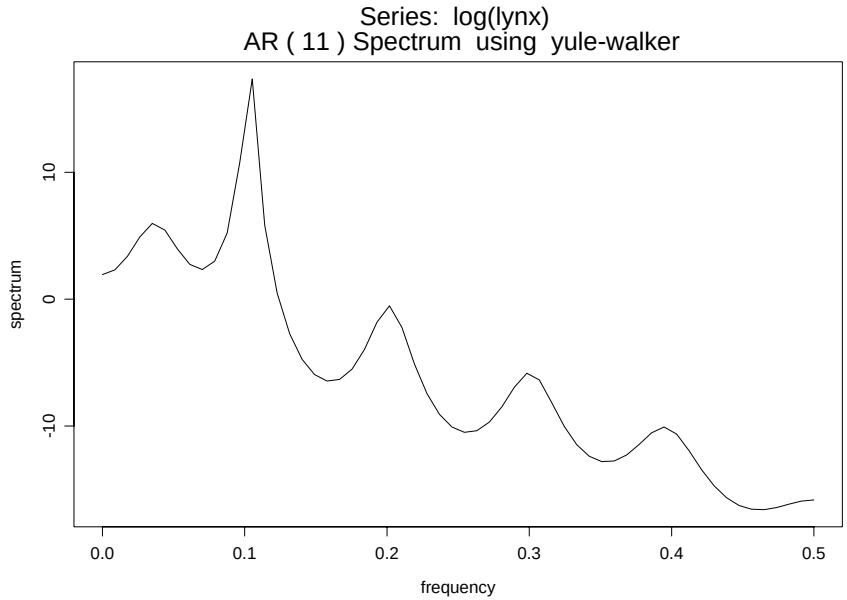

Figure 21.7: *Autoregression spectral estimate for the lynx data.*

The function `spectrum` can be used in the same way as `spec.pgram`. This function allows for different types of spectrum estimates. The function `spec.plot` can be used to plot the output of any spectrum estimation function.

Tapering

Tapering is a technique applied to time series to reduce the *leakage* phenomenon in spectral estimates. Leakage occurs when there is a large amplitude peak at a particular frequency f. Then the spectral estimates at frequencies near f can be higher than expected, and can easily obscure nearby lower amplitude peaks.

A *data taper* w_t, $0 \leq w_t \leq 1$, applied to a time series x_t produces a new tapered series.

$$\tilde{x}_t = w_t x_t \qquad t = 1, \ldots, n\,. \tag{21.60}$$

Typically the values of w_t are close to zero at the ends and close to one in the central part of the data.

The function `spec.taper` implements a split cosine bell taper. Let p be the portion to be tapered at each end of the series and n the length of the series, then for $m = np$ the split cosine bell taper is .

$$w_t = \begin{cases} \frac{1}{2}[1 - cos(\pi(t - 0.5)/m)] & t=1, \ldots, m \\ 1 & t=m+1, \ldots, n-m \\ \frac{1}{2}[1 - cos(\pi(n - t + 0.5)/m)] & t=n-m+1, \ldots, n. \end{cases} \quad (21.61)$$

Examples of Simple Use

```
> lynx.taper <- spec.taper(lynx)
> lynx.taper.5 <- spec.taper(lynx, .05)
```

All the values in `lynx.taper` are smaller than the corresponding value in `lynx`. In `lynx.taper.5`, five percent of the values on each end are tapered.

21.6 LINEAR FILTERS

The most important and widely used type of *filter* (referred to as a *digital filter* by engineers) is a linear time-invariant filter; that is, a filter in which the relationship between the input series x_t and the filtered output series is described by a constant coefficient linear difference equation. The class of linear time invariant (digital) filters has two primary types:

1. *Convolution* filters, which are usually called *finite-impulse response* (FIR) filters in the engineering literature, and *moving average* (MA) filters in the statistical literature.
2. *Recursive* filters, which are usually referred to as *infinite-impulse response* (IIR) filters in the engineering literature, and are called *autoregressive* (AR) filters in the statistics literature.

Convolution Filters

If x_t is the original series and $\boldsymbol{a} = (a_0, \ldots, a_q)$ is the set of filter coefficients, then the filtered series y_t is related to the original series x_t by the convolution equation.

$$y_t = \sum_{j=0}^{q} a_j x_{t-j} \qquad t = 0, \pm 1, \ldots . \quad (21.62)$$

We note that the filter is a "causal" in that each y_t is formed as a linear

combination of present and past x_t's, namely, x_t, x_{t-1}, ... , x_{t-q}. If one is dealing with a spatial series rather than a time series, or one is dealing with a time series in an "off-line" mode as opposed to a real-time application (as is usually the case for users of S-PLUS), then one can use the *non-causal* symmetric form of convolution filter

$$y_t = \sum_{j=-q/2}^{q/2} a_j x_{t-j} \tag{21.63}$$

where the filter coefficients are now $a_{-q/2}$, $a_{-q/2+1}$, …, a_0, a_1, …, $a_{q/2}$ with q an even integer. Usually in this case the a_j are symmetric; that is, $a_{-j} = a_j$, for $j = 1,..., q/2$.

Recursive Filters. A recursive filter uses an autoregressive-type recursion to transform the series. If x_t is the original series, and $\boldsymbol{a} = (a_0, \ldots, a_q)$ are the coefficients, then the filtered series y_t is obtained by the recursion.

$$y_t = \sum_{j=0}^{p} a_j y_{t-j-1} + x_t. \tag{21.64}$$

Examples of Simple Use Here are two examples using convolution filters:

```
> flynx <- filter(log(lynx), rep(.2,5))
> ts.plot(log(lynx), flynx)
> gaussfilt <- exp(-((-15:15)^2/7))
> gaussfilt <- gaussfilt/sum(gaussfilt)
> gflynx <- filter(log(lynx), gaussfilt)
> ts.plot(log(lynx), gflynx)
```

The resulting plots are shown in figures 21.8 and 21.9] .

The `flynx` structure is a simple equal weight moving average of the logarithm of the lynx data, while `gflynx` is filtered with a Gaussian filter.

Here is an an example using a recursive filter:

```
> set.seed(14) # set the seed to reproduce this example
> ar.sim <- filter(rnorm(500),c(.5,-.3, .35),"r",
+ init=rnorm(3))
> ar.sim <- ar.sim[101:500]
> ts.plot(ar.sim,main="AR(3) simulation")
```

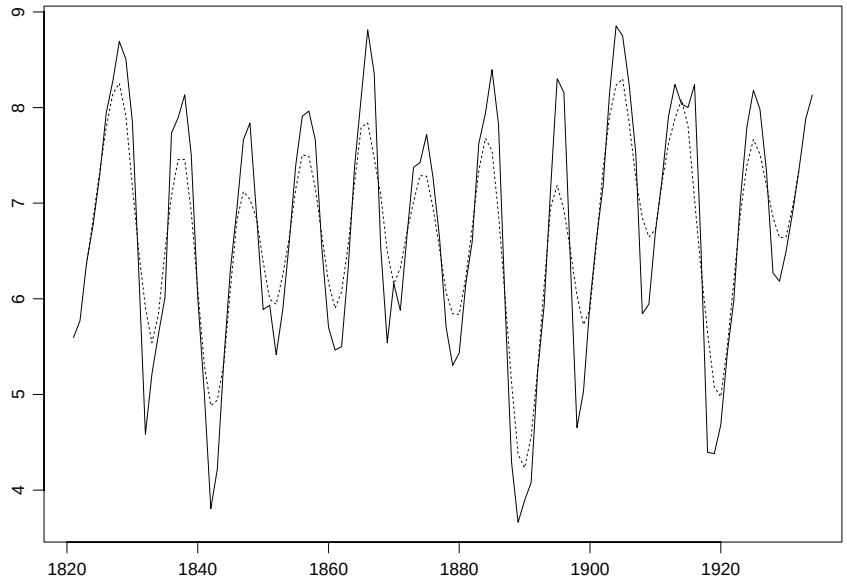

Figure 21.8: *Moving average of the lynx data.*

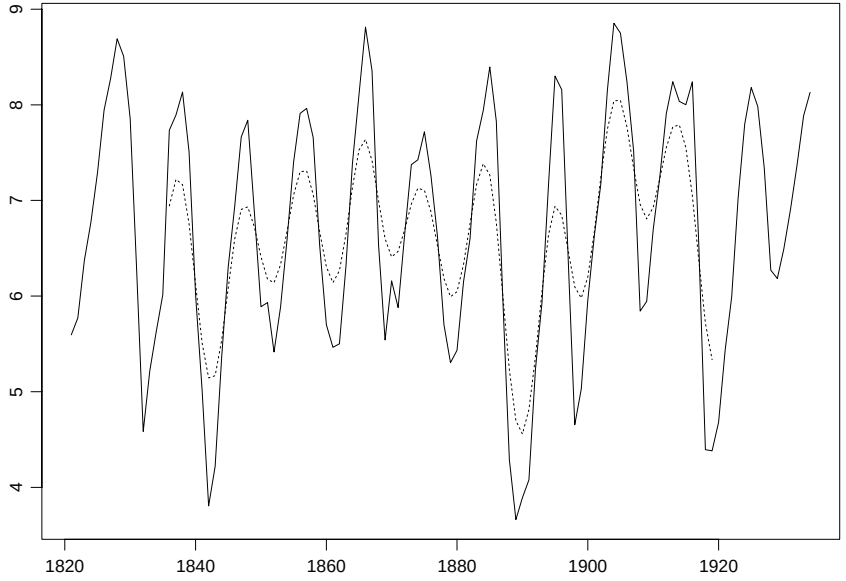

Figure 21.9: *Gaussian filtering of the lynx data.*

The above example is a simulation of an AR(3) process. The first part of the simulation is removed to more closely approximate a stationary process. The resulting plot is shown in figure 21.10 .

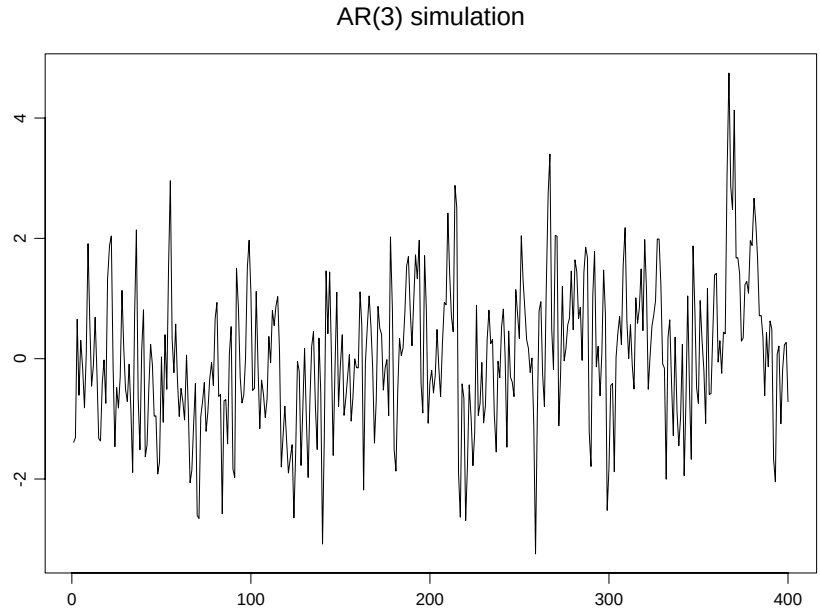

Figure 21.10: *Simulated autoregression.*

Complex Demodulation and Least Squares Low-Pass Filtering

Complex demodulation is a technique for analyzing a time series which does not assume stationarity. Inherent in the technique is the use of a low-pass filter. Hence these two topics are presented together. The function `demod` can be used not only to perform complex demodulation of a time series but also to generate a least squares low-pass filter with specific qualities.

Complex Demodulation.

Suppose that a time series x_t satisfies

$$x_t = R_t cos(\lambda t + \phi_t) + z_t \tag{21.65}$$

where R_t and ϕ_t are smooth processes (that is, they vary slowly over time) and z_t is a process without a component at frequency λ. R_t is the amplitude at time t of the periodic component with frequency λ, and ϕ_t is the phase at time t of this component.

Hence the model is of a series with an oscillation at some given frequency λ that changes slowly over time.

Equation 21.65 may be rewritten using complex numbers.

$$x_t = \frac{1}{2}R_t[e^{i(\lambda t + \phi_t)} + e^{-i(\lambda t + \phi_t)}] + z_t \tag{21.66}$$

Now the series is transformed into

$$y_t = x_t e^{-i\lambda t} = \frac{1}{2}R_t e^{i\phi_t} + \frac{1}{2}R_t e^{-i(2\lambda t + \phi_t)} + z_t e^{-i\lambda t}. \tag{21.67}$$

A smooth component of y_t will yield estimates of R_t and ϕ_t. The problem is to extract this component.

Least Squares Low-Pass Filtering

An ideal low-pass filter with cutoff frequency f_c has transfer function

$$H(f) = \begin{cases} 1 & \text{if } f \le f_c \\ 0 & \text{if } f > f_c. \end{cases} \tag{21.68}$$

That is, all frequencies less than f_c are left unchanged while no frequencies higher than f_c are allowed to pass through. Such an ideal filter does not exist, but it can be approximated arbitrarily well by using a sufficiently complex filter. A common approach is to design a fixed length filter using the least squares approximation method. The approximation will get better the longer the filter length. See Bloomfield (1976) for details.

Examples of Simple Use

In the commands below, the `lynx` data are demodulated at the peak frequency of the raw periodogram. The phase and amplitude of the demodulation are plotted separately.

```
> lynx.sp <- spectrum(log(lynx))
> lynx.pk <- lynx.sp$freq[lynx.sp$spec==max(lynx.sp$spec)]
> lynx.dem <- demod(log(lynx), lynx.pk, .05, .10)
> ts.plot(lynx.dem$phase, xlab="Time", ylab="Phase")
> ts.plot(lynx.dem$amp, xlab="Time", ylab="Amplitude")
```

Figure 21.11 shows the phase estimate of demodulation of the lynx data, while figure 21.12 shows the amplitude estimate of demodulation.

A method for obtaining a low-pass filter of length 50 with cutoff frequency 0.08 when the data are sampled at intervals of one time unit is shown below.

Figure 21.11: *Phase estimate in the demodulation of the lynx data.*

```
> filt50 <- demod(rnorm(200), .1, .08-1/49,
+ .08+1/49)$filter
```

Figure 21.12: *Amplitude estimate in the demodulation of the lynx data.*

21.7 ROBUST METHODS

Outliers in time series typically cause bias and an increase in the variability of conventional Gaussian maximum likelihood or least squares type estimates. Furthermore, unacceptably large biases may result even in large sample size situations when the fraction of outliers is not negligibly small. This problem occurs in particular for both the Yule-Walker and Burg methods of fitting autoregressions.

As a simple example, consider the Yule-Walker estimate of the first-order autoregression parameter $\hat{\phi}$ (which is also the lag 1 autocorrelation):

$$\hat{\phi} = \frac{\sum_{t=1}^{n-1} y_t y_{t-1}}{\sum_{t=1}^{n} y_t^2}.$$

Suppose that y_{t_0} is an outlier for some given time t_0, say $y_{t_0} = \xi$, with $|\xi|$ large. Then $\hat{\phi}$ will be "small," and in fact, $\hat{\phi} \to 0$ as $|\xi| \to \infty$, as is easily verified.

The robust procedures described in this section are designed to minimize the increased bias and variability due to outliers, either in isolation or in patches. We shall describe four functions, `ar.gm`, `acm.filt`, `acm.ave`, and `acm.smo`.

Typically, `ar.gm` and `acm.ave` will be used in conjunction. The `ar.gm` function provides initial robust autoregression parameter estimates which are used by the robust "smoother" algorithm `acm.ave`. The function `acm.smo` is an alternative robust smoother, and both `acm.ave` and `acm.smo` use the robust filter `acm.filt` as a basic building block.

We elaborate on our setup and terminology. Consider the general replacement (RO) type outliers model

$$y_t = (1 - z_t)x_t + z_t w_t \tag{21.69}$$

where x_t is a pth-order autoregression, z_t is a 0-1 process with probability 1-γ of being 1; that is, $P(z_t = 1) = 1\text{-}\gamma$, and w_t is a "contamination" process. Here γ is the fraction of contamination. This general replacement model contains the so-called additive outliers (AO) model

$$y_t = x_t + v_t \tag{21.70}$$

as a special case where $w_t = x_t + \tilde{v}_t$ with $v_t = 0$ when $z_t = 0$ and $v_t = \tilde{v}_t$ when $z_t = 1$. Although the methods described in this section work for the general replacement (RO) type outliers model, it is sometimes convenient for purposes of discussion to use the AO model. In doing so, we think in terms of the v_t having a contaminated normal distribution

$$F_v = (1 - \gamma)N(0, \sigma_0^2) + \gamma H,$$

where H is an arbitrary outlier-generating distribution. $N(0, \sigma_0^2)$ is the "nominal" Gaussian distribution of the additive noise v_t. In the context of 21.69 or 21.70, a *filter* $\hat{x}_t = \hat{x}_t(y_1, \ldots, y_t)$ is an estimate of the unobservable "signal" x_t which depends on the present and past observations $y_1, \ldots, y_t$ at time t. A *smoother* $\hat{x}_t = \hat{x}_t(y_1, \ldots, y_n)$ is an estimate of x_t which for each time $t = 1, \ldots, n$ depends upon all the observations $y_1, \ldots, y_n$. This is common terminology in the engineering literature. Both filters and smoothers often perform a "smoothing" operation in the sense that linear filters or smoothers are a weighted linear combination of $y_1, \ldots, y_t$ and $y_1, \ldots, y_n$, respectively, which often act approximately like local weighted means of the observations.

Robust filters and smoothers are nonlinear functions of the data which are designed to give good estimates of x_t in the presence of outliers generated by the model 21.69 or 21.70.

Although `acm.filt`, `acm.ave`, and `acm.smo` are capable of robust filtering and smoothing, respectively, for the case of σ_0^2 known and positive, neither these functions nor `ar.gm` are capable of estimating σ_0^2 from the data. (Estimation of σ_0^2 along with the autoregression parameters for x_t is a more difficult problem which we will hopefully address in future releases of S-PLUS.) Thus, we shall for the most part assume that $\sigma_0^2 = 0$. This corresponds to the frequently occurring situation where the autoregression x_t is observed perfectly a large fraction $1 - \gamma$ of the time and observed with additive outliers a fraction γ of the time.

In the case where $\sigma_0^2 = 0$, the values of x_t are observed perfectly a fraction $1 - \gamma$ of the time (that is, when $z_t = 0$ in 21.69 or when $v_t = 0$ in 21.70) but are unobservable a fraction γ of the time. For this case, we replace the terms robust filter and robust smoother by *robust filter-cleaner* and *robust smoother-*

cleaner, respectively. We often shorten these terms to simply *filter-cleaner* and *smoother-cleaner*.

A well-designed filter-cleaner has the following intuitively desirable property: For times at which $y_t = x_t$ by virtue of $v_t = 0$ (or $z_t = 0$) we will have $\ddot{x}_t = y_t$. This will occur a large fraction $1 - \gamma$ of the time. For times at which y_t is a gross outlier by virtue of v_t having a large magnitude, $\hat{x}_t$ will be a pure prediction based on the previous (filter) cleaned values $\hat{x}_1, \ldots, \ddot{x}_{t-1}$. A well designed smoother-cleaner behaves similarly except that at the time of occurrence of gross outliers, $\hat{x}_t$ is a pure interpolation based on all the other (smoother) cleaned data $\hat{x}_1, \ldots, \hat{x}_{t-1}, \hat{x}_{t+1}, \ldots, \hat{x}_n$.

In order to use a robust filter cleaner or smoother cleaner for autoregression models

$$x_t = \phi_1 x_{t-1} + \cdots + \phi_p x_{t-p} + \varepsilon_t \tag{21.71}$$

we must specify the unknown parameters $\phi_1, \phi_2, \ldots, \phi_p$, and s_ϵ, where s_ϵ is the scale parameter for the distribution F_ϵ for the *innovations* ϵ_t. In the case where $F_\epsilon = N(0, s_\epsilon^2)$, we have $s_\epsilon^2 = \sigma_\varepsilon^2$.

Since we seldom know the parameters $\phi_1, \phi_2, \ldots, \phi_p$, or s_ϵ, we must estimate them robustly from the data. This may be done using `ar.gm` which is a so-called generalized M-estimate or GM estimate (or bounded influence autoregression estimate) which is described in section on Generalized M-Estimates for Autoregression. The GM estimate produces robust parameter estimates $\ddot{\phi}_1, \ddot{\phi}_2, \ldots, \ddot{\phi}_p$, and $\ddot{s}_\varepsilon$ which may be used in any one of the robust filter or smoother functions `acm.ave`, `acm.filt`, and `acm.smo`.

Typically, one will use least squares autoregression model fitting (via `ar.yw` or `ar.burg`) to produce improved parameter estimates $\ddot{\phi}_1, \ddot{\phi}_2, \ldots, \ddot{\phi}_p, \ddot{s}$. These can in turn be used to run `acm.ave` again to obtain improved smoother-cleaned values and further improved least squares estimates of these autoregression parameters. Although one could iterate this procedure several times, we recommend using just one complete iteration of this form, which produces a *second* set of improved values $\hat{x}_1, \hat{x}_2, \ldots, \hat{x}_p, \hat{\phi}_1, \hat{\phi}_2, \ldots, \hat{\phi}_p, \hat{s}_\varepsilon$. (Because of the strongly nonlinear nature of `acm.ave`, further iteration can lead to poor solutions.)

Generalized M-Estimates for Autoregression

Generalized M-estimates (GM estimates) $\hat{\phi}$, $\hat{s}_\varepsilon$ of autoregression parameters $\phi^T = (\phi_1, \phi_2, \ldots, \phi_p)$ and innovations scale s_ϵ are obtained by solving the equations

$$\sum_{t=p}^{n-1} W(\boldsymbol{y}_t)\boldsymbol{y}_t w_t \cdot (y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}) = 0 \tag{21.72}$$

$$\sum_{t=p}^{n-1} \chi\left(\frac{y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}}{\hat{s}_\varepsilon}\right) = 0 \tag{21.73}$$

where the observed time series is $y_1, y_2, \ldots, y_n$, $\boldsymbol{y}_t^T = (y_t, y_{t-1}, \ldots, y_{t-p+1})$, χ is a bounded and continuous function, and $W(\boldsymbol{y}_t)$, w_t are nonnegative data-dependent weight functions. As we shall see below, w_t depends on $\hat{\phi}$ as well. We shall focus our description on 21.72, details concerning 21.73 being available in Martin (1980).

The equation 21.72 provides a linear weighted least squares estimate, linear in the case where the "big" weights $W(\boldsymbol{y}_t)$ and the "little" weights w_t are replaced by fixed weights; that is, weights independent of both the data $\boldsymbol{y}_t$ and the estimate $\hat{\phi}$. Because the w_t (but not $W(\boldsymbol{y}_t)$) depend upon $\hat{\phi}$, the equations 21.72 are nonlinear. They are solved by an iterative weighted least squares method:

$$\sum_{t=p}^{n-1} W(\boldsymbol{y}_t)\boldsymbol{y}_t w_t^j \cdot (y_{t+1} - \boldsymbol{y}_t^T\hat{\phi}^{j+1}) = 0 \qquad j = 0, 1, \ldots, \textbf{\textit{iter}} \tag{21.74}$$

where ***iter*** is the desired number of iterations, starting with the least squares estimate $\hat{\phi}^0 = \hat{\phi}_{L.S.}$. The equation 21.73 is also iterated, yielding estimate \widehat $\hat{s}_\varepsilon^j$ at iteration j.

The big weights $W(\boldsymbol{y}_t)$ are constructed so that $W(\boldsymbol{y}_t)\boldsymbol{y}_t$ is bounded and continuous, and the little weights w_t are constructed so that $w_t \cdot (y_{t+1} - \boldsymbol{y}_t^T\hat{\phi})$ is bounded and continuous. This achieves the basic requirement for robustness that the summands of the estimating equation 21.72 be bounded

and continuous. Specifically, the weights w_t^j are obtained from a psi-function ψ_c, with tuning constant c, as follows:

$$w_t^j = \frac{\hat{s}_\varepsilon^j \psi_c((y_{t+1} - \boldsymbol{y}_t^T \hat{\phi}^j)/\hat{s}_\varepsilon^j)}{y_{t+1} - \boldsymbol{y}_t^T \hat{\phi}^j}.$$

Two types of psi-functions are used, namely Huber's (Huber (1964)) favorite psi:

$$\psi_{H,\,chr}(r) = \begin{cases} r & |r| < c \\ c \cdot sgn(r) & |r| \geq c \end{cases}$$

and Tukey's bisquare functions (see Mosteller and Tukey, (1977)):

$$\psi_{B,\,cbr}(r) = \begin{cases} r(1-r^2)^2 & |r| \leq c \\ 0 & |r| > c. \end{cases}$$

The separate tuning constants *chr* and *cbr* for the ψ function applied to residuals are adjusted to obtain a compromise between high efficiency when the data are actually Gaussian, and robustness towards outliers.

The "big" weights $W(\boldsymbol{y}_t)$ are also derived from a psi function of either the Huber or Tukey type. As a default, `ar.gm` uses the Tukey type psi-function. Details concerning the formation of the weights $W(\boldsymbol{y}_t)$ may be found in Martin (1980).

The main ideas behind the choice of big weights and little weights is as follows. The use of the default choice of basing the big weights $W(\boldsymbol{y}_t)$ on the Tukey bisquare is that when $\boldsymbol{y}_t$ is not too large, $W(\boldsymbol{y}_t)$ will be close to one and therefore have little effect, but when $\boldsymbol{y}_t$ is "very large" (that is, when $\boldsymbol{y}_t$ is a gross outlier in the vector sense), $W(\boldsymbol{y}_t)$ will be zero and $\boldsymbol{y}_t$ will have no influence on the estimate $\hat{\phi}$.

Similar comments apply when w_t is based on the Tukey bisquare. When the residual $r_t = (y_{t+1} - \boldsymbol{y}_t^T \hat{\phi})$ is not too large, w_t will be close to one, whereas when $|r_t|$ is "very large;" for example, when y_{t+1} is a gross outlier, w_t will be zero.

The only difficulty is that when w_t is based on the Tukey bisquare $\psi_{B,cbr}$, the equations 21.72 have multiple solutions and starting the iteration 21.74 with least squares might lead to a poor solution. This difficulty is avoided when w_t

is based on the Huber psi-function $\psi_{H,chr}$, since then 21.72 has an essentially unique solution. However, basing w_t on $\psi_{H,chr}$ does not result in as much robustness toward large outliers as does basing w_t on $\psi_{B,cbr}$. Thus the strategy adopted is to iterate 21.74 a number of times *iterh* using w_t based on the Huber psi-function, followed by a number of iterations *iterb* using w_t based on the Tukey psi-function.

Examples of Simples Use

```
> robar <- ar.gm(bicoal.tons, 2)
```

Robust Filtering

First consider the special case where x_t in 21.70 is an AR(1) process with known parameter ϕ. In this case the robust filtering algorithm is given by

$$\hat{x}_t = \phi\hat{x}_{t-1} + \frac{m_t}{s_t}\psi\left(\frac{y_t - \phi\hat{x}_{t-1}}{s_t}\right)$$

where s_t is a measure of scale for the observation prediction residuals $r_t = y_t - \phi\hat{x}_{t-1}$. The quantity s_t is computed using an auxiliary data-dependent recursion. (See Martin (1981) for details). The psi-function we use is the Hampel 2-part redescending type:

$$\psi_{HA,\,a,\,b}(r) = \begin{cases} r & |r| \le a \\ sgn(r)\dfrac{a}{b-a}(b - |r|) & a < |r| \le b \\ 0 & |r| > b. \end{cases}$$

The robust filter has the property that if y_t is a gross outlier large enough that the scaled residual $(y_t - \phi\hat{x}_{t-1})/R_t$ is larger in absolute value than b, then $\hat{x}_t$ is a pure prediction based on the previous robust filter value: $\hat{x}_t = \phi\hat{x}_{t-1}$.

Now consider the case where x_t is a pth order autoregression. In this case, x_t may be represented in state space form

$$\boldsymbol{x}_t = \boldsymbol{\Phi}\boldsymbol{x}_{t-1} + \varepsilon_t$$

where $\varepsilon_t^T = (\epsilon_t, 0, \ldots, 0)$ and $\boldsymbol{x}_t^T = (x_t, x_{t-1}, \ldots, x_{t-p+1})$ are p-dimensional vectors, and

$$\mathbf{\Phi} = \begin{bmatrix} \phi_1 & \phi_2 & \dots & \phi_p \\ 1 & 0 & \dots & 0 \\ 0 & 1 & 0 & 0 \\ \vdots & & & \vdots \\ 0 & \dots & 0 & 1 \end{bmatrix}$$

is the so-called state transition matrix. In this case the robust filter value of time t is

$$\hat{x}_t = (\hat{\hat{\boldsymbol{x}}}_t)_1$$

namely the first component of the *vector* filtered value $\hat{\boldsymbol{x}}_t$ obtained from the recursion

$$\hat{\boldsymbol{x}}_t = \mathbf{\Phi}\hat{\boldsymbol{x}}_{t-1} + \frac{\boldsymbol{m}_t}{\boldsymbol{s}_t}\boldsymbol{\Psi}\left(\frac{y_t - \hat{y}_t^{t-1}}{s}\right)$$

where $\tilde{\boldsymbol{m}}_t$ is obtained from an auxiliary data-dependent recursion (see Martin and Thomson (1982), for details), and

$$\hat{y}_t^{t-1} = (\mathbf{\Phi}\hat{\boldsymbol{x}}_{t-1})_1$$

is the first component of the vector one-step-ahead prediction $\mathbf{\Phi}\hat{\hat{\boldsymbol{x}}}_{t-1}$.

In the usual case where we can use `acm.filt` as a filter-cleaner by setting σ_0 (= s_0 below) equal to zero, it turns out that

$$\tilde{\boldsymbol{m}}_t^T = (s_t, 0, \dots, 0)\,.$$

Then it is easy to check that when the Hampel two-part psi-function $\Psi_{HA,a,b}$ is used and y_t is a "good" datapoint by virtue of $(y_t - \hat{y}_t^{t-1})/s_t$ being less than a in magnitude, then $\hat{\hat{x}}_t = y_t$, so y_t is not altered if it is "good". This will usually be the case for most of the data points when `acm.filt` is used in the filter-cleaner mode.

Examples of Simple Use

```
> gm <- ar.gm(bicoal.tons, 3)
> bicoal.filt <- acm.filt(bicoal.tons, gm)
```

Two-Filter Robust Smoother

The robust smoother `acm.ave` is constructed using two `acm.filt` robust filters, one "forward" filter $\hat{x}_t^+$ going forward in time over the data and one "backward" filter $\hat{x}_t^-$ going backward in time over the data.

Let $\hat{x}^{-}_{t+1,t}$ denote the backward one-step-ahead predictor of x_t given the data $y_{t+1}, \ldots, y_n$. Let p^{+}_{t} denote conditional mean squared error, conditioned on $y_1, \ldots, y_t$, for filtering for the forward filter (this is computed in `acm.filt`). And let m^{-}_{t} be the conditional mean-squared error, conditioned on $y_{t+1}, \ldots, y_n$, for predicting x_t for the backward filter (this is also computed in `acm.filt`). Then the robust smoother $\hat{x}^{n}_{t}$ is obtained by confining $\hat{x}^{+}_{t}$ and $\hat{x}^{-}_{t+1,t}$ in the natural Bayesian way

$$\hat{x}^{n}_{t} = \frac{m^{-}_{t} x^{+}_{t} + p^{+}_{t} x^{-}_{t+1,t}}{p^{+}_{t} + m^{-}_{t}}.$$

This smoother has the following characteristics when used as a smoother-cleaner by setting $\sigma_0 = s_0 = 0$; "Good" data points y_t are left unaltered, while gross outliers y_t are replaced by interpolates based on the cleaner data $\hat{x}_1, \ldots, \hat{x}_{t-1}, \hat{x}_{t+1}, \ldots, \hat{x}_n$.

Examples of Simple Use

```
> gm <- ar.gm(bicoal.tons, 3)
> bicoal.smo <- acm.ave(bicoal.tons, gm)
```

Alternative Robust Smoother

The alternative robust smoother `acm.smo` is an approximate conditional mean type robust smoother. For details, see Martin (1979).

Examples of Simple Use

```
> gm <- ar.gm(bicoal.tons, 3)
> bicoal.smo <- acm.smo(bicoal.tons, gm)
```

21.8 REFERENCES

Ansley, C. F. (1979). An algorithm for the exact likelihood of a mixed autoregressive-moving average process. *Biometrika* 66: 59–65.

Bell, W. and Hillmer, S. (1987). Initializing the Kalman filter in the non-stationary case. Research Report CENSUS/SRC/RR-87/33, Statistical Research Division, Bureau of the Census, Washington, DC, 20233.

Bloomfield, P. (1976). *Fourier Analysis of Time Series: An Introduction*. *Wiley,* New York.

Box, G. E. P. and Jenkins, G. M. (1976). *Time Series Analysis: Forecasting and Control.* Holden-Day, Oakland, CA.

Bruce, A. and Martin, R. D. (1989). Leave-k-out diagnostics for time series. *Journal of the Royal Statistical Society, Series B/* 51:363–401

Burg, J. P. (1967). Maximum Entropy Spectral Analysis. Paper presented at the 37th Annual International S. E. G. Meeting, Oklahoma City, OK.

Chatfield, C. (1984). *The Analysis of Time Series: An Introduction*, 3rd ed. Chapman and Hall, London.

Dennis, J. E., Gay, D. M. and Welsch, R. E. (1980). An adaptive nonlinear least-squares algorithm. *ACM Transaction Mathematical Software* 7:348–383.

Harvey, A. C. and Pierse, A. G. (1984). Estimating missing observations in economic time series. *Journal of the American Statistical Association* 79:125–131.

Haslett, J. and Raftery, A.E. (1989). Space-time modelling with long-memory dependence: Assessing Ireland's wind power resource (with Discussion). *Journal of the Royal Statistical Society, series C—Applied Statistics*, 38:1–50.

Huber, P. J. (1964). Robust estimation of a location parameter. *Annals of Mathematical Statistics*, 35:73–101.

James, D. A., and Pregibon, D. (1992). Chronological Objects in S. AT&T Technical Report. AT&T Bell Laboratories, Muray Hill, NJ 07974.

Jones, R. H. (1980). Maximum likelihood fitting of ARIMA models to time series with missing observations. *Technometrics* 22:389–395.

Kohn, R. and Ansley, C. F. (1985). Efficient estimation and prediction in time series regression models. *Biometrika*, 72:694–697.

Kohn, R. and Ansley, C. F. (1986). Estimation, prediction, and interpolation for ARIMA models with missing data. *Journal of the American Statistical Association*, 81:751–761.

Mandelbrot, B.B. (1977). *Fractals: Form, Chance and Dimension*. Freeman,

San Francisco.

Martin, R. D. (1979). Approximate conditional mean type smoothers and interpolators. In *Smoothing Techniques for Curve Estimation*, pp. 117–143. T. Gasser and M. Rosenblatt, eds. Springer Verlag, Berlin.

Martin, R. D. (1980). Robust estimation of autoregressive models. In *Directions in Time Series*, pp. 228–254. D. R. Brillinger and G. C. Tiao, eds. Institute of Mathematical Statistics, Hayward, CA.

Martin, R. D. (1981). Robust methods for time series, pp. 683–759. In *Applied Time Series Analysis.* D. F. Findley, ed. Academic Press, New York.

Martin, R. D. and Thomson, D. J. (1982). Robust resistant spectrum estimates. *Proceedings of the IEEE*, 70:1097–1115.

Mosteller, F. and Tukey, J. W. (1977). *Data Analysis and Regression.* Addison-Wesley, Reading, MA.

Priestley, M. B. (1981). *Spectral Analysis and Time Series.* Academic Press, London.

Shumway, R. H. (1988). *Applied Statistical Time Series Analysis.* Prentice Hall, Englewood Cliffs, NJ.

Singleton, R. C. (1969). An algorithm for computing the mixed radix fast Fourier transform. *IEEE Transactions on Audio and Electronics*, Au-17:93–103.

Whittle, P. (1983) *Prediction and Regulation by Linear Least-Square Methods*, 2nd ed. University of Minnesota Press, Minneapolis.

OVERVIEW OF SURVIVAL ANALYSIS

22

S-PLUS functions are available for modeling survival times using parametric and nonparametric approaches.

OVERVIEW OF SURVIVAL ANALYSIS

The term *survival analysis* originated in the study and analysis of times to death (that is, survival times) for medical patients diagnosed with some fatal disease. Survival analysis is now a well-developed field of statistical research and methodology pertaining to modeling and testing hypotheses of *failure time data* for humans as well as animals, machines, electronic equipment, automobile components, etc. Hence, the methodology is far more general than the analysis of survival times. In fact, fields of study other than medicine have given other names to the identical methodology discussed here. This chapter might just a well have been called any one of the following:

- Analysis of Failure Time Data
- Reliability Analysis
- Event History Analysis

However, because of the focus of most of the examples and because of the history of the development of this material we call it *Survival Analysis*. This helps to simplify the presentation. In examples, we will simply refer to *patients* (or people or subjects) and their *survival* times. You can substitute the appropriate terminology for your field of study as you read if you wish.

Modeling of survival times is based on two distinct approaches—parametric and nonparametric. The material in this and the following chapters covers both approaches. The addition of parametric survival models extends the functionality of earlier versions of S-PLUS to include methods that predate the nonparametric methods but are still widely used in industrial and manufacturing settings where estimation of component and system reliability may require extrapolation from accelerated tests. The nonparametric methods, widely used in clinical trials, include Kaplan-Meier estimates of survival, Cox proportional hazards regression models and extensions due to Andersen and Gill (1982). Miller (1981) and Kalbfleisch and Prentice (1980) are excellent references.

22.1 OVERVIEW OF S-PLUS FUNCTIONS

Survival analysis in S-PLUS Version 3.3 is based on the **survival4.1**[1] StatLib entry produced by Terry Therneau of the Mayo Clinic. It differs only slightly from the version 4.1 code found in StatLib. The expected survival routines have been modified to use `"dates"` objects for dates, and there have been

1. Copyright © 1994, Mayo Foundation for Medical Education and Research. All Rights Reserved.

some minor bug fixes and enhancements. Terry Therneau has been an important contributor to this documentation of survival analysis in S-PLUS. A companion document which discusses more fully some of the topics presented here can be retrieved from StatLib as **survival4.doc**.

Table 22.1 displays the relative differences between the survival analysis

Table 22.1: *Survival Analysis functions in S-PLUS Version 3.3, or higher, compared to S-PLUS Version 3.2*

Function Description	Version 3.3 or higher	Version 3.2
Parametric regression models	`survreg`	None
Cox proportional hazards models	`coxph`	`coxreg`
Andersen-Gill extension toCox models	`coxph`	`agreg`
Formula representation for failure/status	`Surv`	None
Fit a survival curve	`survfit`	`surv.fit`
Compare survival curves	`survdiff`	`surv.diff`
Compute a test for proportional hazards	`cox.zph`	None
Plot a survival curve	`plot.survfit`	`plot.surv.fit`
Plot method for Cox model	`plot.coxph`	None
Predicted survival for age-sex matched cohort	`survexp`	None
Handling of missing values	`naresid`, `naprint`	None

functions in S-PLUS Version 3.3, or higher, compared to Version 3.2. Survival analysis in Version 3.3 is a considerable enhancement to that in Version 3.2. New functionality includes parametric survival models, formula-based model specification, a test for proportional hazards in a Cox model, and predicted survival based on an age and sex matched cohort from a known population. All of the Version 3.2 functionality is maintained in Version 3.3 but the function names have changed. In somes cases functionality has been combined into fewer S-PLUS functions. For example, the functionality of `coxreg` and `agreg` in Version 3.2 is now contained in the single function `coxph`. The old survival functions are now deprecated. They will be removed from S-PLUS in some future version after 3.3.

In this section we present a brief overview of the functions used for doing survival analysis in S-PLUS. This section provides an overview of the type of

computations, model fitting, and graphical displays available for doing survival analysis in S-PLUS. More in depth information is contained in the chapters that follow.

Survival Curve Estimates

The function `survfit` fits a Kaplan-Meier or a Fleming-Harrington survival curve or computes the predicted survival curve for a Cox proportional hazards model.

Examples

- Simple Kaplan-Meier estimate

```
survfit(Surv(time, status), data = leukemia)
```

- Print the survival curve estimate, standard errors, and confidence intervals.

```
summary(survfit(Surv(time, status), data = leukemia))
```

- Fleming-Harrington estimate

```
survfit(Surv(time, status), data = leukemia, type =
+ "fleming-harrington")
```

- Kaplan-Meier estimate with two groups

```
survfit(Surv(time, status) ~ group, data = leukemia)
```

- Predicted survival at the average predictor for a Cox model

```
survfit(coxph(Surv(futime, fustat) ~ age, data =ovarian))
```

- Predicted survival at other than the average predictor for a Cox model.

```
survfit(coxph(Surv(futime, fustat) ~ age, data =
+ ovarian), newdata = data.frame(age = 70))
```

Important Options

Kaplan-Meier or Fleming-Harrington estimate of survival.

Greenwood or Tsiatis variance estimate.

Comparing Kaplan-Meier Survival Curves

The function `survdiff` computes one and k-sample versions of the Fleming-Harrington G^{ρ} family of tests. This includes the log-rank and Gehan-Wilcoxon tests as special cases.

Examples

- Test for the presence of a separate baseline survival for each sex.

```
survdiff(Surv(time, status) ~ sex, data = lung)
```

- One-sample test

```
pred <- survexp(time ~ ratetable(sex = sex, year = 1970,
+      age = age * 365.25), data = lung, cohort = F)
survdiff(Surv(time, status) ~ offset(pred), data = lung)
```

Cox Proportional Hazards Models

The function coxph fits a Cox proportional hazards model.

Examples

- Standard Cox model

```
coxph(Surv(time, status) ~ group, data = leukemia)
```

- Time dependent data.

```
coxph(Surv(start, stop, event) ~ (age + surgery) *
+      transplant, data = heart)
```

- Stratified model, with a separate baseline per institution, and institution specific effects for sex.

```
coxph(Surv(time, status) ~ strata(sex) * age, data=lung)
```

- Force in a known term, age, without estimating a coefficient for it.

```
coxph(Surv(time, status) ~ offset(age) + sex, data=lung)
```

Important Options

Breslow, Efron, or exact partial likelihood methods for handling ties.

cox.zph computes a test of proportional hazards for the fitted Cox model, and estimates of time-dependent coefficients suitable for graphing.

Examples

- Compute proportional hazards test for fitted model.

```
cox.zph(coxph(Surv(time, status) ~ age + sex + ph.ecog,
+      data = lung, na.action = na.omit))
```

- Display the estimated coefficients as a function of time.

```
plot(cox.zph(coxph(Surv(time, status) ~ age + sex +
+      ph.ecog, data = lung, na.action = na.omit)))
```

Important Option

Global test in addition to the tests for each covariate.

Parametric Survival Models

The function `survreg` fits a parametric survival model.

Examples

- Fit a Weibull distribution. The default link function is `log`.

```
survreg(Surv(days, event) ~ voltage, data = capacitor)
```

- Fit an extreme value distribution.

```
survreg(Surv(days, event) ~ voltage, link = "identity",
+  data = capacitor)
```

- Fit a log-logistic distribution.

```
survreg(Surv(days, event) ~ voltage, dist = "logistic",
+  data = capacitor)
```

Important Options

Distributions: smallest extreme value, Weibull, logistic, log-logistic, normal , log-normal, exponential, or Rayleigh.

Fix the scale parameter.

Predicted Survival

The function `survexp` predicts survival for an age and sex matched cohort of subjects given a baseline matrix of known hazard rates for the population. Most often these are U.S. mortality tables. Also, a prior Cox model can act as the rate table.

Examples

- Average conditional cohort survival, defaults to U.S. white.

```
survexp(time ~ ratetable(sex = sex, year = 1970, age = age
+  * 365.25), conditional = T, data = lung)
```

- Data to enter into a one sample test for comparing the given group to a known population.

```
pred <- survexp(time ~ ratetable(sex = sex, year = 1970,
+  age = age * 365.25), data = lung, cohort = F)
```

Important Options

Matrix of known hazards: U.S., Arizona, Florida, and Minnesota are included.

Estimates of `"individual"` or `"cohort"` expected survival.

Utility Functions

`Surv` is a *packaging* function; like `I` and `C` it doesn't transform its argument. This is used for the left hand side of all formulas used by the survival model fitting functions.

Examples

- Right censored data with `status = 1` for death and `status = 0` for censored.

```
Surv(time, status)
```

- Right censored data, a value of 3 corresponds to a death

```
Surv(time, status == 3)
```

- Counting process data, as in the `agreg` function of Version 3.2.

```
Surv(start, stop, event)
```

- Left censored data

```
Surv(time, status, type = "left")
```

`naresid`, `naprint` provide a new way for handling missing values.

- Can specify a global `NA` action through the *options* list. For example:

```
options(na.action = "na.omit")
```

- The `print` methods label the output of the action taken. For example, when `na.omit` is the action a message similar to the following is printed with the fit object:

```
"14 observations deleted due to missing".
```

- `NA`s are inserted in prediction and residual vectors so they match the length of the original data. This makes, for example, the plotting of residuals versus the original variables easier.

`strata` marks a variable or group of variables as strata.

- If there are multiple variables, each unique combination forms a stratum.

Examples

These examples use the variables in the `ovarian` data frame.

- Specify `rx` as a stratification variable.

```
strata(rx)
```

- Specify `rx` and `residual.dz` as stratification variables.

```
strata(rx, residual.dz)
```

- Make `NA` a separate group rather than omitting `NA`.

```
strata(rx, na.group=T)
```

- `cluster` identifies correlated groups of observations, and is used on the right-hand side of a formula. For example:

```
cluster(group)
```

22.2 MISSING VALUES

The handling of missing values (NA) for the survival analysis functions has been enriched as outlined in the Utility Functions section of section 22.1, Overview of S-PLUS Functions. The main improvements follow:

1. You can specify a global default function for handling missing values. This frees you from having to do it in the call to the model fitting function. For example, to set the global missing value action to delete missing values row-wise you do

   ```
   > options(na.action = "na.omit")
   ```

2. A brief report of the action taken is included when printing a fitted model. For example, if `na.omit` is the action, a message something like the following will be included when the fit object is printed:

   ```
   "14 observations deleted due to missing".
   ```

3. When residuals and predictions are computed, NAs are appropriately inserted so that the resulting vectors are the same length as the original variables. This allows you to plot, for example, the residuals versus the predictors without having to worry about the residual vector being a different length than the original data. Because of this feature you can do the following:

   ```
   fit <- coxph(Surv(time, status) ~ age + sex + ph.ecog
      + ph.karno, data = lung, na.action =na.omit)
   plot(lung$age, residuals(fit))
   ```

Warning

Specifying a global default for handling NAs through the options list effects all the model fitting functions that call `model.frame.default` (which is most of them). The `tree` function doesn't, so it is immune to the global setting. However, virtually all the rest of the model fitting functions do call `model.frame.default`, so the global setting will be in effect for those functions. It is known that there are some side effects (errors produced) when fitting generalized additive models (the `gam` function). Because the global action for handling NAs has not been thoroughly tested for all the fitting functions, it is recommended that you provide the NA action function (e.g., `na.omit`) as the `na.action` argument to the fitting function rather than rely on the global action.

Additionally, if you fit a survival model relying on a global NA action and use the fitted model in later computations, errors and/or incorrect values can result if the global NA action is different than at the time of fitting the model. If you expect to change the global NA action, it is safer to provide the NA action function as the `na.action` argument to the fitting function rather than as a global action.

22.3 REFERENCES

Andersen, P. K. and Gill, R. D. (1982). *Cox's regression model for counting processes: A large sample study.* Annals of Statistics, 10:1100–1120.

Kalbfleisch, J. and Prentice, R. L. (1980). *The Statistical Analysis of Failure Time Data*. Wiley, New York.

Miller, Rupert G. (1981). *Survival Analysis*, pp. 49–50. Wiley, New York.

ESTIMATING SURVIVAL 23

This discussion of estimating the survival distribution, S(T), will familiarize you with the methods.

ESTIMATING SURVIVAL

A survival function defined over time t is, by definition, the probability that a person survives *at least* to time t. More formally, let T be a positive random variable with distribution function $F(t)$ and density $f(t)$. The survival function $S(t)$ is

$$S(t)= 1 - F(t) = P\{T>t\},$$

and the hazard rate or hazard function $\lambda(t)$

$$\lambda(t) = \frac{f(t)}{S(t)}.$$

The hazard rate has the interpretation $\lambda(t)$ = P{patient dies in the next small unit of time, Δt, given they have survived to time t}. A constant hazard indicates that, over each interval, a constant proportion of surviving subjects is expected to die. A familiar example is radioactive decay, where the "death" of a molecule corresponds to its decay. Constant hazard may also be associated with some fatal diseases, such as metastatic cancer.

The cumulative hazard $\Lambda(t)$ is defined as

$$\Lambda(t) = \int_0^t \lambda(t)dt = -\log S(t).$$

What distinguishes survival analysis from most other statistical methods is the presence of *censoring*. In a study of survival following two different treatment regimens, for example, analysis of the trial typically occurs well before all of the patients have died. For those still alive at the time of analysis, the true survival time is known only to be greater than the time observed to date. Such an observation is said to be *censored*. Survival data is presented to the computer program as a pair (t_i, δ_i), where t_i is the observed survival time and $\delta_i = 0$ if the observation is censored, $\delta_i = 1$ if a death is observed. Survival data is often presented using a + for the censored observation, so that a set of times might be 8, 11+, 14, 22, 36+, etc.

Let $t_1^* < t_2^* < \cdots < t_m^*$ denote the m *distinct* death times. Let $Y_i(s)$ be an indicator function which is 1 if person i is still at risk at time s and 0 otherwise, that is, $Y_i(s) = 1$ if $s \leq t_i^*$. Then the number at risk at time s is $r(s) = \sum_1^n Y_i(s)$. We can similarly define $d(s)$ as the number of deaths occurring at time s.

In order to discuss some of the more recent methods in survival analysis, it is helpful to recast the problem as a counting process, a notation found in Andersen and Gill (1982) and others. A good reference is Fleming and Harrington (1981). Let $N_i(t)$ be a counting process associated with the ith subject, so N_i increases by 1 at each observed event (for example, heart attack). In this notation a subject can have multiple events. $Y_i(t)$ is an indicator function as before, but now can have multiple transitions from 0 (zero) to 1 (one), with a subject entering and leaving the risk set.

23.1 KAPLAN-MEIER ESTIMATOR

The most common estimate of the survival distribution, the Kaplan-Meier (KM) estimate, is a product of survival probabilities

$$\hat{S}_{KM}(t) = \prod_{t_i < t} \frac{r(t_i) - d(t_i)}{r(t_i)},$$

where r and d are the number at risk and the number of deaths, respectively, as defined above. Graphically, the Kaplan-Meier survival curve appears as a step function with a drop at each death. Censoring times are often marked on the plot as "+" symbols.

Example: AML Study

The data presented in table 23.1 are preliminary results from a clinical trial to evaluate the efficacy of maintenance chemotherapy for acute myelogenous leukemia (AML). The study was conducted by Embury *et al.* (1977) at Stanford University. After reaching a status of remission through treatment by chemotherapy, the patients who entered the study were assigned randomly to two groups. The first group received maintenance chemotherapy; the second, or control, group did not. The objective of the trial was to see if maintenance chemotherapy prolonged the time until relapse.

Table 23.1: *Data for AML maintenance study. A + indicates a censored value.*

Group	Length of complete remission (in weeks)
Maintained	9, 13, 13+, 18, 23, 28+, 31, 34, 45+, 48, 161+
Non-maintained	5, 5, 8, 8, 12, 16+, 23, 27, 30, 33, 43, 45

The Kaplan-Meier estimator of survival for the maintained group is computed by hand as follows:

$$
\begin{aligned}
S(0) &= 1, \\
S(9) &= S(0) \times \frac{10}{11} = 0.91, \\
S(13) &= S(9) \times \frac{9}{10} = 0.82, \\
S(18) &= S(13) \times \frac{7}{8} = 0.72, \\
S(23) &= S(18) \times \frac{6}{7} = 0.61, \\
S(28) &= S(23) \times \frac{6}{6} = 0.61, \\
S(31) &= S(23) \times \frac{4}{5} = 0.49, \\
S(34) &= S(31) \times \frac{3}{4} = 0.37, \\
S(48) &= S(34) \times \frac{1}{2} = 0.18 \quad .
\end{aligned}
$$

In S-PLUS , the `survfit` function produces Kaplan-Meier survival curve estimates by default. Suppose the data displayed in table 23.1 is in a data frame named `leukemia`, with variables

`time`	time to relapse
`status`	indicator whether the observed time was a relapse (1) or censored (0).
`group`	treatment group indicator taking values `Maintained` and `Nonmaintained`.

You compute the KM estimate as follows:

```
> leukemia.surv <- survfit(Surv(time,status) ~ group,
 + leukemia)
> summary(leukemia.surv)
Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia)

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   9     11       1    0.909  0.0867       0.7541        1.000
  13     10       1    0.818  0.1163       0.6192        1.000
  18      8       1    0.716  0.1397       0.4884        1.000
```

```
  23      7       1    0.614  0.1526       0.3769        0.999
  31      5       1    0.491  0.1642       0.2549        0.946
  34      4       1    0.368  0.1627       0.1549        0.875
  48      2       1    0.184  0.1535       0.0359        0.944

                group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   5     12       2   0.8333  0.1076       0.6470        1.000
   8     10       2   0.6667  0.1361       0.4468        0.995
  12      8       1   0.5833  0.1423       0.3616        0.941
  23      6       1   0.4861  0.1481       0.2675        0.883
  27      5       1   0.3889  0.1470       0.1854        0.816
  30      4       1   0.2917  0.1387       0.1148        0.741
  33      3       1   0.1944  0.1219       0.0569        0.664
  43      2       1   0.0972  0.0919       0.0153        0.620
  45      1       1   0.0000      NA           NA           NA
```

The survfit function returns an object of class `"survfit"`. The function produces the tabled output including columns for the survival estimates, the standard errors of the estimates, and confidence bounds for the estimates. The `NA`s on the last line result from not being able to estimate a standard error and, consequently, a confidence interval for *zero* survival on a log survival scale.

23.2 NELSON AND FLEMING-HARRINGTON ESTIMATORS

Another approach is to estimate Λ, the cumulative hazard, using Nelson's estimate,

$$\hat{\Lambda}_N(t) = \sum_{t_i < t} \frac{d(t_i)}{r(t_i)},$$

or, using counting process notation,

$$\hat{\Lambda}_N(t) = \sum_{i=1}^{n} \int_0^t \frac{dN_i(s)}{r(s)}.$$

The Nelson estimate is also a step function. It starts at zero and has a step of size *d(t)/r(t)* at each death.

One problem with the Nelson estimate is that it is susceptible to ties in the data. For example, assume that 3 subjects die at 3 nearby times t_1, t_2, t_3, with 7 other subjects also at risk. Then the total increment in the Nelson estimate will be 1/10 + 1/9 + 1/8. However, if time data were grouped such that the

distinction between t_1, t_2, and t_3 was lost, the increment would be the smaller step 3/10. If there are a large number of ties this can introduce significant bias. One solution is to employ a modified Nelson estimate that always uses the larger increment, as suggested by Nelson and Fleming and Harrington (1984). This is not an issue with the Kaplan-Meier estimate. With or without ties the multiplicative step will be 7/10.

The relationship $\Lambda(t) = -\log S(t)$, which holds for any continuous distribution, leads to the Fleming-Harrington (FH) [Fleming and Harrington (1984)] estimate of survival:

$$\hat{S}_{FH}(t_j) = e^{-\hat{\Lambda}_N(t_j)} \tag{23.1}$$

This estimate has natural connections to survival curves for a Cox model. For sufficiently large sample sizes the FH and KM estimates will be arbitrarily close to one another, but keep in mind that unless there is heavy censoring the number at risk, *r(t)*, is always small in the right hand tail of the estimated curve.

Example: AML Study (cont.)

You produce the Fleming-Harrington estimate of survival for the data in table 23.1 by specifying the `type` argument in the call to `survfit`.

```
> summary(survfit(Surv(time, status) ~ group,
+ data = leukemia, type = "fleming-harrington"))

Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia, type = "fleming-harrington")

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   9     11       1    0.913  0.0871       0.7575        1.000
  13     10       1    0.826  0.1174       0.6253        1.000
  18      8       1    0.729  0.1422       0.4974        1.000
  23      7       1    0.632  0.1572       0.3882        1.000
  31      5       1    0.517  0.1731       0.2687        0.997
  34      4       1    0.403  0.1781       0.1695        0.958
  48      2       1    0.244  0.2038       0.0477        1.000

                group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   5     12       2   0.8465   0.109       0.6572        1.000
   8     10       2   0.6930   0.141       0.4645        1.000
  12      8       1   0.6116   0.149       0.3791        0.987
```

```
23      6       1    0.5177   0.158       0.2849        0.941
27      5       1    0.4239   0.160       0.2021        0.889
30      4       1    0.3301   0.157       0.1300        0.838
33      3       1    0.2365   0.148       0.0692        0.808
43      2       1    0.1435   0.136       0.0225        0.914
45      1       1    0.0528     Inf       0.0000        1.000
```

You produce the usual Nelson estimate, similarly, by specifying `type = "fh"`. Specifying `type = "fh2"` produces a modified Nelson estimate. For example, you produce the Fleming-Harrington and Nelson estimates more simply as follows:

```
# Fleming-Harrington
> survfit(Surv(time, status) ~ group, data = leukemia,
+ type = "flem")

# Nelson Estimate
> survfit(Surv(time, status) ~ group, data = leukemia,
+ type = "fh")
```

23.3 VARIANCE ESTIMATION

Several estimates of the varaiance of $\hat{\Lambda}_N$ are possible. Since $\hat{\Lambda}_N$ can be treated as a sum of independent increments, the variance is a cumulative sum with terms of

$$\frac{d(t)}{r(t)[r(t)-d(t)]} \quad \text{Greenwood,}$$

$$\frac{d(t)}{r^2(t)} \quad \text{Tsiatis,}$$

$$\frac{d(t)[r(t)-d(t)]}{r^3(t)} \quad \text{Klein.}$$

See Klein (1991) for details. Using equation (23.1) and the simple Taylor series approximation $\text{var} \log f \approx \text{var}\, f/f^2$, the variance of the KM or FH estimators is

$$\text{var}(\hat{S}(t)) = \hat{S}^2(t)\text{var}(\hat{\Lambda}_N(t)). \tag{23.2}$$

Klein also considers two other forms for the variance of S, but concludes

- For computing the variance of $\hat{\Lambda}_N$ the Tsiatis formula is preferred.

- For computing the variance of $\hat{S}$, the Greenwood formula along with equation (23.2) is preferred.

Confidence intervals for $S(t)$ can be computed on the plain (identity) scale,

$$S \pm 1.96\,\mathrm{se}(S), \tag{23.3}$$

on the cumulative hazard or log-survival scale,,

$$\exp(\log S \pm 1.96\ \mathrm{se}(\Lambda)) \tag{23.4}$$

or on the log-hazard or log-log survival scale,,

$$\exp(-\exp(\log(-\log S) \pm 1.96\ \mathrm{se}(\log \Lambda))) \tag{23.5}$$

where se refers to the standard error.

Confidence intervals based on equation (23.3) may give survival probabilities that are greater than 1 or less than zero. Those based on equation (23.4) may sometimes be greater than 1, but those based on equation (23.5) are always between 0 and 1. For this reason many users prefer the log-hazard formulation. Link (1984), (1986), however, suggests that confidence intervals based on the cumulative-hazard scale have the best performance. All three methods have been implemented in the `survfit` function and are referred to as the `"plain"`, `"log"`, and `"log-log"` confidence types. By default, the `summary.survfit` confidence intervals based on the log-survival (or cumulative hazard) scale. Intervals on the two other scales may be specified through the `conf.type` argument to `survfit`. Intervals on the other scales are computed based on the following relationships:

$$\mathrm{se}(S) \cong S\,\mathrm{se}(\Lambda)$$

$$\mathrm{se}(\log \Lambda) \cong \frac{1}{\Lambda}\,\mathrm{se}(\Lambda)$$

A further refinement to the confidence intervals is suggested by Dorey and Korn (1987). When the tail of the survival curve contains much censoring and few deaths, there will be one or more long flat segments. Confidence intervals based strictly on equation (23.3), (23.4), or (23.5) are constant across these intervals. Dorey and Korn point out that, as censored subjects are removed from the sample, the effective sample size decreases, so the actual reliability of the curve should also decrease. Their correction retains the original upper confidence limit and a modified lower limit which agrees with the standard limits at each death time but is based on the *effective number at risk* between death times.

Three lower confidence limit methods (the `conf.lower` argument) are implemented in `survfit`. The usual method (`conf.lower = "usual"`) uses, optionally, either the Greenwood or the Tsiatis formulation unaltered. Peto's method (`conf.lower = "peto"`) assumes that

$$\widehat{\text{var}}(\Lambda_N(t)) = c/r(t),$$

where $r(t)$ is the number at risk, and $c \equiv 1 - \hat{S}(t)$. The Peto limit is known to be conservative. The *modified* Peto limit (`conf.lower = "modified"`) chooses c such that the variance at each death time is equal to the usual estimate but between death times the usual variance estimate is multiplied by $r^*(t)/r(t)$, where $r(t)$ is the number at risk and $r^*(t)$ is the number at risk at the last jump in the curve (last death time). This is almost identical to Dorey and Korn's estimator and is the recommended procedure.

Example: AML Study (cont.)

Applying the methods of this section to the `leukemia` data, you can compute the conservative lower confidence intervals of Peto for survival based on the log-hazard scale as follows:

```
> summary(survfit(Surv(time, status) ~ group,
+ data = leukemia,
+ conf.type = "log-log", conf.lower = "peto"))
Call: survfit(formula = Surv(time, status) ~ group, data =
 leukemia, conf.type = "log-log", conf.lower = "peto")

                group=Maintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   9     11       1    0.909  0.0867       0.5390        0.987
  13     10       1    0.818  0.1163       0.4729        0.951
  18      8       1    0.716  0.1397       0.3645        0.899
  23      7       1    0.614  0.1526       0.2854        0.835
  31      5       1    0.491  0.1642       0.1802        0.753
  34      4       1    0.368  0.1627       0.1132        0.657
  48      2       1    0.184  0.1535       0.0288        0.525

                group=Nonmaintained
time n.risk n.event survival std.err lower 95% CI upper 95% CI
   5     12       2   0.8333  0.1076       0.5235        0.956
   8     10       2   0.6667  0.1361       0.3753        0.860
  12      8       1   0.5833  0.1423       0.2906        0.801
  23      6       1   0.4861  0.1481       0.2024        0.730
  27      5       1   0.3889  0.1470       0.1421        0.650
  30      4       1   0.2917  0.1387       0.0901        0.561
```

```
33      3       1   0.1944  0.1219      0.0476       0.461
43      2       1   0.0972  0.0919      0.0166       0.349
45      1       1   0.0000      NA          NA          NA
```

23.4 MEAN AND MEDIAN SURVIVAL

For the Kaplan-Meier estimate, the estimated mean survival is undefined if the last observation is censored. The procedure used by S-PLUS is to redefine the estimate to be zero beyond the last observation. This gives an estimated mean that is biased towards zero, but there are no compelling alternatives that do better. With this definition, the mean is estimated as

$$\hat{\mu} = \int_0^T \hat{S}(t)dt,$$

where $\hat{S}$ is the Kaplan-Meier estimate and T is the maximum observed follow-up time in the study. The variance of the mean is

$$\text{var}(\hat{\mu}) = \int_0^T \left(\int_t^T \hat{S}(u)du \right)^2 \frac{dN(t)}{r(t)(r(t) - N(t))},$$

where $N(t) = \sum N_i(t) = d(t)$ is the total number of deaths up to time t, and $r(t) = \sum Y_i(t)$ is the number at risk at time t.

The sample median is defined as the first time at which $\hat{S}(t) \leq 0.5$. Upper and lower confidence intervals for the median are defined in terms of the confidence intervals for S: the upper confidence interval is the first time at which the upper confidence interval for $\hat{S}$ is ≤ 0.5. This corresponds to drawing a horizontal line at 0.5 on the graph of the survival curve, and using intersections of this line with the curve and its upper and lower confidence bands. In the event that the survival curve has a horizontal portion at exactly 0.5 (for example, an even number of subjects and no censoring before the median) then the average time of that horizontal segment is used. This agrees with the usual definition of the median for uncensored data when the sample size is an even number. If neither confidence band for *S(t)* reaches 0.5, as in the example which follows, then the corresponding confidence limit for the median is unknown and is reported as an NA.

Example: AML Study (cont.)

The mean, median, and confidence intervals for the median survival time are part of the table produced by printing a "survfit" object. For the leukemia data set these statistics are produced as follows:

```
> leukemia.surv <- survfit(Surv(time, status) ~ group,
+ leukemia)
> leukemia.surv
Call: survfit(formula = Surv(time, status) ~ group, data =
leukemia)

                     n events mean se(mean) median 0.95LCL 0.95UCL
   group=Maintained 11      7 52.6    19.83     31      18      NA
group=Nonmaintained 12     11 22.7     4.18     23       8      NA
```

Printing the object returned by survfit produces a brief report of the resulting fits; for each fit, the print method prints the number of subjects in the cohort (n), the total number of events (events), as well as the mean, its standard error (se(mean)), the median, and confidence intervals for the median survival time (the last two columns).

23.5 COMPARISON OF SURVIVAL CURVES

Assume that we wish to compare p different groups with respect to their survival distributions. One method is to form the $p \times 2$ table at each death time.

Groups:	1	2	...	p	
Deaths	d_1	d_2		d_p	d
Alive and at risk	a_1	a_2		a_p	a
Totals	n_1	n_2		n_p	N

If there are no tied deaths, then $d = 1$ for each table. Treating this table as a simple multinomial experiment with d events in N trials, the expected number of deaths in each group is dn_i/N with a standard multinomial variance matrix V.

Treating each of the k unique death time tables as independent, we can sum over the tables to obtain an observed and an expected number of deaths for each group. This “O-E" vector has variance matrix $\sum V_k$. The argument

may be generalized by the inclusion of weights w_k for each death time. The overall weighted vector is then $\sum w_k(O_k - E_k)$, where O_k is the top row of table k, E_k is the expected, and the variance is $\sum w_k^2 V_k$. When $w_k = 1$ this is the Mantel-Haenszel or log-rank test, for $w_k = n_k$ it is the Gehan-Wilcoxon test, and for $w_k = S_{KM}(t_k)$ it is the Peto-Peto modification of the Wilcoxon test.

The `survdiff` function implements a family of tests suggested by Fleming and Harrington (1981) for comparing two or more survival curves. A single parameter ρ controls the weights given to different survival times; $\rho = 0$ yields the log-rank test and $\rho = 1$ the Peto-Wilcoxon. Other values give a test that is intermediate to these two. The default value is $\rho = 0$.

The log rank test is most powerful for a proportional hazards alternative, that is, when $\lambda_i(t)/\lambda_j(t) = c_{ij}$ for any two groups i and j, and some constant c which is independent of time. This assumption is found to hold, at least approximately, in many clinical trials. Other values for ρ produce tests more sensitive to early differences in S ($\rho > 0$) or to later differences ($\rho < 0$).

Example: AML Study (cont.)

Returning to the `leukemia` data frame, compare the two treatment groups using `survdiff`. The `survdiff` function takes a formula and a data frame as its first two arguments. Recalling that $\rho = 0$ by default, the log-rank test for difference between the maintained and non-maintained groups is produced as follows:

```
> survdiff(Surv(time, status) ~ group, leukemia)
                      N Observed Expected (O-E)^2/E
   group=Maintained 11        7   10.689     1.273
group=Nonmaintained 12       11    7.311     1.862

 Chisq= 3.4 on 1 degrees of freedom, p= 0.06534
```

Thus, there is mild evidence to suggest that the maintained group has better survival than the non-maintained group.

23.6 MORE ON SURVFIT

The `survfit` function fits Kaplan-Meier or, optionally, Fleming-Harrington survival curves. For example,

```
> sf <- survfit(Surv(futime, fustat) ~ rx + residual.dz,
+ ovarian)
> sf
Call: survfit(Surv(futime, fustat) ~ rx + residual.dz,
data = ovarian)

                    n events mean se(mean) median 0.95CI 0.95CI
rx=1, residual.dz=1 5      1  989      101     NA    638      NA
rx=1, residual.dz=2 8      6  430      131    298    156      NA
rx=2, residual.dz=1 6      2  943      161     NA    563      NA
rx=2, residual.dz=2 7      3  833      156     NA    464      NA
```

results in four Kaplan-Meier survival curves, indexed by the two levels of treatment (`rx`) and the two levels of residual disease (`residual.dz`). The right hand side of the formula is interpreted differently than it would be for an ordinary linear or Cox model. The `survfit` function uses the + operator to specify an interaction.

A summary of important options to `survfit` are:

`weights`	case weights
`type`	Type of fit—`"kaplan-meier"`, `"fleming-harrington"` or `"fh2"`.
`error`	Type of variance estimate—`"greenwood"` or `"tsiatis"`.
`conf.int=0.95`	Level for the two-sided confidence interval of median survival.
`conf.type="log"`	One of `"none"`, `"plain"`, `"log"`, or `"log-log"`.
`conf.lower`	One of `"usual"`, `"peto"`, or `"modified"`.

The `plot.survfit` function plots survival curves returned by `survfit`. For the AML data, you can plot survival curves, and add a title and legend by doing

```
> plot(leukemia.surv, xlab = "Survival Time in Weeks",
+ ylab = "Proportion Surviving", cex = 2, lty = 2:3)
> title("AML Maintenance Study")
> legend(c(75, 130), c(0.95, 0.85),
+ c("Maintenance", "No Maintenance"), lty = 2:3)
```

Figure 23.1 displays the results of plotting the Kaplan-Meier estimates of survival stratified by the maintenance grouping variable `group`. Some

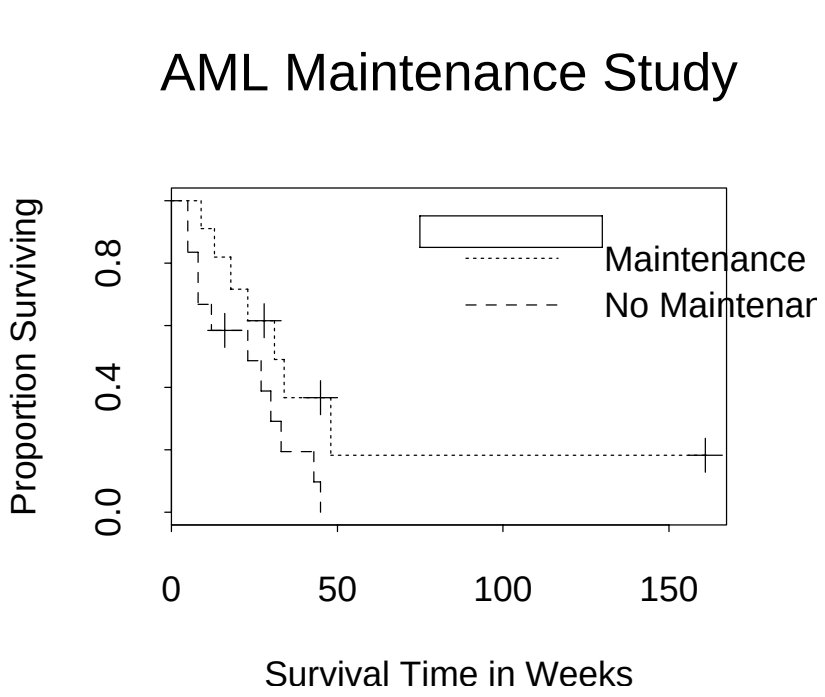

Figure 23.1: *Kaplan-Meier estimates of survival for the maintained and non-maintained groups of the AML study.*

important optional arguments to `plot.survfit` are as follows:

`conf.int`	Plot confidence intervals for the curves. Default is `TRUE` for a single curve and `FALSE` for multiple curves.
`mark.time`	If logical, indicates whether to mark the curves at censoring times. If a numeric vector, the curve is marked at each time indicated.
`mark = 3`	A vector of characters or integers specifying special symbols used to mark the curve. The default value produces a + at the censored values.
`cex = 1`	The character size of the censor marks.

By default, confidence intervals are suppressed if there are multiple curves. Marks are normally placed on the curve(s) at each censoring time. If there are a large number of censored observations, this can make the plot too "busy", and the `mark.time` option would be used to specify the time values at which curves are labeled.

23.7 REFERENCES

Andersen, P. K. and Gill, R. D. (1982). *Cox's regression model for counting processes: A large sample study.* Annals of Statistics, 10:1100–1120.

Dorey, F. J. and Korn, E. L. (1987). *Effective sample sizes for confidence intervals for survival probabilities*. Statistics in Medicine, 6:679–687.

Embury, S. H. and Elias, L. and Heller, P. H. and Hood, C. E. and Greenberg, P. L. and Schrier, S. L. (1977). *Remission maintenance therapy in acute myelogenous leukemia*. Western Journal of Medicine, 126:267-272.

Fleming, T. R. and Harrington, D. P. (1981). *A Class of Hypothesis Tests for One and Two Sample Censored Survival Data*. Communications in Statistics, A10(8):763-794.

Fleming, T. R. and Harrington, D. P. (1984). *Nonparametric estimation of the survival distribution in censored data*. Communications in Statistics, 13(20):2469-2486.

Fleming, T. and Harrington, D. (1991). *Counting Processes and Survival Analysis*. Wiley, New York.

Klein, J. P. (1991). *Small sample moments of the estimators of the variance of the Kaplan-Meier and Nelson-Aalen estimators*. Scandinavian Journal of Statistics, 18:333-340.

Link, C. L. (1984). *Confidence Intervals for the Survival Function using Cox's Proportional-Hazard Model with Covariates*. Biometrics, 40:601-610.

Link, C. L. (1986). *Confidence Intervals for the Survival Function in the Presence of Covariates*. Biometrics, 42:219-220.

THE COX PROPORTIONAL HAZARDS MODEL

24

The Cox model is the most commonly used regression model for survival data.

THE COX PROPORTIONAL HAZARDS MODEL

The Cox proportional hazards model is the most commonly used regression model for survival data. If $Z_i(t)$ is the vector of covariates for the ith individual at time t, the model assumes that the hazard for a subject is of the form

$$\lambda(t; Z_i)=\lambda_0(t)r_i(t),$$

where

$$r_i(t) = e^{\beta' Z_i(t)}$$

is referred to as the *risk score* for the ith subject, β is a vector of regression parameters, and $\lambda_0(t)$ is an arbitrary and unspecified baseline hazard function. The vector of coefficients β does not include an intercept term; it is absorbed into λ_0. The exponential function guarantees that λ is positive for any β. Assume that a death has occurred at time t^*. Then conditional on this death occurring, the likelihood that it would be subject i rather than some other subject is

$$L_i(\beta) = \frac{\lambda_0(t^*)\, r_i(t^*)}{\sum_j Y_j(t^*)\lambda_0(t^*)\, r_j(t^*)} = \frac{r_i(t^*)}{\sum_j Y_j(t^*) r_j(t^*)}. \qquad (24.1)$$

The product of the terms (equation (24.1)) over all death times, $L(\beta) = \prod L_i(\beta)$, was termed a partial likelihood by Cox (1972). Maximization of $log(L(\beta))$ gives an estimate for β without the need to estimate the nuisance parameter $\lambda_0(t)$. An estimator of the covariance matrix is given by the inverse of the second derivative matrix. The proportional hazards model is non-parametric in the sense that it depends only on the ranks of the survival times. It remains sensitive, however, to skewed

covariates. The first derivative of $log(L(\beta))$ is the p by 1 vector

$$U(\beta) = \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta, t)] dN_i(t) \tag{24.2}$$

$$= \sum_{i=1}^{n} \int_0^{\infty} [Z_i(t) - \bar{Z}(\beta, t)] dM_i(\beta, t) \tag{24.3}$$

and the p by p information matrix is

$$I(\beta) = \sum_{i=1}^{n} \int_0^{\infty} \frac{\sum_j Y_j(t) r_j(t) [Z_i(t) - \bar{Z}(t)][Z_i(t) - \bar{Z}(\beta, t)]'}{\sum_j Y_j(t) r_j(t)} dN_i(t), \tag{24.4}$$

where $\bar{Z}$ is the weighted covariate mean for those still at risk at time t

$$\bar{Z}(\beta, t) = \frac{\sum Y_j(t) r_j(t) Z_i(t)}{\sum Y_i(t) r_i(t)}.$$

Cox proposed, and it was later shown by Efron (1977) and Oakes (1977), that the partial likelihood contains "nearly" all of the information about β. That is, the calendar times when deaths occur give information about the overall hazard rate λ_0 but little about the relative rates for different values of Z. The Cox model thus gives very efficient estimates as compared to a parametric proportional hazards model, such as the Weibull, even when the data actually come from the parametric model. The notation for L_i in equation (24.1) is derived from the counting process representation found in Fleming and Harrington (1991). It allows for several extensions to the original Cox model formulation including:

- multiple events per subject,
- time-dependent covariates including cation variables,
- discontinuous intervals of risk—Y_i may change states from 1 to 0 and back again multiple times,

- left truncation—subjects need not enter the risk set at time 0.

This extension is known as the *multiplicative hazards model*.

Example: Ovarian Cancer

This example uses data from a study of ovarian cancer [EFD+79]. The variables are:

`futime`	The number of days from enrollment until death or censoring, whichever comes first.
`fustat`	An indicator of death (1) or censoring (0).
`age`	The patient age in years (actually, the age in days divided by 365.25)
`residual.dz`	An indicator of the extent of residual disease.
`rx`	An indicator of the treatment given.
`ecog.ps`	A measure of performance score or functional status, using the Eastern Cooperative Oncology Group's scale. It ranges from 0 (fully functional) to 4 (completely disabled). Level 4 subjects are usually considered too ill to enter a randomized trial such as this.

The data is stored in a data frame named `ovarian`. A summary produces the following:

```
> summary(ovarian)
      futime          fustat           age
 Min.   :  59.0  Min.   :0.0000  Min.   :38.89
 1st Qu.: 368.0  1st Qu.:0.0000  1st Qu.:50.17
 Median : 476.0  Median :0.0000  Median :56.85
 Mean   : 599.5  Mean   :0.4615  Mean   :56.17
 3rd Qu.: 794.8  3rd Qu.:1.0000  3rd Qu.:62.38
 Max.   :1227.0  Max.   :1.0000  Max.   :74.50
  residual.dz         rx            ecog.ps
 Min.   :1.000   Min.   :1.0     Min.   :1.000
 1st Qu.:1.000   1st Qu.:1.0     1st Qu.:1.000
 Median :2.000   Median :1.5     Median :1.000
 Mean   :1.577   Mean   :1.5     Mean   :1.462
 3rd Qu.:2.000   3rd Qu.:2.0     3rd Qu.:2.000
 Max.   :2.000   Max.   :2.0     Max.   :2.000
```

Start by modeling survival as a function of `age` only:

```
> ov.fit1 <- coxph(Surv(futime, fustat) ~ age, ovarian)
> ov.fit1
```

```
Call: coxph(formula = Surv(futime,fustat) ~ age,
data=ovarian)
     coef exp(coef) se(coef)    z      p
age 0.162      1.18   0.0497 3.25 0.0012
Likelihood ratio test=14.3  on 1 df, p=0.000156  n=26
```

Printing the resulting fit produces the estimated coefficient $(\hat{\beta})$, the estimated relative risk for a one unit change in the variable $e^{(\hat{\beta})}$, the standard error of the estimated coefficient, a z-test $((\hat{\beta})/se(\hat{\beta}))$ along with its p-value for the significance of the estimated coefficient, and a likelihood ratio test for goodness of fit. The z-test is sometimes referred to as Wald's test. An estimate of the relative risk of dying of ovarian cancer for two patients in the study differing in age by one year is 1.18 which is significantly larger than one (p = 0.000156). The older patient has an estimated 1.18 times higher risk of dying of ovarian cancer than the younger patient. You produce a summary of the survival curve with a combination of the `summary` function and the `survfit` function. For example,

```
> summary(survfit(ov.fit1))
Call: survfit.coxph(object = ov.fit1)
 time n.risk n.event survival std.err lower 95% CI upper 95% CI
   59     26       1    0.988  0.0142        0.961        1.000
  115     25       1    0.974  0.0244        0.927        1.000
  156     24       1    0.955  0.0364        0.886        1.000
  268     23       1    0.933  0.0482        0.844        1.000
  329     22       1    0.897  0.0621        0.783        1.000
  353     21       1    0.862  0.0724        0.732        1.000
  365     20       1    0.824  0.0819        0.678        1.000
  431     17       1    0.775  0.0934        0.612        0.982
  464     15       1    0.724  0.1032        0.548        0.958
  475     14       1    0.673  0.1112        0.487        0.931
  563     12       1    0.596  0.1226        0.398        0.892
  638     11       1    0.520  0.1287        0.321        0.845
```

The Fleming-Harrington estimate of survival for a patient with age equal to the *average* is produced in this case because the model was fit using `coxph` and survival for a particular age was not specified with the `newdata` argument. Produce a plot of the survival curve, figure 24.1, at the average age as follows:

```
> plot(survfit(ov.fit1), xlab = "Survival in Days",
+ ylab = "Proportion Surviving")
> title("Suvival for Ovarian Cancer Study")
```

The default, when you plot only one curve, is to add confidence limits.

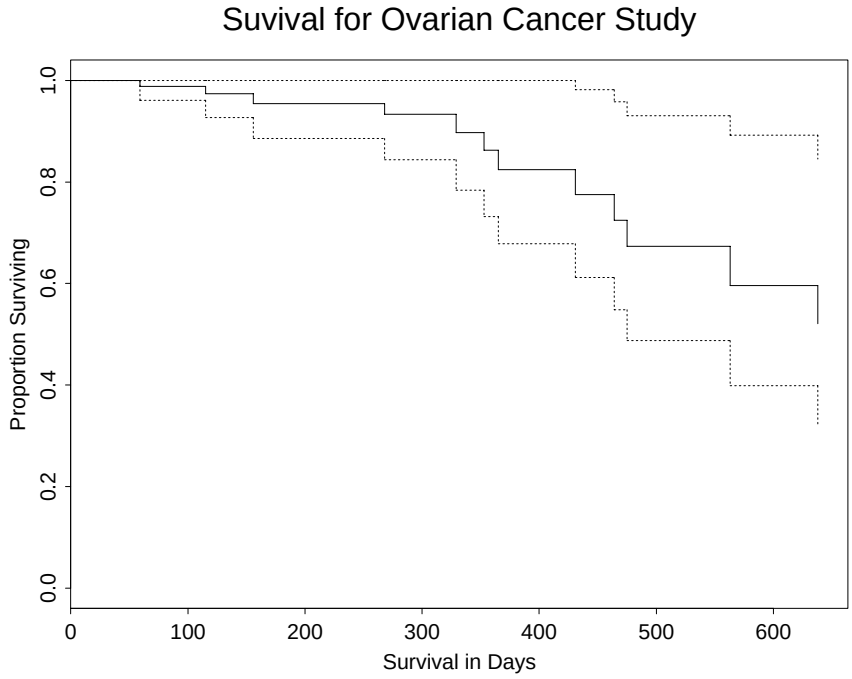

Figure 24.1: *Cox regression estimate of survival for a subject of average age (*56.17 *years), from the ovarian cancer study.*

24.1 HYPOTHESIS TESTS

Once you fit a Cox model, three tests of hypothesis are produced which are asymptotically equivalent but not always in practice. Let β_0 be the initial value of the coefficients and $\hat{\beta}$ the solution after fitting the model. The *likelihood ratio test* is defined as

$$-2\{log(L(\beta_0)) - log(L(\hat{\beta}))\}$$

and is the most reliable. The *Wald statistic*, $(\hat{\beta} - \beta_0)'\hat{\Sigma}_{\hat{\beta}}^{-1}(\hat{\beta} - \beta_0)$, where $\hat{\Sigma}_{\hat{\beta}}^{-1}$ is the estimated variance-covariance matrix, is perhaps the most natural because it provides a per variable test rather than an overall measure of significance. The *score test* is defined as $U^t I U$ where U is the vector of derivatives given by equation (24.3) and I is the information matrix given by equation (24.4), both evaluated at β_0. The score test does not require iteration and, consequently, is more computationally efficient if a large number of models are to be tested.

Example: Ovarian Cancer (cont.)

For the ovarian cancer example, you can compute all three tests by computing a summary of the resulting fit.

```
> summary(ov.fit1)
Call: coxph(formula = Surv(futime, fustat) ~ age, data =
ovarian)
  n= 26
     coef exp(coef) se(coef)  z        p
age 0.162      1.18   0.0497 3.25 0.0012
    exp(coef) exp(-coef) lower .95 upper .95
age      1.18      0.851      1.07       1.3

Rsquare= 0.423   (max possible= 0.932 )
Likelihood ratio test= 14.3 on 1 df,    p=0.000156
Wald test            = 10.6 on 1 df,    p=0.00116
Efficient score test = 12.3 on 1 df,    p=0.000463
```

The summary of a fit returns the *efficient score test* in addition to the likelihood ratio test and Wald's test resulting from simply printing the fit. Additionally, a confidence interval is estimated for the relative risk estimated by `exp(coef)`, $e^{\hat{\beta}}$. To produce confidence limits with a different confidence level use the `conf.int` argument in the call to `summary`. For example, specifying `conf.int = .99` produces 99% confidence intervals for the relative risk. It is clear that age is an important predictor of survival. Let's add the other predictors to the model.

```
> ov.fit2 <- coxph(Surv(futime, fustat) ~ age +
+ residual.dz + rx + ecog.ps, ovarian)
> ov.fit2
Call:
coxph(formula = Surv(futime, fustat) ~ age +
        residual.dz + rx + ecog.ps, data = ovarian)

              coef exp(coef) se(coef)      z      p
age          0.125     1.133   0.0469  2.662 0.0078
residual.dz  0.826     2.285   0.7896  1.046 0.3000
rx          -0.914     0.401   0.6533 -1.400 0.1600
ecog.ps      0.336     1.400   0.6439  0.522 0.6000

Likelihood ratio test=17 on 4 df, p=0.0019 n= 26
```

To check for an overall improved fit over the `age` only model compute the likelihood ratio test between the models as follows:

```
> -2*(ov.fit1loglik[2] - ov.fit2loglik[2])
[1] 2.749708
```

The `loglik` component of the fit is a vector of the log likelihoods for two fits. The null model (intercept only) is the first value, and the current model is the second value. Noting that there is a difference of three degrees of freedom between the models, the p-value for the likelihood ratio test is computed as follows:

```
> pchisq(2.75, df = 3)
[1] 0.5682029
```

There is no significant difference between the two models indicating that `residual.dz`, `rx`, and `ecog.ps` don't improve the fit. This will not work if there are missing values.

24.2 STRATIFICATION

A simple extension of the Cox model is to allow multiple strata. The hazard for a subject contained in stratum *j* is then

$$\lambda(t, Z) = \lambda_j(t)\, e^{\beta Z(t)}.$$

When a variable is entered into the model as a stratification factor rather than as a covariate it allows for non-proportional hazards to exist between levels of the variable. However, the disadvantage is that no β is available to estimate the effect of that variable. For instance, in a multi-center drug study the enrolling center is often entered into the model as a stratum variable. Because of different patient populations, for example, a higher proportion of acute cases, the centers may well have different shapes for their baseline survival curves, and if modeled as a covariate this non-proportionality could bias the estimate of the treatment effect.

Example: Ovarian Cancer (cont.)

You can stratify the ovarian cancer fit with respect to treatment, `rx`, still fitting `age` as a covariate, as follows:

```
> ov.fit3 <- coxph(Surv(futime, fustat) ~ age + strata(rx),
+ data = ovarian)
> survfit(ov.fit3)
Call: survfit.coxph(object = ov.fit3)
                n events mean se(mean) median 0.95LCL 0.95UCL
strata(rx)=rx=1 13      7  512     72.8    638     329      NA
strata(rx)=rx=2 13      5  522     22.5     NA     475      NA
```

Printing the resulting fit displays the usual summary statistics for the survival curve for each stratum. Applying the `summary` function to the fit produces a more detailed table which includes the survival curve, standard errors and confidence intervals for each stratum:

```
> summary(survfit(ov.fit3))
Call: survfit.coxph(object = ov.fit3)

                strata(rx)=rx=1
time n.risk n.event survival std.err lower 95% CI upper 95% CI
  59     13       1    0.978  0.0269        0.9264            1
 115     12       1    0.950  0.0481        0.8607            1
 156     11       1    0.910  0.0758        0.7725            1
 268     10       1    0.862  0.1050        0.6793            1
 329      9       1    0.736  0.1525        0.4902            1
 431      8       1    0.625  0.1698        0.3671            1
 638      5       1    0.341  0.2225        0.0947            1

                strata(rx)=rx=2
time n.risk n.event survival std.err lower 95% CI upper 95% CI
 353     13       1    0.943  0.0560         0.840        1.000
 365     12       1    0.880  0.0814         0.734        1.000
 464      9       1    0.791  0.1126         0.598        1.000
 475      8       1    0.701  0.1319         0.484        1.000
 563      7       1    0.602  0.1461         0.374        0.968
```

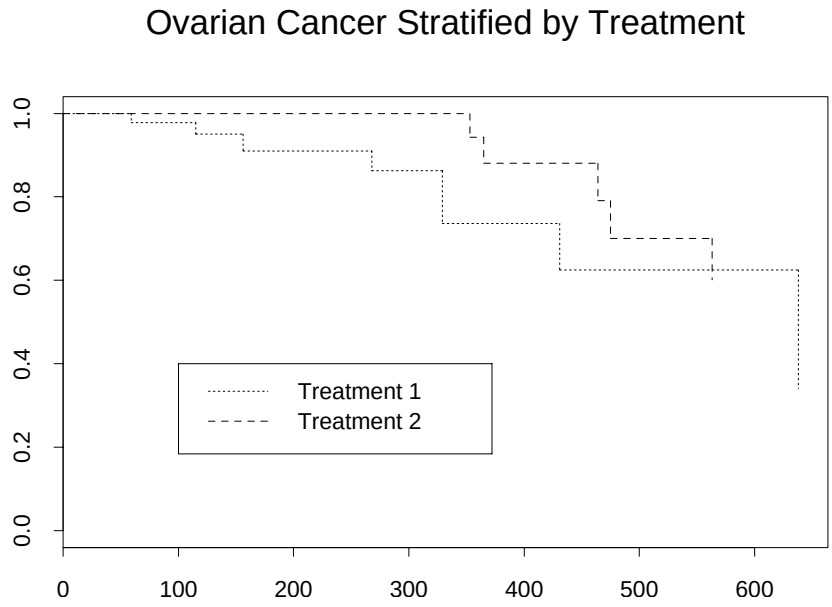

Figure 24.2: *A plot of the stratified fit of the ovarian cancer data adjusted for average age.*

You produce a plot of the stratified fit as follows:

```
> plot(survfit(ov.fit3), lty = 2:3)
> legend(100, .6, c("Treatment 1","Treatment 2"), lty= 2:3)
> title("Ovarian Cancer Stratified by Treatment")
```

The plot is one method to view a non-parametric estimate of treatment effect, after adjusting for possible differences in age distributions.

24.3 RESIDUALS

The Breslow (or Tsiatis, Link, or Nelson-Aalen) estimate of the baseline hazard is

$$\hat{\Lambda}_0(\beta, t) = \int_0^t \frac{\sum_{i=1}^n dN_i(s)}{\sum_{i=1}^n Y_i(s)\, r_i(\beta, s)}.$$

The martingale residual at time t is

$$M_i(t) = N_i(t) - \int_0^t r_i(\beta, s) Y_i(s) d\hat{\Lambda}_0(\beta, s). \tag{24.5}$$

The residual is computed at $t = \infty$ and $\beta = \hat{\beta}$. If there are no time-dependent covariates, then $r_i(t)$ can be factored out of the integral, giving $\hat{M}_i = N_i - \hat{r}_i \Lambda_0(\hat{\beta}, t_i)$. The deviance residual is a normalizing transform of the martingale residual

$$d_i = sign(\hat{M}_i) \cdot \sqrt{-\hat{M}_i - N_i log((N_i - \hat{M}_i)/N_i)}$$

The other two residuals are based on the score process $U_{ij}(b,t)$ for the ith subject and the jth variable:

$$U_{ij}(\beta, t) = \int_0^t (Z_{ij}(s) - Z_j(\beta, s)) d\hat{M}_i(\beta, s).$$

The score residual is defined, for each subject and each variable (an n by p matrix) as $U_{ij}(\hat{\beta}, \infty)$. It is the sum of the score process over time. The usual score vector $U(\beta)$ (equation (24.2)) is the column sum of the matrix of score residuals. The martingale and score residuals are integrals over time for a given subject. Specifically, in setting up a multiplicative hazards model, a single subject is represented by multiple lines of the input data, as though the subject was a set of different individuals observed over disjoint times. The

residual for that person is the sum of the residuals for these "pseudo" subjects. The Schoenfeld residuals (1982) are defined as a matrix

$$s_{ij}(\beta) = Z_{ij}(t_i) - \bar{Z}_j(\beta, t_i) \tag{24.6}$$

with one row per death and one column per covariate, where i and t_i are the subject and the time that the event occurred. The Schoenfeld residuals are related to the score process $U_{ij}(\beta,t)$. Sum the score process over individuals to get a total score process $\sum_i U_{ij}(\beta, t) = U(\beta, t)$. This is just the score vector at time t, so that at $\hat{\beta}$ we must have $U(\hat{\beta}, 0) = U(\hat{\beta}, \infty) = 0$. Because $\hat{\Lambda}$ is discrete, our estimated score process will also be discrete, having jumps at each of the unique death times. There are two simplifying identities for these residuals:

$$\begin{aligned} U(\beta, t) &= \sum_i \int_0^t Z_{ij}(s)\, dM_i(\beta, s) \\ &= \sum_i \int_0^t (Z_{ij}(s) - \bar{Z}_j(\beta, s))\, dN_i(s) \end{aligned} \tag{24.7}$$

Note that $d\hat{M}_i(t)$ is zero when subject i is not in the risk set at time t. Since the sums are the same for all t, each increment of the processes must be the same as well. Comparing the second of these to equation (24.6), we see that the Schoenfeld residuals are the increments or jumps in the total score process. There is a small nuisance with tied death times: under the integral formulation the *O-E* process has a single jump at each death time, leading to one residual for each unique event time, while under the Schoenfeld representation there is one residual for each event. In practice, the latter formulation has been found to work better for both plots and diagnostics, as it leads to residuals that are approximately equivariant. For the alternative of one residual per *unique* death time, both the size and variance of the residual is proportional to the number of events.

The last and most general residual is the entire score process R_{ijk} where i indexes subjects, j indexes the covariates, and k indexes the event times.

$$R_{ijk} = [Z_{ij}(t_k) - \bar{Z}_j(t_k)][d(N_i(t_k) - r_i(t_k)d\hat{\Lambda}_0(t_k))] .$$

The score and Schoenfeld residuals are seen to be marginal sums of this array. Lin, Wei and Ying (1992) suggest a global test of the proportional hazards model based on the maximum of the array.

Uses for the Residuals

Four possible uses of residuals are addressed in this section.

1. Discovering the correct functional form for a predictor.
2. Identifying subjects who are poorly predicted by the model.
3. Identifying influential points, that is, points with high leverage.
4. Assessing the proportional hazards assumption.

Discovering the Functional Form for a Predictor

The martingale residual, M_i, is given by equation (24.5) evaluated at $t = \infty$. Assume that the true functional form for a covariate in the exponent is $h(Z)$. Then Therneau, Grambsch, and Fleming show that the martingale residuals, after regression on the other variables, satisfy

$$E(M_i) \doteq (h(t) - \bar{h})\, E(N_i)$$

A smoothed plot of the M_i versus x will give an approximate image of the true functional form, with the y-axis scaled by a constant that depends on the proportion of censoring. If there are several covariates, then the martingale residuals from a model with all of the covariates except Z_1, say, can be plotted against the residuals of a regression of Z_1 on the others, similar to *adjusted variable plots* for the linear model in Chambers *et al.* (1983).

Another use is to plot the residuals from a null model, that is, with `iter.max=0`, against each predictor. This is roughly equivalent to the standard scatter plots of y against each Z that is used for uncensored data, before a model is fit. Addition of a local regression smooth curve using `loess` gives, in both cases, a first approximation to what transformations, if any, might be appropriate for each Z.

Identifying Poorly Predicted Subjects

The martingale residuals can be highly skewed. The deviance residual, d_i, is a normalized transform of M_i. Recent experience has shown that deviance residuals do not work well and cannot be recommended.

Identifying Influential Points

In a linear model, the influence of a point on the fit depends on both its residual and its distance from the center of the predictor space, roughly $\text{resid}_i \cdot (Z_i - \bar{Z})$. In a Cox model, the mean of the covariates changes over time as subjects leave the risk set, which suggests an average of some sort. The score residuals are a decomposition of the first derivative or score vector; large values indicate a point with high leverage. In particular, $-I^{-1}L_i$, where I^{-1} is the Cox model variance matrix, is approximately the change that would occur in β if observation i were dropped from the model. These changes in β are returned when you specify `type = "dfbeta"` or `type = "dfbetas"` to the `residuals` function.

Assessing the Proportional Hazards Assumption

The Schoenfeld residuals are increments in time for the total score process. See equation (24.6). If the proportional hazards assumption holds, the Schoenfeld residuals should be a random walk. Conversely, assume that some variable, such as treatment, has a large positive effect early but that the effect trails off. The treatment might influence how many patients survive to some point *t*, but once they are "cured" it has no influence on survival beyond *t*. In this case, proportional hazards does not hold and the fitted models will underestimate the true treatment effect for small *t*, and overestimate it for large *t*. If treatment has a beneficial effect, that is, $\beta < 0$, then the Schoenfeld residuals would have an early negative trend followed by a late positive trend. Harrell (1986) suggests using the correlation of `rank(time)` with this residual as a test for non-proportional hazards. Therneau *et al.* (1990) use the maximum of the absolute cumulative summed Schoenfeld residual, a Kolmogorov type test. Grambsch and Therneau further show that a rescaled Schoenfeld residual can correct for correlation among the covariates and be more interpretable. This result is the basis for the `cox.zph` function.

Example: Lung Cancer

This example examines data from a study of lung cancer patients conducted by the North Central Cancer Treatment Group. The `lung` data frame includes the usual survival times (`time`) and indicator variable of death or censoring (`status`) plus the following additional variables on each patient:

`inst`	A numeric code for the institution at which the patient was hospitalized.
`age`	Patient's age.
`sex`	1=male, 2=female.
`ph.ecog`	Physician's estimate of the ECOG performance score (0–4).
`ph.karno`	Physician's estimate of the Karnofsky score, a competitor to the ECOG performance score.
`pat.karno`	Patient's assessment of his/her Karnofsky score.
`meal.cal`	Calories consumed at meals excluding beverages and snacks.
`wt.loss`	Weight loss in the last 6 months.

A summary of the `lung` data frame follows:

```
> summary(lung)
      inst             time          status           age
 Min.   : 1.00   Min.   :   5.0   Min.   :1.000   Min.   :39.00
 1st Qu.: 3.00   1st Qu.: 166.8   1st Qu.:1.000   1st Qu.:56.00
 Median :11.00   Median : 255.5   Median :2.000   Median :63.00
 Mean   :11.09   Mean   : 305.2   Mean   :1.724   Mean   :62.45
```

```
3rd Qu.:16.00   3rd Qu.: 396.5   3rd Qu.:2.000   3rd Qu.:69.00
Max.   :33.00   Max.   :1022.0   Max.   :2.000   Max.   :82.00
NA's   : 1.00
sex ph.ecog ph.karno pat.karno
Min.   :1.000   Min.   :0.0000   Min.   : 50.00   Min.   : 30.00
1st Qu.:1.000   1st Qu.:0.0000   1st Qu.: 75.00   1st Qu.: 70.00
Median :1.000   Median :1.0000   Median : 80.00   Median : 80.00
Mean   :1.395   Mean   :0.9515   Mean   : 81.94   Mean   : 79.96
3rd Qu.:2.000   3rd Qu.:1.0000   3rd Qu.: 90.00   3rd Qu.: 90.00
Max.   :2.000   Max.   :3.0000   Max.   :100.00   Max.   :100.00
                NA's   :1.0000   NA's   : 1.00    NA's   :  3.00
    meal.cal        wt.loss
Min.   :  96.0   Min.   :-24.000
1st Qu.: 635.0   1st Qu.:  0.000
Median : 975.0   Median :  7.000
Mean   : 928.8   Mean   :  9.832
3rd Qu.:1150.0   3rd Qu.: 15.750
Max.   :2600.0   Max.   : 68.000
NA's   :  47.0   NA's   : 14.000
```

Note that the `status` variable takes values one (censoring) and two (event) as does the `sex` variable (1 = Male, 2 = Female). The `coxph` function recognizes either a 0/1 or a 1/2 binary variable as an indicator of censored/event status so you needn't transform the `status` variable in this case. Let's start the example by fitting a model on all the variables stratified by `sex`.

```
> lung.fit1 <- coxph(Surv(time, status) ~ strata(sex) +
+     age + ph.ecog + ph.karno + pat.karno + meal.cal +
+     wt.loss, data = lung, na.action = na.omit)
> lung.fit1
Call: coxph(formula = Surv(time, status) ~ strata(sex) +
             age + ph.ecog + ph.karno + pat.karno +
             meal.cal + wt.loss, data = lung,
             na.action = na.omit)
               coef exp(coef) se(coef)     z      p
age         9.05e-03     1.009 0.011601  0.78 0.4400
ph.ecog     7.07e-01     2.029 0.222773  3.17 0.0015
ph.karno    2.07e-02     1.021 0.011282  1.84 0.0660
pat.karno  -1.33e-02     0.987 0.008050 -1.65 0.0980
meal.cal   -5.27e-06     1.000 0.000263 -0.02 0.9800
wt.loss    -1.52e-02     0.985 0.007890 -1.93 0.0540

Likelihood ratio test=21.6 on 6 df, p=0.00145 n=168
   (60 observations deleted due to missing)
```

The resulting fit indicates that `age` and `meal.cal` are not important predictors of survival. Let's drop them from the model.

```
> lung.fit2

Call:
coxph(formula = Surv(time, status) ~ strata(sex) +
        ph.ecog + ph.karno + pat.karno + wt.loss,
        data = lung, na.action = na.omit)
              coef exp(coef) se(coef)     z p
ph.ecog    0.6495     1.915  0.20070  3.24 0.0012
ph.karno   0.0173     1.017  0.01031  1.68 0.0930
pat.karno -0.0167     0.983  0.00726 -2.30 0.0220
wt.loss   -0.0137     0.986  0.00691 -1.99 0.0470
Likelihood ratio test=25.7 on 4 df, p=3.61e-05 n=210
   (18 observations deleted due to missing)
```

Because of the different number of missing values for these two models, you cannot compare them directly using a likelihood ratio like we did for the `ovarian` data.

Assessing Functional Form

Now take a look at the functional form of the relationship with respect to each of the important predictors in the model. Do this by plotting the martingale residuals from a model with the variable of interest removed versus the variable of interest. Then add a `loess` smooth line to estimate the relationship. You can accomplish both the plot and adding the smooth by using the `scatter.smooth` function. To make the handling of `NA`s (missing values) a bit easier, begin by creating a new data frame with just the variables in the model and with the `NA`s removed.

```
> nlung <- na.omit(lung[, c("time", "status", "sex",
+ "ph.ecog", "ph.karno", "pat.karno", "wt.loss")])
```

Note the 18 row difference between the two data frames is confirmed by the number of `NA`s that were deleted in fitting `lung.fit2`.

```
> dim(nlung)
[1] 210 7
> dim(lung)
[1] 228 10
```

The four plots displayed in figure 24.3 show the estimated relationships for each predictor.

```
> par(mfrow = c(2,2))
> attach(nlung)
```

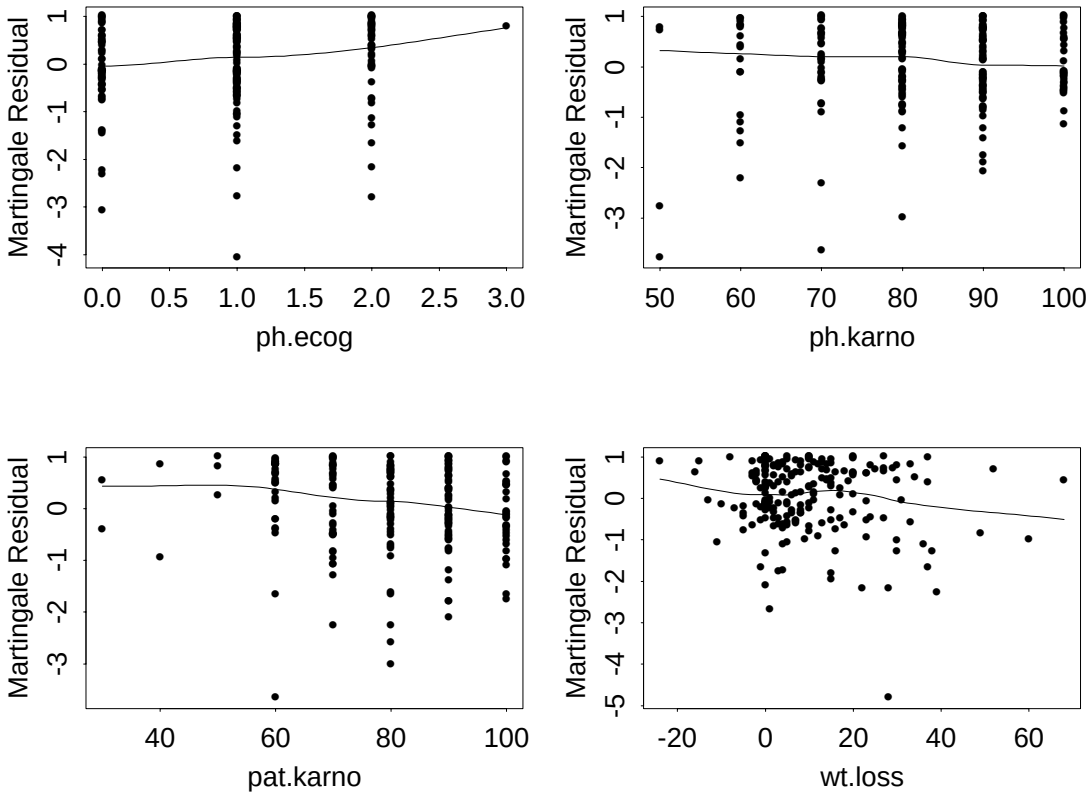

Figure 24.3: *Plots of the martingale residuals for four models with each variable in turn left out of the model for the the lung cancer study.*

```
> fit1 <- coxph(Surv(time,status) ~ strata(sex) + ph.karno +
+     pat.karno + wt.loss, data = nlung)
> scatter.smooth(ph.ecog, resid(fit1))
> fit2 <- coxph(Surv(time,status) ~ strata(sex) + ph.ecog +
+     pat.karno + wt.loss, data = nlung)
> scatter.smooth(ph.karno, resid(fit2))
> fit3 <- coxph(Surv(time,status) ~ strata(sex) + ph.ecog +
+     ph.karno + wt.loss, data = nlung)
> scatter.smooth(pat.karno, resid(fit3))
> fit4 <- coxph(Surv(time,status) ~ strata(sex) + ph.ecog +
+     ph.karno + pat.karno, data = nlung)
> scatter.smooth(wt.loss, resid(fit4))
```

All of the relationships look reasonably linear.

Poorly Predicted Subjects

Subjects with large deviance residuals are poorly predicted by the model. You produce the deviance residual plot for the second lung cancer model as follows:

```
> plot(resid(lung.fit2, type = "deviance"))
```

Figure 24.4 displays the resulting plot. There are no wildly deviant observations.

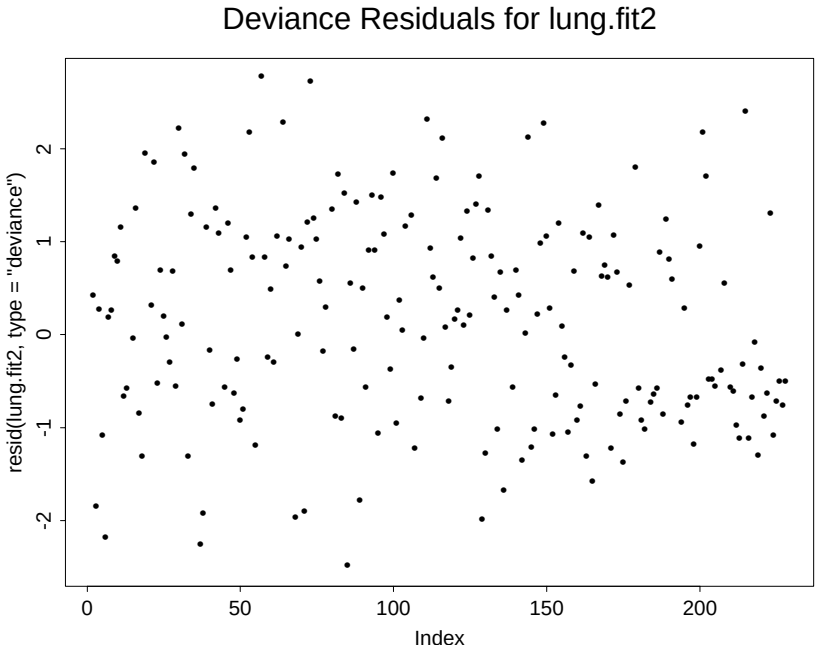

Figure 24.4: *Plots of the deviance residuals for model* `lung.fit2` *of the lung cancer study.*

Influence

Another set of plots examines the influence of individual observations on the parameter estimates. Use the changes in the estimated scaled coefficient due to dropping each observation from the fit (`type = "dfbetas"`) as a measure of influence. The first of the four plots is created as follows:

```
> bresid <- resid(lung.fit2, type = "dfbetas")
> plot(1:228, bresid[,1], type = "h",
+          ylab = "Scaled change in coef",
+          xlab = "Observation")
> title("ph.ecog")
```

The remaining plots are created by selecting the appropriate columns of `bresid` and changing labels on the plots. The resulting plots are displayed in figure 24.5. Note the use of `1:228` to generate the indices for the

observations even though the fit had only 210 observations after deleting missing values. The dimension of `bresid` is 228 × 4. The number of rows matches that of `lung` because the `naresid` method for omitting missing values (`na.omit`) inserts `NA`s in the residual matrix returned.

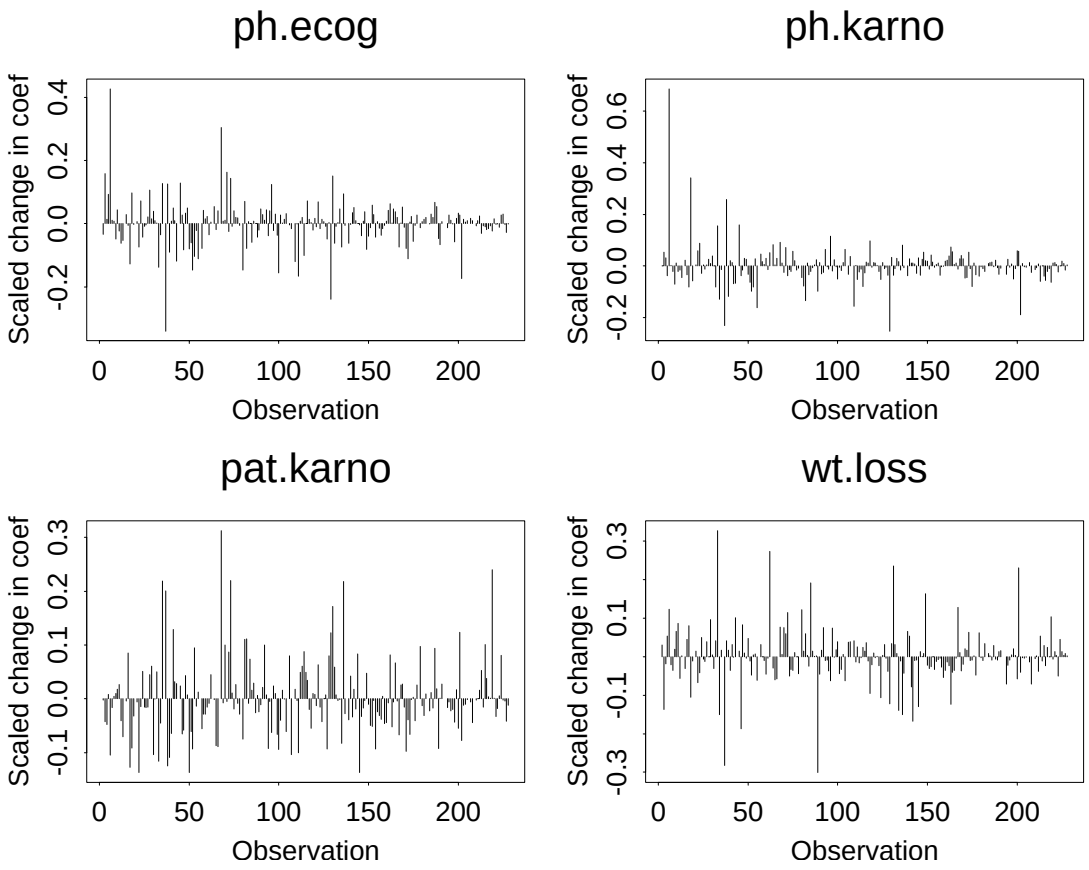

Figure 24.5: *A plot of influence by observation number for the four important predictors for the lung cancer study.*

The largest change in a regression coefficient is 0.6 standard errors of the coefficient for `ph.karno` (upper right corner plot). Since the coefficient for `ph.karno` is marginally significant at best you need not worry much about this observation. The other plots are reasonable.

Assessing Proportional Hazards

You can examine the assumption of proportional hazards both graphically and statistically for the `lung.fit2` model. The plot, figure 24.6, is produced as follows:

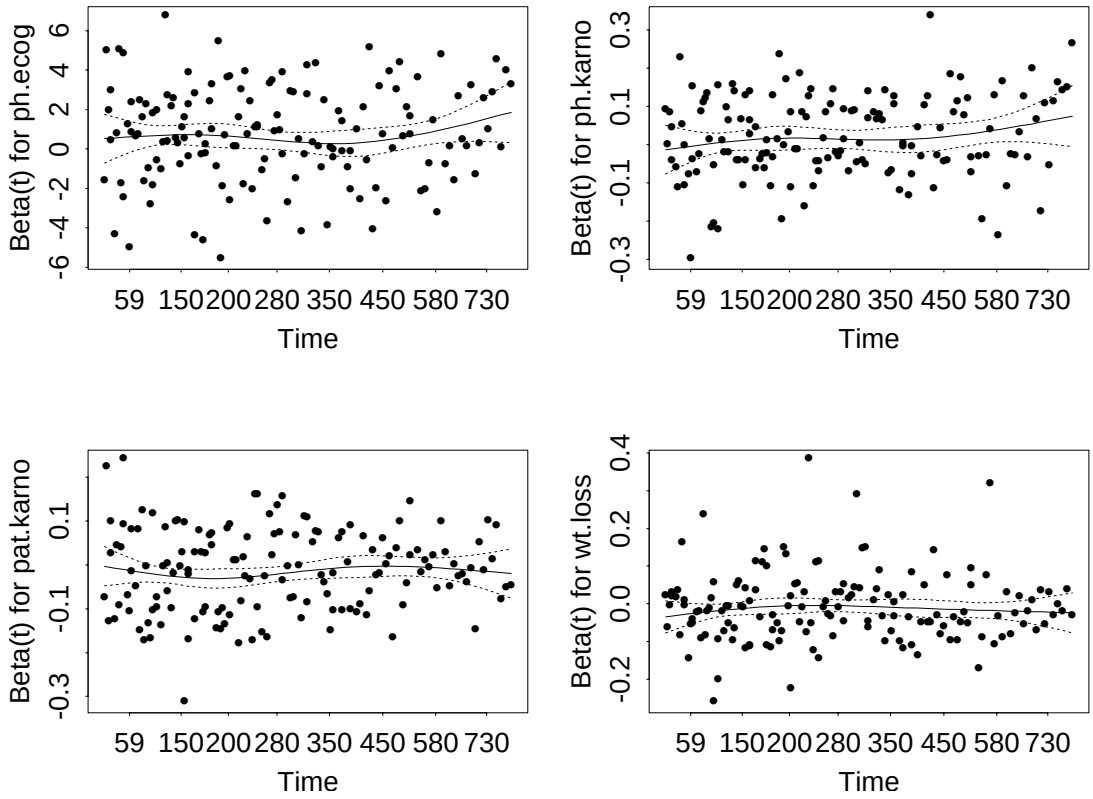

Figure 24.6: *A plot of the rescaled Schoenfeld residuals to assess the proportional hazards assumption for four covariates in lung cancer study.*

```
> plot(cox.zph(lung.fit2))
```

All of the smooth curves are flat indicating proportional hazards is a reasonable assumption. Statistical tests for significant slope in the scatter plots of figure 24.6 support the interpretation of the graphical displays.

```
> cox.zph(lung.fit2)
               rho  chisq     p
  ph.ecog  0.05189 0.3905 0.532
 ph.karno  0.14216 2.2081 0.137
pat.karno  0.04773 0.3812 0.537
  wt.loss  0.00857 0.0131 0.909
   GLOBAL       NA 4.4476 0.349
```

Plotting the Resulting Fit

Finally, you can plot estimated survival curves for the `lung.fit2` model as follows:

```
> plot(survfit(lung.fit2), lty = 2:3)
```

```
> legend(500, .9, c("Male", "Female"), lty = 2:3)
> title("Survival for Male and Female
+ Patients\nwith Average Covariates")
```

The fitted Cox models are presented in figure 24.7. Recall that the model was stratified on `sex`. The resulting survival curves are for two pseudo patients (a male and a female) with average values for each of `ph.ecog`, `ph.karno`, `pat.karno` and `wt.loss`.

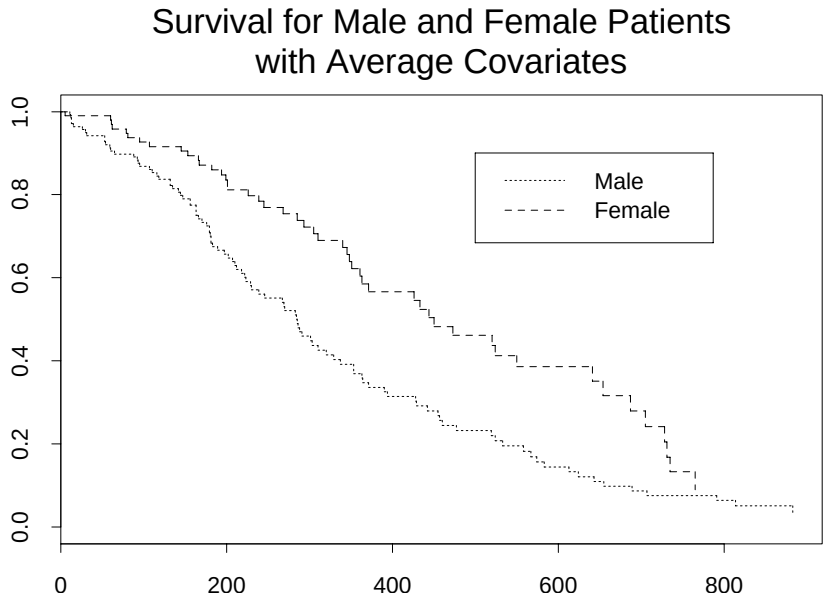

Figure 24.7: *Cox regression estimation of baseline survival curves for a sample of lung cancer patients.*

24.4 USING THE COUNTING PROCESS NOTATION

The Anderson-Gill formulation of the proportional hazards model as a counting process is useful not only theoretically, but also in the practice of fitting models. From a data analysis point of view, each subject is treated as an observation of a (very slow) Poisson process. A censored subject is thought of not as *incomplete data*, but as one whose event count is still zero. Time-dependent covariates effect the rate for upcoming events, and can depend in any way on past observation of the subject. Furthermore, intervals of observation need not be contiguous. Organizing data in this framework has advantages. Each subject is represented by a set of observations: s_{ij}, t_{ij}, δ_{ij}, x_{ij}, k_{ij}, j=1, …, n_i, where $(s_{ij}, t_{ij}]$ is an interval of risk, open on the left and closed on the right, $\delta_{ij} = 1$ if the subject had an event at time t_{ij}, x_{ij} is the covariate

vector over the interval, and k_{ij} is the stratum the subject belongs to during the interval. Data sets like this are easy to construct in S-PLUS. Following are a few specific examples to aid in constructing the analysis data frame.

Multiple Events

This example comes from a study of myocardial infarction (heart attack) patients where one of the events of interest is fatal or non-fatal re-infarction. Several patients had multiple events. The maximum number of events was three. Analysis was done using the counting process formulation by breaking any patient with multiple events into multiple intervals of risk. For example, one patient had infarctions on days 100 and 185 and was followed until day 250. This patient had three rows of data with time intervals (0, 100], (100, 185], and (185, 250] and corresponding event status codes of 1, 1, and 0.

Time-Dependent Covariates

The most common type of time-dependent covariates are repeated measurements on a subject or a change in the subject's treatment. Both of these situations are easily handled by the counting process formulation. As an example consider the Stanford heart transplant study, where treatment is a time-dependent covariate. Suppose there are two patients whose time from enrollment to death is 102 and 343 days, respectively, and that the second patient had a heart transplant 21 days after enrollment. The data for these two patients displayed is in table 24.1.

Table 24.1: *Data for two hypothetical patients in the Stanford heart transplant study*

Interval	Status	Transplant	Age	Prior Surgery
(0,102]	1	0	41	0
(0,21]	0	0	48	1
(21,343]	1	1	48	0

The static covariates such as `age` and `surgery` are repeated over the multiple rows for a given patient. A minor modification is needed when there is a tie between the event or censoring time and the time at which a time-dependent covariate changes value. In this case, decrease the time for the time-dependent covariate slightly so it precedes the event or censoring time. For the heart transplant study for a patient who is transplanted and dies on day 5, the transplant time is set to 4.9 and the death is recorded at 5. Multiple test results are easily coded as well. For a patient with tests on days 0, 60, and 120, and follow-up to day 140, the data would be coded as three time intervals, 0–60, 60–120, and 120–140. This implicitly assumes that the time-dependent covariate is a step function with jumps at the measurement

points. Alternatively, you can break at the midpoints between the measurement times or interpolate the test measurements over smaller intervals of time. If test results vary *markedly* from visit to visit, interpolation of the measurements or redesign of the study may be required.

Discontinuous Intervals of Risk

In a study of tumor progression and it relationship to a particular blood marker, the key time-dependent variable is the monthly measurement of the marker. A few patients, however, had a gaps in their visit record. One choice for analysis is to interpolate these patients values over the missing time periods. An alternate, more conservative, course is to treat the values on the marker as missing. This strategy effectively removes these subjects from the risk set for the missing visit times, but they are not removed entirely from the study.

Another application of discontinuous risk intervals results when multiple events are possible, but the treatment for an event temporarily protects the patient from another event. In the study of hip-fracture in the elderly, hospitalization following a fracture protects the patient from further fractures. For studies with *low* event rates, discontinuous risk intervals will probably have little impact on the analysis.

Multiple Time Scales

The usual Cox model forms risk groups based on time since entry. For some studies a more logical grouping might be based on another alignment, such as age or time since diagnosis. An example is with Parkinson's disease patients. Natural history of the disease suggests that risk groups be based on the time since diagnosis. The Mayo Clinic is a referral center and frequently receives such patients sometime after diagnosis. Using the counting process formulation, the interval for a referred patient who is enrolled one year after diagnosis and who has an event in the second year is (1,2]. This patient is not in the risk set for an early enrollee with an event at six months. The risk set for the event at two years is all subjects. This is known as left truncation.

Time-Dependent Strata

Another case where alignment is a potential issue concerns time-dependent strata. The example is a study of Dutch patients with primary biliary cirrhosis of the liver (PBC). PBC is a rare but fatal chronic liver disease of unknown cause. The hazard rate for patients with the disease grows over time, as does the rate of degeneration in their hepatic function, tracked by various blood tests. A portion of the patients receive a liver transplant at some point during the follow-up. One objective of the study was to assess the value of covariates such as age and bilirubin in predicting patient outcome, both before and after transplantation. Transplant was treated as a time-dependent stratification variable. In the post transplant strata, the most natural hazard function is based on time since transplant. Surgical death is a major risk for such an extensive procedure, and this time scale properly aligns the patient's clock with the dominating hazard.

Proper alignment for time-dependent strata is not always clear. One appealing method of analysis for the myocardial infarction study is to place patients into new strata after each cardiac event. The baseline hazard for a patient with multiple events may be quite different than the group as a whole. It is not obvious, however, whether time since enrollment or time since last event is the better index of an appropriate risk group.

24.5 MORE DETAILED EXAMPLES

Complex Cox models usually involve time-dependent data which is handled by using the *counting process* notation developed by Andersen and Gill (1982). For a technical reference see Fleming and Harrington (1991). The examples in this section involve time-dependent variables in some way. In the first example, the Stanford Heart Transplant Study, the time dependency is on a binary covariate indicating whether the patient has had a heart transplant. For patients that received a heart transplant during the study, the `transplant` variable changes. The second example involves a bladder cancer study for patients with multiple occurrences of bladder tumors. The multiple events are modeled using the counting process notation and an additional notion of correlated responses.

Stanford Heart Transplant Study

The example below reproduces an analysis of the Stanford heart transplant study found in Kalbfleisch and Prentice (1980), section 5.5.3. The data itself is taken from Crowley and Hu (1977) because the values listed in the appendix of Kalbfleisch and Prentice are rounded and do not reproduce the results of their section 5.5. The covariates in the study, contained in the `heart` data frame, are described as follows:

`transplant`	patient received a heart transplant (1) or not (0)
`age`	(age at acceptance in days)/365.25 - 48
`year`	(date of acceptance in days since 1 Oct 1967) / 365.25
`surgery`	prior surgery (1=yes, 0=no),

The `transplant` variable is the only time-dependent variable. From the time of admission into the study until the time of death a patient was eligible for a heart transplant. The time to transplant depends on the next available donor heart with an appropriate tissue-type match. In the `heart` data frame, a transplanted patient is represented by two rows of data. The first is over the time period from enrollment (time 0) until the transplant, and has `transplant` = 0. The second is over the period from transplant to death or last follow-up and has `transplant` = 1. All other covariates are the same on the two lines. Subjects without a transplant are represented by a single row of data. Each row of data contains two variables `start` and `stop` which mark

the time interval (`start, stop`] for the data, as well as an indicator variable `event` which is 1 (one) if there was a death at time `stop` and 0 (zero) otherwise. For example, a subject who was transplanted at day 10 and followed up until day 31, has a first row of data corresponding to the time interval (0,10] and a second row corresponding to the interval (10,31]. Here is the code to fit the six models found in Kalbfleisch and Prentice. Note the use of the `options` call, which forces the factors to be coded as dummy variables. See the help file on `contr.treatment` for more details. Since the data set contains tied death times, you must use the Breslow approximation to match the coefficients that Kalbfleisch and Prentice produce. See the section Computations for Tied Deaths for more details on methods for handling ties.

```
> options(contrasts=c("contr.treatment", "contr.poly"))
> heart.fit1 <- coxph(Surv(start, stop, event) ~
+                     (age + surgery)*transplant,
+                     data = heart, method = "breslow")
> heart.fit2 <- coxph(Surv(start, stop, event) ~
+                     year * transplant,
+                     data = heart, method="breslow")
> heart.fit3 <- coxph(Surv(start, stop, event) ~
+                     (age + year)*transplant,
+                     data = heart, method="breslow")
> heart.fit4 <- coxph(Surv(start, stop, event) ~
+                     (year + surgery)*transplant,
+                     data= heart, method="breslow")
> heart.fit5 <- coxph(Surv(start, stop, event) ~
+                     (age + surgery)*transplant + year,
+                     data= heart, method="breslow")
> heart.fit6 <- coxph(Surv(start, stop, event) ~
+                     age*transplant + surgery + year,
+                     data= heart, method="breslow")
```

A summary of the first fit produces the following:

```
> summary(heart.fit1)
Call:
coxph(formula = Surv(start, stop, event) ~ (age +
        surgery) * transplant, data = heart, method =
        "breslow")

n= 172
```

```
                  coef         exp(coef) se(coef)      z
age               0.0138     1.014   0.0181  0.763
surgery          -0.5457     0.579   0.6109 -0.893
transplant        0.1181     1.125   0.3277  0.360
age:transplant    0.0348     1.035   0.0273  1.276
surgery:transplant -0.2916     0.747   0.7582 -0.385

                     p
age               0.45
surgery           0.37
transplant        0.72
age:transplant    0.20
surgery:transplant 0.70

                     exp(coef) exp(-coef) lower .95
age                     1.014      0.986     0.979
surgery                 0.579      1.726     0.175
transplant              1.125      0.889     0.592
age:transplant          1.035      0.966     0.982
surgery:transplant      0.747      1.339     0.169
                     upper .95
age                       1.05
surgery                   1.92
transplant                2.14
age:transplant            1.09
surgery:transplant        3.30

Rsquare= 0.07   (max possible= 0.969 )
Likelihood ratio test= 12.4 on 5 df,   p=0.0291
Wald test            = 11.6 on 5 df,   p=0.0402
Efficient score test = 12 on 5 df,   p=0.0345
```

Note that the sixth line of the summary indicates that `n = 172`. This is the number of *observations* in the study, not the number of subjects. There are actually 103 patients, of which 69 had a transplant and are thus represented using 2 rows of data. You can create a table of coefficients similar to Kalbfleisch and Prentice's table 5.2 as follows:

```
> var.names <- c("age","year","surgery","transplant",
+   "age:transplant", "year:transplant",
+   "surgery:transplant")
> round(rbind(heart.fit1$coef[var.names],
+   heart.fit2$coef[var.names], heart.fit3$coef[var.names],
```

```
+   heart.fit4$coef[var.names], heart.fit5$coef[var.names],
+   heart.fit6$coef[var.names]), digits = 4)
       age   year surgery transplant age:transplant
[1,] 0.014     NA  -0.546      0.118          0.035
[2,]    NA -0.265      NA     -0.282             NA
[3,] 0.016 -0.274      NA     -0.588          0.034
[4,]    NA -0.254  -0.236     -0.292             NA
[5,] 0.015 -0.136  -0.419      0.077          0.027
[6,] 0.015 -0.136  -0.621      0.047          0.027
     year:transplant surgery:transplant
[1,]              NA             -0.292
[2,]           0.136                 NA
[3,]           0.201                 NA
[4,]           0.164             -0.550
[5,]              NA             -0.298
[6,]              NA                 NA
```

When there are time-dependent covariates, the predicted survival curve can present something of a dilemma. The usual call to `survfit` is for a *pseudo* cohort whose covariates do not change:

```
> heart.surv1 <- survfit(heart.fit2,
+                    data.frame(year=2, transplant=0) )
> heart.surv2 <- survfit(heart.fit2,
+                    data.frame(year=2, transplant=1) )
```

The second curve, `heart.surv2`, represents a cohort of patients whose `transplant` variable is always 1, even on day 0, that is, patients who had no waiting time for a transplant. There were none of these in the study, so just what does it represent? Time-dependent covariates that represent repeated measurements on a patient, such as a blood enzyme level, are particularly problematic. With time-dependent covariates, it is easy to create predicted survival curves for "patients" that never would or perhaps never could exist.

Because the model depends on the time-dependent covariate, `transplant`, a proper predicted survival requires specification of a *future covariate history* for the patient in question. (See the discussion of *internal* and *external* covariates in section 5.3 of Kalbfleisch and Prentice for a more complete exposition on these issues.) It is possible to obtain the projected survival for some particular pattern of change in the covariates by supplying a multiple-line data frame that reflects that pattern and setting `individual = T`. The example below produces the survival curve for a cohort aged 50 with prior surgery and a transplant at 6 months. That is, over the time interval (0,.5] the covariate set is (50, 1, 0), and over the time interval (.5, 3] it is (50, 1, 1). Note that `start` and `stop` times are in days rather than years. In order to

specify the time points the failure time variables, `start`, `stop`, and `event`, must be specified in the data frame as well as the covariates, though the value for `event` will be ignored.

```
> newdata <- data.frame(start=c(0,183), stop=c(183,3*365),
+     event=c(1,1), age=c(50,50), surgery=c(1,1),
+     transplant=c(0,1))
> survfit(heart.fit1, newdata, individual=T)
```

Bladder Cancer Study

This example is taken from the paper by Wei, Lin, and Weissfeld (1989). The study is of time to recurrence of bladder cancer and the data is contained in the `bladder` data frame. The `bladder` data frame has either 4 or 5 rows for each subject. Each subject had four recurrences of bladder cancer and some were followed beyond the fourth recurrence. The variables in `bladder` are defined as follows:

`id`	patient ID
`rx`	treatment group (1 = placebo, 2 = thiopeta)
`size`	size of the largest initial tumor
`number`	the number of initial tumors
`start`	entry into the study or the time of last recurrence
`stop`	time to event (months)
`event`	indicator of cancer recurrence (1) or censoring (0)
`enum`	number of recurrences of bladder cancer

A summary of `bladder` follows:

```
> summary(bladder)
       id              rx              size
 Min.   : 1   Min.   :1.000   Min.   :1.000
 1st Qu.:22   1st Qu.:1.000   1st Qu.:1.000
 Median :43   Median :1.000   Median :1.000
 Mean   :43   Mean   :1.447   Mean   :2.106
 3rd Qu.:64   3rd Qu.:2.000   3rd Qu.:3.000
 Max.   :85   Max.   :2.000   Max.   :8.000

     number           stop            event
 Min.   :1.000   Min.   : 1.00   Min.   :0.0000
 1st Qu.:1.000   1st Qu.:12.00   1st Qu.:0.0000
 Median :1.000   Median :25.00   Median :0.0000
 Mean   :2.012   Mean   :25.06   Mean   :0.3294
```

```
 3rd Qu.:3.000   3rd Qu.:37.00   3rd Qu.:1.0000
 Max.   :7.000   Max.   :59.00   Max.   :1.0000

      enum
 Min.   :1.00
 1st Qu.:1.75
 Median :2.50
 Mean   :2.50
 3rd Qu.:3.25
 Max.   :4.00
```

We create two data frames for analysis. The first one has only the first four rows for each subject and has `start` removed.

```
> bladder1 <- bladder[bladderenum<5,]
> bladder1start <- NULL
```

The second ones has removed all rows for which `start` and `stop` are equal.

```
> bladder2 <- bladder[bladderstart< bladderstop, ]
```

WLW fit four separate models, one for each recurrence, and then combined the results. The first of the individual fits is based on time from the start of the study until the first event, for all patients; the second fit is based on time from the start of the study until the second event, again for all patients, etc. The model estimated by WLW is fit by the following commands. The key addition to the model is `cluster(id)`, which asserts that subjects with the same value of the variable `id` may be correlated. In order to compare the results directly to Wei, Lin, and Weissfeld (1989), we first set the factor contrasts to `"contr.treatment"`.

```
> options(contrasts='contr.treatment')
> wfit <- coxph(Surv(stop, event) ~ (rx + size + number)*
+ strata(enum) + cluster(id), bladder1, method='breslow')
> rx <- c(1,4,5,6) # coefficients for the treatment effect
> cmat <- diag(4); cmat[,1] <- 1 # contrast matrix
> cmat %*% wfit$coef[rx]   # coefs in WLW (table 5)
           [,1]
[1,] -0.5175702
[2,] -0.6194396
[3,] -0.6998691
[4,] -0.6504161
> wvar <- cmat %*% wfit$var[rx,rx] %*% t(cmat)
>           # var matrix (eqn 3.2)
> sqrt(diag(wvar))
[1] 0.3075006 0.3639071 0.4151602 0.4896743
```

The same coefficients can also be obtained, as WLW do, by performing four separate fits but it takes more work. A major advantage of the fitting the model as above is that it allows us to fit submodels that are not available using separate fits for each stratum. In particular, the model

```
> Surv(stop, event) ~ rx + (size + number) * strata(enum)
+        + cluster(id)
```

differs only in that there is no treatment by strata interaction, and gives an average treatment coefficient of -.60, which is near to the weighted average of the marginal fits (based on the diagonal of `wvar`) suggested by WLW. WLW also give the results for two suggestions proposed by Prentice *et al.* (1981). For time to first event these are the same as above. For the second event they use only patients who experienced at least one event, and use either the time from start of study (method a) or the time since the occurrence of the last event (method b). The code for these is follows:

```
> fit2pa <- coxph(Surv(stop, event) ~ rx + size + number,
+ bladder2, subset = (enum==2))
> fit2pb <- coxph(Surv(stop-start, event) ~ rx + size +
+ number, bladder2, subset = (enum==2))
```

Lastly, the authors also make use of an Andersen-Gill model for comparison. This model has the advantage that it uses all of the data directly, but because of correlation it may underestimate the variance of the relevant coefficients. A method to address this is given in a paper by Lee, Wei, and Amato (1992); it is essentially the same method found in the WLW paper. This method for variance estimation is invoked by specifying the `cluster(id)` term.

```
> afit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number + cluster(id), data=bladder2)
> afit
Call:
coxph(formula = Surv(start, stop, event) ~ rx + size +
number + cluster(id), data = bladder2)

          coef exp(coef) se(coef) robust se      z     p
    rx -0.4116     0.663   0.1999    0.2415 -1.704 0.088
  size  0.1637     1.178   0.0478    0.0569  2.876 0.004
number -0.0411     0.960   0.0703    0.0723 -0.568 0.570

Likelihood ratio test=14.7  on 3 df, p=0.00213 n= 190
> sqrt(diag(afit$var))
[1] 0.24151999 0.05690736 0.07228107
```

```
> sqrt(diag(afit$naive.var))
[1] 0.19989234 0.04776578 0.07029462
```

The naive estimate of standard error is .20, the correct estimate of .24 is intermediate between the naive estimate and the linear combination estimate. Further discussion on these estimators can be found in the section Robust Variance Estimation.

24.6 ADDITIONAL TECHNICAL DETAILS

The remaining subsections provide additional details on computations and options available for fitting proportional hazards models, including:

- the handling of ties
- the effect of ties on the definitions of residuals
- tests for proportional hazards
- robust variance estimation
- the handling of case weights
- details about the computations of `coxph`

Computations for Tied Deaths

For untied data, the terms in the partial likelihood (equation (24.1)) look like

$$\left(\frac{r_1}{\sum_i r_i}\right)\left(\frac{r_2}{\sum_{i>1} r_i}\right)\cdots,$$

where $r_1, r_2, \ldots, r_n$ are the subject risk scores. Assume that the real data are continuous, but the recorded data have tied death times. For example, several subjects might die on the first day of their hospital stay but they do not all perish at the same moment. For a simple example, assume 5 subjects, ordered by time of death or censoring, are in a study and the first two die at the same recorded time. If the time data had been more precise, then the first two terms in the likelihood would be either

$$\left(\frac{r_1}{r_1+r_2+r_3+r_4+r_5}\right)\left(\frac{r_2}{r_2+r_3+r_4+r_5}\right)$$

or

$$\left(\frac{r_2}{r_1+r_2+r_3+r_4+r_5}\right)\left(\frac{r_1}{r_1+r_3+r_4+r_5}\right).$$

Notice that the numerators remain constant, but the denominators do not.

The question is how do you approximate the the correct term for the likelihood?

The Breslow approximation is the most commonly used because it is the easiest to program. It simply uses the complete sum, $r_1 + r_2 + r_3 + r_4 + r_5$, for both denominators. Clearly, if the proportion of ties is large this will deflate the partial likelihood.

The Efron approximation uses $.5r_1 + .5r_2 + r_3 + r_4 + r_5$ as the second denominator, based on the idea that r_1 and r_2 each have a 50% chance of appearing in the "true" second term. If there were 4 tied deaths, then the ratios for r_1 to r_4 would be 1, 3/4, 1/2, and 1/4 in each of the four denominator terms, respectively. Though it is not widely used, the Efron approximation is only slightly more difficult to program than the Breslow version. In particular, since the down-weighting is independent of any case weights and thus of b, the form of the derivatives of the likelihood is unchanged.

An alternate approach attempts an "exact" computation. The exact partial likelihood, comes from viewing the data as genuinely discrete. The denominator in this case is $\sum_{i \neq j} r_i r_j$ if there are two subjects tied, $\sum_{i \neq j \neq k} r_i r_j r_k$ if there are three subjects tied, etc.

When using the `coxph` function to fit proportional hazards models, you can specify any of the above three methods for handling ties. The default is the Efron approximation (`method = "efron"`). The other two may be specified by setting `method = "breslow"` or `method = "exact"`. Note that when there are no ties, all three methods produce the same likelihood function.

Effect of Ties on Residual Definitions

The Efron approximation induces changes in the residuals' definitions. In particular, the Cox score statistic is still

$$U = \sum_{i=1}^{n} \int_0^{\infty} (Z_i(s) - \bar{Z}(s)) dN_i(s), \tag{24.8}$$

but the definition of $\bar{Z}(s)$ has changed if there are tied deaths at time *s*. If there are *d* deaths at *s*, then there are *d* different values of $\bar{Z}$ used at the time

point. The Schoenfeld residuals use $\bar{Z}$, the average of these d values, in the computation. The martingale and score residuals require a new definition of $\hat{\Lambda}$. If there are d tied deaths at time t, we again assume that in the exact (but unknown) untied data there are events and corresponding jumps in the cumulative hazard at $t \pm \epsilon_1 < \ldots < t \pm \epsilon_d$. Then each of the tied subjects will in expectation experience all of the first hazard increment, but only $(d\text{-}1)/d$ of the second, $(d\text{-}2)/d$ of the next, and etc. If we equate observed to expected hazard at each of the d deaths, then the total increment in hazard at the time point is the sum of the denominators of the weighted means. Returning to our earlier example of 5 subjects of which 1 and 2 have tied deaths:

$$d\hat{\Lambda}(t) = \frac{1}{r_1 + r_2 + r_3 + r_4 + r_5} + \frac{1}{r_1/2 + r_2/2 + r_3 + r_4 + r_5}.$$

For the null model where r_i=1 for all i, this agrees with the suggestion of Nelson (1969) to use 1/5+1/4 rather than 2/5 as the increment to the cumulative hazard. The formula for the score residuals is demonstrated using, again, our previous example with five subjects the first two being tied. For subject one the residual at time one is the sum $a+b$ where

$$a = \left(Z_1 - \frac{r_1 Z_1 + r_2 Z_2 + \ldots + r_5 Z_5}{r_1 + r_2 + \ldots + r_5}\right)\left(\frac{dN_1}{2} - \frac{r_1}{r_1 + r_2 + \ldots + r_5}\right) \text{ and}$$

$$b = \left(Z_1 - \frac{r_1 Z_1/2 + r_2 Z_2/2 + \ldots + r_5 Z_5}{r_1/2 + r_2/2 + \ldots + r_5}\right)\left(\frac{dN_1}{2} - \frac{r_1/2}{r_1/2 + r_2/2 + \ldots + r_5}\right).$$

This product does not neatly collapse into $(Z_1 - \bar{Z})d\hat{M}$ but is easy to compute. The connection between residuals and the exact partial likelihood is not as precise and are thus not implemented. If residuals are requested after a Cox fit with `method = "exact"` the Breslow formulae are used.

Tests for Proportional Hazards

The key ideas of this section are taken from Grambsch and Therneau (1994). Most of the common alternatives to the hypothesis test of proportional hazards can be cast in terms of a *time-varying coefficient* model. That is, we assume that

$$\lambda(t;Z) = \lambda_0(t) e^{\beta_1(t) Z_1 + \beta_2(t) Z_2 + \ldots}.$$

(If Z_j is a 0/1 covariate such as treatment, this formulation is completely general in that it encompasses all alternatives to proportional hazards.) The proportional hazards assumption is then a test for $\beta(t) = \beta$, which is a test for zero slope in the appropriate plot of $\hat{\beta}(t)$ on t. Let i index subjects, j index

variables, and k index the death times. Then let s_k be the Schoenfeld residual and V_k be the contribution to the information matrix (equation (24.4)) at time t_k. Define the rescaled Schoenfeld residual as

$$s_k^* = \hat{\beta} + s_k V_k^{-1}.$$

The main results are:

- $E(s_k^*) = \beta(t_k)$, so that a smoothed plot of s^* versus time gives a direct estimate of $\hat{\beta}(t)$.
- Many of the common tests for proportional hazards are linear tests for zero slope, applied to the plot of s^* versus $g(t)$ for some function g. In particular, the Z:PH test popularized in the SAS PHGLM procedure corresponds to $g(t)$ = rank of the death time. The test of Lin (1991) corresponds to $g(t) = K(t)$, where K is the Kaplan-Meier.
- Confidence bands, tests for individual variables, and a global test are available, and all have the fairly standard "linear models" form.
- The estimates and tests are affected very little if the individual variance estimates V_k are replaced by their global average $\bar{V} = \sum V_k / d = I/d$. Calculations then require only the Schoenfeld residuals and the standard Cox variance estimate I^{-1}.

For the global test, let $g(t)$ be the desired transformation of time, and $g_k = g(t_k)$ be the value of g at the kth death time. Then

$$T = \left(\sum g_k s_k\right)' D^{-1} \left(\sum g_k s_k\right)$$

is asymptotically χ^2 on p degrees of freedom, where

$$D = \sum g_k^2 V_k - \left(\sum g_k V_k\right)\left(\sum V_k\right)^{-1}\left(\sum g_k V_k\right)'.$$

Because the s_k sum to zero, a little algebra shows that the above expression is invariant if g_k is replaced by g_k - c for any constant c. Subtraction of a mean will, however, result in less computer round-off error. A further simplification

occurs by using $\bar{V}$, leading to

$$T = \left[\sum (g_k - \bar{g}) s_k\right]' \left[\frac{dI^{-1}}{\sum (g_k - \bar{g})^2}\right] \left[\sum (g_k - \bar{g}) s_k\right]. \qquad (24.9)$$

For a given covariate j, the diagnostic plot will have s^*_{kj} on the vertical axis and g_k on the horizontal. The variance matrix of s^*_{kj} is $\Sigma_j = (A - cJ) + cI$, where A is a $d \times d$ diagonal matrix whose kth diagonal element is $V^{-1}_{k,jj}$, $c = I^{-1}_{jj}$, J is a $d \times d$ matrix of ones and I is the identity matrix. The constant cI reflects the uncertainty in s^* due to the $\hat{\beta}$ term. If only the shape of $\beta(t)$ is of interest (for example, is it linear or sigmoid) the c could be dropped. If absolute values are important (for example, $\beta(t)=0$ for $t>2$ years) it should be retained. For smooths that are linear operators, such as splines or the loess function, the final smooth is $\hat{s}* = Hs*$ for some matrix H. Then $\hat{s}*$ is asymptotically normal with mean 0 and variance $H\Sigma_j H'$. Standard errors are computed using ordinary linear model methods. If V_k is replaced with $\bar{V}$, then S_j simplifies to $I^{-1}_{jj}((d+1)I - J)$. With the same substitution, the component-wise test for linear association is

$$t_j = \frac{\sum (g_k - \bar{g}) y_k}{\sum dI^{-1}_{jj} (g_k - \bar{g})^2} \qquad (24.10)$$

The `cox.zph` function uses equation (24.9) as a global test of proportional hazards, and equation (24.10) to test individual covariates. The plot method for `cox.zph` uses a natural spline smoother. Confidence bands for the smooth are based on the full covariance matrix, with $\bar{V}$ replacing V_k.

Though the simulations in Grambsch and Therneau (1993) did not uncover any situations where the simpler formulae based on $\bar{V}$ are less reliable, such cases could arise. The substitution trades a possible increase in bias for a substantial reduction in the variance of the individual V_k. It is likely to be unwise in those cases where the variance of the covariates, within the risk sets,

differs substantially between different risk sets. Two examples come to mind. The first would be a stratified Cox model, where the strata represent different populations. In a multi-center clinical trial, for instance, inner city, Veterans Administration, and suburban hospitals often service quite disparate populations. In this case a separate average $\bar{V}$ should be formed for each strata. A second example is where the covariate mix changes markedly over time, perhaps because of aggressive censoring of certain patient types. These cases have not been addressed directly in the software. However, `coxph.detail` returns all of the V_k matrices, which can then be used to construct specialized tests for such situations.

Clearly, no one scaling function $g(t)$ will be optimal for all situations. The `cox.zph` function directly supports four common choices: identity, log, rank, and 1 – Kaplan-Meier. By default, it will use the last of these, based on the following rationale. Since the test for proportional hazards is essentially a test for significant regression of the scaled residual modeled linearly in the g_k, we would expect this test to be adversely effected if there are outliers in the g_k. We would also like the test to be at most mildly affected by the censoring pattern of the data. The Kaplan-Meier transform appears to satisfy both of these criteria.

Robust Variance Estimation

The following technical discussion of robust variance estimation for Cox models leads to a rather simple implementation conceptually. The basic idea is to compute an approximate matrix of changes in estimated coefficients, L, resulting from leaving out each observation one at a time. The robust estimate of variance is then $L'L$. $L'L$ relates to other variance estimators as follows:

- $L'L$ is equivalent to the "working independence" estimate in generalized estimating equations models.
- $L'L$ is an approximate jackknife estimate of variance.
- $L'L$ is equivalent to the Wei, Lin, and Weissfeld (1989) variance estimate for a Cox model.
- $L'L$ is a robust *sandwich estimate* as discussed in Huber (1967).

If the observations are grouped and correlated within groups, the above idea works if entire groups (rather than individual observations) are left out for computing the approximate jackknife variance estimate. This case corresponds to Cox models with a counting process formulation and multiple observations per subject. The resulting estimator of variance is called the *grouped jackknife* estimator.

The Sandwich Estimator

The following discussion describes the general sandwich estimator, a modification of the sandwich estimator for grouped data, and implementation for Cox models. Robust variance calculations are based on the *sandwich estimate*

$$V = ABA'$$

where $A^{-1} = I$ is the usual information matrix, and B is a "correction term". The genesis of this formula can be found in Huber (1967), who discusses the behavior of any solution to an estimating equation

$$\sum_{i=1}^{n} \phi(x_i, \hat{\beta}) = 0.$$

Of particular interest is the case of a maximum likelihood estimate based on distribution f (so that $\phi = \partial log(f)/\partial\beta$), when in fact the data are observations from distribution g. Then, under appropriate conditions, $\hat{\beta}$ is asymptotically normal with mean β and covariance $V = ABA'$, where

$$A = \left(\frac{\partial E\Phi(\beta)}{\partial\beta}\right)^{-1}$$

and B is the covariance matrix for $\Phi = \sum\phi(x_i, \beta)$. Under most situations the derivative can be moved inside the expectation, and A will be the inverse of the usual information matrix. This formula was rediscovered by White (1980), (1982), and is also known in the econometric literature as White's method. Under the common case of maximum likelihood estimation we have

$$\sum_{i=1}^{n} \phi(x_i, \hat{\beta}) = \sum_{i=1}^{n} \frac{\partial log f(x_i)}{\partial\beta}$$

$$\equiv \sum_{i=1}^{n} u_i(\beta).$$

By interchanging the order of the expectation and the derivative, A^{-1} is the expected value of the information matrix, which will be estimated by the observed information I. Since $E[u_i(\beta)] = 0$,

$$B = \text{var}(\Phi) = E(\Phi^2)$$
$$= \sum_{i=1}^{n} E[u'_i(\beta)u_i(\beta)] + \sum_{i \neq j}^{n} E[u'_i(\beta)u_i(\beta)] \quad (24.11)$$

where $u_i(\beta)$ is assumed to be a row vector. If the observations are independent, then the u_i will also be independent and the cross terms in equation (24.11) will be zero. A natural estimator of B is

$$\hat{B} = \sum_{i=1}^{n} u'_i(\hat{\beta})u_i(\hat{\beta})$$
$$= U'U,$$

where U is the matrix of *score residuals*, the ith row of U equals $u_i(\hat{\beta})$. The column sums of U are the efficient score vector Φ.

As a simple example consider generalized linear models. McCullagh and Nelder (1989) maintain that overdispersion "is the norm in practice and nominal dispersion the exception." To account for overdispersion they recommend inflating the nominal covariance matrix of the regression coefficients $A = (X'WX)^{-1}$ by a factor

$$c = \sum_{i=1}^{n} \frac{(y_i - \mu_i)^2}{V_i} / (n-p),$$

where V_i is the nominal variance. Smith and Heitjan (to appear) show that AB may be regarded as a multivariate version of this variance adjustment factor, and that c and AB may be interpreted as the average ratio of actual variance $(y_i - \mu_i)^2$ to nominal variance V_i. By premultiplying by AB, each element of the nominal variance-covariance matrix A is adjusted differentially for departures from nominal dispersion.

Modified Sandwich Estimator

When the observations are not independent, the estimator B must be adjusted accordingly. The "natural" choice $\left(\sum u_i\right)^2$ is not available of course, since $\Phi(\hat{\beta}) = 0$ by definition. However, a reasonable estimate is available when the correlation is confined to subgroups. In particular, assume

that the data comes from clustered sampling with j=1, 2, …, k clusters, where there may be correlation within clusters but observations from different clusters are independent. Using equation (24.11), the cross-product terms between clusters can be eliminated, and the resulting equation rearranged as

$$var(\Phi) = \sum_{j=1}^{k} \tilde{u}(\beta)'\tilde{u}(\beta),$$

where $\tilde{u}_j$ is the sum of u_i over all subjects in the jth cluster. This leads to the *modified sandwich estimator*

$$V = A(\tilde{U}'\tilde{U})A,$$

where the collapsed score matrix $\tilde{U}$ is obtained by replacement of each cluster of rows in U by the sum of those rows. If the total number of clusters is small, then this estimate will be sharply biased towards zero, and some other estimate must be considered. In fact, $rank(V) < k$, where k is the number of clusters. Asymptotic results for the modified sandwich estimator require that the number of clusters tend to infinity.

Implementation for Cox Models

Application of these results to the Cox model proceeds by defining a weighted Cox partial likelihood and letting

$$u_i(\beta) \equiv \left(\frac{\partial U}{\partial w_i}\right)_{w=1},$$

where w is the vector of weights. This approach is used by Cain and Lange to define a leverage or influence measure for Cox regression. In particular, they derive the leverage matrix

$$L = UI^{-1},$$

where L_{ij} is the approximate change in β_j when observation i is removed from the data set. Their estimate can be recognized as a form of the *infinitesimal jackknife* (see, for example, the discussion in Efron (1982) for the linear models case).

The connection to the jackknife is quite general. For any model stated as an estimating equation, the Newton-Raphson iteration has step

$$\Delta\beta = 1'(UI^{-1}),$$

the column sums of the matrix $L = UI^{-1}$. At the solution β the iteration's

step size is, by definition, zero. Consider the following approximation to the jackknife

1. treat the information matrix I as fixed
2. remove observation i
3. beginning at the full data solution $\hat{\beta}$, do one Newton-Raphson iteration.

This is equivalent to removing one row from L, and using the new column sum as the increment. Since the column sums of $L(\hat{\beta}) = 0$ are zero, the increment must be $\Delta\beta = -L_{i.}$. That is, the rows of L are an approximation to the jackknife, and the sandwich estimate of variance $L'L$ is an approximation to the jackknife estimate of variance. Lin and Wei (1989) show the applicability of Huber's work to the partial likelihood, and derive the ordinary Huber sandwich estimate $V = I^{-1}(U'U)I^{-1} = L'L$, the approximate jackknife. When the data are correlated, the appropriate form of the jackknife is to leave out an entire *subject* at time, rather than one observation, that is, the grouped jackknife. To approximate this, we leave out groups of rows from L, leading to $\tilde{L}'\tilde{L}$ as the approximation to the jackknife.

Examples:

Lee, Wei, and Amato (1992) consider highly stratified data sets which arise from inter-observation correlation. As an example they use paired eye data on visual loss due to diabetic retinopathy, where photocoagulation was randomly assigned to one eye of each patient. There are $n/2$=1742 clusters (patients) with 2 observations per cluster. Treating each pair of eyes as a cluster, they derive the modified sandwich estimate $V = \tilde{L}'\tilde{L}$, where $\tilde{L}$ is derived from L in the following way. L will have one row, or observation, per eye. Because of possible correlation, we want to reduce this to a leverage matrix $\tilde{L}$ with one row per individual. The leverage (or row) for an individual is simply the sum of the rows for each of their eyes. (A subject, if any, with only one eye would retain that row of leverage data unchanged). The resulting estimator is shown to be much more efficient than analysis stratified by cluster. A second example given in Lee, Wei, and Amato concerns a litter-matched experiment. In this case the number of rats per litter may vary.

Wei, Lin, and Weissfeld (1989) consider multivariate survival times. An example is the measurement of both time to progression of disease and time to death for a group of cancer patients. The data set again contains $2n$ observations, time and status variables, subject id, and covariates. It also contains an indicator variable `etype` to distinguish the event type,

progression vs. survival. The suggested model is stratified on event type, and includes all strata × covariate interaction terms. One way to do this with `coxph` is

```
> fit2 <- coxph(Surv(time,status) ~ (rx + size + number)*
+ strata(etype))
> Ltilde <- residuals(fit2, type='dfbeta',
+ collapse=subject.id)
> newvar <- t(Ltilde)
```

The per *subject* leverage matrix $\tilde{L}'\tilde{L}$ is `newvar`. An alternate way to do this is

```
> fit2a <- coxph(Surv(time,status) ~ (rx + size + number)*
+ strata(etype) + cluster(id))
```

The `cluster` argument asserts that subjects with the same value of `id` may be correlated. The data for fitting the above two models is not built into S-PLUS. However, similar computations can be performed using the `bladder` data frame for comparison. Two ways of producing the robust variance estimate follow:

```
> bladder2 <- bladder[bladder$start< bladder$stop, ]
> afit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number + cluster(id), data=bladder2)
> sqrt(diag(afitvar))
[1] 0.24151999 0.05690736 0.07228107
```

Now doing it an alternate way:

```
> bfit <- coxph(Surv(start, stop, event) ~ rx + size +
+ number, data = bladder2)
> db <- resid(bfit, type="dfbeta", collapse = bladder2$id)
> sqrt(diag(t(db)
[1] 0.24876453 0.05842243 0.07421445
```

Using the grouped jackknife approach, as suggested here, rather than separate fits for each event type has some practical advantages:

- It is easier to program, particularly when the number of events per subject is large.
- Other models can be encompassed, in particular one need not include all of the strata × covariate interaction terms.
- There need not be the same number of events for each subject. The method for building up a joint variance matrix requires that all of the score residual matrices be of the same dimension, which is not the case if information on one of the failure types was not collected for some subjects.

Weighted Cox Models

A Cox model that includes case weights has been suggested by Binder (1992) in the context of survey data. If w_i are the weights, then the modified score statistic is

$$U(\beta) = \sum_{i=1}^{n} w_i u_i(\beta). \tag{24.12}$$

The individual terms u_i are still $Z_i(t) - \bar{Z}(t)$ but the weighted mean $\bar{Z}$ is changed in the obvious way to include both the risk weights r and the external weights w. The information matrix can be written as $I = \sum \delta_i w_i v_i$, where δ_i is the censoring variable and v_i is a weighted covariance matrix. The definition of v_i changes in the obvious way from equation (24.4). If all of the weights are integers, then for the Breslow approximation this reduces to ordinary case weights, that is, the solution is identical to what you obtain by replicating each observation w_i times. With the Efron approximation or the exact partial likelihood approximation replication of a subject results in a correction for ties. The `coxph` function allows general case weights. Residuals from the fit are such that the sum of weighted residuals is zero, and the returned values from the `coxph.detail` function are the individual terms u_i and v_i, so that U and I are weighted sums. The sandwich estimator of variance has $L'WL$ as its central term, where W is the diagonal matrix of weights. The estimate of $\hat{\beta}$ and the sandwich estimate of its variance are unchanged if each w_i is replace by cw_i for any $c>0$.

For either of the Breslow or the Efron approximations, the extra programming to handle weights is modest. For the Breslow method the logic behind the addition is straightforward, and corresponds to the derivation given above. For tied data and the Efron approximation, the formula is based on extending the basic idea of the approximation,

$$E(f(r_1, r_2, \ldots)) \approx f(E(r_1), E(r_2), \ldots)$$

to include the weights, as necessary. Returning to the simple example of the section Computations for Tied Deaths, the second term of the partial likelihood is either

$$\left(\frac{w_1 r_1}{w_1 r_1 + w_3 r_3 + w_4 r_4 + w_5 r_5}\right)$$

or

$$\left(\frac{w_2 r_2}{w_2 r_2 + w_3 r_3 + w_4 r_4 + w_5 r_5}\right).$$

To compute the Efron approximation, separately replace the numerator with $.5(w_1 r_1 + w_2 r_2)$ and the denominator with $.5w_1 r_1 + .5w_2 r_2 + w_3 r_3 + w_4 r_4 + w_5 r_5$.

An exciting use of weights is presented in Pugh *et al.* (1993), for inference with missing covariate data. Let π_i be the probability that none of the covariates for subject i is missing, and p_i be an indicator function which is 0 if any of the covariates actually is `NA`, so that $E(p_i) = \pi_i$. The usual strategy is to compute the Cox model fit over only the complete cases, that is, those with p_i=1. If information is not missing at random, this can lead to serious bias in the estimate of $\hat{\beta}$. A weighted analysis with weights of p_i/π_i will correct for this imbalance. There is an obvious connection between this idea and survey sampling. Both reweight cases from underrepresented groups.

In practice π_i will be unknown, and the authors suggest estimating it using a logistic regression with p_i as the dependent variable. The covariates for the logistic regression may be some subset of the Cox model covariates (those without missing information), as well as others. In an example, the authors use a logistic model with follow-up time and status as the predictors. Let T be the matrix of score residuals from the logistic model, that is,

$$T_{ij} = \frac{\partial}{\partial \alpha_j}[p_i \log \pi_i(\alpha) + (1 - p_i)\log(1 - \pi_i(\alpha))],$$

where α are the coefficients of the fitted logistic regression. Then the estimated variance matrix for $\hat{\beta}$ is the sandwich estimator $I^{-1}BI^{-1}$, where

$$B = U'U - [U'T][T'T]^{-1}[T'U].$$

This is equivalent to first replacing each row of U with the residuals from a regression of U on T, and then forming the product $U'U$. Note that if the logistic regression is completely uninformative ($\hat{\pi}_i$ = constant), this reduces to the ordinary sandwich estimate.

Computations

The `coxph` function is used to fit Cox proportional hazards models. The input data is assumed to consist of observations or rows of data, each of which contains the covariate values Z, a status indicator variable (`1 = event, 0 = censored`), an optional stratum indicator variable

(referenced by the `strata` function), along with the time interval (*start*, *stop*] over which this information applies. This means that each row is treated as a separate subject whose Y_i variable is 1 (one) on the interval (*start*, *stop*] and 0 (zero) otherwise. and that the risk set at time *t* only uses the applicable rows of the data.

The code for `coxph` does not specifically accommodate time-dependent covariates, time-dependent strata, multiple events, or any of the other special features mentioned. Consequently, *it is your responsibility to construct an appropriate data set*. This strategy leads to a fitting program that is simpler, shorter, easier to debug, and more computationally efficient than one with multiple specific options. A significantly more important benefit is that the flexibility inherent in building the proper data set allows analyses not originally considered—left truncation is a case in point.

The more common way to deal with time-dependent Cox models is to do a computation for *each* death time. For example, BMDP and SAS PHREG do this. One advantage of this over the algorithm implemented in `coxph` is the ability to code *continuously* varying time-dependent covariates. The `coxph` function only accommodates step functions. However, this does not appear to be a deficiency in practice. For the common case of repeated measurements on each subject, the data for `coxph` are quite easy to set up since they correspond to the original measurements of one line of data per visit.

The `coxph` function typically runs much faster when there are stratification variables in the model. When strata are introduced, `coxph` spends less time locating the current risk set because it only looks within the stratum it is estimating.

If the start time is omitted, it is assumed to be zero for all cases. In this case the algorithm is equivalent to the standard Cox model.

24.7 REFERENCES

Andersen, P. K. and Gill, R. D. (1982). *Cox's regression model for counting processes: A large sample study.* Annals of Statistics, 10:1100–1120.

Binder, D.A. (1992). *Fitting Cox's proportional hazards models from survey data*. Biometrika, 79:139–147.

Chambers, J. M. and Cleveland, W. S. and Kleiner, B. and Tukey, P. A. (1983). *Graphical Methods for Data Analysis*. Wadsworth, Belmont, CA.

Cox, D. R. (1972). *Regression models and life-tables*. Journal of the Royal Statistical Society, Series B, 34:187–202.

Crowley, J. and Hu, M. (1977). *Covariance analysis of heart transplant data*.

Journal of the American Statistical Association, 72:27–36.

Efron, B. (1977). *The efficiency of Cox's likelihood function for censored data*. Journal of the American Statistical Association, 72:557–565.

Efron, B. (1982). *The jackknife, the bootstrap, and other resampling plans*. Technical Report CMBS-NSF Monograph 38, SIAM.

Fleming, T. and Harrington, D. (1991). *Counting Processes and Survival Analysis*. Wiley, New York.

Grambsch, P. and Therneau, T.M. (1994). *Proportional hazards tests and diagnostics based on weighted residuals*. Biometrika, 81:515–526.

Harrell, F. (1986). *The PHGLM procedure*. SAS Supplemental Library User's Guide, Version 5. SAS Institute, Inc., Cary, NC.

Huber, P.J. (1967). *The behavior of maximum likelihood estimates under non-standard conditions*. In Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, 1:221–233.

Kalbfleisch, J. and Prentice, R. L. (1980). *The Statistical Analysis of Failure Time Data*. Wiley, New York.

Lee, E.W., Wei, L.J., and Amato, D. (1992). *Cox-type regression analysis for large number of small groups of correlated failure time observations*. In J.P Klein and P.K. Goel, editors, Survival Analysis, State of the Art, pages 237–247. Kluwer Academic Publishers, Netherlands.

Lin, D.Y. and Wei, L.J. (1989). *The robust inference for the Cox proportional hazards model*. Journal of the American Statistical Association, 84:1074–1079.

Lin, D.Y. (1991). *Goodness-of-fit analysis for the Cox regression model based on a class of parameter estimators*. Journal of the American Statistical Association, 86:725--728.

Lin, D.Y., Wei, L.J., and Ying, Z. (1992). *Checking the Cox model with cumulative sums of martingale-based residuals*. Technical Report #111, Dept. of Biostatistics, U. of Washington.

McCullagh, P. and Nelder, J.A. (1989). *Generalized Linear Models*, 2nd edition. Chapman and Hall, London.

Oakes, D. (1977). *The asympotic information in censored survival data*. Biometrika, 64:441-448.

Pugh, M., Robins, J., Lipsitz, S., and Harrington, D. (1993). *Inference in the Cox proportional hazards model with missing covariate data*, in press.

Prentice, R.L., Williams, B.J., and Peterson, A.V. (1981). *On the regression analysis of multivariate failure time data*. Biometrika, 68:373--89.

Schonfeld, D. (1982). *Partial residuals for the proportional hazards regression*

mode. Biometrika, 69:239-241.

Smith, P.J. and Hietjan, D.F. (to appear). *Testing and adjusting for overdispersion in generalized linear models*.

Therneau, T. M. and Grambsch, P. M. and Fleming, T. R. (1990). *Martingale-based residuals for survival models*. Biometrika, 77:147–160.

Wei, L.J., Lin, D.Y., and Weissfeld, L (1989). *Regression analysis of multivariate incomplete failure time data by modeling marginal distributions*.

White, H. (1980). *A heteroskedasticity-consistent covariance matrix estimator and a direct test for heteroskedasticity.* Econometrica, 48:817–830.

White, H. (1982). *Maximum likelihood estimation of misspecified models*. Econometrika, 50:1–25.

PARAMETRIC REGRESSION IN SURVIVAL MODELS

25

Parametric methods are used where extrapolation of results is necessary to predict failure rates.

PARAMETRIC REGRESSION IN SURVIVAL MODELS

In contrast to the non-parametric (and semi-parametric) survival curve estimates of Kaplan-Meier, Fleming-Harrington, and Cox, among others, this chapter presents a parametric formulation to the estimation problem. The development and use of parametric survival models actually predates that of the non-parametric methods. Although, non-parametric methods now dominate in fields of study where the primary concern is to assess the relative risk of failing (for example, dying) for two subjects that have different covariables (for example, treatment regimens), parametric methods are still vitally important in situations where extrapolation of results is necessary to predict failure rates under different conditions than those in the original study. A typical question addressed by non-parametric methodology is "How much does the risk of dying decrease if a *new* treatment is given to a lung cancer patient." A typical question addressed by the parametric methodology in an accelerated testing setting is "What proportion of heaters will fail when run at $1100°F$ for 2 years" even though the original study ran heaters at temperatures ranging from $1520°F$ to $1710°F$ for only four months.

In a manufacturing setting, studies of failure rates for new products cannot typically be done under normal operating conditions because they take too long to complete. Consequently, accelerated tests are conducted, exposing the product to more severe stresses than normal so that failures occur and then extrapolation is used to estimate failure rates under normal operating conditions. If the data are reasonably well modeled by one of the parametric distributions, parametric models provide information for assessing properties of the baseline hazard function which the non-parametric models don't.

The parametric survival distributions implemented in S-PLUS are

- Normal (Gaussian) and log-normal
- Smallest extreme value and Weibull
- Logistic and log-logistic
- Exponential
- Rayleigh

25.1 IRLS FORMULATION

With some care, parametric survival can be formulated as an iteratively reweighted least squares (IRLS) problem used in Generalized Linear Models

(GLM) of McCullagh and Nelder (1989). A detailed description of this setup for general maximum likelihood computation is found in Green (1984). This is the approach used for fitting parametric survival curves in S-PLUS.

Let y be the response vector, and x_i be the vector of covariates for the ith observation. Assume that

$$z_i \equiv \frac{t(y_i) - x'_i\beta}{\sigma} \sim f \tag{25.1}$$

for some distribution f, where y may be censored and t is a differentiable transformation function.

Then the likelihood for $t(y)$ is

$$l = \left(\prod_{\text{exact}} f(z_i)/\sigma\right)\left(\prod_{\text{right}} \int_{z_i}^{\infty} f(u)du\right)\left(\prod_{\text{left}} \int_{-\infty}^{z_i} f(u)du\right)\left(\prod_{\text{interval}} \int_{z_i^*}^{z_i} f(u)du\right),$$

where *exact*, *right*, *left*, and *interval* refer to uncensored, right censored, left censored, and interval censored observations, respectively, and z_i^* is the lower endpoint of a censoring interval. Then the log likelihood is defined as

$$\log(l) = \sum_{\text{exact}} g_1(z_i) - \log(\sigma) + \sum_{\text{right}} g_2(z_i) + \sum_{\text{left}} g_3(z_i) + \sum_{\text{interval}} g_4(z_i, z_i^*). \tag{25.2}$$

Derivatives of the log likelihood with respect to the regression parameters are

$$\frac{\partial \log(l)}{\partial \beta_j} = \sum_{i=1}^{n} x_{ij} \frac{\partial g}{\partial n_i} \tag{25.3}$$

$$\frac{\partial^2 \log(l)}{\partial \beta_j \beta_k} = \sum x_{ij} x_{ik} \frac{\partial^2 g}{\partial \eta_i^2}, \tag{25.4}$$

where $\eta = X'\beta$ is the vector of linear predictors.

Thus if we treat σ as fixed, then iteration is equivalent to IRLS with weights of $-g''$ and adjusted dependent variable of $\eta - g'/g''$. The Newton-

Raphson step defines an update δ by

$$(X^T DX)\delta = X^T U, \tag{25.5}$$

where D is the diagonal matrix formed from $-g''$, and U is the vector g'. The current estimate β satisfies $X\beta = \eta$, so that the new estimate $\beta + \delta$ will have

$$(X^T DX)(\beta + \delta) = X^T D_\eta + X^T U = (X^T D)(\eta + D^{-1}U). \tag{25.6}$$

This implementation allows the returned fit object to inherit from class `"glm"`. Consequently, stepwise methods are inherited allowing one step update approximations effectively under the assumption that the extra parameters are fixed. This is a useful and quick first approximation for new fits.

There are several differences between this parametric survival formulation and generalized linear models (GLM).

1. A GLM assumes that *y* comes from a particular distribution and a transformation (called the *link* function) of the mean of *y* ($\mu = \mathrm{E}(y)$) is linearly related to the predictors. That is, if *t* is the link function then $t(\mu) = \eta$. For parametric *survival* models, we assume that the transformed data, *t(y)*, follows the given distribution. If the data are uncensored, the fit of a gaussian model with log link will *not* be the same as a GLM fit with log link. In this case, the survival fitting function assumes that the error distribution is log-normal.
2. The starting estimate procedure for a GM has been modified. The regression coefficients are not independent of σ. The procedure starts with a naive estimate of σ, the variance of *y*, ignoring censoring. With sigma fixed, the parameters for the other variables can be estimated by doing one iteration with $\eta = t(y)$. For interval censoring, the midpoint is used. Then σ is reestimated as the variance of the unweighted residuals and iteration continues.
3. The maximized value of the log likelihood for a right or left censored observation is 0, since by making η sufficiently large or small the relevant integral is 1. Thus, there is no “deviance correction". There is, however, for interval censored values.

Derivatives of the Log Likelihood

This section is very similar to the appendix of Escobar and Meeker (1992). Let f and F denote the density and cumulative distribution functions, respectively, of one of the parametric survival distributions. Using equation (25.2) for defining $g_1,\ldots,g_4$, we have

$$\frac{\partial g_1}{\partial \eta} = -\frac{1}{\sigma}\left[\frac{f'(z)}{f(z)}\right]$$

$$\frac{\partial g_4}{\partial \eta} = -\frac{1}{\sigma}\left[\frac{f(z^u)-f(z^l)}{F(z^u)-F(z^l)}\right]$$

$$\frac{\partial^2 g_1}{\partial \eta^2} = \frac{1}{\sigma^2}\left[\frac{f''(z)}{f(z)}\right] - (\partial g_1/\partial \eta)^2$$

$$\frac{\partial^2 g_4}{\partial \eta^2} = \frac{1}{\sigma^2}\left[\frac{f'(z^u)-f'(z^l)}{F(z^u)-F(z^l)}\right] - (\partial g_4/\partial \eta)^2$$

$$\frac{\partial g_1}{\partial \log\sigma} = -\left[\frac{zf'(z)}{f(z)}\right]$$

$$\frac{\partial g_4}{\partial \log\sigma} = -\left[\frac{z^u f(z^u)-z^l f(z^l)}{F(z^u)-F(z^l)}\right]$$

$$\frac{\partial^2 g_1}{\partial (\log\sigma)^2} = \left[\frac{z^2 f''(z)+zf'(z)}{f(z)}\right] - (\partial g_1/\partial \log\sigma)^2$$

$$\frac{\partial^2 g_4}{\partial (\log\sigma)^2} = \left[\frac{(z^u)^2 f'(z^u)-(z^l)^2 f'(z^l)+[z^u f(z^u)-z^l f(z^l)]}{F(z^u)-F(z^l)}\right] - (\partial g_1/\partial \log\sigma)^2$$

$$\frac{\partial^2 g_1}{\partial \eta \partial \log\sigma} = \frac{zf''(z)}{\sigma f(z)} - \partial g_1/\partial \eta(1+\partial g_1/\partial \log\sigma)$$

$$\frac{\partial^2 g_4}{\partial \eta \partial \log\sigma} = \frac{z^u f'(z^u)-z^l f'(z^l)}{\sigma[F(z^u)-F(z^l)]} - \partial g_4/\partial \eta(1+\partial g_4/\partial \log\sigma)$$

To obtain the derivatives for g_2, set the upper endpoint z_u to ∞ in the equations for g_4. To obtain the equations for g_3, left censored data, set the lower endpoint to $-\infty$. See section 25.3, Computations, for comments on parameterization with respect to $\log\sigma$ instead of σ.

25.2 DISTRIBUTIONS

The presentation of the distributions contained in this section are similar to that in Nelson (1982). Derivatives of the terms in the log likelihood, equation (25.2), are presented following the details for each distribution.

For each distribution the *standardized variable*, *z*, is defined by equation (25.1) where $\eta = x_i'\beta$ is the linear predictor and σ is the scale parameter. The details for each distribution are written in terms of the standardized variable, *z*.

Gaussian

This is, perhaps, the most frequently used distribution in applied statistics. It is more commonly known as the *normal* distribution. The standardized variable, *z*, defined by equation (25.1) has mean 0 (zero) and variance 1 (one). The standard normal distribution is then defined by

$$F(z) = \int_{-\infty}^{z} f(t)dt = \Phi(z)$$

$$f(z) = \frac{e^{-z^2/2}}{\sqrt{2\pi}}$$

$$f'(z) = -zf(z)$$

$$f''(z) = (z^2 - 1)f(z)$$

The derivatives of the terms in the log likelihood, equation (25.2), are given by

$$g_1 = -z^2/2 - \log(\sqrt{2\pi}) \qquad g_2 = \log(1 - \Phi(z))$$

$$g'_1 = -z \qquad g'_2 = -f(z)/(1 - \Phi(z))$$

$$g''_1 = -1 \qquad g''_2 = -f'(z)/(1 - \Phi(z)) - (g'_2)^2$$

For uncensored data, the "standard" GLM results are obtained by substituting g_1 into equations (25.2)-(25.6). The first derivative vector is equal to $X'r$ where $r = -z/\sigma$ is a scaled residual, the update step $D^{-1}U$ is independent of the estimate of σ, and the maximum likelihood estimate of $n\sigma$ is the sum of squared residuals. None of these hold so neatly for right censored data.

Smallest Extreme Value

If *y* has the smallest extreme value distribution, then e^y has a Weibull distribution and e^{-y} has a Gompertz distribution. To fit a Weibull model let *t(y) = log(y)*. If the scale is constrained to be 1 (one) and a log transform is applied to *y*, an exponential model is fit. A log transform of *y* with the scale

constrained to be 0.5 yields a Rayleigh distribution. If y is reflected about zero, that is, $t(y) = -log(y)$ and right and left censoring indicators are exchanged, the resulting fit is equivalent to fitting a Gompertz distribution. The signs of the returned coefficients and residuals will be reversed, however. The standardized variable, z, defined by equation (25.1) has mean 0.5722 and variance $\pi^2/6$. Let $w = e^z$, then the standard smallest extreme value distribution is defined as

$$\begin{aligned} F(z) &= 1 - e^{-w} \\ f(z) &= we^{-w} \\ f'(z) &= (1-w)f(z) \\ f''(z) &= (w^2 - 3w + 1)f(z) \end{aligned}$$

The derivatives for the terms in the log likelihood, equation (25.2), are given by:

$$g_1 = z - w \qquad g_2 = -w \qquad g_3 = \log(1 - e^{-w})$$

$$g'_1 = w - 1 \qquad g'_2 = w \qquad g'_3 = \frac{-we^{-w}}{1 - e^{-w}}$$

$$g''_1 = -w \qquad g''_2 = -w \qquad g''_3 = \frac{we^{-w}(1-w)}{1 - e^{-w}} - (g'_3)^2$$

The mode of the distribution is at $z = 0$ with $f(0) = 1/e$. For an exact observation the deviance term has $\hat{y} = y$. For interval censored data where the interval is of length $b = z^u - z^l$, most mass is covered if the interval has a lower endpoint of $a = log(b/(e^b - 1))$, so that the contribution to the deviance is

$$\log(e^{-e^a} - e^{-e^{a+b}}).$$

In the literature, the cumulative distribution function for the Gompertz is sometimes written as $F(z) = e^{-\rho e^{-z}}$. Rewriting this with $\rho = e^{\sigma \log(\rho)/\sigma}$, however, we see that ρ can be absorbed into the intercept term.

Logistic

This distribution is very close to the Gaussian except in the extreme tails, but it is far easier work with. All the computations are closed form. However, very small data values combined with a log transformation lead to extreme values, in which case the results will differ. (In such cases the rationality of a Gaussian fit may also be in question). The standardized variable, z, defined

by equation (25.1) has mean 0 (zero) and variance $\pi^2/3$. Let $w = e^z$, then the standard logistic distribution is defined by

$$\begin{aligned} F(z) &= w/(1+w) \\ f(z) &= w/(1+w)^2 \\ f'(z) &= f(z)(1-w)/(1+w) \\ f''(z) &= f(z)(w^2 - 4w + 1)/(1+w)^2 \end{aligned}$$

The derivatives for the terms in the log likelihood, equation (25.2), are given by:

$$g_1 = z - 2\log(1+w) \qquad g_2 = -\log(1+w) \qquad g_3 = z - \log(1+w)$$

$$g'_1 = \frac{w-1}{w+1} \qquad g'_2 = \frac{w}{1+w} \qquad g'_3 = \frac{-1}{1+w}$$

$$g''_1 = -\frac{2w}{(1+w)^2} \qquad g''_2 = -\frac{w}{(1+w)^2} \qquad g''_3 = -\frac{w}{(1+w)^2}$$

The distribution is symmetric about 0, so for an exact observation the contribution to the deviance term is *-log(4)*. For an interval censored observation with span *2b* the contribution is

$$\log(F(b) - F(-b)) = \log\left(\frac{e^b - 1}{e^b + 1}\right)$$

Other Distributions

This general approach would seem to apply to any distribution with a support over the entire line. The connection to GLM, however, requires that the weights, $-g''$, be positive, that is, that the distribution is log-concave. This is not true for the Cauchy distribution, for instance, or any other whose tails are heavier than the double exponential.

25.3 COMPUTATIONS

Perhaps surprisingly, these likelihoods do not always behave well. The iteration seems to be particularly sensitive to the scale parameter. Our starting estimate may not be so good, since it does not account for the amount of censoring. For small data sets it is easy to find starting estimates that lead to divergence of the Newton-Raphson method.

Because of this, the internal routine used for estimating the parameters has been updated to a ridge-stabilized weighted likelihood. The basic operations of this are

- A Choleski decomposition is used on the information matrix I, which returns an indicator of whether the matrix is positive definite. The matrix is always symmetric by construction. Denote symmetric positive definite by SPD.
- If the matrix is not SPD, then the step $(I + \tau D)\delta = U$ is taken instead of the the usual step defined by $I\delta = U$, where D is the diagonal of I. That is, the diagonal of I is multiplied by $1 + \tau$. For large enough τ, the matrix will become diagonally dominant and therefore SPD. The value of τ is increased through the series 1,2,4, ... until $I + \tau D$ is SPD.
- The ridge parameter τ is reduced if possible by a factor of 4 in each iteration. It never returns to 0 (zero), however, once ridge stabilization has been invoked.
- If an iteration leads to a worsening of the log likelihood, then step halving is employed.

Though quite stable, it can take a surprising number of steps for the program to leave a bad region of the parameter space. If I is far from SPD, the ridge correction can cause the iteration steps to be very small. It may easily take 8-10 iterations to get “back on track”. Normally the GLM type of parameter initialization is very good, a ridge estimate is not required, and convergence occurs in 3-5 iterations.

One side effect of this procedure is that the iteration is immune to any singularities in the model matrix. If the model matrix is singular then its coefficients may be indeterminate, but the values of η are still well defined. The procedure finishes by doing one iteration of the IRLS algorithm by calling the `lm.wfit` function. Nice side effects of this are the handling of the `singular.ok` option and creation of most of the data items needed to inherit from class `"glm"`.

Parameterization is in terms of $\log(\sigma)$. This avoids the boundary condition at 0 (zero), and decreases the number of iterations considerably for some cases. This implementation is made by applying the chain rule:

$$\frac{\partial \log(l)}{\partial \log\sigma} = \sigma\frac{\partial \log(l)}{\partial \sigma}$$

$$\frac{\partial^2 \log(l)}{\partial (\log\sigma)^2} = \sigma^2\frac{\partial^2 \log(l)}{\partial \sigma^2} + \frac{\partial \log(l)}{\partial \log\sigma}$$

$$\frac{\partial^2 \log(l)}{\partial \eta \partial \log\sigma} = \sigma\frac{\partial^2}{\partial \eta \partial \sigma}$$

Though deviance is reported, minimization is based on the log-likelihood. In these models the scale parameter is a part of our iteration set, unlike GLM where scale is estimated after the end of iteration. Like GLM models, however, the deviance is calculated as though the scale parameter were fixed in advance.

25.4 RESIDUALS

The `residuals` function returns a matrix containing the following columns

- The component of the deviance residual
- $\partial L_i / \partial \eta$
- $\partial^2 L_i / \partial \eta^2$
- $\partial L_i / \partial \log\sigma$
- $\partial^2 L_i / \partial (\log\sigma)^2$
- $\partial^2 L_i / \partial \eta \partial \log\sigma$

These data can be used for various influence diagnostics, in the fashion of Escobar and Meeker (1992). (Note that their μ is our η.) "User friendly" functions that make use of these for plots have yet to be written.

25.5 EXAMPLE

Parametric survival curves are estimated using the `survreg` function. The `capacitor` data frame contains data from a simulated life testing of capacitors from Meeker and Duke (1982). The `capacitor` data frame is close enough to the data modeled in Nelson (1990), page 302, that it works as a verification data set. The variables in `capacitor` are:

- `days`time to failure
- `event`indicator of failure (1) or censoring (0)
- `voltage`voltage at which the test was run

A summary of this data frame follows:

```
> summary(capacitor)
      days            event          voltage
 Min.   :  0.68   Min.   :0.000   Min.   :20.00
 1st Qu.: 73.87   1st Qu.:0.000   1st Qu.:26.00
 Median :300.00   Median :0.000   Median :26.00
 Mean   :205.20   Mean   :0.432   Mean   :26.72
 3rd Qu.:300.00   3rd Qu.:1.000   3rd Qu.:29.00
 Max.   :300.00   Max.   :1.000   Max.   :32.00
```

You fit a Weibull model to the `capacitor` data as follows:

```
> capac.fit1 <- survreg(Surv(days, event) ~ voltage,
+ data = capacitor)
```

You don't have to specify the distribution or the link function in this case because `survreg` defaults to `link = "log"` and `dist = "extreme"`, the smallest extreme value distribution, together which correspond to the Weibull distribution. Printing the resulting fit produces the following display:

```
> capac.fit1
> survreg(Surv(days, event) ~ voltage, data = capacitor)
Call:
survreg(formula = Surv(days, event) ~ voltage, data =
capacitor)

Coefficients:
 (Intercept)     voltage Log(scale)
    24.14074  -0.6403556   0.1855962

Dispersion (scale) = 1.203936
Degrees of Freedom: 125 Total; 122 Residual
Residual Deviance: 116.3684
```

The summary of the fit object looks almost identical to the summary of a `glm` fit object:

```
> summary(capac.fit1)
Call:
survreg(formula = Surv(days, event) ~ voltage, data =
capacitor)
```

```
Deviance Residuals:
   Min     1Q Median    3Q  Max
 -2.49 -0.762  0.136 0.673 1.57

Coefficients:
             Value Std. Error z value        p
(Intercept)  24.141     2.4495    9.86 6.49e-23
    voltage  -0.640     0.0811   -7.89 2.93e-15
 Log(scale)   0.186     0.1113    1.67 9.54e-02

Extreme value distribution: Dispersion (scale) = 1.203936

    Null Deviance: 241 on 124 degrees of freedom
Residual Deviance: 116 on 122 degrees of freedom (LL= -122)
Number of Newton-Raphson Iterations: 6

Correlation of Coefficients:
           (Intercept) voltage
   voltage -0.998
Log(scale)  0.561      -0.559
```

Voltage is clearly quite significant in the model. Let's examine a plot of the deviance residuals to see how well the model fits. The deviance residuals versus the logged fitted values are displayed in figure 25.1.

```
> plot(log(fitted(capac.fit1)), resid(capac.fit1))
> title("Deviance Residuals vs Log Fitted Values")
```

You can now compute percentiles from the model. The percentile formulas are taken from Nelson (1990), page 64. Noting that for a Weibull distribution the *p*th percentile is given by

$$T_p = \alpha[-\log_e(1-p)]^{1/\beta}$$

and that

$$\log_e(\alpha) = \alpha_0 + \alpha_1 \text{voltage}$$

$$\beta = 1/(\text{scale parameter from the fit})$$

A function for estimating the percentiles follows:

```
> "weib.percentile" <-
function(p, stress, coefs, scale)
{
        up <- - log(1 - p)
        alpha <- exp(sum(coefs * c(1, stress)))
```

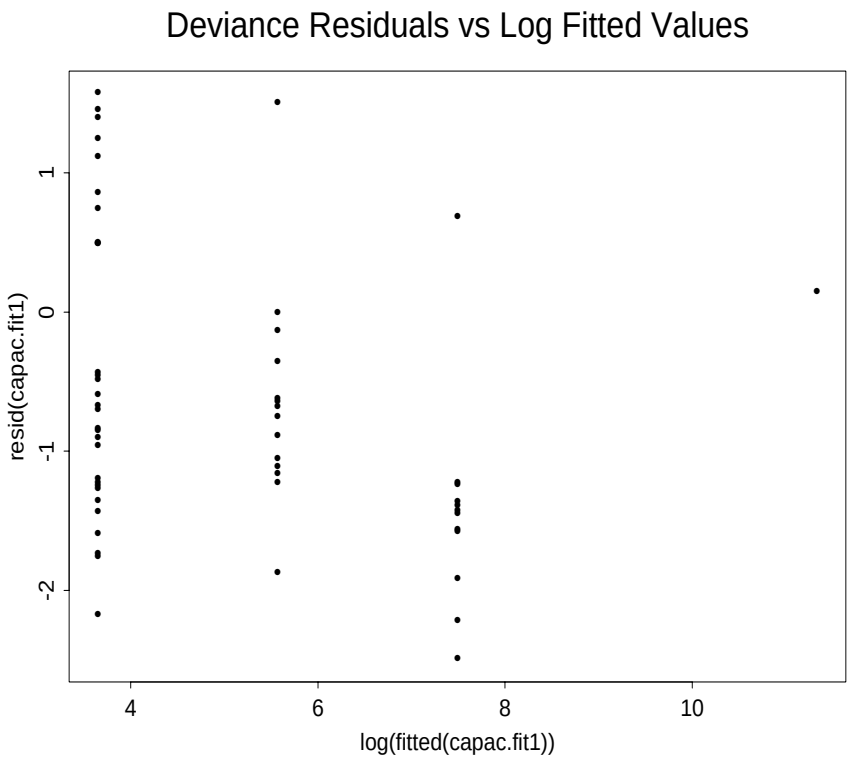

Figure 25.1: *Deviance residuals versus fitted values for a model of capacitor failure times versus voltage.*

```
        percentile <- alpha * (up^scale)
        names(percentile) <- paste(100 * p, "%", sep = "")
        percentile
}
```

Using the model information we now estimate the time in years for for various percentages of units to fail when operating at 20 volts.

```
> weib.percentile(c(.01,.05,.1), 20, coef(capac.fit1),
+ 1.204)/365.25
        1%       5%      10%
 0.8998401 6.404023 15.23506
```

Thus, it takes about 6.4 years for 5% of the units to fail when operating at 20 volts.

The example in Nelson (1990), page 302, displays a Weibull model with the

logged scale parameter, $\log_e(\alpha)$, modeled as a linear function of $\log_e(\text{voltage})$. We fit and display a partial summary of this second model as follows:

```
> capac.fit2 <- survreg(Surv(days, event) ~ log(voltage),
+ data = capacitor)
> summary(capac.fit2)
Call:
survreg(formula = Surv(days, event) ~ log(voltage), data =
capacitor)
Deviance Residuals:
   Min     1Q Median    3Q   Max
 -2.5 -0.718 0.0894 0.668 1.54

Coefficients:
               Value Std. Error z value        p
 (Intercept)  67.945      8.151    8.34 7.69e-17
log(voltage) -18.546      2.395   -7.74 9.78e-15
  Log(scale)   0.191      0.111    1.71 8.67e-02

Extreme value distribution: Dispersion (scale) = 1.210292

    Null Deviance: 241 on 124 degrees of freedom
Residual Deviance: 116 on 122 degrees of freedom (LL= -12 )
. . .
```

Now applying the `weib.percentile` function to `capac.fit2` we obtain percentiles similar to those (56.0794, 394.2983, 915.1363) on page 302 of Nelson (1990).

```
> weib.percentile(c(.001, .005, .01), log(20),
+ coef(capac.fit2), 1.2103)
   0.1%     0.5%       1%
 56.075 394.2635 915.0514
```

Note that the `survreg` function does not take a `weights` argument like most of the other model fitting functions. Consequently, if you have a variable of case weights, you have to expand the data frame by replicating the cases according to their weights. This is what we did to create the `capacitor` data frame. The original data frame had 25 capacitors censored at 300 days for voltage 20, 39 capacitors censored at 300 days for voltage 26, and 7 capacitors censored at 300 days for voltage 29. Originally, there was a `case.weights` variable so that only three rows in the `capacitor` data frame represented these censored observations. They looked like the

following:

```
  days censor case.weights voltage
1  300      1           25      20
2  300      1           39      26
3  300      1            7      29
```

We expanded these rows in the following way:

```
> capacitor <- capacitor[rep(1:dim(capacitor)[1],
+ capacitor$case.weights), ]
```

The indicies to the rows of the data frame are replicated according to `case.weights` and the subscripting repeats the appropriate rows.

Warning

You can apply other transformation functions (for example, `^2`, `sqrt`), to the predictor variable(s) as you can for other model fitting functions in S-PLUS. However, there is no safe prediction function like `predict.gam` is for linear models. Consequently, you need to be wary of using functions like `poly`, `bs` and `ns` which produce parameter estimates dependent on the original data. For models created using these transformations, functions like `weib.percentile` will not produce correct computations because simply multiplying the estimated coefficient(s) times the new data values is not valid for these complex transformations. See the help file for `predict.gam` for a discussion of safe prediction.

25.6 REFERENCES

Escobar, L.A. and Meeker, Jr, W.Q. (1982). *Assessing influence in regression analysis with censored data*. Biometrics, 48:507–528.

Green, P.J. (1984). *Iteratively reweighted least squares for maximum likelihood estimation, and some robust and resistant alternatives (with discussion)*. Journal of the Royal Statistical Society, Series B, 46:149–192.

Meeker, Jr., W. Q. and Duke, S.D. (1982). *User's Manual for CENSOR—A User-Oriented Computed Program for Life Data Analysis*. Statistical Laboratory, Iowa State University, Ames, IA.

Nelson, W. (1982). *Applied Life Data Analysis*. Wiley, New York.

Nelson, W. (1990). *Accelerated Testing*. Wiley, New York.

EXPECTED SURVIVAL 26

Expected survival curves are typically used for comparison purposes with other survival studies.

EXPECTED SURVIVAL

This chapter describes several methods for estimating expected survival curves. Typically expected curves are used for comparison with another study. Sometimes the results of an earlier study are compared with a later one to assess, for example, improvement in treatment. Expected survival curves can be computed from tables of hazards rates or from a previously computed Cox model.

The methodology described in this chapter includes the computation of *individual* and *cohort* expected survival curves. Individual expected curves are typically used to compute tests to compare the observed survival with that expected (for example, the one-sample log-rank test) for a matched (for example, on age, sex, and year of entry) control population. Cohort expected curves are useful for graphical comparisons, sample size computations, and forecasting.

Three methods are available for computing cohort expected survival curves: the Ederer or "exact" method, Hakulinen's method, and the conditional estimate. In the Cox model literature, these have been called the "direct-adjusted," "Bonsel," and "expected survival" curves. Each method generates a matched control for each subject in the study and then computes the expected survival for the matched controls. The difference between the methods lies in the assumptions made when computing the expected survival. The basic assumptions of each and a brief description of its utility follows:

`Ederer`	Assumes *complete follow-up*, that is, no censoring. Each control is followed until death. This is most appropriate when doing forecasting, sample size calculations or other predictions of the "future" where censoring is not an issue.
`Hakulinen`	Assumes *maximal potential follow-up*. Each control is followed until death or censoring of its matched case. Useful for graphical comparison with the study population.
`conditional`	Has the same assumptions and is *asymptotically equivalent to Hakulinen's method*.

The implementation of expected survival curve estimation allows inputing you own table of hazard rates or computing expected survival based on a previous Cox model. Additionally, the notion of *person years* of follow-up time is discussed as an example.

26.1 INDIVIDUAL EXPECTED SURVIVAL

Let $\lambda_i(t)$ and $\Lambda_i(t)$ be the derived hazard and cumulative hazard functions, respectively, for subject *i*, starting at their time of entry to the study. Then $S_i(t) = \exp(-\Lambda_i(t))$ is the subject's expected survival function.

Some authors use the product form $S = 1 - \Pi(1 - q_k)$ where the q are yearly probabilities of death, and yet others an equation similar to actuarial survival estimates. Numerically it makes little difference which form is chosen, and the *S* functions use the hazard based formulation for its convenience.

The survival tables published by the Department of the Census contain one year survival probabilities by age and sex, optionally subgrouped as well by race and geographic region. The entry for age 21 in 1950 is the probability that a subject who turns 21 during 1950 will live to his or her 22nd birthday. The tables stored in S contain the daily hazard rate λ rather than the probability of survival *p*

$$p = \exp(-365.25 \times \lambda)$$

for convenience. If *a*, *s*, and *y* are subscripts into the age by sex by calendar year table of rates, then the cumulative hazard for a given subject is simply the sequential sum of $\lambda_{asy} \times$ number of days in state(a, s, y). That is, the patient progresses through the rate table on a diagonal line whose starting point is (date of entry, age at entry, sex), see Berry (1983) for a nice graphical illustration.

26.2 COHORT EXPECTED SURVIVAL

The expected survival curve for a cohort of *n* subjects is an "average" of the *n* individual survival curves for the subjects. There are 3 main methods for combining these; for some data sets they can give substantially different results. Let S_e be the expected survival for the cohort as a whole, and S_i, λ_i be the individual survival and hazard functions. All three methods can be written as

$$S_e(t) = \exp\left(-\int_0^t \frac{\sum \lambda_i(s) w_i(s)}{\sum w_i(s)} ds\right) \tag{26.1}$$

and differ only in the weight function w_i.

The cohort curve should be distinguished from the individual curve for an average subject. For example, assume we had a cohort of grandfathers and their grandsons, the grandfathers average 70 years and the grandsons average 10 year of age. The cohort curve, which is an estimate of the curve we would expect from long term follow-up of these subjects, is considerably different than the curve for the "average" subject with mean age of 40 years.

The Exact Method

A weight function of $w_i(t) = S_i(t)$ corresponds to the *exact* method. This is the oldest and most commonly used technique, and is described in Ederer, Axtel and Cutler (1961). An equivalent expression for the estimate is

$$S_e(t) = (1/n)\sum S_i(t) \tag{26.2}$$

The exact method corresponds to selecting a population matched control for each subject in the study, and then computing the expected survival of this cohort *assuming complete follow-up*. The exact method is most appropriate when doing forecasting, sample size calculations or other predictions of the "future" where censoring is not an issue.

A common use of the expected survival curve is to plot it along with the Kaplan-Meier estimate of the sample in order to assess the relative survival of the study group. When used in this way, several authors have shown that the exact method can be misleading if censoring is not independent of age and sex (or whatever the matching factors are for the referent population). Indeed, independence is often not the case. For example, in a long study it is not uncommon to allow older patients to enroll only after the initial phase. A severe example of this is demonstrated in Verheul *et al.* (1993), concerning aortic valve replacement over a 20 year period. The proportion of patients over 70 years of age was 1% in the first ten years, and 27% in the second ten years. Assume that analysis of the data took place immediately at the end of the study period. Then the Kaplan-Meier curve for the later years of follow-up time will be too flat, since it is computed only over the early enrollees, who are *younger* on the average. The Ederer or exact curve will not reflect this bias, and makes the treatment look better than it is. The exact expected survival curve forms a reference line, in reality, for what the Kaplan-Meier will be when follow-up is complete, rather than for what the Kaplan-Meier is now.

Hakulinen's Method

In Hakulinen's method (1982, 1985), each study subject is again paired with a fictional referent from the cohort population, but this referent is now treated as though he/she were followed in the same way as the study patients. Each referent thus has a maximum *potential* follow-up; that is, they will

become censored at the analysis date. Let $c_i(t)$ be a censoring indicator which is 1 during the period of potential follow-up and 0 thereafter; the weight function for the Hakulinen or *cohort* method is $w_i(t) = S_i(t)c_i(t)$.

If the study subject is censored then the referent would presumably be censored at the same time, but if the study subject dies the censoring time for his/her matched referent will be the time at which the study subject *would have been censored*. For observational studies or clinical trials where censoring is induced by the analysis date this should be straightforward, but determination of the potential follow-up could be a problem if there are large numbers lost to follow-up. (However, as pointed out long ago by Berkeson, if a large number of subjects are lost to follow-up then any conclusion is subject to doubt. Did patients stop responding to follow-up letters at random, because they were cured, or because they were at death's door?)

In practice, the program will be invoked using the actual follow-up time for those patients who are censored, and the *maximum* potential follow-up for those who have died. By the maximum potential follow-up we mean the difference between enrollment date and the average last contact date; for example, if patients are contacted every 3 months on average and the study was closed six months ago this date would be 7.5 months ago. It may be true that the (hypothetical) matched control for a case who died 30 years ago would have little actual chance of such long follow-up, but this is not really important. Almost all of the numerical difference between the Ederer and Hakulinen estimates results from censoring those patients who most recently entered the study. For these recent patients, presumably, enough is known about the operation of the study to give a rational estimate of potential follow-up.

The Hakulinen formula can be expressed in a product form

$$S_e(t+s) = S_e(t) \times \frac{\sum p_i(t,s) S_i(t) c_i(t)}{\sum S_i(t) c_i(t)}, \qquad (26.3)$$

where $p_i(t,s)$ is the conditional probability of surviving from time t to time $t+s$, which is $\exp(\Lambda_i(t) - \Lambda_i(t+s))$. The formula is technically correct only over time intervals $(t, t+s)$ for which c_i is constant for all i; that is, censoring only at the ends of the interval.

The Conditional Method

The conditional estimate is advocated by Verheul (1993), and was also suggested as a computation simplification of the exact method by Ederer and Heise (1977). For this estimate the weight function $w_i(t)$ is defined to be 1 while the subject is alive and at risk and 0 otherwise. It is clearly related to Hakulinen's method, since $E(w_i(t)) = S_i(t)c_i(t)$. Most authors present the estimator in the product-limit form $\Pi[1 - d(t)/n(t)]$, where d and n are the numerator and denominator terms within the integral of equation (26.1). One disadvantage of the product-limit form is that the value of the estimate at time t depends on the number of intervals into which the time axis has been divided, for this reason we use the integral form (equation (26.1)) directly.

One advantage of the conditional estimate, shared with Hakulinen's method, is that it remains consistent when the censoring pattern differs between age-sex strata. A problem with the conditional estimator is that it has a much larger variance than either the exact or Hakulinen estimate. In fact, the variance of these latter two can usually be assumed to be zero, at least in comparison to the variance of the Kaplan-Meier of the sample. Rate tables are normally based on a very large sample size so the individual λ_i are very precise, and the censoring indicators c_i are based on the study design rather than on patient outcomes. The conditional estimate $S_c(t)$, however, depends on the actual death times and wi is a random variable.

The main use of the conditional estimate is when making conditional statements about survival. For example, in studies of surgical intervention such as hip replacement, the observed and expected survival curves often initially diverge due to surgical mortality, and then appear to become parallel. It is tempting to say that survival beyond hospital discharge is *equivalent to expected*. This is a conditional probability statement, and it should not be made unless a conditional estimate is used.

A hypothetical example may make this clearer. For simplicity assume no censoring. Suppose we have studies of two diseases, and that their age distributions at entry are identical. Disease A kills 10% of the subjects in the first month, independent of age or sex, and thereafter has no effect. Disease B also kills 10% of its subjects in the first month, but predominately affects the old. After the first month it exerts a continuing though much smaller force of mortality, still biased toward the older ages. With proper choice of the age effect, studies A and B will have almost identical survival curves; as the patients in B are always younger, on average, than those in A. Two different questions can be asked under the guise of "expected survival":

- What is the overall effect of the disease? In this sense both A and B have the same effect, in that the 5 year survival probability for a diseased group is *x*% below that of a matched population cohort. The Hakulinen estimate would be preferred because of its lower variance. It estimates the curve we "would have gotten" if the study had included a control group.
- What is the ongoing effect of the disease? Detection of the differential effects of A and B after the first month requires the conditional estimator. We can look at the slopes of the curves to judge if they have become parallel.

The actual curve generated by the conditional estimator remains difficult to interpret, however. The difficulty lies in the fact that the control subject is removed from the calculation whenever his/her matching case dies. In general, Hakulinen's cohort estimate is probably best. If there is a question about delayed effects, as in the above example, there would be an apparent flattening of the Kaplan-Meier curves after the first month. Then one can plot a new curve using only those patients who survived at least one month.

26.3 APPROXIMATIONS

The Hakulinen cohort estimate (equation (26.3)) is "Kaplan-Meier like" in that it is a product of conditional probabilities and that the time axis is partitioned according to the observed death and censoring times. Both the exact and conditional estimators can be written in this way as well. They are unlike a KM calculation, however, in that the ingredients of each conditional estimate are the *n* distinct individual survival probabilities at that time point rather than just a count of the number at risk. For a large data set this requirement for *O(n)* temporary variables can be a problem. An approximation is to use longer intervals, and allow subjects to contribute partial information to each interval. For instance, in equation (26.3) replace the 0/1 weight $c_i(t)$ by $\int_t^{t+s} c_i(u)du/s$, which is the proportion of time that subject *i* was uncensored during the interval $(t, t+s)$. If those with fractional weights form a minority of those at risk during the interval the approximation should be reliable. (More formally, if the sum of their weights is a minority of the total sum of weights). By Jensen's inequality the approximation will always be biased upwards, but it is very small. For the Stanford heart transplant data used in the examples an exact 5 year estimate using the cohort method is 0.94728, an approximate cohort computation using only the half year intervals yields 0.94841. The exact estimate is unchanged under re-partitioning of the time axis.

26.4 TESTING

All of the above discussion has been geared towards a plot of $S_e(t) = \exp(-\Lambda_e(t))$ which attempts to capture the proportion of patients who will have died by *t*. When comparing observed to expected survival for testing purposes, an appropriate test is the one-sample log-rank test (Harrington and Fleming (1982)) $(O\text{-}E)^2/E$, where O is the observed number of deaths and

$$E = \sum_{i=1}^{n} e_i = \sum_{i=1}^{n} \int \lambda_i(s) Y_i(s) \tag{26.4}$$

is the expected number of deaths, given the observation time of each subject. This follows Mantel's concept of 'exposure to death' (Mantel (1966)), and is the expected number of deaths during this exposure. Notice how this differs from the expected number of deaths $nS_e(t)$ in the matched cohort at time *t*. In particular, E can be greater than n. Equation (26.4) is referred to as the person-years estimate of the expected number of deaths. The log-rank test is usually more powerful than one based on comparing the observed survival at time *t* to $S_e(t)$; the former is a comparison of the entire observed curve to the expected, and the latter is a test for difference at one point in time.

Tests at a particular time point, though less powerful, will be appropriate if some fixed time is of particular interest, such as 5 year survival. In this case the test should be based on the cohort estimate. The H_0 of the test is "Is survival different that what a control-group's survival would have been?" A pointwise test based on the exact estimate may well be invalid if there is censoring. A pointwise test based on the conditional estimate has two problems. The first is that an appropriate variance is difficult to construct. The second, and more serious one, is that it is unclear exactly what alternative is being tested against.

Hartz, Giefer and Hoffman (1983) argue strongly for the pointwise tests based on a expected survival estimate equivalent to equation (26.3), and claim that such a test is both more powerful and more logical than the person-years approach. Subsequent letters to the editor (Hartz, Giefer, and Hoffmann (1984, 1985)) challenged these views, and it appears that the person-years method is preferred.

Berry (1983) provides an excellent overview of the person-years method. Let the e_i be the expected number of events for each subject, treating them as an $n = 1$ Poisson process. We have

$$\begin{aligned} e_i &= \int_0^\infty Y_i(s)\lambda_i(s)ds \\ &= \Lambda_i(t_i) \end{aligned}$$

where t_i is the observed survival or censoring time for a subjects. This quantity e_i is the total amount of hazard that would have been experienced by the population-matched referent subject, over the time interval that subject i was actually under observation. If we treat e_i as though it were the follow-up time, this corrects for the background mortality by, in effect, mapping each subject onto a time scale where the baseline hazard is 1.

Tests can now be based on a Poisson model, using δ_i as the response variable (1=dead, 0=censored), and e_i as the time of observation (an `offset` of log e_i). The intercept term of the model estimates the overall difference in hazard between the study subjects and the expected population. An intercept-only model is equivalent to the one sample log-rank test. Covariates in the model estimate the effect of a predictor on *excess* mortality, whereas an ordinary Poisson or Cox model would estimate its effect on total mortality.

Andersen and Væth (1989) consider both multiplicative and additive models for excess risk. Let λ_i^* be the actual hazard function for the individual at risk and λ_i be, as before, that for his/her matched control from the population. The multiplicative hazard model is

$$\lambda_i^*(t) = \beta(t)\lambda_i(t).$$

If $\beta(t)$ were constant, then

$$\hat{\beta}_0 \equiv \frac{\sum N_i}{\sum e_i}$$

is an estimate of the *standard mortality ratio* or SMR, which is identical to `exp(intercept)` in the Poisson model used by Berry (assuming a log link). Their estimate over time is based on a modified Nelson hazard estimate

$$\hat{B}'(t) = \int_0^t \frac{\sum dN_i(s)}{\sum Y_i(s)\lambda_i(s)} ds,$$

which estimates the integral of $\beta(t)$. If the SMR is constant then a plot of $\hat{B}'(t)$ versus *t* should be a straight line through the origin.

For the additive hazard model

$$\lambda_i^*(t) = \alpha(t) + \lambda_i(t)$$

the integral *A(t)* of α is estimated as $log[S_{KM}(t)/S_c(t)]$, the difference between the Kaplan-Meier and the conditional estimator, when plotted on log scale. Under the hypothesis of a constant additive risk, a plot of $\hat{A}(t)$ versus *t* should approximate a line through the origin.

26.5 COMPUTING EXPECTED SURVIVAL CURVES

The function used to compute expected survival curves is `survexp`. Besides taking the typical arguments of a model fitting function, `survexp` also takes the following arguments:

`times`	vector of follow-up times at which the resulting survival curve is evaluated. If absent, the result will be reported for each unique value of the vector of follow-up times supplied in the formula.
`cohort`	logical value: if `FALSE`, each subject is treated as a subgroup of size 1. The default is `TRUE`.
`conditional`	logical value: if `TRUE`, the follow-up times supplied in the formula are death times and conditional expected survival is computed. If `FALSE`, the follow-up times are potential censoring times. If follow-up times are missing in the formula, this argument is ignored.
`ratetable`	table of event rates, such as `survexp.uswhite`, or a fitted Cox model.

Table 26.1 summarizes the argument settings used to compute expected survival curves by the various methods. The real-life examples of the following section show the use of the various argument settings to obtain the different estimates of expected survival.

Table 26.1: *Summary of arguments settings for invoking the various methods of estimating expected survival*

Method	conditional = F	cohort = T	Follow-up Times
Individual survival	not used	F	yes
Cohort survival			
Ederer	F	T	no
Hakulinen	F	T	yes
Conditional	T	T	yes

26.6 EXAMPLES

The examples of this section show how the methods discussed earlier in this chapter are implemented in S-PLUS. In addition to computing various expected survival curves an example of a closely related topic, person years of follow-up, is provided. The person-years example uses a function called `pyears` and the expected survival examples use the `survexp` function.

All of the examples use a data frame, `hearta`, computed from `heart` as follows:

```
> hearta <- by(heart, heart$id,
     function(x)x[x$stop == max(x$stop), ])
> hearta <- do.call("rbind",hearta)
```

Because the transplanted patients are represented by two rows in the `heart` data frame you first need to extract only those rows that correspond to death or censoring. Do this by selecting all rows for which `stop` is a maximum for each patient and then use `rbind` to put them back together into the data frame called `hearta`. Once this is done, `stop` contains only the total follow-up times for each patient. Note that this depends on each patient having a start time of 0 (zero).

Computing Expected Survival from National Hazard Rate Tables

The computation of expected survival curves requires either a table of hazard rates or a fitted Cox model to act as a hazard rate table. Several rate tables are built into S-PLUS. There are tables for the U.S. population, Minnesota, Florida, and Arizona. U.S. and state rate tables contain the expected hazard rate for a subject, stratified by age, sex, calendar year, and optionally by race.

You can add new rate tables for other areas if you wish. Created rate tables have no restrictions on the number or names of the stratification variables. See the help file for `survexp.us` for details.

Warning

When using a rate table, it is important that all time variables be in the same units as were used for the table—for the U.S. tables this is hazard/day, so time must be in days. (Year is an exception; see the examples below.) All time variables must also have the same start date.

The following example computes the *conditional* expected survival curves for the two surgery groups in the heart transplant study. A rate table array is not provided (no `ratetable` argument is supplied), so the default table, `survexp.us`, is used.

```
> expsurv <- survexp(stop ~ surgery +
                ratetable(age = (age + 48) * 365.25,
                sex = "male", year = year + 1967.75),
                data = hearta, conditional = T)
```

The formula contains follow-up times, `stop`, a grouping variable, `surgery`, which causes the output to contain 2 curves, and a special function, `ratetable`. The `ratetable` function matches the data frame's variables to the corresponding dimensions of the rate table. The order of the arguments to the `ratetable` function is not important. The necessary key words `age`, `sex`, and `year` are contained in the `"dimid"` attribute of the rate table providing the hazard rates, `survexp.us`. The `hearta` data frame does not contain a sex variable so `sex` is set, conservatively, to `"male"`. Setting values such as this must be done by providing an integer subscript or a match to one of the `"dimnames"`.

This example produces a cohort survival curve which is almost always plotted along with the observed (Kaplan-Meier) survival of the data for visual comparison. For this example, you can plot the survival curves together as follows:

```
> plot(survfit(Surv(stop, event) ~ surgery, data = hearta),
        lty = 2:3)
> lines(expsurv, lty=2:3)
> legend(750, .9, c("No Prior Surgery","Prior Surgery"),
        lty = 2:3)
```

Figure 26.1displays the resulting plot.

There are 3 different methods for calculating the cohort curve, which are discussed in more detail in section 26.2. They are the conditional method shown above, which uses the actual death or censoring time, the method of Hakulinen, which instead uses the potential follow-up time of each subject, and the uncensored population method of Ederer, Axtel and Cutler, which requires no response variable.

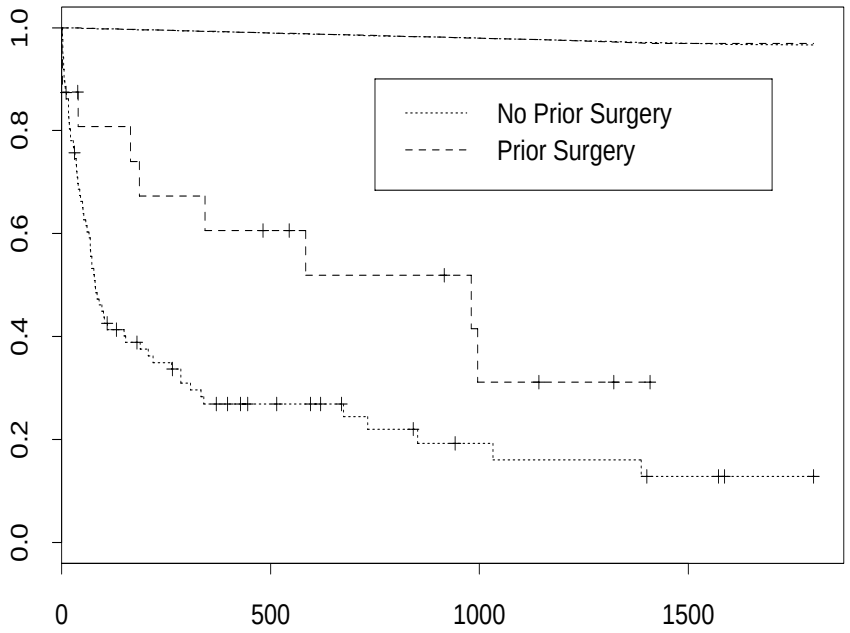

Figure 26.1: *Comparison of the heart transplant study population stratified according to prior surgery to a matched cohort from a national survival rate table.*

Individual Expected Survival Probabilities

Formal tests of observed versus expected survival are usually based not on the cohort curve directly but on the individual expected survival probabilities for each subject. These probabilities are always based on the actual death/censoring time:

```
> surv.prob <- survexp(stop ~ ratetable(age = (age + 48) *
+               365.25, sex = 'male', year = year * 365.25),
+               data = hearta, cohort = F)
> # convert from survival to hazard
> newtime <- -log(surv.prob)
> summary(glm(stop ~ offset(log(newtime)),
+             family=poisson, data = hearta))
Call: glm(formula = stop ~ offset(log(newtime)),
          family = poisson, data = hearta)
Deviance Residuals:
      Min       1Q     Median       3Q      Max
 -34.0455 -3.60184 -0.5740423 4.342719 39.94973
```

```
Coefficients:
               Value  Std. Error  t value
(Intercept) 10.77885 0.005593555 1927.013
        .
        .
        .
```

When `cohort = F`, the `survexp` function returns a vector of survival probabilities, one per subject. The negative log of the survival probability can be treated as an "adjusted time" for the subject for the purposes of modeling. The one-sample log-rank test for equivalence of the observed survival to the expected survival is the test for intercept equal to 0 (zero) in the Poisson regression model shown. A test for treatment difference, adjusted for any age-sex differences between the two arms, is obtained by adding a treatment variable to the model.

Computing Person Years

Expected survival is closely related to a standard method in epidemiology called *person years*, which consists of counting the total amount of follow-up time contributed by the subjects within any of several strata. Person-years analysis is accomplished in S-PLUS with the `pyears` function. The main complication in computing person years is that a subject may contribute to several different cells of the output array during his/her follow-up. For example, if the desired output table were treatment group by age in years, a subject with 4 years of observation would contribute to 5 different cells of the table (4 cells if she entered the study exactly on her birthdate). This example counts up years of observation for the Stanford heart patients by age group and surgical status.

Using the `hearta` data frame computed above, the person-years table is produced as follows:

```
> pyears(stop/365.25 ~ tcut(age + 48, c(0,50,60,70,100)) +
+        surgery, data = hearta, scale = 1)
$call:
pyears(formula = stop/365.25 ~ tcut(age + 48,
c(0,50,60,70,100)) + surgery, data = hearta, scale = 1)

$pyears:
                      0         1
 0+ thru  50 44.9253936 18.960986
50+ thru  60 16.7501711  6.093087
60+ thru  70  0.7556468  0.000000
70+ thru 100  0.0000000  0.000000
```

```
$n:
              0  1
 0+ thru  50 56 13
50+ thru  60 33  6
60+ thru  70  3  0
70+ thru 100  0  0

$offtable:
[1] 0
```

The `scale` argument is provided because `pyears` defaults to input times in days and output times in years (`scale = 365.25`). A 48 is added to `age` to relocate it back to its original scale. For `surgery`, a 0 (zero) corresponds to no prior surgery and a 1 (one) corresponds to prior surgery. See the help file for `heart` for more detail.

The `tcut` function has the same arguments as `cut`, but also indicates that the category is time based. If you use `cut` in the formula above, the final table would be based only on each subject's age at entry. With `tcut`, a subject who entered at age 58.5 and had 4 years of follow-up would contribute 1.5 years to the 50-60 category and 2.5 years to the 60-70 category. A consequence of this is that the `age` and `stop` variables must be in the same units for the calculation to proceed correctly. In this case both should be in years given the cutpoints that were chosen. The surgery variable is treated as a factor, exactly as it is treated by `survfit`.

The output of `pyears` is a list of arrays containing the total amount of time contributed to each cell and the number of subjects who contributed some fraction of time to each cell. The `offtable` component that is returned is the number of person years of exposure in the cohort that is not part of any cell in the `pyears` component. This is often useful as an error check. If there is a mismatch of units between two variables, nearly all the person years may be in `offtable`.

If the response variable is a `"Surv"` object, then the output also contains an array with the observed number of events for each cell. If a rate table is supplied, the output contains an array with the expected number of events in each cell. These can be used to compute observed and expected rates, along with confidence intervals.

Using a Cox Model as a Rate Table

Many times a study group will be compared to a historical control. If the comparison is to be adjusted for differences in certain covariates, it is usually based on a Cox model fit to the historical data. The methods used in this example are parallel to the previous examples using national rate tables (for

example, `survexp.us`), but in this example a prior Cox model acts as the "rate table" for `survexp`.

Individual survival curves can be obtained using `survfit`, as described in chapter 24, The Cox Proportional Hazards Model. Extending that example,

```
> s1 <- survfit(ov.fit1, newdata = data.frame(age = 35))
```

gives the expected curve for a 35 year old subject, and

```
> s2 <- survfit(ov.fit1, newdat = ovarian)
```

gives a matrix of 26 survival curves, one for each subject in the ovarian data set.

The Ederer estimate is the average of the 26 survival curves in `s2` and can be obtained as follows:

```
> s3 <- survexp(~ ratetable(age = age), data = ovarian,
ratetable = ov.fit1)
```

In the Cox model literature the Ederer estimate had been called the *direct adjusted* survival curve. Thomsen, Keiding, and Altman (1991) point out the importance of the Ederer estimate and the difference between the Ederer estimate, average survival, and the individual survival of a subject with the average `age`.

The equivalent of Hakulinen's estimate has been labeled as the *Bonsel* estimator. For studies with a short accrual, it will usually not differ from the Ederer method. Thomsen *et al.* (1991) also discuss the conditional estimator, but conclude that the final curve is "hard to interpret".

26.7 REFERENCES

Andersen, P.K. and Væth, M (1989). *Simple parametric and non-parametric models for excess and relative mortality.* Biometrics, 45:523–535.

Berry, G. (1983). *The analysis of mortality by the subject years method*. Biometrics, 39:173–184.

Ederer, F., Axtell, L.M., and Cutler, S.J. (1961). *The relative survival rate: A statistical methodology.* National Cancer Institute Mongraphs, 6:101–121.

Ederer, F. and Heise, H. (1977). *Instructions to IBM 650 programmers in processing survival computations*. Methodological Note No. 10, End Results Evaluation Section, National Cancer Institute.

Hakulinen, T. (1982). *Cancer survival corrected for heterogeneity in patient withdrawal*. Biometrics, 38:933.

Hakulinen, T. and Abeywickrama, K.H. (1985). *A computer program package for relative survival analysis*. Computer Programs in Biomedicine, 19:197–207.

Harrington, D.P. and Fleming, T.R. (1982). *A class of rank test procedures for censored survival data*. Biometrika, 69:553–566.

Hartz, A.J., Giefer, E.E., and Hoffmann, G.G. (1983). *A comparison of two methods for calculating expected mortality.* Statistics in Medicine, 2:381–386.

Hartz, A.J., Giefer, E.E., and Hoffmann, G.G. (1984). *Letter and rejoinder on "A comparison of two method for calculating expected mortality"*. Statistics in Medicine, 3:301–302.

Hartz, A.J., Giefer, E.E., and Hoffmann, G.G. (1985). *Letters and rejoinder on "A comparison of two method for calculating expected mortality"*. Statistics in Medicine, 4:105–109.

Keiding, N., Thomsen, B.L. and Altman, D.G. (1991). *A note on the calculation of expected survival, illustrated by the survival of liver transplant patients*. Statistics in Medicine, 10:733–738.

Mantel, N. (1966). *Evaluation of survival data and two new rank order statistics arising in its consideration*. Cancer Chemotherapy Reports, 50:163–166.

Verheul, H.A., Dekker, E., Bossuyt, P., Moulijn, A.C., and Dunning, A.J. (1993). *Background mortality in clinical survival studies*. Lancet, 341:872–875.

QUALITY CONTROL CHARTS 27

Quality control charts show how far some elements deviate above or below the process center.

QUALITY CONTROL CHARTS

S-PLUS provides several functions for doing quality control charts. Table 27.1 lists the type of charts available. Both Shewhart charts and cusum charts are available for each chart type, except for the R chart for which a cusum chart has not been implemented. Ryan (1989) provides a good discussion of the use and utility of both Shewhart and cusum charts

Table 27.1: *Types of quality control charts available in S-PLUS.*

Type	Statistic Charted	Chart Description
xbar	mean	means of a continuous process variable
s	standard deviation	standard deviations of a continuous variable
R	range	ranges of a continuous variable
np	count	number of nonconforming units
p	proportion	proportion of nonconforming units
c	count	number of nonconforming units
u	count	number of nonconforming units for variable unit sizes

27.1 CONTROL CHART OBJECTS

Quality control charts are produced in two steps:

1. Create a `"qcc"` object from process data known to be gathered when the process was in a state of control.
2. Create a chart of new data using the `"qcc"` object of step 1 as the reference data.

You can think of the `"qcc"` object as containing the data necessary to calibrate the control chart. It contains information on the type of chart being plotted and the process center and variability which are necessary to compute the control limits.

The `qcc` function produces an object of class `"qcc"`. Its only required arguments are `data` (grouped appropriately) and the `type` of chart. A simple example follows:

```
> set.seed(15)
> qcdata <- matrix(10 + rnorm(100), ncol = 5)
> qccobj <- qcc(qcdata, type = "xbar")
```

A print method summarizes the "qcc" object.

```
> qccobj
xbar based on qcdata

Summary of Group Statistics:
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.655  10.14 10.09   10.51 11.31

 Group Sample Size:  5
 Number of Groups:  20
 Center of Group Statistics:  10.09016
 Standard Deviation:  1.022341
```

Each row in the matrix represents a group. If you have unequal group sizes you have to put the data in a list with one component for each group.

The arguments to qcc are:

data	the control data in the form of a vector, matrix, data frame, or list.
type	a character string or function specifying group statistics to compute.
std.dev	a numeric vector or function for specifying the within-group standard deviation(s).
sizes	a numeric vector specifying the sample sizes associated with each group.
labels	a character vector of labels for each group.

You can pass functions to the type and std.dev arguments to extend the built-in capabilities of qcc. The function that is used by default to compute the group summary statistics and the center of the group summary statistics is named stats.*type*, where *type* corresponds to the value of the type argument. For example, the default summary statistics and center for an xbar chart are computed by stats.xbar. Similarly, the default function that computes the standard deviation for an xbar chart is sd.xbar. When type is given as a function, std.dev must also be given (usually as a function as well, though not necessarily).

An example of a function that computes the summary statistics and the center as medians follows:

```
> stats.med
function(data, sizes)
{
        if(is.list(data)) {
                statistics <- sapply(data, median)
                center <- median(unlist(data))
        }
        else {
                statistics <- apply(data, 1, median)
                center <- median(data)
        }
        list(statistics = statistics, center = center)
}
```

The `stats.med` function depends on `data` being given as a matrix or list. The `qcc` function insures this by coercing a vector to a matrix. You can create other functions for computing the summary statistics and center of the process by using `stats.xbar` as a template as was done in creating `stats.med`.

As example of a function that computes the standard deviation based upon the median absolute deviation (mad) is `sd.med`. The `sd.xbar` function was used as a template for `sd.med`.

```
> sd.med
function(data, sizes)
{
  if(is.list(data))
    std.dev.within <- sapply(data, mad)
  else {
    std.dev.within <- apply(data, 1, mad)
    if(dim(data)[2] == 1)
      warning("MAD computation based on group sizes of 1")
  }
  if(length(sizes) == 1)
    sizes <- rep(sizes, length = length(std.dev.within))
  sum(sizes * std.dev.within)/sum(sizes)
}
```

You can now compute a `"qcc"` object with the center estimated as the median and the standard deviation estimated from `mad` as follows:

```
> qccobj.med <- qcc(qcdata, type = "med")
> qccobj.med
```

```
med based on qcdata

Summary of Group Statistics:
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 8.782   9.599  10.06 9.989   10.52 11.16

 Group Sample Size:  5
 Number of Groups:  20
 Center of Group Statistics:  10.14026
 Standard Deviation:  0.8418576
```

If the functions are not named with the proper prefixes (`stats.` and `sd.`, respectively), you have to pass the function names to the `type` and `std.dev` arguments. For example if the two functions are named `st.med` and `sd.mad`, respectively, you would have to type:

```
> qccobj.med <- qcc(qcdata,type=st.med,std.dev=sd.mad)
```

To chart the control data and any ongoing process data, you can produce Shewhart or cusum charts with the S-PLUS functions `shewhart` or `cusum`, respectively. Typically, Shewhart charts are used for detecting large shifts in a process (two to three sigma shifts), whereas cusum charts are used to detect smaller shifts in a process (one-half to one sigma shifts).

27.2 SHEWHART CHARTS

You can produce a Shewhart chart of the data in `qcdata` which is preserved as a `"qcc"` object in `qccobj` by using the `shewhart` function. For example:

```
> shewhart(qccobj)
```

Figure 27.1 displays the resulting chart. The text at the bottom of the chart displays pertinent statistics. The `target` value is taken as the center of the group summary statistics unless given as a separate argument. The `Number beyond limits` indicates the number of points beyond the control limits, and `Number violating runs` indicates how many points violate the runs criterion which is, by default, 5 or more consecutive points on one side of the center. You can change the run length by passing an additional argument to the `shewhart` function.

```
> shewhart(qccobj, run.length = 8)
```

By default, the `shewhart` function computes the control limits based on the `center` and `std.dev` components of `qccobj`. Both of these can be overridden, however, by providing additional arguments in the call to `shewhart`. The arguments to `shewhart` are as follows:

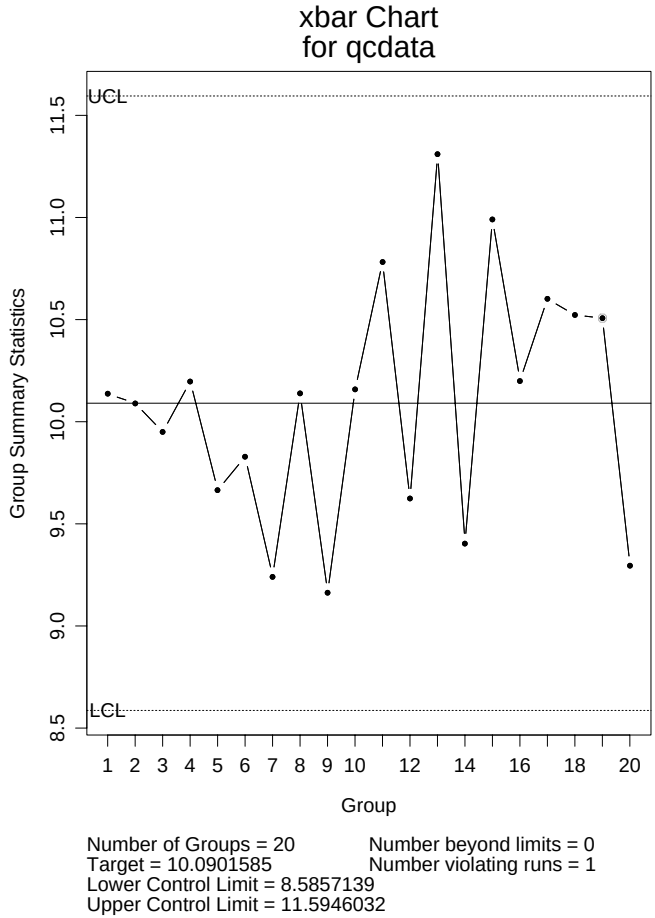

Figure 27.1: *Shewhart chart of the data in* `qccobj`.

`object` an object of class `"qcc"` which provides information on the type of group summary statistics to plot and the within-group standard deviation necessary for computing the control limits.

`newdata` vector, matrix, data frame, or list to be charted.

`type` a character string or function specifying the group summary statistics to compute.

`limits` a numeric vector or matrix or a function specifying the control limits.

`target` a numeric value specifying the center of the process if other than the `center` component of `object`.

`std.dev` a numeric value specifying the overall within-group standard deviation.

`sizes` vector of the number of observations or number of units examined in each group of `newdata`.

`labels` character vector of labels for each group in `newdata`.

`label.limits` a character vector of length two with labels for the control limits.

`confidence.level` a numeric value between 0 and 1 specifying the confidence level of the computed probability limits.

`nsigmas` a numeric value specifying one-half the width of the control limits in the number of standard errors of the group summary statistics. If given, `confidence.level` is ignored.

`add.stats` a logical value indicating whether statistics should be listed at the bottom of the chart.

`chart.all` a logical value indicating whether the `statistics` component of `object` should be plotted along with the `new.statistics` component of `object` if present and the summary statistics of `newdata` if given.

`ylim.min` a numeric vector of values to be included in the computation of the approximate y-axis limits for the control chart.

`rules` a function of rules to apply to the chart.

`highlight` a list of plotting parameters to be used for highlighting the points violating `rules`.

`...` additional arguments to `rules`.

See the `shewhart` help file for more detailed descriptions of the arguments listed above.

By default, the control limits produced by `shewhart` are *probability* limits for all the charts except the u chart. Probability limits are centered in probability about the estimate of the center of the distribution of the summary statistics or the `target` value if provided. If you want *sigma* limits, specify them through the `nsigmas` argument. In this case, the control limits

are placed at the center plus or minus `nsigmas` times the standard errors of the group summary statistics. For u charts only sigma limits are implemented. If the sample sizes vary, the standard errors will vary, and a step function will be plotted for each control limit.

The `newdata` function argument allows you to chart new data with a reference `"qcc"` object provided as the `object` argument. As an example, let's add one-half to the last six rows of `qcdata` and call it `newdata`.

```
> newdata <- qcdata
> newdata[15:20,] <- newdata[15:20,] + 1/2
```

You produce the Shewhart chart of `newdata` as follows:

```
> qccobj.shew <- shewhart(qccobj, newdata,
+  labels=paste("Lot", 1:20, sep = ""))
```

The `labels` argument is not necessary but is added to show the printing of labels on the chart and for greater clarity in later paragraphs.

Printing the invisible return value of `shewhart` shows a summary of `qccobj` as well as `newdata`.

```
> qccobj.shew
xbar based on qcdata

Summary of Statistics in qcdata.
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.655  10.14 10.09   10.51 11.31

 Group Sample Size:  5
 Number of Groups:  20
 Center of Statistics:  10.09016
 Standard Deviation:  1.022341

Summary of New Data Statistics in newdata.
  Min. 1st Qu. Median  Mean 3rd Qu.  Max.
 9.163   9.762  10.14 10.24   10.84 11.49

 Group Sample Size:  5
 Number of Groups:  20

 Target Value: 10.09016

 Control Limits:
      LCL     UCL
 8.585714 11.5946
```

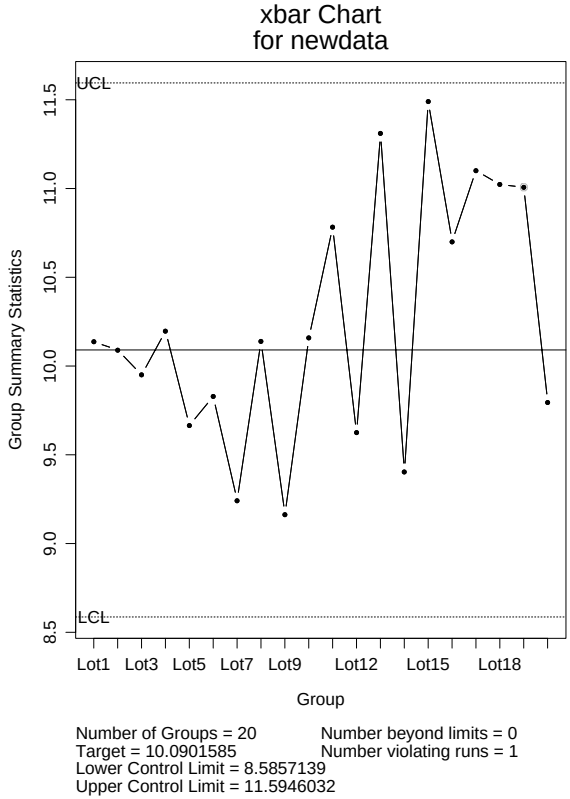

Figure 27.2: *Shewhart chart of* `newdata` *using* `qccobj` *as the reference data plotting only the new data.*

Figure 27.2 displays the chart for `newdata`. If you want to see `newdata` displayed alongside the original calibration data ask `shewhart` to chart it all. Having saved the `"shewhart"` object, `qccobj.shew`, you can chart it directly.

```
> shewhart(qccobj.shew, chart.all = T)
```

Figure 27.3 shows the resulting Shewhart chart with both old and new data. The vertical dashed line separates the in-control calibration data from the ongoing process data.

To do an s chart of the same data, you would type:

```
> shewhart(qcc(qcdata, "s"), newdata)
```

The `type` argument allows you to specify a different kind of summary statistic for `newdata` than what is in the reference data in `object`. For example, `qccobj.med` computed in section 27.1 contains robust estimates

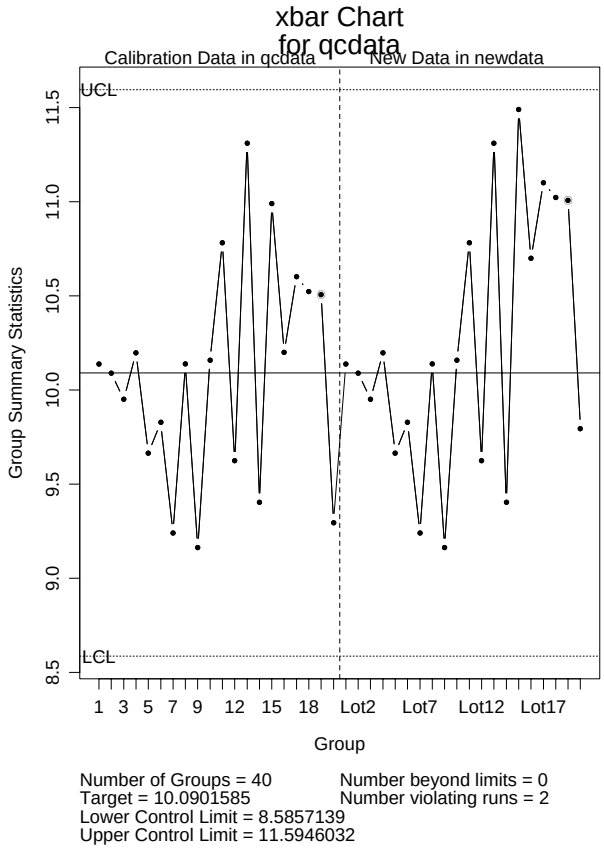

Figure 27.3: *Shewhart chart of* `newdata` *using* `qccobj` *as the reference data plotting new and old data.*

of location and scale for the reference data `qcdata`. You wouldn't, however, typically want to estimate the location of the ongoing process robustly, since extreme values are what you are looking for. In this case you can compute the control limits based on the robust estimates and then compute the group summary statistics of the ongoing process by specifying the usual `type` for the data you are using. Thus you could chart `newdata` with control limits based on the robust estimates of location and scale as follows:

```
> shewhart(qccobj.med, newdata, type = "xbar")
```

If you want to compute the summary statistics for `newdata` in the same way you did for the reference data, you don't have to specify `type`. Thus,

```
> shewhart(qccobj.med, newdata, limits = limits.xbar)
```

would continue to estimate the group summary statistics with `stats.med`,

that is, robustly. The `limits` argument must be provided when using a summary statistics function, as specified by `type`, other than one of the built-in ones, or a function must be available with name produced by `paste("limits.", type, sep = "")`.

Since `limits.xbar` simply uses the `center` and `std.dev` components of `object` to compute the control limits based on having normally distributed data, it is reasonable although not exactly correct to use `limits.xbar` here. Ideally, you would write a `limits.med` function to compute the control limits in this case. For more information on the way the control limits are computed by `shewhart`, see the help file for `"shewhart.limits"`. You can use the `limits.xbar` as a template for writing your own limits function.

The `shewhart` function returns an object that contains all the information necessary to redo the chart. It contains all the components of `object`, the `"qcc"` object, plus the following additional components:

`new.statistics`	a vector of group summary statistics for `newdata`.
`new.sizes`	vector of group sample sizes for `newdata`.
`target`	the `target` argument if specified.
`cntrl.limits`	the control limits.
`newdata.name`	a character string containing the name of the input data passed as the argument to `newdata`.

When you are tracking a process, you can repeatedly capture the return value from `shewhart`, passing it as the new `object` argument to a subsequent call to `shewhart`, and providing even newer data as the `newdata` argument. The `shewhart` function will incorporate the newest data into the `new.statistics` component of `object` and chart all the new data. The function calls might look something like the following:

```
> qccobj.shew.1 <- shewhart(qccobj, newdata.1)
> qccobj.shew.2 <- shewhart(qccobj.shew.1, newdata.2)
```

Other arguments to `shewhart`, listed above, allow you to specify a target value for the process, sample sizes, the confidence level of the probability limits, and a rules function for applying to the chart. By specifying sample sizes, you can supply a vector of group summary statistics instead of the entire data matrix. In this case, however, you must also specify the within-group standard deviations.

A rules function refers to a way of examining the plotted summary statistics to see if there are patterns suggesting a shift in the process. For example, five or more successive points on one side of the center may indicate a shift in the process. The function `runs.target` is provided for checking for runs in a

process and `beyond.limits` is provided for locating points beyond the control limits. Look at the help files of these functions for more detail. By default, `shewhart` applies both `runs.target` and `beyond.limits`, through a wrapper function called `shewhart.rules`, to a chart by highlighting violating points. The default is to highlight the points in the same way, regardless of which rule is violated. If you want to highlight them differently, give a list of lists of `par` parameters to the `highlight` argument.

```
> shewhart(qccobj.shew, highlight=list(list(pch=1,col=2),
+ list(pch=2, col=3)))
```

Any of the three rules functions provided can be applied directly to the return object of the `shewhart` function to produce a list of violating points. For example,

```
> shewhart.rules(qccobj.shew)
[[1]]:
  o  q
 15 17
attr([[1]], "label"):
[1] "beyond limits"

[[2]]:
  s  t
 19 20
attr([[2]], "label"):
[1] "violating runs"
```

The value returned is a list with a component for each rule containing the indices of the violators appropriately labeled.

To add labeling information to a chart you can use the `identify` function. There is an identify method for objects of class `"shewhart"`. You proceed by charting the object with no statistics and then applying `identify` to the chart.

```
> shewhart(qccobj.shew, add.stats = F)
> identify(qccobj.shew)
[1] 19
```

Figure 27.4 displays the resulting chart with the 19th observation labeled.

Applying rules such as `runs.target` usually makes a Shewhart chart more sensitive to small shifts off the center. However, such rules are typically *ad hoc*. A better way to detect small shifts is through the use of cusum charts.

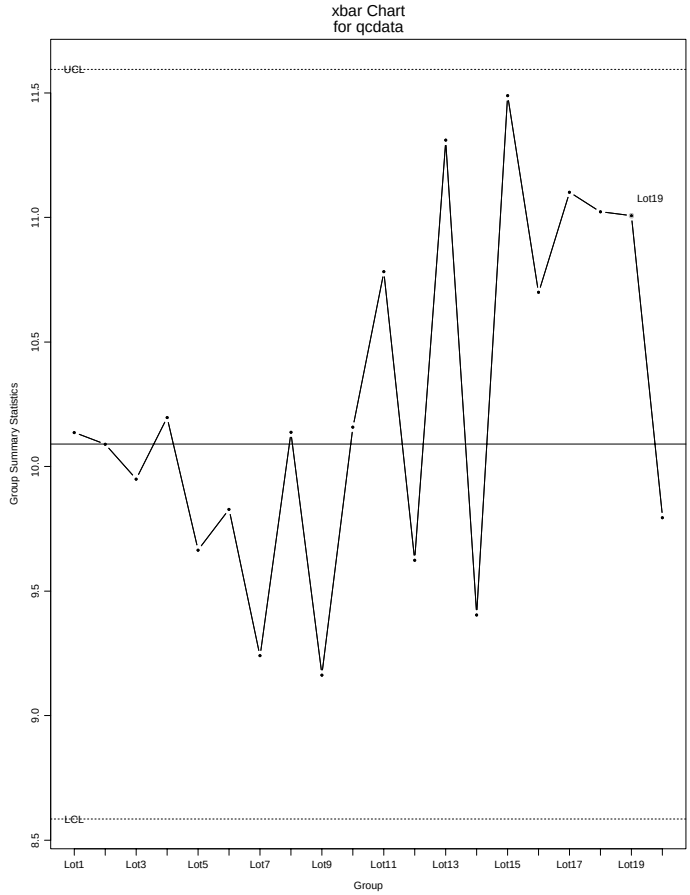

Figure 27.4: *Shewhart chart of the new data in* `qccobj.shew` *with the* 19*th observation labeled.*

27.3 CUSUM CHARTS

Cusum charts display how the group summary statistics deviate above or below the process center or target value relative to the standard errors of the summary statistics. In essence, a cusum chart accumulates *z*-scores of deviations above (below) the center and charts them. Consequently, the points plotted are not the original data but cumulative sums of deviations in standard errors from the center.

For an xbar chart, the upper, S_{Ui}, and lower, S_{Li}, cumulative sums are defined

as follows:

$$S_{Ui} = max\{0, (z_i - k) + S_{Ui-1}\} \quad (27.1)$$

$$S_{Li} = max\{0, (-z_i - k) + S_{Li-1}\} \quad (27.2)$$

where

$$z_i = \frac{\bar{x}_i - \bar{\bar{x}}}{\sigma_{\bar{x}_i}}$$

is the *z*-score for the *i*th group centered about the center of the group summary statistics denoted here as $\bar{\bar{x}}$. The lower cumulative sums are charted as $-S_{Li}$. Cusum charting in S-PLUS follows a decision interval scheme discussed in detail by Ryan (1989) and Wetherill and Brown (1991).

The *k* in equations 27.1 and 27.2 is called the *reference value* and corresponds to the amount that the absolute *z*-score must exceed the target before the either cumulative sum increases.

The cusum chart in S-PLUS is really a composite of two charts; a chart of the upper cumulative sums and a chart of negative the lower cumulative sums. The two sums, typically charted separately in standard quality control text books, are plotted on the same graph by the `cusum` function in S-PLUS.

For our simulated data sets you can do a cusum chart of the original data as follows:

```
> cusum(qccobj)
```

To see the new data charted, request it in addition to specifying the reference data in `qccobj`. You can also plot both old and new data by specifying `chart.all = T`. For example:

```
> cusum(qccobj, newdata, chart.all = T)
```

Figure 27.5 displays the cusum chart for both old and new data. Comparing figure 27.5 with the Shewhart chart displayed in figure 27.2 reveals how dramatically cusum charts signal a detectable shift in the process. In `newdata`, the last six observations were shifted up one standard deviation of the population which is about two standard errors of the summary statistics.

Various arguments to `cusum` control different aspects of the cusum chart. A summary of the arguments to `cusum` are:

`object` an object of class `"qcc"` which provides information on the type of group summary statistics to compute and

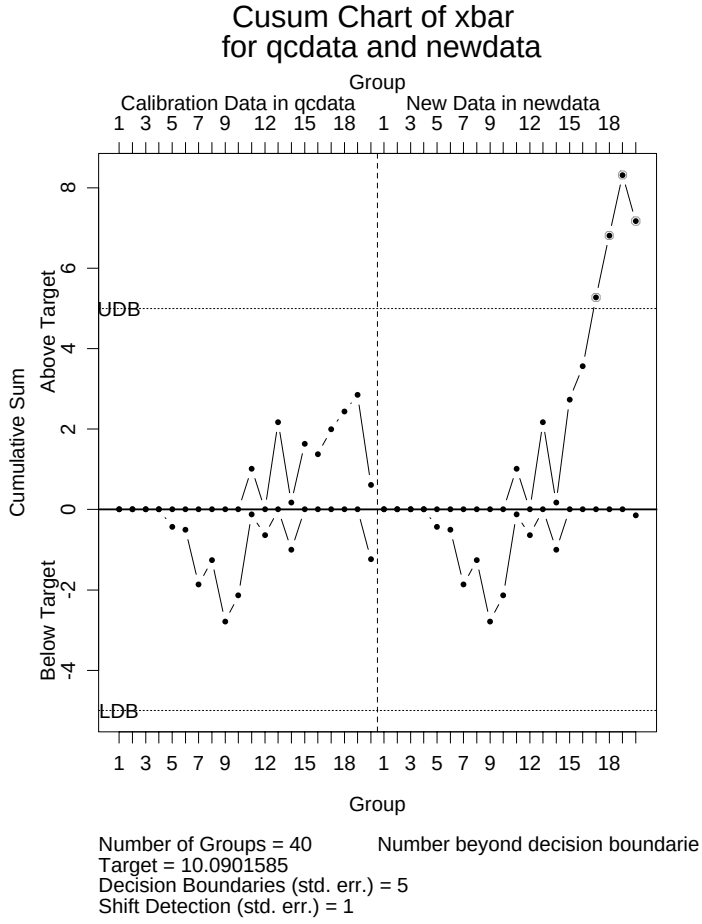

Figure 27.5: *Cusum chart of* `newdata` *using* `qccobj` *as the reference data plotting both old and new data.*

the within group standard deviation necessary for computing the *z*-scores.

`newdata` vector, matrix, data frame, or list to be charted.

`type` a character string or function specifying group statistics to compute.

`z.scores` optional function to be used to compute the *z*-scores. This argument is required if `type` is not one of `"xbar"`, `"s"`, `"R"`, `"p"`, `"np"`, `"u"`, or `"c"`, or if there does not exist a function with name produced by

`paste("zs.", type, sep = "")`.

`decision.int` a numeric value in number of standard errors of the summary statistics at which the cumulative sum signals out of control.

`se.shift` the amount of shift to detect in the process measured in standard errors of the summary statistics.

`target` a numeric value specifying the center of the process if other than the `center` component of `object`.

`std.dev` a numeric value specifying the overall within group standard deviation.

`sizes` a numeric vector specifying the sample sizes associated with each group of `newdata`.

`labels` character vector of labels to associate with each group of `newdata`.

`label.bounds` a character vector of length two with labels for the decision interval boundaries.

`headstart` a numeric value in standard errors of the group summary statistics at which to start the cumulative sums when `reset = TRUE`.

`reset` a logical value indicating whether the cumulative sums should be reset after an out-of-control signal.

`add.stats` a logical value indicating whether statistics should be listed at the bottom of the chart.

`chart.all` a logical value indicating whether the cusums of the `statistics` component of `object` should be charted along with the cusums of the `new.statistics` component of `object` if present and the cusums of the summary statistics of `newdata` if given.

`ylim.min` a numeric vector of values to be included in the computation of the approximate *y*-axis limits for the control chart.

`check.cl` a logical value indicating whether the summary statistics beyond the control limits of the Shewhart chart should be highlighted on the chart in addition to the decision boundary violations of the cumulative sums of the summary statistics.

`highlight` a list of plotting parameters to be used for highlighting the points outside the decision boundaries or beyond the Shewhart control limits.

The `type` argument is the same as that specified for the `shewhart` function. If `type` is one of `"xbar"`, `"s"`, `"R"`, `"p"`, `"np"`, `"u"`, or `"c"` there are built in functions for computing the group summary statistics and the *z*-scores. If `type` is not one of these, then you either need to produce two functions with names produced by `paste("stats.", type, sep = "")` and `paste("zs.", type, sep = "")` or pass functions to the `type` and `z.scores` arguments in the call to `cusum`.

The `type` and `z.scores` arguments are useful when charting is based on non-standard summary statistics. Going back to the example where the estimate of the center of the process is based on the median and the standard deviation is based on the mad (median absolute deviation) estimator, you can generate cusum charts in several different ways. If the type component of `qccobj.med` is equal to `"med"`, and you have defined the functions `stats.med` and `zs.med`, you can simply type

```
> cusum(qccobj.med, newdata)
```

If you haven't defined appropriately functions or if you want to use some function other than the one that would be found automatically, you have to specify their names explicitly in the call to `cusum`. For example, to do a cusum chart of the group *means* of `newdata` with center and standard deviation based on the median and mad, respectively, use the built in functions by specifying `type = "xbar"`. Not only will `stats.xbar` be used to compute the summary statistics, but the *z*-score function associated with xbar charts, `zs.xbar`, will be used as well.

```
> cusum(qccobj.med, newdata, type = "xbar")
```

The `se.shift` argument is twice the reference value, *k*, in equations 27.1 and 27.2. This corresponds roughly to the sensitivity of the cusum chart in terms of detecting shifts in standard errors of the summary statistics. Setting `se.shift = 1` (the default) corresponds to a cusum chart being sensitive to one standard error shifts and is equivalent to setting $k = 1/2$ in equations 27.1 and 27.2.

Usually when an out-of-control signal is generated by a large (in absolute value) cumulative sum, a search is conducted and a cause is assigned and removed if possible to correct the process. In this case, the cumulative sums are reset and monitoring continues. By resetting the sums to something other than zero (called a headstart), you can produce a fast initial response (FIR) cusum. This is useful for quickly detecting a process that hasn't been fully corrected. When `reset = TRUE` the cusums will be reset to `headstart`

each time a cumulative sum exceeds one of the decision boundaries.

One additional improvement to cusum charts results from checking for a large deviation from the target value of a single group summary statistic. A group summary statistic greater than 3 standard errors from the target is equivalent to that summary statistic being outside 3-sigma Shewhart control limits. When `check.cl = TRUE`, summary statistics violating Shewhart control limits are flagged as well as large cumulative sums. If `object` is of class `"shewhart"`, it will have a `cntrl.limits` component which will be used to check for violating summary statistics. Otherwise, 3-sigma Shewhart control limits, centered about `target`, are computed to check for violating summary statistics.

27.4 PROCESS MONITORING

In many manufacturing situations processes are monitored in real time by production engineers and product managers. You can use S-PLUS for real-time monitoring with a few simple functions. Examples are presented below of two functions, `monitor` and `get.process`, which you can use to monitor a process data file and update a control chart as data comes in.

The basic idea is the following:

1. Create a file for accumulating the process data; call it **Process**.
2. Track the growth of **Process** with `get.process` and `monitor`, updating the control chart only when new data has been added to the file.

Suppose a typical line of the data file looks like the following:

Lot1 9.496215 8.718396 11.470395 9.671888 11.328800

Also, suppose you want to accumulate the data in a matrix. Then you could write the data-reading function, `get.process`, as follows:

```
> get.process
function(file, skip = 0)
{
  data <- scan(file, what = list(names = "",0,0,0,0,0),
                skip = skip)
  nm <- data$names
  data <- cbind(data[[2]],data[[3]],data[[4]],data[[5]],
                data[[6]])
  dimnames(data) <- list(nm, NULL)
  data
}
```

The configuration of the data fields are built into the `get.process` function. The first field is a character label and the remaining 5 fields are

numeric data. The `skip` argument is added so that previously read data can be skipped when it is time to update the chart.

The `monitor` function keeps track of which data have already been read and updates the chart. An example of what `monitor` might look like is the following:

```
> monitor
function(file, qc.object, sleep.time = 5)
{
# define a subfunction
   file.length <- function(file)
   as.numeric(unix(paste("wc",
   file, "| awk '{print $1}'")))
#
#
   old.length <- file.length(file)
   new.data <- get.process(file)
#
# put up initial chart
#
   qcc.shew <- shewhart(qc.object, new.data,
   add.stats = F)
   cat("to quit type CNTRL-C\n")
   repeat {
      new.length <- file.length(file)
      if(new.length > old.length) {
#
# new data have come in, we need to update the plot
#
         new.data <- get.process(file, skip = old.length)
         old.length <- new.length
         qcc.shew <- shewhart(qcc.shew, new.data,
          add.stats = F)
      }
      unix(paste("sleep", sleep.time))
   }
}
```

The statistics on the bottom of the chart have been turned off so that a number of charts can be efficiently placed within a single figure. The `monitor` function makes use of the fact that `shewhart` updates its return object so that all you need to scan each time is the data that has just been added to the file.

Suppose now that `qcdata`, defined in section 27.1, is coming in one row (corresponding to one lot) at a time. Start the monitoring by putting the first lot in the file **Process** and then running `monitor` as follows:

```
> monitor("Process", qccobj)
to quit type CNTRL-C
```

S-PLUS now monitors **Process** for a change in size. When one is detected, the new data is read in and the chart is updated. Figure 27.6 displays the results of 19 updates.

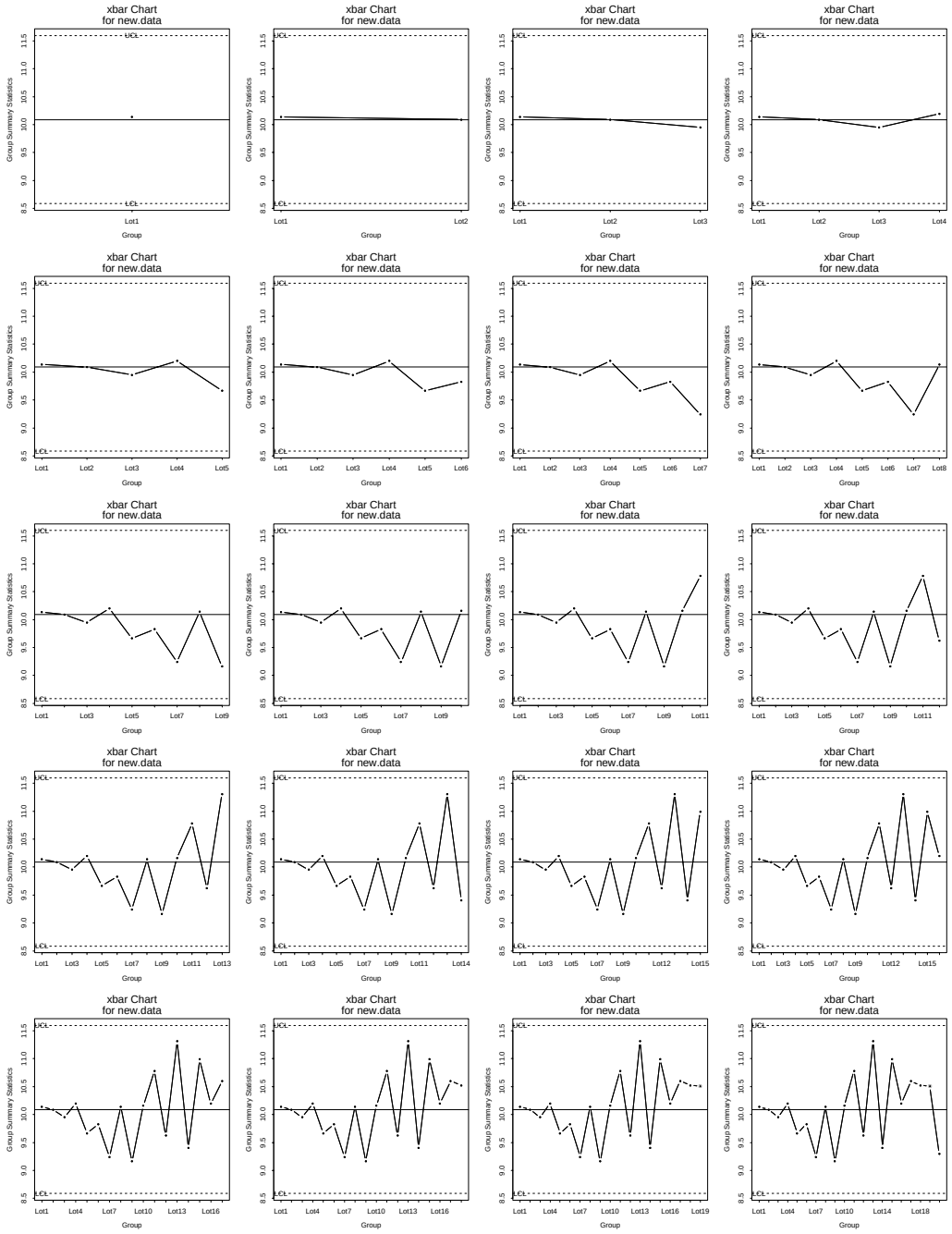

Figure 27.6: *A series of Shewhart charts of the data resulting from running* `monitor` *on a growing process data file.*

27.5 REFERENCES

Ryan, T. P. (1989). *Statistical Methods for Quality Improvement*. John Wiley and Sons, New York.

Wetherill, G. B. and D. W. Brown (1991). *Statistical Process Control*. Chapman and Hall, New York.

MATHEMATICAL COMPUTING IN S-PLUS

28

Numerical tasks, from arithmetic to complex matrix operations, can be built into a statistical analysis.

MATHEMATICAL COMPUTING IN S-PLUS

S-PLUS was designed for data analysis, so it is rich in quantitative methods. Many of these methods, while designed for particular data analysis tasks, have been implemented as general mathematical tools. These tools can be applied to a wide variety of numerical applications. This chapter is a brief survey of mathematical computing in S-PLUS.

In this chapter, we assume a basic familiarity with the operation of the command line. For the most part, however, this chapter is self-contained and can be read independently of the other chapters in this manual.

28.1 ARITHMETIC OPERATIONS

You perform basic arithmetic in S-PLUS as you would with a calculator, using the operators `+`, `-`, `*`, and `/`:

```
> 2 + 2
[1] 4
> 9 - 3
[1] 6
> 3 * 8
[1] 24
> 17 / 4
[1] 4.25
```

Use the operator `^` for exponentiation, including root extraction:

```
> 3 ^ 2
[1] 9
> 7 ^ (1 / 3)
[1] 1.912931
```

Operators have their usual precedence (powers, multiplication/division, addition/subtraction), and parentheses can be used (as in the previous example) to group calculations. Two other operators provide integer quotients and remainders. The integer divide operator, `%/%`, returns the integer quotient q and the modulo operator, `%%`, returns the remainder r of two numbers y and x, so that $y=qx+r$:

```
> 24.5 %/% 3.2
[1] 7
> 24.5 %% 3.2
[1] 2.1
> 7 * 3.2 + 2.1
[1] 24.5
```

The `abs` function returns the absolute value of a number:

```
> abs(-4.5)
[1] 4.5
```

The greatest-integer function $\lfloor x \rfloor$ is obtained using `floor`:

```
> floor(2.3)
[1] 2
```

Similarly, the "next integer" $\lceil x \rceil$ is obtained using `ceiling`:

```
> ceiling(2.3)
[1] 3
```

A *vector* in S-PLUS is an ordered set of values. Simple numeric vectors can be created with the `c` function or the sequence operator (`:`):

```
> x <- c(3,1,7)
> x
[1] 3 1 7
> w <- 1:6
> w
[1] 1 2 3 4 5 6
```

A *matrix*, in S-PLUS, is simply a vector with a specified number of rows and columns, that is, an ordered set of data in a rectangular array. You can create matrices with the `matrix` command:

```
> A <- matrix(c(19,8,11,2,18,17,15,19,10),nrow=3)
     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
```

You can also build matrices from existing vectors using `rbind` (which assigns vectors to the *rows* of the matrix) or `cbind` (which assigns vectors to the *columns* of the matrix):

```
> m <- c(14,13,10)
> n <- c(10,11,15)
> o <- c(19,3,15)
> B <- cbind(m,n,o)
> B
      m  n  o
[1,] 14 10 19
[2,] 13 11  3
[3,] 10 15 15
```

Most calculations on vectors or matrices are carried out *element by element*, so

for example, if $X=\{x_{ij}\}$ and $Y=\{y_{ij}\}$, we have $X*Y=\{x_{ij}y_{ij}\}$. Multiplying `A` times `B` with the standard `*` operator yields the following:

```
> A*B
     [,1] [,2] [,3]
[1,]  266   20  285
[2,]  104  198   57
[3,]  198  255  150
```

For matrices, these element by element operations require that the matrices have the same dimension; that is, the same number of rows and the same number of columns, so that the matrices are *conformable for addition*. For vectors, if one vector is shorter than the other, the shorter vector is repeated cyclically to match the length of the longer vector:

```
> x + w
[1]  4  3 10  7  6 13
```

Mathematical operations on combinations of vectors and matrices are permitted, but may have unexpected results. For example, suppose you define the matrix `E` as follows:

```
> E <- matrix(1:4,nrow=2)
```

Dividing by the previously defined vectors `x` and `w` yields the following results:

```
> E/w
[1] 1.0000000 1.0000000 1.0000000 1.0000000 0.2000000
[6] 0.3333333
Warning messages:
  Length of longer object is not a multiple of the
 length of the shorter object in: E/w
> E/x
          [,1]      [,2]
[1,] 0.3333333 0.4285714
[2,] 2.0000000 1.3333333
Warning messages:
  Length of longer object is not a multiple of the
  length of the shorter object in: E/x
```

S-PLUS returns an object with the attributes of the longer object in the calculation. Since `length(E) < length(w)`, `E/w` returned an object matching the attributes of `w`, namely a vector of length 6. On the other hand, since `length(E) > length(x)`, `E/x` returned an object matching the attributes of `E`, namely, a matrix of length 4 with `dim=c(2,2)`.

To perform matrix multiplication, use the *matrix multiplication operator* `%*%`

```
> A %*% B
     [,1] [,2] [,3]
[1,]  562  437  592
[2,]  688  563  491
[3,]  555  447  410
```

The two matrices must be *conformable for multiplication*, that is, the number of columns of `A` must be the same as the number of rows of `B`.

Using the matrix multiplication operator on two equal length vectors yields the *vector dot product*:

```
> z <- c(1,0,3,4,8)
> y <- c(2,9,3,2,7)
> z %*% y
     [,1]
[1,]   75
```

28.2 COMPLEX ARITHMETIC

In addition to the ordinary operators described in section 28.1, Arithmetic Operations, five special operators are provided for manipulating complex numbers.

`Re` and `Im` are used to extract the real and imaginary parts, respectively, from a complex number. `Mod` and `Arg` return the *modulus* and *argument* for the polar representation of the complex number. `Conj` returns the complex conjugate of the complex number.

When you graph a vector of complex numbers with `plot`, the real parts are graphed along the *x*-axis and the imaginary parts are graphed along the *y*-axis.

28.3 ELEMENTARY FUNCTIONS

The elementary functions included in S-PLUS are listed in table 28.1.

Each function acts *element-by-element* on its argument:

```
> J
     [,1] [,2] [,3] [,4]
[1,]   12   15    6   10
[2,]    2    9    2    7
[3,]   19   14   11   19
> sqrt(J)
         [,1]     [,2]     [,3]     [,4]
[1,] 3.464102 3.872983 2.449490 3.162278
```

Table 28.1: *Elementary Functions in S-PLUS .*

Name	Operation
`sqrt`	Square root
`abs`	Absolute value
`sin, cos, tan`	Trigonometric functions (radians)
`asin, acos, atan`	Inverse trigonometric functions (radians)
`sinh, cosh, tanh`	Hyperbolic trigonometric functions (radians)
`asinh, acosh, atanh`	Inverse hyperbolic trigonometric functions (radians)
`exp, log`	Exponential and natural logarithm
`log10`	Common logarithm
`gamma, lgamma`	Gamma function and its natural logarithm

```
[2,] 1.414214 3.000000 1.414214 2.645751
[3,] 4.358899 3.741657 3.316625 4.358899
> tan(J)
           [,1]       [,2]         [,3]      [,4]
[1,] -0.6358599 -0.8559934   -0.2910062 0.6483608
[2,] -2.1850399 -0.4523157   -2.1850399 0.8714480
[3,]  0.1515895  7.2446066 -225.9508465 0.1515895
```

You can use `log` to compute logarithms of any base with the optional argument `base=`. For example, to compute $\log_2 7$:

```
> log(7,base=2)
[1] 2.807355
```

28.4 VECTOR AND MATRIX COMPUTATIONS

The p-norm of a vector $\mathbf{x}$ of length n is defined as:

$$[\mathbf{x}_1^p + \mathbf{x}_2^p + \cdots + \mathbf{x}_n^p]^{1/p}$$

for $p \geq 1$. To obtain the p-norm of a vector in S-PLUS, use the `vecnorm` function (by default, p=2):

```
> vecnorm(1:2)
[1] 2.236068
```

```
> ( sum( (1:2) ^ 2) ) ^ (1/2)
[1] 2.236068
```

The `vecnorm` function works with both real and complex vectors:

```
> vecnorm(1+2i)
[1] 2.236068
```

You can specify the type of norm desired with the `p` argument. Possible values include real numbers greater than or equal to 1, `Inf`, and the character strings `"euclidean"` or `"maximum"`:

```
> vecnorm(1:2, p = 1)
[1] 3
> vecnorm(1:2, p = "maximum")
[1] 2
> vecnorm(1:2, p = Inf)
[1] 2
```

To obtain the transpose of a matrix, use the `t` function:

```
> J
     [,1] [,2] [,3] [,4]
[1,]   12   15    6   10
[2,]    2    9    2    7
[3,]   19   14   11   19
> t(J)
     [,1] [,2] [,3]
[1,]   12    2   19
[2,]   15    9   14
[3,]    6    2   11
[4,]   10    7   19
```

You can obtain the diagonal of a matrix with the `diag` function:

```
> diag(J)
[1] 12 9 11
```

You can also use `diag` to construct diagonal matrices:

```
> x <- c(3,1,7)
> diag(x)
     [,1] [,2] [,3]
[1,]    3    0    0
[2,]    0    1    0
[3,]    0    0    7
```

To obtain the *trace* of a square matrix, use `sum` with `diag`, as follows:

```
> sum(diag(A))
[1] 47
```

For another approach to vector and matrix computations, see also chapter 29, The Object-Oriented Matrix Library.

Identity Matrices

To generate identity matrices in S-PLUS, use `diag` with an integer argument representing the rank *n* as follows:

```
> diag(n)
```

For example, the rank 4 identity matrix is created as follows:

```
> diag(4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
```

Determinants

There is no built-in S-PLUS function to calculate determinants. However, the following one-line function can be used to calculate determinants for *real*-valued matrices:

```
> det <- function(x) prod(eigen(x)$values))
```

(The `eigen` function is discussed in section 28.6.)

Kronecker Products

A *Kronecker product* of two matrices $\boldsymbol{A}_{p\times q}$ and $\boldsymbol{B}_{m\times n}$ is the matrix

$$\begin{bmatrix} a_{11}\boldsymbol{B} & \cdots & a_{1q}\boldsymbol{B} \\ \vdots & & \vdots \\ a_{p1}\boldsymbol{B} & \cdots & a_{pq}\boldsymbol{B} \end{bmatrix}$$

To calculate a Kronecker product in S-PLUS, use the `kronecker` function:

```
> N <- matrix(5:8,nrow=2)
> O <- matrix(4:1,nrow=2)
> kronecker(N,O)
     [,1] [,2] [,3] [,4]
[1,]   20   10   28   14
[2,]   15    5   21    7
[3,]   24   12   32   16
[4,]   18    6   24    8
```

You can generalize `kronecker` to other operations besides multiplication by changing the operator with the `fun` argument:

```
> kronecker(N,O,fun="+")
```

```
     [,1] [,2] [,3] [,4]
[1,]    9    7   11    9
[2,]    8    6   10    8
[3,]   10    8   12   10
[4,]    9    7   11    9
```

28.5 SOLVING SYSTEMS OF LINEAR EQUATIONS

S-PLUS provides several methods for solving systems of linear equations such as the following:

$$19a + 2b + 15c = 9$$
$$8a + 18b + 19c = 5$$
$$11a + 17b + 10c = 14$$

This system of equations can be expressed as the matrix equation $\mathbf{A}x=y$, where $\mathbf{A}$ is the matrix of coefficients, x is the (column) vector of unknowns (a,b,c), and y is the column vector of known values (9,5,14). The `solve` function takes the square matrix of coefficients and the vector of known values as arguments and returns the solution vector:

```
> solve(A,c(9,5,14))
[1]  0.9914429  0.6161109 -0.7379758
```

You can also use `solve` to obtain the inverse of a matrix:

```
> solve(A)
            [,1]         [,2]        [,3]
[1,]  0.04219534 -0.069341989  0.06845677
[2,] -0.03806433 -0.007376807  0.07111242
[3,]  0.01829448  0.088816760 -0.09619357
```

If the matrix is singular, `solve` returns an error message:

```
> S <- matrix(c(9,3,3,3,1,1,2,4,7),ncol=3,byrow=T)
> solve(S)
Error in solve.qr(a): apparently singular matrix
Dumped
```

If the matrix of coefficients is upper triangular, you can use `backsolve` to solve the system of equations:

```
> U
     [,1] [,2] [,3]
[1,]    3    1    4
[2,]    0    1    5
[3,]    0    0    9
```

```
> backsolve(U,c(9,5,14))
[1]  1.851852 -2.777778  1.555556
```

Sections on Choleski Decomposition, QR Decomposition, and The Singular Value Decomposition follow. Information on using the Matrix library for matrix decompositions can be found in section 29.3.

Choleski Decomposition

For symmetric, positive-definite matrices, the *Choleski decomposition* factors the matrix $\boldsymbol{X}$ uniquely in the form $\boldsymbol{X}=\boldsymbol{R}^{\mathrm{T}}\boldsymbol{R}$, where $\boldsymbol{R}$ is upper triangular. You can use the Choleski decomposition to generate upper triangular matrices for use with `backsolve`. S-PLUS now has two functions for performing Choleski decomposition: `chol` and `choleski`. The `chol` function is most useful for obtaining new matrices, since it returns simply the upper triangular matrix $\boldsymbol{R}$. The `choleski` function returns a list with the $\boldsymbol{R}$ matrix as one of its components.

For more information on the Choleski decomposition, see the `chol` help file and chapter 8 of the *LINPACK User's Guide* by Dongarra, et al.

QR Decomposition

The *QR decomposition* expresses an $n \times p$ matrix $\boldsymbol{X}$ as the product of an $n \times n$ orthogonal matrix $\boldsymbol{Q}$ and an $n \times p$ upper triangular matrix $\boldsymbol{R}$. The *QR* decomposition is the foundation for `solve` and `lsfit`, the (non-robust) least-squares fit function.

To obtain a representation of the *QR* decomposition, use the `qr` function. The value returned by `qr` is a list representing the *QR* numerical decomposition. The first component of the list is an $n \times p$ matrix in which the upper triangle, including the diagonal, is the $\boldsymbol{R}$ matrix and the entries under the diagonal contain most of a compact representation of $\boldsymbol{Q}$. To obtain $\boldsymbol{R}$ and $\boldsymbol{Q}$ explicitly from this numerical representation, use the functions `qr.R` and `qr.Q`, respectively. Another function, `qr.X`, reconstructs the original $n \times p$ matrix $\boldsymbol{X}$ from the numerical decomposition:

```
> qr(A)
$qr:
           [,1]       [,2]       [,3]
[1,] -5.9160798 -4.9018947 -7.9444500
[2,]  0.5070926  2.2296701  3.6136032
[3,]  0.8451543  0.7681395 -0.9097177

$qraux:
[1] 1.169031 1.640282 0.000000

$rank:
[1] 3
```

```
$pivot:
[1] 1 2 3
> qr.Q(qr(A))
           [,1]        [,2]       [,3]
[1,] -0.1690309  0.97387888 -0.1516196
[2,] -0.5070926 -0.21784133 -0.8339078
[3,] -0.8451543 -0.06407098  0.5306686
> qr.R(qr(A))
         [,1]      [,2]       [,3]
[1,] -5.91608 -4.901895 -7.9444500
[2,]  0.00000  2.229670  3.6136032
[3,]  0.00000  0.000000 -0.9097177
> qr.X(qr(A))
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    3    2    4
[3,]    5    4    6
```

The following functions use the return value from `qr` to perform additional calculations:

`qr.coef` Returns the coefficients obtained by a least-squares fit of response data *y* to the ***X*** matrix on which `qr` was used.

`qr.fitted` Returns the fitted values obtained by a least-squares fit of response data *y* to the ***X*** matrix on which `qr` was used.

`qr.resid` Returns the residuals obtained by a least-squares fit of response data *y* to the ***X*** matrix on which `qr` was used.

`qr.qy` Returns the results of the matrix multiplication `Q %*% y`, where ***Q*** is the order-`nrow(X)` orthogonal transformation represented by `qr` and *y* is the response data.

`qr.qty` Returns the results of the matrix multiplication `t(Q) %*% y`, where ***Q*** is the order-`nrow(X)` orthogonal transformation represented by `qr` and *y* is the response data.

For more details on the *QR* decomposition, see the help files for `qr`, `qr.coef`, and `qr.Q`, and chapter 9 of the *LINPACK User's Guide* by Dongarra, et al.

The Singular Value Decomposition

The *singular value decomposition* takes an $n \times p$ matrix ***X*** and decomposes it into two orthogonal matrices and a diagonal matrix. The elements of the diagonal matrix are the *singular values* of ***X***. The squares of the singular values of ***X*** are the eigenvalues of $\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X}$.

To obtain the singular value decomposition in S-PLUS, use the `svd` function, which returns a list in which the first component is the vector of singular values, the second component is the orthogonal matrix ***V***, and the third component is the orthogonal matrix ***U***:

```
> svd(A)
$d:
[1] 40.000114 14.687207  5.768609

$v:
           [,1]       [,2]       [,3]
[1,] -0.5280363  0.6449356  0.5524814
[2,] -0.5533835 -0.7547957  0.3522074
[3,] -0.6441618  0.1197558 -0.7554563

$u:
           [,1]       [,2]        [,3]
[1,] -0.5200456  0.8538399 -0.02258456
[2,] -0.6606048 -0.4188323 -0.62304157
[3,] -0.5414369 -0.3090905  0.78186261
```

The singular value decomposition can be used as a numerically stable way to perform many operations that are used in multivariate statistics. One such operation is estimating the *rank* of a matrix ***X***.

For more information on the singular value decomposition, see the `svd` help file and chapter 10 of the *LINPACK User's Guide* by Dongarra, et al.

28.6 EIGENVALUES AND EIGENVECTORS

If ***A*** is a square matrix and $\boldsymbol{A}x = \lambda x$, where λ is a scalar and x is a vector, then λ is an *eigenvalue* of ***A*** and x is an *eigenvector* of ***A***.

The S-PLUS function `eigen` returns both the eigenvalues and the eigenvectors associated with them:

```
> eigen(A)
$values:
[1]  39.581985  13.677784  -6.259769

$vectors:
          [,1]       [,2]       [,3]
[1,] 0.6224278  0.8664541  0.3124109
[2,] 0.8793762 -0.6095730  0.3450415
[3,] 0.7368032 -0.2261540 -0.5721007
```

For more information on the `eigen` function, see the `eigen` help file. See also the section The Eigen Decomposition, in chapter 29, The Object-Oriented Matrix Library.

28.7 INTEGRALS, DIFFERENCES, AND DERIVATIVES

Use the `integrate` function to compute the integral of a real-valued function over a given interval. The `integrate` function returns a list, of which the first two components are the integral and the absolute error:

```
> integrate(sin, 0, pi)[1:2]
$integral:
[1] 2

$abs.error:
[1] 2.220446e-14
>  (-cos(pi)) -  -cos(0)
[1] 2
```

Like many of the S-PLUS mathematical functions, `integrate` is most commonly used inside other function definitions. The following "wrapper" function provides a convenient command-line interface, and returns a single numeric value:

```
> integral <- function(f, lower, upper, ...) {
+        results <- integrate(f, lower, upper, ...)
+        if(results$message != "normal termination")
+                results$message
+        else results$integral
+ }
```

Use the `diff` function to obtain the *n*th difference of lag *k* for a set of data *x*. The default for both *k* and *n* is 1. The data may be in the form of a vector, time series, or matrix:

```
> y <- (1:10)^2
> diff(y)
[1]  3  5  7  9 11 13 15 17 19
> diff(corn.rain)
1891:  3.3 -3.0 -1.2 -1.9  5.7  0.5 -2.9  0.0  0.0  0.7
1901: -3.0  8.4 -2.1 -3.5 -0.6  1.5  2.1 -1.5 -0.1 -2.7
1911: -1.6  3.3 -4.1  2.6  7.0 -7.2  0.1 -0.7  0.8  2.1
1921:  0.5 -4.1  2.7  3.2 -2.6  0.3 -1.2
```

Differences on matrices are performed on each column separately:

```
> K
     [,1] [,2]
[1,]   12   10
[2,]    2   16
[3,]   13    7
[4,]    5    1
> diff(K)
     [,1] [,2]
[1,]  -10    6
[2,]   11   -9
[3,]   -8   -6
```

You can use `diff` to write a function for approximating the derivative of a data set:

```
> numdiff <- function(y, x = seq(along = y))
+ diff(y)/diff(x)
```

To perform symbolic differentiation, use the `D` function. (AT&T suggests the `deriv` function, but `deriv` is most useful for providing derivatives to other S-PLUS functions. The `D` function is more useful for obtaining an isolated derivative.)

```
> D(expression(3*x^2),"x")
3 * (2 * x)
> D(expression(exp(x^2)),"x")
exp(x^2) * (2 * x)
> D(expression(log(y)),"y")
1/y
```

28.8 INTERPOLATION AND APPROXIMATION

S-PLUS has a variety of functions for interpolation and approximation, most of them developed to aid in fitting curves and lines to data. However, they are sufficiently general to have wide application in mathematical settings.

Linear Interpolation

To find interpolated values in S-PLUS, use the `approx` function. You provide a vector of *x* values and a vector of associated *y* values, and (optionally) a vector of *x* values at which you want interpolated values. S-PLUS returns a list of *x* values and the associated *y* values:

```
> approx(1:10,(1:10)^2,xout=c(2.5,3.5))
$x:
[1] 2.5 3.5

$y:
[1]  6.5 12.5
```

A more specialized interpolation function, `interp`, can be used to generate input for the three-dimensional plotting functions `image`, `contour`, and `persp`. The `interp` function interpolates the value of the *z* variable onto an evenly spaced grid of the *x* and *y* variables:

```
> x <- cos(seq(-pi,pi,len=9))
> y <- sin(seq(-pi,pi,len=9))
> z <- x + y
> slanted.disk <- interp(x,y,z)
> persp(slanted.disk)
```

The resulting plot is shown in figure 28.1.

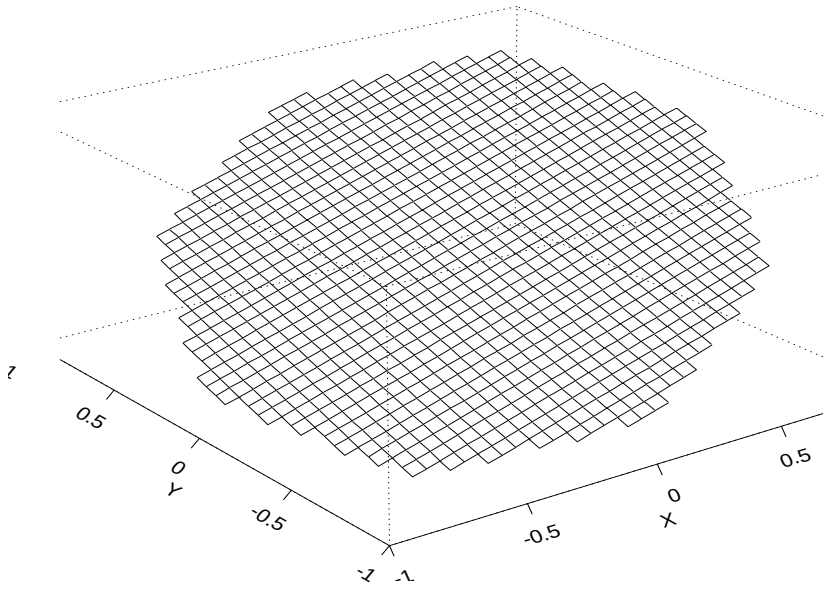

Figure 28.1: *A perspective plot created using* `interp`.

Convex Hull

To obtain the convex hull of a planar set of points, use the `chull` function, which returns the indices of the points belonging to the hull:

```
> chull(corn.rain)
 [1]  1  2 13 26 35 37 38 33 24  5
```

The `peel` option allows you to peel off the convex hull, take the convex hull of the remaining points, peel off *that* hull, and so on, until either all points are assigned to a hull or a user-specified limit is reached:

```
> chull(corn.rain,peel=T)
$depth:
 [1] 1 1 2 2 1 2 2 3 4 5 4 2 1 2 6 5 5 3 4 4 3 2 5 1 4 1 3
[28] 4 2 3 3 2 1 3 1 2 1 1

$hull:
 [1]  1  2 13 26 35 37 38 33 24  5  4  3  6  7 14 32 36 29
[19] 22 12 21  8 18 31 34 30 27  9 11 19 20 28 25 10 17 23
[37] 16 15

$count:
[1] 10 10  7  6  4  1
```

The `depth` component specifies which hull each point belongs to; `1` is the outermost hull. The `hull` component gives the indices of the points belonging to each hull. The first `count[1]` points belong to the outermost hull, the next `count[2]` points belong to the next hull, and so on.

Cubic Spline Approximation

Splines approximate a function with a set of polynomials defined on subintervals. A cubic spline is a collection of polynomials of degree less than or equal to 3 such that the second derivatives agree at the "knots;" that is, the spline has a continuous second derivative.

When interpolating a number of points, a spline can be a much better solution than a polynomial interpolation, since the polynomial can oscillate wildly in order to hit all of the points (polynomials fit the data globally while splines fit the data locally).

Use the `spline` function to obtain a cubic spline approximation:

```
> x <- 1:5
> y <- c(5,-5,0,-5,5)
> spline(x,y)
$x:
 [1] 1.000000 1.333333 1.666667 2.000000 2.333333 2.666667
 [7] 3.000000 3.333333 3.666667 4.000000 4.333333 4.666667

$y:
 [1]  5.0000000  0.1851852 -3.5185184 -5.0000000 -3.7037036
 [6] -1.2962964  0.0000000 -1.2962964 -3.7037036 -5.0000000
[11] -3.5185184  0.1851852
```

The `spline` function is primarily used for graphing, so by default it returns approximately three times as many output points as input points. For more details, see the `spline` help file.

Step Functions

The S-PLUS function `stepfun` creates a step function from either two vectors or a list with components named `x` and `y`. You can specify whether the step function is left or right continuous—the default is left.

```
> x <- seq(1,15,length=5)
> y <- x^2
> stepfun(x,y)
$x:
[1]  1.0  4.5  4.5  8.0  8.0 11.5 11.5 15.0 15.0

$y:
[1]   1.00   1.00  20.25  20.25  64.00  64.00 132.25 132.25
[9] 225.00
> plot(stepfun(x,y),type="l")
```

The resulting plot is shown in figure 28.2.

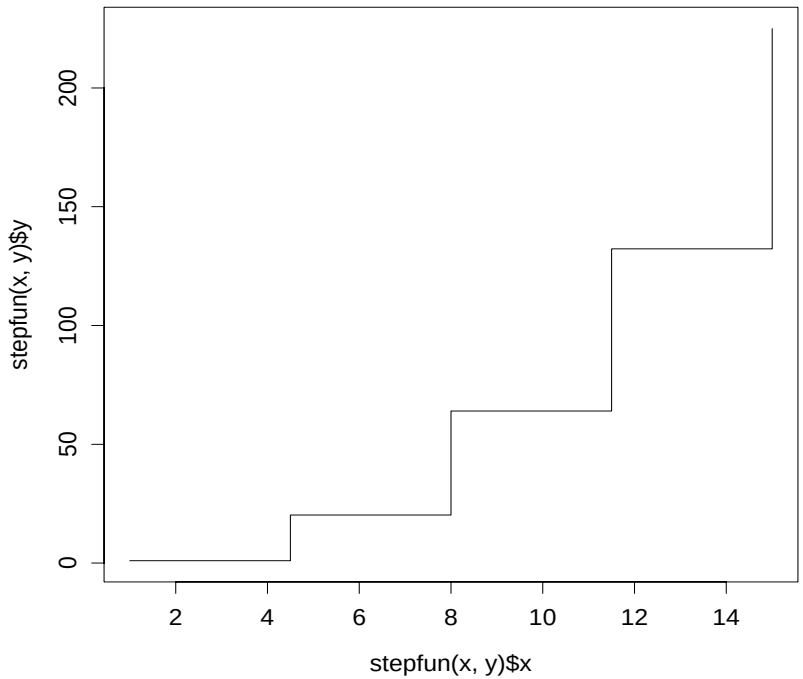

Figure 28.2: *A (left-continuous) step function.*

28.9 SIGNAL PROCESSING

S-PLUS has several functions useful for signal processing, including the fast Fourier transform and its inverse and several types of filters: convolution,

recursive, and low-pass. For a complete description of filters in S-PLUS, see section 21.6, Linear Filters.

The function `fft` calculates the unnormalized discrete Fourier transform of the input data, which can be any numeric or complex vector or array, including time series. The output is of mode `complex`.

```
> fft(1:10)
 [1]  55+ 0.000000i  -5+15.388418i  -5+ 6.881910i
 [4]  -5+ 3.632713i  -5+ 1.624598i  -5+ 0.000000i
 [7]  -5- 1.624598i  -5- 3.632713i  -5- 6.881910i
[10]  -5-15.388418i
```

If the input data is an array (for example, a matrix), `fft` returns the multi-dimensional unnormalized discrete Fourier transform of the array—a complex array with the same shape as the input data. Therefore, using `fft` on a multivariate time series does not compute the time transform.

```
> fft(A)
                  [,1]              [,2]              [,3]
[1,]  119.0+0.000000i  -2.5+ 6.062178i  -2.5- 6.062178i
[2,]   -5.5-6.062178i  23.0+20.784610i  11.0- 6.928203i
[3,]   -5.5+6.062178i  11.0+ 6.928203i  23.0-20.784610i
```

To compute the inverse transform, use `fft` with the argument `inverse=TRUE`.

```
> cuberoot.1 <- (cos(2*pi/3) + sin(2*pi/3)*1i)^(0:2)
> cuberoot.1
[1]  1.0+0.0000000i -0.5+0.8660254i -0.5-0.8660254i
> fft(cuberoot.1,inverse=T)
[1] 0.000000e+00+3.330669e-16i 2.220446e-16+3.142072e-16i
[3] 3.000000e+00-6.472741e-16i
```

28.10 PROBABILITY AND RANDOM NUMBERS

S-PLUS has many functions for performing probability calculations, including random number generation, in any of the most common distributions. Each of these functions has a name beginning with one of the following four one-letter codes indicating the type of function:

`r` Random number generator. Requires argument specifying sample size, plus any required distribution parameters.

`p` Probability function. Requires a vector of quantiles, plus any required distribution parameters.

`d` Density function. Requires a vector of quantiles, plus any required distribution parameters.

q Quantile function. Requires a vector of probabilities, plus any required distribution parameters.

The function code is concatenated with a code representing the desired distribution to form the function name. For example, the probability that a value from a standard normal distribution is less than *x* is calculated with the expression `pnorm(x)`. Table 28.2 lists the distributions currently supported by S-PLUS, along with the codes used to identify them.

For example, to compute the .95 quantile from a chi-square distribution with 5 degrees of freedom, use the following expression:

```
> qchisq(.95,5)
[1] 11.0705
```

The result says that 95% of numbers drawn from the given chi-square distribution will be less than 11.0705.

To generate 25 random numbers from a uniform distribution between -5 and 5, use `runif` as follows:

```
> runif(25,-5,5)
 [1] -1.03983 -0.11714 -2.41342  2.01498  0.48760
 [6]  1.55474 -3.83878 -4.04518 -2.39230 -0.47260
[11] -1.16530 -3.42732 -2.09373  2.24609  3.70265
[16]  3.67131  4.37430 -3.06433 -2.34121 -1.28586
[21] -0.91553  2.18947  2.12163 -2.04341 -2.87031
```

28.11 PRIMES AND FACTORS

S-PLUS can be useful in many number-theoretic computations, as we have already seen with the `%%` and `%/%` operators. You can define simple functions to list prime numbers and perform factorization; although they will not set computational records, you may find them useful.

The `primes` function returns all prime numbers less than or equal to a given *n*, where by default *n*=100:

```
> primes <- function(n = 100) {
+         n <- as.integer(abs(n))
+         if(n < 2)
+                 return(integer(0))
+         p <- 2:n
+         smallp <- integer(0)    # the sieve
+         repeat {
+                 i <- p[1]
+                 smallp <- c(smallp, i)
+                 p <- p[p %% i != 0]
```

Table 28.2: *Probability distributions in S-PLUS.*

Code	Distribution	Required Parameters	Optional Parameters	Defaults
`beta`	beta	shape1, shape2		
`binom`	binomial	size, prob		
`cauchy`	Cauchy		location, scale	0, 1
`chisq`	chi-square	df		
`exp`	exponential		rate	1
`f`	F	df1, df2		
`gamma`	Gamma	shape		
`geom`	geometric	prob		
`hyper`	hypergeometric	m, n, k		
`lnorm`	log-normal		mean, sd	exp(.5), exp(1)*(exp(1)-1)
`logis`	logistic		location, scale	0, 1
`nbinom`	Negative binomial	size, prob		
`norm`	normal		mean, sd	0, 1
`pois`	Poisson	lambda		
`stab`	stable	index	skewness	0
`t`	Student's t	df		
`unif`	uniform		min,max	0,1
`weibull`	Weibull	shape		
`wilcox`	Wilcoxon rank sum	m, n		

```
+                 if(i > sqrt(n))
+                         break
+         }
+         c(smallp, p)
+ }
```

```
> primes(75)
 [1]  2  3  5  7 11 13 17 19 23 29 31 37 41 43 47 53 59 61
[19] 67 71 73
```

The `factors` function returns the prime factors of an integer *n*:

```
> factors <- function(n) {
+        n <- as.integer(abs(n))
+        if(!exists(".Primes") || max(.Primes) < sqrt(n))
+                assign(".Primes", primes(as.integer(1.3 *
+                        sqrt(n))), where = 1)
+        pfactors <- integer(0)
+        while(n > 1) {
+                new.factors <- .Primes[n %% .Primes == 0]
+                if(length(new.factors) == 0)
+                        new.factors <- n
+                n <- as.integer(n/(prod(new.factors)))
+                pfactors <- c(pfactors, new.factors)
+        }
+        sort(pfactors)
+ }
> factors(3012)
[1]   2   2   3 251
```

28.12 INTERFACE TO MATHEMATICA

Note
The Mathematica interface is present in the Unix versions of S-PLUS only. It is not present in S-PLUS for Windows.

The S-PLUS interface to Mathematica allows you to use Mathematica as a numerical coprocessor and symbolic calculator from S-PLUS. The S-PLUS functions for the interface are in the library section `mathematica`. To make the `mathematica` library available during an S-PLUS session, use the following command:

```
> library(mathematica)
```

You send S-PLUS expressions to Mathematica with the `Math` function, which takes as its required argument a character string containing a valid Mathematica expression.

As a simple example, here is Mathematica's computation of the first 30 digits of π:

```
> Math("N[Pi,30]")
[1] "3.14159265358979323846264338328"
```

As another example, here is the numerical integral of the function $exp(x^2)$ from -10 to 10:

```
> Math("NIntegrate[Exp[-x^2],{x,-10,10}]")
[1] "1.772453850906074"
```

For complete details on the Mathematica interface, see *AT&T Bell Laboratories Statistical Research Report No. 97*, "An Interface from S to Mathematica," by Javier F. Cabrera and Allan R. Wilks. When the library section is attached to your search path, you can also find out more from the following help files:

```
D               as.math         math.remove
Integral        all.objects
Math            math.assign
```

28.13 A NOTE ON COMPUTATIONAL ACCURACY

S-PLUS performs its computations in double precision, unless specifically written as integer or single precision. Computed values are accurate to approximately 14 decimal places. However, computed values can provide no more significant digits than the data they are computed from.

The exact limits on computations in S-PLUS are determined by the parameters of machine arithmetic stored in the S-PLUS object `.Machine`. The object `.Machine` is a list with various numeric components whose names are made up of the characters `single.` or `double.` followed by the name of a particular parameter of machine arithmetic. For example, `single.digits` is the number of base `single.base` digits in the floating point representation of a single-precision number. In addition, the component `integer.max` is the largest integer.

See the `.Machine` help file for more information.

THE OBJECT-ORIENTED MATRIX LIBRARY 29

The traditional algebraic view of matrices, with a host of matrix operations, is available in the MATRIX library.

THE OBJECT-ORIENTED MATRIX LIBRARY

The Matrix library in S-PLUS provides a consistent, efficient, and fully object-oriented set of matrix operations and functions that reflect the traditional linear algebraic viewpoint. The functions are based on the LAPACK library of numerical Fortran routines. See the *LAPACK User's Guide* (1994) for details. The library includes constructor functions for a new `Matrix` class and numerous subclasses, and methods for many common matrix computations, including basic matrix arithmetic, decompositions, and solutions to systems of linear equations.

29.1 ATTACHING THE MATRIX LIBRARY

To use the Matrix library, you must first attach it using the `library` function:

```
> library(Matrix)
```

You can view the full list of Matrix functions with the following command:

```
> objects(grep("Matrix", search()))
 [1] "%*%.Matrix"                ".First.lib"
 [3] ".laenv"                    "Arg.Identity"
 [5] "Arg.Matrix"                "ColOrthogonal.test"
 [7] "ColOrthonormal.test"       "ColPermutation"
 [9] "Diagonal"                  "Diagonal.test"
[11] "Hermitian.test"            "Identity"
[13] "Identity.test"             "Im.Diagonal"
[15] "Im.Identity"               "Im.Matrix"
[17] "LowerTriangular.test"      "Matrix"
 . . .
```

29.2 BASIC MATRIX OPERATIONS

Working with objects of the new `Matrix` class is, in most simple cases, exactly like working with traditional S-PLUS matrices. However, throughout the chapter, we will use the word *Matrix*, with its initial capital, whenever we refer specifically to objects of this new class. A lower-case "m" indicates traditional S-PLUS matrices.

To construct a Matrix, use the `Matrix` function (which has the same arguments as the old `matrix` function):

```
> Matrix(1:12, nrow=3, ncol=4)
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
attr(, "class"):
[1] "Matrix"
```

By default, Matrices are filled in "by column." To fill the Matrix by rows, use the argument `byrow=T`:

```
> Matrix(1:12, nrow=3, ncol=4, byrow=T)
     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
attr(, "class"):
[1] "Matrix"
```

You can add row and column names by providing a list with two components (one of length `nrow` and one of length `ncol`) to the `dimnames` argument:

```
> Matrix(1:12, nrow=3, ncol=4, dimnames=list(
+ c("Row 1", "Row 2", "Row 3"),
+ c("Col 1", "Col 2", "Col 3", "Col 4")))

      Col 1 Col 2 Col 3 Col 4
Row 1     1     4     7    10
Row 2     2     5     8    11
Row 3     3     6     9    12
attr(, "class"):
[1] "Matrix"
```

As with any S-PLUS expression, the returned value can be stored as an S-PLUS object:

```
> A <- Matrix(c(19,8,11,2,18,17,15,19,10), nrow=3)
> B <- Matrix(c(14,13,10,10,11,15,19,3,15), nrow=3)
```

Warning
If you create Matrices and other objects from linear algebra using the `Matrix` library, you must always attach the `Matrix` library before working with those objects. Otherwise, you may encounter potentially confusing error messages. Also, do not expect classed Matrices to be suitable inputs for all functions which expect matrix inputs.

Matrix Arithmetic

Two Matrices with the same dimension—that is, the same number of rows and the same number of columns—are said to be *conformable for addition*. Such Matrices can be combined using the normal arithmetic operators `+`, `-`, `*`, and `/`; these operators act *element-by-element*, so that for $X = \{x_{ij}\}$ and $Y = \{y_{ij}\}$, $X*Y = \{x_{ij}y_{ij}\}$. Thus, multiplying `A` times `B` with the standard `*` operator yields the following:

```
> A*B
     [,1] [,2] [,3]
[1,]  266   20  285
[2,]  104  198   57
[3,]  110  255  150
attr(, "class"):
[1] "Matrix"
```

If you attempt to add a vector to a Matrix, you may be surprised by the results. In standard S-PLUS, if you operate on two objects with different lengths, S-PLUS returns an object with the attributes of the longer object. Thus, if you add a 3×3 matrix and a length 4 vector, you get a 3×3 matrix, and the length 4 vector is replicated to be the same length as the matrix before the addition is performed:

```
> matrix(1:9, ncol=3) + 1:4
     [,1] [,2] [,3]
[1,]    2    8   10
[2,]    4    6   12
[3,]    6    8   10
Warning messages:
  Length of longer object is not a multiple of the length
    of the shorter object in: matrix(1:9, ncol = 3) + 1:4
```

The same calculation is illegal with Matrices:

```
> Matrix(1:9, ncol=3) + 1:4
Error in e1 + e2: Dimension attributes do not match
Dumped
```

However, if the vector you want to add is *sweep-conformable* with your matrix—that is, if it is the same length as the number of rows or columns of your matrix—the operation can proceed:

```
>  Matrix(1:9, ncol=3) + 1:3
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    4    7   10
[3,]    6    9   12
```

```
attr(, "class"):
[1] "Matrix"
>  Matrix(1:9, ncol=3) + t(1:3)
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
attr(, "class"):
[1] "Matrix"
```

The first example above shows a *column sweep* operation, in which the column vector `1:3` is added to each column of the Matrix in turn. The second example shows a *row sweep* operation, in which the row vector `1:3` is added to each row of the Matrix in turn.

You can obtain the same results using the `sweep` function, but the basic operators are usually more convenient:

```
> sweep(Matrix(1:9, ncol=3), 2, 1:3, "+")
     [,1] [,2] [,3]
[1,]    2    6   10
[2,]    3    7   11
[3,]    4    8   12
attr(, "class"):
[1] "Matrix"
> sweep(Matrix(1:9, ncol=3),1, 1:3, "+")
     [,1] [,2] [,3]
[1,]    2    5    8
[2,]    4    7   10
[3,]    6    9   12
attr(, "class"):
[1] "Matrix"
```

Matrix multiplication requires that two Matrices X and Y be *conformable for multiplication*; that is, that the number of columns of X equal the number of rows of Y. Thus, if X is an $m \times n$ Matrix and Y is an $n \times p$ Matrix, the Matrix product XY is defined, but the Matrix product YX is not:

```
> X <- Matrix(rnorm(12), ncol=3)
> Y <- Matrix(rnorm(15), nrow=3)
> X %*% Y
           [,1]       [,2]       [,3]      [,4]       [,5]
[1,] -2.2592886  0.7046133  0.4268365  2.783703 -1.2639460
[2,]  0.5056705 -0.4573891  1.6069626  2.997998 -0.6174152
```

```
[3,]  0.4616003 -2.6992292 -2.6528110 -2.682271 -0.6169322
[4,] -2.5606158  2.2183770  0.4736680  1.556856 -0.3746409
attr(, "class"):
[1] "Matrix"
> Y %*% X
Error in "%*%.default"(x, y): Number of columns of x
   should be the same as number of rows of y
Dumped
```

For square Matrices A and B of the same dimension, both products are defined, but are not equal, in general:

```
> A %*% B
     [,1] [,2] [,3]
[1,]  442  437  592
[2,]  536  563  491
[3,]  475  447  410
attr(, "class"):
[1] "Matrix"
> B %*% A
     [,1] [,2] [,3]
[1,]  555  531  590
[2,]  368  275  434
[3,]  475  545  585
attr(, "class"):
[1] "Matrix"
```

One significant difference between the Matrix library and standard S-PLUS is in the behavior of matrix multiplication involving a vector. In standard S-PLUS, we have the following behavior when multiplying a matrix by a vector on the left:

```
> 1:3 %*% matrix(rnorm(9), ncol=3)
         [,1]     [,2]      [,3]
[1,] 10.88121 3.174175 -8.284594
```

The vector `1:3` is treated as a three-column row vector for purposes of the multiplication, and so the multiplication proceeds.

If we try the same multiplication with a Matrix, we get an error:

```
> 1:3 %*% Matrix(rnorm(9), ncol=3)
Error in "%*%.default"(1:3, Matrix(rnorm(9), ncol ..:
        Number of columns of x should be the same
        as number of rows of y
Dumped
```

The error occurs because the Matrix library consistently treats S-PLUS vectors as *column vectors*. To obtain a row vector, you must take the transpose of a column vector. Thus, we can obtain the desired product as follows:

```
> t(1:3) %*% Matrix(rnorm(9), ncol=3)
          [,1]     [,2]      [,3]
[1,] 5.231949 0.546737 -8.637152
attr(, "class"):
[1] "Matrix"
```

Subscripting Matrices

For the most part, you subscript Matrices just as you would standard matrices; use a subscript of the form [*i*,*j*], where *i* indexes the rows and *j* indexes the columns:

```
> A[1,2]
     [,1]
[1,]    2
attr(, "class"):
[1] "Matrix"
```

The difference from standard matrix subscripting is obvious from the output; the return value is a Matrix, even if the result could be simplified by dropping the `dim` attribute.

```
> A[1,]
     [,1] [,2] [,3]
[1,]   19    2   15
attr(, "class"):
[1] "Matrix"
> A[,2]
     [,1]
[1,]    2
[2,]   18
[3,]   17
attr(, "class"):
[1] "Matrix"
```

In standard S-PLUS, each of these subscripting examples would, by default, return a vector with no matrix character whatsoever. The matrix character could be retained by using the `drop=F` argument. In Matrix subscripting, `drop=F` is the default, and `drop=T` is not allowed:

```
> A[,2, drop=T]
Error in "[.Matrix"(x, , 2, drop = drop):
        drop = T not allowed
Dumped
```

If both subscripts are omitted, the entire Matrix is returned:

```
> A[]
     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
attr(, "class"):
[1] "Matrix"
```

Standard S-PLUS matrix subscripting allows arbitrary numeric and complex subscripts; fractional subscripts are truncated to integer, while complex subscripts have their imaginary parts ignored and fractional real parts truncated to integer. The `Matrix` library forbids such non-integer subscripts:

```
> A[c(1.74, 2.26),]
Error in "[.Matrix"(A, c(1.74, 2.26),  ):
        non-integer numeric row subscript
Dumped
> A[,c(1.74, 2.26)]
Error in "[.Matrix"(A,  , c(1.74, 2.26)):
        non-integer numeric column subscript
Dumped
> A[1+2.3i,]
Error in "[.Matrix"(A, 1+2.3i,  ): row subscript must have
        numeric, logical or character mode
Dumped
```

Character string subscripts for Matrices work much the same as the standard matrix operations:

```
> dimnames(A) <- list(c("Sun","Mon","Tue"),
+ c("Apr","May", "Jun"))
> A[,"Apr"]
    Apr
Sun  19
Mon   8
Tue  11
attr(, "class"):
[1] "Matrix"
> A["Mon",]
    Apr May Jun
Mon   8  18  19
attr(, "class"):
[1] "Matrix"
```

Logical subscripts must either be vectors of length `nrow` or `ncol`, selecting rows or columns respectively, a matrix of the same dimension as the original Matrix, or a vector with the same length as the original Matrix:

```
> A[c(T,F,T),]
    Apr May Jun
Sun  19   2  15
Tue  11  17  10
attr(, "class"):
[1] "Matrix"
> A[c(T,F),]
Error in "[.Matrix"(A, c(T, F),  ): logical row subscript
        length must equal matrix row dimension
Dumped
> A[, c(F,T,T)]
    May Jun
Sun   2  15
Mon  18  19
Tue  17  10
attr(, "class"):
[1] "Matrix"
> A[A > 10]
[1] 19 11 18 17 15 19
> A[sample(c(T,F), size=9, replace=T)]
[1]  8 11  2 18 17 19 10
```

Standard S-PLUS matrix subscripting permits short logical subscripts, which are then replicated to the appropriate length. This replication is often confusing, and in algebraic applications usually not desired. The Matrix library expressly forbids such short subscripts, as the second row subscript example above demonstrates.

The Matrix library does support the irregular subscripting performed by a two column matrix, in which each row represents the row and column of a value to be extracted. In standard S-PLUS, the extraction matrix must consist of numeric values, but for Matrices, a character matrix using the dimnames of the Matrix is acceptable:

```
> nummat <- Matrix(c(1,1,2,2,3,3), ncol=2, byrow=T)
> A[nummat]
[1] 19 18 10
> submat <- Matrix(c("Sun", "Apr", "Mon", "May",
+ "Tue", "Jun"), ncol=2, byrow=T)
```

```
> A[submat]
[1] 19 18 10
> statemat <- Matrix(c("California", "Murder",
+ "Wyoming", "Frost"), ncol=2, byrow=T)
> state.x77[statemat]
 California Wyoming Murder Frost
         NA      NA     NA    NA
> as.Matrix(state.x77)[statemat]
[1]  10.3 173.0
```

Creating Specialized Matrices

In addition to the general `"Matrix"` class, the Matrix library supports a variety of subclasses for Matrices with specialized structures, such as identity and diagonal Matrices, upper and lower triangular Matrices, and Hermitian and orthogonal Matrices. Constructor functions exist for identity and diagonal Matrices, but in most cases you build these specialized Matrices in two steps—first, construct the Matrix using the `Matrix` function, then assign its class using the `Matrix.class` function. `Matrix.class` performs a variety of tests on the Matrix to determine its specialized structure, and returns an appropriate vector of subclasses.

Creating Identity Matrices

Make an identity Matrix of any (square) dimension with the `Identity` function. Identity matrices formed in this way are stored as a single number, the length of the diagonal:

```
> Id.4 <- Identity(4)
> Id.4
[1] 4
attr(, "class"):
[1] "Identity" "Matrix"
```

Use the `unpack` function to display the Matrix in "natural" form:

```
> unpack(Id.4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    1    0    0
[3,]    0    0    1    0
[4,]    0    0    0    1
attr(, "class"):
[1] "UnitLowerTriangular" "UnitUpperTriangular"
[3] "Lower Triangular"    "Upper Triangular"
[5] "Hermitian"           "Orthonormal"
[7] "Matrix"
```

Note that the "unpacked" form of the identity Matrix no longer inherits from class `"Identity"`, although it belongs to several other subclasses.

Creating Diagonal Matrices

Diagonal Matrices can be created and stored as the vector of diagonal values using the `Diagonal` function:

```
> D4 <- Diagonal(1:4)
> D4
[1] 1 2 3 4
attr(, "class"):
[1] "Diagonal" "Matrix"
> unpack(D4)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
attr(, "class"):
[1] "LowerTriangular" "UpperTriangular" "Hermitian"
[4] "RowOrthogonal"   "ColOrthogonal"   "Matrix"
```

As with identity Matrices, unpacking a diagonal Matrix causes the Matrix to lose its inheritance from the `"Diagonal"` class, but gain inheritance from several other classes.

You can also create rectangular diagonal Matrices by specifying the dimensions desired. One of these dimensions must match the length of the vector of values:

```
> D5 <- Diagonal(1:4, c(5,4))
> unpack(D5)
     [,1] [,2] [,3] [,4]
[1,]    1    0    0    0
[2,]    0    2    0    0
[3,]    0    0    3    0
[4,]    0    0    0    4
[5,]    0    0    0    0
attr(, "class"):
[1] "RowOrthogonal" "ColOrthogonal" "Matrix"
> D6 <- Diagonal(1:4, c(4,6))
> unpack(D6)
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    0    0    0    0    0
[2,]    0    2    0    0    0    0
[3,]    0    0    3    0    0    0
[4,]    0    0    0    4    0    0
```

```
attr(, "class"):
[1] "RowOrthogonal" "ColOrthogonal" "Matrix"
```

You can, of course, use Identity and Diagonal matrices without unpacking them:

```
> xx <- Matrix(1:16, nrow=4)
> xx %*% D4
     [,1] [,2] [,3] [,4]
[1,]    1   10   27   52
[2,]    2   12   30   56
[3,]    3   14   33   60
[4,]    4   16   36   64
attr(, "class"):
[1] "Matrix"
```

Creating Symmetric and Hermitian Matrices

A matrix is *Hermitian* if and only if each element a_{ij} is equal to the complex conjugate of the element a_{ji}; that is, if the Matrix is equal to its conjugate transpose.

```
> my.Herm<-Matrix( c(1, 2+3i, 3-4i, 2-3i, 3,
+ 4-2i, 3+4i, 4+2i, 2), nrow=3)
> my.Herm
     [,1] [,2] [,3]
[1,] 1+0i 2-3i 3+4i
[2,] 2+3i 3+0i 4+2i
[3,] 3-4i 4-2i 2+0i
attr(, "class"):
[1] "Matrix"
```

There is no constructor function for Hermitian matrices. Instead, use the function `Matrix.class` to assign the appropriate subclasses to a Matrix. `Matrix.class` tests its argument and returns a vector of subclasses to which the Matrix belongs:

```
> Matrix.class(my.Herm)
[1] "Hermitian" "Matrix"
> class(my.Herm) <- Matrix.class(my.Herm)
```

All symmetric *real* matrices are Hermitian:

```
> Sym <- Matrix( c(4, -3, 5, -3, 2, 1, 5, 1, -6), nrow=3)
> class(Sym) <- Matrix.class(Sym)
> Sym
```

```
     [,1] [,2] [,3]
[1,]    4   -3    5
[2,]   -3    2    1
[3,]    5    1   -6
attr(, "class"):
[1] "Hermitian" "Matrix"
```

In the rest of this chapter, we will use the term "Hermitian" whenever we mean a complex Hermitian or real symmetric matrix.

Creating Orthonormal Matrices

An *orthonormal* Matrix is a Matrix that has the following two properties:

1. the transpose of the Matrix is equal to its inverse.
2. all rows and columns are unit vectors (have norm 1 for vector 2-norm).

Orthonormal Matrices are easy to generate in S-PLUS using the `qr` function, which performs the *QR* decomposition of a matrix into an orthonormal matrix *Q* and an upper triangular (or trapezoidal) matrix *R*. See page 810, The QR Decomposition, for complete details.

Creating Triangular Matrices

A *triangular* Matrix is one in which all entries are zero either below (upper triangular) or above (lower triangular) the diagonal. You can easily convert any S-PLUS Matrix into a triangular Matrix, simply by "zeroing out" the appropriate entries. For example, to convert our Matrix `A` into lower triangular form, we can replace the upper diagonal entries with 0's as follows:

```
> A.tri <- A
> A.tri[row(A.tri)< col(A.tri)] <- 0
> class(A.tri) <- Matrix.class(A.tri)
> A.tri
    Apr May Jun
Sun  19   0   0
Mon   8  18   0
Tue  11  17  10
attr(, "class"):
[1] "LowerTriangular" "Matrix"
```

Further, once you've created a lower (upper) triangular Matrix, its transpose is an upper (lower) triangular Matrix.

Creating Permutation Matrices

A *permutation* Matrix is an identity Matrix with one or more rows or columns permuted. For example, the following Matrix is an identity Matrix with the first and third rows permuted:

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    0    0
[3,]    1    0    0    0
[4,]    0    0    0    1
attr(, "class"):
[1] "Orthonormal" "Matrix"
```

The Matrix library contains two functions for generating permutation Matrices: `RowPermutation` generates row permutations, while `ColPermutation` generates column permutations. Both functions take a single argument, a permutation of the integers 1 to *n*. Thus, the Matrix above can be generated using either of the two functions as follows:

```
> unpack(RowPermutation(c(3,2,1,4)))
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    0    0
[3,]    1    0    0    0
[4,]    0    0    0    1
attr(, "class"):
[1] "Orthonormal" "Matrix"
> unpack(ColPermutation(c(3,2,1,4)))
     [,1] [,2] [,3] [,4]
[1,]    0    0    1    0
[2,]    0    1    0    0
[3,]    1    0    0    0
[4,]    0    0    0    1

attr(, "class"):
[1] "Orthonormal" "Matrix"
```

The compact form returned by the two functions does differ, however:

```
> RowPermutation(c(3,2,1,4))
[1] 3 2 1 4
attr(, "class"):
[1] "RowPermutation" "Matrix"
> ColPermutation(c(3,2,1,4))
[1] 3 2 1 4
attr(, "class"):
[1] "ColPermutation" "Matrix"
```

Matrix Norms

A *Matrix norm* is a measure of the size of a Matrix (or, more accurately, a measure of distance in the space of Matrices). There are several commonly used Matrix norms:

- Frobenius norm:

$$\|A\|_F = \sqrt{\sum_{i=1}^{m}\sum_{j=1}^{n}|a_{ij}|^2}$$

- p-norms:

$$\|A\|_p = \sup_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p}$$

  where $|x|_p$ is the vector p-norm. Three p-norms (1, 2, and ∞) are widely used, and can be computed in S-PLUS. They can be characterized as follows:

  - $p = 1$: The maximum sum of magnitudes of elements in each *column* of the Matrix.

$$\|A\|_1 = \max_{1 \leq j \leq n} \sum_{i=1}^{m}|a_{ij}|$$

  - $p = \infty$: The maximum sum of magnitudes of elements in each *row/* of the Matrix.

$$\|A\|_\infty = \max_{1 \leq i \leq m} \sum_{j=1}^{n}|a_{ij}|$$

  - $p = 1$: The largest singular value of the Matrix

- Maximum-modulus norm:.

$$\|A\|_\Delta = \max|a_{ij}|$$

You calculate Matrix norms using the Matrix library's `norm` function. For the 1-norm, ∞-norm, Frobenius norm, and maximum-modulus norm, you call

norm by specifying the Matrix and the type of norm (maximum-modulus is the default):

```
> norm(A)
[1] 19
> norm(A, type="Frobenius")
[1] 43
> norm(A, type="1")
[1] 44
> norm(A, type="Inf")
[1] 45
```

Only the first letter of the type string is needed (or used):

```
> norm(A, "F")
[1] 43
```

To compute the 2-norm, you must first compute the singular value decomposition (SVD) or the eigen decomposition (for Hermitian matrices):

```
> norm(svd(A, vectors=F))
[1] 40.00011
```

See page 799, The Singular Value Decomposition, for details on the SVD and the svd function; see page 807, The Eigen Decomposition, for details on the eigen decomposition and the eigen function.

Condition Estimates

For a square Matrix A, the *condition number* $\kappa(A)$ is defined as follows:

$$\kappa(A) = \|A\|\|A^{-1}\|$$

For singular A, $\kappa(A) = \infty$. The exact value of the condition number is norm-dependent. The condition number can be thought of as a measure of the closeness of a square Matrix to singularity. It falls in the range $[1, \infty)$, where the value ∞ implies singularity. Matrices with large condition numbers are said to be *ill-conditioned*. Because the reciprocal of the condition number is a bounded quantity, falling in the interval [0,1], S-PLUS computes the reciprocal, rather than the condition number itself. In most cases, the computed result is an estimate of the reciprocal condition number rather than a direct computation; the estimate is in any case at least as large as the actual condition number.

To obtain the reciprocal condition estimate for a Matrix, use the rcond function. By default, rcond gives the one-norm condition estimate, although the infinity norm is also available:

```
> rcond(A)
[1] 0.1125994
> rcond(A, one.norm=F)
[1] 0.0707946
```

As with Matrix norms, 2-norm condition numbers can be obtained by first taking the singular value decomposition of the Matrix (or the eigenvalue decomposition of a Hermitian Matrix):

```
> rcond(svd(A))
[1] 0.1442148
```

For a rectangular matrix, the notion of condition number can be defined by replacing the inverse of the matrix in the original definition with the *pseudo-inverse*, which is the unique minimal (in Frobenius norm) solution to the following problem:

$$min_{X \in R^{n \times m}} \|AX - I_m\|_F$$

For rectangular matrices, the reciprocal condition estimate is based on the *QR* decomposition (see page 810, The Eigen Decomposition, for a complete description of the *QR* decomposition):

```
> rect.Mat <- Matrix(sample(-9:9, size=12, replace=T),
+ nrow=4, ncol=3)
> rect.Mat
     [,1] [,2] [,3]
[1,]    2   -2    2
[2,]    3    6    0
[3,]    3   -5    5
[4,]   -6   -4    5
attr(, "class"):
[1] "Matrix"
> rcond(rect.Mat)
[1] 0.2722501
> rcond(rect.Mat, one.norm=F)
[1] 0.2123998
```

Determinants

The *determinant* of a 1×1 Matrix $A = (a_{11})$ is simply a_{11}. For an $n \times n$ Matrix, the determinant is defined in terms of the determinants of $(n - 1) \times (n - 1)$ Matrices, as follows. If $A \in R^{n \times n}$,

$$det(A) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} det(A_{1j})$$

where A_{1j} is the $(n - 1) \times (n - 1)$ Matrix obtained by deleting the first row and jth column of A. See Golub and Van Loan (1989) for further details.

Determinants in S-PLUS are computed using the `det` function, which returns the determinant as a list containing by default the logarithm of the modulus of the determinant and the sign of the determinant. The argument `logarithm=F` tells S-PLUS to return the modulus of the determinant instead of its logarithm:

```
> det(A)
$modulus:
[1] 8.12829
attr($modulus, "logarithm"):
[1] T

$sign:
[1] -1
> det(A, log=F)
$modulus:
[1] 3389
attr($modulus, "logarithm"):
[1] F

$sign:
[1] -1
```

Special methods for various types of Matrices, such as QR and SVD decompositions, take advantage of computational efficiencies. In some cases, however, sign information is lost:

```
> det(svd(A))
$modulus:
[1] 8.12829
attr($modulus, "logarithm"):
[1] T
$sign:
[1] NA
```

```
> det(eigen(A))
$modulus:
[1] 8.12829
attr($modulus, "logarithm"):
[1] T

$sign:
[1] -1
> det(qr(A))
$modulus:
[1] 8.12829
attr($modulus, "logarithm"):
[1] T

$sign:
[1] NA
```

The following function, `numdet`, always returns a number (numeric or complex):

```
> numdet
function(det){
        if(attributes(det$modulus)$logarithm)
                val <- exp(det$modulus)
        else val <- det$modulus
        if(!is.na(det$sign))
                val <- val * det$sign
        else warning("Sign information not available")
        val
```

29.3 MATRIX DECOMPOSITIONS

Standard S-PLUS has long had a variety of matrix decomposition functions; these are used internally by the various S-PLUS regression functions, and have wide applicability. The `Matrix` library includes additional decomposition functions, with many specific methods designed to take advantage of specialized Matrix structures. The following decompositions are available in the `Matrix` library:

- *Singular value decomposition*.
- *LU decomposition*, and the closely related *symmetric indefinite decomposition*.
- *Eigen decomposition*.

- *QR decomposition*.
- *Schur decomposition*.

The *Choleski* decomposition is available in standard S-PLUS, but is not part of the `Matrix` library. However, for Matrices which satisfy the requirements for the Choleski decomposition, the symmetric indefinite decomposition provides all the components necessary to compute the Choleski decomposition explicitly. See page 803, The Hermitian Indefinite Decomposition, for details.

This section describes the available decompositions and the functions for computing them in the `Matrix` library.

The Singular Value Decomposition

For any real $m \times n$ matrix A, there exist orthogonal matrices U and V and a diagonal matrix Σ so that

$$U^T A V = \Sigma.$$

The $p = \min(m,n)$ diagonal elements $\sigma_1 \geq \sigma_1 \geq \ldots \geq \sigma_p \geq 0$ are called the *singular values* of A. The both the 2-norm and the Frobenius norm can be characterized readily in terms of the singular values:

$$\|A\|_F = \sum_{i=1}^{p} \sigma_i^2 \qquad \|A\|_2 = \sigma_1$$

To obtain the singular value decomposition, use the `svd` function:

```
> svd(A)
$values:
[1] 40.000114 14.687207  5.768609

$vectors:
$vectors$left:
           [,1]       [,2]        [,3]
[1,] -0.5200456  0.8538399 -0.02258456
[2,] -0.6606048 -0.4188323 -0.62304157
[3,] -0.5414369 -0.3090905  0.78186261
attr($vectors$left, "class"):
[1] "Orthonormal" "Matrix"

$vectors$right:
           [,1]       [,2]       [,3]
[1,] -0.5280363  0.6449356  0.5524814
[2,] -0.5533835 -0.7547957  0.3522074
[3,] -0.6441618  0.1197558 -0.7554563
```

```
attr($vectors$right, "class"):
[1] "Orthonormal" "Matrix"

attr(, "class"):
[1] "svd.Matrix" "decomp"

other attributes:
[1] "call"       "complex"    "dimensions"
[4] "dimlabels"  "workspace"
```

The `svd` function returns a list with two components, `values` and `vectors`; the `vectors` component is also a list with two components, `left` containing the orthogonal Matrix *U*, `right` containing the orthogonal Matrix *V*. You can verify the decomposition as follows:

```
> A.svd <- svd(A)
> round(t(A.svd$vectors$left) %*% A %*%
+ A.svd$vectors$right, digits=3)
     [,1]   [,2]   [,3]
[1,]   40  0.000  0.000
[2,]    0 14.687  0.000
[3,]    0  0.000  5.769
attr(, "class"):
[1] "Matrix"
> round(A.svd$values, digits=3)
[1] 40.000 14.687  5.769
```

Once you obtain the SVD, you can easily obtain the 2-norm and the 2-norm reciprocal condition number for the original Matrix.

```
> norm(A.svd)
[1] 40.00011
> rcond(A.svd)
[1] 0.1442148
```

The SVD also provides for efficient calculation of the determinant, although sign information is lost:

```
> det(A.svd)
$modulus:
[1] 8.12829
attr($modulus, "logarithm"):
[1] T

$sign:
[1] NA
```

The number of positive singular values also gives a useful measure of the *rank* of the Matrix:

```
> Matrix.rank <- function(Matrix){
+ length(svd(Matrix)$values)}
> Matrix.rank(A)
[1] 3
> Matrix.rank(rect.Mat)
[1] 3
```

The LU Decomposition

If X is a square matrix, then there is a row permutation P, a lower triangular matrix L with 1's on its diagonal, and an upper triangular matrix U such that

$$PX = LU$$

For rectangular matrices, a similar decomposition exists, except that either L or U is trapezoidal, depending on whether the matrix has more or less rows than columns. This decomposition is called the LU decomposition.

To obtain the LU decomposition, use the `lu` function:

```
> lu(A)
$factors:
           Apr        May       Jun
Sun 19.0000000  2.0000000  15.00000
Mon  0.4210526 17.1578947  12.68421
Tue  0.5789474  0.9233129 -10.39571

$pivot:
[1] 1 2 3

attr(, "class"):
[1] "lu.Matrix" "decomp"

other attributes:
[1] "call"      "dimlabels" "norm"
```

The `lu` function returns a list with two components, `factors` and `pivot`. The `factors` component is a compact representation of both L and U, taking advantage of the fact that L is known to have 1's along its diagonal. The `pivot` component is the row permutation P, expressed as a numeric vector.

To obtain L, U, and P explicitly, use the `expand` function:

```
> expand(lu(A))
$l:
          Apr       May Jun
Sun 1.0000000 0.0000000   0
Mon 0.4210526 1.0000000   0
Tue 0.5789474 0.9233129   1
attr($l, "class"):
[1] "UnitLowerTriangular" "LowerTriangular"   "Matrix"

$u:
    Apr      May       Jun
Sun  19  2.00000  15.00000
Mon   0 17.15789  12.68421
Tue   0  0.00000 -10.39571
attr($u, "class"):
[1] "UpperTriangular" "Matrix"

$permutation:
[1] 3
attr($permutation, "class"):
[1] "Identity" "Matrix"

attr(, "class"):
[1] "expand.lu.Matrix"
```

If you want to multiply one of the factors by some other Matrix, but don't need the remainder of the decomposition, use the `facmul` function to perform the multiplication. For example, to multiply the factor `"L"` by the original Matrix `A`, use `facmul` as follows:

```
> facmul(lu(A), "L", y=A)
        [,1]     [,2]     [,3]
[1,] 19.0000  2.00000 15.00000
[2,] 16.0000 18.84211 25.31579
[3,] 29.3865 34.77753 36.22716
attr(, "class"):
[1] "Matrix"
```

Using `facmul` without the `y` argument gives a convenient method for extracting a single factor:

```
> facmul(lu(A), "L")
          Apr       May Jun
Sun 1.0000000 0.0000000   0
```

```
Mon 0.4210526 1.0000000   0
Tue 0.5789474 0.9233129   1
attr(, "class"):
[1] "UnitLowerTriangular" "LowerTriangular"   "Matrix"
> facmul(lu(A), "U")
    Apr      May       Jun
Sun  19  2.00000  15.00000
Mon   0 17.15789  12.68421
Tue   0  0.00000 -10.39571
attr(, "class"):
[1] "UpperTriangular" "Matrix"
> facmul(lu(A), "P")
[1] 3
attr(, "class"):
[1] "Identity" "Matrix"
```

By default, `lu` computes the 1-norm and ∞-norm of the Matrix, and stores these as attributes:

```
> attributes(lu(A))$norm
 one infinity
  44       45
```

These norms should be computed if `solve` will eventually be applied to the factorization with condition estimation. The infinity norm is needed for solves involving the underlying matrix, and the one norm is needed for solves involving its transpose. One or both of the norms can be omitted from the computation by specifying appropriate logical values in the `norm.comp` argument to `lu`:

```
> lu.A <-  lu(A, norm.comp=c(F,T))
> attributes(lu.A)$norm
 infinity
       45
```

The Hermitian Indefinite Decomposition

If X is a Hermitian matrix, then there is a permutation P, a triangular matrix T with diagonal elements all equal to one, and a Hermitian block diagonal matrix B with either 1×1 or 2×2 blocks, such that

$$PXP^T = TBT^H$$

This is called the Hermitian Indefinite Decomposition. If X is positive (semi-) definite, the blocks are 1×1, real, and positive (non-negative), in which case the decomposition reduces essentially to the Choleski decomposition.

To obtain the Hermitian Indefinite Decomposition, use the `lu` function:

```
> lu(my.Herm, lower=F)
$factors:
           [,1]                [,2]                  [,3]
[1,] 4.130435+0i 0.6956522-0.6956522i -0.4347826+0.6521739i
[2,] 2.000000+3i 1.0000000+0.0000000i  3.0000000+4.0000000i
[3,] 3.000000-4i 4.0000000-2.0000000i  2.0000000+0.0000000i
attr($factors, "uplo"):
[1] "U"

$pivot:
[1]  1 -1 -1

attr(, "class"):
[1] "lu.Hermitian" "decomp"

other attributes:
[1] "call"      "norm"      "workspace"
```

You can obtain the explicit matrices *P*, *T*, and *B* using `expand` or `facmul` as before for *L* and *U*:

```
> expand(lu(my.Herm, lower=F))
$triangular:
     [,1]                [,2]                  [,3]
[1,] 1+0i 0.6956522-0.6956522i -0.4347826+0.6521739i
[2,] 0+0i 1.0000000+0.0000000i  0.0000000+0.0000000i
[3,] 0+0i 0.0000000+0.0000000i  1.0000000+0.0000000i
attr($triangular, "class"):
[1] "UnitUpperTriangular" "UpperTriangular"   "Matrix"

other attributes:
[1] "uplo"

$block.diagonal:
           [,1] [,2] [,3]
[1,] 4.130435+0i 0+0i 0+0i
[2,] 0.000000+0i 1+0i 3+4i
[3,] 0.000000+0i 3-4i 2+0i
attr($block.diagonal, "class"):
[1] "Hermitian" "Matrix"
```

```
other attributes:
[1] "uplo"

$permutation:
[1] 2 1 3
attr($permutation, "class"):
[1] "RowPermutation" "Matrix"

attr(, "class"):
[1] "expand.lu.Hermitian"
> facmul(lu(my.Herm), "P")
[1] 2 1 3
attr(, "class"):
[1] "RowPermutation" "Matrix"
> facmul(lu(my.Herm), "T")
                      [,1]                   [,2] [,3]
[1,] 1.0000000+0.0000000i  0.0000000+0.0000000i 0+0i
[2,] 0.0000000+0.0000000i  1.0000000+0.0000000i 0+0i
[3,] 0.6956522-0.6956522i -0.4347826+0.6521739i 1+0i
attr(, "class"):
[1] "UnitLowerTriangular" "LowerTriangular"   "Matrix"

other attributes:
[1] "uplo"
> facmul(lu(my.Herm), "B")
     [,1] [,2]        [,3]
[1,] 1+0i 3+4i 0.000000+0i
[2,] 3-4i 2+0i 0.000000+0i
[3,] 0+0i 0+0i 4.130435+0i
attr(, "class"):
[1] "Hermitian" "Matrix"

other attributes:
[1] "uplo"
```

In the positive definite case, B is diagonal, P is the identity matrix, and the indefinite Hermitian decomposition reduces (via the transformation $G = T\sqrt{B}$) to the Choleski decomposition, which decomposes X into the product of an upper triangular matrix G and its conjugate transpose $X = GG^H$:

```
> posdef  <- Matrix(sample(-1:1, size=9, replace=T),
+ nrow=3, ncol=3)
> posdef <- posdef %*% t(posdef)
> class(posdef) <- Matrix.class(posdef)
> posdef
     [,1] [,2] [,3]
[1,]    3    2    0
[2,]    2    2    1
[3,]    0    1    2
attr(, "class"):
[1] "Hermitian" "Matrix"
> posdef.g <- facmul(lu(posdef), "T") %*%
+ sqrt(facmul(lu(posdef), "B"))
> posdef.g %*% t(posdef.g)
     [,1] [,2] [,3]
[1,]    3    2    0
[2,]    2    2    1
[3,]    0    1    2
attr(, "class"):
[1] "Matrix"
```

You can use `lu` and `facmul` to define a Choleski function to take the Choleski decomposition directly:

```
> Choleski
function(x)
{
        if(!inherits(x, "Matrix"))
                x <- as.Matrix(x)
        class(x) <- Matrix.class(x)
        if(!inherits(x, "Hermitian"))
                stop("x must be a Hermitian matrix")
        val <- facmul(lu(x, lower=F), "T") %*%
                        sqrt(facmul(lu(x, lower=F), "B"))
        class(val) <- Matrix.class(val)
        val
}
```

We can try it out on our simple positive definite matrix:

```
> posdef
     [,1] [,2] [,3]
[1,]    3    2    0
[2,]    2    2    1
[3,]    0    1    2
```

```
attr(, "class"):
[1] "Hermitian" "Matrix"
> Choleski(posdef) %*% t(Choleski(posdef))
     [,1] [,2] [,3]
[1,]    3    2    0
[2,]    2    2    1
[3,]    0    1    2
attr(, "class"):
[1] "Matrix"
```

The Eigen Decomposition

For any $n \times n$ Matrix X, there are scalar values λ_i and vectors v_i and u_i, $i = 1, \ldots, n$, for which

$$Xv_i = \lambda_i v_i u_i^H X = \lambda_i u_i^H$$

The λ_i are called the eigenvalues of X, while the vectors v_i and u_i are called, respectively, the right and left eigenvectors of X. To compute eigenvalues and eigenvectors in S-PLUS, use the `eigen` function:

```
> eigen(A)
$values:
[1]  39.581985  13.677784  -6.259769

$vectors:
$vectors$left:
          [,1]        [,2]       [,3]
[1,] 0.5499003  0.78919610 -0.1781937
[2,] 0.5476794 -0.61095405 -0.5547945
[3,] 0.6306005  0.06248726  0.8126808
attr($vectors$left, "class"):
[1] "Matrix"

$vectors$right:
          [,1]       [,2]       [,3]
[1,] 0.4768760  0.7998527 -0.4235897
[2,] 0.6737382 -0.5627172 -0.4678325
[3,] 0.5645052 -0.2087703  0.7756961
attr($vectors$right, "class"):
[1] "Matrix"

attr(, "class"):
[1] "eigen.Matrix" "decomp"
```

```
 other attributes:
[1] "call"      "dimlabels" "one.norm"  "workspace"
```

If X has any complex eigenvalues, some of its eigenvectors come in conjugate pairs; in this case the vectors component contains the real and imaginary parts of the eigenvectors. To extract the true eigenvectors, you need to use the expand function:

```
> eigen(B)
$values:
[1] 36.755743+0.00000i  1.622129+6.49027i  1.622129-6.49027i

$vectors:
$vectors$left:
           [,1]       [,2]       [,3]
[1,] -0.5805813 -0.4189235 -0.3008385
[2,] -0.5661857 -0.2525614  0.4955897
[3,] -0.5851146  0.6516156  0.0000000
attr($vectors$left, "class"):
[1] "Matrix"

$vectors$right:
           [,1]       [,2]       [,3]
[1,] -0.6836358 -0.4208512  0.4218487
[2,] -0.4149883  0.6514816  0.0000000
[3,] -0.6003556 -0.2128151 -0.4185803
attr($vectors$right, "class"):
[1] "Matrix"


attr(, "class"):
[1] "eigen.Matrix" "decomp"

other attributes:
[1] "call"      "one.norm"  "workspace"

> expand(eigen(B))
$values:
[1] 36.755743+0.00000i  1.622129+6.49027i  1.622129-6.49027i


$vectors:
$vectors$left:
              [,1]                  [,2]                  [,3]
[1,] -0.5805813+0i -0.4189235-0.3008385i -0.4189235+0.3008385i
[2,] -0.5661857+0i -0.2525614+0.4955897i -0.2525614-0.4955897i
```

```
[3,] -0.5851146+0i  0.6516156+0.0000000i  0.6516156+0.0000000i
attr($vectors$left, "class"):
[1] "Matrix"

$vectors$right:
             [,1]                  [,2]                  [,3]
[1,] -0.6836358+0i -0.4208512+0.4218487i -0.4208512-0.4218487i
[2,] -0.4149883+0i  0.6514816+0.0000000i  0.6514816+0.0000000i
[3,] -0.6003556+0i -0.2128151-0.4185803i -0.2128151+0.4185803i
attr($vectors$right, "class"):
[1] "Matrix"


attr(, "class"):
[1] "expand.eigen.Matrix" "decomp"

other attributes:
[1] "call"      "one.norm"  "workspace"
```

When X is Hermitian, the left and right eigenvectors are the same, and can be written as the columns of a unitary matrix Z. Taking $\Lambda = \mathrm{diag}(\lambda_1, \ldots, \lambda_n)$, we have $X = Z\Lambda Z^H$.

```
> eigen(my.Herm)
$values:
[1] -5.550724  1.745483  9.805241

$vectors:
                      [,1]                  [,2]                 [,3]
[1,]  0.1678486+0.60792927i -0.3575879+0.4547576i 0.4516168+0.2522251i
[2,]  0.4638801-0.05615832i  0.1054296-0.6459414i 0.3834491+0.4541725i
[3,] -0.6196050+0.00000000i  0.4867964+0.0000000i 0.6157263+0.0000000i
attr($vectors, "class"):
[1] "Orthonormal" "Matrix"

attr(, "class"):
[1] "eigen.Hermitian" "decomp"

other attributes:
[1] "call"      "uplo"      "workspace"
```

The eigen decomposition can sometimes be simplified by *balancing* the Matrix before computing the decomposition. There are two operations that may be performed during balancing: row and column permutations to make the Matrix more nearly upper triangular, and diagonal scaling to make the rows and columns more nearly equal in norm. You can specify neither, one, or both of the balancing operations, using the `balance` argument to `eigen`.

The default is no balancing; the following call to `eigen` uses permutation balancing:

```
> eigen(A, balance=c(T,F))
```

See Golub and Van Loan (1989) and the *LAPACK User's Manual* (1994) for further details on balancing.

The QR Decomposition

If X is an $m \times n$ matrix, then there is an $m \times m$ unitary matrix Q and an upper triangular matrix R such that

$$X = Q\begin{bmatrix} R & S \\ 0 & 0 \end{bmatrix}$$

This is called the *QR* decomposition.

To obtain the *QR* decomposition in S-PLUS, use the `qr` function:

```
> qr(rect.Mat)
$factors:
$factors[[1]]:


           [,1]       [,2]       [,3]
[1,] -7.6157731 -3.0200480  1.4443708
[2,]  0.3119874 -8.4781667  5.2650452
[3,]  0.3119874 -0.3755841 -4.9186474
[4,] -0.6239748 -0.2375377  0.5264209

$factors[[2]]:
[1] 1.262613 1.670164 1.566025


$pivot:
NULL

attr(, "class"):
[1] "qr.Matrix" "decomp"

other attributes:
[1] "call"      "workspace"
```

As with the *LU* decomposition, the explicit factors Q and R can be computed using the `expand` and `facmul` functions:

```
> expand(qr(rect.Mat))
$q:
           [,1]       [,2]       [,3]        [,4]
[1,] -0.2626129  0.3294466 -0.1310846  0.89739413
[2,] -0.3939193 -0.5673803 -0.7230135 -0.01259501
[3,] -0.3939193  0.7300700 -0.3507293 -0.43452768
[4,]  0.7878386  0.1911604 -0.5805663  0.07557003
attr($q, "class"):
[1] "Matrix"
$r:
          [,1]      [,2]      [,3]
[1,] -7.615773 -3.020048  1.444371
[2,]  0.000000 -8.478167  5.265045
[3,]  0.000000  0.000000 -4.918647
[4,]  0.000000  0.000000  0.000000
attr($r, "class"):
[1] "Matrix"

$permutation:
[1] 3
attr($permutation, "class"):
[1] "Identity" "Matrix"

attr(, "class"):
[1] "expand.qr.Matrix"
> facmul(qr(rect.Mat), "R")
          [,1]      [,2]      [,3]
[1,] -7.615773 -3.020048  1.444371
[2,]  0.000000 -8.478167  5.265045
[3,]  0.000000  0.000000 -4.918647
[4,]  0.000000  0.000000  0.000000
attr(, "class"):
[1] "Matrix"
> facmul(qr(rect.Mat), "Q")
           [,1]       [,2]       [,3]        [,4]
[1,] -0.2626129  0.3294466 -0.1310846  0.89739413
[2,] -0.3939193 -0.5673803 -0.7230135 -0.01259501
[3,] -0.3939193  0.7300700 -0.3507293 -0.43452768
[4,]  0.7878386  0.1911604 -0.5805663  0.07557003
attr(, "class"):
[1] "Matrix"
```

The *QR* decomposition is a useful source of both orthonormal and lower triangular Matrices. For example, here we obtain both an orthonormal Matrix `A.q` and an upper triangular Matrix `A.u` from the expansion of the *QR* decomposition of `A`:

```
> A.qr <- qr(A)
> A.q <- expand(A.qr)$q
> A.u <- expand(A.qr)$r
> A.q
           [,1]       [,2]       [,3]
[1,] -0.8131249  0.5653998 -0.1383870
[2,] -0.3423684 -0.6568151 -0.6718465
[3,] -0.4707565 -0.4989158  0.7276478
attr(, "class"):
[1] "Matrix"
> A.u
          [,1]      [,2]       [,3]
[1,] -23.36664 -15.79174 -23.409439
[2,]   0.00000 -19.17344  -8.987649
[3,]   0.00000   0.00000  -7.564412
attr(, "class"):
[1] "UpperTriangular" "Matrix"
```

The Schur Decomposition

If X is a square matrix, then there is a unitary matrix Z and a matrix S such that

$$X = ZSZ^H$$

If X is real, S is *upper quasi-triangular*—nearly upper triangular with either 1×1 or 2×2 blocks on the diagonal. If X is complex, S is upper triangular. The eigenvalues of X appear on the diagonal of S; the 2×2 diagonal blocks in the real case correspond to the complex conjugate eigenvalues. This decomposition is called the *Schur decomposition*. An important property of the Schur decomposition is that Z can be chosen so that the eigenvalues of X appear in any order on the diagonal of S.

To obtain the Schur decomposition, use the `schur` function:

```
> schur(A)
$form:
         Apr       May       Jun
Sun 39.58198  3.013294  4.541067
Mon  0.00000 13.677784  5.128232
Tue  0.00000  0.000000 -6.259769
```

```
attr($form, "class"):
[1] "UpperTriangular" "Matrix"

$vectors:
          [,1]       [,2]       [,3]
[1,] 0.4768760  0.8607185 -0.1781937
[2,] 0.6737382 -0.4881393 -0.5547945
[3,] 0.5645052 -0.1445122  0.8126808
attr($vectors, "class"):
[1] "Orthonormal" "Matrix"

attr(, "class"):
[1] "schur.Matrix" "decomp"

other attributes:
[1] "call"       "dimlabels"   "eigenvalues" "workspace"
> eigen(A)$values
[1]  39.581985  13.677784  -6.259769
```

One useful application of the Schur decomposition is in the definition of Matrix functions. If $f(z)$ is a scalar function defined on the eigenvalues of a Matrix A, then you can informally define a Matrix function $f(A)$ by substituting "A" for "z" in the formula defining f, making suitable allowances between scalar operations and Matrix operations. For example, if $f(z) = z^2$, we can meaningfully define $f(A)$ as follows:

$$f(A) = A^2$$

where $A^2 = A \times A$, with "$\times$" taken to be Matrix multiplication. Unfortunately, such definitions don't take you very far computationally. However, if $A = QTQ^H$ is the Schur decomposition of A, then $f(A) = Qf(T)Q^H$

Thus, we only need to be able to calculate Matrix functions for triangular Matrices. The following S-PLUS function, `Matrix.fun`, implements an algorithm from Golub and Van Loan (1989) for doing precisely that—computing a Matrix function $F = f(T)$, where T is upper triangular. (A further requirement of the algorithm is that T have distinct eigenvalues; this implementation does not check for this.)

```
> Matrix.fun <- function(Tmat, FUN)
+ {
+   Fmat <- Tmat
+   diag(Fmat) <- diag(FUN(Tmat))
```

```
+   for(p in 1:(nrow(Tmat) - 1))
+       for(i in 1:(nrow(Tmat) - p)) {
+           j <- i + p
+           s <- Tmat[i, j] * (Fmat[j, j] - Fmat[i, i])
+           if((j - 1) >= (i + 1)) {
+               k <- (i + 1):(j - 1)
+               s <- s + Tmat[i, k] %*% Fmat[k, j] -
+                        Fmat[i, k] %*% Tmat[k, j]
+               }
+           Fmat[i, j] <- s/(Tmat[j, j] - Tmat[i, i])
+       }
+   Fmat
+ }
```

As a simple example, compare the Matrix function `f(A) = A^2` to simple matrix multiplication:

```
> small <- Matrix(c(1,0,1,2), ncol=2)
> small %*% small
     [,1] [,2]
[1,]    1    3
[2,]    0    4
attr(, "class"):
[1] "Matrix"
> Matrix.fun(small, function(x)x^2)
     [,1] [,2]
[1,]    1    3
[2,]    0    4
attr(, "class"):
[1] "Matrix"
```

For more complicated functions, or for matrices with eigenvalues that are nearly equal, the computations of matrix functions become more complicated. See Golub and Van Loan (1989) for a fuller description.

29.4 SOLVING SYSTEMS OF LINEAR EQUATIONS

One of the most widespread applications of linear algebra is in solving systems of equations of the form

$$AX = B$$

A related problem is finding the inverse (or pseudo-inverse) of a Matrix A. Both problems are solved in S-PLUS using the function `solve`, which now has a variety of methods which take advantage of specific Matrix structures. Most of these methods require A to be of full rank, although some (singular value and eigen) work with rank-deficient matrices.

Solving Square Linear Systems

Consider the following system of linear equations:

$$\begin{aligned} 19a + 2b + 15c &= 9 \\ 8a + 18b + 19c &= 5 \\ 11a + 17b + 10c &= 14 \end{aligned}$$

This is the familiar case of n equations in n unknowns, and can easily be solved by elementary linear algebra. The basic Matrix method for `solve` uses the LU decomposition to solve the system and estimate the condition number:

```
> A.solve(A, c(9,5,14))
          [,1]
Apr  0.9914429
May  0.6161109
Jun -0.7379758
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
> attr(A.solv, "rcond")
[1] 0.09639891
```

If the coefficient Matrix is upper or lower triangular, special solve methods exploit this structure:

```
> my.Upper <- Matrix(c(2,0,0,3,5,0,1,4,6),ncol=3)
> class(my.Upper) <- Matrix.class(my.Upper)
> my.Upper
     [,1] [,2] [,3]
[1,]    2    3    1
[2,]    0    5    4
[3,]    0    0    6
attr(, "class"):
[1] "UpperTriangular" "Matrix"
> solve(my.Upper, c(9,5,14))
           [,1]
[1,]  4.6333333
[2,] -0.8666667
[3,]  2.3333333
attr(, "class"):
[1] "Matrix"
```

```
other attributes:
[1] "rcond" "call"
> my.Lower <- t(my.Upper)
> solve(my.Lower, c(9,5,14))
          [,1]
[1,]  4.500000
[2,] -1.700000
[3,]  2.716667
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
```

Similarly, if the Matrix is symmetric or Hermitian, another solve method exploits that structure:

```
> my.sym3
            [,1]       [,2]        [,3]
[1,] -1.32119473  0.7576395  0.06296236
[2,]  0.75763953 -0.4710585  0.52317150
[3,]  0.06296236  0.5231715 -0.62392715
attr(, "class"):
[1] "Hermitian" "Matrix"
> solve(my.sym3, c(9,5,14))
         [,1]
[1,] 22.63464
[2,] 49.57063
[3,] 21.41127
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
```

In some cases, you may find it convenient to work with a matrix in factored form. You can solve square systems of full rank using either the LU or QR decomposition:

```
> A.lu <- lu(A)
> solve(A.lu, c(9,5,14))
```

```
          [,1]
Apr  0.9914429
May  0.6161109
Jun -0.7379758
attr(, "class"):
[1] "Matrix"
other attributes:
[1] "rcond" "call"

> A.qr <- qr(A)
> solve(A.qr, c(9,5,14))
          [,1]
Apr  0.9914429
May  0.6161109
Jun -0.7379758
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
```

In the Hermitian case, `lu` yields the Hermitian indefinite decomposition, which can also be used explicitly in `solve`:

```
> my.sym3.lu <- lu(my.sym3)
> solve(my.sym3.lu, c(9,5,14))
         [,1]
[1,] 22.63464
[2,] 49.57063
[3,] 21.41127
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond" "call"
```

Solving Over-determined Systems

In many applications, particularly data acquisition and control systems, there may be many more observations (equations) than unknowns. Such a system yields an overdetermined linear system. For example, consider the following five equations in three unknowns:

$$
\begin{aligned}
19a + 2b + 15c &= 9 \\
8a + 18b + 19c &= 5 \\
11a + 17b + 10c &= 14 \\
12a + 9b + 13c &= 11 \\
9a + 14b + 20c &= 8
\end{aligned}
$$

Such a system has a unique least-squares solution. The `solve.Matrix` function computes this solution using the QR decomposition:

```
> Aug <- Matrix(c(19,8,11,12,9,2,18,17,9,14,15,19,10,13,20),
+               ncol=3)
> Aug
     [,1] [,2] [,3]
[1,]   19    2   15
[2,]    8   18   19
[3,]   11   17   10
[4,]   12    9   13
[5,]    9   14   20
attr(, "class"):
[1] "Matrix"
> solve(Aug, c(9,5,14,11,8))
           [,1]
[1,]  0.8430639
[2,]  0.5332060
[3,] -0.4558612
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "workspace" "rcond"     "call"
> attr(.Last.value, "rcond")
[1] 0.1270667
```

If you are working with the QR form already, a special solve method takes advantage of the decomposition:

```
> Aug.qr <- qr(Aug)
> solve(Aug.qr, c(9,5,14,11,8))
           [,1]
[1,]  0.8430639
[2,]  0.5332060
[3,] -0.4558612
```

```
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
```

Solving Underdetermined Systems

There may be cases where you have many more variables than equations; such a system is called underdetermined. An underdetermined system has an infinite number of solutions; `solve.Matrix` finds the unique solution with minimum l_2 norm. For example, consider the following Matrix `wide.A`:

```
> wide.A <-Matrix(c(19,8,11,12,9,2,18,17,9,14,15,
+ 19,10,13,20), ncol=5)
> wide.A
     [,1] [,2] [,3] [,4] [,5]
[1,]   19   12   18   14   10
[2,]    8    9   17   15   13
[3,]   11    2    9   19   20
attr(, "class"):
[1] "Matrix"
> solve(wide.A, c(9,5,14))
           [,1]
[1,]  0.5638695
[2,] -0.2266716
[3,] -0.3116442
[4,]  0.2418549
[5,]  0.3230167
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "workspace" "rcond"     "call"
```

If you are working with the QR decomposition, the solve method does not compute the minimum l_2 solution; it does, however, compute one basic solution:

```
> wide.A.qr <- qr(wide.A)
> solve(wide.A.qr, c(9, 5, 14))
           [,1]
[1,]  0.8356868
[2,] -2.0616175
[3,]  0.9922978
```

```
[4,]  0.0000000
[5,]  0.0000000
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
Warning messages:
  Imaginary parts of complex data ignored in:
      as.double(if(rows <= k) b else
        rbind(b, matrix(0i, rows - k, l)))
```

That this is indeed a solution to the original problem can be verified as follows:

```
> wide.A %*% .Last.value - c(9,5,14)
              [,1]
[1,] -7.105427e-15
[2,] -1.776357e-15
[3,]  0.000000e+00
attr(, "class"):
[1] "Matrix"
```

Solving Rank-Deficient Systems

All of the methods described so far in this chapter have applied to full-rank systems; that is, systems in which the coefficient Matrix is non-singular. What about systems in which the coefficient Matrix is singular or nearly so? The Matrix library includes two solve methods for rank-deficient systems, both requiring decomposition of the original Matrix.

The first method uses the singular value decomposition:

```
> S <- Matrix(c(9,3,3,3,1,1,2,4,7),ncol=3,byrow=T)
> y <- c(9,5,14)
> solve(S, y)
Error in solve.Matrix(S, y): the matrix
      is exactly singular
Dumped
> x <- solve(svd(S), y, tol=1e-10)
Warning messages:
 singular solve in: solve(svd(S), y, tol=1e-10)
> x
          [,1]
[1,] 0.3140426
[2,] 0.8110638
```

```
[3,] 1.4468085
attr(, "class"):
[1] "Matrix"
other attributes:
[1] "rcond" "rank"  "call"
```

We can see how well this solves the original equation by computing $S^T(Sx-y)$; it should come close to vanishing.

```
> t(S) %*% (S %*% x - y)
              [,1]
[1,] -2.842171e-14
[2,] -2.131628e-14
[3,] -3.197442e-14
attr(, "class"):
[1] "Matrix"
```

If the coefficient Matrix is Hermitian, then the eigenvalue decomposition can be used as an alternative to the singular value decomposition to compute a least-squares solution.

```
> u <- 1:3
> v <- c(8,4,4)
> A <- u %*% t(u) + v %*% t(v)
> class(A) <- Matrix.class(A)
> class(A)
[1] "Hermitian"  "Matrix"
> solve(A, tol=.Machine$double.eps)
Error in solve.Hermitian(A, tol = .Machine$double...:
      prescribed tolerance exceeds reciprocal
      condition estimate 3.68233175663402e-18
Dumped
> y <- c(9,5,14)
> x <- solve(eigen(A), y, tol=.Machine$double.eps)
Warning messages:
  singular solve in: solve(eigen(A), y,
      tol = .Machine$double.eps)
```

We can see how well this solves the original equation by computing $A^T(Ax-y)$; it should come close to vanishing.

```
> t(A) %*% (A %*% x - y)
              [,1]
[1,] -1.605827e-12
[2,] -9.237056e-13
[3,] -9.876544e-13
```

```
attr(, "class"):
[1] "Matrix"
```

In both the singular value and Hermitian eigen solves, the computed solution is the minimum l_2 norm solution relative to `tol`.

Finding Matrix Inverses and Pseudo-Inverses

For most of the solve methods described in this section, you can obtain a calculation of the inverse of the coefficient Matrix simply by omitting the right hand side vector or Matrix. For example, the inverse of the full-rank Matrix *A* can be obtained as follows:

```
> solve(A) # uses solve.Matrix
            Sun           Mon          Tue
Apr  0.04219534 -0.069341989  0.06845677
May -0.03806433 -0.007376807  0.07111242
Jun  0.01829448  0.088816760 -0.09619357
attr(, "class"):
[1] "Matrix"
other attributes:
[1] "workspace" "rcond"     "call"
> solve(A) %*% A
    Apr           May Jun
Apr   1  4.440892e-16   0
May   0  1.000000e+00   0
Jun   0 -2.220446e-16   1
attr(, "class"):
[1] "Matrix"
```

The inverses of specialized Matrices are found using the specific methods for those classes:

```
> solve(my.Herm) # uses solve.Hermitian
                      [,1]                 [,2]                   [,3]
[1,]  0.14736842+0.0000000i -0.1684211-0.1684211i -0.05263158+0.2105263i
[2,] -0.16842105+0.1684211i  0.2421053+0.0000000i  0.10526316-0.1578947i
[3,] -0.05263158-0.2105263i  0.1052632+0.1578947i  0.10526316+0.0000000i
attr(, "class"):
[1] "Hermitian" "Matrix"

other attributes:
[1] "rcond"     "workspace" "call"
> solve(A.u) # uses solve.UpperTriangular
           [,1]       [,2]       [,3]
[1,] -0.04279605  0.03524793  0.09056031
[2,]  0.00000000 -0.05215548  0.06196848
[3,]  0.00000000  0.00000000 -0.13219799
```

```
attr(, "class"):
[1] "UpperTriangular" "Matrix"

other attributes:
[1] "rcond" "call"
```

For rectangular Matrices, `solve` with no right hand side produces a pseudo-inverse, the unique F-norm solution to the problem

$$min_{X \in R^{n \times m}} \|AX - I_m\| .$$

```
> solve(rect.Mat)
            [,1]       [,2]        [,3]        [,4]
[1,]  0.04838342 0.01686479  0.08183540 -0.10118876
[2,] -0.02230793 0.15820783 -0.04182985  0.05075302
[3,]  0.02665054 0.14699438  0.07130605  0.11803373
attr(, "class"):
[1] "Matrix"

other attributes:
[1] "workspace" "rcond"     "call"
```

29.5 CONTROLLING THE COMPUTATIONS

LAPACK has six machine- and problem-dependent parameters that you can adjust within S-PLUS to affect the performance of some functions:

NB	optimal block size
NBMIN	minimum block size for the block routine
NX	crossover point for switching from unblocked to block routine
NS	number of shifts for unsymmetric eigenvalues.

NXSVD — used to decide whether to apply *QR* factorization before computing singular values

MAXB — crossover point for unsymmetric eigenvalues

The LAPACK names are retained for the parameters for consistency with the *LAPACK User's Guide* (1994). See chapter 3 of that reference, Performance of LAPACK, for a general discussion of performance issues in LAPACK, and chapter 6, Installing LAPACK Routines, for a discussion of the tuning parameters. The NB, NBMIN, and NX parameters apply only to machines that allow parallel processing, and affect block size for distributed memory processing. The other parameters, which may affect performance on sequential machines as well as parallel, occur in singular-value, Schur, and non-Hermitian eigenvalue computations. You can adjust all the LAPACK tuning parameters using the `la.env` function; to see the current settings, call `la.env` with no arguments:

```
> la.env()
$NB:
[1] 1

$NBMIN:
[1] -1

$NX:
[1] -1

$NS:
[1] 2

$NXSVD:
[1] 16

$MAXB:
[1] 50
```

The `la.env` function initializes a Fortran common block for use within LAPACK. Each method that calls LAPACK calls `la.env` automatically, and has an argument `tune` that allows you to pass different tuning parameters. For example, in calculating a singular value decomposition, you might want to modify the NXSVD parameter:

```
> longmat <- Matrix(rnorm(1600), nrow=200)
> unix.time(svd(longmat))
[1] 0.5166664 0.3666668 3.0000000 0.0000000 0.0000000
```

```
> unix.time(svd(longmat, tune=list(NXSVD=30)))
[1] 0.45000076 0.04999924 0.00000000 0.00000000
[5] 0.00000000
```

Note
`unix.time` works only in Unix versions of S-PLUS. Often a change in tuning parameters implies a change in the amount of workspace needed for an LAPACK function to meet that specification. To accommodate this, a workspace parameter is provided in the relevant S-PLUS functions. LAPACK does not always provide a direct mapping between tuning parameter settings and the optimal workspace, but rather gives only the minimum workspace necessary to obtain the result of that function. The functions usually return the optimal workspace on completion and that information is included in the attributes of the S-PLUS functions that call them.

29.6 REFERENCES

Anderson, E. Bai, Z. and Bischof, C. and Demmel, J. and Dongarra, J. and Du Croz, J. and Greenbaum, A. and Hammarling, S. and McKenney, A. and Ostrouchov, S. and Sorensen, D. (1994). *LAPACK User's Guide*, 2nd edition. SIAM, Philadelphia.

Golub, Gene H. and Van Loan, Charles F. (1989). *Matrix Computations*, 2nd edition. Johns Hopkins University Press, Baltimore.

RESAMPLING TECHNIQUES: BOOTSTRAP AND JACKKNIFE

30

Use resampling methods for hypothesis tests or confidence intervals, when the distribution is not known.

RESAMPLING TECHNIQUES: BOOTSTRAP AND JACKKNIFE

In statistical analysis, the researcher is usually interested in obtaining not only a point estimate of a statistic but also an estimate of the variation in this point estimate, and a confidence interval for the true value of the parameter. For example, a researcher may calculate not only a sample mean but also the standard error of the mean and a confidence interval for the mean.

Traditionally, researchers have relied on the central limit theorem and normal approximations to obtain standard errors and confidence intervals. These techniques are valid only if the statistic, or some known transformation of it, is asymptotically normally distributed. Hence, if the normality assumption does not hold, then the traditional methods should not be used to obtain confidence intervals.

A major motivation for the traditional reliance on normal-theory methods has been computational tractability. Now, with the availability of modern computing power, researchers need no longer rely on asymptotic theory to estimate the distribution of a statistic. Instead, they may use resampling methods which return inferential results for either normal or non-normal distributions.

Resampling techniques such as the bootstrap and jackknife provide estimates of the standard error, confidence intervals, and distributions for any statistic. In the bootstrap, for example, *B* new samples, each of the same size as the observed data, are drawn *with replacement* from the observed data. The statistic is calculated for each new set of data, yielding a bootstrap distribution for the statistic. The fundamental assumption of bootstrapping is that the observed data are representative of the underlying population. By resampling observations from the observed data, the process of sampling observations from the population is mimicked. For more detailed descriptions of bootstrapping, see Efron and Tibshirani (1993), and Shao and Tu (1995).

S-PLUS 4.0 includes a suite of functions for bootstrapping and jackknifing with the following basic capabilities:

- Given a vector, matrix, or data frame, create bootstrap or jackknife resamples of observations, and use these to calculate resampling replicates of a specified statistic. The statistic may be a scalar, vector, or matrix, and may be specified as an S-PLUS function or call.

- Produce informative summaries and plots for a resample object (`resamp`) produced by bootstrapping or jackknifing.
- Calculate empirical percentile and BCa confidence limits for a bootstrap object, and empirical percentiles for a jackknife object.
- Use jackknife after bootstrap to examine the influence of observations, and to estimate the standard error of a functional of the bootstrap distribution for a statistic.

A list of the bootstrapping and jackknifing functions is presented in table 30.1.

Table 30.1: *S-PLUS bootstrapping and jackknifing functions.*

Function	Description
bootstrap	Main bootstrap function
jackknife	Main jackknife function
summary.bootstrap	Summary method for `bootstrap` objects
print.resamp, plot.resamp, qqnorm.resamp,summary.resamp	Methods for `resamp` objects
limits.emp, limits.bca	Calculate empirical and BCa percentiles
jack.after.bootstrap	Perform jackknife after bootstrap
print.jack.after.bootstrap, plot.jack.after.bootstrap	Methods for jackknife after bootstrap object
update.bootstrap	Add more replicates to a `boot` object
bootstats, jackstats	Called by `bootstrap` and `jackknife` to calculate resampling statistics
samp.boot.mc, samp.boot.bal, samp.permute	Functions to generate resampling indices

30.1 CREATING A RESAMPLE OBJECT

There are two types of resample objects: bootstrap objects and jackknife objects. The main functions for generating these objects are `bootstrap` and `jackknife`. These functions call the more primitive functions `bootstats` and `jackstats`, which use the replicated parameter values and other information to calculate the bootstrap or jackknife statistics, and return an object of the appropriate class.

The Bootstrap

In bootstrap resampling, *B* new samples, each of the same size as the observed data, are drawn *with replacement* from the observed data. The statistic is first calculated using the observed data, and then re-calculated using each of the new samples, yielding a bootstrap distribution. The resulting replicates are used to calculate the bootstrap estimates of bias, mean, and standard error for the statistic.

Main Arguments

The main arguments in bootstrapping are the `data` (a vector, matrix, or data frame) and a `statistic` (returning a scalar, vector, or matrix). This statistic may be an S-PLUS function or an unevaluated call (that is, any expression that one might type at the command line). Additional arguments to `statistic` may be passed as a list through `args.stat`.

The user may specify the number *B* of resamples to draw. The default is 1000, which is the recommended minimum for estimating percentiles. Although a smaller *B* may be specified, 250 is recommended as a minimum for estimating standard errors.

Optional Arguments

- `seed`: sets the random number seed. It may be a legal random number seed, or an integer between 0 and 1000.
- `group`: specifies a stratifying variable. If specified, then resampling is performed independently within each stratum. This argument can be used to bootstrap a two-sample or multiple-sample statistic. Note that the bootstrap estimates are not adjusted based on stratifying.
- `sampler`: generates resampling indices. The default function `samp.boot.mc` performs standard Monte Carlo bootstrapping of observations. The `samp.boot.bal` function performs balanced bootstrapping. In some cases, the `bootstrap` function may be used to perform a permutation test by using `samp.permute` with an appropriately defined statistic.

- `block.size`: controls computational details of the bootstrapping. By default, this is set to min(*B*, 100) and the bootstrapping is performed using one large `lapply`. If the sample size *n* and number *B* of resamples are large, then this default may be slower than the alternative of performing a `for` loop over smaller blocks of observations. The `block.size` argument specifies the size of each block over which a `for` is applied. For example, if `n=1000` and `B=1000`, then it may be preferable to do 10 loops with `block.size=100` rather than a single `lapply`.[1]
- `block.size`: controls computational details of the bootstrapping. For efficiency, the samples are drawn in blocks of size `block.size` and `lapply` is used over each block to evaluate the statistic. The drawing of blocks is embedded within a `for` loop to draw a total of `B` samples. When `n` is small it is most efficient to perform a single `lapply` so that `block.size=B`. When `n` is large it is more efficient to use a smaller `block.size`. For example, if `n=1000` and `B=1000`, then it may be preferable to do 10 loops with `block.size=100` rather than a single `lapply`. By default the `block.size` is set to min(100, *B*).
- `assign.frame1`: logical flag indicating whether the resampled data should be assigned to frame 1 before evaluating the statistic. This may be necessary if the statistic is reevaluating the call of a model object. If all bootstrap estimates are identical, try setting `assign.frame1=T`. Note that this will slow down the algorithm.
- `trace`: logical flag indicating whether to print a message indicating which set of replicates is currently being drawn.
- `save.indices`: logical flag indicating whether to save the matrix of resampling indices. By default, the value of the random number seed used is saved, and the sampler used is specified in the call, which is enough information to reproduce the resampling indices in later analyses. The matrix of resampling indices may be saved as part of the object by setting `save.indices=T`. This matrix has dimension `n` × `B`.

Additional arguments are described in the help file.

1. Pressing ESC during the looping interrupts the process and saves the replicates computed so far.

Other Functions

The `bootstrap` function calls the `bootstats` function to calculate bootstrap statistics. If the user specifies the required information, then `bootstats` may be called directly to produce a bootstrap object. The main caveat is that `limits.bca` and `jack.after.bootstrap` will look at the `call` component of the object, so the function calling `bootstats` should pass along an appropriate call if these functions are to be used on the resulting object.

Components of the Object

A `bootstrap` object has components `call`, `observed`, `replicates`, `estimate`, `B`, `n`, `dim.obs`, `group`, `seed.start`, and `seed.end`. The `observed` component contains the observed parameter values calculated using the original data. The `estimate` data frame contains bootstrap estimates of bias, mean, and standard error. The `replicates` are the bootstrap replicates of the parameters. The `call` component, starting random number seed `seed.start`, ending random number seed `seed.end`, and `group` are stored for future reference, as are the number `B` of replicates and the sample size `n`. If `statistic` returns a matrix, then its dimension is stored as `dim.obs` for use in the layout of plots. In many cases, `dim.obs` and `group` will be `NULL`.

The Jackknife

In jackknife resampling, a statistic is calculated for the `n` possible samples of size `n-1`, each with one observation left out. The default sample size is `n-1`, but more than one observation may be removed using the `group.size` argument (see below). Jackknife estimates of bias, mean, and standard error are available and are calculated differently than the equivalent bootstrap statistics.

Arguments

The `jackknife` function takes the arguments `data`, `statistic`, `args.stat`, and `assign.frame1`, which have the same meanings as for `bootstrap`.

The `seed` argument may be used to specify a seed for randomization done by the statistic, and for random assignment of observations to groups if `group.size` is not equal to one. It may be a legal random number seed, or an integer between 0 and 1000.

The `group.size` argument may be used to specify the removal of more than one point in each sample. This argument is useful in partial jackknifing for calculating the acceleration when forming BCa percentiles. It forms `floor(n/group.size)` replicates, each missing `group.size` observations. These replicates are treated as a jackknife sample of size `floor(n/group.size)`.

Other Functions

The `jackstats` function calculates the jackknife statistics.

30.2 METHODS FOR RESAMPLE OBJECTS

Print

The `print` method for a resample object prints out the call; the number of resamples used; and a table giving the values of the statistic for the original data and resampling estimates of bias, mean, and standard error for the statistic.

Summary

The `summary` method for a resample object prints out the same information as `print.resamp`, followed by the empirical percentiles of the replicates. The `summary` of a `bootstrap` object also calculates BCa percentiles. If the statistic is vector-valued, then a correlation matrix for the components of the vector is also printed. The optional `probs` argument specifies probabilities at which the empirical quantiles are calculated.

Additional arguments useful in `limits.bca` may be specified with `summary.bootstrap`. These arguments include `z0`, `acceleration`, and `group.size`. By default, a `group.size` of `floor(n/20)` is used in `limits.bca` for reasons of speed. To do a full jackknifing when estimating `acceleration`, specify `group.size=1`.

Plot

The `plot` method for a resample object produces plots of the distributions of the statistics. For each statistic, a histogram of the replicates is displayed with an overlaid smooth density estimate. A solid vertical line is plotted at the observed parameter value, and a dashed vertical line at the mean of the replicates.

The distance between the dotted line and the solid line is the estimated bias. The shape of the distribution may be examined to assess issues such as skewness of the distribution of the statistic.

The user may specify `plot` with a `bandwidth.func` argument to calculate the bandwidth of the density estimate. By default, the normal reference density estimate is used. In addition, the user may specify `plot` with a `nclass.func` argument to calculate the number of classes in the histogram. By default, the Freedman and Diaconis rule is used. Arguments may also be passed to `histogram` through the ellipsis (`...`).

Plots are displayed in a grid (`grid=T`) by default. Use `nrow` to specify the number of rows in the grid. If the statistic is a matrix, then by default the plots will be arranged in the same order as the terms appear in the matrix.

Normal Quantile-Quantile Plots

The `qqnorm` method for a resample object produces a plot with the same layout as in `plot.resamp`, but with each plot containing a normal quantile-quantile plot for the relevant statistic. If the argument `lines=T`, as is the default, then a `qqline` is also added to each plot.

This plot is used to assess the normality of the distribution of each statistic. If the points fall on a straight line, then the empirical distribution of the replicates is similar to that of a normal random variate.

30.3 PERCENTILE ESTIMATES

Two types of percentile estimates are supported: empirical percentiles, and bias-corrected and adjusted (BCa) percentiles. These are calculated by `limits.emp` and `limits.bca`, respectively. The empirical percentiles are available for bootstrap and jackknife objects, while BCa percentiles are available only for bootstrap objects. The empirical percentiles are easy to calculate, but may not be very accurate unless the sample size is very large. The BCa percentiles require more computation but are more accurate. For either type of percentile, using at least 1000 replications is recommended for accurate estimation.

The `probs` argument specifies which percentiles are computed.

Empirical Percentiles

The empirical percentiles are simply the percentiles of the empirical distribution of the replicates. Linear interpolation is used if necessary to obtain the specified percentiles.

BCa Percentiles

The BCa method transforms the specified `prob` values to determine which percentiles of the empirical distribution most accurately estimate the percentiles of interest. The percentiles of the empirical distribution corresponding to these values are then returned.

To estimate the BCa percentiles, the bias correction (denoted z_0) and the acceleration must be calculated. If these values are not specified (and they usually are not), then the bias correction will be obtained from the replicates, and the acceleration will be obtained using jackknifing. Note that rather than doing a complete delete-1 jackknife, the data are broken into groups of size `group.size`, and the groups are jackknifed. If `group.size` is not specified, then it is calculated as `floor(n/20)`, which will yield roughly 20 jackknife replicates, depending on the magnitude of `n`.

To return the values of `z0`, `acceleration`, and the empirical percentile level for each BCa percentile, set `detail=T`.

30.4 JACKKNIFE AFTER BOOTSTRAP

Jackknife after bootstrap is a technique for obtaining estimates of the variation in functionals of a bootstrap distribution, such as the bias or standard error of a statistic, without performing a second level of

bootstrapping. It also provides information on the influence of each observation on the functionals. See Efron and Tibshirani (pp. 275-280) for details on this procedure.

Simulation studies have shown that, in general, jackknife after bootstrap standard error estimates tend to be too large. A technique called weighted jackknife after bootstrap may resolve some of these difficulties. This technique is currently under investigation and has not yet been implemented.

The Jackknife After Bootstrap Object

The jackknife after bootstrap object has components `call`, `functional`, `rel.influence`, `large.rel.influence`, `values.functional`, `dim.obs`, and `threshold`. The value of the functional for the bootstrapped parameter replicates, and for the jackknife after bootstrap estimates of standard errors, is given as the `functional` data frame. The value of the functional over the samples with each point removed is given in `values.functional`. Normalized versions of these values are given in `rel.influence`. The list `large.rel.influence` gives the relative influence values for points with absolute relative influences in excess of `tolerance`. The `call` is the call to `jack.after.bootstrap`. The `dim.obs` is the corresponding component of the `bootstrap` object. The jackknife after bootstrap object is of class `jack.after.bootstrap`.

Print Method

The `print` method for a `jack.after.bootstrap` object displays the `call`, the description of the functional under consideration, the data frame of functional values and standard errors, and the list of large relative influences.

Plot Method

The `plot` method for a `jack.after.bootstrap` object produces a plot for each parameter, indicating the relative influence of each observation. Values greater than a specified tolerance (default = 2) are flagged as being particularly influential.

30.5 EXAMPLES

This section describes three examples. The first is a bootstrap of a variance, and discusses the output and basic plots associated with the bootstrap object. The second example resamples a correlation coefficient, and details the application of bootstrap, jackknife after bootstrap, and jackknife tools. The third example shows how to test linear regression coefficients using the bootstrap and jackknife after bootstrap.

Resampling the Variance

This example uses data from the `swiss.x` matrix, which contains socioeconomic indicators for the provinces of Switzerland in 1888. More particularly, this example resamples the variance of the `Education` variable, the percent of the population whose education is beyond primary school.

First, `Education` is separated from the `swiss.x` matrix.

```
> Education <- swiss.x[,3]
> Education
 [1] 12  9  5  7 15  7  7  8  7 13  6 12  7 12  5  2  8 28 20
[20]  9 10  3 12  6  1  8  3 10 19  8  2  6  2  6  3  9  3 13
[39] 12 11 13 32  7  7 53 29 29
```

The `bootstrap` function is used to draw resamples and construct a bootstrap object.[1]

```
> boot.obj1 <- bootstrap(Education, var, B=1000, seed=0)
Forming replications  1  to  100
Forming replications  101  to  200
Forming replications  201  to  300
Forming replications  301  to  400
Forming replications  401  to  500
Forming replications  501  to  600
Forming replications  601  to  700
Forming replications  701  to  800
Forming replications  801  to  900
Forming replications  901  to  1000
```

(To prevent the preceding messages from being displayed, set `trace=F`.)

Printing the object displays the call used to construct it, the number of replications used, and summary statistics for the parameter. The summary statistics are the observed value of the parameter, the mean of the parameter estimate replicates, and bootstrap estimates of bias and standard error.

```
> boot.obj1
Call:
bootstrap(data = Education, statistic = var, B = 1000,
seed = 0)

Number of Replications: 1000

Summary Statistics:
    Observed   Bias  Mean    SE
var    92.46 -3.362 89.09 38.67
```

1. All examples use `B` = 1000, the number of resamples recommended for accurate estimation of percentiles. Users who want to replicate the examples might use a lower number of resamples (say, `B` = 250) to speed up estimation. Note, however, that results will differ slightly from those shown here.

A more complete summary of the bootstrap object, obtained via the `summary` function, includes empirical and BCa percentiles for the statistic. The BCa percentiles, for example, show that the 95% confidence interval for the `Education` variance has endpoints 45.34 and 221.2.

```
> summary(boot.obj1)

Call:
bootstrap(data = Education, statistic = var, B = 1000,
seed = 0)

Number of Replications: 1000

Summary Statistics:
    Observed   Bias  Mean    SE
var    92.46 -3.362 89.09 38.67

Empirical Percentiles:
    2.5%    5%   95% 97.5%
var 32.9 36.17 163.9 177.1
BCa Percentiles:
     2.5%    5%   95% 97.5%
var 45.34 51.44 211.6 221.2
```

Empirical and BCa percentiles may also be obtained separately using the `limits.emp` and `limits.bca` functions, respectively.

```
> limits.emp(boot.obj1)

         2.5%       5%      95%    97.5%
var 32.89544 36.16716 163.8941 177.1408

> limits.bca(boot.obj1)

         2.5%      5%      95%    97.5%
var 45.33665 51.4373 211.6284 221.1731
```

Plotting the bootstrap object provides a histogram of the replicated variances along with a smooth density estimate (figure 30.1). The solid line indicates the observed parameter value, and the dotted line indicates the mean of the replicates. The difference between these two values is the bootstrap estimate of bias.

```
> plot(boot.obj1)
```

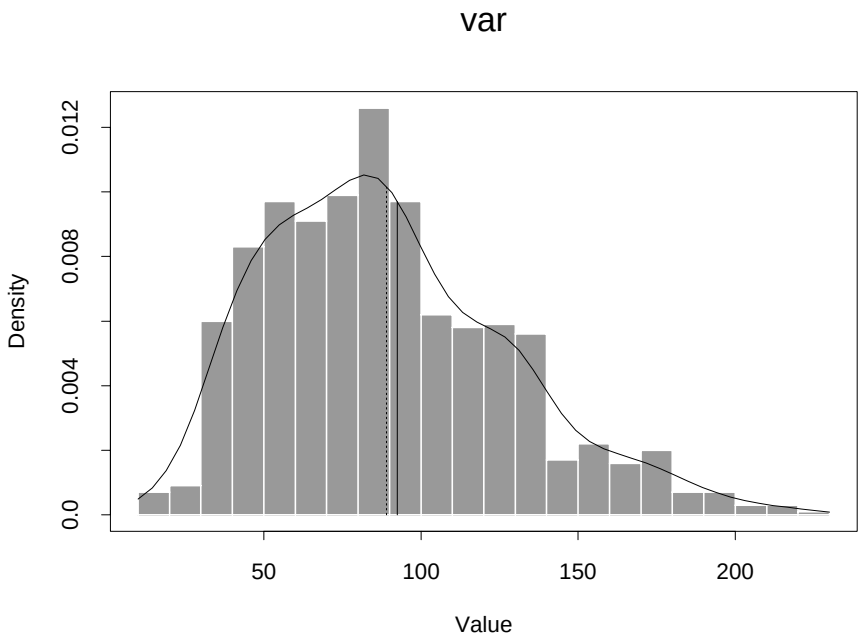

Figure 30.1: *Histogram of replicated variances.*

The histogram in figure 30.1 shows that the distribution of replicated variances is highly skewed. A normal quantile-quantile plot can be used to further assess deviation from the normal distribution. Figure 30.2 suggests that both tails of the distribution of replicated variances deviate from the normal distribution. Thus there is evidence that bootstrapping is a better approach than normal-based methods.

```
> qqnorm(boot.obj1)
```

Resampling the Correlation Coefficient

This example uses the law school data from Efron and Tibshirani (p.19). Starting with 82 American law schools participating in a study of admission practices, they constructed a random sample of 15 schools. Efron and Tibshirani then examined the correlation between LSAT score and GPA for the 1973 entering classes at these schools (p. 49).

Traditionally, Fisher's transformation would be used to transform the correlation coefficient into a normally distributed variable on which normal-based inference would be used. This example uses resampling to obtain inferential quantities instead of employing Fisher's transformation.

First, the data are entered into S-PLUS and stored as a data frame.

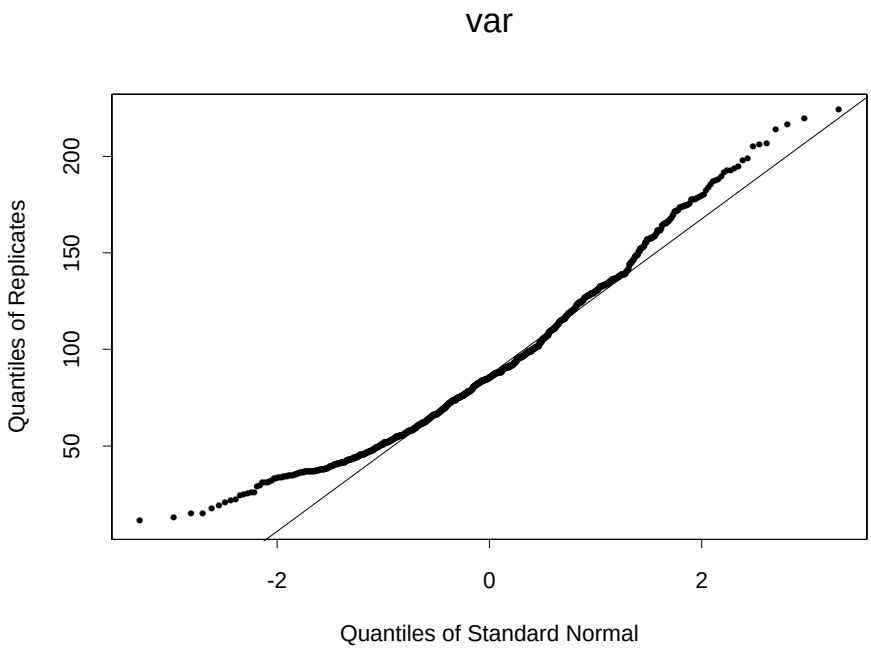

Figure 30.2: *Normal Q-Q plot of replicated variances.*

```
> school <- 1:15
> lsat <- c(576,635,558,578,666,580,555,661,651,605,653,
+ 575,545,572,594)
> gpa <- c(3.39,3.30,2.81,3.03,3.44,3.07,3.00,3.43,3.36,
+ 3.13,3.12,2.74,2.76,2.88,2.96)
> law.data <- data.frame(School=school,LSAT=lsat,GPA=gpa)
```

Next, the `bootstrap` function is used, and the summary of the resulting object displayed.

```
> boot.obj2 <- bootstrap(law.data, cor(LSAT,GPA),
+ B=1000, seed=0, trace=F)
> summary(boot.obj2)
Call:
bootstrap(data = law.data, statistic = cor(LSAT, GPA),
B = 1000, seed = 0, trace = F)

Number of Replications: 1000

Summary Statistics:
      Observed      Bias   Mean     SE
Param   0.7764 -0.008768 0.7676 0.1322
```

```
Empirical Percentiles:
        2.5%    5%    95%  97.5%
Param 0.4673 0.523 0.9432 0.9593

BCa Percentiles:
        2.5%    5%    95%  97.5%
Param 0.3443 0.453 0.9255 0.9384
```

The bootstrap object is plotted to obtain a histogram of the replicated correlation values along with a smooth density estimate (figure 30.2). The distribution is clearly skewed.

```
> plot(boot.obj2)
```

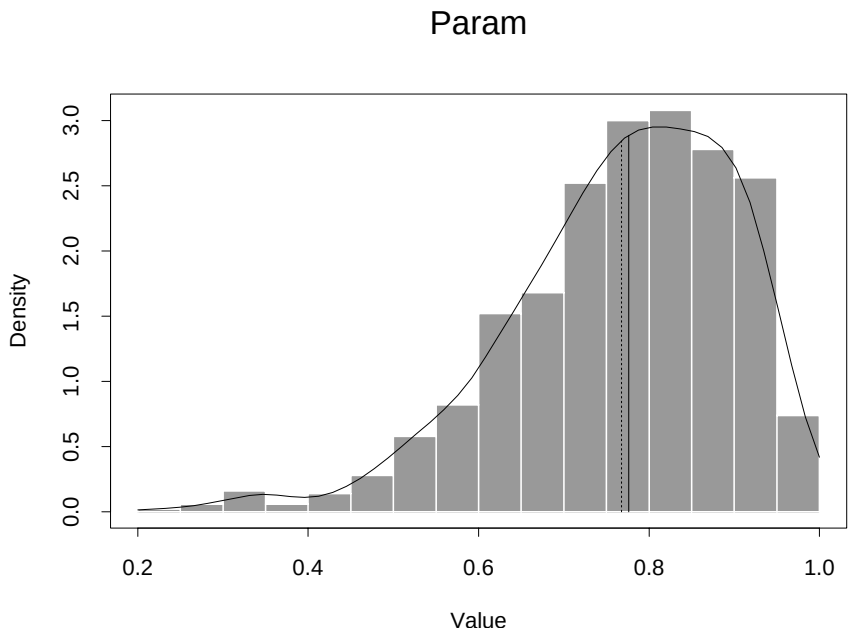

Figure 30.3: *Histogram of replicated correlations.*

Another tool available for exploring the bootstrap object is the jackknife after bootstrap (Efron and Tibshirani, p. 275). This technique provides standard error estimates for functionals of the bootstrap distribution, and influence measures for each observation. By default, the functional is the mean of the distribution. In this case, the standard error of the functional is the standard error of the mean, and the influence indicates the influence of each observation on the mean. Jackknife after bootstrap is commonly used to get standard error estimates for the bootstrap estimate of standard error.

```
> jab.obj2 <- jack.after.bootstrap(boot.obj2)
> jab.obj2

Call:
jack.after.bootstrap(boot.obj = boot.obj2, functional =
mean)

Functional Under Consideration:
mean

Functional of Bootstrap Distribution of Parameters:
        Func SE.Func
Param 0.7676  0.1432

Observations with Large Influence on Functional:
$Param:
   Param
1 -3.025
```

Plotting the `jack.after.bootstrap` object provides an influence plot similar to a Cook's distance plot (figure 30.4). Observations with absolute relative influence greater than 2 are considered particularly influential.

```
> plot(jab.obj2)
```

The jackknife after bootstrap identifies observation 1 as being particularly influential. A plot of LSAT versus GPA with this observation plotted as a circle shows that this point is indeed an outlying observation (figure 30.5).

```
> plot(lsat[-1], gpa[-1], xlab="LSAT", ylab="GPA")
> points(lsat[1], gpa[1], pch=2)
```

Jackknife summary statistics for the correlation may be obtained also.

```
> jackknife(law.data,cor(LSAT,GPA))

Call:
jackknife(data = law.data, statistic = cor(LSAT, GPA))

Number of Replications: 15


Summary Statistics:
      Observed      Bias   Mean     SE
Param   0.7764 -0.006473 0.7759 0.1425
```

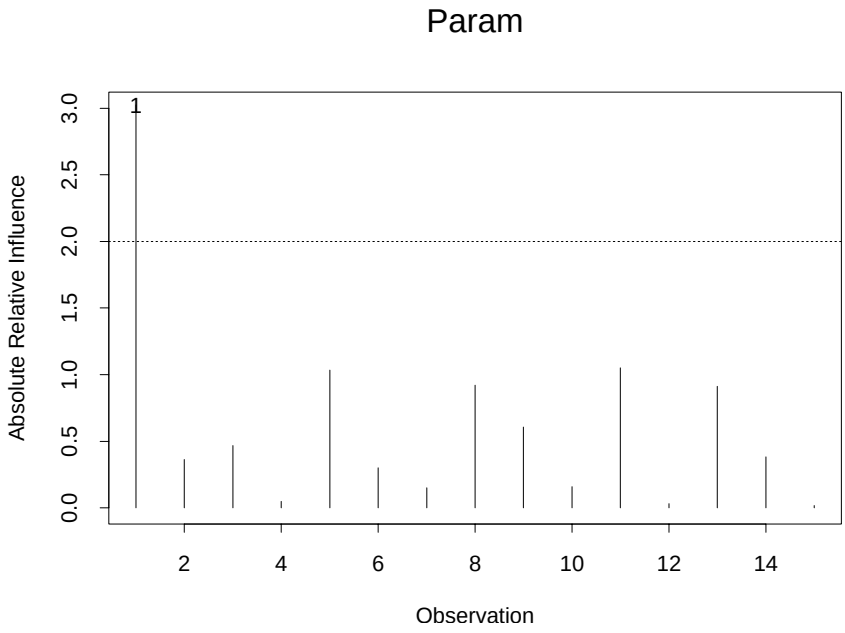

Figure 30.4: *Influence plot for correlation.*

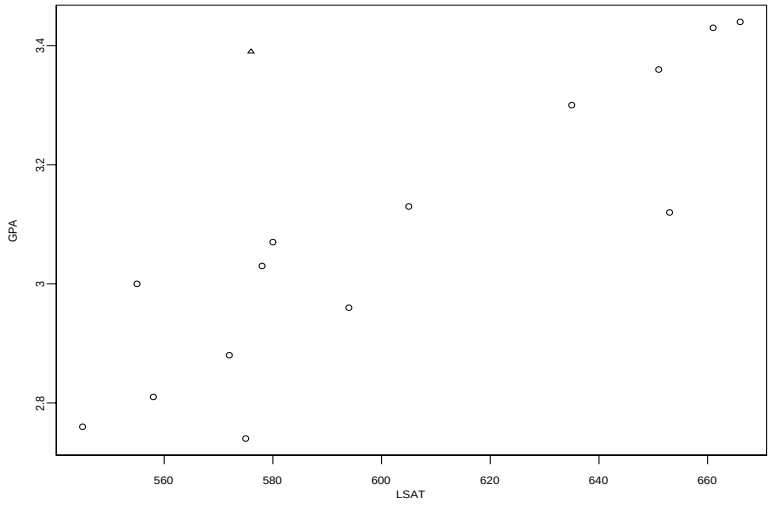

Figure 30.5: *LSAT versus GPA.*

Resampling Regression Coefficients

The last example shows how to test linear regression coefficients, and uses the bootstrap to obtain standard error estimates and confidence intervals.

The data are from operation of a plant for the oxidation of ammonia to nitric acid, measured on 21 consecutive days. See the S-PLUS help file for `stack` for details.

First, the `stack.loss` vector and `stack.x` matrix are combined into a data frame.

```
> stack <- data.frame(stack.loss, stack.x)
> names(stack)
[1] "stack.loss" "Air.Flow"   "Water.Temp" "Acid.Conc."
```

The `bootstrap` function resamples the vector of linear regression coefficients from the model of `stack.loss` regressed on `Air.Flow`, `Water.Temp`, and `Acid.Conc.`.

```
> boot.obj3 <- bootstrap(stack,
+ coef(lm(stack.loss~Air.Flow+Water.Temp+Acid.Conc.,
+ stack)), B=1000, seed=0, trace=F)
> boot.obj3
Call:
bootstrap(data = stack, statistic = coef(lm(stack.loss ~
Air.Flow + Water.Temp + Acid.Conc., stack)), B = 1000,
seed = 0, trace = F)

Number of Replications: 1000

Summary Statistics:
            Observed      Bias     Mean     SE
(Intercept) -39.9197  0.829215 -39.0905 8.8239
   Air.Flow   0.7156  0.004886   0.7205 0.1749
 Water.Temp   1.2953 -0.031415   1.2639 0.4753
 Acid.Conc.  -0.1521 -0.005164  -0.1573 0.1180
```

The summary for a vector statistic includes the correlation matrix for the replicate values. Based on the 95% confidence limits, for either the empirical or the BCa percentiles, all coefficients except the `Acid.Conc.` coefficient are significantly different from zero.

```
> summary(boot.obj3)
Call:
bootstrap(data = stack, statistic = coef(lm(stack.loss ~
Air.Flow + Water.Temp + Acid.Conc., stack)), B = 1000,
seed = 0, trace = F)
```

```
Number of Replications: 1000

Summary Statistics:
            Observed      Bias     Mean     SE
(Intercept) -39.9197  0.829215 -39.0905 8.8239
   Air.Flow   0.7156  0.004886   0.7205 0.1749
 Water.Temp   1.2953 -0.031415   1.2639 0.4753
 Acid.Conc.  -0.1521 -0.005164  -0.1573 0.1180

Empirical Percentiles:
               2.5%       5%      95%     97.5%
(Intercept) -55.4846 -52.7583 -23.4913 -17.84522
   Air.Flow   0.3844   0.4454   1.0136   1.05255
 Water.Temp   0.3913   0.4768   2.0544   2.23920
 Acid.Conc.  -0.4181  -0.3604   0.0209   0.06103

BCa Percentiles:
               2.5%       5%        95%     97.5%
(Intercept) -58.8427 -54.3320 -25.385390 -21.48317
   Air.Flow   0.3197   0.3897   0.987308   1.01691
 Water.Temp   0.4977   0.5811   2.278439   2.46017
 Acid.Conc.  -0.4250  -0.3743   0.008729   0.04447

Correlation of Replicates:
            (Intercept) Air.Flow Water.Temp Acid.Conc.
(Intercept)     1.00000  -0.1376    0.03551    -0.7848
   Air.Flow    -0.13760   1.0000   -0.79387    -0.1096
 Water.Temp     0.03551  -0.7939    1.00000    -0.2007
 Acid.Conc.    -0.78483  -0.1096   -0.20067     1.0000
```

The plot function provides histograms of the replicated regression coefficients (figure 30.6). Skewness is particularly evident in the `Acid.Conc.` coefficients.

```
> plot(boot.obj3)
```

Next, the jackknife after bootstrap is used to assess the accuracy of the standard error estimates, and the influence of each observation on these estimates.

```
> jab.obj3 <- jack.after.bootstrap(boot.obj3,"SE")
> jab.obj3
Call:
jack.after.bootstrap(boot.obj = boot.obj3, functional = "SE")
```

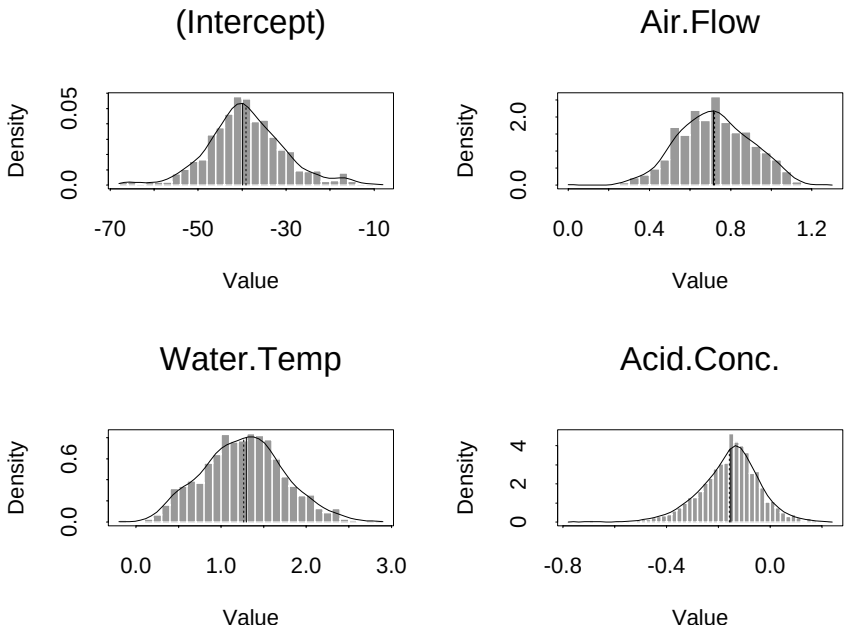

Figure 30.6: *Histograms of replicated regression coefficients.*

```
Functional Under Consideration:
[1] "SE"

Functional of Bootstrap Distribution of Parameters:
             Func SE.Func
(Intercept) 8.8239 3.67775
   Air.Flow 0.1749 0.06149
 Water.Temp 0.4753 0.17850
 Acid.Conc. 0.1180 0.05395

Observations with Large Influence on Functional:

$"(Intercept)":
   (Intercept)
21       2.863

$Air.Flow:
   Air.Flow
21    3.672
```

```
$Water.Temp:
   Water.Temp
21      3.214

$Acid.Conc.:
   Acid.Conc.
14     -2.184
21      2.589
```

The jackknife after bootstrap and the corresponding influence plot (figure 30.7) suggest that points 14 and 21 are particularly influential.

```
> plot(jab.obj3)
```

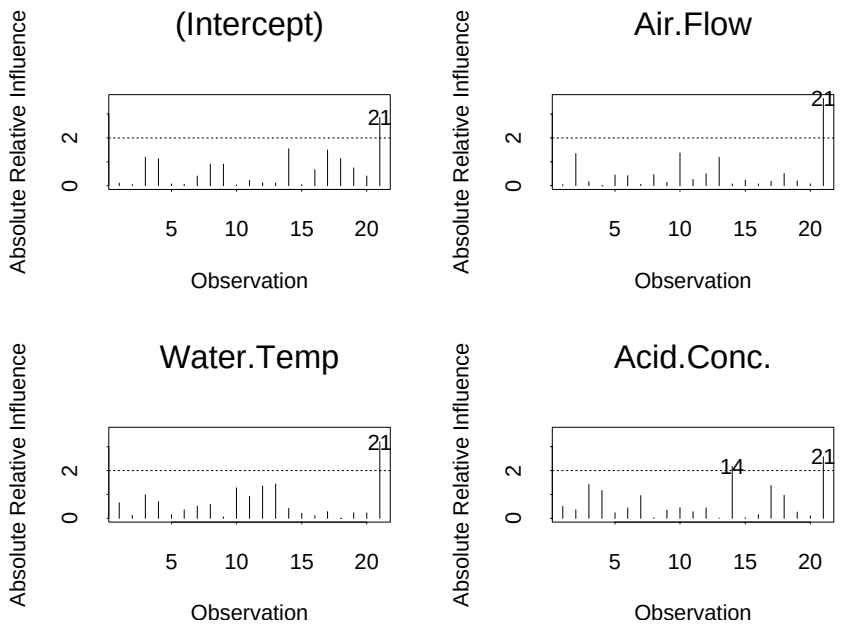

Figure 30.7: *Influence plots for regression coefficients.*

30.6 REFERENCES

Efron, B. and Tibshirani, R. J. (1993). *An Introduction to the Bootstrap*. Chapman & Hall: San Francisco.

Shao, J. and Tu, D. (1995). *The Jackknife and Bootstrap*. Springer-Verlag: New York.

INDEX

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Trademarks

- S-PLUS is a registered trademark, and StatServer, S+INTERFACE, S+SPATIALSTATS, S+GISLINK, S+DOX, S+WAVELETS, and AXUM are trademarks of MathSoft Inc.
- Trellis, S and New S are trademarks of Lucent Technologies, Inc.
- Intel is a registered trademark and 486, SX and Pentium are trademarks of Intel Corporation.
- Microsoft, Windows, MS-DOS, and Excel are registered trademarks and Windows NT is a trademark of Microsoft Corporation.
- All other trademarks are acknowledged.