An Introduction to S and The Hmisc and Design Libraries

Carlos Alzola, MS
Statistical Consultant
501 SE Glyndon Street
Vienna, Va 22180
calzola@cox.net

Frank Harrell, PhD
Professor of Biostatistics and Statistics
Division of Biostatistics and Epidemiology
Department of Health Evaluation Sciences
Box 800717, University of Virginia School of Medicine
Charlottesville Va 22908 USA
fharrell@virginia.edu
http://hesweb1.med.virginia.edu/biostat/s/splus.html

March 4, 2004

Updates to this document may be obtained from hesweb1.med.virginia.edu/biostat/s/doc/splus.pdf.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 S, S-Plus, R, and Source References

S-Plus and R are supersets of the **S** language[1], an interactive programming environment for data analysis and graphics. Insightful Corporation in Seattle took the AT&T Bell Labs S code and enhanced it producing many new statistical functions and graphical interfaces. In this text we use S to refer to both S-Plus and R languages.

S is a unique combination of a powerful language and flexible, high-quality graphics functions. What is most important about S is that it was designed to be extendable. Insightful, AT&T (now Lucent Technologies), and a large community of S-Plus users and R developers and users are constantly adding new capabilities to the system, all using the same high-level language. S allows users to take advantage of an explosion of powerful new data analysis and statistical modeling techniques. The richness of the S language and its planned extendability allow users to perform comprehensive analyses and data explorations with a minimum of programming. As an example, S functions in the Design library (see Chapter 9) can perform analyses and make graphical representations that would take pages of programming in other systems if they could be done at all:

[1] S, which may stand for *statistics*, was developed by the same lab that developed the C language.

```
# Fit binary logistic model without assuming linearity for age or
# equal shapes of the age relationship for the two sexes
# Represent age using a restricted cubic spline function with 4 knots
# This requires 3 age parameters per sex.  Model has intercept + 6
# coefficients.  x=T, y=T causes design matrix and response vector to
# be stored in the fit object f.  This allows certain residuals to be
# computed later, and it allows the original data to be re-analyzed
# later (e.g., bootstrapping and cross-validation)

f ← lrm(death ~ rcs(age,4)*sex, x=T, y=T)

# Test for age*sex interaction (3 d.f.), linearity in age (4 d.f.),
# overall age effect (6 d.f.), overall sex effect (4 d.f.),
# linearity of age interaction with sex (2 d.f.)
anova(f)

# Compute the 60:40 year odds ratio for females
summary(f, age=c(40,60), sex='female')

# Plot the age effects separately by sex, with confidence bands
plot(f, age=NA, sex=NA)

# Validate the model using the bootstrap - check for overfitting
validate(f)

# Draw a nomogram depicting the model, adding an axis for the
# predicted probability of death
nomogram(f, fun=plogis, funlabel='Prob(death)')

# Get predicted log odds of death for 40 year old male
predict(f, data.frame(age=40,sex='male'))

# Make a new S-Plus function which analytically computes predicted
# values from the fitted model
g ← Function(f)

# Use this function to duplicate the above prediction for 40 year old male
g(age=40, sex='male')
```

By making a high-level language the cornerstone of S-Plus, you could say that S-Plus is designed to be inefficient for some applications from a pure CPU time point of view. However, computer time is inexpensive in comparison with personnel time, and analysts who have learned S-Plus can be very much more productive in doing data analyses. They can usually do more complex and exploratory analyses in the same time that standard analyses take using other systems.

In its most simple use, S-Plus is an interactive calculator. Commands are executed (or debugged) as they are entered. The S language is based on the use of functions to perform calculations, open graphics windows, set system options, and even for exiting the system. Variables can refer to single-valued scalars, vectors, matrices, or other forms. Ordinarily a variable is stored as a vector, e.g., `age` will refer to all the ages of subjects in a dataset. Perhaps the biggest challenge to learning S-Plus for long-time users of single observation oriented packages such as SAS is to think in terms of vectors instead of a variable value for a single subject. In SAS you might say

```
PROC MEANS; VAR age;   *Get mean and other statistics for age;
DATA new; SET old;
  IF age < 16 THEN ageg='Young'; ELSE ageg='Old';
```

The IF statement would be executed separately for each input observation. In contrast, to reference a single value of `age` in S-Plus, say for the 13^{th} subject, you would type `age[13]`. To create the `ageg` variable for all subjects you would use the S-Plus `ifelse` function, which operates on vectors[2]:

```
mean(age)  # Computed immediately, not in a separate step
ageg ← ifelse(age < 16, 'Young', 'Old')
```

The assignment operator ← is typed as `<-`.

To show how function calls can be intermixed with other operations, look how easy it is to compute the number of subjects having `age` < the mean age:

```
sum(age < mean(age))   # could have used table(age < mean(age))
                       # or to get the proportion, use mean(age < mean(age))
```

In S-Plus you can create and operate on very complex objects. For example, a flexible type of object called a *list* can contain any arbitrary collection of other objects. This makes examination of regression model fits quite easy, as a "fit object" can contain a variety of objects of differing shapes, such as the vector of regression coefficients, covariance matrix, scalar R^2 value, number of observations, functions specifying how the predictors were transformed, etc.

S-Plus is *object oriented*. Many of its objects have one or more *classes*, and there are generic functions that know what to do with objects of certain classes. For example, if you use S-Plus's linear model function `lm` to create a "fit object" called `f`, this object will have class `'lm'`. Typing the commands `print(f)`, `summary(f)`, or `plot(f)` will cause the `print.lm`, `summary.lm`, or `plot.lm` functions to be executed. Typing `methods(lm)` or `methods(class='lm')` will give useful information about methods for creating or operating on `lm` objects.

Basic sources for learning S-Plus are the manuals that come with the software, especially the manual for the S-Plus Student Edition (1999). Also see *A Gentle Introduction to S-Plus*) and the extensive online manuals (for Windows). Another basic source for learning S (and hence, S-Plus) is a book called the *New S language* (a.k.a. "the blue book"), by Becker, Chambers and Wilks (1988). One step above the previous one is Chambers and Hastie (1992). Good introductions are Spector (1994) and Krause and Olson (2000). Other excellent books are Venables and Ripley (1999, 2000). Ripley has many useful S-Plus functions and other valuable material available from his Web page (http://www.stats.ox.ac.uk/~ripley/)[3]. The book by Lam entitled *An Instroduction to S-Plus for Windows* is a good book for learning the Windows S-Plus graphical user interface. A variety of manuals come with S-Plus, from beginner's guides to more advanced programmer's manuals. Also see F.E. Harrell's book Regression Modeling Strategies (which has long case studies using S-Plus with commands and printed and graphical output), and other references listed in the bibliography. Another source of help is the S-news mailing list maintained by

[2]Note that a missing value for `age` in SAS would result in the person being categorized as `'Young'`. In S-Plus the result would be a missing value (`NA`) for such subjects.

[3]Venables and Ripley's `MASS` S-Plus library has a wide variety of useful functions as well as many datasets useful for learning both biostatistical methods and S-Plus.

the Department of Biostatistics at Washington University. To subscribe, send an electronic message to `s-news-request@wubios.wustl.edu` with the message 'subscribe'. Washington University also maintains an excellent searchable S-news archive — see `http://www.biostat.wustl.edu/s-news/`.

Although not exclusively related to S, the `statlib` Web server `lib.stat.cmu.edu` can provide specific software for some problems. Also consult Insightful's Web page `http://www.insightful.-com`. The AT&T / Lucent Technologies Web page (`http://www.research.att.com/areas/stat/`) points to many valuable technical reports related to the S language. Finally, you can E-mail the authors of this document, and watch the movies (with soundtracks, no less) that the second author has placed on the UVa web page. These movies contain actual Windows S-Plus sessions demonstrating the author's Hmisc and Design libraries although they are now somewhat out of date. The "Visual Demo", available from the `Help` button in Windows S-Plus is a helpful introduction to the system.

We will concentrate on using S-Plus from a Linux or UNIX workstation or Windows S-Plus for Microsoft Windows or NT. When we do not distinguish between the two platforms, most of the commands described will work exactly the same in both contexts.

1.1.1 R

R is an open-source version of the S language (strictly speaking R uses a language that is very compatible with but not identical to S). R runs on all major platforms including running in native mode on some Macintosh operating systems. All of R's source code is available for inspection and modification. The system and its documentation is available from `http://www.r-project.org`. The Hmisc and Design libraries are fully available for R. Almost all of the command syntax used in this book can be used identically for R. There are many subtle differences in how graphical parameters are handled. R uses essentially Version 3 of the S language but with different rules for how objects are found from inside functions when they are not passed as arguments.

R has no graphical user interface on Linux and UNIX and only a rudimentary one on Windows. It lacks many of the Microsoft Office linkages and data import/export capabilities that S-Plus has. It has most of the functions S-Plus has, however. R runs slightly faster than S-Plus for certain applications (especially highly iterative ones) and provides easy-to-use functions for downloading and updating add-on libraries (which R calls "packages"). As R is free, it can readily be used in conjunction with web servers. For a software developer, R's online help files are somewhat better organized than those in S-Plus.

1.2 Starting S

1.2.1 UNIX/Linux

For now, we will discuss the use of S-Plus interactively. Before you start S-Plus, you should have created a directory where you will keep the data and code related to the particular project. For instance, in UNIX from an upper level directory, type

```
mkdir sproject
cd sproject
```

Next type

```
mkdir .Data
```

At this point you may want to set up so that S-Plus does not write to an ever-growing audit file. The `.Audit` file accumulates a file of all S-Plus activity across sessions. As this file can become quite large, you can turn it off by forming a new empty `.Audit` using `touch .Data/.Audit`, and setting the file to be non-writable using `chmod -w .Data/.Audit`.

Now you're ready to invoke S-Plus.

```
Splus
S-PLUS : Copyright (c) 1988, 1995 MathSoft, Inc.
S : Copyright AT&T.
Version 3.3 Release 1 for Sun SPARC, SunOS 4.1.x : 1995
Working data will be in .Data
>
```

If you had not created a `.Data` (in what follows assume the name is `_Data` for Windows) directory under your project area, S-Plus would have done it for you, but it would have placed it under your home directory instead of the project-specific directory. Creating `.Data` yourself results in more efficient management of your data, since (for now) everything will be stored *permanently* under `.Data`.

In Linux/UNIX, R is invoked by issuing the command `R` at the shell prompt. R data management is discussed in Section 4.2.

While in S-Plus, you have access to all the operating system commands. The command to escape to the shell is `!`. So, if you want a list of the files in your .Data directory, including hidden files, creation date and group ownership, you could type:

```
> !ls -lag .Data
total 90
drwxr-xr-x  2 cfa      staff        1024 Jun 18 10:28 .
drwxr-xr-x  7 cfa      staff        1536 Aug 11  1992 ..
-rw-r--r--  1 cfa      staff       85135 Jun 18 10:28 .Audit
-rw-r--r--  1 cfa      staff         132 Feb 14  1992 .First
-rw-r--r--  1 cfa      staff          16 Jun 18 10:10 .Last.value
-rw-r--r--  1 cfa      staff          64 May  5  1992 .Random.seed
-rw-r--r--  1 cfa      staff         229 May  5  1992 freqs
-rw-r--r--  1 cfa      staff          24 May  5  1992 i
-rw-r--r--  1 cfa      staff      520431 Nov 12  1992 impute.dframe
```

In R, use the `system` function to issue operating system commands. In either R or S-Plus you can use the Hmisc`sys` command.

Notice the file called `.First`. Its purpose is similar to that of an `autoexec.sas`, that is, it executes commands that you want done every time you start S. More on it later. (You could also have a .Last as well, for things you want S to do when you leave the system). Another way to execute operating system commands is to type `unix("command")`. The `unix` command is used more frequently in a programming environment.

1.2.2 Windows

Windows users first need to decide whether they want to put all objects created by S-Plus in one central `.Data`[4] directory or in a project-specific area. The former is OK when you are first learning the system. Later it's usually best to put the data into a project-specific directory. That way the directory stays relatively small, is easier to decide which objects can be deleted when you do spring cleaning, and you can more quickly back up directories for active projects. Users can manage multiple project `.Data` directories using the *Object Explorer* in S-Plus(see Section 4.2.3), but this method alone does not do away with the need for the `Start in` or current working directory to be set so that the `File` menu will start searching for files with your project area. Defining the `Start in` directory also allows S-Plus commands that read and write external files (e.g., `source`, `cat`, `scan`, `read.table`) to easily access your project directory and not some hidden area. Note that S-Plus 6 has a menu option for easily changing between project areas.

The existence of the current working directory is what distinguishes S-Plus from Microsoft Word or Excel, applications that can be easily started from isolated files. These applications do not need to link binary data files with raw data files, program files, GUI preference files, graphics and other types of objects. Therefore you do not need to create customized Windows shortcuts to invoke Word, Excel, etc., although Microsoft Office Binder can be used to link related files when the need arises.

The best way to set up for using Windows S-Plus is to use `My Computer` or `Explorer` to create a shortcut to S-Plus from within your project directory (if you don't have a project directory you can create one using `My Computer` or `Explorer`). Right click and select `New ... Shortcut`. Then `Browse` to select the file where `Splus` is stored. This will be under the `cmd` directory (under something like `splus` or `splusxx`) and will have the regular S-Plus icon next to it. After creating the basic default short cut, right click on its icon and select `Properties`. In the `Command line:` box click to the right of `Splus.exe` and add something like `S_DATA=.\.Data S_CWD=.` . In the `Start in` box type the full path name of your project directory, e.g. `c:\projects\myproject`. By specifying `S_CWD` and `S_DATA`, S-Plus will use a central area such as `\splusxx\users\yourname` for storing the `_Prefs` directory. `_Prefs` holds details about the graphical user interface and any changes you may have made to it such as system options. As `_Prefs` is about 100K in size, this will save disk space when you have many project, and let settings carry over from one project to another. If you want a separate `_Prefs` directory in each project area, substitute `S_PROJ=.` for `S_CWD` and `S_DATA` in the shortcut's command line.

Creation of the S-Plus shortcut only needs to be done once per project (for S-Plus 6 you may not need to do it at all). Then to enter S-Plus with everything pointed to `my project`, click or double-click on the new S-Plus icon. Depending on how your default `Object Explorer` is set up (see Section 4.2.3), once you are inside S-Plus you will sometimes need to tell the `Object Explorer` where your `.Data` area is located so that its objects will actually appear in the explorer. Right mouse click while in the `Object Explorer` left pane, and select `Filtering`. Then click on your `.Data` area in the `Databases` window and click `OK`.

To quit S, simply type `q()` from the command line (i.e., after the > prompt), or click on `File` → `Exit` under Windows. Do not exit by clicking on the `X` in the upper right corner as this may not close all of the files properly.

To execute DOS commands while under S-Plus use the `dos` function or `!`. Under R use `system()`, and under the Hmisc library use `sys()` on any platform. For example, you can list the contents of

[4]In S-Plus 2000 or earlier on Windows, `.Data` is `_Data`.

the current working directory by typing `!dir`. To execute Windows programs, use the `win3` function. The Hmisc library comes with a generic function `sys` that will issue the command to UNIX or DOS depending on which system is actually running at the time.

See Chapter 13 for methods of running S-Plus in batch mode.

1.3 Commands vs. GUIs

Windows S-Plus is built around a menu-based point–and–click graphical user interface. This kind of interface is especially useful for analysts who use S-Plus less than once per week as there are no commands to remember. However, relying solely on the GUI has disadvantages:

1. You can't do simple computations such as $\sqrt{5}$.
2. You may want to do further calculations on quantities computed by using a menu or dialog box, but the dialogs are designed to produce only a single result. If for example you want to compute 2-sided P-values for a series of z-statistics, the distributions dialog box may only provide 1-tailed probabilities.
3. There are many commands and options that have not been implemented in the GUI.
4. If you produce a complete analysis and then new data are obtained, you will have to re-select all the menu choices to re-run the analysis[5].

It is difficult to decide how to learn S-Plus because of the availability of both the graphical and the command interface. Because of the richness of the commands, the fact that GUIs are not available for add-on libraries, and the ability to save and re-use commands when data corrections or additions are made, we will emphasize the command orientation. To introduce yourself to the GUI approach, invoke the `Visual Demo` from the `Help` menu tab while S-Plus is running, or from the S-Plus program directory. At first, go through the `Data Import`, `Object Explorer`, `Creating a Graph`, and `Correlations` demonstrations. Also read Chapter 2 of the online *S-Plus User's Guide* and go through the Tutorial in Chapter 3. To access built-in example datasets for the tutorial, press `File ... Open` and look for the `Examples` object explorer file (`.sbf` file) in for example the `\splusxx\cmd` directory.

1.4 Basic S Commands

In its simplest form, the S command window serves as a fancy hand calculator. In the examples below S expressions are entered after the command prompt (>). For Windows S-Plus you must first open the Commands window by clicking on its icon, which looks like:

```
>
>x|
```

[5] It is possible to save the commands produced by the dialogs and re-run these, but not all commands will run properly in non-interactive mode, and the automatically generated commands are verbose.

Results are displayed following the command line. Results are prefaced with a number in brackets that indicates the element number of the first numeric result in each line. As the following commands produce single numbers, these element numbers are not useful. Later we will see that when a long series of results spanning several lines is produced, these counters are useful placeholders. Also note that comments (prefaced with #) appear below.

```
> 1+1
[1] 2
> 1+2*3+10 # note multiplication done before addition
[1] 17
> sqrt(16)
[1] 4
> 1+2^3    # note exponentiation (2 to the 3rd power) done first
[1] 9
> 1+2^3*7  # exponentiation done first, addition last
[1] 57
> 2*(3+4)
[1] 14
> 2*(3+4)^2
[1] 98
> x ← 4  # store 4 in variable x
> sqrt(x)-3/2
[1] 0.5
```

Even though S is useful for temporary calculations such as those above, it is more useful for operating on variables, datasets, and other objects using higher-level functions for plotting, regression analysis, etc.

The following series of S commands demonstrate a complete session in which data are defined, a new variable is derived, two variables are displayed in a scatterplot, two variables are summarized using the three quartiles, and a correlation coefficient is used to quantify the strength of relationship between two variables.

```
> # Define a small dataset using commands rather than the usual
> # method of reading an external file
> Age    <- c( 6,  5,  4,  8, 10,  5)
> height <- c(42, 39, 36, 47, 51, 37)
> Height <- height*2.54    # convert from in. to cm.
> options(digits=4)        # don't show so many decimal places
> Height                   # prints Height values
[1] 106.68  99.06  91.44 119.38 129.54  93.98
> plot(Age, Height)
> quantile(Age, c(.25,.5,.75))
 25% 50% 75%
   5 5.5 7.5
> quantile(Height, c(.25, .5, .75))   # also try summary(Height)
   25%   50%   75%
 95.25 102.9 116.2
> cor(Age, Height)
[1] 0.9884
> cor.test(Age,Height)
```

```
    Pearson's product-moment correlation

data:  Age and Height
t = 13.03, df = 4, p-value = 0.0002
alternative hypothesis: true coef is not equal to 0
sample estimates:
    cor
 0.9884
```

1.5 Methods for Entering and Saving S Commands

Once can choose from many approaches for developing S-Plus code, entering code interactively, and saving code that runs successfully. A few of these are as follows.

1. You can enter commands to S one at a time, directly at the command prompt. Command recall and editing (using ↑ and ↓ keys and within-line editing through the use of the `Home` and `End` keys on Windows, for example) can be of great help in correcting statements. Typing `Enter` while the cursor is anywhere inside the command will cause that line to be executed.

2. Commands can be written in an editor window (Notepad, Emacs, Xemacs, Word, Xedit, PFE, WinEdt, UltraEdit, NoteTab, etc.) and then you can highlight/copy/paste desired commands into the S-Plus command window. You can also save the file every time you edit it, and bring it into S-Plus using the `source` command. You can save typing by doing something like:

   ```
   k ← 'c:/mydir/myprog.s'
   source(k)       # input code to S-Plus
   # Move to edit window and save
   source(k)       # redefine code to S-Plus
   # Or use Hmisc's src function:
   src(myprog)     # note absence of quotes and of .s
   # Move to edit window and save
   src()           # redefines myprog.s to S-Plus
                   # file name remembered by src
   ```

   See Section 1.5.1 for details on file name specification.

3. You can run the Emacs or Xemacs ESS package with its own interactive S window (especially in Linux/UNIX) to edit the code in an Emacs window and easily execute parts of the code.

4. After entering commands interactively, selected commands (and possibly their output) can be highlighted in the S command window and pasted into an editor window.

5. After entering commands interactively, the S-Plus History log can be copied to a file.

6. If your code is contained in an S function, you can have S edit the function.

```
myfunction ← edit(myfunction)
```

You may want to override the default editor, using `options(editor = 'editorname')`. [6]Under Windows you can also specify the editor using `Options ... General Settings ... Computations`. You can also use the `edit` function to edit objects. This is especially handy for character strings. In the following example, the `levels` of a categorical variable are changed interactively:

```
levels(disease) ← edit(levels(disease))
```

A major problem with the use of `edit` is that if a function contains syntax errors you will lose any changes made.

7. The `fix` function is an easier to use version of `edit` for editing functions and other objects:

```
fix(myfunction)    # assigns result to myfunction
fix()              # edit myfunction again - also allows
                   # editing of file used in previous invocation
                   # of fix when file contained syntax errors
```

When first learning S-Plus, method 1 is very expeditious. After learning S-Plus, method 2 has some advantages. One of these is that multiple-line commands that are not part of functions can easily be re-executed as needed. Windows S-Plus has a builtin script editor which includes a facility for syntax checking code before it is submitted for execution. It also provides for easy submission of selected statements for execution after they are highlighted in the script editor.

One of the advantages of saving all the S code in a file is that the program can be run again in batch mode if the data or some of the initial commands change.

For managing analysis projects we have found it advantageous to have a "History" file in each project directory, where key results and decisions are noted chronologically. The History file can be constructed by copying and pasting from a batch output listing file or from the command window if using S-Plus interactively. Other options for saving pieces of output include the `sink` function described in Section 3.5.4, and running the program in batch mode as described in Section 13.1.

As alluded to above, Windows S-Plus has a new option for entering and editing code and saving results. You can open an existing "script" file (suffix `.scr`) by clicking on `File : Open...` or start a new one by clicking on `File : New`. You can submit code for execution using the `F10` key. If you highlight code, `F10` will cause only the highlighted code to be executed. Otherwise, the entire program will be executed. You can also highlight a function name (if it is a built-in function), right click, and select `Help` to see that function's documentation. By default, results will be displayed in a lower part of the window showing your code. You may want to drag the horizontal bar separating the program from its output to allow more space for the output window. You can control where results are outputted by clicking on `Options` then `Text output routing`. One place to store output is a `Report` window, which can be saved to a file in rich text format (`rtf`). Unlike

[6]For Windows Emacs you would use for example `options(editor='gnuclientw')`. This will cause the Emacs server (assuming Emacs had already been invoked before running the `edit` command) to open a new buffer containing the character representation of the object being edited, but to not return control to S until the buffer has been closed using for example `Ctrl x #`.

the lower half of the script program window, the report window has a scroll bar that makes it easy to show analyses done much earlier. After clicking on `Options : Text output routing : Report` click on `Window Tile Vertical` to see the report window alongside the program window. Another advantage of the report window is that you can copy from the graph sheet into the report.

If you want to store a program you've edited in the script program window click on `File : Save` or `File : Save As`. If you do use a suffix in the file name box, the suffix will be `.scr`. If you name a suffix such as `.s`, that suffix will be used instead. If you like `.s` to denote S programs as many users do, you will have to click on `File : Open` then select `All Files` to view non−`.scr` files.

In S-Plus for Windows, the `Script` editor does bracket, brace, and parenthesis matching and context-sensitive indenting. By default, it will also type the matching right brace when you type a left brace.

Those who want commands executed immediately (without hitting `F10`) should open a command window. The output from commands can be interspersed with the commands or they can be directed to a report window.

1.5.1 Specifying System File Names in S

In UNIX, directory levels are separated using `/` both at the system prompt as well as inside S. In Windows, file names use \ outside of S (e.g., when defining shortcuts or in pop-up windows from the S-Plus `File ...` menu). Inside Windows S you must use \\ inside quoted file names. You can also use `/` (single slash), as S is kind enough to translate `/` to \\. \\ is used instead of \ because inside a quoted character string \ is considered an "escape" character that modifies the meaning of another character. For example, the character string `'\n'` is a `newline` character.

1.6 Differences Between S and SAS

Four of the most important distinctions between S and SAS are (1) the S language was designed to be extendable; (2) it is very easy for users to write their own S functions; (3) SAS graphics require a large amount of programming, are non-interactive, are inflexible, and have poor appearance; and (4) SAS is much more efficient than S for analyzing very large databases. On (1), S makes it very easy for users to add to the basic S language. For example, they can add new operators and new data attributes such as `comment` attributes for variables or data frames and flags to mark that some values are imputed.

Regarding (2), when SAS first began to be widely used around 1969, it was very easy for users to write their own procedures in Fortran. They could easily define the notation to be used for their new `PROC` statement, and read SAS datasets using Fortran. Many users wrote SAS procedures, including Harrell's `PROC`s `PHGLM` and `LOGIST`, which gave SAS the capability to fit logistic and Cox regression models in 1978 and 1979, respectively. In the late 1980's, SAS converted to a new mode for writing procedures, first in PL/I then in C. The interface became much more difficult to program, and in fact SAS started selling the interface as a separate product (the SAS Toolkit). So not only did all old SAS procedures written by users all over the U.S. become obsolete, but users had great difficulty in writing add-on procedures. On the other hand, the most basic S language texts tell you how to write your own functions, in the same language that S and S developers use. Within your own functions you can also call Fortran or C subroutines extremely easily. As a result, modern statistical methods are available in S long before they become available in SAS if at all.

In terms of ease of learning, anecdotal reports indicate that S is easier to learn than SAS for users who don't already know SAS. For previous SAS users, the vector and interactive programming orientation of S may take a bit of getting used to.

The following table compares SAS with S in several areas.

Table 1.1: *Comparisons of SAS and S*

Feature	**SAS**	**S**
Numeric value storage	Floating point, 3-8 bytes	Integer, float single, float double (4, 4, 8 bytes)
Character value storage	1-200 bytes, fixed length (although dataset may be compressed)	no limit, variable length
Variable names	Up to 8 letters (31 for Version 8), case-insensitive (case-sensitive for V8), special character possible: _	Any length, case-sensitive, special character possible: .
Variable labels	Up to 40 letters (256 for V8)	Any length, user-defined attribute
Value labels	Created by `PROC FORMAT`, stored separate from data	Intrinsic attribute stored with data in `factor` variables
Standard missing values	., check using `x=.`	`NA`, check using `is.na(x)`
Special missing values	Values `.A,...,.Z`, part of standard language	User-added attributes created automatically by `sas.get` function
Missing values in logical expressions	Treated as the smallest number, logical expression will never result in missing	Uses correct rules, e.g., `T | NA` is `T`, `F | NA` is `NA`, `NA < 50` is `NA`

Feature	**SAS**	**S**
User-defined attributes	Not possible	Added at will. Examples: `comment(x) ← 'Variable was corrected 4/1/97'` `is.imputed(x)`, partial dates, name of image file containing page of data form where variable was entered
Processing of Observations	Record by record	As vectors or matrices
Dataset format	dataset = rectangular table	data frame = list of vectors and matrices; can attach attributes to data frame
By-processing	Run `PROC SORT` then use `BY` statement on `PROC` to group analysis	Execute functions in a loop, for different subsets (subscripts) of observations, or use `tapply` or related functions
Post-processing of analysis output	Some printed output not available in procedure output datasets (Output Delivery System does help). Hard for user to derive secondary estimates/simulate/bootstrap.	All calculated values are stored in objects created by functions. Easy to compute other estimates or feed output to bootstrap procedure.
Handling huge datasets (e.g., 100,000 observations on 50 variables)	Limited only by disk space	Limited by memory. Will be very slow if data must be stored in virtual memory that is swapped to disk.

Feature	**SAS**	**S**
Speed	Linear in dataset size	Faster for small-moderate datasets; slower for large ones if use virtual RAM
Merging	General, efficient	General, slower
Inputting Raw Data	Flexible, reads non-standard data formats	Flexible for ASCII files
Processing steps	Separate `DATA` and `PROC` steps executed in batch mode	Line-by-line interactive, can mix data generation and analysis steps
User-written procedures	Computational modules can be written using a separate procedure (`IML`). Can not mix standard `PROC`s using this mode. Symbolic macro language can mix `PROC,DATA` steps. Macro language is harder to write and is not "live". `PROC`s are very difficult to write, and users cannot add online help files for them.	User writes functions using standard S language. No symbolic macros are needed. Commands are "live", i.e., can sense data values and attributes at time of execution. For example, the **`describe`** function has a statement like the following to give output appropriate to the type of input variable: **`if(is.category(x) | length(unique(x)) < 20) table(x) else quantile(x)`**. Easy to call C or Fortran routines from S functions. User-written online help looks builtin.
Vector and matrix operations	Available while running `PROC IML`	Intrinsic part of language

Feature	**SAS**	**S**
System Source Code	Not available	Visible for most functions by typing function name. Can learn from, adopt, modify, correct system functions.
Graphics	non-interactive, difficult to program, restrictive, ugly	interactive and batch, best statistical graphics available
Handling of categorical variables in regression models	Some procedures allow `CLASS` statement and generate dummy variables; many do not.	Dummies always automatically generated
Nonlinear effects in models	One or two procedures will generate quadratic terms; most require user to code nonlinear component variables.	All models allow general transformations of predictors directly in the model formula
Interaction effects	Few `PROC`s will generate these; users must code products (in `DATA` step) and test them manually	Automatic
Tests of nonlinearity and pooled interaction effects	Must be done manually	Automatic when using the Design library
Plot how each predictor is represented in model	Must create auxilliary datasets and program	Single statement using Design

Feature	SAS	S
Robust covariance estimation for fitted models	Macros for "sandwich" estimator available for certain models	"Sandwich" or bootstrap, with cluster sampling adjustment, available using a single statement with Design
Model validation	Not available	Single statement using Design
Computing Predicted Values	Must create dataset containing predictor settings, add to original dataset, and re-run model fit	If saved result of fitting function ("fit object") can obtain predictions for any desired predictor settings using `predict(fit,...)` or using the Design library's `Function` function
Graphical summary of model	Not available	Effect plots and nomograms with Design
Missing value imputation	`PROC MI` for linear imputations models with normal distributions	General method using Hmisc's `aregImpute` and `impute` functions
Bayesian inference	Not available	BUGS package interfaces with S
Mixed models	`PROC MIXED` for linear models has nice features for Gaussian, binary, Poisson responses	A few models are available, including nonlinear mixed models; computational properties not as good as `PROC MIXED`.

Feature	**SAS**	**S**
Penalized maximum likelihood estimation	Ridge regression for linear Gaussian models	More general penalized MLE for linear Gaussian model and binary and ordinal logistic models, with differential penalization by type of term in model
Penalized estimation with variable selection	Not available	`lasso` function in `Statlib`
Tree models (CART)	Not available	`tree` function and graphical representation
Generalized additive models	Not available	Builtin
Nonparametric smoothing	Extremely slow `PROC IML` macro; new features for V8	Builtin, variety of smoothers

The following table lists SAS procedures and corresponding S functions. In this table, `ols`,`lrm`,`psm`,`bj`,

Table 1.2: *SAS Procedures and Corresponding S Functions*

SAS Procedures	S Functions
`ANOVA`	`aov`
`REG,GLM`	`lm,glm,ols,bj,manova`
`LOGISTIC`	`glm,lrm`
`LIFEREG`	`survreg,psm,bj`
`LIFETEST`	`surv.diff,survfit,cph`
`PHREG`	`coxph,cph`
`FREQ`	`table,crosstabs,summary.formula`
	`mantelhaen.test,fisher.test,chisq.test`
`TABULATE`	`summary.formula`
`MEANS,SUMMARY,UNIVARIATE`	`mean,var,quantile,summary,describe`
`CORR`	`corr,rcorr`
`VARCLUS`	`varclus`
`PRINQUAL`	`transcan`
`BY statement`	`tapply,by,aggregate,split,summary.formula,`
	`summarize,for`

and `cph` are from the Design library, and `summary.formula, summarize, rcorr, describe, varclus`, and `transcan` are from the Hmisc library. Other functions are built-in.

1.7 A Comparison of UNIX/Linux and Windows for Running S

The UNIX/Linux operating system is a better environment for software developers because of the wide variety of tools available[7]. UNIX/Linux is also a good choice if you are processing large databases, as it is cost-effective to have a "compute server" on your UNIX/Linux network that can be used by many users for large applications[8]. Having used both UNIX and Windows extensively, we feel that UNIX (and hence Linux) is a more efficient and reliable environment for every day S users, as UNIX window navigation is more efficient than Windows. Windows users tend to spend too much time navigating menus and Windows operates significantly slower than Linux because of the design and massive size of Windows operating systems. However, the greatest advantage of UNIX is probably that a nice system administrator would have already installed the tools you need, including Emacs, Ghostview, LaTeX, and a variety of print utilities. Versions of Linux such as RedHat and Mandrake come with all of these tools automatically. But Windows has a few advantages also: (1) ease of installing add-on S-Plus and R libraries; (2) faster online help for S-Plus; (3) outputting graphs in Windows metafile and other formats for easy inclusion and editing

[7]Windows users can not-so-easily install versions (e.g., GNU) of many of the UNIX tools, such as a `bash` shell command window.

[8]You can also program a UNIX system to compress large databases that haven't been read in a week. That way your disks will not fill up nearly as quickly.

using Microsoft PowerPoint or Word[9]; and (4) only the Windows version of S-PLUS has menus for doing standard analyses and graphics. S-PLUS 6 is available for UNIX, Linux, and Windows. This has resulted in a partial convergence of Linux/UNIX and Windows S-PLUS, with a more or less a common graphical user interface[10].

See `http://hesweb1.med.virginia.edu/biostat/s/linux.setup.html` for more information on setting up a Linux system and installing software of interest to data analysts.

1.8 System Requirements

For UNIX/Linux a minimum amount of RAM is 64MB. For PCs, 128MB is minimal. If you will be analyzing large databases (roughly speaking, > 40000 observations), you may need at least 256MB of RAM. For analyzing very large databases (say > 100000 observations), more than 256MB of RAM will usually be needed. Windows 2000 and XP use memory much more inefficiently than earlier versions of Windows, so add more RAM accordingly. RAM is cheap, so it's best to order your PC with 256MB. If you have only occasional need for more than 256MB of RAM, you may want to endure the slowness of virtual memory for those applications.

A minimum PC CPU for running Windows S-PLUS is a 400 MHz Pentium. R requires less memory to run than S-PLUS.

1.9 Some Useful System Tools

There are several system tools that can greatly assist the S user. UNIX users usually have an advantage in that their system administrator would have already installed most of the tools, and many linux packages come with all of the important tools pre-installed. For Windows users, Web addresses for obtaining the software are provided. `hesweb1.med.virginia.edu/s/EmacsTeX` has a large amount of information on obtaining an installing `Emacs`, LaTeX, and related programs.

Emacs editor: Emacs is an incredibly powerful editor for editing text files of various types. Emacs is especially powerful for editing S code, as it has a special mode which highlights different kinds of S statements in different colors or fonts and it does indentation according to the level of nesting. It also makes it easy to check for matching parentheses, brackets, and braces. Emacs for Windows (all 32MB of it when uncompressed!) is available from ftp://ftp.gnu.org/-gnu/windows/emacs/latest. Harrell's version of the Emacs startup file (`.emacs`) is available from the `Utilities` area of the UVa Web page This `.emacs` file has several useful default settings for how Emacs operates. S-mode for Emacs using the ESS Emacs package may be obtained from `http://software.biostat.washington.edu/statsoft/ess`. S-mode can also run S-PLUS or R itself, allowing for such capabilities as object name completion in the editing window if you enter the first few letters of an object's name. This mode is known to work well under UNIX/Linux.

[9]Windows S-PLUS can output graphics directly into Powerpoint Presentation format as well as Adobe Acrobat `.pdf` files (see below), and R can make `.pdf` files. Note however that using Windows metafiles to include graphics into Microsoft Office applications frequently does not preserve all aspects of the graphics. Postscript is still the most reliable graphics format.

[10]This version is based on the version 4 engine of the S language, which will require some functions to be modified unfortunately. All modifications have been made in Harrell's libraries.

Windows users may find that **Xemacs** is a bit more user-friendly, and Xemacs has a menu for automatically downloading and installing packages such as ESS. Like Emacs, Xemacs can be automatically installed when you install Linux. Windows users may obtain Xemacs from `www.xemacs.org`.

Ghostview: This is a previewer for postscript graphics and documents. It is available for Windows from `http://www.cs.wisc.edu/~ghost/`. Ghostview comes with Ghostscript, which can convert postscript files to `.pdf` files (but not as efficiently as Adobe Acrobat) among other things.

LaTeX: This system is excellent for composing technical documents and advanced tables. It is the typesetting system used to make this document, and it is used by many book publishers. An excellent commercial version of LaTeX for Windows can be obtained by contacting Personal TeX Inc. at `texsales@pctex.com` or `http://www.pctex.com`. If you want to be able to produce electronic documents (e.g., `.pdf` files) with hyperlinks, the full TeX package from Y&Y Inc. is recommended. See `www.YandY.com`. Perhaps the best versions of LaTeX for Windows are free versions, FPTeX by Fabrice Popineau and MikTeX, both available at `http://www.ctan.org`. FPTeX's `DVI` previewer allows postscript graphics to be displayed, assuming you have installed `Ghostscript`. Several tools for creating `.pdf` files are also included in FPTeX. See http://ctan.tug.org/tex-archive/info/lshort/english/lshort.pdf for a nice free book for learning LaTeX.

Adobe Acrobat Reader: Available from `www.adobe.com`, this free program nicely displays `.pdf` files. You can create these graphics files directly in Windows S-Plus using the `pdf.graph` device function. Occasionally this will get around printer memory problems when printing complex graphs, and a few graphs can only be faithfully printed in Windows this way.

Metafile Companion: This program, for which a free trial version is available from `www.companionsoftware.com`, allows you to edit Windows metafiles, a graphics format you can produce either with the `dev.print` function in S 3.2+ or using the `File ... Export Graph` dialog. `Metafile Companion` is one of the nicest graphics editors available anywhere. It allows you to edit any detail of the graph.

Mayura Draw: This shareware program is a nice scientific drawing program. It can take as input an Adobe Illustrator file, which can be converted by `Ghostscript` from a postscript file. Using that combination of programs gives you the ability to nicely edit postscript graphs. See `www.mayura.com` for information about `Mayura Draw`.

graphviz: This is an amazing command language from AT&T for drawing complex tree diagrams. Linux, UNIX, and Windows versions are available from `http://www.graphviz.org`

Xmouse: You can make a Windows 95 mouse work like a mouse in UNIX X-windows by installing Microsoft's PCToys package and running its `Xmouse` program. That way when you move the mouse from an editor window to the S command window you do not need to click the left mouse button to make the S window have the mouse's focus. This really helps in copying text from the editor to S. Also, if you had to click the left mouse button, the editor window would usually disappear. For Windows 95, obtain `Xmouse` from the Powertoys package at `www.microsoft.com/windows95/downloads/contents/wutoys/w95pwrtoysset`. For

Windows 98, this functionality is in the `tweakUI` package that is an optionally installed component of the Win 98 installation disk. With Windows 98 `tweakUI` you can also specify an option to have the "currently focused on window" automatically move to the top.

UltraEdit: Users who want a powerful programmer's editor that is not as comprehensive (or as large) as Emacs may want to consider buying UltraEdit (`www.idmcomp.com`).

WinEdt: Next to Emacs this is probably the best editor for Windows/NT users, especially when used in conjunction with LaTeX. Trial and licensed copies may be ordered from `www.winedt.-com`.

NoteTab: This is a nice editor for Windows that has a flexible macro language for making the editor language-sensitive and allowing submission of code to an open window (using `ctrl-space` (repeat last macro)). A free version is available from `www.notetab.com`. Dieter Menne (`<dieter.menne@menne-biomed.de>`) wrote the following macros for using NoteTab with R.

```
^!FocusDoc
;Save the file if it has been modified
;^!Save
;Select the highlighted block.
^!If ^$GetSelSize$ = 0 END ELSE SelectLines

:SelectLines
:GetSelection
^!Set %AnyText%=^$GetSelection$
;Write the selected text to a temporary file in the Windows temp. dir.
^!Set %fileName%=^$GetTmpPath$std0001.r
^!Set %fileName%=^$StrReplace(\;/;^%fileName%;True;False)$

^!TextToFile ^%fileName% ^%AnyText%
; Copy "source" to the clipboard
^!SetClipboard source("^%fileName%")
; Switch to R
^!FocusApp RGui*
;ESC to clear the Command window, paste the command, hit enter
^!Keyboard ESC
^!Keyboard CTRL+V
^!Keyboard ENTER
```

Dieter also wrote the following reg-file to start R from Windows Explorer .

```
REGEDIT4


[HKEY_CLASSES_ROOT\Directory\shell\Run R]
```

```
[HKEY_CLASSES_ROOT\Directory\shell\Run R\command]
@="\"C:\\Program Files\\R\\rw1050\\bin\\Rgui.exe\" --internet2"
```

TeXmacs: This is a WYSIWYG front-end to LaTeX for Linux and UNIX users that gives you a full equation editor. It is available from `www.math.u-psud.fr/∼anh/TeXmacs/TeXmacs.html`.

PFE: A nice small and free programmers editor is PFE which may be downloaded from `http://www.-lancs.ac.uk/people/cpaap/pfe/`. PFE is an excellent replacement for NOTEPAD even if you just use it for viewing files. If PFE is already open and you invoke it on another file, it will add the new file to the list of files it is currently managing. Emacs can do this using its GNUCLIENT feature. To use PFE as your default editor, you can issue the S command `options(editor='c:/pfe/pfe32')` if `pfe32.exe` is stored on the `c:\pfe` directory, or enter the `Options ... General Settings ... Computations` dialog.

Microsoft Word Damien Jolley (`djolley@ariel.ucs.unimelb.EDU.AU`) wrote a Microsoft Word macro that allows one to execute send highlighted code to S for execution. His macro definition follows.

```
Sub MAIN
If SelType() <> 2 Then EditSelectAll
'Select all if none current
EditCopy
SendKeys "%w1+{insert}{enter}^{F6}"
AppActivate "S-PLUS for Windows"
End Sub
```

A Word 97 version of the macro follows.

```
Public Sub MAIN()
If WordBasic.SelType() <> 2 Then WordBasic.EditSelectAll
'Select all if none current
WordBasic.EditCopy
WordBasic.SendKeys "%w1+{insert}{enter}^{F6}"
WordBasic.AppActivate "S-PLUS for Windows"
End Sub
```

To quote from Damien: "I have this stored as a macro which I can execute from a user-defined button on the Toolbar. So, when I'm ready to test my bit of code, I just click the button, and Windows switches over to S-Plus, copies the code into the S-Plus command buffer and execution takes place immediately. I use ALT-TAB to return to Word either to fiddle with the code or to save it to a text file". To enter the macro, record and then edit a macro. You start the recorder and enter a random command, then stop the recorder and give the macro a name. Then edit it to make the real macro.

John Miyamoto `jmiyamot@u.washington.edu` has a series of Word 6 macros for interfacing with S-Plus. These macros are available from the `Utilities` area under `Statistical Computing Tools` on the UVa Web page.

JED This is a nice small version of Emacs available from John Davis at `http://space.mit.edu/~davis/jed.html`.

Chapter 2

Objects, Getting Help, Functions, Attributes, and Libraries

2.1 Objects

In SAS, one has several concepts which refer to different types and characteristics of data, like data files, data views, data catalogs, format catalogs, libraries, etc. You get results from these data by using a PROC step. S has different entities representing data such as vectors, factors, matrices, data frames, lists, etc. These entities have different characteristics called *attributes* such as `names, class, dim, dimnames` etc., and we get results by applying *functions* to them. In general, any entity in S is designated by the general name of an *object*.

The names of objects in S can be of any length, and can contain digits, mixtures of lower and upper case letters, and periods. Names may not contain underscores and may not start with a digit. In some cases you will want the names to be very descriptive (e.g., `age.years`) but in other cases it's best to use a short name (e.g., `age`) and then to assign a longer label as an attribute [1].

Names in S are case–sensitive, so that vectors `age` and `Age` would refer to two different objects. This can be handy for distinguishing between various versions of the same basic information. For example, `age` might refer to the original age variable whereas `Age` might refer to age values after certain data corrections or missing value imputations.

2.2 Getting Help

Suppose we want to get help on a function, and see if it has any options that we may want to use. There are several ways to do this. A very simple one is to type `?mean` (or whatever the name of the

[1]This can be done using the `label` function which is in the Hmisc library described below, e.g., `label(age) ← 'Age in years'`. When using the `sas.get` function to convert SAS datasets to S data frames, SAS labels are automatically carried to S in this fashion.

function is). Equivalently, we could type `help(mean)`. In the case that the function contains special characters, its name should be enclosed in quotation marks, thus `help("%*%")` means help for the matrix-product function.

```
>?mean
Mean Value (Arithmetic Average)


DESCRIPTION:
       Returns a number which is the mean of the data.   A  frac-
       tion  to  be trimmed from each end of the ordered data can
       be specified.

USAGE:
       mean(x, trim=0, na.rm=F)

REQUIRED ARGUMENTS:
x:     numeric object.  Missing values (NAs) are allowed.

OPTIONAL ARGUMENTS:
trim:     fraction (between 0 and .5, inclusive) of values to  be
       trimmed  from  each  end of the ordered data.  If trim=.5,
       the result is the median.
na.rm:    logical flag: should missing values be  removed  before
       computation?

VALUE:
       (trimmed) mean of x.

DETAILS:
       If x contains any  NAs,  the  result  will  be  NA  unless
       na.rm=TRUE.   If  trim is nonzero, the computation is per-
       formed to single precision accuracy.
```

When you use either of these two forms of help, the system looks for a file in some directory and then displays the help file. This means that a window will pop up with options to print the help file, search for character strings, etc.[2] If you are running on a UNIX workstation, you may want to initiate the interactive help system. Type `help.start()` and a window listing all functions and categories of functions will appear. Just click on the one you want help about, and a new window will pop-up with help specifically on that function. You can then look at it, close it to keep it around or send it to the printer. With this method you can also type something like `regres*` in the topic field of the help window, to get a list of all functions which start with 'regres'. The disadvantage is that this is slower. To quit the window type `help.off()`.

A third way, if you don't want full help but to just be reminded of what the arguments to the function are, is to use the `args` function built in to S.

[2]Under UNIX X–Windows it is beneficial to use e.g. `options(pager='xless')` to use a full–screen pop–up window instead of the system default in which the `less` command is run inside of the S command window.

```
> args(mean)
function(x, trim = 0, na.rm = F)
```

The function has three arguments, `x` for the vector of which we want the mean, `trim=` if we want trimmed means, `na.rm=`, to remove missing values. The defaults are `trim=0` and `na.rm=F`. Here `T` is the logical true value, so we interpret `na.rm=T` as saying that the `na.rm` argument is turned "on." If you name the arguments, they can be given in any position. For example `mean(x,na.rm=F,trim=.5)`. See Section 2.3 for more about functions and arguments.

You can also use `names(functionname)` to list the arguments, or `functionname$argumentname` to list the default argument value. A quick way to get an alphabetic listing of a function's arguments is to type `sort(names(function.name))`. Note that there is an extra element with a blank name that should be ignored.

```
> sort(names(mean))
[1] ""      "na.rm" "trim"  "x"
```

The object orientation of S can make it difficult to know the full name of the function you are really using. For example, if you need help in plotting a logistic regression fit using the Design library, you may not know that the pertinent `plot` function is `plot.Design`. You can get a list of all of the `plot` methods by typing `methods(plot)`. You can get a list of all of the methods for handling the fit object by typing `methods(class=class(f))` if the fit object is `f`.

If you are having troubles understanding what the function does or how it is doing things, you can always look at the function itself.

```
> mean
function(x, trim = 0, na.rm = F)
{
        if(na.rm)
          x <- x[!is.na(x)]
        else if(any(is.na(x)))
          return(NA)
        if(mode(x) == "complex") {
                if(trim > 0)
                  stop("trimming not allowed for complex data")
                return(sum(x)/length(x))
        }
        x <- as.double(x)
        if(trim > 0) {
                if(trim >= 0.5)
                  return(median(x))
                n <- length(x)
                i1 <- floor(trim * n) + 1
                i2 <- n - i1 + 1
                x <- sort(x, unique(c(i1, i2)))[i1:i2]
        }
        sum(x)/length(x)
}
```

Yet another possibility is to look at the help files without even starting S-PLUS. You may find yourself in this situation if you are running a job in batch mode and want to find out why it didn't work.

In UNIX it's easy to define shell programs to facilitate this, as well as to list help files associated with keywords. Under Windows, you can use Explorer or My Computer to click on a `.hlp` file in the main S-Plus area or in an add–on library area (see below).

Last, but not least, consult the back of the blue book or the S-Plus User's manual. The help here is exactly the same as the on-line help but not all functions are listed. In S-Plus Version 4.x and later the manuals are online with some search capability.

The following is a list of major help topics for S-Plus as it is distributed from MathSoft. This list will help in understanding the components of the system as well as how you can find a function when you don't know its name. In Windows you could click on any of these topics to see all functions related to that topic. In UNIX you use the `help.start()` command to put up the list of topics.

Add to Existing Plot
All Datasets
ANOVA Models
Categorical Data
Character Data Operations
Clustering
Complex Numbers
Computations Related to Plotting
Customizable Dialog functions
Customizable Menu functions
Data Attributes
Data Directories
Data Manipulation
Data Types
Dates Objects
Demo Library
Demonstration of S-PLUS
Deprecated Functions
Documentation
Dynamic Graphics
Error Handling
Graphical Devices
High-Level Plots
Input/Output–Files
Interacting with Plots
Interfaces to Other Languages
Library of Chapter 11 Functions from The New S Language
Library of Chronological Functions
Library of Drawing Functions from Programmer's Manual
Library of Examples from Programmer's Manual
Library of Examples from The New S Language
Linear Algebra
Lists
Loess Objects
Logical Operators

Looping and Iteration
Mathematical Operations
Matrices and Arrays
Methods and Generic Functions
Miscellaneous
Multivariate Techniques
Non-linear Regression
Nonparametric Statistics
Optimization
Ordinary Differential Equations
Printing
Probability Distributions and Random Numbers
Programming
Quality Control
Regression
Regression and Classification Trees
RELEASE NOTES
Robust/Resistant Techniques
Smoothing Operations
S-PLUS Session Environment
Statistical Inference
Statistical Models
Survival Analysis
Time Series
Trellis Displays Library
Utilities

2.3 Functions

You are starting to see that unless you are using the pull–down menu system in S-Plus, almost everything is done by calling functions[3]. A function is an object in S and in many ways it can be operated on as data. Most functions have *arguments* that pass values to the function for it to work on or to specify detailed options on how it should do its work. It is common for example to pass a vector of data (representing a single variable) to a function along with scalars or other shorter vectors specifying options such as confidence levels, quantiles, plotting and printing options, etc.

Arguments are given to the function either by name or by their sequential position in the series of arguments. It is very common to specify a "major" argument without its name, in position one, then to specify "minor" arguments by name. This is because there are so many "minor" arguments and it is hard (and risky) to try to remember their order. For example, we can compute the mean `age` using the command `mean(age, na.rm=T)`, which means to compute the mean of the `age` vector ignoring missing values. We could use the equivalent statement `mean(age, , T)`, i.e., we can assign the logical "true" value (`T`) to the third argument to `mean`, which we can see from `mean`'s help file is the `na.rm` argument. The extra comma is a placeholder to specify that we are not specifying the

[3]Menu choices are actually executed by secretly calling functions.

second argument which is `trim`. `trim` will receive its default value of zero. As mentioned above, this is a dangerous method so we prefer `mean(age, na.rm=T)`.

When we examined the help file for the `mean` function we saw `na.rm=F` in the list of arguments. This means that the default value for `na.rm` is `F`, so that `na.rm` will be assumed to be `F` if you do not specify this argument. Default values can also be vectors, lists, matrices, and other objects as the need arises. Often you will see that the default for an argument is a vector of values when the argument really needs to be a scalar. In these cases, the vector of values frequently specifies the list of possible values of the argument, with the default value listed first. For example, look at the argument list for the `residuals.lm` function:

```
> args(residuals.lm)
function(object, type = c("working", "pearson", "deviance"))
```

Here the `type` argument can take on three possibilities. If you do not specify `type`, `'working'` residuals will be computed.

2.4 Vectors

A statement to create a vector interactively could be something like this

```
> x ← c(3.1,2.6,3.4,5.9,7.6)
```

In creating `x` we used two S operators, the assignment statement "←" which is read "x gets ..." and the concatenation function `c()`. A synonym for ← is the underscore sign (_). Of course the assignment could have been written in a reversed way,

```
> c(3.1,2.6,3.4,5.9,7.6) → x or
> c(3.1,2.6,3.4,5.9,7.6) _ x
```

Two or more assignments could be made on the same line if separated by a semicolon. A line could also be split among two or more lines. Just hit `return` at the end of your line and you will get a continuation prompt `"+"` at the beginning of the next line, then continue typing. You can concatenate two or more existing vectors and include other data as an argument to the `c()` function.

```
> y ← 10.6:2.3;z ← c(x,c(1,2,3),y
+ y^2)
Syntax error: name ("y") used illegally at this point:
z ← c(x,c(1,2,3),y
y
```

Here, we forgot the comma after y on the first line.

```
> y ← 10.6:2.3;z ← c(x,c(1,2,3),y,
+ y^2,y+1)
```

If we want to see what's stored in the vectors `y` and `z` just type their names

```
> y
[1] 10.6  9.6  8.6  7.6  6.6  5.6  4.6  3.6  2.6
> z
```

```
 [1]   3.10   2.60   3.40   5.90   7.60   1.00   2.00   3.00  10.60   9.60
[11]   8.60   7.60   6.60   5.60   4.60   3.60   2.60 112.36  92.16  73.96
[21]  57.76  43.56  31.36  21.16  12.96   6.76  11.60  10.60   9.60   8.60
[31]   7.60   6.60   5.60   4.60   3.60
```

There are several things to notice here. First, the operator `a:b` produces a sequence from `a` to `b` starting with `a` and adding (or subtracting) 1 to each element until you get a number greater in absolute value than `|b|`. (You may want to experiment to see what happens when `a` or `b` are negative). Second, we have `y^2` which just squares each element of y. All functions which return a single numerical result from a single numerical argument such as `exp,sqrt,sin,cos,tan,atan,log`, etc. act on each element of the vector. Finally, adding a number to a vector just adds the number to each component of the vector.

What happens if we add two vectors of different length? Let's see.

```
> x ← 1:9
> y ← 1:10
> x+y
 [1]  2  4  6  8 10 12 14 16 18 11
Warning messages:
  Length of longer object is not a multiple of the length of the shorter object
        in: x + y
```

When adding (or subtracting) two or more vectors of different length, the shorter vectors are recycled until they reach the length of the longest vector and then the operation is performed and a warning message is issued. Also notice that we did not assign the result of the sum, but printed it directly instead.

To list vectors left over from a previous session, use `objects()`. To delete them, use `rm(x,y,z)` where `x`, `y` and `z` are the vectors to be deleted. This function works in exactly the same way with objects other than vectors. You can also use the more versatile `remove` function to delete objects, e.g., `remove(c('x','y','z'))`.

Next, let us do some statistics on these vectors. How many observations do we have? What is the mean? And the standard deviation?

```
> length(z)
[1] 35
> mean(z)
[1] 17.384
> sqrt(var(z))
[1] 26.59949
```

2.4.1 Numeric, Character and Logical Vectors

All elements of a vector must be of the same type, that is integers, real numbers, complex numbers, logical values (T or F), or character strings. Examples of each kind are `c(3,6,9)`, `c(1+2i,.2,-3-5.6i)`,`(T,T,F)` and `c("x","y","z")`. To determine what kind of vector we have, we could type `mode(x)` and this will return a character string telling us the kind of vector. It is also possible to assign a value to the mode of a vector forcing it to be something else.

```
> x ← c(3.1,2.6,3.4,5.9,7.6)
> x
[1] 3.1 2.6 3.4 5.9 7.6
> mode(x)
[1] "numeric"
> mode(x) ← "character"
> x
[1] "3.1" "2.6" "3.4" "5.9" "7.6"
>
```

There are a number of functions to test for the mode of a vector and to change it. In general, if we try to operate on a vector whose mode is not appropriate for that kind of operation, S will automatically convert it to another kind trying to lose the least possible amount of information in the process. Thus, `c(T,F)+c(3,4)` yields `c(4,4)` (`F`s are converted to zeros and `T`s are converted to ones). The functions to test and change modes are

```
is.numeric,as.numeric
is.character,as.character
is.logical,as.logical
```

A useful function in the Hmisc library which may save you some typing is `Cs(a,b,c,d)`. It is equivalent to `c("a","b","c","d")` but it won't work if your character strings have an _ in them (since _ is equivalent to ←).

2.4.2 Missing Values and Logical Comparisons

Missing values in numeric and logical vectors are represented by the symbol `NA` (not available). In general, any operation (mathematical or logical) performed on a missing value will return a missing value. The logical operators are `>, >=, <, <=, ==, !=, &, |,!`. Notice that the operator to test equality is `==` rather than `=`, which is reserved for named arguments to a function. `!` is used for negation and `&` and `|` for logical 'and' and 'or'. Consider for instance

```
> x ← c(3,6,9,10,2.2,NA,NA,6.7);  y ← c(1,6,9,2,NA,5.1,0,-1)
> x > y
> [1] T F F T NA NA NA T
```

The operator `==` is not appropriate to test for missing values. Instead, use the function `is.na`.

```
> is.na(x > y)
[1] F F F F T T T F
```

Suppose that we have two vectors of the same length, and we want to know the joint distribution of their missing values.

```
> x
 [1]  1  1  1 NA NA  2  2  2  2  2  2 NA
> y
 [1]  2  2  2  2  2 NA  4 NA  1  1  1  1
```

One way would be to use the table function

```
> table(is.na(x),is.na(y))
      FALSE TRUE
FALSE     7    2
 TRUE     3    0
```

You can also tabulate all patterns of NAs using the builtin function `na.pattern` (but note that `na.pattern` was omitted from S-Plus 2000):

```
> na.pattern(list(x,y))
 00 01 10
  7  2  3
```

Also see the `naclus` function described under the `varclus` function in the Hmisc library discussed below.

2.4.3 Subscripts and Index Vectors

It is possible to select subsets of a vector by *subscripting* or *indexing* its elements. This is equivalent to using a `WHERE` statement in SAS, but it is more flexible. The expression to use is `x[i]` where `i` could be another vector, or an expression which evaluates to a numeric, logical or character vector. In all cases, we'll think of the elements of `x` as being subscripted by the indexes `1:length(x)` when `[ ]` is not present.

1. If `i` is a numeric vector, all its elements must be `>=0` or all `<=0` (`NA`s are allowed). Before selecting the subset, S drops all zeros from the index vector. If all elements of `i` are positive, then `x[i]` selects only those elements of `x` whose subscripts match the elements of `i`. If the elements of `i` are negative, then `x[i]` selects the elements of `x` whose subscript *does not match any element* in `i`. If the k*th* element of `i` is `NA` then the k*th* element of `x[i]` will be `NA` as well. (0s are ignored). `i` can be any length.

2. If `i` is a logical vector, it is indexed starting at 1 and those elements of `x` whose subscripts have a value of `T` in the corresponding index of `i` are selected. The same rule as in 1. apply to `NA`s. For this case, the length of the index vector should equal `length(x)`.

3. If `i` is a character string (of any length), the rules are a little bit different. In this case `x` is required to have what's called a *names* attribute. A `names` attribute is a vector of character strings of the same length as `x` which effectively names each element of `x`. Assuming that `x` already has a `names` attribute, the expression `x[c("a","b")]` selects the first element of `x` named `a` and the first element named `b`. We will talk more about names when we discuss attributes in general.

Examples:

```
> x
[1]  3.0  6.0  9.0 10.0  2.2   NA   NA  6.7
> y
[1]  1.0  6.0  9.0  2.0   NA  5.1  0.0 -1.0
> x[3]
[1]  9
```

```
> x[1:3]
[1]  3  6  9
> x[-2]
[1]  3.0  9.0  10.0  2.2   NA   NA  6.7
> x[c(F,T,T,F,F,F,F,F)]
[1]  6.0  9.0
> x[x > y]
[1]  3.0 10.0   NA   NA   NA  6.7
> x[!is.na(x)]
[1]  3.0  6.0  9.0 10.0  2.2  6.7
> z ← x[!is.na(x)]  # get rid of missing values
```

It is instructive to look at the help file for the subsetting operator `"["` (type `?"["`) and work out some examples. This is a very useful function that you will be using all the time, but is also very easy to get confused and end up selecting values that you didn't mean to select. Try to always check that you have the right vector by using the `length` function.

For a simple example of character indexing, let's create a simple named vector.

```
> w ← c(cat=1, dog=3, giraffe=11)
> w['cat']
[1]  1
> w[c('cat','giraffe')]
[1]  1  11
```

2.5 Matrices, Lists and Data Frames

2.5.1 Matrices

A collection of vectors may represent several different variables in your dataset, but is not the most convenient way of handling your data. We can construct matrices by putting together vectors of the same length and the *same mode* using the functions `cbind` and `rbind`. The first one takes its arguments and puts them together as columns of a matrix, while the second one makes them into the rows of a matrix.

```
> x1 ← c(2,4,6,8,0)
> x2 ← c(1,3,5,7,9)
> x3 ← c(3,7,11,15,9)
> cx ← cbind(x1,x2,x3)
> rx ← rbind(x1,x2,x3,c(2,6,10,14,8))
> cx
     x1 x2 x3
[1,]  2  1  3
[2,]  4  3  7
[3,]  6  5 11
[4,]  8  7 15
[5,]  0  9  9
>rx
   [,1] [,2] [,3] [,4] [,5]
x1    2    4    6    8    0
```

```
x2    1    3    5    7    9
x3    3    7   11   15    9
      2    6   10   14    8
```

Notice that that the columns of `cx` are labeled and so are the rows of `rx` except for the last one, since the last argument to `rbind` was not given a name. Another way to create a matrix is to use the function `matrix(data,nrow,ncol,...)`. This function will read data in a stream from the data argument and put it in a `nrow` × `ncol` matrix in column order by default. (In fact only one of `nrow` and `ncol` is needed if `data` is of length `nrow*ncol`). The `...` represent other arguments to allow to read the data in row order and give labels to rows and columns.

A useful function to use with matrices is `apply`. It is invoked by `apply(x,margin,fun,...)` where `x` is a matrix, `margin` is the dimension over which the function is to be applied (1 for rows, 2 for columns), and `fun` is the function to be applied to the rows or columns of x.

```
> apply(cx,2,mean)
 x1 x2 x3
  4  5  9
```

gives us the means of the columns of `cx`.

Actually `apply` can be use more generally with multidimensional arrays. Other functions related to matrices are `dim, dimnames, is.matrix, ncol, nrow` and `t`. `t(x)` returns the transpose of `x`.

Matrices can be indexed in a similar way to vectors. Usually, our purpose is to select a few columns (variables we want to look at) and rows (observations) satisfying a given condition. Since we have two indexes now, we can look at both

```
> cx[2:5,c(2,3)]
     x2 x3
[1,]  3  7
[2,]  5 11
[3,]  7 15
[4,]  9  9
> cx[2:5,c("x2","x3")]
     x2 x3
[1,]  3  7
[2,]  5 11
[3,]  7 15
[4,]  9  9
```

The second example above shows another way of selecting two particular columns. Since they are named, we can just list their names in the appropriate place in the indexing bracket. If we don't want to impose any restrictions in a particular dimension, we just leave it blank. Thus, `cx[,c("x2","x3")]` lists all rows of `cx` for columns `x2` and `x3`. There are of course, a number of functions to do mathematical operations on matrices: `*`, `%*%`, `crossprod`, and `outer` which perform element by element multiplication, matrix product, cross products, and outer products, respectively on matrices of the appropriate sizes.

To most efficiently determine which rows of a matrix `x` have a column containing an `NA`, use the expression

```
is.na(x %*% rep(1,ncol(x)))
```

To subset the matrix to contain only rows with all non–missing values you can use the Hmisc `nomiss` function, e.g., `nomiss(x)`.

2.5.2 Lists

Lists are collections of objects of different kinds. The components of a list could be vectors, matrices or other lists and they can have different length and types. An example of a list is the *names* of the rows and columns of a matrix.

```
> dimnames(cx) ← list(1:5,c("x","y","z"))
> cx
   x  y  z
1  2  1  3
2  4  3  7
3  6  5 11
4  8  7 15
5  0  9  9
```

The function `dimnames` is used to name the rows and columns of a matrix and it is required to be a list, so, we used the function `list` to create it. The arguments to `list` could be anything, and they can be name just as the rows or columns of a matrix.

```
> list1 ← list(rowmatrix=rx,dimnames(cx),c("a","b","c"))
> list2 ← list(cx,indexes=1:9)
> list1
$rowmatrix:
   [,1] [,2] [,3] [,4] [,5]
x1    2    4    6    8    0
x2    1    3    5    7    9
x3    3    7   11   15    9
x4    2    6   10   14    8

[[2]]:
[[2]][[1]]:
[1] "1" "2" "3" "4" "5"

[[2]][[2]]:
[1] "x" "y" "z"


[[3]]:
[1] "a" "b" "c"

> list2
[[1]]:
   x  y  z
1  2  1  3
2  4  3  7
```

```
3   6   5 11
4   8   7 15
5   0   9   9

$indexes:
[1] 1 2 3 4 5 6 7 8 9
```

Components of a list can be selected in one of two ways: the more general method extracts the component by referring to it by its position on the list. `list2[[2]]` selects the second component of the list `list2`. If the components are named, we may select them using the expression `list$component` or `list[['component']]`. In the example above, `list1$rowmatrix` selects the matrix `rx`. Ocassionally, you may need the unlisted results. The function `unlist` serves just such purpose.

There is virtually no limit to what can be stored in a list, including other lists:

```
> us ← list(Alabama=list(counties=c('Autauga','Baldwin',
+                                     'Barbour','Bibb',...),
+        pop=4273084,capital='Montgomery'),
+        Alaska=list(counties=c('Aleutians East','Aleutians West',
+                               'Anchorage','Bethel',...),
+                    pop=602545, capital='Juneau'),
+        ...)
> us$Alabama                # Print information for one state
>                           # same as us[[1]] or us[['Alabama']]
> us$Alabama$counties       # Print counties in Alabama
> us$Alabama$counties[1:5]  # Print first 5 counties
> us[c('Alabama','Alaska')] # Print a sub-list containing 2 states
```

Section 2.6.2 provides more information on selecting elements of lists and vectors. You can see that lists provide a natural way to represent hierarchical structures.

In the above example we might as well associate some data with the counties, such as the population:

```
> us ← list(Alabama=list(counties=c(Autauga=40061,Baldwin=123023,
+                                     Barbour=26475,Bibb=18142,...),
+            pop=4273084,capital='Montgomery'),
+        Alaska=list(counties=c('Aleutians East'=2305,
+                               'Aleutians West'=5259,
+                               Anchorage=251336,Bethel=15525,...),
+                    pop=602545, capital='Juneau'),
+        ...)
> # Note: need to enclose non-legal S-Plus object names in quotes
> sum(us$Alabama$counties) - us$Alabama$pop    # should be zero
> us$Alaska$counties['Aleutians East']         # print one county's population
> us$Alaska$counties['Bethel']                 # print another
> us$Alaska$counties[c('Anchorage','Bethel')]  # print two
> Ak ← us$Alaska                               # subset of list for Alaska
> Ak$counties                                  # print Alaska county pops
```

Lists are a very convenient mechanism to summarize in one object all the information related to a particular task. Many functions give as a result a `list` object. For instance, most modeling functions produce a list whose components are quantities of statistical interest. The function `ols` in the Design library, for example, fits an ordinary least squares model and returns an object of mode list. Among its components are: the model formula, vector of coefficients, summary of missing values, and, optionally, vectors of predicted values, residuals, and the design matrix and response variable values.

2.5.3 Data Frames

Data frames are just a particular kind of list where all its components have the same length. They behave pretty much like matrices in the sense that you can operate on rows and columns and select its elements in the same way, except that the components can be of different type. You may have some columns that are character vectors and other columns that are numeric or logical vectors. Moreover, an entire matrix can be part of a data frame, as long as its columns are of the same length as the other components of the data frame. They are the most similar entity to a SAS dataset that you will find in S, and they are used most frequently in modeling situations, thinking of rows as observations and columns as variables.

There are several ways to create data frames. First, there's the `File ... Import` dialog. Second, you can read the data into a data frame from an external ASCII or SAS dataset by using the functions `read.table` or `sas.get` (to be described later), or construct it from existing objects using the function `data.frame`.

```
> obs ← Cs(id1,id2,id3,id4,id5,id6)  # Hmisc shorthand for c('id1',...)
> # Hmisc shorthand for c('id1','id2','id3','id4','id5','id6')
> treat ← c(rep("Treatment 1",3),rep("Treatment 2",3))
> treat
[1] "Treatment 1" "Treatment 1" "Treatment 1" "Treatment 2" "Treatment 2"
[6] "Treatment 2"
> x ← c(2.5,3.5,3.0,4.6,5.5,5.3)
> df ← data.frame(treat,x,row.names=obs)
> df
        treat   x
id1 Treatment 1 2.5
id2 Treatment 1 3.5
id3 Treatment 1 3.0
id4 Treatment 2 4.6
id5 Treatment 2 5.5
id6 Treatment 2 5.3
```

The argument `row.names` gives names to the rows of the data frame. If provided, its values must be unique. If it is not provided S will try to construct it from the arguments to `data.frame`. For instance, if one of the arguments is a matrix with a `dimnames` attribute, it will try to use that. If it can't find any vector to construct the row names, it will simply number them.

The Hmisc `naclus` and `naplot` functions are useful for displaying patterns of `NA`s in data frames in various ways. `naclus` also returns a vector containing the number of missing variables for each observation. `naclus` does this using the statements

```
na ← sapply(my.data.frame, is.na) * 1
na.per.obs ← apply(na, 1, sum)
```

`naclus` also returns the mean number of other variables that are missing in observations for which variable i is missing, for $i = 1, \ldots$. See also the builtin `na.pattern` function (Section 2.4.2, but note that `na.pattern` does not work correctly for `factor` variables).

Data frames may be subsetted using the same notation as matrices (see Section 2.5.1).

2.6 Attributes

We have mentioned certain characteristics of S objects that are typical of that kind of object, and others that are common to all of them. Among the latter ones we can mention the *length* and the *mode* of an object. Length is easy to describe and just counts the number of elements of a vector or matrix, or the number of major components of a list. As a data frame is also a list and its major components are variables, the length of a data frame is the number of variables it contains. The mode refers to the type of object which could be numeric, complex, logical, character (these are called *atomic* objects) or list (which are called *recursive* objects). The functions to find out these characteristics are `length` and `mode` respectively.

The other characteristics that describe an object are referred to as the *attributes* of an object. They include names, dim, dimnames, class, levels, row.names and any other that you may want to create. Corresponding to each of these attributes there is a function to extract them; thus, to know the `dim` attribute of the matrix `cx` type `dim(cx)`. To know if a particular observation is in your data frame, we could use the `row.names` attribute.

```
> row.names(df)[row.names(df)=="id9"]
character(0)
```

The result is a character vector of length zero, meaning that said observation is not in the data frame. Here you could also just print the number of observations with that `id` using the command `sum(row.names(df)=='id9')`.

In many cases the attribute determines just what kind of and object we have. For instance, a matrix (or more generally, an array) is just a vector with a `dim` attribute which allow functions such as `apply` to act accordingly. Other functions do not make that distinction and will consider it just a vector.

```
> length(cx)
[1] 15
```

Attributes can be changed or deleted

```
> dim(rx) ← NULL   # or attr(rx,'dim') ← NULL
> rx
 [1]  2  1  3  2  4  3  7  6  6  5 11 10  8  7 15 14  0  9  9  8
> dim(rx) ← c(5,4)
> rx
     [,1] [,2] [,3] [,4]
[1,]    2    3   11   14
[2,]    1    7   10    0
[3,]    3    6    8    9
[4,]    2    6    7    9
[5,]    4    5   15    8
```

rx was a 4×5 matrix. We first made it into a vector by deleting its dim attribute and then made into a 5×4 matrix by assigning a new one. One could also create a new attribute with the function attr.

```
> # For Windows use date() to get the current date as a character value
> attr(df,"creation date") ← unix("date") ;  attributes(df)
$names:
[1] "treat" "x"

$row.names:
[1] "id1" "id2" "id3" "id4" "id5" "id6"

$class:
[1] "data.frame"

$"creation date":
[1] "Wed Jun 30 10:42:29 EDT 1993"

> names(attributes(df))
[1] "names"         "row.names"     "class"         "creation date"
```

In this example, the attr function assigns a new attribute called "creation date" to the data frame df, by calling the unix command "date", for example. Next, we listed all the attributes of df using the function attributes. This, not only tells us what attributes df has, but also how they are composed. This might be too much information (specially if you have over a thousand ids in row.names). We can reduce it by typing names(attributes(...)). Notice that the attributes is in general a list with named components, which allows us to use the names function on it.

2.6.1 The Class Attribute and Factor Objects

Notice in the example above, that one of the attributes is called class. This is a very special attribute related to the concept of *methods*. When the class attribute is present, functions will act in different ways depending on the class of the object. plot will act in a different way if its arguments have a class of data.frame. As usual, the class attribute of an object can be extracted using the function class.

```
 > class(df)
[1] "data.frame"
```

They can also be unclassified by means of unclass. The result of using unclass is that df will print as a list rather than as a data frame.

```
> df
          treat   x
id1 Treatment 1 2.5
id2 Treatment 1 3.5
id3 Treatment 1 3.0
id4 Treatment 2 4.6
id5 Treatment 2 5.5
id6 Treatment 2 5.3
```

```
> unclass(df)
$treat:
[1] "Treatment 1" "Treatment 1" "Treatment 1" "Treatment 2" "Treatment 2"
[6] "Treatment 2"
$treat: Levels:
[1] "Treatment 1" "Treatment 2"

$x:
[1] 2.5 3.5 3.0 4.6 5.5 5.3

attr(, "row.names"):
[1] "id1" "id2" "id3" "id4" "id5" "id6"
attr(, "creation date"):
[1] "Wed Jun 30 10:42:29 EDT 1993"
```

(Note: there is an implicit use of the `print` function when you type `df`). Of particular interest are the objects of class "factor". A *factor* is an object with a discrete set of levels like those that arise from a classification variable. In SAS we could have a variable `x` taking `k` different values, say $x_1, \ldots, x_k$, with formatted values $l_1, \ldots, l_k$. In S this will become a *factor* object with internal numeric codes $1, \ldots, k$ and *levels* $l_1, \ldots l_k$.

The syntax for the `factor` function is

```
factor(x,levels,labels,exclude=NA)
```

`x` is of course the vector to be factored, `levels` is a vector with the unique set of values of `x` that you want to keep in the factor, and `labels` is the corresponding set of optional labels for the values of `x`. Note the very confusing fact that the `labels` specified to `factor` will become the `levels` attribute of the resulting vector. Those elements of `x` not matching any element of `levels` will be considered `NA`. The `exclude` argument is a vector of values to be excluded from forming levels. For instance, if `x` was already a vector of character strings, you may want to set `exclude` to `""` to prevent empty strings from becoming a level.

If you need to use the internal values of `x` rather than its levels for some reason, the function `unclass` comes in handy again.

```
> x ← c(2,2,2,3,3,3)
> l ← c("2","3")
> f ← factor(x,l)
> x
[1] 2 2 2 3 3 3
> unclass(f)
[1] 1 1 1 2 2 2
attr(, "levels"):
[1] "2" "3"
```

It is not possible to do mathematical transformations of a factor object. The reason is that factors represent categorical variables that may or may not be interval scaled or even ordinal. For example if `x` and `y` are factors, it does not make sense to add them.

In summary a `factor` is a categorical object with a `levels` attribute, but which is treated internally as having the values `1:length(levels(x))`. If no `levels` argument is provided, the sorted unique values of `x` are used.

2.6.2 Summary of Basic Object Types

Table 2.1 summarizes some of the types of objects we have discussed. Note that a `factor` is a special case of a vector, a matrix is a special case of an array, and a data frame is a special case of a list. The table also describes how elements are selected (subscripted) from an object named `x`. There `row` and `col` are vectors of positive, negative, or zero–valued integers, logicals, or character strings (strings are allowed when the pertinent dimension of the object `x` has a `names` or `dimnames` attribute). Zero–valued subscripts are ignored, and negative values denote "get all but the subscripts listed, suppressing their signs." When a subscript is omitted and its place is held by a comma, that means to fetch all elements of the omitted dimension. For lists and data frames, there are 3 methods for selecting elements. The first of these, `x[col]`, results in a new list or data frame containing the elements (usually variables) corresponding to `col`. The last 2 methods result in individual variables. There `colname` is the name of *one* of the elements (variables). Below, `length` is listed as an attribute although it should officially be labeled as a basic property of the object.

Table 2.1: *Comparison of Some S Objects*

Type	Description	Main Attributes
vector `x[row]`	single column of numbers (integer, single, or double precision) or character strings Usually thought of as a *variable*	**length** number of elements **names** (optional) names of elements
factor `x[row]`	categorical variable, with categories coded as integers $1, 2, 3, \ldots$	**length** no. elements **names** (optional) names of elements **class** `'factor'` **levels** vector of character strings defining labels that correspond to integer codes
matrix `x[row,col]`, `x[row,]`, `x[,col]`	rectangular table of numbers or character strings	**length** number of rows $\times$ number of columns **dim** vector of length 2 containing no. rows, no. columns **dimnames** list of length 2 containing a vector of row names (or `NULL`) and a vector of column names (or `NULL`)

Type	Description	Main Attributes
list `x[col]`, `x$colname`, `x[['colname']]`	an arbitrary collection of S objects including other lists; can be thought of as a tree; elements do not need to have equal lengths	**length** number of major elements **names** names of major elements
data frame `x[col]`, `x$colname`, `x[['colname']]`, `x[row,col]`, `x[row,]`, `x[,col]`	a rectangular dataset; a list in which all elements have the same number of rows. Each element in the list is a variable, and some of the variables may be matrices	**length** number of variables **names** names of variables **class** `'data.frame'` **row.names** row (observation) names

2.7 When to Quote Constants and Object Names

In S you can use single quotes, double quotes, or the Hmisc `Cs` function (when the symbols being quoted are legal S names) to specify character strings. Here are some general rules about use of quotes.

character constants : Character constants should always be quoted when appearing in S programs. Examples:

```
age[sex=='female']
dframe[c('patienta','patientb','patientc'),]
sex ← 'female'
```

object names, general : When a data frame or an object naming a vector or matrix is used as the *input* to a function, do not quote the name. Here are examples:

```
summary(dframe)
attach(dframe)
summary(varname)
mean(varname)
attach(dframe[dframe$sex=='male',])
summary(dframe[,c('age','sex')])
```

When giving a function the name of an object to *create*, this name is quoted (see `detach` in the following item).

data frames : When detaching search position 1 into a data frame, quote its name. E.g., `detach(1, 'newdframe')` (failing to quote data frame names in `detach` is a common problem that causes the search list to be corrupted). Otherwise, data frame names are generally not quoted.

variables : These names are generally unquoted except when used to select columns of a data frame, e.g., `dframe[,c('age','sex')]`. If you tried to use `dframe[,c(age,sex)]`, S would combine the *values* of the `age` and `sex` variables and try to use these values as *column numbers* to retrieve.

list elements : These are generally not quoted (e.g., when used with $) unless their names are not legal variable names. In that case use a statement such as `objectname[['element name']]` or `objectname$'element name'`

removing objects : Do not quote object names given to `rm` (e.g., `rm(age, sex, dframe)`). Quote a vector of character constants given to `remove`, e.g. `remove(c('age', 'sex'))`.

`get` and `assign` : These functions need object names to be quoted, but not the object representing a value for `assign` to transmit.

accessing libraries Library names are unquoted when using the `library` function. They are quoted when using `help()`.

2.8 Function Libraries

S comes with over 2000 functions, organized in the main system areas and in a library of advanced graphics functions called `trellis`, as well as other libraries. In Windows S-Plus at least, `trellis` is automatically available to the user without the need of a `library(trellis)` command. Other series of functions which are supplied with S are organized into other libraries which must be requested for attachment by the user using the `library` function. For example, to get access to advanced matrix functions you can type the command `library(Matrix)`. In version 4.5+ you can use the `File ... Load Library` pull–down menu to issue the `library` call. For libraries in need of being loaded early in the search list (i.e., those requiring `first=T`), check the `Attach at top of search list` box.

Many users have developed add–on libraries of S functions for UNIX, Windows, or both platforms. Frank Harrell has developed two freely available S libraries for UNIX and Windows that are available in the `Statlib` archive in `lib.stat.cmu.edu` or from the UVa web page. The Hmisc library ("Harrell Miscellaneous") is described in Section 2.9, and the Design library is described in Chapter 9. Once these libraries are installed[4], get access to their functions and datasets by typing

```
library(Hmisc, T)    # Reference Hmisc before referencing Design
library(Design,T)    # Design requires Hmisc to work
```

The `T` (`first=T` in expanded notation) is needed because Hmisc and Design override a few builtin functions.

Hmisc contains a family of `latex` functions for converting certain S objects to typeset LaTeX representation. The output of these functions is a text file containing LaTeX code. You can also preview typeset LaTeX files while running S.

[4]These functions are built–in to S-Plus2000 and later (on Windows only) but they still must be accessed using `library()` or `File ... Load Library`

2.9 The Hmisc Library

The Hmisc library contains around 200 miscellaneous functions useful for such things as data analysis, high–level graphics, utility operations, functions for computing sample size and power, translating SAS datasets into S, imputing missing values, advanced table making, variable clustering, character string manipulation, conversion of S objects to LaTeX code, recoding variables, and bootstrap repeated measures analysis. The help categories for Hmisc serve to describe the areas covered by this library:

ANOVA Models
Add to Existing Plot
Bootstrap
Categorical Data
Character Data Operations
Clustering
Computations Related to Plotting
Data Directories
Data Manipulation
Documentation
Grouping Observations
High-Level Plots
Interfaces to Other Languages
Linear Algebra
Logistic Regression Model
Mathematical Operations
Matrices and Arrays
Methods and Generic Functions
Miscellaneous
Multivariate Techniques
Nonparametric Statistics
Overview
Power and Sample Size Calculations
Predictive Accuracy
Printing
Probability Distributions and Random Numbers
Regression
Repeated Measures Analysis
Robust/Resistant Techniques
Sampling
Smoothing Operations
Statistical Inference
Statistical Models
Study Design
Survival Analysis
Utilities

A list of functions in Hmisc along with a brief description follows.

Function Name	Purpose
abs.error.pred	Computes various indexes of predictive accuracy based on absolute errors, for linear models
approxExtrap	Linear extrapolation
aregImpute	Multiple imputation based on additive regression, bootstrapping, and predictive mean matching
all.is.numeric	Check if character strings are legal numerics
areg.boot	Nonparametrically estimate transformations for both sides of a multiple additive regression, and bootstrap these estimates and R^2
ballocation	Optimum sample allocations in 2-sample proportion test
binconf	Exact confidence limits for a proportion and more accurate (narrower!) score stat.-based Wilson interval (Rollin Brant, mod. FEH)
bootkm	Bootstrap Kaplan-Meier survival or quantile estimates
bpower	Approximate power of 2-sided test for 2 proportions Includes bpower.sim for exact power by simulation
bpplot	Box-Percentile plot (Jeffrey Banfield, umsfjban@bill.oscs.montana.edu)
bsamsize	Sample size requirements for test of 2 proportions
bystats	Statistics on a single variable by levels of >=1 factors
bystats2	2-way statistics
calltree	Calling tree of functions (David Lubinsky, david@hoqax.att.com)
character.table	Shows numeric equivalents of all latin characters Useful for putting many special chars. in graph titles (Pierre Joyet, pierre.joyet@bluewin.ch)
ciapower	Power of Cox interaction test
cleanup.import	More compactly store variables in a data frame, and clean up problem data when e.g. Excel spreadsheet had a non-numeric value in a numeric column
combine.levels	Combine infrequent levels of a categorical variable
comment	Attach a comment attribute to an object: `comment(fit) <- 'Used old data'` `comment(fit)   # prints comment`
confbar	Draws confidence bars on an existing plot using multiple confidence levels distinguished using color or gray scale
contents	Print the contents (variables, labels, etc.) of a data frame
cpower	Power of Cox 2-sample test allowing for noncompliance
Cs	Vector of character strings from list of unquoted names
csv.get	Enhanced importing of comma separated files labels

```
cut2            Like cut with better endpoint label construction and allows
                    construction of quantile groups or groups with given n
datadensity     Snapshot graph of distributions of all variables in
                    a data frame.  For continuous variables uses scat1d.
dataRep         Quantify representation of new observations in a database
ddmmmyy         SAS "date7" output format for a chron object
deff            Kish design effect and intra-cluster correlation
describe        Function to describe different classes of objects.
                    Invoke by saying describe(object). It calls one of the
                    following:
describe.data.frame
                Describe all variables in a data frame (generalization
                    of SAS UNIVARIATE)
describe.default
                Describe a variable (generalization of SAS UNIVARIATE)
do              Assists with batch analyses
dot.chart       Dot chart for one or two classification variables
Dotplot         Enhancement of Trellis dotplot allowing for matrix
                    x-var., auto generation of Key function, superposition
drawPlot        Simple mouse-driven drawing program, including a function
                    for fitting Bezier curves
ecdf            Empirical cumulative distribution function plot
eip             Edit an object "in-place" (may be dangerous!), e.g.
                    eip(sqrt) will replace the builtin sqrt function
errbar          Plot with error bars (Charles Geyer, U. Chi., mod FEH)
event.chart     Plot general event charts (Jack Lee, jjlee@mdanderson.org,
                    Ken Hess, Joel Dubin; Am Statistician 54:63-70,2000)
event.history   Event history chart with time-dependent cov. status
                    (Joel Dubin, joel.dubin@yale.edu)
find.matches    Find matches (with tolerances) between columns of 2 matrices
first.word      Find the first word in an S expression (R Heiberger)
fit.mult.impute Fit most regression models over multiple transcan imputations,
                    compute imputation-adjusted variances and avg. betas
format.df       Format a matrix or data frame with much user control
                    (R Heiberger and FE Harrell)
ftupwr          Power of 2-sample binomial test using Fleiss, Tytun, Ury
ftuss           Sample size for 2-sample binomial test using  "  "  "  "
                    (Both by Dan Heitjan, dheitjan@biostats.hmc.psu.edu)
gbayes          Bayesian posterior and predictive distributions when both
                the prior and the likelihood are Gaussian
getHdata        Fetch and list datasets on our web site
gs.slide        Sets nice defaults for graph sheets for S-Plus 4.0 for
                copying graphs into Microsoft applications
histbackback    Back-to-back histograms (Pat Burns, Salomon Smith
                    Barney, London, pburns@dorado.sbi.com)
hist.data.frame Matrix of histograms for all numeric vars. in data frame
```

```
                    Use hist.data.frame(data.frame.name)
histSpike       Add high-resolution spike histograms or density estimates
                    to an existing plot
hoeffd          Hoeffding's D test (omnibus test of independence of X and Y)
impute          Impute missing data (generic method)
%in%            Find out which elements a are in b : a %in% b
interaction     More flexible version of builtin function
is.present      Tests for non-blank character values or non-NA numeric values
james.stein     James-Stein shrinkage estimates of cell means from raw data
labcurve        Optimally label a set of curves that have been drawn on
                    an existing plot, on the basis of gaps between curves.
                    Also position legends automatically at emptiest rectangle.
label           Set or fetch a label for an S-object
Lag             Lag a vector, padding on the left with NA or ''
latex           Convert an S object to LaTeX (R Heiberger & FE Harrell)
ldBands         Lan-DeMets bands for group sequential tests
list.tree       Pretty-print the structure of any data object
                    (Alan Zaslavsky, zaslavsk@hcp.med.harvard.edu)
mask            8-bit logical representation of a short integer value
                    (Rick Becker)
matchCases      Match each case on one continuous variable
matxv           Fast matrix * vector, handling intercept(s) and NAs
mem             mem() types quick summary of memory used during session
mgp.axis        Version of axis() that uses appropriate mgp from
                    mgp.axis.labels and gets around bug in axis(2, ...)
                    that causes it to assume las=1
mgp.axis.labels
                Used by survplot and plot in Design library (and other
                    functions in the future) so that different spacing
                    between tick marks and axis tick mark labels may be
                    specified for x- and y-axes.  ps.slide, win.slide,
                    gs.slide set up nice defaults for mgp.axis.labels.
                    Otherwise use mgp.axis.labels('default') to set defaults.
                    Users can set values manually using
                    mgp.axis.labels(x,y) where x and y are 2nd value of
                    par('mgp') to use.  Use mgp.axis.labels(type=w) to
                    retrieve values, where w='x', 'y', 'x and y', 'xy',
                    to get 3 mgp values (first 3 types) or 2 mgp.axis.labels.
minor.tick      Add minor tick marks to an existing plot
mtitle          Add outer titles and subtitles to a multiple plot layout
mulbar.chart    Multiple bar chart for one or two classification variables
%nin%           Opposite of %in%
nomiss          Return a matrix after excluding any row with an NA
panel.bpplot    Panel function for trellis bwplot - box-percentile plots
panel.plsmo     Panel function for trellis xyplot - uses plsmo
pc1             Compute first prin. component and get coefficients on
```

```
                    original scale of variables
plotCorrPrecision  Plot precision of estimate of correlation coefficient
plsmo           Plot smoothed x vs. y with labeling and exclusion of NAs
                    Also allows a grouping variable and plots unsmoothed data
popower         Power and sample size calculations for ordinal responses
                    (two treatments, proportional odds model)
prn             prn(expression) does print(expression) but titles the
                    output with 'expression'.  Do prn(expression,txt) to add
                    a heading ('txt') before the 'expression' title
p.sunflowers    Sunflower plots (Andreas Ruckstuhl, Werner Stahel,
                    Martin Maechler, Tim Hesterberg)
ps.slide        Set up postcript() using nice defaults for different types
                    of graphics media
pstamp          Stamp a plot with date in lower right corner (pstamp())
                    Add ,pwd=T and/or ,time=T to add current directory
                     name or time
                    Put additional text for label as first argument, e.g.
                    pstamp('Figure 1')  will draw 'Figure 1  date'
putKey          Different way to use key()
putKeyEmpty     Put key at most empty part of existing plot
rcorr           Pearson or Spearman correlation matrix with pairwise deletion
                    of missing data
rcorr.cens      Somers' Dyx rank correlation with censored data
rcorrp.cens     Assess difference in concordance for paired predictors
rcspline.eval   Evaluate restricted cubic spline design matrix
rcspline.plot   Plot spline fit with nonparametric smooth and grouped estimates
rcspline.restate
                Restate restricted cubic spline in unrestricted form, and
                    create TeX expression to print the fitted function
recode          Recodes variables
reShape         Reshape a matrix into 3 vectors, reshape serial data
rm.boot         Bootstrap spline fit to repeated measurements model,
                    with simultaneous confidence region - least
                    squares using spline function in time
rMultinom       Generate multinomial random variables with varying prob.
samplesize.bin  Sample size for 2-sample binomial problem
                    (Rick Chappell, chappell@stat.wisc.edu)
sas.get         Convert SAS dataset to S data frame
sasxport.get    Enhanced importing of SAS transport dataset in R
scat1d          Add 1-dimensional scatterplot to an axis of an existing plot
                    (like bar-codes, FEH/Martin Maechler,
                    maechler@stat.math.ethz.ch/Jens Oehlschlaegel-Akiyoshi,
                    oehl@psyres-stuttgart.de)
score.binary    Construct a score from a series of binary variables or
                    expressions
sedit           A set of character handling functions written entirely
```

```
                    in S.  sedit() does much of what the UNIX sed
                    program does.  Other functions included are
                    substring.location, substring<-, replace.string.wild,
                    and functions to check if a string is numeric or
                    contains only the digits 0-9
setpdf          Adobe PDF graphics setup for including graphics in books
                    and reports with nice defaults, minimal wasted space
setps           Postscript graphics setup for including graphics in books
                    and reports with nice defaults, minimal wasted space
                    Internally uses psfig function by
                    Antonio Possolo (antonio@atc.boeing.com).
                    setps works with Ghostscript to convert .ps to .pdf
setTrellis      Set Trellis graphics to use blank conditioning panel strips,
                    line thickness 1 for dot plot reference lines:
                    setTrellis(); 3 optional arguments
show.col        Show colors corresponding to col=0,1,...,99
show.pch        Show all plotting characters specified by pch=.
                    Just type show.pch() to draw the table on the
                    current device.
showPsfrag      Use LaTeX to compile, and dvips and ghostview to
                   display a postscript graphic containing psfrag strings
solvet          Version of solve with argument tol passed to qr
somers2         Somers' rank correlation and c-index for binary y
spearman        Spearman rank correlation coefficient  spearman(x,y)
spearman.test   Spearman 1 d.f. and 2 d.f. rank correlation test
spearman2       Spearman multiple d.f. rho^2, adjusted rho^2, Wilcoxon-Kruskal-
                    Wallis test, for multiple predictors
spower          Simulate power of 2-sample test for survival under
                    complex conditions
                    Also contains the Gompertz2,Weibull2,Lognorm2 functions.
spss.get        Enhanced importing of SPSS files using R's read.spss function
src             src(name) = source("name.s") with memory
store           store an object permanently (easy interface to assign function)
strmatch        Shortest unique identifier match
                    (Terry Therneau, therneau@mayo.edu)
subset          More easily subset a data frame
substi          Substitute one var for another when observations NA
summarize       Generate a data frame containing stratified summary
                    statistics.  Useful for passing to trellis.
summary.formula General table making and plotting functions for summarizing
                    data
symbol.freq     X-Y Frequency plot with circles' area prop. to frequency
sys             Execute unix() or dos() depending on what's running
tex             Enclose a string with the correct syntax for using
                   with the LaTeX psfrag package, for postscript graphics.
transace        ace() packaged for easily automatically transforming all
```

```
                    variables in a matrix
transcan        automatic transformation and imputation of NAs for a
                    series of predictor variables
trap.rule       Area under curve defined by arbitrary x and y vectors,
                using trapezoidal rule
trellis.strip.blank
                To make the strip titles in trellis more visible, you can
                    make the backgrounds blank by saying trellis.strip.blank().
                    Use before opening the graphics device.
t.test.cluster  2-sample t-test for cluster-randomized observations
uncbind         Form individual variables from a matrix
units           Set or fetch "units" attribute - units of measurement for var.
upData          Update a data frame (change names, labels, remove vars, etc.)
varclus         Graph hierarchical clustering of variables using squared
                Pearson or Spearman correlations or Hoeffding D as similarities
                Also includes the naclus function for examining similarities in
                patterns of missing values across variables.
xy.group        Compute mean x vs. function of y by groups of x
xYplot          Like trellis xyplot but supports error bars and multiple
                    response variables that are connected as separate lines
win.slide       Setup win.graph or win.printer using nice defaults for
                presentations/slides/publications
wtd.mean, wtd.var, wtd.quantile, wtd.ecdf, wtd.table, wtd.rank,
wtd.loess.noiter, num.denom.setup
                Set of function for obtaining weighted estimates
zoom            Zoom in on any graphical display
                    (Bill Dunlap, bill@statsci.com)
```

The web page listed at the front of this document contains several datasets useful in learning about the Hmisc and Design libraries. Two of the data frames are especially useful for learning about logistic modeling with the Design library: `titanic` and `titanic2`. Both describe the survival status of individual passengers on the Titanic. The `titanic` data frame does not contain information from the crew, but it does contain actual ages of half of the passengers.

2.10 Installing Add–on Libraries

For Windows, many of the libraries available in `Statlib` are transported as compressed (`.zip`) files. Installation in this case is trivial, as the user merely needs to `unzip`[5] the file into the S-PLUS `library` area[6].

[5]Use a recent version of `WinZip` (from `www.winzip.com`) or a recent version of `unzip` that preserves long file names for Windows 95. A good version of `unzip` is available under `Utilities` in the Web page listed on the cover of this document. The UVa Web page, under `Statistical Computing Tools`, has more instructions for installing add–on libraries using `WinZip`.

[6]E.g., `/splus/library` as most `.zip` files for add–on libraries have been created so that during extraction they will be stored in the correct subdirectory of `/splus/library`.

Windows S libraries that call Fortran or C routines (as Hmisc and Design do) are so easy to install because the object modules for these routines is stored in a standard format that works on all Windows machines[7]. Therefore the user does not have to have a compiler on her machine. UNIX users install the libraries using a `Makefile` which invokes compilers as needed. Some users do not have a Fortran 77 or Fortran 90 compiler on their UNIX system; they have to install such a compiler before installing Hmisc or Design. A Fortran–to–C translator produces Fortran code that is too inefficient to be used. Some of the code that needs to be compiled is actually structured Fortran (Ratfor), which needs a Ratfor pre–processor to translate it to Fortran. Users without Ratfor can get pre–processed code already translated to Fortran from the Virginia Web page[8].

To install or update the Hmisc or Design library for R, download the appropriate file from `http://hesweb1.med.virginia.edu/biostat/s/library/` (`.zip` file for Windows; `.tar.gz` file for Linux/Unix) and store it in a directory for holding temporary files. If using Windows select the appropriate menu to install/update the package from a local file. If using Linux/Unix issue a shell command like `R CMD INSTALL /tmp/packagename.tar.gz` while logged in as superuser. When Hmisc and Design become part of CRAN, they may be installed like other CRAN packages (e.g., by issuing a command like `install.packages('Hmisc')` or `update.packages('Hmisc')` at the R command prompt).

2.11 Accessing Add–On Libraries Automatically

As described in more detail in Section 13.6, you can create a special function in your `_Data` area that is executed each time S is invoked from your project area. The function is called `.First`. A common use of `.First` is to do away with the need to issue a `library` command each time you invoke S. You can define a `.First` function once and for all by entering statements such as these in a `Commands` or `Script` window.

```
.First ← function() {
  library(Hmisc,T)
  invisible()
}
```

The `invisible` function prevents the `.First` function from printing anything when it is invoked.

For R use the command `library(Hmisc)` instead of `library(Hmisc,T)`. If you create a `.First` function for R it will be stored in `.RData`.

Because Hmisc has a variety of basic functions that are useful in routine data analysis and because attaching the Hmisc library carries almost no overhead, it can be a good idea to create such a `.First` function for each project area [9].

[7]Similarly, help files are stored in compiled Microsoft Help format, so these also install easily.

[8]But note that S-Plus comes with a Ratfor pre–processor too.

[9]Hmisc overrides the system subscripting method for `factor` vectors and `date` vectors, and it defines functions `is.na.dates` and `is.na.times` to check for `NA`s in `date` and `time` vectors. The `[.factor` redefinition by Hmisc causes by default unused levels to be dropped from the factor vector's `levels` attribute when the vector is subscripted. This can be overridden by using for example `x ← x[,drop=F]` or by specifying a system option as follows: `options(drop.factor.levels=F)`.

Chapter 3

Data in S

3.1 Importing Data

If you are using Windows S-Plus, most datasets you will need to analyze will be in a format that can be imported easily using the `File ... Import` dialog. For example, Excel spreadsheets, text (ASCII) files, and data from other popular statistical software can be converted to S-Plus internal format this way. This method is fast but not all data attributes (e.g., SAS variable labels and value labels) may be imported (see Section 3.2.3). Watch out for non-numeric values in Excel numeric columns, which S-Plus will import as infinity rather than NA. The Hmisc `cleanup.import` function will change such values to NA as well as set the storage mode of numeric variables to `'single'` or `'integer'` depending on whether fractional values are present. This will result in cutting storage in half for numeric variables, as S-Plus imports these as double precision variables (16 significant digits). `cleanup.import` also fixes another problem where numeric variables are mistakenly converted to factors. The Hmisc `upData` function does some of the same functions of `cleanup.import` in addition to allowing one to change the data frame in many ways (see Section 4.1.5).

The remainder of this chapter deals with commands (functions) for reading and converting data.

3.2 Reading Data into S

3.2.1 Reading Raw Data

The two main functions for reading ASCII datasets into S are `scan` and `read.table`. `scan` is the most versatile of the two, and `read.table` is easier to use. `read.table` expects the input data sets to be arranged in tabular form, where the first line may or may not be the variable names. The syntax is

```
> args(read.table)
```

```
function(file, header = F, sep = "", row.names = NULL,
         col.names = paste("V", 1:
         fields, sep = ""), as.is = F, na.strings = "NA")
```

The first argument is a character string reflecting the dataset name; `header` is set to `T` if the first line of the file contains the variable names; `sep` is the separator between fields (by default, any number of blanks); the `row.names` argument can be an already existing vector of the same length as the number of observations or the name of a variable in the dataset. In either case, it should have no duplicates. `col.names` is used to give names to variables when `header` is `F` and `as.is` controls which fields are converted to factors. By default, character fields are always made into factor objects. Finally, `na.strings` can be used whether certain values in character strings should be included as levels of a factor. The result of `read.table` is a data frame.

The function `scan` is more complicated and we will only give a sketch here.

```
> args(scan)
function(file = "", what = double(0), n = -1, sep = "", multi.line = F, flush
         = F, append = F, skip = 0, widths = NULL, strip.white = NULL)
```

The most important arguments here are `file` and `what`. The first one is just the name of your dataset, and `what` is sort of like an INPUT statement. It is a list giving the names and the modes of the data. Example,

```
> z ← scan("myfile",list(pop=0,city=character()))
```

In this case, we are reading from the dataset `"myfile"` the first two columns and naming them `pop` and `city`. The `0` after the equal sign in `pop` only means that it is going to be read as a numeric variable. Any other number or the expression `numeric(0)` would have had the same effect. Similarly with the `character()` expression.

In S-Plus for Windows you can also read ASCII files using point-and-click methods through the `File` menu.

3.2.2 Reading S-Plus Data into R

The best way to transport S-Plus vectors, matrices, and data frames to other computers or other versions of S-Plus or to R is to run `data.dump()` in S-Plus to create a `dumpdata`-format (S-Plus transport format or `.sdd` file, as described in Section 3.5.2. If using S-Plus version 5 or later, use the `oldStyle=T` option to `data.dump`. Then convert the object to an R object using code such as the following.

```
library(foreign)
data.restore('/tmp/my.sdd')  # name of resulting object comes from
                             # original name when my.sdd created
```

You can read binary S objects in `_Data` or `.Data` directories and convert them to R objects in some cases using R's `read.S` function in R's `foreign` library, if the object was created by S-Plus versions before version 5 (e.g., conversion of S-Plus 2000 binary objects usually works). Here is an example:

```
library(foreign)
# Print file _Data/___nonfi to see mapping of renamed files
```

```
    # to object names
    newobj ← read.S('_Data/__7')  # must provide a name to hold result
```

Check the resulting object carefully, because `read.S` is not foolproof.

3.2.3 Reading SAS Datasets

In many cases, the easiest way to read external files is to read SAS datasets directly. This can be done two ways. First, you can use `File ... Import` or a standalone database conversion utility such as DBMSCOPY. This approach has the advantages of speed of execution, ease of use, and lack of need of creating temporary ASCII files[1]. There are several disadvantages for either fast import method, however: (1) They do not carry SAS variable labels into S. (2) They ignore value labels for categorical variables created using SAS PROC FORMAT. (3) They do not transport SAS special missing values. (4) S variable names constructed from SAS names are in all upper case[2].

The `sas.get` function in the Hmisc library for UNIX or Windows is the other approach to convert SAS datasets. `sas.get` preserves all SAS data attributes, and if categorical variables have customized `FORMAT`s associated with them, `sas.get` has several options for defining the category labels to S (typically as `factor` variables).

Long before converting SAS data to S, you should have prepared the SAS dataset so that it would be as useful as possible in SAS. Then `sas.get` can also profit from this setup. Here are the relevant points to consider when creating your SAS dataset:

1. Define `LABEL`s on all variables that are not totally self-documenting. The labels should contain mostly lower case letters, as such labels are not only easier to read but they will result in prettier SAS and S output. If you did not take the time to create pretty SAS labels, you can create or override labels after reading the data into S.

2. Use the minimum SAS `LENGTH` that will store each character or numeric variable. For number variables, SAS uses a default of 8 bytes of storage, which is 16 significant digits. Such precision is very seldom needed, and it will result in highly inflated SAS and S datasets. Many SAS variables can be stored as 3 byte floating points, which yields 4 significant digits.

3. Define category level definitions using `PROC FORMAT`, and associate the formats permanently with the appropriate variables.

4. Don't store dummy variables and other derived variables (e.g., interaction products) in the permanent SAS dataset, and if you do, don't retrieve them into S as S derives such variables on the fly.

If you do not have nice variable `labels` or category `levels` set up in SAS, you can always create them or redefine them in S:

```
  sex ← factor(sex, 1:2, c('female','male'))
  levels(treatment)[3] ← 'Dextran'
  levels(location) ← edit(levels(location)) # edit them interactively
  label(location) ← 'Location of last inspection'
```

[1]The `sas.get` function has to create temporary ASCII files to do the SAS to S translation.
[2]This can easily be remedied — see Section 3.4.

The `Label` function which is documented under the `label` function will create a text file containing S code defining the existing labels for all the variables in a data frame. You can edit that code, overriding any labels you don't like (including blank ones) and `source` that file back into S. Call `Label` using the syntax `Label(dataframename, file=)`. Omit `,file=` to write labels to the command window for copying and pasting into an editor window.

Here is the help file for the Windows version of `sas.get`. The UNIX version does not have the `sasout` argument, and there are a few other differences. When one or more of the variables you are rescuing from SAS has a `PROC FORMAT` `format` associated with it, it is best to use the `recode=T` option (the default) when invoking `sas.get`.

sas.get Convert a SAS Dataset to an S Dataset **sas.get**

Converts a SAS dataset into an S data frame. You may choose to extract only a subset of variables or a subset of observations in the SAS dataset. You may have the function automatically convert PROC FORMAT-coded variables to factor objects. The original SAS codes are stored in an attribute called `sas.codes` and these may be added back to the `levels` of a `factor` variable using the `code.levels` function. Information about special missing values may be captured in an attribute of each variable having special missing values. This attribute is called `special.miss`, and such variables are given class `special.miss`. There are `print`, `[]`, `format`, and `is.special.miss` methods for such variables. The `chron` function is used to set up date, time, and date-time variables. If a date variable represents a partial date (.5 added if month missing, .25 added if day missing, .75 if both), an attribute `partial.date` is added to the variable, and the variable also becomes a class `imputed` variable. The `describe` function uses information about partial dates and special missing values. There is an option to automatically `PKUNZIP` compressed SAS datasets.

`sas.get` works by composing and running a SAS job that creates various ASCII files that are read and analyzed by `sas.get`. You can also run the SAS `sas_get` macro, which writes the ASCII files for downloading, in a separate step or on another computer, and then tell `sas.get` to access these files instead of running SAS.

```
sas.get(library, member, variables=<<see below>>, ifs=<<see below>>,
     format.library=library, sasout,
     formats=F, recode=formats, special.miss=F, id=<<see below>>,
     as.is=.5, check.unique.id=T, force.single=F,
     keep.log=T, log.file="_temp_.log", macro=sas.get.macro,
     clean.up=T,  sasprog="sas", where, unzip=F)
is.special.miss(x, code)
x[...]
print(x)
format(x)
sas.codes(x)
x ← code.levels(x)
```

ARGUMENTS

`library`: character string naming the directory in which the the dataset is kept. The default is `library="."`, indicating that the current directory is to be used.

`member`: character string giving the second part of the two part SAS dataset name. (The first part is irrelevant here — it is mapped to the directory name.)

`x`: a variable that may have been created by `sas.get` with `special.miss=T` or with `recode` in effect.

`variables`: vector of character strings naming the variables in the SAS dataset. The S dataset will contain only those variables from the SAS dataset. To get all of the variables (the default), an empty string may be given. It is a fatal error if any one of the variables is not in the SAS dataset.

`ifs`: a vector of character strings, each containing one SAS "subsetting if" statement. These will be used to extract a subset of the observations in the SAS dataset.

`format.library`: The directory containing the file `formats.sc2`, which contains the definitions of the user defined formats used in this dataset. By default, we look for the formats in the same directory as the data. The user defined formats must be available (so SAS can read the data).

`sasout`: If SAS has already run to create the ASCII files needed to complete the creation of the S data frame, specify a vector of 4 character strings containing the names of the files (with full path names if the files are not on the current working directory). The files are in the following order: data dictionary, data, formats, special missing values. This is the same order that the file names are specified to the `sas_get` macro. For files which were not created and hence not applicable, specify `""` as the file name. The presence/absence of formats and special missing data files is used to set the `formats` and `special.miss` arguments automatically by `sas.get`.

`sasout` may also be a character string of length one, in which case it is assumed to be the name of a `.zip` file, and `sas.get` automatically runs the DOS `PKUNZIP` command to extract the component files to the current working directory. The files that are present in the `.zip` file must have names `"dict","data","formats","specmiss"` (although `"formats"` and `"specmiss"` do not have to be present). When `sas.get` is finished, these extracted files are automatically deleted. `.zip` files are useful for downloading large datasets.

`formats`: Set `formats` to `T` to examine the `format.library` for appropriate formats and store them as the `formats` attribute of the returned object (see below). A format is used if it is referred to by one or more variables in the dataset, if it contains no ranges of values (i.e., it identifies value labels for single values), and if it is a character format or a numeric format that is not used just to label missing values. If you set `recode` to `T`, 1, or 2, `formats` defaults to `T`. To fetch the values and labels for variable `x` in the dataset `d` you could type: f ←attr(d$x, "format") formats ←attr(d, "formats") formats$f$values; formats$f$labels

`recode`: This parameter defaults to `T` if `formats` is `T`. If it is `T`, variables that have an appropriate format (see above) are recoded as `factor` objects, which map the values to the value labels for the format. Alternatively, set `recode` to 1 to use labels of the form value:label, e.g. 1:good 2:better 3:best. Set `recode` to 2 to use labels such as good(1) better(2) best(3). Since `sas.codes` and `code.levels` add flexibility, the usual choice for `recode` is `T`.

`special.miss`: For numeric variables, any missing values are stored as NA in S. You can recover special missing values by setting `special.miss` to `T`. This will cause the `special.miss` attribute and the `special.miss` class to be added to each variable that has at least one special missing value. Suppose that variable `y` was .E in observation 3 and .G in observation 544. The `special.miss` attribute for `y` then has the value list(codes=c(”E”,”G”),obs=c(3,544))) To fetch this information for variable `y` you would say for example s ←attr(y, ”special.miss”) s$codes; s$obs or use `is.special.miss(x)` or the `print.special.miss` method, which will replace `NA` values for the variable with `E` or `G` if they correspond to special missing values. The describe function uses this information in printing a data summary.

`id`: The name of the variable to be used as the row names of the S dataset. The id variable becomes the `row.names` attribute of a data frame, but the id variable is still retained as a variable in the data frame. You can also specify a vector of variable names as the `id` parameter. After fetching the data from SAS, all these variables will be converted to character format and concatenated (with a space as a separator) to form a (hopefully) unique ID variable.

`as.is`: SAS character variables are converted to S factor objects if `as.is=F` or if `as.is` is a number between 0 and 1 inclusive and the number of unique values of the variable is less than the number of observations (`n`) times `as.is`. The default if `as.is` is .5, so character variables are converted to factors only if they have fewer than `n/2` unique values. The primary purpose of this is to keep unique identification variables as character values in the data frame instead of using more space to store both the integer factor codes and the factor labels.

`check.unique.id`: If `id` is specified, the row names are checked for uniqueness if `check.unique.id=T`. If any are duplicated, a warning is printed. Note that if a data frame is being created with duplicate row names, statements such as `my.data.frame["B23",]` will retrieve only the first row with a row name of `"B23"`.

`force.single`: By default, SAS numeric variables having `LENGTH`s > 4 are stored as S double precision numerics, which allow for the same precision as a SAS `LENGTH` 8 variable. Set `force.single=T` to store every numeric variable in single precision (7 digits of precision). This option is useful when the creator of the SAS dataset has failed to use a `LENGTH` statement.

`keep.log`: logical flag: if `F`, delete the SAS log file upon completion.

`log.file`: the name of the SAS log file.

`macro`: the name of an S object in the current search path that contains the text of the SAS macro called by S. The S object is a character vector that can be edited using, for example, sas.get.macro ←editor(sas.get.macro).

`clean.up`: logical flag: if `T`, remove all temporary files when finished. You may want to keep these while debugging the SAS macro.

`sasprog`: the name of the system command to invoke SAS `unzip`: set to `F` by default. Set it to `T` to automatically invoke the DOS `PKUNZIP` command if `member.zip` exists, to uncompress the SAS dataset before proceeding. This assumes you have the file permissions to allow uncompressing in place. If the file is already uncompressed, this option is ignored.

`where`: by default, a list or data frame which contains all the variables is returned. If you specify `where`, each individual variable is placed into a separate object (whose name is the name of the variable) using the `assign` function with the `where` argument. For example, you can put each variable in its own file in a directory, which in some cases may save memory over attaching a data frame.

`code`: a special missing value code (A through Z or underscore) to check against. If `code` is omitted, `is.special.miss` will return a `T` for each observation that has any special missing value.

VALUE

A data frame resembling the SAS dataset. If `id` was specified, that column of the data frame will be used as the row names of the data frame. Each variable in the data frame or vector in the list will have the attributes `label` and `format` containing SAS labels and formats. Underscores in formats are converted to periods. Formats for character variables have `$` placed in front of their names. If `formats` is `T` and there are any appropriate format definitions in `format.library`, the returned object will have attribute `formats` containing lists named the same as the format names (with periods substituted for underscores and character formats prefixed by $). Each of these lists has a vector called `values` and one called `labels` with the PROC FORMAT; VALUE ... definitions.

SIDE EFFECTS

if a SAS error occurs the SAS log file will be printed under the control of the `pager` function.

DETAILS

If you specify `special.miss=T` and there are no special missing values in the data SAS dataset, the SAS step will bomb.

For variables having a `PROC FORMAT VALUE` format with some of the levels undefined, `sas.get` will interpret those values as `NA` if you are using `recode`.

If you leave the `sasprog` argument at its default value of `"sas"`, be sure that the SAS executable is in the `PATH` specified in your `autoexec.bat` file. Also make sure that you invoke S so that your current project directory is known to be the current working directory. This is best done by creating a shortcut in Windows95, for which the command to execute will be something like `drive:\spluswin\cmd\splus.exe HOME=.` and the program is flagged to start in `drive:\myproject` for example. In this way, you will be able to examine the SAS log file easily since it will be placed in `drive:\myproject` by default.

SAS will create `SASWORK` and `SASUSER` directories in what it thinks are the current working directories. To specify where SAS should put these instead, edit the `config.sas` file or specify a `sasprog` argument of the following form: `sasprog="\sas\sas.exe -saswork c:\saswork -sasuser c:\sasuser"`.

When `sas.get` needs to run SAS it is run in iconized form.

The SAS macro `sas_get` uses record lengths of up to 4096 in two places. If you are exporting records that are very long (because of a large number of variables and/or long character variables), you may want to edit these `LRECL`s to quadruple them, for example.

NOTE

If `sasout` is not given, you must be able to run SAS on your system.

If you are reading time or date-time variables, you will need to execute the command `library(chron)` to print those variables or the data frame.

BACKGROUND

The references cited below explain the structure of SAS datasets and how they are stored. See *SAS Language* for a discussion of the "subsetting if" statement.

AUTHORS

Frank Harrell, University of Virginia, Terry Therneau, Mayo Clinic, Bill Dunlap, University of Washington and MathSoft.

REFERENCES

SAS Institute Inc. (1990). SAS Language: Reference, Version 6. First Edition. SAS Institute Inc., Cary, North Carolina.

SAS Institute Inc. (1988). SAS Technical Report P-176, Using the SAS System, Release 6.03, under UNIX Operating Systems and Derivatives. SAS Institute Inc., Cary, North Carolina.

SAS Institute Inc. (1985). SAS Introductory Guide. Third Edition. SAS Institute Inc., Cary, North Carolina.

SEE ALSO

`data.frame`, `describe`, `impute`, `chron`, `print.display`, `label`

EXAMPLE

```
> mice ← sas.get("saslib", mem="mice", var=c("dose", "strain", "ld50"))
> plot(mice$dose, mice$ld50)
> nude.mice ← sas.get(lib=unix("echo $HOME/saslib"), mem="mice",
    ifs="if strain='nude'")
> nude.mice.dl ← sas.get(lib=unix("echo $HOME/saslib"), mem="mice",
    var=c("dose", "ld50"), ifs="if strain='nude'")
> # Get a dataset from current directory, recode PROC FORMAT; VALUE ...
> # variables into factors with labels of the form "good(1)" "better(2)",
> # get special missing values, recode missing codes .D and .R into new
> # factor levels "Don't know" and "Refused to answer" for variable q1
> d ← sas.get(mem="mydata", recode=2, special.miss=T)
> attach(d)
> nl ← length(levels(q1))
> lev ← c(levels(q1), "Don't know", "Refused")
> q1.new ← as.integer(q1)
> q1.new[is.special.miss(q1,"D")] ← nl+1
> q1.new[is.special.miss(q1,"R")] ← nl+2
> q1.new ← factor(q1.new, 1:(nl+2), lev)
> # Note: would like to use factor() in place of as.integer ... but
> # factor in this case adds "NA" as a category level
>
> d ← sas.get(mem="mydata", recode=T)
> sas.codes(d$x)    # for PROC FORMATted variables returns original data codes
> d$x ← code.levels(d$x)   # or attach(d); x ← code.levels(x)
> # This makes levels such as "good" "better" "best" into e.g.
> # "1:good" "2:better" "3:best", if the original SAS values were 1,2,3
> # For the following example, suppose that SAS is run on a
> # different machine from the one on which S is run.
> # The sas_get macro is used to create files needed by
> # sas.get (To make a text file containing the sas_get macro
> # run the following S command, for example:
> #   cat(sas.get.macro, file='/sasmacro/sas_get.sas', sep='\n')
>
> # Here is the SAS job.  This job assumes that you put
> # sas_get.sas in an autocall macro library.
>
> #  libname db '/my/sasdata/area';
> #  %sas_get(db.mydata, dict, data, formats, specmiss,
> #           formats=1, specmiss=1)
>
> # Substitute whatever file names you may want.
> # Next the 4 files are moved to the S machine (using
> #  ASCII file transfer mode) and the following S
> # program is run:
>
> mydata ← sas.get(sasout=c('dict','data','formats','specmiss'),
+                  id='idvar')
>
> # If PKZIP is run after sas_get, e.g. "PKZIP port dict data formats"
> # (assuming that specmiss was not used here), use
>
> mydata ← sas.get(sasout='a:port', id='idvar')
>
> # which will run PKUNZIP port to unzip a:port.zip, creating the
> # dict, data, and formats files which are interpreted (and leter
> # deleted) by sas.get
```

`sas.get` calls a SAS macro which produces an ASCII dataset and then uses `scan` to read it into an S object. If there are errors during the SAS macro processing step, the log file is displayed on the screen (unless `quiet=T`). This way you can usually know what type of error you have. A common error is that your dataset is in some directory and your formats catalog is in another while omitting the `formats.library` argument to `sas.get` (see below). Another error you may find is the message "file such and such not found". On some systems, this condition may occur if your SAS dataset has not been modified in a while and the system compressed it automatically. Set `uncompress=T` in this case. Also, if you don't have special missing values, do not set `special.miss` to `T`.

The `sas_get` SAS macro specifies the system option `NOFMTERR`, so if customized formats or format libraries are not found, SAS will procede as if the offending variables did not have a format associated with them. This works fine when the undefined formats correspond to variables not requested for retrieval. If however you request a variable having a missing format, you may not know about it until you run `describe` or other functions.

3.2.4 Handling Date Variables in R

R has a comprehensive way of storing and operating on date, time, and date/time values based on `POSIX` notation. Type `?DateTimeClasses` for details. If you import SAS datasets into R using `sas.get`, SAS date, time, and date/time variables are automatically converted into R's `POSIXct` variables.

If you read date/time fields from ASCII text files, the following example shows how to convert into `POSIXct` variables. Suppose that a comma separated file `test.csv` contains the following data:

```
age,date
21,12/31/02
22,01/01/03
23,1/1/02
24,12/1/02
25,12/1/02
26,
```

The following program can read and recode the data.

```
> mydata
  age     date
1  21 12/31/02
2  22 01/01/03
3  23   1/1/02
4  24  12/1/02
5  25  12/1/02
6  26
> d ← mydata$date
> d
[1] 12/31/02 01/01/03 1/1/02   12/1/02  12/1/02
Levels:  01/01/03 1/1/02 12/1/02 12/31/02

> d ← as.POSIXct(strptime(as.character(d),format='%m/%d/%y'))
> # For 4-digit years, use format='%m/%d/%Y'
> # If data were in the format yyyy-mm-dd the conversion would
> # be as simple as d <- as.POSIXct(d)
> d
[1] "2002-12-31 EST" "2003-01-01 EST" "2002-01-01 EST" "2002-12-01 EST"
[5] "2002-12-01 EST" NA

> format(d, '%d%b%Y')
[1] "31Dec2002" "01Jan2003" "01Jan2002" "01Dec2002" "01Dec2002" NA

> # Create a function to make it easy to reformat multiple variables
> dtrans ← function(x, format='%m/%d/%y')
+   as.POSIXct(strptime(as.character(x),format))
>
> mydata$date ← dtrans(mydata$date)
> mydata
  age       date
1  21 2002-12-31
2  22 2003-01-01
3  23 2002-01-01
4  24 2002-12-01
5  25 2002-12-01
6  26       <NA>

> unclass(mydata$date)   # internal values
[1] 1041310800 1041397200 1009861200 1038718800 1038718800         NA
```

3.3 Displaying Metadata

The Hmisc `contents` function displays data about a data frame, including variable `labels` (if any), `units` (if any) storage modes, number of `NA`s, and the number of levels for `factor` variables. Here is an example.

```
> contents(pbc)

418 observations and 19 variables    Maximum # NAs:136

                                       Labels Levels Storage NAs
    bili               Serum Bilirubin (mg/dl)         single   0
 albumin                       Albumin (gm/dl)         single   0
   stage      Histologic Stage, Ludwig Criteria         single   6
 protime                  Prothrombin Time (sec.)       single   2
     sex                                    Sex      2 integer   0
fu.days Time to Death or Liver Transplantation         single   0
     age                                    Age         single   0
 spiders                                Spiders      2 integer 106
 hepatom                           Hepatomagaly      2 integer 106
 ascites                                Ascites      2 integer 106
alk.phos        Alkaline Phosphatase (U/liter)         single 106
    sgot                            SGOT (U/ml)         single 106
    chol                   Cholesterol (mg/dl)         single 134
    trig                  Triglycerides (mg/dl)         single 136
platelet              Platelets (per cm^3/1000)         single 110
    drug                              Treatment      3 integer   0
  status                       Follow-up Status         single   0
   edema                                  Edema      3 integer   0
  copper                  Urine Copper (ug/day)         single 108

> con <- contents(pbc)
> print(con, sort='names')  # or sort='labels','NAs'
```

```
418 observations and 19 variables    Maximum # NAs:136

                                        Labels Levels Storage NAs
     age                                   Age          single   0
 albumin                       Albumin (gm/dl)          single   0
alk.phos        Alkaline Phosphatase (U/liter)          single 106
 ascites                               Ascites      2 integer 106
. . . .
```

3.4 Adjustments to Variables after Input

Whether raw data or a SAS dataset is used to create a data frame, and whether you used a command or a mouse click to import the data, it is frequently the case that variable names, labels, or value codes need adjustment. These items may be easily changed once and for all or they may be changed every time the data frame is "attached" (see Section 4.1.1). To change variable attributes permanently, the recommended approach is to use the Hmisc `upData` function (Section 4.1.5). But here are some of the basic methods that are available. For changing individual variables in a list or data frame we rely first on the `$` operator for addressing individual variables in a permanent list of variables. This was introduced in Section 2.5.2. The advantage of making permanent changes in the data frame is that all interactive analyses of that data frame will take advantage of all the new variable names and annotations without prefacing the analysis with statements such as those found below.

In S-Plus Version 4.x and 2000 it is easy to change variable names by editing column names on a data sheet, but you will have to re-do this every time the source dataset changes and is in need of re-importing. The following method using the `edit` function has the same disadvantage but it works in all versions of S-Plus. Suppose that `df` is the newly created permanent data frame. The names may be edited using

```
names(df) ← edit(names(df))
```

or you can change individual names using for example

```
names(df)[2] ← 'Age'
```

This changed the name of the second variable on the data frame. Here is a trick for changing all the names to lower case:

```
names(df) ← casefold(names(df))  # casefold is builtin
```

Note: When the data are imported from an ASCII file, the best way to specify variable names is to enter them into the "column names" box under the `Options` tab during the file import operation.

To permanently change or define labels for variables, you can use statements such as the following.

```
label(df$age)  ← 'Age in years'
label(df$chol) ← 'Cholesterol (mg%)'
```

To define or change value labels we use the `factor` function and the `levels` attribute (if the variable is already a `factor`). Suppose that one variable, `sex`, has values 1 and 2 and that we need to define these as `'female'` and `'male'`, respectively, so that reports and plots will be annotated. Suppose that another variable is already a `factor` vector, but that we do not like its levels (`'a','b','c'`). The following statements will fix both problems.

```
df$sex ← factor(df$sex, 1:2, c('female','male'))
levels(df$treat) ← c('Treatment A','Treatment B','Treatment C')
# This can also be done with the following command
df$treat ← factor(df$treat, c('a','b','c'),
                  c('Treatment A','Treatment B','Treatment C'))
```

When a variable is already a `factor` and you wish to change its levels, you can also use the `edit` function:

```
levels(v) ← edit(levels(v))
```

Sometimes the input data will contain a factor variable having one or more unused levels. You can delete unused levels from the `levels` attribute of a variable, say `x`, by typing `x ← x[,drop=T]`. If the Hmisc library is in effect you merely have to type `x ← x[]` as Hmisc uses a default value of `drop=T` for its `[.factor` factor subsetting method.

Other sections show how to define labels and value labels when you only want temporary assignments. This is simpler as you do not need the data frame prefix as in the statements above. You can also `attach` the data frame in search position one to alleviate the need for the $ prefixing:

```
attach(df, pos=1, use.names=F)
sex ← factor(sex, 1:2, c('female','male'))
levels(treat) ← ...
label(w3) ← 'A-V area'
detach(1, 'df')
```

See Section 4.1.1 for more on this point. See section Section 4.4 for more details about recoding variables, Section 4.1.3 for how to add new variables, and Section 4.1.4 for how to delete variables. Section 4.5 has a review of the many steps one typically goes through to create ready–to–analyze data frames.

See Section 3.1 for more about the `cleanup.import` function, which can be run on any data frame.

3.5 Writing Out Data

There are generally two instances in which you want to write output to a file. To produce a printed report (which may be enhanced by using some kind of publishing software), or to produce a dataset which may be shared with other users. In the latter case, especially if the other users are not using S-Plus, the most straightforward way is to use `File ... Export` or DBMSCOPY or to write an ASCII file. The latter approach can be done with the function `write.table`.

3.5.1 Writing ASCII files

`write.table` is very similar to `read.table`. Its arguments and an example follow.

```
> args(write.table)
function(data, file = "", sep = ",", append = F, quote.strings = F,
        dimnames.write = T, na = NA, end.of.row = "\n")
> write.table(df,"df.ascii",sep=" ",dimnames.write=F,quote.strings=T)
> !less df.ascii  # escaping to UNIX and using the 'less' pager
> # Could use !notepad df.ascii under Windows
"Treatment 1" 2.5
"Treatment 1" 3.5
"Treatment 1" 3.0
"Treatment 2" 4.6
"Treatment 2" 5.5
"Treatment 2" 5.3
```

3.5.2 Transporting S Data

S-Plus stores objects in an internal binary format that is specific to each hardware platform. Fortunately there is an ASCII transport format that can be used to move objects between any two machines. This format is called *dumpdata* or *transport file* format. You can write *any* S-Plus object to a transport file using the `data.dump` function[3], and you can read such files using `data.restore`. These functions also allow you to write or read a single file containing any number of objects. You can use the `File ... Export Data` or `File ... Import Data` dialogs to write or read transport files. When you read, all the objects are created or re-created into search position one.

3.5.3 Customized Printing

The basic function for producing customized output is the `cat` function. When used in conjunction with other functions like `paste`, `round` and `format`, it can print nicely formatted reports. The basic syntax for `cat` is `cat("character string 1",object,"character string 2")`. Ex:

```
> cat("The mean of x is",mean(x))
The mean of x is 4.06666666666667>
```

Two problems are immediately apparent here: one is that `mean(x)` is producing too many decimals. The other is that `cat` is not going to a new line after being executed. To go to a new line, the newline character `\n` must be included explicitly. To control the number of digits the functions `round` or `format` can be used. `round(mean(x),3)` will round the output of `mean(x)` to three significant digits, while `format(mean(x))` will print `mean(x)` with as many digits as the `digits` options is set.

```
> cat("The mean of x is",round(mean(x),3),"\n")
The mean of x is 4.067
> options()$digits
[1] 7
> options(digits=4)
> cat("The mean of x is",format(mean(x)),"\n")
The mean of x is 4.067
```

The `options` function controls some of the system options that are assumed by default such as maximum object size, number of digits, width of a printed line, etc. You can see all the options by

[3]To make the result backward compatible, specify `oldStyle=T` to `data.dump` when running on S-Plus 5 or 6.

typing `options()`. The result of this action is a list, that's why we typed `options()$digits` to get the value of just the `digits` option. The effect of `format` is to coerce objects to become character strings using a common format.
`cat` prints its arguments in the order in which it encounters them, so, to print something like "value 1 value 2 ... value 10" you would have to type `cat("value 1", ... ,"value 10")`. The `paste` function is more efficient for this purpose

```
> paste("Value",1:10)
 [1] "Value 1"  "Value 2"  "Value 3"  "Value 4"  "Value 5"  "Value 6"
 [7] "Value 7"  "Value 8"  "Value 9"  "Value 10"
```

Using `cat` in conjunction with `paste` will give us a nicer output

```
> cat(paste("Value",1:10),fill=8)
Value 1
Value 2
Value 3
Value 4
Value 5
Value 6
Value 7
Value 8
Value 9
Value 10
```

`paste` returned a character string, using `cat` deleted the quotation marks. The argument `fill` instructed `cat` to put a new line at 8 characters. Other arguments to `cat` include `file` to send the output to a file that you name, `append` to cause `cat` to append any new output to an existing file (or destroy the contents of the file), and `sep` to insert characters between the arguments to `cat` in the output. (`sep=" "` is the default. It can be changed to `""` for no spaces).

The `print.char.matrix` function built-in to S-Plus is useful for printing hierarchical tables, as it automatically draws boxes separating cells of a table, and each cell can comprise multiple output lines. For R, `print.char.matrix` is in the Hmisc library.

3.5.4 Sending Output to a File

You can have S send the output of all commands to a file by using the `sink` function. `cat` will only send the results of its output to a file, while `sink` will send the results of every command to a file you name (or a command) until you instruct it not to do so.

```
> sink("myfile")  # Send output to file myfile
> cat("The mean of x is",round(mean(x),3))
> sink() # Redirect output to the S session
```

3.6 Using the Hmisc Library to Inspect Data

Once the data are read into S, the Hmisc library can be helpful in understanding them as well as checking for "holes" and invalid data. Suppose a data frame named `w` has been created. Here is a suggested program for taking some initial looks. See Section 4.3.3 for more on the `sapply` function.

```
w.des ← describe(w)    # save describe() output
page(w.des, multi=T)    # put it in a Window that can linger
win.graph()  # open graphics window - openlook(), motif(), X11() for UNIX
             # not needed for S-Plus 4.x or later

# First make a dot chart of the number of NAs for each variable,
# sorting variables so that the worst offender is at the top
m ← sapply(w,function(x)sum(is.na(x)))
dotplot(sort(m), xlab='NAs')    # naplot below does this automatically
na.pattern(m)                   # gets frequencies of all NA patterns but
                                # treats factor variables as always non-NA


nac ← naclus(w)   # compute all pairwise proportions of missing
                  # data and cluster variables according to similarity
                  # of occurrences of NAs
nac               # print matrix of pairwise proportions
plot(nac)         # cluster NA patterns graphically
naplot(nac)       # other displays of patterns of NA
hist.data.frame(w) # matrix of histograms for all non-binary variables
                  # also shows number of NAs
datadensity(w)    # make single graph with strip plots (1-dimensional
                  # scatterplots or rug plots) for all variables in w
                  # also consider using builtin plot(w)
ecdf(w)           # draw empirical cumulative distributions for all
                  # continuous variables.  Also consider using bpplot().

# Now depict how the variables cluster, using squared Spearman rank
# correlation coefficients as similarity measures.  varclus uses
# rcorr which does pairwise deletion of NAs

plot(varclus(~ x1 + x2 + x3 + ..., data=w))
# Assumes variables are named x1, x2, x3, ...
# Use plot(varclus(~., data=w)) to analyze all variables

# If any of the variables is missing frequently (say x2), find out what
# predicts its missingness.  Use a regression tree
f ← tree(is.na(x2) ~ x1 + x3, data=w)
# Could have used attach(w) to avoid data= above
plot(f, type='uniform')
text(f)

# Other useful functions for more detailed examinations of the data
# are bwplot, bpplot (box-percentile plots), bwplot with
# panel=panel.bpplot, and symbol.freq (for depicting
# two-way contingency tables).
```

See Section 11.3 for information about the `ecdf`, `datadensity`, and `bpplot` functions, and Section 6.1 for information about `symbol.freq`. See also the builtin function `cdf.compare`. And don't forget a wonderful built-in function 'plot.data.frame' that nicely displays continuous variables (using CDFs turned sideways) and categorical ones (using frequency dot charts). With a high-resolution printer you can see up to 40 variables clearly on a single page. Here is an example.

```
par(mfrow=c(5,8))   # allow up to 40 plots per page
plot(w)             # invokes plot.data.frame since w is a data frame
par(mfrow=c(1,1))   # reset to one plot per screen
```

See Section 11.4 for examples of the use of the `trellis` library instead of `datadensity` for drawing "strip plots" for depicting data distributions and data densities stratified by other variables.

When you permanently store the result of the `describe` function (here, in `w.des`), you can quickly replay it as needed, either by printing it by simply stating its name, or by using `page` to put it in a new window. If `page` had already been run with `multi=T` you merely click on that window's icon to restore it. Note that the `page` command[4] causes the pop-up window to remain after you exit from S-Plus when `multi=T`. That way you can open the data description whether you are currently in S-Plus or not. In addition to displaying the `w.des` object, you can easily display any subset of the variables it describes:

```
w.des[20:30]                # display description of variables 20-30
page(w.des[c(1:10,30:40)])  # page display variables 1-10, 30-40
w.des[c('age','sex')]       # display 2 variables
w.des$age                   # display single variable
```

[4]This is true for Windows, and for UNIX if you set your pager to be a window utility such as `xless`. An excellent pager for Windows is the PFE editor described in Section 1.9. You can set this up by typing `options(pager='/pfe/pfe32')` or clicking on `Options ... General Settings ... Computation`, for example. Then by using multiple commands of the form `page(object,multi=T)` you can have PFE manage all of the pager windows, as by default PFE will add new open files when it is called repeatedly, i.e., it will not invoke an entirely new copy of `pfe32.exe`. Perhaps an even better `pager` is an Emacs client. In Windows 95/NT you would set this up by using the command `options(pager='gnuclient -q')`.

Chapter 4

Operating in S

4.1 Reading and Writing Data Frames and Variables

In the introduction we created a subdirectory of your working directory called `.Data` (or `_Data`) because this allows for more organized data management, and because this is the default location in which S-Plus places new data. This way, all the objects that you create for a particular project are available since S-Plus will search by default in `.Data` if it exists. However, `.Data` is not the only directory available to you to store or search for objects. By default, when you start S, a search list is established and a series of directories is accessed sequentially looking for objects or functions. Said list can be modified. The function to display the search list is `search()`. Its purpose is similar to the `PATH` command in DOS or UNIX. `search()` will give us a list of all the directories that S searches looking for functions and data.

```
> library(Hmisc, T)
> library(Design,T)
> search()
 [1] "_Data"
 [2] "D:\\SPLUSWIN\\library\\Design\\_Data"
 [3] "D:\\SPLUSWIN\\library\\hmisc\\_Data"
 [4] "D:\\SPLUSWIN\\splus\\_Functio"
 [5] "D:\\SPLUSWIN\\stat\\_Functio"
 [6] "D:\\SPLUSWIN\\s\\_Functio"
 [7] "D:\\SPLUSWIN\\s\\_Dataset"
 [8] "D:\\SPLUSWIN\\stat\\_Dataset"
 [9] "D:\\SPLUSWIN\\splus\\_Dataset"
[10] "D:\\SPLUSWIN\\library\\trellis\\_Data"
```

The above `search` list contains directories, but you can also attach data frames to the list. When a data frame is in the `search` list, the variables within that data frame are available without using the name of the data frame as a prefix to the variable name.

4.1.1 The attach and detach Functions

To be able to reference objects (data frames, functions, vectors, etc.) that are not in the default `search` path, you can use the `attach` function. The main argument to `attach` is a directory name in single or double quotes or the name of a data frame without quotes. As an example, let us attach another directory that contains a variety of S objects. Recall that even in Windows we can specify forward slashes in file and directory names inside of S-Plus. You can also use a backward slash but it must be doubled, as `\` is an escape character when inside character strings.

```
> attach('c:/analyses/support/_Data')
> search()
 [1] "_Data"
 [2] "c:/analyses/support/_Data"
 [3] "D:\\SPLUSWIN\\library\\Design\\_Data"
 [4] "D:\\SPLUSWIN\\library\\hmisc\\_Data"
 [5] "D:\\SPLUSWIN\\splus\\_Functio"
 [6] "D:\\SPLUSWIN\\stat\\_Functio"
 [7] "D:\\SPLUSWIN\\s\\_Functio"
 [8] "D:\\SPLUSWIN\\s\\_Dataset"
 [9] "D:\\SPLUSWIN\\stat\\_Dataset"
[10] "D:\\SPLUSWIN\\splus\\_Dataset"
[11] "D:\\SPLUSWIN\\library\\trellis\\_Data"
```

Now list the individual objects in `/analyses/support/_Data`, which is in search position 2. The `objects` function (a replacement for an older function, `ls`) will do this.

```
> objects(2)
 [1] ".First"        ".Last.value"   ".Random.seed"  "backward"
 [5] "combined"      "combphys"      "desc.combined" "dnrprob"
 [9] "last.dump"     "mdemoall"
```

The `objects.summary` function will provide a more detailed listing. First let's find out how to call it.

```
> args(objects.summary)
function(names. = NULL, what = c("data.class", "storage.mode",
    "extent", "object.size", "dataset.date"), where = 1, frame =
    NULL, pattern = NULL, data.class. = NULL, storage.mode. =
    NULL, mode. = "any", all.classes = F, order. = NULL, reverse
     = F, immediate = T)
> objects.summary(where=2)
             data.class storage.mode      extent object.size
       .First   function     function          1         282
  .Last.value   describe     list             14       11904
 .Random.seed    numeric     integer          12          81
     backward data.frame     list       6201 x 9      280180
     combined data.frame     list     10281 x 150    7610275
```

```
      combphys data.frame     list     10281 x 166     7025122
 desc.combined   describe     list             152      129733
       dnrprob data.frame     list      10281 x 27     1283865
     last.dump       list     list               3         353
      mdemoall data.frame     list       1757 x 14      136496
                  dataset.date
        .First 96.04.11   6:28
   .Last.value 97.04.11  10:18
  .Random.seed               0
      backward 96.04.11   6:31
      combined 97.04.08  14:56
      combphys 97.04.11  10:18
 desc.combined 97.04.08  15:01
       dnrprob 96.09.17  17:23
     last.dump 97.03.06  14:07
      mdemoall 97.04.11  10:18
```

For examples to follow we will use the data frames `pbc` and `prostate`. You may obtain these from the UVa web page under `Statistical Computing Tools ... Datasets`. The file suffixes are `.sdd` so they may be easily imported as S-PLUS transport files using `File ... Import`. Let us suppose these datasets have already been imported into the current project area's `_Data` area. If you are using R or a recent version of the Hmisc library (with `wget.exe` installed if using Windows) you can easily download and access datasets from the University of Virginia web site using the Hmisc library's `getHdata` function.

```
> getHdata(prostate)    # downloads, imports, runs cleanup.import
> find(prostate)
[1] "_Data"
```

First let's examine the variables in `prostate` using the `describe` function in Hmisc. We will first call `describe` on individual variables. As `prostate` has not yet been `attach`ed, we must prefix its variables with `prostate`.

```
> names(prostate)
 [1] "patno"  "stage"  "rx"     "dtime"  "status" "age"    "wt"     "pf"
 [9] "hx"     "sbp"    "dbp"    "ekg"    "hg"     "sz"     "sg"     "ap"
[17] "bm"     "sdate"
> describe(prostate$age)
prostate$age : Age in Years
   n missing unique  Mean .05 .10 .25 .50 .75 .90 .95
 501 1       41     71.46 56  60  70  73  76  78  80
lowest : 48 49 50 51 52, highest: 84 85 87 88 89
--------------------------------------------------------------------------------
> describe(prostate$rx)
prostate$rx : Treatment
   n missing unique
 502       0      4
placebo (127, 25%), 0.2 mg estrogen (124, 25%), 1.0 mg estrogen (126, 25%)
5.0 mg estrogen (125, 25%)
--------------------------------------------------------------------------------
```

In this example `names(prostate)` gave us the variables in the data frame and `describe( prostate$age )` and `describe( prostate$rx )` some basic statistics on a couple of variables. `describe` recognizes automatically the type of variable (continuous, categorical (`factor`), or binary) and gives appropriate descriptive statistics (mean and quantiles, frequency table[1], or proportion, respectively), Except for binary variables, the 5 lowest and highest unique values are also given, and for any variable the sample size, number of unique values, and number of missing values is given. When the `impute` function has been used to impute missing values with "best guesses", `describe` prints the number of imputed values. When the variable was imported from SAS using `sas.get`, special missing values were present, and the `special.miss` option was used, `describe` will also report the frequency of the various special missing values.

Notice that since `prostate` is a data frame, we are using the `$` notation to refer to its components. This can be rather inconvenient and cumbersome. To make things simpler, we can use the `attach` function to attach the data frame in position one (or two, or whatever) in the search list. By default, `attach` will place objects (which should be data frames or lists) in position 2. The remaining items move down one position.

```
   > attach(prostate)  # Default placement is search position 2
   > search()
```

```
 [1] "_Data"
 [2] "prostate"
 [3] "c:/analyses/support/_Data"
 [4] "D:\\SPLUSWIN\\library\\Design\\_Data"
 [5] "D:\\SPLUSWIN\\library\\hmisc\\_Data"
 [6] "D:\\SPLUSWIN\\splus\\_Functio"
    . . . .
> describe(age)
age : Age in Years
   n missing unique  Mean .05 .10 .25 .50 .75 .90 .95
 501 1       41      71.46 56  60  70  73  76  78  80
lowest : 48 49 50 51 52, highest: 84 85 87 88 89
------------------------------------------------------------------------------
```

When the data frame (or any other recursive object, e.g., a `list`) is attached to the search list all its components can be accessed directly. This is the case regardless of the position on the search list. The advantage of using position one is that if you have another version of a variable in another dataframe or directory in the search list, then you can be sure you are operating on the intended version since the search list is accessed sequentially (i.e., we could have used `attach(prostate,pos=1,use.names=F)`). However, this will use more memory.

If the object is attached in position one, all objects created from now on will be kept in memory and disappear when we quit S-Plus or `detach` the object unless we intstruct it to save them (using for example `detach(1, 'prostate')`). Keep in mind that for large data frames the `attach` function may take a while to take effect and it will use a lot of memory.

Another way to make `attach` use less memory is to specify the `use.names=F` parameter[2]. By default, `attach`ing a data frame causes the `row.names` attribute of the data frame to be copied to

[1]If the variable has more than 20 unique values, the frequency table is omitted.
[2]R does not have this parameter, and does not put data frame `row.names` as `names` attribute of vectors.

each object within the frame, as that object's `name` attribute. When for example the `row.names` represent a subject ID, this can be helpful in identifying observations. But this can result in a doubling of memory usage. It is more efficient to associate `names` with only the variables whose observations you need to identify, or to just reference the `row.names`. The example below illustrates these.

```
> attach(titanic, use.names=F)
> record.id ← row.names(titanic)
> names(pclass) ← names(age) ← record.id
> # This isn't so effective here as row.names(titanic) were just
> # record numbers in character form, not passenger names
> # We could have done names(pclass) ← name
```

The function to take the data frame off the search list is `detach`. It has two arguments, `what` and `save`. `what` is usually a number denoting a postion in the search list and `save` could be a character string with the name of the object where we will store the (possibly) modified data frame.

```
> attach(prostate,pos=1,use.names=F)
> ageg50 ← age[age>50]
> length(ageg50)
[1] 497
> sqrt.age ← sqrt(age)
> length(sqrt.age)
[1] 502
> detach(1,save="pros")
Deleted before detaching: ageg50
```

Here we had the data frame `prostate` attached in position one. We created two new vectors, `ageg50` and `sqrt.age`. Since `ageg50` is shorter than the rest of the variables in the data frame it was deleted before detaching and not added to the new data frame `pros`.

```
> names(pros)
 [1] "patno"    "stage"    "rx"       "dtime"    "status"   "age"
 [7] "wt"       "pf"       "hx"       "sbp"      "dbp"      "ekg"
[13] "hg"       "sz"       "sg"       "ap"       "bm"       "sdate"
[19] "sqrt.age"
```

`sqrt.age` is a new variable. We could have also said `detach(prostate,save=F)` which would have deleted `sqrt.age` before detaching. This form works much faster than trying to save new variables. There is a way to save the value of `ageg50` with the dataframe by making it into a *parametrized* dataframe. See Spector's book page 37 for an example. Whether it makes any sense to do this is another matter. Also, we question whether it is useful to create easily derived variables such as `sqrt.age`, as `sqrt(age)` may be used in any future S expression where `age` is analyzed. See Section 4.4.3.

4.1.2 Subsetting Data Frames

In many cases, one analyzes all of the observations and most of the variables in a data frame. If a subset of the data needs to be analyzed for a small part of the job, one can easily process temporary subsets as in the following examples.

```
plot(age[sex=='male'],height[sex=='male'])
s ← sex=='male'
plot(age[s], height[s])  # equivalent to last example
f ← lrm(death ∼ age*height, subset=sex=='male')
```

When you want to subset the observations or variables in a data frame for an entire sequence of operations, it may be better to subset the entire data frame. You can do this by creating a new data frame using

```
df.males ← df[df$sex=='male',]
```

but more typically by `attach`ing a subset of the data frame. Here are several examples. One of them uses the `%nin%` operator in the Hmisc library, which returns a vector of `T` and `F` values according to whether the corresponding element of the first vector is not contained in the second vector. `%nin%` is the opposite of the `%in%` operator in Hmisc.

```
attach(df[,c('age','sex')]) # only make age and sex available - save memory
attach(df[c('age','sex')])  # another way to subset variables using fact
                            # that df is a list in addition to a data frame
attach(df[,Cs(age,sex)])    # use the Cs function in Hmisc to save quoting
attach(df[df$sex=='male',]) # get all variables but only for males
                            # need df$sex instead of sex because attach
                            # hasn't taken effect yet
attach(df[1:100,c(1:2,4:7)])# get first 100 rows and variables 1,2,4,5,6,7
attach(df[,-4])             # don't get variable number 4
attach(df[,names(df) %nin% c('age','sex')])  # get all but age and sex
attach(df[df$treat %in% c('a','b','d'), names(df) %nin% Cs(age,sex)])
                            # get rows for treatments a,b,d and all but 2 var
attach(df[!(is.na(df$age) | is.na(df$sex)),]) # omit rows containing NAs
attach(df[!is.na(df$age+df$height),])         # shortcut if both vars numeric
```

After the `attach` is in effect, referencing any of the included variables will reference the desired subset of rows of the data frame which were attached.

In some ways a more elegant approach is to use the Hmisc `subset` function which is a copy of the R `subset` function. The advantages of `subset` are that variable names do not need prefixing by `dataframe$`, and `subset` provides an elegant notation for subsetting variables by looking up column numbers corresponding to column names given by the user, which allows consecutive variables to keep or drop to be specified. Here are some examples:

```
> # Subset a simple vector
> x1 <- 1:4
> sex <- rep(c('male','female'),2)
> subset(x1, sex=='male')
[1] 1 3

> # Subset a data frame
> d <- data.frame(x1=x1, x2=(1:4)/10, x3=(11:14), sex=sex)
> d
  x1  x2 x3    sex
1  1 0.1 11   male
2  2 0.2 12 female
```

```
3  3 0.3 13   male
4  4 0.4 14 female

> subset(d, sex=='male')
  x1  x2 x3  sex
1  1 0.1 11 male
3  3 0.3 13 male

> subset(d, sex=='male' & x2>0.2)
  x1  x2 x3  sex
3  3 0.3 13 male

> subset(d, x1>1, select=-x1)
   x2 x3    sex
2 0.2 12 female
3 0.3 13   male
4 0.4 14 female

> subset(d, select=c(x1,sex))
  x1    sex
1  1   male
2  2 female
3  3   male
4  4 female

> subset(d, x2<0.3, select=x2:sex)
   x2 x3    sex
1 0.1 11   male
2 0.2 12 female

> subset(d, x2<0.3, -(x3:sex))
  x1  x2
1  1 0.1
2  2 0.2
> attach(subset(d, sex=='male' & x3==11, x1:x3))
```

4.1.3 Adding Variables to a Data Frame without Attaching

Attaching your data frame in search position one will allow you to add or change any number of variables. There are other ways to add new variables to an existing data frame if you don't want to have the overhead of attaching it. Suppose that we wish to add two variables, `x1` and `x2`, to an existing data frame called `df`. Here are two approaches:

```
df$x1 ← pmax(df$y1, df$y2, df$y3)
df$x2 ← (df$y1 + df$y2 + df$y3) / 3
df ← data.frame(df, x1=pmax(df$y1, df$y2, df$y3),
                    x2=(df$y1 + df$y2 + df$y3)/3)
```

4.1.4 Deleting Variables from a Data Frame

Setting a variable to the `NULL` value will cause it to be deleted permanently from the list[3]:

```
df$age ← NULL
df[c('age','sex')] ← NULL   # delete 2 variables
df[Cs(age,sex)]    ← NULL   # same thing
```

To remove variables that are inside a data frame currently attached in position 1, use statments such as the following.

```
age ← NULL
sex ← pressure ← NULL
```

Do not use `rm(varname)`, `remove('varname')`, or `remove('df$varname')` to remove a variable from a data frame. Use one of the two methods above, or use the object explorer.

4.1.5 A Better Approach to Changing Data Frames: `upData`

Attaching data frames in search position one turns out to be one of the most confusing and dangerous things to new S-Plus users. New users tend to forget to detach search position one, and attach a data frame again in search position one, which can at worst corrupt the search list and at best make things very confusing. The Hmisc `upData` function provides a unified framework for updating a data frame. It accomplishes the following, listed in order in which changes are executed by the function:

1. optionally changes names of variables to lower case
2. renames variables
3. adds new variables
4. recomputes existing variables from the original variable and/or from other variables in the data frame
5. changes the storage mode of variables to the most efficient mode (as done with `cleanup.import`) (by default, floating point variables are stored in single precision,[4] always integer-valued variables are stored as integers)
6. drops variables
7. adds, changes, and combines `levels` of `factor` variables
8. adds or changes variable `label` attributes
9. adds or changes variable `units` (units of measurement) attributes

Here is an example.

[3]This is assuming that the data frame is in a directory that is in `search` position 1, e.g., the `_Data` directory. This will not work if `store()` is in effect.

[4]This is not in R, which has no single precision.

```
dat ← data.frame(a=(1:3)/7, y=c('a','b1','b2'), z=1:3)
dat2 ← upData(dat, x=x^2, x=x-5, m=x/10,
               rename=c(a='x'), drop='z',
               labels=list(x='X', y='test'),
               levels=list(y=list(a='a',b=c('b1','b2'))))
# Note that levels b1 and b2 of y are collapsed to 'b'

Input object size:   662 bytes;  3 variables
Renamed variable     a  to x
Modified variable    x
Modified variable    x
Added variable       m
Dropped variable     z
New object size:     818 bytes;  3 variables

dat2
          x y          m
1 -4.979592 a -0.4979592
2 -4.918367 b -0.4918367
3 -4.816327 b -0.4816326

describe(dat2)

dat2

 3  Variables      3  Observations
---------------------------------------------------------------------------
x : X
 n missing unique   Mean
 3 0       3     -4.905

-4.816 (1, 33%), -4.918 (1, 33%), -4.980 (1, 33%)
---------------------------------------------------------------------------
y : test
 n missing unique
 3       0      2

a (1, 33%), b (2, 67%)
---------------------------------------------------------------------------
m
 n missing unique    Mean
 3 0       3     -0.4905

-0.4816 (1, 33%), -0.4918 (1, 33%), -0.4980 (1, 33%)
---------------------------------------------------------------------------
```

A safe approach is to return the result of `upData` into a new object name, then to check the object (using `describe`, for example) and to copy it back into the original data frame name. For example,

```
dat ← dat2
rm(dat2)       # remove data frame created by upData
```

There are two ways to turn a variable into a `factor` using `upData`. First, you can use `levels = list(varname = list(...))` as was done above. This is flexible because you can combine levels into "super levels".[5] Note that new levels are on the left hand side of equal signs, and that these only need to be in quotes if they are not legal S names. The second approach involves recomputing a variable, for example:

```
d ← data.frame(a=1:2)
d ← upData(d, a=factor(a,1:2,c('a','b')))
```

4.1.6 `assign` and `store`

Up to now we have been storing any new objects that we created permanently in the `.Data` subdirectory in S-Plus. Another way to work is to attach the data frame in position one and create temporary objects that we may need with the option to save them later along with the dataframe. If we wanted to save them independently of the dataframe, or you want to put an object in any directory of your choice, the function to use is `assign`.

```
> args(assign)
function(x, value, frame, where = NULL, ...)
> assign("ageg50",ageg50,where="_Data")  # or .Data in UNIX
> # use assign(...., where='c:/mine/project/_Data') to use another directory
```

This way of working has the advantage to let us create objects temporarily and save only those that we need. That is very useful in an interactive system such as S where one tends to create objects with names like `x`, `y`, `m`, `f`, etc. The disadvantage is that you have to attach the dataframe in position one, which uses a lot of memory and may slow us down. The `store` function in Hmisc can help you keep your S-Plus `.Data` directory from filling up with temporary objects. It can also help in storing objects in permanent locations of your choosing. For the latter purpose `store` works similarly to `assign` except that the order of its arguments is different.

```
> args(store)
function(object, name = as.character(substitute(object)), where = ".Data")
```

If you type `store()` with no arguments, then a temporary directory is attached in position one; that way any new objects reside in this temporary area until you quit S-Plus or decide to store them, either in a dataframe or directly into a subdirectory.

```
> store()
> attach(prostate)
> ageg50 ← age[age>50]
> sqrt.age ← sqrt(age)
```

```
> search()
 [1] "D:\\SPLUSWIN\\TMP\\file5C9.AD4"
 [2] "_Data"
 [3] "prostate"
 [4] "c:/analyses/support/_Data"
 [5] "D:\\SPLUSWIN\\library\\Design\\_Data"
```

[5]This is done implicitly using the S-Plus `merge.levels` function; see its documentation for details.

```
 . . . .
> objects()
[1] ".Last"       ".Last.value" "ageg50"      "sqrt.age"
```

```
> pros ← data.frame(prostate,sqrt.age)
> store(pros)  # adds a new variable sqrt.age to the prostate data frame
               # and store the result in a new permanent data frame pros
Warning messages:
  "pros" assigned on database 3 but hidden by an object of the same name on
        database 1 in: assign(name, object, where = where, immediate = T)
> store(ageg50,"age.greater.than.50")  # store age50 under the name
                                       # age.greater.than.50, permanently
```

If you have used `store()`, you can use another function, `stores`, which is also documented with the `store` function. `stores` causes the list of objects (without quotes) to be copied from the temporary directory in search position one to `_Data` or `.Data`. Here is an example program that stores two fit objects in the project's `_Data` directory.

```
df ← sas.get('/my/sasdata','sasmem',recode=T)
store()
fit1 ← lrm(death ~ age*sex)
fit2 ← ols(blood.pressure ~ age*sex)
stores(fit1,fit2)  # same as store(fit1);store(fit2)
```

4.2 Managing Project Data in R

R uses a different mechanism from S-Plus for managing objects that does away with the need to use `store()`. By default, R stores all the objects created in your session in a single file `.RData`. When running R interactively, R asks whether you want to update `.RData` to contain newly created objects upon termination of the session. As many of the objects are temporary, it is often best to answer `n` to this question and not use the `.RData` mechanism. It is appropriate however to store some of your newly created data frames and selected other objects (such as regression fit objects that took significant execution time to create) permanently. This can be done using R's `save` function, and if `save`'s `compress` option is used, the resulting file will be stored very compactly. Here is an example session that creates and stores two objects.

```
a ← lm(y ~ x1 + x2)
mydata ← read.csv('/tmp/mydata.csv')  # import, creating data frame
save(a, b, file='my.rda', compress=TRUE)
# same as save(list=c('a','b'), file='my.rda', compress=TRUE)
```

To retrieve the two objects in a future session use

```
load('my.rda')
```

4.2.1 Accessing Remote Objects and Different Objects with the Same Names

Ocassionally, we may want to have access to objects stored in some other directory but we don't really want to attach that directory. For example, a fitted model could be stored in some subdirectory and we need to get predicted values from that model using data in the current directory. The `get` function works very nicely in this case.

```
> z ← get("model", where=" ... ")  # 2nd argument: full or relative path
```

Now the `model` object is available in the temporary directory (under the name `z`) and we can use it for our needs.

Note: As mentioned above, attaching very big data frames takes a lot of memory and may cause S to slow down significantly unless you have a great deal of RAM installed. It is best to attach only part of the dataframe with the variables and observations you need for each particular problem, if these are a small subset of the entire data.

If you have variables with the same name in the data frame attached in position one and in other directories on the search list as well and you are getting strange or unexpected answers it may be the case that you are not doing your calculations on the desired variables. For example, suppose that the data frames `prostate` and `pbc` have different versions of the age variable and they are attached in positions three and two respectively. If we mean to get the statistics for `prostate$age`, `describe(age)` won't do it. This can be problematic if we have to combine data frames from different directories. The `find` function can be helpful here.

```
> attach(prostate)
> attach(pbc)
> search()
 [1] ".Data.temp15097"
 [2] "pbc"
 [3] "prostate"
 [4] ".Data"
 . . . .
> find(age)
[1] "pbc"      "prostate"
```

The `masked` function is also worth trying. You can have complete control in accessing the desired versions of objects using the `get` function, or using for example `pbc$age` and `prostate$age`.

4.2.2 Documenting Data Frames

For long–term projects one frequently needs to document how a data frame came to be, which data were corrected, which data remain suspicious, etc. Besides the obvious method of editing a text document in your project directory, there are at least two ways to have S manage such documentation by linking it to the object.

1. You can attach an attribute to the data frame object. The `comment` function in the Hmisc library can be used for this. `comment` attaches or retrieves a `comment` attribute to the object. You can also invent new attributes. Here are two examples:

```
> comment(dframe) ← 'From SAS Dataset /myproject/mysas on machine A'
> comment(dframe)     # replays the text string

> attr(dframe,'doc') ← 'From SAS Dataset /myproject/mysas on machine A'
> attr(dframe,'doc')  # prints doc attribute
```

See the definition of `comment` to see how to package the `doc` attribute more elegantly.

2. You can create a help file for any object you create, so that typing `help(objectname)` or `?objectname` will replay the help file[6]. The help file can contain any text of your choosing, and it should be in a `_Help` directory underneath the project's `_Data` directory (`.Data/.Help` for UNIX).

4.2.3 Accessing Data in Windows S-Plus

S-Plus Windows has an "Object Explorer" that can access data frames (and other objects) and their variables in a way similar to how Windows Explorer traverses directories and opens files. Users can create object explorers that point to one or more data frames in a mixture of `_Data` directories. Suppose for example that we want to create an object explorer that pointed to all data frames, lists, and matrices in directory `c:\projects\one\_Data`. Here are the required steps. If an object explorer is already open you may want to skip step 1 and use that explorer as a starting point.

1. Click on `File ... New ... Object Explorer`. You'll see a new default object explorer named `Object Explorer 1` pop up.

2. Left click on `SearchPath` in the left pane of the object explorer to see the directories currently accessible. Right click on `SearchPath` and then on `Attach Database` to add a new area that's not listed. Fill the full path name in the empty box, or use `Browse`. If you don't want the area put in search position 1 (i.e., you don't want to put all new variables there), select another search position such as 2. Right click in the empty space in the left pain of the `Object Explorer`. Click on `Insert Folder` and name it `SearchPath`. Right click on the new `SearchPath` folder and select `Advanced`. Under `Interface Objects` select `SearchPath` and click on `OK`. Right click on the `+` next to the `SearchPath` folder and right click on `SearchPath` under `SearchPath`. Select `Attach Database` and specify the directory to add to the search list. Choose search position one for this database if this is where you will be *writing* data.

3. Now you will see the above directory listed with a number, the search position, after its name in the right pane of `Object Explorer1`.

4. If you want to specify which kinds of objects in the new area are to be listed by the object explorer, right click while the the cursor is in the open area in the left pane and select `Filtering`. Now you may see a `+` in front of the left pane's `data.frame` entry, and the list of data frames, vectors, and lists on the right pane. Right click on `Data` and select `Advanced`. Shift-left-click to add databases to the existing filter list, or regular left click to replace the ones already selected with your new choice. Now `Data` will show the objects from this new directory.

[6] In S-Plus 3.3 or earlier for Windows this only works if the object name is a legal DOS name with no suffix; otherwise S-Plus will search for a help file with the name equal to the shortened DOS–version file name, and it won't find it.

5. Before saving your new object explorer permanently, you may want to modify its name to be more descriptive. While the cursor is in the right pane of your explorer, right click and select `Right Pane ...`. Click on the `Explorer` tab and type what you want in the `Name` and `Description` fields and click on `OK`.

6. To save this `Object Explorer` click on `File` then `Save As ...`. You can save it in a central `_Prefs` area or under your project area (by navigating the window which just popped up). For the latter location, click on the ↑ folder to get to the directory and/or disk drive you desire. For this example we get to `c:\projects\one`. You can override the `File name` box in the window, to e.g. `Project A.sbf`. Be sure to include the `.sbf` at the end of the name.

7. When you exit and re–start S-Plus, you can pop–up your project–specific object explorer by clicking on `File`. At this point, your object explorer may be on the list at the bottom of the menu so that you can just double click on that. If it's not there, you can click on `Open` then search for the directory containing it, e.g., `c:\projects\one`.

To use your object explorer, click the `+` to the left of `data.frame` (`data` for `object explorer`, which will expand this item to list all the data frames present. You can double–click on any of the data frames in this list on the left pane to expand it, i.e., to display its variables and some of their attributes. To display the actual data, double–click on the data frame name in the right pane. This will open a window containing a data sheet.

S-Plus has a facility for saving and restoring `workspaces`. This is a good way to organize not only databases as was discussed above, but also to link them with reports, graphs, and data sheets. You can set up the workspace so that when it is opened, then active databases not needed by the workspace are detached. Also, when you open a workspace, opened reports and other files are automatically closed.

4.3 Miscellaneous Functions

4.3.1 Functions for Sorting

Table 4.1 displays the functions available for sorting. The obvious choice for sorting a vector

Table 4.1: *Functions for Sorting*

Function	Description	Comments
`sort`	`sort(x)`	sorts elements of a vector
`order`	`order(c(x,y,...))`	returns the order permutation
`rev`	`rev(x)`	reverses the elements of an object

is `sort`. It takes a vector as an argument and returns the vector sorted in ascending order. An optional argument `na.last` determines if missing values will be discarded, placed at the begining or at the end of the sorted vector (Use `na.last=NA, F` or `T` respectively). If it is desired to sort the vector in descending order use `rev(sort(x))`.

`order` is more flexible than `sort`. It returns the order permutation of a vector, that is, its first element is the index corresponding to the smallest element, the second is the index corresponding to the second smallest element, etc. Thus `x[order(x)]` is equivalent to `sort(x)`. The advantage of `order` is that it can operate on more than one vector simultaneously. For example `order(x,y)` will give an order based on `x`; ties are resolved according to the values of `y`. To sort a single numeric vector in reverse order you can use `-sort(-x)` or `x[order(-x)]`. In the following we sort `x` alphabetically by `state`, and within `state` by descending `median.income`.

```
i ← order(state, -median.income)
xs ← x[i]
```

4.3.2 By Processing

You can process observations in groups according to combinations of stratification variables using subscripts, as in the following where we compute the mean `age` stratified by `sex`. Assuming that `sex` is a `factor` object, we can fetch the list of its possible values using the `levels` function.

```
> means ← single(2)  # set aside one position per sex code
> i ← 0
> for(sx in levels(sex)) {
+   i ← i+1
+   s ← sex==sx
+   means[i] ← mean(age[s])
+ }
```

This method is tedious but flexible. We can add logic, for example, to also compute the grand mean.

```
> means ← single(3)
> i ← 0
> for(sx in c('ALL',levels(sex))) {
+   i ← i+1
+   s ← sex==sx | sx=='ALL'
+   means[i] ← mean(age[s])
+ }
```

When `sx=='ALL'`, `s` is a vector of all `T`s, indicating that all observations should be used in calculating the grand mean.

The above examples are not efficient when typical by processing is to be done. Instead, the function `tapply` can be used in many situations. In the `pbc` file, we could get the mean age by stage by doing

```
> tapply(age,stage,mean,na.rm=T)
        1        2        3        4       NA
 46.84101 49.46583 48.96247 53.76548 57.33333
```

The syntax is similar to that of `apply` that we discussed earlier. As we can see, there is no need to sort the data previously. If we wanted to get means by the combination of the levels of two variables we could use

```
> tapply(age,interaction(stage,status,drop=T),mean,na.rm=T)
      1.0      2.0      3.0      4.0      1.1      2.1      3.1
 46.42739 48.75489 47.65943 51.01578 50.77036 51.59864 51.86719
```

```
       4.1
  55.72956
```

The drop=T argument indicates to drop combinations with no observations in them. Better still, you can easily produce multi–dimensional summaries in array form.

```
> tapply(age,list(stage,status),mean,na.rm=T)
          0        1
 1 46.42739 50.77036
 2 48.75489 51.59864
 3 47.65943 51.86719
 4 51.01578 55.72956
NA 61.00000 55.50000
```

The builtin function by is an excellent way to do by–processing on all the variables in a data frame when the summarization function operates on data frames. Here are some examples.

```
> by(age, list(Stage=stage), FUN=describe, descript=label(age))
> # descript passed to describe.  list(Stage=stage) allows nice labels.
Stage:1
Age

 1  Variables      21  Observations
---------------------------------------------------------------------------
x : Age
  n missing unique  Mean   .05   .10   .25   .50   .75   .90   .95
 21 0       21     46.84 34.60 34.99 38.49 46.35 53.00 59.00 61.99

lowest : 28.88 34.60 34.99 36.00 38.40, highest: 55.57 56.57 59.00 61.99 62.52
---------------------------------------------------------------------------
Stage:2
Age

 1  Variables      92  Observations
---------------------------------------------------------------------------
x : Age
  n missing unique  Mean   .05   .10   .25   .50   .75   .90   .95
 92 0       87     49.47 33.83 36.58 42.46 49.00 56.39 61.96 63.74

lowest : 30.28 30.57 33.15 33.48 33.62, highest: 63.88 66.41 67.57 68.51 75.01
. . . .

> by(pbc[Cs(age,bili)], list(stage,status), FUN=summary) # or FUN=describe
stage:1
status:0
       age            bili
 Min.   :28.9   Min.   :0.500
 1st Qu.:38.4   1st Qu.:0.600
 Median :46.0   Median :0.700
 Mean   :46.4   Mean   :0.805
 3rd Qu.:54.3   3rd Qu.:1.000
 Max.   :62.5   Max.   :1.400
-------------------------------
stage:2
status:0
       age           bili
 Min.   :30.3   Min.   : 0.30
 1st Qu.:41.8   1st Qu.: 0.60
 Median :48.9   Median : 0.70
 Mean   :48.8   Mean   : 1.66
 3rd Qu.:56.2   3rd Qu.: 1.40
 Max.   :75.0   Max.   :18.00
-------------------------------
stage:3
status:0
. . . .
```

The `summary.formula` function in Hmisc provides a general way to do by–processing (see Section 6.2). A related function is Hmisc's `summarize` function , which is designed to compute descriptive statistics stratified by one or more non–continuous variables. `summarize` creates a data frame useful for processing by other S functions, especially `trellis` graphics functions, as discussed in Section 11.4.3.

4.3.3 Sending Multiple Variables to Functions Expecting only One

Many of the common S function operate on vectors, e.g., `mean`, `quantile`, etc. You can operate on a series of variables or on all the variables in a data frame by looping over the variable names or subscripts, or by using the `lapply` and `sapply` functions. The `lapply` function applies a single function to every element of a list (e.g., every variable in a data frame), and returns a `list` as the final result, with one `list` element per variable. For example, let us create a data frame having two variables, and apply the `quantile` function to each variable:

```
> set.seed(193)
> d ← data.frame(x1=rnorm(1000), x2=runif(1000))
> lapply(d, quantile, probs=c(.25,.5,.75))

$x1:
        25%        50%       75%
 -0.6290425 0.07898111 0.6710022

$x2:
       25%       50%       75%
 0.2410647 0.4988862 0.7453622
```

The `sapply` function formats the results differently. It will produce a vector if the function is single–valued. Here it returns a matrix:

```
> sapply(d, quantile, probs=c(.25,.5,.75))
              x1        x2
[1,] -0.62904248 0.2410647
```

```
[2,]  0.07898111 0.4988862
[3,]  0.67100218 0.7453622
```

sapply was used in Section 3.6 to plot the number of missing values for all of the variables in a data frame.

When you need to perform a repetitive operation for several variables and you need to be able to access the labels or names of the variables during processing, Hmisc's llist function (documented with the label function) can help. llist tries to use the best available labels for each variable in a list, and it allows you to access these labels using the label function. Here is an example where a series of variables are plotted against a common variable, and each plot is titled with the current variable's label. The sapply (or lapply) methods are preferred if you want to store the result of the function evaluations into a global result.

```
sapply(llist(age, height, pmin(weight,200)),
       function(x) {
         plot(x, blood.pressure)
         title(label(x))
       })

# Equivalent to:
for(x in llist(age, height, pmin(weight,200))) {
  plot(x, blood.pressure)
  title(label(x))
  })
```

The S builtin function aggregate is another good method for performing separate analyses of multiple variables in a data frame, with simultaneous stratification on by–variables:

```
>  attach(pbc)   # so can reference stage without prefix
>  aggregate(pbc[Cs(bili,albumin,age)], stage, FUN=mean)

  stage bili albumin  age
1     1 1.36    3.71 46.8
2     2 2.45    3.61 49.5
3     3 2.83    3.59 49.0
4     4 4.43    3.30 53.8
5    NA 2.75    3.32 57.3

>  # Same as aggregate(data.frame(bili,albumin,age), stage, FUN=mean)
>  # since we're assuming pbc is attached
```

You must give aggregate a stratification variable. If you want to use aggregate to process multiple variables with no stratification, give it a stratification variable that is constant, e.g., rep(1, length(age)). When there are multiple stratification variables, enclosing them in the Hmisc llist function will cause aggregate to use their names in the data frame it forms. For example, se

```
>  aggregate(data.frame(systolic,diastolic), llist(race,sex), mean)
```

aggregate can only use FUNs that return a single value although it is able to compute this single value on several response variables. aggregate does not preserve numeric stratification variables (it converts them to factors) so it is not suitable for aggregating some datasets for plotting with xyplot. See p. 234 for a comparison of methods for aggregating data for plotting.

4.3.4 Functions for Data Manipulation and Management

These functions are listed in Table 4.2. `seq` is a generalization of the "`:`" operator. It allows us to specify a starting point, an ending point and the distance between them, or alternatively, the length of the resulting vector. We could also specify `seq(along=x)` which will produce the sequence `1:length(x)`, even if `length(x)=0` (i.e. `x` is a `NULL` or `numeric(0)` vector). `duplicated` returns a logical vector with `T` if the index corresponds to a duplicate value, and `F` if not. `unique` can be used for example to find the five smallest values of a vector: `sort(unique(x))[1:5]`. (See the output of `describe`).

Table 4.2: *Functions for Data Manipulation and Management*

Function	Description	Comments
`seq`	`seq(a,b,by=z)`	creates a sequence from `a` to `b` with an increment of `z` in between them
`duplicated`	`duplicated(x)`	checks for duplicate values
`unique`	`unique(x)`	returns a vector like `x` without repeated values
`match`	`match(x,table)`	returns the position in `table` of the elements of `x`
`table`	`table(x,y,...)`	
`abbreviate`	`abbreviate(x,...)`	abbreviate text
`pmatch`	`pmatch(x,table)`	partial matching
`expand.grid`	`expand.grid(...)`	easy way to construct dataframes
`cut2`	`cut2(x,...)`	an improved version of `cut` (Hmisc)
`merge`	`merge(x, y, by, by.x, by.y, ...)`	merge two data frames
`find.matches`	`find.matches(x,y,...)`	find closest matches to observations (Hmisc)
`llist`	`llist(x,y,...)`	labeled list of several variables (Hmisc)
`sedit`		advanced character string manipulation (Hmisc)
`casefold`	`casefold(strings)` or `casefold(strings,upper=T)`	change case of vector of character strings
`substring`	`substring(strings,start,end)`	subset char. strings
`combine.levels`	`combine.levels(x)`	combine infrequent levels (Hmisc)
`score.binary`		recoding (Hmisc)
`recode`		recoding (Hmisc)
`merge.levels`		merge levels of factor
`reShape`	`reShape(...)`	re–shape vectors or matrices (Hmisc)

The function `match` looks up the elements of `x` in `table` for each element of `x`; when it finds a match, it returns the position in `table` of the match. This can be useful for instance, to join objects, holding the places of non–matching values with missing values.

```
> x ← 1:10
> y ← seq(5,15,by=2)
> match(x,y)
 [1] NA NA NA NA  1 NA  2 NA  3 NA
> z ← cbind(x,y=y[match(x,y)])
> z
       x  y
 [1,]  1 NA
 [2,]  2 NA
 [3,]  3 NA
 [4,]  4 NA
 [5,]  5  5
 [6,]  6 NA
 [7,]  7  7
 [8,]  8 NA
 [9,]  9  9
[10,] 10 NA
```

If x and y were dataframes and the matching had been done in their row.names attribute, the result would have been a merged dataframe with NAs for the variables in y where the observation did not match an observation in x. See [3, page 58] for a more detailed example. The merge function is a more general solution to this problem.

abbreviate is especially useful for shortening variable names, row.names, or variable labels, for making output fit on a regular page size. Here are some examples.

```
names(df) ← abbreviate(names(df))         # abbreviate all data frame names
row.names(df) ← abbreviate(row.names(df)) # abbreviate row names
label(x) ← abbreviate(label(x))           # abbreviate single label
prostate2 ← prostate
for(i in 1:length(prostate2))
  label(prostate2[[i]]) ← abbreviate(label(prostate2[[i]]))
```

The function expand.grid is very useful to produce dataframes with a combination of all levels of specified variables.

```
> z ← expand.grid(age=median(age),rx=levels(rx),bm=c(0,1))
> z
  age              rx bm
1  73         placebo  0
2  73 0.2 mg estrogen  0
3  73 1.0 mg estrogen  0
4  73 5.0 mg estrogen  0
5  73         placebo  1
6  73 0.2 mg estrogen  1
7  73 1.0 mg estrogen  1
8  73 5.0 mg estrogen  1
```

The cut2 function in Hmisc can be used to categorize variables. Unlike cut, the default S function, cut2 returns a factor that is much more useful for analytic purposes. cut2 also has more options and creates better labels for levels of the resulting factor variable.

```
> table(cut2(prostate$age,g=5))
```

```
  [48,68) [68,72) [72,74) [74,77) [77,89]
       96      92      89     123     101
```

The main argument to `cut2` is a numeric vector we wish to categorize; it then classifies its argument into `g` intervals with approximately the same number of observations in them. Instead of `g`, we could supply the desired cuts via the `cuts=` argument or the minimum number of observations in each group using the `m=` argument.

The `casefold` function was exemplified in Section 3.4.

The `substring` function is used for pulling apart pieces of character strings. For example, `substring('abc',1,2)` is `'ab'` and `substring('abc',2)` is `'bc'`. `substring` can be useful for restructuring complex data after input. For example, suppose that dates and times had been stored together in a single character value in a vector `x`:

```
> x
[1] "98/09/01 00:10" "98/09/01 14:17"
```

To get the date portion, we substring the first 8 characters of each string and convert it to internal date storage (`chron` object):

```
> d ← chron(substring(x,1,8), format='y/m/d')
> d
[1] 98/09/01 98/09/01
```

A time variable can be constructed from columns 10-14 of each of these strings. As these times did not include seconds, we paste 0 seconds on to the end of each time using the `paste` function.

```
> tm ← chron(times=paste(substring(x, 10, 14),':00',sep=''))
> tm
[1] 0.006944444 0.595138889
```

Times are stored in hours from midnight. 0.00694 = 10 minutes past midnight, 0.5951 = 14:17:00. Now the dates and times can be recombined into a single date/time `chron` object:

```
> y ← chron(d, tm)
> y
[1] 98/09/01 98/09/01
> print.default(y)
[1] 14123.01 14123.60
```

Dates are stored by default as the number of days from 1Jan1960. If the S `chron` library is attached, there are more features for printing dates and times.

```
> library(chron)
> tm
[1] 00:10:00 14:17:00
> y
[1] (98/09/01 00:10:00) (98/09/01 14:17:00)
```

Hmisc's `combine.levels` function is useful for restructuring the levels of a categorical variable, by combining levels have a small proportion of the total frequency. This can be useful for modeling when you want to prevent the construction of a multitude of dummy variables.

See Section 4.4 for examples where the `score.binary` and `recode` functions are used. See Sections 6.1 and 4.3.8 for examples of the `reShape` function.

4.3.5 Merging Data Frames

The `merge` function is a general function to do one–one, many–one, or many–many joining of two data frames using any number of matching variables. The help file for `merge` has some excellent examples. Let us consider a common setup where we have a data frame `base` containing baseline data (one record per subject) and a data frame `follow` containing multiple records per subject. Both data frames contain a subject identifier variable `id`. You can use the `merge` function to do a one–to–many matching merge of the two data frames as in the following example.

```
> base   ← data.frame(id=c('a','b','c'), age=c(10,20,30))
> follow ← data.frame(id=c('a','a','b','b','b','d'),
                    month=c(1, 2,  1,  2,  3,  2),
                    cholesterol=c(225,226, 320,319,318, 270))
> base
  id age
1  a  10
2  b  20
3  c  30

> follow
  id month cholesterol
1  a     1         225
2  a     2         226
3  b     1         320
4  b     2         319
5  b     3         318
6  d     2         270

> combined ← merge(base, follow, by='id', all.x=T)
> # Specify all=T or all.x=T, all.y=T to keep records w/no baseline data
> combined
  id age month cholesterol
1  a  10     1         225
2  a  10     2         226
3  b  20     1         320
4  b  20     2         319
5  b  20     3         318
6  c  30    NA          NA
```

One advantage of this storage format is that it works well with graphics commands in which `month` is on the x–axis. For example, we can use `trellis` to plot `cholesterol` profiles for all subjects with treatment groups in separate panels (if the `treatment` variable had been stored in the `base` data frame):

```
xyplot(cholesterol ~ month | treatment, groups=id,
       panel=panel.superpose, data=combined)
```

4.3.6 Merging Baseline Data with One–Number Summaries of Follow–up Data

Instead of duplicating baseline data to "spread" it with follow–up data, we often want to summarize the follow–up data into a single number for each subject, and merge this number with the baseline data. In the following example we summarize serial `cholesterol` measurements using two statistics: the maximum and the mean. We could easily summarize variables besides cholesterol and add them to the `chol.summaries` data frame below.

```
chol.mean  ← tapply(follow$cholesterol, follow$id, mean, na.rm=T)
chol.worst ← tapply(follow$cholesterol, follow$id, max,  na.rm=T)
chol.summaries ← data.frame(chol.mean,chol.worst,id=names(chol.mean))

> chol.summaries
  chol.mean chol.worst id
a     225.5        226  a
b     319.0        320  b
d     270.0        270  d

> combined ← merge(base, chol.summaries, by='id', all.x=T)
> combined
  id age chol.mean chol.worst
1  a  10     225.5        226
2  b  20     319.0        320
3  c  30        NA         NA
```

4.3.7 Constructing More Complex Summaries of Follow-up Data

Often serial data need summarizations that involve multiple variables simultaneously. In the following example, we have a single date variable and two follow-up measurements (`cholest` and `sys.bp`) and we want to save the date and the value of the last non-missing measurements of `cholest` and `sys.bp`. The example uses the Hmisc `mApply` function is a matrix version of `tapply`.

```
d ← data.frame(id=c('a','a','a','b','b','b','b'),
               mdate=chron(c('04/02/2001','04/04/2001','05/17/2002',
                 '07/06/2002','07/07/2002','08/03/2002','08/13/2002')),
               cholest  =c(210,NA,205, 248,252,251,NA),
               sys.bp   =c(141,136,NA, 152,NA, 149,151))
# For R use strptime(c('04/02/2001','''), format='%m/%d/%Y')
d

  id    mdate cholest sys.bp
1  a 04/02/01     210    141
2  a 04/04/01      NA    136
3  a 05/17/02     205     NA
4  b 07/06/02     248    152
5  b 07/07/02     252     NA
6  b 08/03/02     251    149
7  b 08/13/02      NA    151
```

```
attach(d)
x ← cbind(mdate, cholest, sys.bp)
g ← function(w) {
  mdate   ← w[,'mdate']
  cholest ← w[,'cholest']
  sys.bp  ← w[,'sys.bp']
  dcholest ← max(mdate[!is.na(cholest)],na.rm=T)
  cholest  ← mean(cholest[mdate==dcholest],na.rm=T)
  dsys.bp  ← max(mdate[!is.na(sys.bp)],na.rm=T)
  sys.bp   ← mean(sys.bp[mdate==dsys.bp],na.rm=T)
  c(dcholest=dcholest,cholest=cholest,dsys.bp=dsys.bp,sys.bp=sys.bp)
}

w ← mApply(x, id, g)
w

  dcholest cholest dsys.bp sys.bp
a    15477     205   15069    136
b    15555     251   15565    151

w ← data.frame(w, id=dimnames(w)[[1]])
w$dcholest ← as.chron(w$dcholest)
# For R: strptime(w$dcholest,format='%Y-%m-%d')
# For S-Plus 6: use dates(w$dcholest)
w$dsys.bp <- as.chron(w$dsys.bp)
w

  dcholest cholest  dsys.bp sys.bp id
a 05/17/02     205 04/04/01    136  a
b 08/03/02     251 08/13/02    151  b
```

The data frame w can be merged (by id) with baseline data as before.

Alternatively, the builtin by function may be used to give useful printed output. However by does not store the result in a useful format.

```
g ← function(w) {
  mdate   ← w$mdate
  cholest ← w$cholest
  sys.bp  ← w$sys.bp
  dcholest ← max(mdate[!is.na(cholest)],na.rm=T)
  cholest  ← mean(cholest[mdate==dcholest],na.rm=T)
  dsys.bp  ← max(mdate[!is.na(sys.bp)],na.rm=T)
  sys.bp   ← mean(sys.bp[mdate==dsys.bp],na.rm=T)
  data.frame(dcholest=dcholest,cholest=cholest,dsys.bp=dsys.bp,sys.bp=sys.bp)
}
by(d, d$id, g)

d$id:a
  dcholest cholest  dsys.bp sys.bp
1 05/17/02     205 04/04/01    136
```

```
------------------------------------------------------------
d$id:b
  dcholest cholest  dsys.bp sys.bp
1 08/03/02     251 08/13/02    151
```

4.3.8 Converting Between Matrices and Vectors: Re–shaping Serial Data

Frequently there is a need to convert between matrices and vectors for reformatting how serial measurements are stored. Suppose that a matrix `x` has one row per subject and one column for each time of data collection, with subject IDs and time points documented in `x`'s `dimnames` attribute. To string out this matrix while creating new vectors containing the IDs and times, you can use the following commands.

```
> y     ← as.vector(x)   # strung-out vector
> ids   ← dimnames(x)[[1]][row(x)]
> times ← as.numeric(dimnames(x)[[2]][col(x)])
```

This process is automated with the Hmisc `reShape` function:

```
> w ← reShape(x)
```

This creates a list `w` with elements `rowvar`, `colvar`, and `x`. `rowvar` contains the row names of the input matrix (converted to numeric if they are all numeric) corresponding to the current row being represented (variable `ids` above). `colvar` contains the corresponding column designations (variable `times` above), converted to numeric form if possible.

To construct a matrix from an irregular vector of measurements where the subject IDs and time points are defined by `ids` and `times` vectors, the following will work.

```
> y      ← 1:12
> ids    ← c('a','b','a','a','b','b','c','c','c','c','d','d')
> times  ← c(  1,  1,  3,  5,  4,  5,  1,  3,  4,  5,  3,  5)
> idf    ← as.factor(ids)
> timesf ← as.factor(times)
> x      ← matrix(NA,nrow=length(levels(idf)),
+                  ncol=length(levels(timesf)),
+                  dimnames=list(levels(idf),levels(timesf)))
> x[cbind(idf,timesf)] ← y
> x
   1  3  4  5
a  1  3 NA  4
b  2 NA  5  6
c  7  8  9 10
d NA 11 NA 12
```

This is done automatically with the `reShape` function again. Here `reShape` reverses course to reconstruct a matrix because the first argument is now a vector, and the `id` and `colvar` arguments are given.

```
> x ← reShape(y, id=ids, colvar=times)
```

To create multiple matrices (e.g., one for systolic blood pressure and one for diastolic) and store the re–shaped results in a new data frame, where each matrix column becomes a separate variable, one could do the following:

```
> Sysbp ← Diasbp ←
+   matrix(NA,nrow=length(levels(idf)),ncol=length(levels(timesf)),
+          dimnames=list(levels(idf),levels(timesf)))
> dimnames(Sysbp)[[2]]  ← paste('sbp',dimnames(Sysbp)[[2]],sep='.')
> dimnames(Diasbp)[[2]] ← paste('dbp',dimnames(Diasbp)[[2]],sep='.')
> age ← c(15, 13, 9, 18)
> # age comes from a non-time-oriented dataframe
> Sysbp[cbind(idf,timesf)]  ← sysbp  # sysbp is strung-out vector
> Diasbp[cbind(idf,timesf)] ← diasbp
> newdata ← data.frame(age, Sysbp, Diasbp,
+                      row.names=levels(idf))
```

The reShape function will create a list containing the multiple matrices:

```
> sys.dias.bp ← reShape(sysbp, diasbp, id=idf, colvar=timesf)
> newdata ← data.frame(age, Sysbp=sys.dias.bp[[1]],
+                      Diasbp=sys.dias.bp[[2]])
```

Here is a similar example using data the base and follow data frames created above. In this example we merge the re–structured variables with baseline data, forming a new data frame.

```
> idf ← as.factor(follow$id)
> monthf ← as.factor(follow$month)
> serial.chol ← matrix(NA, nrow=length(levels(idf)),
+                          ncol=length(levels(monthf)),
+                      dimnames=list(levels(idf),
+                       paste('chol',levels(monthf),sep='.')))
> serial.chol[cbind(idf,monthf)] ← follow$cholesterol
> serial.chol
  chol.1 chol.2 chol.3
a    225    226     NA
b    320    319    318
d     NA    270     NA

> # Or serial.chol ← reShape(follow$cholesterol,
> #                          id=follow$id, colvar=follow$month)
> follow.t ← data.frame(serial.chol, id=levels(idf))
> combined ← merge(base, follow.t, by='id', all.x=T)
> combined
  id age chol.1 chol.2 chol.3
1  a  10    225    226     NA
2  b  20    320    319    318
3  c  30     NA     NA     NA
```

reShape is also handy for converting predictions for regression models into a table. The expand.grid is frequently used to get predicted values for systematically varying predictors. In the following example there are 3 predictors, of which we allow 2 to vary for getting predicted values. We use `reShape` to convert the predictions into a matrix, with rows corresponding to the predictor having the most values, and columns corresponding to the other predictor.

```
> d ← expand.grid(x2=0:1, x1=1:100, x3=median(x3))
> pred ← predict(fit, d)
> reShape(pred, id=d$x1, colvar=d$x2)  # makes 100 x 2 matrix
```

`reShape` has a different action when arguments `base` and `reps` are specified. It will then reshape a variety of repeated and non-repeated measurements. Serial measurements must have the integers $1, 2, \ldots$ `reps` at the end of their names. Non-repeated (e.g., baseline) variables are duplicated `reps` times, and repeated variables are transposed, as shown in the following example.

```
> set.seed(33)
> n ← 4
> w ← data.frame(age=rnorm(n, 40, 10),
+                sex=sample(c('female','male'), n, T),
+                sbp1=rnorm(n, 120, 15),
+                sbp2=rnorm(n, 120, 15),
+                sbp3=rnorm(n, 120, 15),
+                dbp1=rnorm(n,  80, 15),
+                dbp2=rnorm(n,  80, 15),
+                dbp3=rnorm(n,  80, 15), row.names=letters[1:n])
> options(digits=3)
> w

   age    sex sbp1 sbp2  sbp3 dbp1 dbp2 dbp3
a 35.8 female  126  138  90.2 73.6 60.8 64.4
b 42.5 female  121  133 127.8 86.9 73.8 71.1
c 43.2   male  106  117 138.9 68.6 68.9 83.3
d 50.2 female  127  128 126.8 72.1 66.1 69.7

> u ← reShape(w, base=c('sbp','dbp'), reps=3)
> u

  seqno  age    sex   sbp  dbp
a     1 35.8 female 125.8 73.6
a     2 35.8 female 138.3 60.8
a     3 35.8 female  90.2 64.4
b     1 42.5 female 121.4 86.9
b     2 42.5 female 133.0 73.8
b     3 42.5 female 127.8 71.1
c     1 43.2   male 106.1 68.6
c     2 43.2   male 117.4 68.9
c     3 43.2   male 138.9 83.3
d     1 50.2 female 126.9 72.1
d     2 50.2 female 128.3 66.1
d     3 50.2 female 126.8 69.7
```

If is sometimes the case that multiple variables are represented in "long" form with the name of the variable being stored in a column, and the value of any of the variables stored as a numeric variable `value`. If in addition to this, a variable is measured on multiple dates within subjects, the situation is a bit more complicated. In the following example, different laboratory measurements are denoted by the values of a character variable `lab`, and the value of the variable noted in `lab` is contained in the numeric variable `value`. The `id` and `date` variables can be concatenated together to provide a single unique record identifier, then reshaping can be done on the `lab,value` pairs.

```
> id ← c('a','a','a','b','b','b')
> dt ← c(rep('03/12/1992',3),rep('04/17/1993',2),'05/21/1993')
> date ← if(.R.) strptime(dt, format='%m/%d/%Y') else chron(dt)
> # .R. is defined by Hmisc; TRUE if running R
> lab ← c('CBC','HA1C','ALT','CBC','HA1C','HA1C')
> value ← 1:6
> data.frame(id, date, lab, value)   # show all data (R output follows)

  id       date  lab value
1  a 1992-03-12  CBC     1
2  a 1992-03-12 HA1C     2
3  a 1992-03-12  ALT     3
4  b 1993-04-17  CBC     4
5  b 1993-04-17 HA1C     5
6  b 1993-05-21 HA1C     6

> w ← paste(id,date,sep=';')
> d ← reShape(value, id=w, colvar=lab)
> if(!.R.) d ← as.data.frame(d)
> z ← if(.R.) unPaste(row.names(d),';') else unpaste(row.names(d),';')

> d ← data.frame(d, id=z[[1]],
+                  date=if(.R.) strptime(z[[2]], format='%Y-%m-%d') else
+                       as.chron(as.numeric(z[[2]])))
> d

              ALT CBC HA1C id       date
a;1992-03-12    3   1    2  a 1992-03-12
b;1993-04-17   NA   4    5  b 1993-04-17
b;1993-05-21   NA  NA    6  b 1993-05-21
```

If using S-Plus 6 and the `date` variable is a "dates" variable, use `dates(z[[2]])` above.

4.3.9 Computing Changes in Serial Observations

One often wants to compute the change in certain variables from one observation to the next. When the observations are aligned into discrete time slots such as month or follow–up visit, it is easiest to reshape serial data into columns and compute differences between columns. In general though, data may not be collected in discrete time slots, and we may want to compute differences between successive observations no matter what time lapses exist. When one of the variables we want to "difference" is the date of the measurements, we can compute time lapses (differences in dates) to compare against the differences in the measurements.

A general approach involves sorting a data frame by subject id and then date within id, and subtracting from each variable of interest the same variable shifted earlier one observation. This will cause the first observation for each subject to be compared with the last observation for the previous subject, but we will have to delete the first observation from each subject anyway, as there is no baseline to subtract from that observation. In the following example we use serial data for three subjects.

```
> d ← data.frame(id=c('a','a','a','b','c','c'),
+                visit.date=chron(c('02/03/1997','01/17/1997',
+                                   '03/01/1997','05/01/1998',
+                                   '06/01/1998','05/03/1998')),
+                height=c(45.2,45,45.8,  52,  56.1, 56),
+                hormone=c(1.3,1.3,1.8,  2.1, 1.9, 1.8))
> # Sort data frame by id, then date
> i ← order(d$id, d$visit.date)
> i
[1] 2 1 3 4 6 5
> d ← d[i,]
> d

  id visit.date height hormone
2  a   01/17/97   45.0     1.3
1  a   02/03/97   45.2     1.3
3  a   03/01/97   45.8     1.8
4  b   05/01/98   52.0     2.1
6  c   05/03/98   56.0     1.8
5  c   06/01/98   56.1     1.9
```

Now we subtract from the current date and height the values from the previous observation. We can shift vectors one observation earlier by putting an `NA` in front of the vector and ignoring the last element of the vector. This can be done with the following function, which also does preserves attributes of the input variable. This function is one of the undocumented functions in Hmisc.

```
Lag ← function(x, shift=1) {
  if(is.factor(x)) {
    isf ← T
    atr ← attributes(x)
    atr$class ← if(length(atr$class)==1) NULL else
      atr$class[atr$class!='factor']
    atr$levels ← NULL
    x ← as.character(x)
  } else isf ← F
  n ← length(x)
  x ← x[1:(n-shift)]
  if(!isf) atr ← attributes(x)
  if(length(atr$label)) atr$label ←
    paste(atr$label,'lagged',shift,'observations')
  x ← c(rep(if(is.character(x))'' else NA,shift), unclass(x))
  attributes(x) ← atr
  x
}
```

In what follows the `hormone` level we want to associate with each interval is the value at the start of the interval.

```
> # Put data frame in search position 1 to make permanent changes
> attach(d, pos=1, use.names=F)
> time.lapse ← visit.date  - Lag(visit.date)
> height.change ← height - Lag(height)
> hormone.at.interval.start ← Lag(hormone)
> visit.date ← height ← hormone ← NULL  # Remove old variables
> detach(1, 'd2')
> d2

  id time.lapse height.change hormone.at.interval.start
2  a         NA            NA                        NA
1  a         17           0.2                       1.3
3  a         26           0.6                       1.3
4  b        426           6.2                       1.8
6  c          2           4.0                       2.1
5  c         29           0.1                       1.8
```

Now to delete the first record for each subject we must flag these records.

```
> first.id ← d2$id != Lag(d2$id)
> first.id
[1] T F F T T F

> d2 ← d2[!first.id,]
> d2

  id time.lapse height.change hormone.at.interval.start
1  a         17           0.2                       1.3
3  a         26           0.6                       1.3
5  c         29           0.1                       1.8
```

4.4 Recoding Variables and Creating Derived Variables

As discussed in Section 9.4, there are many types of derived variables for which the logical place to state the derivation formula is in a regression model formula. For example, creating dummy variables and storing `age^2` as a separate variable means that you are not using the real power of the S modeling language. Still, there are plenty of occasions for creating or recoding variables. Here is a series of examples showing common ways of creating new variables. See the help file for `merge.levels` for details about how to change the levels of a `factor` variable.

```
# compute min(wbc,100000) for each patient
wbc.curtailed ← pmin(wbc, 100000)
# Still may be better to do this inside a model formula

# Compute a function of height and weight that is different
# for 2 sexes
size ← ifelse(sex=='female',.2*weight^.66/height^.33,
        .25*weight^.6/height^.3)
```

```
# Six ways to combine treatments B and C into one group
# First four assume that treat is a factor object
levels(treat)[levels(treat) %in% c('B','C')] ← 'BC'
levels(treat) ← c('A','BC','BC')
levels(treat) ← list(c('B','C'))
# list method causes merge.levels to combine B and C into 'B, C'
levels(treat) ← list(BC=c('B','C'))  # name it BC instead of B, C
# To make multiple merges, do e.g. list(c('B','C'),c('D','E'))
treat2 ← ifelse(treat=='A',treat,'BC')
treat2 ← ifelse(treat %in% c('B','C'), 'BC', treat)

# Group several levels of a categorical variable.
# Leave old variable alone.
# Group levels a b c d into group A, e f g into B, h i into C
y2 ← y
levels(y2) ← Cs(A,A,A,A,B,B,B,C,C)  # or:
levels(y2) ← list(A=Cs(a,b,c,d), B=Cs(e,f,g), C=Cs(h,i))  # or:
levels(y2) ← list(Cs(a,b,c,d), Cs(e,f,g), Cs(h,i))  # auto naming

# Categorize a continuous variable (why?)
agecat ← (age>=30)+(age>=40)+(age>=50)+(age>=60)
# age=41 yields agecat=2.  Missing age yields missing agecat
# Could also use agecat ← cut2(age,c(30,40,50,60))

# Create a 3-category variable coded none, either of two
# conditions is true, or both are true
# First assume that both x1 and x2 are logical or 0-1 variables

z ← x1 + x2

# Instead, create temporary logical variables from expressions
z ← (x1=='present')+(x2=='present')
z ← (x1 > 30) + (x2 > 1000)
# Results in x1 <= 30 & x2 <= 1000 : 0
#            x1 > 30 or x2 > 1000  : 1
#            x1 > 30 &  x2 > 1000  : 2
# Could create a self-documenting variable by:
z2 ← c('neither','either','both')[z+1]

# Create a 3-category variable in which the presence of a true for
# the second variable overrides the value of the first variable

z ← (x1=='present' & x2=='absent') + 2*(x2=='present')
# Results in x2=='present'& x1=='present' : 2
#            x2=='absent' & x1=='present' : 1
#            x2=='absent' & x1=='absent'  : 0
z ← ifelse(x2=='present', 'x2 present',
           ifelse(x1=='present', 'x1 but not x2', 'neither'))
# Results in x2=='present'                : 'x2 present'
#           x2=='absent' & x1=='present' : 'x1 but not x2'
#           x2=='absent' & x1=='absent'  : 'neither'

# Create a new categorical variable on the basis of sex and
# whether age>=50.  First two ways will produce the same
# coding, all 3 ways produce a good result
g ← ifelse(sex=='male',
           ifelse(age>=50,'M >= 50','M < 50'),
           ifelse(age>=50,'F >= 50','F < 50'))
g ← paste(ifelse(sex=='male','M','F'),
          ifelse(age>=50,'>= 50','<50'))
g ← interaction(sex, age>=50)
```

Recodes to character values can sometimes be done easily by first recoding into integers and then looking up correspondences between the integers and the intended character strings, as shown below.

```
> x ← c('cat','dog','giraffe')
> x ← c('domestic','wild')[1*(x %in% c('cat','dog')) +
+      2*(x=='giraffe')]
> x
[1] "domestic" "domestic" "wild"
```

The second line above applies a sequences of ones and twos as subscripts of the 2 element vector `c('domestic','wild')`. The result is a vector of character strings of the same length as the vector `x`, as duplicate ones and twos will result in multiple uses of the character constants. Manipulating `levels` of a factor variable is easier, implicitly using the `merge.levels` built-in function.

```
> x ← factor(x)
> levels(x) ← list(domestic=c('cat','dog'),wild='giraffe')
```

Recodes from single character string values to numeric or other character values can also be accomplished using a named vector and the subscript operator:

```
> newcodes ← c(cat='feline',dog='canine','guinea pig'='gpig')
> animals ← c('cat','cat','guinea pig','dog')
> animals ← newcodes[animals]
> animals
      cat      cat guinea pig      dog
 "feline" "feline" "gpig"     "canine"
```

In the final two lines of output, the `animals` vector is seen to have a `names` attribute showing he original values. Note that the name `'guinea pig'` had to be enclosed in quotes since it is not a legal S name. We could have done many–to–one recodes by having multiple `names` for the same character looked–up value. But again, manipulating `factor levels` is more elegant:

```
> animals ← factor(c('cat','cat','guinea pig','dog'))
> levels(animals) ← list(feline='cat',canine='dog',
+                        gpig='guinea pig')
```

Often it is the case that a large number of categories needs to be recoded into broad groupings. For example, one might wish to categorize medical diagnoses into organ systems or other groupings.

It is much easier to do this by constructing a data frame of all individual categories, creating a new character variable to contain the broad category names (initialized to blanks), and to edit the latter (using a data sheet in Windows S-Plus). In the following example we sort categories by descending frequency of occurrence, initialize categories not used in at least 3 observations to `'other'`, create a data frame suitable for editing, and then show how to do a table look-up from this new data frame containing category definitions to use them in our main data frame.

```
> tab ← table(diagnosis)
> tab ← tab[order(-tab)]
> DX ← data.frame(diagnosis=names(tab))
> DX$dxgroup ← ifelse(tab < 3, 'other', '')
# This adds dxgroup to the DX data frame without
# converting it to factor; we can edit levels arbitrarily
# Edit DX data frame, then merge new dxgroup definitions
> dxgroup.def ← DX$dxgroup
> names(dxgroup.def) ← as.character(DX$diagnosis)
# Be sure to store dxgroup.def permanently as separate object
> dxgroup ← dxgroup.def[as.character(diagnosis)]   # fast lookup
# Enclose right hand side in factor() if desired,
# to save storage space
```

The final variable `dxgroup` is the same length as `diagnosis`.

Many of the functions in Hmisc and Design use the `'label'` attributes of variables. If you are creating printed or graphical output using one of those functions, be sure to define labels to variables you create using Hmisc's `label` function. You may also want to use Hmisc's `units` function to define units of measurement. At present, this is only used for survival time objects and for the `describe` function.

```
map ← (2*diastolic+systolic)/3
label(map) ← 'Mean Arterial Blood Pressure'
units(map) ← 'mm Hg'
```

4.4.1 The `score.binary` Function

When you wish to create a new categorical, ordinal, or numeric variable from a series of binary or logical values, the `score.binary` function in Hmisc may be useful. `score.binary` summarizes the binary conditions using a hierarchical rule in which the last true value of the series applies, an additive sum is computed, or other user–specified summaries are computed. By default, `score.binary` uses the first of these three methods is used, i.e., logical true/false variables are examined from left to right, and each observation is classified into the category corresponding to the rightmost true value. The `x1,x2` recode example above did this using builtin S language features. Here are the examples from `score.binary`'s help file.

```
# Hierarchical scale, highest of 1:age>70  2:previous.disease
# Here score.binary will return a numeric variable with values 0,1,2
x ← score.binary(age>70, previous.disease, retfactor=F)
# Same as above but return factor variable with levels "none" "age>70"
# "previous.disease":
x ← score.binary(age>70, previous.disease)
```

```
# Additive scale with weights 1:age>70  2:previous.disease
x ← score.binary(age>70, previous.disease, fun=sum)
# Additive scale, equal weights
x ← score.binary(age>70, previous.disease, fun=sum, points=c(1,1))
# Same as saying points=1

# Union of variables, to create a new binary variable
x ← score.binary(age>70, previous.disease, fun=any)
```

4.4.2 The `recode` Function

An undocumented function in Hmisc, `recode`, may save some time in recoding variable values. `recode` can handle numeric quantities. When dealing with character or factor vectors it is better to manipulate `levels` as shown in Section 4.4. Here are some `recode` examples.

```
> x ← c('cat','dog','rat')
> recode(Catdog=x=='cat'|x=='dog')
[1] Catdog Catdog none
> recode(Catdog=x=='cat'|x=='dog',Rat=x=='rat')
[1] Catdog Catdog Rat
> recode(Catdog=x %in% c('cat','dog'), rat=x=='rat')
[1] Catdog Catdog rat
> # Also use x ← factor(x); levels(x) ← list(Catdog=c('cat','dog'),...)
> x ← 1:3
> recode('22'=x==1 | x==3,'2'=x==2)
[1] 22  2 22
```

Note that `recode` returned a numeric variable in the last example even though the argument names given to `recode` were `'22'` and `'2'`. `recode` checked to see that all of the target values were numeric and that being the case it transformed the result to numeric. Target codes were specified on the left hand side of the equal sign, and when the target codes are legal S names they need not be enclosed in quotes.

`recode` can also be called a different way as shown below.

```
> x ← c(1,2,3,3)
> recode(x,1:3,3:1)
[1] 3 2 1 1
> recode(x,1:3,c('a','b','c'))
[1] "a" "b" "c" "c"
> recode(x,1:3,c('cat','dog','rat'))
[1] "cat" "dog" "rat" "rat"
```

`recode` has some optional arguments. One of them is `none`, which can be used to set the value to return if the original value is not matched by one of the values to recode from.

The `match` function is also handy for recoding.

```
> x_c('a','b','c','c')
> match(x,c('a','b','c'))
[1] 1 2 3 3
```

```
> c('a','b','c')[c(1,2,3,3)]
[1] "a" "b" "c" "c"
```

In the following example, we use a small function `rec` to recode a vector.

```
> rec ← function(x, from, to) {
+   i ← match(x, from)
+   to[i]
+ }
> x
> rec(x,c('a','b','c'),c('A','B','C'))
[1] "A" "B" "C" "C"
> rec(x,c('a','b'),c('A','B'))
[1] "A" "B" ""  ""
> rec(x,c('a','b','c'),c('ab','ab','c'))
[1] "ab" "ab" "c"  "c"
```

4.4.3 Should Derived Variables be Stored Permanently?

If upon leaving S-Plus you want to be able to re–start S-Plus and pick up right where you left off, it may be best to store derived variables permanently as separate vectors in `_Data` or in the input data frame:

```
store()
x.derived ← some formula of x
store(x.derived)
# Or:
x.derived ← some formula of x
my.data.frame$x.derived ← x.derived
# works when store() not in effect
# Or: attach(dframe,pos=1,use.names=F)
#     x.derived ← ...;detach(1,'dframe')
```

However, derived variables do take up disk space and they will not automatically be re–derived should you correct one of the original variables used to compute the derived ones. Neither will they be re–derived if you change the derivation formulas. It is thus often better to copy and paste the derivation formulas into the command window from an editor window or to otherwise save the derivation formula for later use. A fancy approach would be to store the derivation formulas as an attribute to the input data frame as shown in the following example.

```
derived ← expression(x2 ← x^2; y2 ← y^2)
eval(derived)                         # evaluate derived variables now
attr(my.data.frame,'derived') ← derived
eval(attr(my.data.frame,'derived'))
# useful for re-evaluating them later

derive ← function(obj) {      # define a function to do this
  eval(attr(obj, "derived"), local = sys.parent(1))
  invisible()
}

derive(my.data.frame)                 # same as eval(attr(my...))
```

4.5 Review of Data Frame Creation, Annotation, and Analysis

It is often useful to create, modify, and process datasets in the following order:

1. import external data
2. make global changes to a data frame (e.g., changing variable names)
3. change attributes or values of variables within a data frame
4. do analyses involving the whole data frame (without attaching it)
5. do analyses of individual variables (after attaching the data frame)

The following program is an example. Here we are processing Rosner's `FEV` data. First, we do steps that create or manipulate the data frame in its entirety. These are done with `_Data` in search position one (the S-PLUS default at the start of the session). The `cleanup.import` function changes numeric variables that are always whole numbers to be stored as integers, the remaining numerics to single precision, strange values from Excel to `NA`s, and character variables that always contain legal numeric values to numeric variables. `cleanup.import` typically halves the size of the data frame.

```
# The data were imported into data frame FEV to distinguish
# this name from the variable fev, using File ... Import
# Source data: Rosner fev.asc (documented in fev.txt)

FEV ← cleanup.import(FEV)
names(FEV)[6] ← 'smoke'
# or names(FEV)[names(FEV)=='smoking'] ← 'smoke'
# or names(FEV) ← edit(names(FEV)) or edit in Object Explorer
```

The renaming of `smoking` to `smoke` can also be done using `upData`. Next we make changes to individual variables within the data frame. When changing more than one or two variables it is most convenient to use `upData` so that we can omit the data frame and $ prefix before all the variable names being changed.

```
FEV2 ← upData(FEV,
              rename=c(smoking='smoke'), # omit if renamed above
              levels=list(sex  =list(female=0,male=1),
                          smoke=list('non-current smoker'=0,
                                     'current smoker'=1)),
              units=list(age='years', fev='L', height='inches'),
              labels=list(fev='Forced Expiratory Volume'))

# Check the data frame
page(describe(FEV2), multi=T)
# page makes results go to a new window
# multi=T allows that window to persist while control
# is returned to other windows

# The new data frame is OK.  Store it on top of the old FEV and
# then use the graphical user interface to delete FEV2 (click on it
# and hit the Delete key, or type rm(FEV))
FEV ← FEV2
```

Next, analyses are done that refer to all or almost all variables in the data frame. This is best done without attaching the data frame.

```
summary(FEV)             # basic summary function
plot(FEV)
datadensity(FEV)
hist.data.frame(FEV)
by(FEV, FEV$smoke, summary)  # use basic summary with stratification
```

Now, to do detailed analyses involving individual variables, attach the data frame in search position 2.

```
attach(FEV)
options(width=80)
summary(height ∼ age + sex,
        fun=function(y)c(smean.sd(y),
                         smedian.hilow(y,conf.int=.5)))
# This computes mean height, S.D., median, outer quartiles

# Run generic summary function on height and fev, stratified by sex
by(data.frame(height,fev), sex, summary)

# Cross-classify into 4 sex x smoke groups
by(FEV, list(sex,smoke), summary)

# Plot 5 quantiles
s ← summary(fev ∼ age + sex + height,
             fun=function(y)quantile(y,c(.1,.25,.5,.75,.9)))
s
plot(s, which=1:5, pch=c(1,2,15,2,1), # pch=c('=','[','o',']','='),
     main='A Discovery', xlab='FEV')

# Use the nonparametric bootstrap to compute a 0.95 confidence
# interval for the population mean fev
smean.cl.boot(fev)

# Use the Statistics ... Compare Samples ... One Sample keys to get
# a normal-theory-based C.I.  Then do it more manually.  The following
# method assumes that there are no NAs in fev

sd ← sqrt(var(fev))
xbar ← mean(fev)
xbar
sd
n ← length(fev)
qt(.975,n-1)     # prints 0.975 critical value of t dist. with n-1 d.f.
```

```
xbar + c(-1,1)*sd/sqrt(n)*qt(.975,n-1)   # prints confidence limits

# Fit a linear model
fit ← lm(fev ~ other variables ...)
```

See Section 3.4 for more details about creating and modifying data frames.

4.6 Missing Value Imputation using Hmisc

When developing multivariable regression models, the default action of many S functions (and every other system) is to delete an entire row of data when *any* of the variables are missing. In many cases it is a shame to exclude observations missing on X_1 while studying the relationship between X_2 and Y, as this loss of data reduces power and increases variances. Also, deletion of observations containing missing data causes a bias when the data are not missing at random. It is usually better to estimate missing values than to discard valuable data.

When a predictor variable is uncorrelated with all of the other predictors, one can obtain nearly unbiased estimates of regression coefficients (at least in ordinary multiple regression) by replacing missing values with constants. The `impute` function in Hmisc makes this easy to do. By default, `impute` will replace missing values with the median non–missing value for continuous variables, and with the most frequent category for categorical (`factor`) or logical variables. One can specify other statistical functions for use with continuous variables instead of the default `fun=median` (e.g., `fun=mean`), or constants or randomly sampled values to insert for numeric or categorical variables. There are methods for printing, subsetting, summarizing, and describing variables having imputed values. There is also a function, `is.imputed`, that allows easy detection of imputed values. Here are some examples from the `impute` help file:

```
> age ← c(1,2,NA,4)
> age.i ← impute(age)
# Could have used impute(age,2.5), impute(age,mean), impute(age,"random")
> age.i    # Note that print.impute places * after imputed values
  1 2  3 4
  1 2 2* 4

> summary(age.i)
 1 values imputed to 2

 Min. 1st Qu. Median Mean 3rd Qu. Max.
    1    1.75      2 2.25     2.5    4
> describe(age.i)
 age.i

 n missing imputed unique Mean
 4 0       1       3      2.25

1 (1, 25%), 2 (2, 50%), 4 (1, 25%)

> is.imputed(age.i)
[1] F F T F
```

If one developed a model after imputing `NA`s, it's easy to re–fit the model to see if imputation caused any of the estimated regression shapes to change:

```
> f.noimpute ← update(f, subset=!is.imputed(age))
```

When variables containing `NA`s are correlated with other variables, it is more accurate to impute these values by predicting them from the other variables. If relationships between variables are monotonic, a tree model may be a convenient approach. In general, customized regression equations may be needed. Hmisc's `aregImpute` function finds transformations that optimize how each variable is predicted from each other variable using additive semiparametric models (using `ace` or `avas` functions). In some cases, one variable can be predicted from another only after a non–monotonic transformation is made on each one. For example, heart rate does not correlate well with blood pressure, but the absolute difference between heart rate and a normal value for heart rate does correlate with the absolute difference between blood pressure and a normal value for blood pressure. `aregImpute` can find such transformations and base imputations on them. It does imputations, even allowing for missing values in the variables currently being used to predict `NA`s in the specific variable.

Once `aregImpute` develops all of the customized imputation models automatically, a special form of the `impute` function (`impute.transcan`) can apply the imputations:

```
> xt ← aregImpute(~ age + blood.pressure + hrate + race)
> blood.pressure ← impute(xt, blood.pressure, imputation=1)
> hrate           ← impute(xt, hrate, imputation=1)
> impute(xt)     # causes all variables to be imputed, storing
>                # imputed variables under their original names
```

But note that the use of `fit.mult.impute` (see below) is a better approach. Continuous and categorical variables are imputed by `aregImpute` using predictive mean matching.

It is well known that ignoring the fact that imputations were done will bias standard error estimates downward. Estimated standard errors can be corrected using multiple imputation, but it is also easy to use the bootstrap. The bootstrap can also adjust for other sources of variation such as stepwise variable selection or estimating transformations of the response variabel. Bootstrapping is not usually practical when using `aregImpute` because `aregImpute` often runs too slowly to be called inside a bootstrap loop. Here is an example when imputations are done using a constant. See Section 4.7 for another bootstrap example.

```
> store()   # don't keep any objects from this session
> # Generate data with no missing values
> n ← 200
> set.seed(231)
> x1 ← rnorm(n)
> x2 ← sample(0:1, n, replace=T)
> y  ← x1 + 2*x2 + rnorm(n)/3

# Make 40 of the x1 values missing at random
> x1[sample(1:length(x1), 40)] ← NA
> describe(x1)
x1
   n missing unique    Mean      .05      .10      .25      .50
```

```
 160 40      160    0.02925 -1.82198 -1.40211 -0.62995  0.05152
      .75      .90      .95
  0.83800  1.27483  1.65581

> # Impute missing x1s using the median of non-missing x1s
> x1i ← impute(x1)
> # Fit linear regression model using Design library's ols function

> f ← ols(y ∼ x1i + x2, x=T, y=T)
> # Print standard errors that were computed using the standard formula
> sqrt(diag(f$var))
[1] 0.05802961 0.04385920 0.08295235

> # Compute bootstrap estimates of standard errors not corrected for imputation
> B ← 300
> sqrt(diag(bootcov(f, B=300, pr=T)$var))
[1] 0.05441155 0.02582960 0.08077475

> # Note that these standard errors are unconditional estimates whereas
> # Standard formulas use variables conditional on covariable values

> # Now correct for imputation.  The following calculations are the same
> # used by bootcov except that imputation is done inside the bootstrap loop
> betas ← matrix(NA, nrow=B, ncol=3)
> for(i in 1:B) {
+   cat(i,'')
+   j ← sample(1:n, n, rep=T)  # bootstrap sample
+   x1b  ← x1[j]
+   x1bi ← impute(x1b)
+   x2b  ← x2[j]
+   yb   ← y[j]
+   cof    ← lm.fit.qr(cbind(1,x1bi,x2b), yb)$coefficients
+   # lm.fit.qr is used internally by ols and lm.  Use it here for
+   # raw speed.  Even faster- use undocumented Hmisc function:
+   # cof ← lm.fit.qr.bare(cbind(x1bi,x2b), yb)$coefficients
+   betas[i,] ← cof
+ }

> sqrt(diag(var(betas)))
[1] 0.06436752 0.02908877 0.08260003
```

We see that when correcting for imputation the standard error of the intercept and of the regression coefficient for `x1` increased the most.

Caution: In most situations, especially ordinary multiple regression, imputing "best guess" expected values for missing values results in biases. Deriving imputation models ignoring the response variable will bias final regression coefficients downward in absolute value. So it is usually better to develop imputations using the response variable to predict the independent variables, and to impute using randow values (random draws) for the predictors, i.e., to add random residuals into imputed values. You can easily obtain random draws using `impute(x, 'random')`, but these do not allow for relationships among predictors or between `x` and the response.

The `aregImpute` function generates multiple imputations without making distributional assumptions. After running `aregImpute` you can run the Hmisc `fit.mult.impute` function to fit the chosen model separately for each artifically completed dataset corresponding to each imputation. After `fit.mult.impute` fits all of the models, it averages the sets of regression coefficients and computes variance and covariance estimates that are adjusted for imputation using a standard formula. Here is an example:

```
> # Optimally transform all 4 variables and make 10 sets of random
> # imputations on the distribution of each variable conditional on
> # all the others
> xtrans ← aregImpute(∼ y + x1 + x2 + x3, n.impute=10)
> # Fit 10 models for 10 completed datasets
> f ← fit.mult.impute(y ∼ x1*x2 + x3, lm, xtrans)
> Varcov(f)  # prints imputation-corrected covariance matrix
```

Here `fit.mult.impute` fitted the 10 models using the built–in `lm` multiple regression function. A drawback to using `lm` here is that when you do `summary(f)` to get coefficients, standard errors, P–values, etc., only the coefficients are correct. Using the Design `ols` function in place of `lm` gets around this problem, and also allows for flexible ways to relax linearity assumptions. Here is an example:

```
> f ← fit.mult.impute(y ∼ rcs(x1)*x2 + x3, ols, xtrans)
> f   # prints corrected coefficients, standard errors, Z-statistics, P
```

4.7 Using S for Simulations and Bootstrapping

The S language and the ease of referencing the result of statistical functions makes S an ideal language for doing traditional Monte Carlo simulation as well as bootstrapping. When the dataset forming the basis of the simulation is large or when the number of iterations is very large, S will run slower than other systems. However the savings in programming time usually more than makes up for the slower execution time.

For a first example let us use Monte Carlo simulation to estimate the population variance of the sample median for a sample of size $n = 50$ from a log–normal distribution. In the following code note the importance of setting aside `reps` locations in which to store the medians. If instead you set `meds` to `NULL` and continually concatenated sample medians to `meds` (e.g., `meds ← c(meds, median(x))`, the memory usage of the program would be very inefficient.

```
> store()
> n ← 50
> reps ← 400
> meds ← single(reps)  # set aside 400 of them
> set.seed(171)        # allows us to reproduce results
> for(i in 1:reps) {
+   x ← exp(rnorm(n))
+   meds[i] ← median(x)
+ }
> var(meds)
[1] 0.02887161
```

This took 1.8 seconds on a Pentium 166. Another approach would be to generate all the data up front, and to apply a matrix operation to compute the needed statistics:

```
> set.seed(171)
> x ← matrix(exp(rnorm(n*reps)), nrow=reps, ncol=n, byrow=T)
> # byrow=T forces x to be built in same order as first example,
> # so we get identical results
> meds ← apply(x, 1, median)
> var(meds)
[1] 0.02887161
```

This also took 1.8 seconds.

Now consider how to compute a simple bootstrap estimate (see also Section 4.6). Suppose the data consists of the heights in feet of a sample of 20 adults and we want to derive a 90% confidence interval for the population median height without making distributional assumptions. We take 500 samples with replacement, each of size 20, from the 20 heights, and compute the sample median. We then get the sample 0.05 and 0.95 quantiles of these 500 medians to form the desired confidence interval. The built–in `sample` function makes bootstrapping easy.

```
> h ← c(5.5, 5.7, 5.2, 5.0, 6.2, 5.9, 6.4, 6.1, 5.5, 5.8,
+       6.0, 6.4, 5.0, 4.9, 5.7, 5.8, 5.3, 6.2, 6.1, 5.6)
> median(h)
[1] 5.75
> B    ← 500
> meds ← single(B)
> set.seed(113)              # if want to reproduce this later
> for(i in 1:B) {
+   s ← sample(1:20, 20, replace=T)
+   meds[i] ← median(h[s])  # h[s] samples h using subscripts s
+ }
> table(meds)
 5.1 5.25 5.3 5.4 5.45 5.5 5.55 5.6 5.65 5.7 5.7 5.75 5.8 5.8 5.85
   1    1   2   3    1  25   18  31   40   8  88   71  99   3   36
 5.9 5.95 5.95   6 6.05 6.1 6.15
  27    5   17 12    7   4    1
> quantile(meds, c(.05, .95))
  5%  95%
 5.5 5.95
```

This program ran in 3.9 seconds. The program can be shortened considerably because of built–in bootstrap functions:

```
> b ← bootstrap(h, median, B=B)
> limits.emp(b)
```

This ran in 5.0 seconds.

The `bootstrap` function is quite flexible. In the following example we use it to provide confidence limits for a type of estimate for which computing limits would be quite difficult by other means. We compute 0.95 confidence on the ranks of departments in a hospital, where what is being ranked is the mean satisfaction level with the departments, based on responses to a 5–point satisfaction scale. This is a common problem in “scorecarding” or “provider profiling” of departments, hospitals, or other entities. Just ranking the mean satisfaction scores across departments does not take into account the fact that the mean scores are estimates themselves. By using the bootstrap to derive confidence limits for ranks, we will lessen the chance that ranks will be misinterpreted. In what follows we stratify the overall analysis by the sex of the questionnaire’s respondant.

First we do a more traditional analysis where individual mean satisfaction scores and t–based confidence limits are computed by department and by sex. The Hmisc `summarize`, `smean.cl.normal`, and `Dotplot` functions are useful. See Section 11.4.2 for more about `Dotplot`, which nicely displays results for dozens of departments along with “error bars”, and see Section 11.4.3 for more examples of using `summarize` with `trellis` graphics functions. Here we downplay point estimates by using small tick marks (plus sign, `pch=3`) to mark them, and emphasize 0.99 confidence limits drawn with horizontal lines. The S `reorder.factor` function nicely orders the departments by the mean of the male and female satisfaction scores within each department. This makes the graph easier to read.

```
> w ← summarize(Rating, llist(Department, Sex),
+                 smean.cl.normal, conf.int=0.99)
> attach(w,1)

> Department ← reorder.factor(Department, Rating, mean)

> Dotplot(Department ~ Cbind(Rating, Lower, Upper) | Sex,
+         main='Means and 0.99 C.L.', pch=3, xlim=c(1,5),
+         xlab='Rating')
```

Next we analyze the data separately by sex to compute the ranks of the mean scores and 0.95 confidence limits for these ranks. We could use the `bootstrap` function easily by creating a data frame to contain department codes and satisfaction ratings, but it is faster to have `bootstrap` operate on a matrix. As the matrix (here, `d`) needs to be all numeric, we temporarily convert the `Department` factor variable into integer codes. Factor levels are later re–associated with these codes for nice plotting. We use the `group` argument to `bootstrap` so that resampling is done separately within each department, to ensure that when these individual resamples are combined into one overall resample, each department’s sample size equals the original sample size. After the bootstraps are done, results for both bootstraps are combined into a single data frame `D`.

```
> detach(1)    # detach w, go back to raw data
# Define a function to compute stratified means and then rank them
# We can use this function for both overall ranks and ranks within
# bootstrap resamples.  After all bootstrap resamples are done, we can
# use limits.emp to compute sample quantiles of these ranks,
# stratified by department
> rankdept ← function(d) {
+   w ← tapply(d[,2], d[,1], mean, na.rm=T)
+   r ← rank(w)
+   names(r) ← names(w)
+   r
+ }

> D ← NULL
> for(sx in levels(Sex)) {
```

```
+   s ← Sex==sx    # analyze each sex separately
+   d ← cbind(as.integer(Department)[s], Rating[s])
+   ranks ← rankdept(d)
+   ranksb ← bootstrap(d, rankdept(d), B=500, group=Department[s])
+   lim ← limits.emp(ranksb)
+   w ← data.frame(Sex=rep(sx,length(ranks)),
+                  Department=
+                   levels(Department)[as.integer(names(ranks))],
+                  Rank=ranks,
+                  Lower=lim[,1],   # 0.025 quantile of 500 estimates
+                  Upper=lim[,4])   # 0.975 quantile
+   D ← rbind(D, w)
+ }
> # The 'as.integer(names(ranks))' trick uses the fact that we told
> # the rankdept function to retain the department codes as the names
> # attribute of the rank vector.  Names are always stored as
> # character strings so we had to convert them to numeric

> # Arrange levels so that Dotplot will order categories by
> # average over female, male
> D$Department ← reorder.factor(D$Department, D$Rank, mean)

> Dotplot(Department ~ Cbind(Rank, Lower, Upper) | Sex,
+         data=D, pch=3, xlab='Rank',
+         main='Ranks and 0.95 Confidence Limits for Mean Overall Satisfaction')
```

The analyst will usually find confidence limits for ranks to be quite wide, as they should be. Trying to rank small differences can be quite difficult.

Another place where bootstrapping ranks is useful is the situation where an investigator wishes to conclude that one predictor variable is better than another in terms of the Wald partial χ^2 minus the degrees of freedom (to level the playing field) contributed by the variables. The online help for the Design library's `anova` function has the following example, which uses the `plot` method for an `anova.Design` object without actually plotting the χ^2s at each re-sample. We rank the negative of the adjusted χ^2s so that a rank of 1 is assigned to the highest. It is important to tell `plot.anova.Design` not to sort the results, or every bootstrap replication would have ranks of 1,2,3 for the statistics.

```
> b ← bootstrap(mydata,
+               rank(-plot(anova(
+                lrm(y ~ rcs(x1,4)+pol(x2,2)+sex,mydata)), sort='none', pl=F)),
+               B=50)  # should really do B=500 but will take a while
> Rank ← b$observed
> lim ← limits.emp(b)[,c(1,4)]  # get 0.025 and 0.975 quantiles

# Use the Hmisc Dotplot function to display ranks and their confidence
# intervals.  Sort the categories by descending adj. chi-square, for ranks
> original.chisq ← plot(anova(lrm(y ~ rcs(x1,4)+pol(x2,2)+sex,data=mydata)),
+                       sort='none', pl=F)
> predictor ← as.factor(names(original.chisq))
> predictor ← reorder.factor(predictor, -original.chisq)

> Dotplot(predictor ~ Cbind(Rank, lim), pch=3, xlab='Rank',
+        main='Ranks and 0.95 Confidence Limits for Chi-square - d.f.')
```

See the Hmisc `bootkm` function as another example of bootstrapping. For obtaining basic non-parametric confidence intervals for a population mean using the bootstrap percentile method as was used above, use the blazing fast `smean.cl.boot` function.

It is easy to call Fortran or C programs from within S. So if you are doing extensive simulations that run too slowly, you may want to isolate the slow code (particularly when subscripting must be done in a loop) and program it in Fortran or C.

For power simulations it is often very easy to program repeated regression model fitting. It is very important that you call the lowest possible level of fitting routine so that at each iteration S does not need to interpret model formulas, check for missing data, form design matrices, etc. Here is an example where we estimate the power for testing the effects of one predictor on a binary outcome, adjusted for another predictor. The population correlation between the two predictors is 0.75. The population regression coefficients are -1.4, 1, and 0.7, for the intercept and two predictors, respectively. This program simulates power unconditional on `x1` and `x2`. To simulate conditional power, generate these variables once before the loop.

```
store()

n     ← 50
nsim  ← 1000
rho   ← .75
beta1 ← 1
beta2 ← .7
intercept ← -1.4

# Show inter-quartile-range odds ratios in effect
cat('IQR OR for adjustor:',format(exp(beta1*1.34898)),'\n')
cat('IQR OR for predictor:',format(exp(beta2*1.34898)),'\n')

r ← prop ← chisq ← single(nsim)

for(i in 1:nsim) {
  cat(i,'')
  x1 ← rnorm(n)  # unconditional power on x1,x2
  x2 ← rnorm(n) + (rho/sqrt(1 - rho*rho)) * x1
  L ← intercept+beta1*x1+beta2*x2
  y ← ifelse(runif(n)<=plogis(L),1,0)  # or better:
  y ← rbinom(n, size = 1, prob = plogis(L))
  r[i] ← cor(x1,x2)
  prop[i] ← mean(y)
  x ← cbind(x1, x2)
  f ← lrm.fit(x,y)   # lrm.fit in Design, called by lrm
  chisq[i] ← (f$coef[3])^2/f$var[3,3]
}

prn(mean(r),'Mean correlation between x1 and x2')   # prn in Hmisc
prn(mean(prop),'Mean proportion Y=1')
prn(mean(chisq>3.84),'Power')
```

For some problems you may have sample predictor values in a previous study. If the sample size used in the simulation is less than that from the previous study, you can form the `x1,x2` vectors by sampling without replacement from the study's values. Otherwise, you might consider sampling with replacement. Either of these approaches will allow you to use actual rather than hypothetical normal distributions. There is still the problem of what population regression coefficient values to use in the simulations. Here is an example.

```
# Let df be a data frame containing a sample of predictor values
# Sample from these
i  ← sample(1:nrow(df), n)
x1 ← df$x1[i]
x2 ← df$x2[i]
# At this point start simulation loop to get power conditional on
# x1,x2
```

See Section 5.3 for an example where the Hmisc `spower` function is used to simulate the power of a survival time comparison. See Section 7.2.1 for an example where multivariate normal repeated measurement data are simulated.

Chapter 5

Probability and Statistical Functions

5.1 Basic Functions for Statistical Summaries

There are many functions to produce statistical summaries. We already used `describe` and `table`. Table 5.1 gives a concise list of some other basic functions. All the functions below `pmax` are in the Hmisc library.

A few details about these functions: `cor` computes the ordinary Pearson product–moment linear correlation coefficient. `cor`,`mean`,`var`,`median`, `min`, `max`, and `quantile` do not accept `NA`s without extra effort. The `cor.test` function will automatically exclude `NA`s. All but `var` and `cor` have an optional parameter `na.rm` which can be set to `T` to cause `NA`s to be deleted before doing any calculations. For `var` and `cor` you will have to delete the `NA`s from the input variables yourself. `mean` and `median` do not operate separately in columns of matrices. Use a combination of the function and `apply` for this purpose. `min` and `max` have the same limitation as `mean`, but `pmin` and `pmax` can be used to obtain the min or max of several vectors simultaneously.

`rcorr` efficiently computes Pearson and Spearman rank correlation matrices and P–values, doing pairwise deletion of `NA`s. `hoeffd` uses pairwise deletion of `NA`s in computing Hoeffding's general measure of dependence between any two variables.

The functions `bystats` and `bystats2` in Hmisc can be used to obtain statistics on a variable by the levels of several classification variables (i.e., by–processing). These have been superceded to some extent by Hmisc's `summary.formula` and `summarize` functions, but `bystats` can still be useful for stratification by more than two variables. See Section 6.2 for examples of `summary.formula`.

Table 5.1: *Functions for Statistical Summaries*

Function	Description	Comments
`cor`	`cor(x,y)`	correlations between `x` and `y`
`cor.test`	`cor.test(x,y,method)`	Pearson, Spearman, Kendall corr. and tests
`var`	`var(x,y)`	variances and covariances
`cumsum`	`cumsum(x)`	cumulative sums
`mean`	`mean(x)`	mean of a vector
`median`	`median`	median of a vector
`quantile`	`quantile(x,probs=...)`	quantiles
`min`	`min(...)`	overall minimum value of all arguments
`max`	`max(...)`	overall maximum value of all arguments
`pmin`	`pmin(...)`	minimum for each row over several vectors
`pmax`	`pmax(...)`	maximum for each row over several vectors
`describe`	`describe(..)`	describe data frame or any type of var.
`bystats`	`bystats(y,...,fun=...)`	stratified statistics
`summary.formula`	`summary(y ∼ ...)`	flexible stratified statistics
`summarize`	`summarize(x,byvar,FUN)`	multi–way stratified statistics
`cumcategory`	`cumcategory(y)`	make dummies to summarize ordinal `y`
`binconf`	`binconf(successes,events,alpha)`	exact and Wilson C.L. for probability
`smean.cl.normal`	`smean.cl.normal(y)`	compute normal (t) C.L.
`smean.sd`	`smean.sd(y)`	mean and std. dev.
`smean.sdl`	`smean.sdl(y)`	mean $\pm$ constant $\times$ s.d.
`smean.cl.boot`	`smean.cl.boot(y)`	nonparametric boot. C.L. for mean
`smedian.hilow`	`smedian.hilow(y)`	median and 2 symmetric tailed quantiles
`hoeffd`	`hoeffd(x,y)`	Hoeffding D statistic
`rcorr`	`rcorr(...)`	linear or rank correlation matrix
`rcorr.cens`	`rcorr.cens(...)`	Somers D_{xy} rank correlation for censored data
`rcorrp.cens`	`rcorrp.cens(...)`	modification of `rcorrp.cens` for paired predictors
`bootkm`	`bootkm(S,q=,times=)`	Bootstrap Kaplan-Meier estimates

```
> bystats(age,stage,status,fun=quantile)

 quantile of age by stage, status

                                N Missing  0%   25%  50%   75% 100%
3 alive                        96       0  49 67.75 72.0 74.00   83
4 alive                        51       1  50 68.00 72.0 75.00   84
3 dead - prostatic ca          35       0  55 66.50 72.0 74.50   78
4 dead - prostatic ca          95       0  49 64.00 73.0 76.00   82
3 dead - heart or vascular     66       0  54 71.00 73.0 76.00   88
4 dead - heart or vascular     30       0  68 71.00 73.0 75.00   87
3 dead - cerebrovascular       21       0  62 72.00 75.0 76.00   80
4 dead - cerebrovascular       10       0  48 72.25 74.0 76.00   78
3 dead - pulmonary embolus     10       0  68 70.50 76.0 79.00   83
4 dead - pulmonary embolus      4       0  62 68.75 74.0 77.25   78
3 dead - other ca              20       0  51 72.00 74.5 76.00   80
4 dead - other ca               5       0  66 72.00 73.0 73.00   76
3 dead - respiratory disease   12       0  59 72.00 75.0 81.00   89
4 dead - respiratory disease    4       0  70 73.75 76.0 77.75   80
3 dead - other specific non-ca 19       0  62 68.50 76.0 78.00   81
4 dead - other specific non-ca  9       0  61 71.00 72.0 73.00   83
3 dead - unspecified non-ca     3       0  73 73.50 74.0 76.00   78
4 dead - unspecified non-ca     4       0  71 71.75 75.5 79.25   80
3 dead - unknown cause          7       0  52 64.50 71.0 73.00   76
                           ALL 501      1  48 70.00 73.0 76.00   89
```

The default value for `fun` is `mean`.

Several statistical summary functions are useful with `summary.formula`, `summarize`, `tapply`, `apply`, and by themselves. These functions (`cumcategory` through `smedian.hilow` in the table) provide statistical summaries for printing and plotting (including "error bars"; see Section 11.4.3). Here are some examples based on a sample of size 500 from a uniform(0,1) distribution:

```
> set.seed(2)  # so can replicate example
> x ← runif(500)
> smean.cl.normal(x)
  Mean Lower Upper
 0.501 0.475 0.527
> smean.cl.boot(x)
  Mean Lower Upper
 0.501 0.476 0.526
> smean.sd(x)
  Mean    SD
 0.501 0.292
> smean.sdl(x)
> # mean +- 2 s.d. (smean.sdl(x,1) to get mean +- s.d.)
  Mean   Lower Upper
 0.501 -0.0831  1.08
> smedian.hilow(x)
> # median and .025, .975 quantiles (conf.int=.5 for quartiles)
 Median  Lower Upper
```

```
    0.522 0.0201 0.971
```

The `rcorr.cens`, `rcorrp.cens`, and `bootkm` functions in Hmisc are used with right–censored failure–time data. The first two compute rank correlation measures for censored response data, and `bootkm` bootstraps Kaplan–Meier survival probability or quantile estimates. The help files for these functions give more information.

5.2 Functions for Probability Distributions

For each distribution in Table 5.2, four functions are available: one for densities, one for cumulative distribution functions, one for quantiles and one to obtain random samples from that distribution. Add the prefixes *d, p, q* or *r* to the *Name* column to get the name of the desired function. Functions in Hmisc related to probability distributions include `ecdf` (which plots the step–function empirical cumulative distribution function of a vector or of all the continuous variables in a data frame) and `bpplot` (box–percentile plots). See Section 11.3 for more about these as well as information about the Hmisc `scat1d` and `histSpike` functions for drawing rug plots, histograms, and density estimates.

Table 5.2: *Probability Distribution Functions*

Distribution	Name	Parameters
Beta	`beta`	`shape1`
		`shape2`
Binomial	`binom`	`size`
		`prob`
Cauchy	`cauchy`	`location`
		`scale`
χ^2	`chisq`	`df`
Exponential	`exp`	`rate`
F	`f`	`df1`
		`df2`
Gamma	`gamma`	`shape`
Geometric	`geom`	`shape`
Lognormal	`lnorm`	`meanlog`
		`sdlog`
Logistic	`logis`	`location`
		`scale`
Negative Binomial	`nbinom`	`size`
		`prob`
Normal (Gaussian)	`norm`	`mean`
		`sd`
Poisson	`pois`	`lambda`
Student's t	`t`	`df`
Uniform	`unif`	`min`
		`max`
Weibull	`weibull`	`shape`
Empirical cdf	`ecdf`	
Box–percentile plot	`bpplot`	list of vectors

Here are some examples.

```
# Compute the probability of getting 3 or fewer heads out of 10
# tosses of a fair coin

> pbinom(3, 10, .5)
[1] 0.171875

# Use the Hmisc function binconf to compute an exact 0.99 confidence
# interval for the unknown probability of an event given 6 events
# were observed out of 10 trials.  The Wilson score test-based
# interval has been shown to offer more accurate coverage than the
# exact beta-distribution-based method.  It's often the case that
# so-called 'exact' methods are conservative (Fisher's exact test for
# comparing two proportions can be quite conservative)

> binconf(6, 10, .01)

       Lower Upper
 Exact 0.191 0.923
Wilson 0.248 0.872

# Do an F-test to test H0: two variances are equal, given
# a sample standard deviation of 35.6 from n=100 and an
# s.d. of 17.3 from n=74.  See Rosner 4th Edition Ex. 8.15 P. 268

> vratio ← (35.6/17.3)^2
> vratio
[1] 4.234555
> 2*(1 - pf(vratio, 99, 73))  # 2-tailed P-value
[1] 8.612535e-010

# Compute a 0.95 confidence interval for the population variance ratio

> ratios ← qf(c(.025,.975), 99, 73)
> ratios
[1] 0.654760 1.549079
> vratio/ratios
[1] 6.467339 2.733596
```

Had we been using the raw data for the last example, the calculations could have been done using the S builtin function `var.test` if we provided it the raw data. We can thankfully work backwards to generate raw data having the needed mean and standard deviation, just so we can use `var.test`. The following function will generate a vector of length `n` having sample mean and standard deviation exactly equal to given constants. After the appropriate vectors are computed we can use `var.test`.

```
> gen.mean.sd ← function(n, xbar = 0, sd = 1) {
+   x ← 1:n
+   xbar + sd * (x - mean(x))/sqrt(var(x))
+ }
> y1 ← gen.mean.sd(100, sd=35.6)
> mean(y1)
[1] 3.552714e-016
> sqrt(var(y1))
[1] 35.6
> y2 ← gen.mean.sd(74, sd=17.3)
> var.test(y1,y2)

    F test for variance equality

data:  y1 and y2
F = 4.2346, num df = 99, denom df = 73, p-value = 0
alternative hypothesis: true ratio of variances is not equal to 1
95 percent confidence interval:
 2.733596 6.467339
sample estimates:
 variance of x variance of y
       1267.36        299.29
```

5.3 Hmisc Functions for Power and Sample Size Calculations

Table 5.3 lists functions in Hmisc related to statistical power.

Table 5.3: *Hmisc Functions for Power/Sample Size*

Function	Purpose
`ballocation`	Find optimum allocation ratio for treatments for binary responses
`bpower`	Power of two–sample binomial test (approximate; for comparing two proportions)
`bpower.sim`	Power of two-sample binomial test using simulation
`bsamsize`	Sample size for two–sample binomial test
`ciapower`	Power of interaction test for exponential survival (and for Cox model)
`cpower`	Power of Cox/log–rank two–sample Test
`gbayes`	Gaussian Bayesian posterior and predictive distributions (and simple conditional power dist.)
`popower`	Power for two–sample test for ordinal responses
`posamsize`	Sample size for two–sample ordinal responses
`samplesize.bin`	Sample size for 2-sample binomial using $\sin^{-1}\sqrt{p}$ transformation (by Rick Chappell)
`spower`	Power of Cox/log–rank two–sample test for complex situations via simulation

The `ballocation`, `bpower.sim` and `bsamsize` functions are documented under the heading of the `bpower` function. `posamsize` is listed under `popower`.

In the following example, both the bpower and bpower.sim functions are used to estimate power for the two–sample binomial test (comparison of two proportions). bpower.sim can simulate power of the χ^2_1 test very quickly because S has a builtin binomial random number generator. By default, 10000 simulations are done (this takes about 5 seconds). 0.95 confidence limits for the estimated power based on the simulations are reported.

```
> args(bpower)
function(p1, p2, odds.ratio, percent.reduction, n, n1, n2, alpha = 0.05)
> args(bpower.sim)
function(p1, p2, odds.ratio, percent.reduction, n, n1, n2, alpha = 0.05,
         nsim = 10000)
> bpower(.2, odds.ratio=2, n=200)
     Power
 0.5690973
> bpower.sim(.2, odds.ratio=2, n=200)
  Power     Lower     Upper
 0.5581 0.5483664 0.5678336

> bpower.sim(.2, odds.ratio=2, n=200, nsim=25000)
   Power     Lower     Upper
 0.56256 0.5564106 0.5687094
> args(bsamsize)
function(p1, p2, fraction = 0.5, alpha = 0.05, power = 0.8)
> bsamsize(.2, plogis(qlogis(.2)+log(2)), power=.5690973)
       n1       n2
 100.0041 100.0041
```

Note that bsamsize does not allow specification of an odds ratio. We used the logistic and inverse logistic transform to get the second proportion by applying an odds of 2 to the first proportion (.2).

Next we compute power for a proportional odds two–sample test for comparing two ordinal responses. The first calculation is for an ordinal response with only two levels. Power for this situation should be close to the 0.56 just computed, but in fact it is different, as popower uses a normal approximation for the log odds ratio instead of subtracting the two proportions. This approximation is not as good as the method used by bpower when there are two categories. For each application of popower we assume that the marginal frequencies of responses are equal across response categories. You can see that when more ordered categories are used, the power increases (especially when the cell frequencies are equal).

```
> args(popower)
function(p, odds.ratio, n, n1, n2, alpha = 0.05)
> popower(c(.5,.5), 2, 200)
Power: 0.684
Efficiency of design compared with continuous response: 0.75

> popower(c(1,1,1)/3, 2, 200)
Power: 0.756
Efficiency of design compared with continuous response: 0.889

> popower(c(1,1,1,1)/4, 2, 200)
Power: 0.778
```

```
Efficiency of design compared with continuous response: 0.938

> popower(c(1,1,1,1,1)/5, 2, 200)
Power: 0.788
Efficiency of design compared with continuous response: 0.96
```

In the next example, taken from the `bpower` help file, we plot power vs. the total sample size n for various odds ratios, using 0.1 as the probability of the event in the control group. A separate curve is plotted for each odds ratio, and the odds ratio is drawn just below the curve at $n = 350$.

```
n  ← seq(10, 1000, by=10)
OR ← seq(.2,.9,by=.1)
plot(0, 0, xlim=range(n), ylim=c(0,1), xlab="n", ylab="Power", type="n")
for(or in OR) {
  lines(n, bpower(.1, odds.ratio=or, n=n))
  text(350, bpower(.1, odds.ratio=or, n=350)-.02, format(or))
}
```

Now re–do the plot, letting Hmisc's `labcurve` function do the work of drawing the curves, determining overall axis limits, and labeling curves at points of maximum separation.

```
pow ← lapply(OR, function(or,n)list(x=n,y=bpower(p1=.1,odds.ratio=or,n=n)),
             n=n)
names(pow) ← format(OR)
labcurve(pow, pl=T, xlab='n', ylab='Power')
```

The `cpower` function for estimating power for the Cox/log–rank two–sample test has many options that allow a time–to–event study to have several complexities. Here is an excerpt from the help file.

```
DESCRIPTION

Assumes exponential distributions for both treatment groups. Uses the
George-Desu method along with formulas of Schoenfeld that allow
estimation of the expected number of events in the two groups.  To
allow for drop-ins (noncompliance to control therapy, crossover to
intervention) and noncompliance of the intervention, the method of
Lachin and Foulkes is used.

USAGE

cpower(tref, n, mc, r, accrual, tmin, noncomp.c=0, noncomp.i=0,
       alpha=0.05, nc, ni, pr=T)

REQUIRED ARGUMENTS

tref    time at which mortalities estimated
n       total sample size (both groups combined).  If allocation is unequal
        so that there are not n/2 observations in each group, you may
        specify the sample sizes in nc and ni.
mc      tref-year mortality, control
r       % reduction in mc by intervention
accrual duration of accrual period
tmin    minimum follow-up time

OPTIONAL ARGUMENTS

noncomp.c  % non-compliant in control group (drop-ins)
noncomp.i  % non-compliant in intervention group (non-adherers)
alpha      type I error probability.  A 2-tailed test is assumed.
nc         number of subjects in control group
ni         number of subjects in intervention group.
           nc and ni are specified exclusive of n.
pr         set to F to suppress printing of details
```

The help file for `cpower` has an example in which 4 plots are drawn on one page, one plot for each combination of noncompliance percentage. Within a plot, the 5–year mortality % in the control group is on the x–axis, and separate curves are drawn for several % reductions in mortality with the intervention. The accrual period is 1.5y, with all patients followed at least 5y and some 6.5y.

The `spower` function is much slower than `cpower` as it relies on simulation, but it allows for very complex clinical trial setups. `cpower` works with `Quantiles2` and other functions documented under the `spower` heading. The following paragraph is taken from `spower`'s help file:

Given functions to generate random variables for survival times and censoring times, `spower` simulates the power of a user–given 2–sample test for censored data. By default, the logrank (Cox 2–sample) test is used, and a `logrank` function for comparing 2 groups is provided. For composing S functions to generate random survival times under complex conditions, the `Quantile2` function allows the user to specify the intervention:control hazard ratio as a function of time, the probability of a control subject actually receiving the intervention (dropin) as a function of time, and the probability that an intervention subject receives only the control agent as a function of time (non–compliance, dropout). `Quantile2` returns a function that generates either control or intervention uncensored survival times subject to non–constant treatment effect, dropin, and dropout. There is a `plot` method for plotting the results of `Quantile2`, which will aid in understanding the effects of the two types of non–compliance and non–constant treatment effects. `Quantile2` assumes that the hazard function for either treatment group is a mixture of the control and intervention hazard functions, with mixing proportions defined by the dropin and dropout probabilities. It computes hazards and survival distributions by numerical differentiation and integration using a grid of (by default) 7500 equally–spaced time points.

Besides providing the `Quantile2` function, the `spower` package also contains three functions which compose S functions that compute survival probabilities for Weibull, log–normal, and Gompertz distributions. These functions (`Weibull2`,`Lognorm2`, and `Gompertz2`) work by solving for the two parameters of each of these distributions which make them fit two user–specified times and survival probabilities. The 3 types of functions so created are useful as the first argument to `Quantile2`.

The following example demonstrates the flexibility of `spower` and related functions. We simulate a 2–arm (350 subjects/arm) 5–year follow–up study for which the control group's survival distribution is Weibull with 1–year survival of .95 and 3–year survival of .7. All subjects are followed at least one year, and patients enter the study with linearly increasing probability starting with zero. Assume (1) there is no chance of dropin for the first 6 months, then the probability increases linearly up to .15 at 5 years; (2) there is a linearly increasing chance of dropout up to .3 at 5 years; and (3) the treatment has no effect for the first 9 months, then it has a constant effect (hazard ratio of .75).

```
> # First find the right Weibull distribution for compliant control patients
> # Weibull2 is bundled with spower
> sc ← Weibull2(c(1,3), c(.95,.7))
> sc

function(times, alpha = 0.0512932943875506, gamma = 1.76519490623438
   )

   exp( - alpha * (times^gamma))


> # Inverse cumulative distribution for case where all subjects are followed
> # at least a years and then between a and b years the density rises
> # as (time - a) ^ d is a + (b-a) * u ^ (1/(d+1))

> rcens ← function(n) 1 + (5-1) * (runif(n) ^ .5)
> # To check this, type hist(rcens(10000), nclass=50)

> # Put it all together

> f ← Quantile2(sc,
+                 hratio=function(x)ifelse(x <= .75, 1, .75),
+                 dropin=function(x)ifelse(x <= .5, 0, .15*(x-.5)/(5-.5)),
+                 dropout=function(x).3*x/5)

> par(mfrow=c(2,2))  # 2x2 matrix of plots
> plot(f, 'all', label.curves=list(keys='lines'))
> #omitting label.curves= will cause labcurve to label curves directly
```

The function `f` created by `Quantile2` has as its main arguments `n`, the number of random variates to draw, and `what`, telling the function whether to draw samples from the uncensored survival times for control or intervention. The `plot(f,...)` statement produced Figure 5.1.

Now we ask `spower` to simulate the needed results, basing the survival distribution comparison on the log–rank test.

```
> rcontrol ← function(n) f(n, 'control')
> rinterv  ← function(n) f(n, 'intervention')

> set.seed(211)
> spower(rcontrol, rinterv, rcens, nc=350, ni=350, test=logrank, nsim=300)
[1] 0.4033333
```

See Section 4.7 for an example in which power for a logistic model is estimated via simulation.

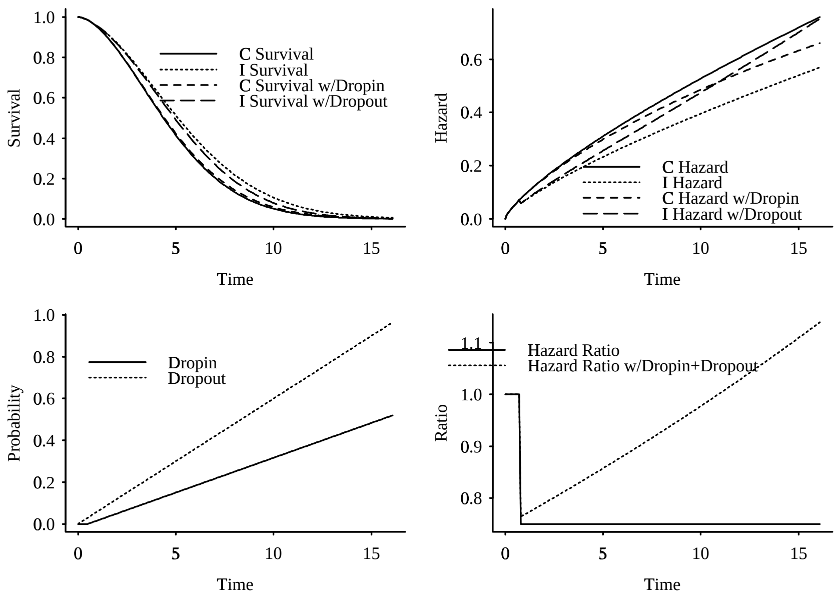

Figure 5.1: *Characteristics of control and intervention groups with a lag in the treatment effect and with non–compliance in two directions*

5.4 Statistical Tests

In general, we do not prefer to use specialized functions for many of the common statistical tests for three reasons. (1) Many tests are special cases of regression models. (2) The notation used in regression models unifies many of the concepts involved in statistical inference. (3) Regression models can provide estimates of effects, not just tests of (sometimes inappropriate) null hypotheses. Regarding point (2), the two–sample t–test is a special case of the linear regression model with a single binary predictor variable. The two–sample Wilcoxon test is a special case of the proportional odds ordinal logistic model again with a single binary predictor. The two–sample Wilcoxon test is also a special case of the Spearman rank correlation test. The k–sample generalizations of these tests (analysis of variance and Kruskal–Wallis test) may be obtained by using the two above mentioned regression models with $k-1$ dummy variables. The χ^2 test for a $k \times 2$ contingency table is a special case of a binary logistic model with $k-1$ dummy predictors, and the likelihood ratio χ^2 test from this model yields P–values that are more accurate than the traditional χ^2 test. The log–rank test is a special case of the Cox model.

The entire list of statistical tests builtin to S-Plus may be obtained under Microsoft Windows by clicking under `Index` and entering `Statistical Inference`. You can also get the list by typing the command `help('Statistical Inference')`. Table 5.4 lists these functions.

Table 5.4: *S Functions for Statistical Tests*

Description	Function
χ^2 Goodness–of–fit	`chisq.gof`
Exact binomial 1–sample	`binom.test`
F test for variances	`var.test`
Fisher's exact test for $p \times q$ table	`fisher.test`
Friedman rank sum	`friedman.test`
Graph two cumulative distributions	`cdf.compare`
Kolmogorov–Smirnov goodness–of–fit	`ks.gof`
Kruskal–Wallis rank sum	`kruskal.test`
Mantel–Haenszel χ^2	`mantelhaen.test`
McNemar χ^2	`mcnemar.test`
Pearson χ^2	`chisq.test`
Proportion tests	`prop.test`
Student t	`t.test`
Correlation	`cor.test`
Wilcoxon 1– and 2–sample	`wilcox.test`

It is not clear why Fisher's exact test is implemented in S, as this test is known to lose power when compared with unconditional χ^2 tests[1]

[1]Note that the "rule" that ordinary χ^2 tests should not be used when an expected cell frequency is < 5 is not correct. Pearson χ^2 works well in situations more extreme than that, and the likelihood ratio χ^2 may work even better.

5.4.1 Nonparametric Tests

The Spearman correlation test (and hence the two–sample Wilcoxon test) may be obtained using the Hmisc `spearman.test` function:

```
> set.seed(17) # so can reproduce results
> sex ← factor(sample(c('female','male'),100,T))
> blood.pressure ← rnorm(100, 100, 8)+3*(sex=='male')
> options(digits=3)
> spearman.test(sex, blood.pressure)
 Rsquare    F df1 df2  pvalue   n
  0.0713 7.52   1  98 0.00725 100
```

You can also obtain the Spearman test from the Hmisc `rcorr` function [better still, S has a builtin function `cor.test` that does Spearman and Pearson linear correlation tests]. Note that $0.27^2 = 0.071$ and the two methods give identical two–tailed P–values.

```
> rcorr(sex, blood.pressure, 'spearman')
     x    y
x 1.00 0.27
y 0.27 1.00

n= 100


P
       x      y
x        0.0073
y 0.0073
```

The Hmisc `somers2` function provides a more easily interpreted correlation measure for the case where one variable is binary. Here Somers' D_{xy} rank correlation between x=blood pressure and y=sex is computed, along with the probability of concordance between the two variables denoted by `C`.

```
> somers2(blood.pressure,sex='male')
     C   Dxy   n Missing
 0.654 0.308 100       0
```

The Hmisc `rcorr.cens` can also compute this correlation as a special case where censoring is absent.

```
> rcorr.cens(blood.pressure,sex)
 C Index   Dxy  S.D.   n missing uncensored Relevant Pairs Concordant Uncertain
   0.654 0.308 0.109 100       0        100           4992       3266         0
```

This can be used to get a statistical test using a normal approximation.. Here we compute the two–tailed P–value.

```
> (1-pnorm(.308/.109))*2
[1] 0.00471
>
```

`spearman.test` can also test for non–monotonic relationships between two continuous variables, by allowing the user to specify an order of the polynomial of the ranks used in the correlation test. For example, to get a two d.f. test of association between age and blood pressure, allowing for one "turn" in the non–monotonic function, one could use `spearman.test(age, blood.pressure, 2)`.

The `spearman2` function in Hmisc is the most general of the Spearman–type functions. It uses the F approximation to do a Spearman and second–order generalized Spearman test (as done by `spearman.test`, if the predictor variable is continuous), the Wilcoxon–Mann–Whitney two–sample test, and the Kruskal–Wallis test (for factor predictors having more than 2 levels). `spearman2` can test a series of predictors against a common response variable, with pairwise deletion of missing data. Here is an example in which the numerator degrees of freedom are 1, 1, and 4, respectively, for age (continuous), sex (binary), and race (5 levels).

```
> spearman2(blood.pressure ~ age + sex + race)
```

The S builtin function `wilcox.test` has more features for one[2]– and two–sample Wilcoxon tests. It's use is somewhat awkward for the two–sample case:

```
> wilcox.test(blood.pressure[sex=='female'], blood.pressure[sex=='male'])

data:  blood.pressure[sex == "female"] and blood.pressure[sex == "male"]
rank-sum normal statistic with correction Z = -2.65, p-value = 0.008
alternative hypothesis: true mu is not equal to 0
```

For a one–sample test, omit the second argument to `wilcox.test`.

Next obtain the two–sample Wilcoxon test as a special case of the proportional odds model. This approach will give very accurate P–values (as well as an effect measure — the odds ratio) although it takes computer time and RAM to fit 99 intercepts for the 100 observations that contain no tied response values[3]

```
> library(Design, T)
> lrm(blood.pressure ∼ sex)
 Obs Max Deriv Model L.R. d.f.     P     C   Dxy Gamma Tau-a   R2 Brier
 100    5e-013       7.27    1 0.007 0.578 0.156 0.308 0.156 0.07  0.01

                          Coef   S.E. Wald Z      P
y>=83.1881961293538   4.214947 1.0140   4.16  0.0000
y>=85.4474231763989   3.511529 0.7269   4.83  0.0000
y>=86.1909623865412   3.092591 0.6018   5.14  0.0000
. . . . .
y>=116.829586001076 -4.065680 0.6321 -6.43  0.0000
y>=118.266935916644 -4.485844 0.7529 -5.96  0.0000
 y>=119.44799460213 -5.193478 1.0332 -5.03  0.0000
           sex=male  0.951687 0.3569  2.67  0.0077
```

The Wald test P–value is 0.008 and the somewhat more accurate likelihood ratio test yields P=0.007, in good agreement with what we obtained from the simpler tests. The D_{xy} rank correlation printed above does not agree with the earlier ones because we have reversed the roles of x and y.

[2]Wilcoxon signed–rank test

[3]Note that there are no statistical problems in fitting 100 parameters for 100 observations here, as the intercepts are constrained to be in order.

Various other nonparametric testing functions are listed in table 5.4. These include tests for goodness–of–fit, frequencies, proportions, blocked data, and tests for distributional shapes.

5.4.2 Parametric Tests

The *t* test may be obtained using the builtin `t.test` function.

```
> t.test(blood.pressure[sex=='female'], blood.pressure[sex=='male'])

     Standard Two-Sample t-Test

data:  blood.pressure[sex == "female"] and blood.pressure[sex == "male"]
t = -2.6769, df = 98, p-value = 0.0087
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -6.82129 -1.01326
sample estimates:
 mean of x mean of y
    98.656   102.573
```

Consider Example 8.20 on P. 275 of Rosner (4th Edition) in which the `hospital` dataset is used. Here we test for zero mean difference (why the mean?) in duration of hospitalization, for patients receiving an antibiotic compared with those who didn't.

```
> attach(hospital)
> t.test(duration[antibiotic=='yes'],duration[antibiotic=='no'])

    Standard Two-Sample t-Test

data:  duration[antibiotic == "yes"] and duration[antibiotic == "no"]
t = 1.6816, df = 23, p-value = 0.1062
alternative hypothesis: true difference in means is not equal to 0
95 percent confidence interval:
 -0.9497745  9.2037428
sample estimates:
 mean of x mean of y
  11.57143  7.444444
```

Note that the confidence interval does not agree with Rosner's calculations, as Rosner inappropriately used 6 d.f. for the *t* distribution instead of 23 d.f. In S-Plus 4.x these results may be obtained using the menus: `Statistics .. Compare Samples .. Two Samples .. t-test`; use `antibiotic` as a grouping variable.

Now consider a one–sample *t*–test using the data on the effects of oral contraceptive (OC) on systolic blood pressure found in Rosner Table 8.1 on P. 253. Here is a listing of the data file named `table81.asc`:

```
sbp.noOC  sbp.OC
115   128
112   115
107   106
```

```
119   128
115   122
138   145
126   132
105   109
104   102
115   117
```

Note that variable names are in the first record. Here fields are separated by a tab. In S-Plus 4.x we may import this data file using `File` ... `Import Data` ... `From File` ... and then browsing to find the file. Then click `OK`, using all defaults. Under any version of S we may import the data using the command

```
> table81 ← read.table('/directoryname/table81.asc',header=T)
```

The one–sample t–test and associated confidence interval for the difference in means may be obtained as follows:

```
> t.test(sbp.OC,sbp.noOC,paired=T)

    Paired t-Test

data:  sbp.OC and sbp.noOC
t = 3.3247, df = 9, p-value = 0.0089
alternative hypothesis: true mean of differences is not equal to 0
95 percent confidence interval:
 1.533987 8.066013
sample estimates:
 mean of x - y
           4.8
```

Under S-Plus 4.x you can use `Statistics` ... `Compare Samples` ... `Two Samples` ... `t Test`. Check the box marked `Paired t`. Had you already computed a new variable containing the difference in the two columns, you could use `Statistics` ... `Compare Samples` ... `One Sample` ... `t Test`.

Other parametric testing functions are shown in Table 5.4. These include tests for equality of variances and tests for zero correlations (why zero?).

Chapter 6

Making Tables

6.1 S-Plus–supplied Functions

Section 4.3.2 showed how to use functions such as `tapply` to make simple tables. The S `print.char.matrix` function may be used to format many tables into attractively boxed cells. The `crosstabs` function produces frequency tables and computes Pearson χ^2 statistics, printing results using `print.char.matrix`. Here is an example from the online help, using the S-Plus–supplied `solder` dataset.

```
> crosstabs(~ Solder+Opening, data=solder, subset = skips>10)
Call:
```

```
crosstabs( ~ Solder + Opening, data = solder, subset = skips > 10)
158 cases in table
+----------+
|N         |
|N/RowTotal|
|N/ColTotal|
|N/Total   |
+----------+
Solder |Opening
       |S      |M      |L      |RowTotl|
-------+-------+-------+-------+-------+
Thin   |99     |15     | 9     |123    |
       |0.805  |0.122  |0.073  |0.78   |
       |0.805  |0.577  |1.000  |       |
       |0.627  |0.095  |0.057  |       |
-------+-------+-------+-------+-------+
Thick  |24     |11     | 0     |35     |
       |0.686  |0.314  |0.000  |0.22   |
       |0.195  |0.423  |0.000  |       |
       |0.152  |0.070  |0.000  |       |
-------+-------+-------+-------+-------+
ColTotl|123    |26     |9      |158    |
       |0.778  |0.165  |0.057  |       |
-------+-------+-------+-------+-------+
Test for independence of all factors
    Chi^2 = 9.18309 d.f.= 2 (p=0.01013719)
    Yates' correction not used
    Some expected values are less than 5, don't trust stated p-value
```

Note that the first argument to `crosstabs` is an S *formula*. Normally a formula has a dependent or response variable followed by a tilde followed by one or more independent or predictor variables, separated by "+". A contingency table as such does not have a response variable as it treats row and column variables symmetrically. Therefore a formula given to `crosstabs` specifies only a series of "independent" variables. Functions which operate on formulas provide a number of advantages:

1. Formulas allow the user to specify any number of variables to analyze.
2. Functions which use formulas also allow for an argument called `data` that specifies a data frame or list that contains the analysis variables. You need not attach the data frame to get access to these variables.
3. Functions which use formulas also allow the user to specify a `subset` argument, to easily specify that an analysis is to be run on a subset of the observations. The value specified to `subset` is a logical vector or a vector of integer subscripts.

The object created by `crosstabs` contains much useful information including marginal summaries that can be plotted. Let's re–run the last table, saving the result and then printing part of it.

```
> g ← crosstabs( ~ Solder + Opening, data = solder, subset = skips > 10)
> rowpct ← 100*attr(g,'marginals')$"N/RowTotal"
> #                               $'N/ColTotal' to get col. percents
> #                               $'N/Total'    to get overall percent
> options(digits=3)
> rowpct
         S    M    L
 Thin 80.5 12.2 7.32
Thick 68.6 31.4 0.00
```

The `rowpct` matrix contains the row percentages, as can be seen by comparing with the full table above. To plot these row percents using `trellis` graphics (see Section 11.4) we first need to reshape the `rowpct` matrix into a a vector as was done in Section 4.3.8:

```
> y       ← as.vector(rowpct)   # strung-out vector
> Solder  ← dimnames(rowpct)[[1]][row(rowpct)]
> Opening ← dimnames(rowpct)[[2]][col(rowpct)]
> data.frame(Solder, Opening, y)

  Solder Opening     y
1   Thin       S 80.49
2  Thick       S 68.57
3   Thin       M 12.20
4  Thick       M 31.43
5   Thin       L  7.32
6  Thick       L  0.00

> dotplot(Solder ~ y | Opening)
> dotplot(Solder ~ y, groups=Opening, panel=panel.superpose)
> barchart(Opening ~ y | Solder)
```

The second dot plot is probably more effective, as the sum of values indicated by all the points on each line is 100%. The Hmisc `reShape` function provides a shortcut:

```
> w ← reShape(rowpct)
> w
$Solder:
[1] "Thin"  "Thick" "Thin"  "Thick" "Thin"  "Thick"

$Opening:
[1] "S" "S" "M" "M" "L" "L"

$rowpct:
[1] 80.487805 68.571429 12.195122 31.428571  7.317073
[6]  0.000000

> dotplot(Solder ~ rowpct, groups=Opening,
+         panel=panel.superpose, data=w)
> # Note w has variable named rowpct (name of argument to reShape)
> # Other variables got their names originally from crosstabs formula
```

Note that you can also compute row or column proportions or percents using the `table`, `apply`, and `sweep` functions.

6.2 The Hmisc `summary.formula` Function

The `summary.formula` function (called using the `summary` command on a `formula` object) constructs a large variety of tables of descriptive statistics. The tables can automatically be typeset using LaTeX. The default format for typesetting tables this way is the "Biometrika/New England Journal of Medicine" format, i.e., it makes minimal use of vertical lines.

Some of the tables can automatically be converted into dot charts by one of `summary.formula`'s `plot` methods.

Part of what makes `summary.formula` work is that the user can specify her own function (`fun`) to compute descriptive statistics[1] and this function may be multivariate. For example, it may operate on two response variables, producing two or more summary statistics, or it may compute a single summary statistic on the two responses. If the two responses are a survival time and an event/censoring indicator, you can summarize the survival times using Kaplan–Meier or other estimators. If the two responses are the predicted probability of a disease and whether or not the disease is actually present, the summary measure could be a "receiving operator characteristic curve area." You can also specify that `fun` is to return several statistics from each response variable (e.g., mean and median).

In addition to its flexibility, `summary.formula` has two general advantages over builtin S functions. First, it removes `NA`s before passing vectors to standard S statistical functions (`mean`, `median`, etc.) so that you do not need to worry about using an `na.rm=T` argument. Second, statistical summaries made by `summary.formula` automatically include *marginal* summaries. For example, if you stratify data on a variable you will also see unstratified estimates, and if you cross–classify on two or more variable you will also see estimates stratified on all subsets of the variables. Thus cross–classifying on `race` and `sex` and computing the median `cholesterol` will (unless you specify an argument to suppress them) also compute medians stratified separately by `race` and by `sex` as well as the grand median `cholesterol`.

There are three major ways of using `summary.formula`, as defined by the `method` parameter. `method='response'` (the default) causes the function to summarize one or more response variables separately by levels of any number of right–hand–side variables. `method='cross'` results in a multiway breakdown. Categorical right–hand variables are broken down into all of their levels. Continuous variables are grouped, by default, into quartiles, to summarize the responses. The `'cross'` method causes `summary.formula` to output a data frame containing summary statistics, which is the format in which `trellis` expects to find raw data. This makes it easy to plot summary statistics using `trellis` although the `summarize` function works better for this. `method='reverse'` reverses the meaning of the left–hand and right–hand–side variables. For example, `summary(treatment ~ age + blood.pressure, method='reverse')` will print k columns, where k is the number of levels in `treatment`. For each column, descriptive statistics will be computed for `age` and `blood.pressure`. For continuous variables, the descriptive statistics default to the three quartiles. For categorical ones, frequencies and percentages are computed.

As discussed in Section 3.2.3, nice `labels` and category `levels` should have been created early in the process. `summary.formula` will take full advantage of this.

Here are some of the examples from the online help.

```
> options(digits=3)
```

[1]Note the inconsistency here: functions such as `tapply`, `aggregate`, and `by` which are built–in to S capitalize the argument into the name `FUN`.

```
> # Generate some data
> set.seed(173)    # so can replicate results
> sex ← factor(sample(c("m","f"), 500, rep=T))
> age ← rnorm(500, 50, 5)
> treatment ← factor(sample(c("Drug","Placebo"), 500, rep=T))
> # Frequency table sex*treatment
> summary(sex ∼ treatment, fun=table)



sex    N=500

---------+-------+---+---+---+
         |       |N  |f  |m  |
---------+-------+---+---+---+
treatment|Drug   |246|123|123|
         |Placebo|254|129|125|
---------+-------+---+---+---+
Overall  |       |500|252|248|
---------+-------+---+---+---+

> # Compute mean age, separately by sex and treatment
> summary(age ∼ sex + treatment)


age    N=500

---------+-------+---+----+
         |       |N  |age |
---------+-------+---+----+
sex      |f      |252|49.8|
         |m      |248|49.9|
---------+-------+---+----+
treatment|Drug   |246|49.7|
         |Placebo|254|50.0|
---------+-------+---+----+
Overall  |       |500|49.9|
---------+-------+---+----+
> summary(age ∼ sex + treatment, method="cross")

 Mean of age by sex, treatment

+---+
|N  |
|age|
+---+
sex|treatment
   |Drug|Plac|ALL |
---+----+----+----+
f  |123 |129 |252 |
   |49.4|50.3|49.8|
---+----+----+----+
m  |123 |125 |248 |
   |50.1|49.7|49.9|
---+----+----+----+
ALL|246 |254 |500 |
   |49.7|50.0|49.9|
---+----+----+----+


> summary(treatment ∼ age + sex, method="reverse")

Descriptive Statistics by treatment

-------+--------------+--------------+
       |Drug          |Placebo       |
       |(N=246)       |(N=254)       |
-------+--------------+--------------+
age    |46.5/49.8/52.5|46.4/50.1/53.4|
-------+--------------+--------------+
sex : m|   50% (123)  |   49% (125)  |
-------+--------------+--------------+

> # a/b/c represents the lower quartile, median, upper quartile


> # Compute predicted probability from a logistic regression model
> # For different stratifications compute receiver operating
> # characteristic curve areas (C-indexes)
> predicted ← plogis(.4*(sex=="m")+.15*(age-50))
> positive.diagnosis ← ifelse(runif(500)<=predicted, 1, 0)

> roc ← function(z) {
+   x ← z[,1];
+   y ← z[,2];
+   n ← length(x);
+   if(n<2)return(c(ROC=NA));
+   n1 ← sum(y==1);
+   c(ROC= (mean(rank(x)[y==1])-(n1+1)/2)/(n-n1) );
+ }
> y ← cbind(predicted, positive.diagnosis)
> options(digits=2)
> summary(y ∼ age + sex, fun=roc)



y    N=500

-------+-----------+---+----+
       |           |N  |ROC |
-------+-----------+---+----+
age    |[32.3,46.4)|125|0.62|
       |[46.4,50.0)|125|0.59|
       |[50.0,52.9)|125|0.61|
       |[52.9,68.6]|125|0.70|
-------+-----------+---+----+
sex    |f          |252|0.71|
       |m          |248|0.70|
-------+-----------+---+----+
Overall|           |500|0.72|
-------+-----------+---+----+


> options(digits=3)
> summary(y ∼ age + sex, fun=roc, method="cross")
```

```
 roc by age, sex

+---+
|N  |
|ROC|
+---+
age        |sex
           |f    |m    |ALL  |
-----------+-----+-----+-----+
[32.3,46.4)| 61  | 64  |125  |
           |0.632|0.527|0.620|
-----------+-----+-----+-----+
[46.4,50.0)| 67  | 58  |125  |
           |0.602|0.671|0.595|
-----------+-----+-----+-----+
[50.0,52.9)| 59  | 66  |125  |
           |0.517|0.445|0.613|
-----------+-----+-----+-----+
[52.9,68.6]| 65  | 60  |125  |
           |0.734|0.558|0.703|
-----------+-----+-----+-----+
ALL        |252  |248  |500  |
           |0.711|0.702|0.718|
-----------+-----+-----+-----+


# Plot estimated mean life length (assuming an exponential distribution)
# separately by levels of 4 other variables.  Repeat the analysis
# by levels of a column stratification variable, drug.  Automatically break
# continuous variables into tertiles (g=3).
# We are using the default, method='response'

life.expect ← function(y) c(Years=sum(y[,1])/sum(y[,2]))
attach(pbc)  # pbc is in UVa biostat web page
S ← Surv(fu.days/365.25, status)
options(digits=3)

summary(S ∼ age + albumin + ascites + edema + stratify(drug),
        fun=life.expect, g=3)
```

Here's an example using the `prostate` data frame.

```
> detach(2)    #  detach pbc
> attach(prostate)
> bone ← factor(bm,labels=c("no mets","bone mets"))
> summary(ap ∼ sz + bone,
+         fun=function(y) c(Mean=mean(y),quantile(y,c(.25,.5,.75))),
+         method='cross')
 c(Mean = mean(y), quantile(y, c(0.25, 0.5, 0.75))) by sz, bone

+----+
|N   |
|Mean|
|25% |
|50% |
|75% |
+----+
sz     |bone
       |no met|bone m|ALL   |
-------+------+------+------+
[ 0, 5)|105   |  5   |110   |
       |  1.0 | 17.7 |  1.8 |
       |0.40  |2.10  |0.40  |
       | 0.6  | 3.7  | 0.6  |
       |  1.00| 38.50|  1.10|
-------+------+------+------+
[ 5,11)|119   | 17   |136   |
       |  2.2 | 19.1 |  4.4 |
       |0.40  |1.20  |0.40  |
       | 0.6  | 1.7  | 0.6  |
       |  0.85|  9.30|  1.12|
-------+------+------+------+
[11,21)|103   | 19   |122   |
       |  6.5 |139.8 | 27.3 |
       |0.45  |5.30  |0.50  |
       | 0.6  |28.9  | 0.8  |
       |  1.45|134.14|  3.17|
-------+------+------+------+
[21,69]| 88   | 41   |129   |
       |  6.3 | 34.8 | 15.4 |
       |0.60  |4.60  |0.70  |
       | 1.0  |20.0  | 2.8  |
       |  5.67| 35.70| 11.80|
-------+------+------+------+
NA     |  5   |  0   |  5   |
       |  2.7 |      |  2.7 |
       |0.50  |      |0.50  |
       | 0.7  |      | 0.7  |
       |  1.70|      |  1.70|
-------+------+------+------+
ALL    |420   | 82   |502   |
       |  3.8 | 54.8 | 12.2 |
       |0.50  |2.02  |0.50  |
       | 0.7  |10.2  | 0.7  |
       |  1.30| 37.42|  2.97|
-------+------+------+------+
```

Table 6.1: *Descriptive Statistics by Treatment*

	N	D-penicillamine ($N = 154$)	placebo ($N = 158$)
Serum Bilirubin (mg/dl)	418	0.725 1.300 3.600	0.800 1.400 3.200
Albumin (gm/dl)	418	3.34 3.54 3.78	3.21 3.56 3.83
Histologic Stage, Ludwig Criteria : 1	412	3% $\frac{4}{154}$	8% $\frac{12}{158}$
2		21% $\frac{32}{154}$	22% $\frac{35}{158}$
3		42% $\frac{64}{154}$	35% $\frac{56}{158}$
4		35% $\frac{54}{154}$	35% $\frac{55}{158}$
Prothrombin Time (sec.)	416	10.0 10.6 11.4	10.0 10.6 11.0
Sex : female	418	90% $\frac{139}{154}$	87% $\frac{137}{158}$
Age	418	41.4 48.1 55.8	43.0 51.9 58.9
Spiders	312	29% $\frac{45}{154}$	28% $\frac{45}{158}$

$_a\ b\ _c$ represent the lower quartile a, the median b, and the upper quartile c for continuous variables. N is the number of non–missing values.

Note that it is possible to get wider column labels using some of `summary.formula`'s options. Also, where you send the output of the function to the Hmisc library's `latex` function, you get nicely typesetted tables. Here is an example using the `latex` function (actually `latex.default`) in conjunction with LaTeX. A `print` method in the Hmisc library for `latex` objects can be used in this setting for easy on-screen previewing of the typeset table.

```
> attach(pbc)
> s ← summary(drug ∼ bili + albumin + stage + protime + sex + age + spiders,
+              method='reverse')

> options(digits=3)
> latex(s, npct='both', npct.size='normalsize', here=T)
# npct='both' : print both numerators and denominators
# Use normalsize font for numerator and denominator of percents
```

The LaTeX output is in Table 6.1. The table legend at the bottom was produced by the `latex` function (actually `latex.summary.formula.reverse`). If you run the command `plot(s)`, a dot chart will be produced showing the proportions of various categories stratified by `drug`, and a separate dot chart is drawn for continuous variables. The latter chart shows by default the 3 quartiles of each variable, stratified by `drug`.

To obtain a comprehensive guide to `summary.formula` that includes many examples, graphical output, and LaTeX commands for putting an entire clinical report together, download the document entitled "Statistical Tables and Plots using S and LaTeX" from `hesweb1.med.virginia.edu/-biostat/s/doc/summary.pdf`. This document also contains graphical representations of may of the example tables. See `hesweb1.med.virginia.edu/biostat/s/LiveDoc.html` for useful related material.

6.2.1 Implementing Other Interfaces

The LaTeX output can be pasted into a word processed (e.g., Microsoft Word) document in graphics mode if you use PCTEX, with some loss of resolution. A more general solution would be to write S interface functions (e.g., `word`) that are analogous to the `latex` family of functions. Such functions would do the needed character string manipulation to write tables and other S output in Word format. It may be easier to implement an S to HTML interface, and Microsoft Word 97 can import HTML files and convert them to Word format. S-Plus 4.5 and later for Windows has a function `html.table` for producing simple HTML tables from S matrices. One other possibility is to convert the LaTeX code produced by S using a general convertor such as `Hevea` (see `http://www.-arch.ohio-state.edu/crp/faculty/pviton/support/hevea.html` or `hesweb1.med.virginia.-edu/biostat/s/EmacsTeX/index.html`). One problem with this approach is that HTML has some table making features that are not respected by Microsoft Word.

6.3 Graphical Depiction of Two–Way Contingency Tables

The Hmisc `symbol.freq` function can be used to represent contingency tables graphically. Frequency counts are represented as the heights of "thermometers" by default; you can also specify `symbol='circle'` to the function. There is an option to include marginal frequencies, which are plotted on a halved scale so as to not overwhelm the plot. Other useful options in this function include `orig.scale` (set to `T` when the first two arguments are numeric variables; this uses their original values for x and y coordinates), `subset` (the usual subsetting argument as used in regression fits), and `srtx` (a rotation angle for x–axis labels). If you do not ask for marginal frequencies to be plotted using `marginals=T`, `symbol.freq` will ask you to click the mouse where a reference symbol is to be drawn to assist in reading the scale of the frequencies. As an example consider

```
win.graph()        # or postscript(), etc.
attach(titanic)
age.tertile ← cut2(age, g=3)
symbol.freq(age.tertile, pclass, marginals=T, srtx=45)
```

The output is shown in Figure 6.1. See Section 6.1 for ways to display row or column proportions from contingency tables.

Another way to display frequency data is to use the built-in `image` function to plot the column values vs. the row values, with boxes whose density of shading is a function of the frequency of that cell. To display a two-dimensional histogram for two continuous variables in this way you can run the raw values through the `hist2d` function.

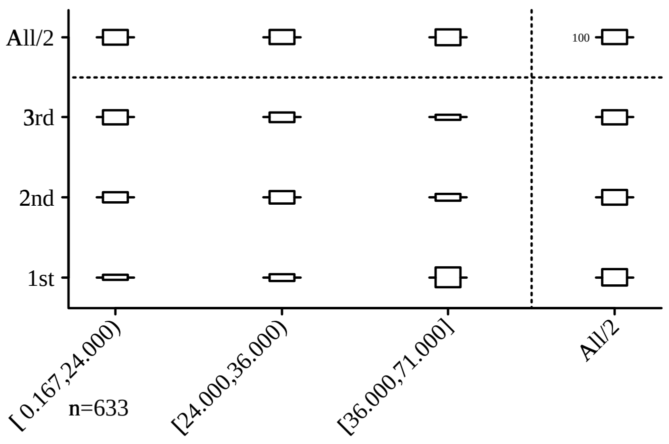

Figure 6.1: *A two–way contingency table*

Chapter 7

Hmisc Generalized Least Squares Modeling Functions

7.1 Automatically Transforming Predictor and Response Variables

Fitting multiple regression models by the method of least squares is one of the most commonly used methods in statistics. There are a number of challenges to the use of least squares, even when it is only used for estimation and not inference, for example:

1. How should continuous predictors be transformed so as to get a good fit?
2. Is it better to transform the response variable? How does one find a good transformation that simplifies the right hand side of the equation?
3. What if Y needs to be transformed non–monotonicially (e.g., $|Y - 100|$ or $(Y - 120)^2$) before it will have any correlation with X?

When one is trying to draw inference about population effects using confidence limits or hypothesis tests, the most common approach is to assume that the residuals have a normal distribution. This is equivalent to assuming that the conditional distribution of the response Y given the set of predictors X is normal with mean depending on X and variance that is (hopefully) a constant independent of X. The need for a distributional assumption to enable us to draw inferences creates a number of other challenges, including:

1. If for the untransformed original scale of the response Y the distribution of the residuals is not normal with constant spread, ordinary methods will not yield correct inferences (e.g., confidence intervals will not have the desired coverage probability and the intervals will need to be asymmetric).

2. Quite often there is a transformation of Y that will yield well–behaving residuals. How do you find this transformation? Can you find transformation for the Xs at the same time?
3. All classical statistical inferential methods assume that the full model was pre–specified, i.e., the model was not modified after examining the data. How does one correct confidence limits, for example, for data–based model selection?

On the last point, Faraway [15] demonstrated that the more stops done by the analyst, such as looking for transformations, outliers, overly influential observations, and stepwise variable selection, the more the variance of estimates increases. This assumes of course that one properly estimates the variances (e.g., using a simulation technique such as the bootstrap); *apparent* variances will typically *decrease* as the model is refined. Faraway showed that the greatest source of inflation of actual variances is letting the data dictate the transformation of Y. He concluded that since we currently have no statistical theory for deriving proper variance estimates, it is preferable to automate the analysis and to use the bootstrap to estimate variances and construct confidence limits, taking into account all sources of variability induced by the modeling strategy.

S has a powerful function, `gam`, for fitting generalized additive regression models. `gam` automatically estimates the transformation each right hand side variable should receive so as to optimize prediction of Y, and a number of distributions are allowed for Y. When one hopes to assume normality and the left hand side of the model also needs transformation, either to improve R^2 or to achieve constant variance of the residuals (which increases the chances of satisfying the normality assumption), S has two powerful nonparametric regression functions: `ace` and `avas`. Both functions allow categorical predictors, allow predictor transformations to be non–monotonic, and allow the analyst to restrict the transformations to be monotonic. `ace` stands for "alternating conditional expectation" [16], an algorithm directed solely at finding transformations for all variables simultaneously so as to optimize R^2. `ace` will allow Y to be non–monotonically transformed, and it is based on the "super smoother" (see the `supsmu` function). `avas` stands for "additivity and variance stabilization" [17]. `avas` tries to maximize R^2 while forcing the transformation for Y to result in nearly constant variance of residuals. `avas` restricts the transformation of Y to be monotonic.

`ace` and `avas` are quite powerful, but they can result in overfitting, and they provide no statistical inferential measures. In addition, they do not use the S modeling language, so they are slightly more difficult to use. The Hmisc `areg.boot` ("additive regression using the bootstrap") solves these problems. The bootstrap is used to estimate the optimism (bias) in the apparent R^2, and this optimism is subtracted from the apparent R^2 to get a more trustworthy estimate. The online help file has the details.

Note: `areg.boot` has been extended to allow one to estimate any quantity of interest, such as the mean response, on the original scale, using Duan's smearing estimator. The output below is from the previous version of `areg.boot` which did not include this facility. See Chapter 15 of Harrell's book Regression Modeling Strategies for an updated example.

As an example consider an excellent dataset provided by Dr. John Schorling, Department of Medicine, University of Virginia School of Medicine. The data consist of 19 variables on 403 subjects from 1046 subjects who were interviewed in a study to understand the prevalence of obesity and diabetes in central Virginia for African Americans. According to Dr. John Hong, Diabetes Mellitus Type II (adult onset diabetes) is associated most strongly with obesity. The waist/hip ratio may be a predictor of diabetes and heart disease. DM II is also associated with hypertension - they may both be part of "Syndrome X". The 403 subjects were the ones who were actually screened for diabetes. Glycosolated hemoglobin > 7.0 is usually taken as a positive diagnosis of diabetes.

At first glance some analysts might think that the best way to develop a model for diagnosing diabetes might be to fit a binary logistic model with glycosolated hemoglobin > 7 as the response variable. This is very wasteful of information, as it does not distinguish a hemoglobin value of 2 from a 6.9, or a 7.1 from a 10. The waste of information will result larger standard errors of $\hat{\beta}$, wider confidence bands, larger P–values, and lower power to detect risk factors. A better approach is to predict the continuous hemoglobin value using a continuous response model such as ordinary multiple regression or using ordinal logistic regression. Then this model can be converted to predict the probability that hemoglobin exceeds any cutoff of interest. For an ordinal logistic model having one intercept per possible value of hemoglobin in the dataset (except for the lowest value), all probabilities are easy to compute. For ordinary regression this probability depends on the distribution of the residuals from the model.

Let us proceed with a least squares approach. An initial series of trial transformations for the response indicated that the reciprocal of glycosolated hemoglobin resulted in a model having residuals of nearly constant spread when plotted against predicted values. In addition, the residuals appeared well approximated by a normal distribution. On the other hand, a model developed on the original scale did not have constant spread of the residuals. It will be interesting to see if the nonparametric variance stabilizing function determined by `avas` will resemble the reciprocal of hemoglobin.

Let's consider the following predictors: age, systolic blood pressure, total cholesterol, body frame (small, medium, large), weight, and hip circumference. 12 subjects have missing body frame, and we should be able to impute this variable from other body size measurements. Let's do this using recursive partitioning with Atkinson and Therneau's `rpart` function. See the UVa Web page for a link to obtain the `rpart` library. The advantage of `rpart` over the builtin `tree` function is that `rpart` can handle missing predictor variables using "surrogate splits." In other words, when a predictor needed for classifying an observation is missing, other predictors that are not missing can be used as stand–ins. `rpart` will predict the probability that the polytomous response `frame` equals each of its three levels.

```
> library(rpart)
> r ← rpart(frame ∼ gender + height + weight + waist + hip)
> plot(r); text(r)  # shows first split on waist, then height,weight
> probs ← predict(r, diabetes)
> # Within each row of probs order from largest to smallest
> # Find column # of largest
> most.probable.category ← (t(apply(-probs, 1, order)))[,1]
> frame.pred ← levels(frame)[most.probable.category]
> table(frame, frame.pred)

       large medium small
 small     2     45    57
medium    10    158    16
 large    35     67     1

> frame ← impute(frame, frame.pred[is.na(frame)])
> describe(frame)
frame : Body Frame
   n missing imputed unique
 403       0      12      3
```

```
small (106, 26%), medium (193, 48%), large (104, 26%)

> table(frame[is.imputed(frame)])
 small medium large
     2      9     1
```

Other predictors are only missing on a handful of cases. Impute them with constants to avoid excluding any observations from the fit.

```
> bp.1s  ← impute(bp.1s)
> chol   ← impute(chol)
> weight ← impute(weight)
> hip    ← impute(hip)
```

Now fit the `avas` model. Do only 30 bootstrap repetitions so we can clearly see how the bootstrap re–estimates of transformations vary on the next plot. Use subject matter knowledge to restrict the transformations of `age`, `weight`, and `hip` to be monotonic. Had we wanted to restrict transformations to be linear, we would have specified the identity function, e.g., `I(weight)`.

```
> f ← areg.boot(glyhb ~ monotone(age) + bp.1s + chol + frame +
+                monotone(weight) + monotone(hip), B=30)
> options(digits=3)
> f

avas Additive Regression Model

areg.boot(x = glyhb ~ monotone(age) + bp.1s + chol + frame +
          monotone(weight) + monotone(hip), B = 30)

Categorical variables: frame
```

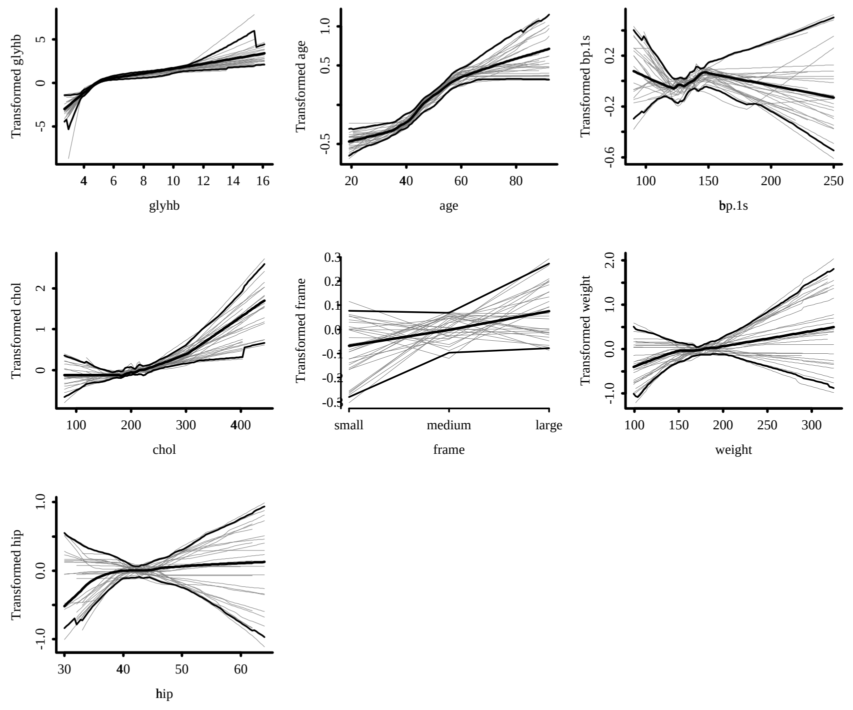

Figure 7.1: *avas transformations: overall estimates, pointwise 0.95 confidence bands, and 30 bootstrap estimates.*

```
Frequencies of Missing Values Due to Each Variable
 glyhb monotone(age) bp.1s chol frame monotone(weight) monotone(hip)
    13             0     0    0     0                0             0

n= 390   p= 6

Apparent R2 on transformed Y scale: 0.265
Bootstrap validated R2            : 0.207

Coefficients of standardized transformations:

  Intercept  age bp.1s  chol frame weight   hip
 -4.34e-009 1.06  1.51 0.953 0.708   1.26 0.653
```

Note that the coefficients above do not mean very much as the scale of the transformations is arbitrary. We see that the model was overfit a moderate amount (optimism in R^2 is 0.265 - 0.207).

Next we plot the transformations (bold lines in the center), pointwise 0.95 confidence bands (shown with bold lines), and bootstrap estimates (smaller lines).

```
> plot(f, col.boot=.75)  # use grayscale instead of color for bootstraps
```

The plot is shown in Figure 7.1. Apparently, `age` and `chol` are the important predictor.

Let's see how effective the transformation of `glyhb` was in stabilizing variance and making the residuals normally distributed.

```
> par(mfrow=c(2,2))
> plot(fitted(f), resid(f))
> plot(predict(f), resid(f))
> qqnorm(resid(f)); abline(a=0, b=1) # draws line of identity
```

We see from Figure 7.2 that the residuals have reasonably uniform spread and are distributed almost normally. A multiple regression run on untransformed variables did not fare nearly as well.

Now check whether the response transformation is close to the reciprocal of `glyhb`. First derive an S representation of the fitted transformations. For nonparametric function estimates these are really table lookups. `Function` creates a `list` of functions, named according to the variables in the model.

```
> funs ← Function(f)
> plot(1/glyhb, funs$glyhb(glyhb))
```

Results are in Figure 7.3. An almost linear relationship is evidence that the reciprocal is a good transformation[1].

Now let's get approximate tests of effects of each predictor. `summary` does this by setting all other predictors to reference values (e.g., medians), and comparing predicted untransformed responses for a given level of the predictor with predictions for the lowest setting of X. We will use the three quartiles for continuous variables, but specify `age` settings manually.

[1]Beware that it may not help to know this, because if we re–do the analysis using an ordinary linear model on `1/glyhb`, standard errors would not take model selection into account[15].

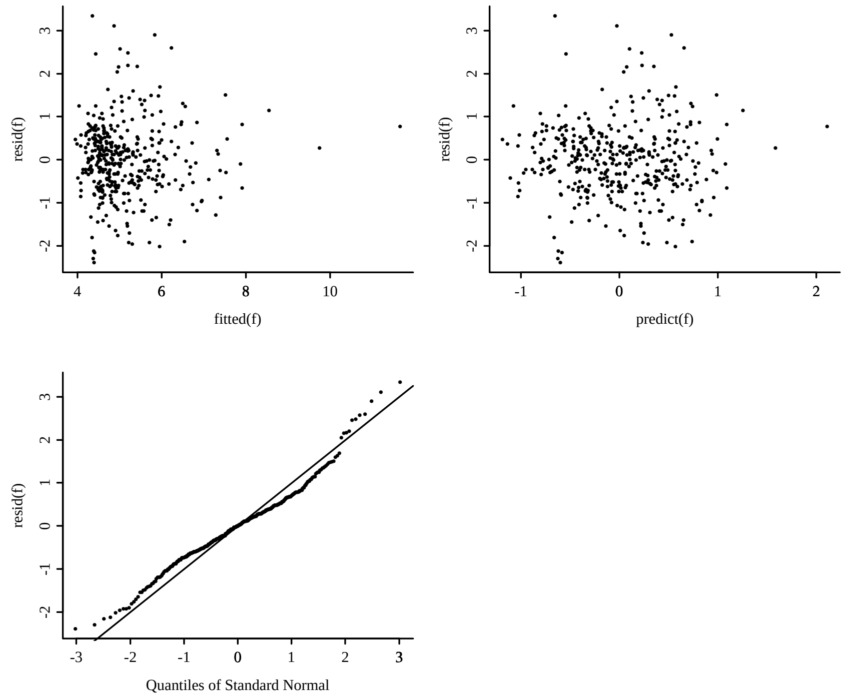

Figure 7.2: *Distribution of residuals from the* `avas` *fit. The top left panel x–axis has $\hat{Y}$ on the original Y scale. The top right panel uses the transformed $\hat{Y}$ for the x–axis.*

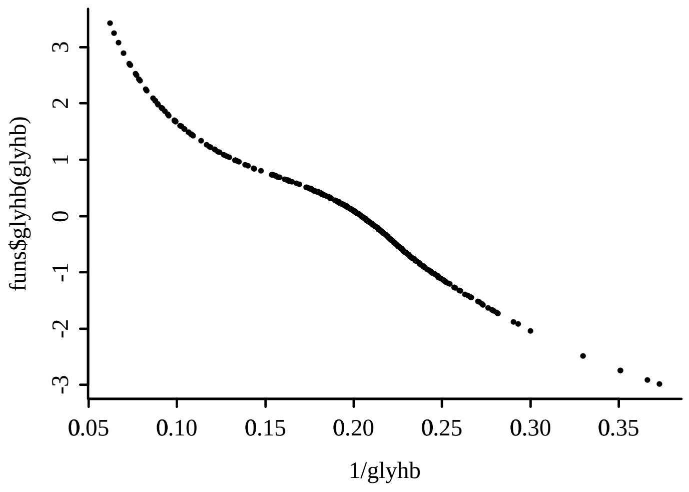

Figure 7.3: *Agreement between the* `avas` *transformation for* `glyhb` *and the reciprocal of* `glyhb`*.*

```
> summary(f, values=list(age=c(20,30,40,50,60,70,80)))

Values to which predictors are set when estimating
effects of other predictors:

 glyhb age bp.1s chol frame weight hip
  4.84  50   136  204     2    173  42

Estimates of differences of effects on Y (from first X value),
and bootstrap standard errors of these differences.
Settings for X are shown as row headings.


Predictor: age
   Differences    S.E Lower 0.95 Upper 0.95    Z   Pr(|Z|)
20       0.000     NA         NA         NA   NA        NA
30       0.064 0.0434    -0.0210      0.149 1.48 1.40e-001
40       0.184 0.0770     0.0326      0.335 2.38 1.72e-002
50       0.527 0.1084     0.3149      0.740 4.87 1.14e-006
60       0.868 0.1551     0.5645      1.172 5.60 2.14e-008
70       1.122 0.2311     0.6691      1.575 4.86 1.20e-006
80       1.428 0.4591     0.5278      2.327 3.11 1.87e-003


Predictor: bp.1s
    Differences    S.E Lower 0.95 Upper 0.95    Z Pr(|Z|)
122      0.0000     NA         NA         NA   NA      NA
136      0.0863 0.0715    -0.0540      0.227 1.21  0.2279
148      0.2275 0.1303    -0.0278      0.483 1.75  0.0807


Predictor: chol
    Differences    S.E Lower 0.95 Upper 0.95    Z Pr(|Z|)
179      0.0000     NA         NA         NA   NA      NA
204      0.0715 0.0606    -0.0473      0.190 1.18  0.2381
229      0.1912 0.1111    -0.0265      0.409 1.72  0.0852


Predictor: frame
       Differences   S.E Lower 0.95 Upper 0.95     Z Pr(|Z|)
 small      0.0000    NA         NA         NA    NA      NA
medium      0.0514 0.119     -0.182      0.285 0.433   0.665
 large      0.1141 0.176     -0.231      0.459 0.648   0.517


Predictor: weight
    Differences   S.E Lower 0.95 Upper 0.95     Z Pr(|Z|)
150      0.0000    NA         NA         NA    NA      NA
173      0.0219 0.109     -0.192      0.236 0.201   0.841
200      0.1576 0.245     -0.322      0.637 0.644   0.520
```

```
Predictor: hip
   Differences   S.E Lower 0.95 Upper 0.95      Z Pr(|Z|)
39      0.0000    NA         NA         NA     NA      NA
42      0.0097 0.102     -0.189      0.209 0.0955   0.924
46      0.0299 0.200     -0.362      0.422 0.1496   0.881

Warning messages:
  For 5 bootstrap samples a predicted value for one of the settings for age
could not be computed.  These bootstrap samples ignored.
Consider using less extreme predictor settings.
 in: summary.areg.boot(f, values = list(age = c(20, 30, 40, 50, 60, 70, 80)))
```

For example, when `age` increases from 20 to 70 we predict an increast in `glyhb` by 1.122 with standard error 0.2311, when all other predictors are help to constants listed above. Setting them to other constants will yield different estimates of the `age` effect, as the transformation of `glyhb` is nonlinear. We see that only for `age` do some of the confidence intervals for effects exclude zero.

Let's depict the fitted model by plotting predicted values, with `age` varying on the x–axis, and 3 curves corresponding to three values of `chol`. Set all other predictors to representative values.

```
> newdat ← expand.grid(age=20:80, chol=quantile(chol,c(.25,.5,.75)),
+                          bp.1s=136, frame='medium', weight=173, hip=42)
> yhat ← predict(f, newdat, type='fitted')
> xYplot(yhat ∼ age, groups=chol, data=newdat,
+       type='l', col=1,
+       ylab='Glycosolated Hemoglobin', label.curve=list(method='on top'))
```

The result is Figure 7.4. Note that none of the predictions is above 7.0. Let's see how many predictions in the entire dataset are above 7.0.

```
> yhat.all ← predict(f, type='fitted')
> # length of yhat.all is 390 because 13 obs were dropped due to NAs
> sum(yhat.all > 7)
[1] 15
```

So the model is not very useful for finding clinical levels of diabetes. Let's make sure that a dedicated binary model would not do any better.

```
> library(Design,T)
> h ← lrm(glyhb > 7 ∼ rcs(age,4) + rcs(bp.1s,3) + rcs(chol,3) +
+         frame + rcs(weight,4) + rcs(hip,3))
> h
```

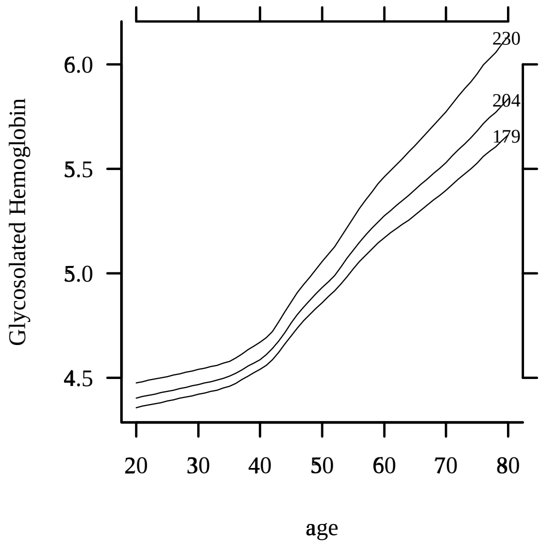

Figure 7.4: *Predicted median* `glyhb` *as a function of* `age` *and* `chol`*.*

```
Logistic Regression Model

lrm(formula = glyhb > 7 ~ rcs(age, 4) + rcs(bp.1s, 3) +
    rcs(chol, 3) + frame + rcs(weight, 4) + rcs(hip, 3))


Frequencies of Responses
 FALSE TRUE
   330   60

Frequencies of Missing Values Due to Each Variable
 glyhb > 7 age bp.1s chol frame weight hip
        13   0     0    0     0      0   0

 Obs Max Deriv Model L.R. d.f. P     C   Dxy Gamma Tau-a   R2 Brier
 390    1e-007       71.3   14 0 0.819 0.637 0.639 0.166 0.29 0.105

                  Coef    S.E. Wald Z      P
   Intercept -16.804027 6.40143 -2.63  0.0087
         age   0.023219 0.08806  0.26  0.7920
        age'   0.266699 0.25501  1.05  0.2956
       age''  -0.852166 0.63708 -1.34  0.1810
       bp.1s   0.028259 0.02476  1.14  0.2537
      bp.1s'  -0.025207 0.02404 -1.05  0.2944
        chol   0.004649 0.01004  0.46  0.6432
       chol'   0.003535 0.01046  0.34  0.7354
frame=medium  -0.246480 0.48146 -0.51  0.6087
 frame=large  -0.266503 0.53384 -0.50  0.6176
```

```
     weight   0.042962 0.03073  1.40  0.1621
    weight'  -0.088281 0.09463 -0.93  0.3509
   weight''   0.264845 0.29109  0.91  0.3629
        hip   0.033904 0.12479  0.27  0.7859
       hip'  -0.053349 0.13979 -0.38  0.7027
```

```
> anova(h)
              Wald Statistics        Response: glyhb > 7

     Factor     Chi-Square d.f.      P
 age            23.85       3    0.0000
  Nonlinear      6.82       2    0.0331
 bp.1s           1.32       2    0.5178
  Nonlinear      1.10       1    0.2944
 chol            5.45       2    0.0657
  Nonlinear      0.11       1    0.7354
 frame           0.29       2    0.8630
 weight          5.41       3    0.1443
  Nonlinear      0.87       2    0.6457
 hip             0.19       2    0.9111
  Nonlinear      0.15       1    0.7027
 TOTAL NONLINEAR 10.48      7    0.1630
 TOTAL          45.65      14    0.0000
```

So far the results seem to be the same as using a continuous response. How many predicted probabilities of diabetes are in the “rule–in” range?

```
> p ← predict(h, type='fitted')
> sum(p > .9, na.rm=T)
[1] 0
```

Only one patient had a predicted probability > 0.8. So the risk factors are just not very strong, although `age` does explain some pre–clinical variation in `glyhb`.

7.2 Robust Serial Data Models: Time– and Dose–Response Profiles

Serial data (repeated measurements) are commonly encountered in biostatistical analysis. Specialized methods exist for fitted repeated measurements but it is advantageous to fit time– and dose–response data using a flexible parametric approach while allowing calculation of simultaneous (and pointwise) confidence limits for the true trends. The approach taken by Hmisc's `rm.boot` function is to use a “working independence” model allowing for intercepts to vary by subjects, and then to account for intra–subject correlations when deriving confidence bands. Regression splines restricted to be linear beyond the outer join points (knots) are used to fit the overall trend. Here all the serial data are analyzed in a common model with dummy variables used to absorb subject effects. Regression estimates which do not take the correlation structure into account are often quite efficient. Then a cluster bootstrap (sampling with replacement from *subjects* rather than data points) [18] is used to compute confidence bands in a nearly nonparametric fashion.

The text below, taken from the help file for `rm.boot`, describes the details. In what follows, "time" can be replaced with other variables such as the dose of a drug given multiple times to the same subjects.

For a dataset containing a time variable, a scalar response variable, and an optional subject identification variable, `rm.boot` obtains least squares estimates of the coefficients of a restricted cubic spline function or a linear regression in time after adjusting for subject effects through the use of subject dummy variables. Then the fit is bootstrapped `B` times, either by treating time and subject `id` as fixed (i.e., conditioning the analysis on them) or as random variables. For the former, the residuals from the original model fit are used as the basis of the bootstrap distribution. For the latter, samples are taken jointly from the time, subject `id`, and response vectors to obtain unconditional distributions.

If a subject `id` variable is given, the bootstrap sampling will be based on samples with replacement from subjects rather than from individual data points. In other words, either none or all of a given subject's data will appear in a bootstrap sample. This cluster sampling takes into account any correlation structure that might exist within subjects, so that confidence limits are nonparametrically corrected for within-subject correlation. Assuming that ordinary least squares estimates, which ignore the correlation structure, are consistent (which is almost always true) and efficient (which would not be true for certain correlation structures or for datasets in which the number of observation times vary greatly from subject to subject), the resulting analysis will be a robust, efficient repeated measures analysis for the one-sample problem.

Predicted values of the fitted models are evaluated by default at a grid of 100 equally spaced time points ranging from the minimum to maximum observed time points. Predictions are for the average subject effect. Pointwise confidence intervals are optionally computed separately for each of the points on the time grid. However, simultaneous confidence regions that control the level of confidence for the entire regression curve lying within a band are often more appropriate, as they allow the analyst to draw conclusions about nuances in the mean time response profile that were not stated apriori. The method of Tibshirani and Knight [19] is used to easily obtain simultaneous confidence sets for the set of coefficients of the spline or linear regression function as well as the average intercept parameter (over subjects). Here one computes the objective criterion (here both the -2 log likelihood evaluated at the bootstrap estimate of beta but with respect to the original design matrix and response vector, and the sum of squared errors in predicting the original response vector) for the original fit as well as for all of the bootstrap fits. The confidence set of the regression coefficients is the set of all coefficients that are associated with objective function values that are less than or equal to say the 0.95 quantile of the vector of `B + 1` objective function values. For the coefficients satisfying this condition, predicted curves are computed at the time grid, and minima and maxima of these curves are computed separately at each time point to derive the final simultaneous confidence band.

By default, the log likelihoods that are computed for obtaining the simultaneous confidence band assume independence within subject. This will cause problems unless such log likelihoods have very high rank correlation with the log likelihood allowing for dependence. To allow for correlation or to estimate the correlation function, see the `cor.pattern` and `rho` arguments to `rm.boot`.

As most repeated measurement studies consider the times as design points, the fixed covariable case is the default. Bootstrapping the residuals from the initial fit assumes that the model is correctly specified. Even if the covariables are fixed, doing an unconditional bootstrap is still appropriate, and for moderate to large sample sizes unconditional confidence intervals are only slightly wider than conditional ones if subject effects (intercepts) are small. For `bootstrap.type="x random"` in the presence of significant subject effects, the analysis is approximate as the subjects used in any one bootstrap fit will not be the entire list of subjects. The average (over subjects used in the bootstrap sample) intercept is used from that bootstrap sample as a predictor of average subject effects in the overall sample.

rm.boot can handle two–sample problems in which trends are fitted separately within each of two groups and then the differences in the trends (and bootstrap confidence bands for these) are computed to measure the group effect.

7.2.1 Example

The following example demonstrates how correlated response data may be simulated and then analyzed using rm.boot. We simulate data for 20 subjects each with 11 response measurements. The population response function is piecewise linear (flat in the left and right tails) and large true subject effects are present.

```
store()
# Don't keep any of the objects created (store is in Hmisc)

# Function to generate n p-variate normal variates with
# mean vector u and covariance matrix S
# Slight modification of function written by Bill Venables

mvrnorm ← function(n, p = 1, u = rep(0, p), S = diag(p)) {
  Z ← matrix(rnorm(n * p), p, n)
  t(u + t(chol(S)) %*% Z)
}

# Simulate serial data

n     ← 20        # Number of subjects
sub   ← .5*(1:n)  # Subject effects

# Specify functional form for time trend and compute
# non-stochastic component
times ← seq(0, 1, by=.1)
g     ← function(times) 5*pmax(abs(times-.5),.3)
ey    ← g(times)

# Generate multivariate normal errors for 20 subjects at 11 times
# Assume equal correlations of rho=.7, independent subjects

nt    ← length(times)
rho   ← .7

set.seed(19)
errors ← mvrnorm(n, p=nt, S=diag(rep(1-rho,nt))+rho)
# Note:  first random number seed used gave rise to
# mean(errors)=0.24!
```

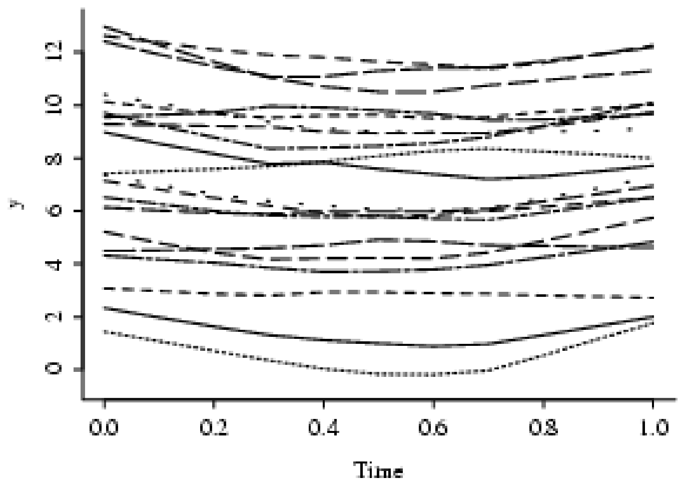

Figure 7.5: *Nonparametric estimates of time trends for individual subjects*

```
# Add E[Y], error components, and subject effects
y      ← matrix(rep(ey,n), ncol=nt, byrow=T) + errors +
           matrix(rep(sub,nt), ncol=nt)

# String out data into long vectors for times, responses,
# and subject ID
y      ← as.vector(t(y))
times  ← rep(times, n)
id     ← sort(rep(1:n, nt))

# Do 400 bootstrap repetitions, sampling from residuals (grouped by
# subjects) rather than from the design matrix and responses for
# subjects

f ← rm.boot(times, y, id, plot.individual=T, B=400,
             smoother=lowess, bootstrap.type='x fixed', nk=6)
```

To compute a dependent–structure log–likelihood in addition to one assuming independence, add e.g. the argument `cor.pattern='estimate'` or `rho=.5`.

`plot.individual=T, smoother=lowess` causes nonparametric estimates of trends for individual subjects to be plotted on a single plot. The output from this object is shown in Figure 7.5.

Next we plot a random sample of 75 of the 400 bootstrap fits of the time trends. These fits use as intercepts the average intercept over subjects.

```
plot(f, individual.boot=T, ncurves=75, ylim=c(6,8.5))
```

The plot is in Figure 7.6.

Finally the main plot of interest is shown in Figure 7.7. Both simultaneous and pointwise confidence regions are shown.

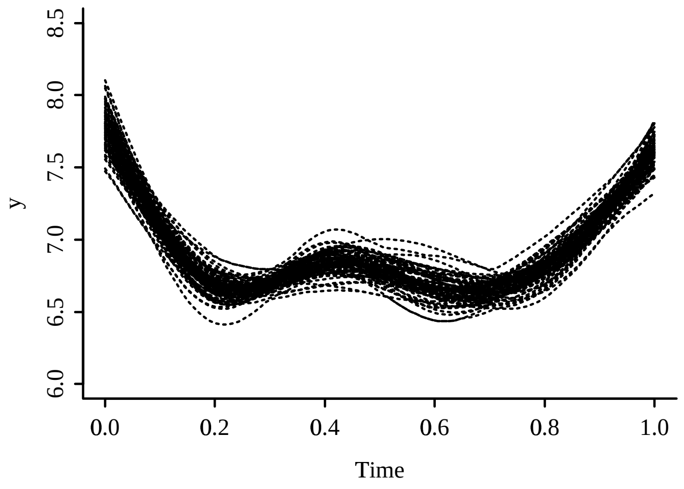

Figure 7.6: *75 of the 400 bootstrap estimates of the average time trend over subjects.*

```
plot(f, pointwise.band=T, col.pointwise=1, ylim=c(6,8.5))

# Plot population response curve at average subject effect
ts ← seq(0, 1, length=100)
lines(ts, g(ts)+mean(sub), lwd=3)
```

`rm.boot` assumes that any missing measurements are missing completely at random. The Design `rm.impute` function can analyze non–randomly missing serial data using multiple imputation, assuming that the probability that a measurement is missing is a function only of baseline variables and of previous measurements.

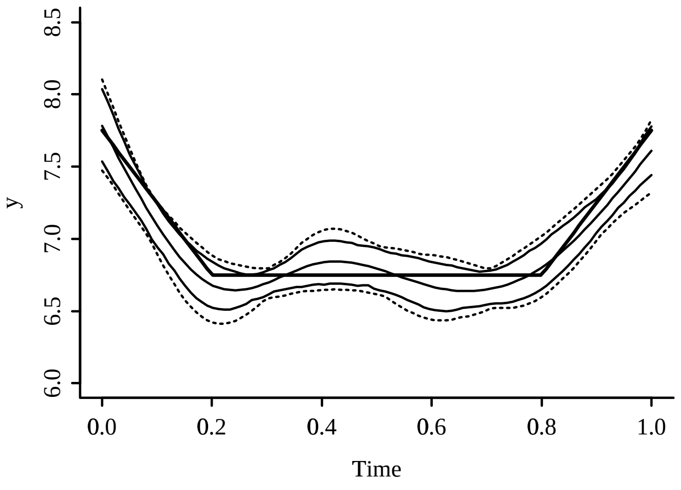

Figure 7.7: *Simultaneous (dotted outer curves) and pointwise (solid curves) 0.95 confidence regions for the average time trend. The plot also has the overall fitted time trend as the solid curve in the middle, and the true piecewise linear population time trend for the true average subject effect. The confidence intervals assume that a restricted cubic spline function with 6 knots contains the population profile as a special case, which is not exactly true.*

Chapter 8

Builtin S Functions for Multiple Linear Regression

`lm` is the builtin function for fitting multiple linear regression models, and it works with several other functions to summarize results, make hypothesis tests, get predicted values, and display model diagnostics. Suppose that the response variable is named `y` and the predictors are `x1, x2, x3`. The following examples show how to use the basic functions. Note that the *fit object* below (`f`) is a list containing several components such as `coefficients` and `fitted.values`.

```
# Use the following command to cause dummy variables to be created
# the conventional way from categorical predictors
options(contrasts=c('contr.treatment','contr.poly'))
# Fitting functions in the Design library make this the default

f ← lm(y ∼ x1 + x2 + x3, data=dframe, na.action=na.omit)
# Attach dframe if you don't use data=, omit both if using
# standalone variables
# Omit na.action= if there are no NAs in the variables in the model
# na.omit causes any observations containing NAs to be deleted
# before fitting

f           # or print(f): prints coefficients and sigma hat
summary(f)  # prints crude residual diagnostics, coefficients,
            # s.e., t statistics, P-values, sigma hat, overall F and P,
            # R^2, correlations of coefficients

plot(f)     # Draws 6 graphs
# Plots residuals vs. fitted values (with 3 most extreme
# points identified),
# sqrt(abs(residuals)) vs. yhat (for identifying outliers)
# y vs. yhat
# normal quantile plot of residuals (to check for normality)
# a residual-fit spread plot (r-f plot) to compare the spread of the
#   fitted values with the spread of residuals (which should be less)
# Cook's distance plot to look for overly influential observations
plot(f, smooths=T, rugplot=T)
# adds trend lines and x data density ticks

coef(f)     # or coefficients(f) or f$coefficients: get coef.
fitted(f)   # or fitted.values(f) or predict(f) or
            # f$fitted.values: computes yhat
resid(f)    # or residuals(f) or f$residuals: computes residuals
plot(x2, resid(f))   # plot residuals vs. x2 alone

predict(f, se.fit=T)
# original hats and se for E(y|x)

predict(f, data.frame(x1=1,x2=2,x3=17))
# yhat for user-given x's

predict(f, expand.grid(x1=1,x2=2:3,x3=1:10))
# yhat for 20 combinations of x's

# Use e.g. sex=factor('female',levels=...)  to specify settings of
# categorical predictors

drop1(f)    # compute SSR due to each variable by
            # dropping one at a time
aov(f)      # sums of squares and d.f.
anova(f)    # anova table with sums of squares computed by
            # sequentially adding predictors (in order in formula),
            # F, P-values

f2 ← lm(y ∼ x3)     # sub-model
anova(f2, f)         # partial F test for x1+x2 combined | x3 plus
                     # sequentially added sums of squares
# To get partial F tests for all variables, you must leave out each
# at a time

# Without controlling x2 and x3, plot yhat vs. observed x1 with
# pointwise 0.99 CI
pred ← predict(f, se.fit=T)
ci   ← pointwise(pred, coverage=0.99)
plot(x1, fitted(f))
points(x1, ci$upper, pch=2)
points(x1, ci$lower, pch=2)

# Better:
# Let x1 vary over a grid of 100 equally spaced points and set
# x2 and x3 to their means.  Get predicted values and s.e., then
# pass predictions through pointwise() to get pointwise CI to plot
x1s  ← seq(min(x1),max(x1), length=100)
pred ← predict(f, expand.grid(x1=x1s, x2=mean(x2), x3=mean(x3)),
                se.fit=T)
pred$fit      # print yhat
pred$se.fit   # print estimated se of yhat
ci ← pointwise(pred, coverage=.95)
ci$upper      # print upper CL
ci$lower      # print lower CL
plot(x1s,  pred$fit, type='l', ylab='Yhat')
lines(x1s, ci$upper, lty=2)    # dotted line
lines(x1s, ci$lower, lty=2)

# Add confidence bands for predicting individual y's
# pred$residual.scale^2 is MSE, and this is for the '1'
# in the se(E(y|x)) formula
pred$se.fit ← sqrt(pred$se.fit^2 + pred$residual.scale^2)
cii ← pointwise(pred, coverage=.95)
lines(x1s, cii$lower, lty=2)
lines(x1s, cii$upper, lty=2)

# An example where we get predictions by letting two predictors
# vary, and we plot two sets of confidence bands
ages   ← seq(3,16,length=100)
combos ← expand.grid(sex=factor(levels(sex),levels(sex)), age=ages)
pred   ← predict(fit, combos, se.fit=T)
ci     ← pointwise(pred, coverage=.99)
par(mfrow=c(1,2))
for(sx in levels(sex)) {
   s ← combos$sex==sx
   plot(combos$age[s], ci$fit[s], xlab='Age', ylab='Y hat',
        ylim=range(unlist(ci)), type='l')
   # range(unlist(ci)) takes range over all of fit, upper, lower
   title(sx)
   lines(combos$age[s], ci$upper[s], lty=2)
   lines(combos$age[s], ci$lower[s], lty=2)
}
```

To obtain partial residual plots you can use the `Statistics .. Regression .. Linear` menu in version 4.0 and later.

Warning: When the response or any of the predictor variables contain `NA`s, `na.action=na.omit` will cause `lm` to delete observations containing `NA`s, but unfortunately the `fitted`, `resid`, and `predict` (when no data frame argument is given) functions compute $\hat{y}$ or residuals only for the observations actually used in the fit. In other words, the results of these functions will be vectors that are shorter than the original variables used in the fit, and the observations will no longer align. A command such as `plot(x1, resid(fit))` will fail. One solution to the problem is to `attach` only the subset of the data frame that corresponds to observations not containing `NA`s on variables used in modeling. This approach does not work well when different sets of variables are to be used in different models.

The survival analysis modeling functions builtin to S solve this problem in an elegant way. Regression fitting functions (such as `ols`) in the `Design` library use this same solution. These fitting functions set up so that `resid` and related functions add `NA`s back to predicted values or residuals so that they are aligned with the original data.

Warning: When the `lm` model contains a categorical (`factor`) predictor, you must give `predict` a data frame that has exactly the same factor levels for such predictors as appeared in the original variable given to `lm`. For example, specifying `predict(fit, data.frame(age=10, sex='male'))` can result in incorrect predictions, as the temporary `sex` variable contains only one level, and `predict` for `lm` does not know how to construct the dummy variables correctly. Instead, specify for example

```
predict(fit, expand.grid(age=c(10,20,30),
        sex=factor('male',c('female','male'))))
```

if the original `sex` variable had levels `c('female','male')` *in that order.* A more automatic approach is to specify `sex=factor('male', levels(sex))` in the previous command. To obtain predictions for all values of a categorical predictor, use for example `sex=factor(c('male','female'),levels(sex))` or `sex=factor(levels(sex),levels(sex))` as in one of the examples above.

All fitting functions in the `Design` library solve this problem by looking up the original `levels` for all predictors. No other S functions handle this automatically.

When using `lm` instead of `Design`'s `ols` function, you may want to put the `contrasts` option in your `.First` function, e.g.:

```
.First ← function() {
  library(Hmisc,T)
  options(contrasts=c('contr.treatment','contr.poly'))
  }
```

8.1 Sequential and Partial Sums of Squares and F–tests

Sequential sums of squares (called Type I SS in SAS) are increments in SSR's as predictors are added to a model[1]. Sequential SS can be quite arbitrary, because the SS for all predictors depend on the order that predictors were listed in the model formula. Sequential F–statistics are defined as sequential mean squares divided by the MSE from the full model. These statistics test the hypothesis that the current predictor is associated with the response after adjusted for the list of predictors *that preceeded it.* In other words, a sequential F–test tests whether the current predictor adds predictive information to those listed before it. Only the last predictor's sequential SS is adjusted for all of the other predictors. The total of all the sequential SS equals the SSR for the entire model.

In S, sequential F–tests are obtained by the command `anova(fitobject)`.

Partial sums of squares (called Type II, III, or IV SS in SAS[2]) are increments in SSR's when each predictor is added to *all* of the other predictors. Partial F–statistics are partial mean squares divided by the MSE. A partial test tests whether the current predictor is associated with the response after adjustment for *all* other predictors. In other words, the partial test assesses whether

[1]Note that increments in SSR are decrements in SSE. SSE is called RSS (residual sums of squares) in Rosner.

[2]These three types are identical when there are no interactions involving the predictors being tested

the predictor adds information to all of the other predictors. The sequential SS for the last predictor in a model equals its partial SS. The total of all partial SS does not mean anything.

Recall that the regression coefficients are also called partial regression coefficients, and that all of the t–statistics that are printed with the model fit (by e.g., `summary(fit)`) are test statistics for testing partial effects. When a predictor has only one degree of freedom associated with it (i.e., it is represented by one regression coefficient), its partial F–statistic is the square of the t–statistic (obtained by dividing the coefficient estimate by its estimated standard error).

Partial F–tests for multiple degree of freedom predictors are obtained in S by fitting a sub–model in which the predictor of interest is deleted, and then issuing a command such as `anova(fit.submodel,fit.full)`. The difference in SSR's for the full and reduced models the partial SS for the omitted predictor. The partial test thus assesses how much predictive information is lost by deleting that predictor.

Consider the following table of sequential and partial SS for a model containing predictors `age`, `sex`, and `exposure` (in that order):

Predictor	Sequential SS	Partial SS
Age	1000	755
Sex	300	100
Exposure	5	5
Total	1305	

As `exposure` is listed last, its sequential SS equals its partial SS. If the order of variables were to be reversed, we might see the following table:

Predictor	Sequential SS	Partial SS
Exposure	150	5
Sex	400	100
Age	755	755
Total	1305	

Now we see that if `age` and `sex` are not adjusted for, `exposure` explains more of the variation in the response. In contrast, `exposure` adds only 5 to SSR once `age` and `sex` are held constant. `sex` adds 100 more to SSR when only `exposure` is adjusted for, compared to when only `age` is adjusted for.

The easiest way to get partial F–tests and P–values for predictors that have one parameter associated with them is to use the partial t–tests that are printed by the S `summary` command (you can square t to get F). The easiest way to partial SS and F–tests in general is to run the `anova` function on a model that has the variable of interest as the last variable in the model, e.g.:

```
anova(lm(y ~ age + sex + cholesterol))
```

This will give an unadjusted test for `age`, a fully adjusted (partial) test for `cholesterol`, and a test for `sex` that is only adjusted for `age`. S-Plus 4.5 and later has extended the `anova` function to compute all partial tests using Type III sums of squares[3]. To obtain these partial tests, use the command `anova(fit,ssType=3)`. Type III tests are problematic, however, when interactions are present in the model — just the situation where Type III F–tests were originally intended to have advantages. For example, in a multi–center randomized drug trial for which treatment $\times$ center interactions are included in the model, Type III tests for the "average" drug effect weight

[3]The `anova` command for the `Design` library prints all partial F or χ^2 tests automatically.

centers contributing very few patients the same as large centers. The weighted mean (over centers) treatment effect associated with the Type III test is strange indeed, as it is a simple unweighted average of center–specific treatment effects and thus has lower precision.

For pre–4.5 versions of S-Plus the shortest command for obtaining a general pooled partial F–test is

```
anova(lm(y ~ subset of variables),
      lm(y ~ full set of variables))
```

where the subset of variables is the set of variables aside from the ones being tested. For the `Design` library, you can just list the variables you want to combine in an `anova` command.

Sometimes the order of variables can result in meaningful sequential SS for all variables. For example, one might list patient measurements in the order of the cost of making the measurements. Then each sequential test assesses how much the current measurement adds to those that are less expensive.

The last k variables in a model may be tested jointly using the sequential SS output of `anova` for `lm` fits, because sequential SS are additive. Suppose that the last three variables were to be tested as a group, and that these variables had a total of 5 parameters. Then the partial F–test with 5 and $n - p - 1$ d.f. is the ratio of the MSR corresponding to these 3 variables to the MSE for the full model. The correct MSR is the sum of the last three sequential SS's divided by 5.

Chapter 9

The Design Library of Modeling Functions

9.1 Statistical Formulas in S

Let us first summarize many of S's general modeling capabilities. S has a battery of functions which make up a statistical modeling language [2]. At the heart of the modeling functions is an *S formula* of the form

```
response ~ terms
```

The terms represent components of a general linear model. Although variables and functions of variables make up the terms, the formula refers to additive combinations, e.g. when terms is `age + blood.pressure`, it refers to $\beta_1 \times$ `age` $+\beta_2 \times$ `blood.pressure`. Some examples of the terms, which describe how predictor variables are modeled, are below:

```
y ~ age + sex            # age + sex main effects
y ~ age + sex + age:sex  # add second-order interaction
y ~ age*sex              # second-order interaction + all main effects
y ~ (age + sex + pressure)^2
                         # age+sex+pressure+age:sex+age:pressure...
y ~ (age + sex + pressure)^2 - sex:pressure
                         # all main effects and all 2nd order
                         # interactions except sex:pressure
y ~ (age + race)*sex      # age+race+sex+age:sex+race:sex
y ~ treatment*(age*race + age*sex) # no interact. with race,sex
sqrt(y) ~ sex*sqrt(age) + race
# functions, with dummy variables generated if
# race is an S factor (classification) variable
y ~ sex + poly(age,2)     # poly generates orthogonal polynomials
```

```
race.sex ← interaction(race,sex)
y ~ age + race.sex        # for when you want dummy variables for
                          # all combinations of the factors (why?)
```

The formula for a regression model is given to a modeling function, e.g.

```
lrm(y ~ rcs(x,4))
```

is read "use a logistic regression model to model y as a function of x, representing x by a restricted cubic spline with 4 default knots". `rcs` and `lrm` are part of Design.

You can use the S function `update` to re–fit a model with changes to the model terms or the data used to fit it:

```
f  ← lrm(y ~ rcs(x,4) + x2 + x3)
f2 ← update(f, subset=sex=="male")
f3 ← update(f, .~.-x2)         # remove x2 from model
f4 ← update(f, .~. + rcs(x5,5))# add rcs(x5,5) to model
f5 ← update(f, y2 ~ .)         # same terms, new response var.
```

The different operators that can be used to express a model are summarized in the following table As shown above, transformations of variables may be included in the formula which makes

Table 9.1: *Operators in Formulae*

Expression	Meaning
`Y~M`	Y is modeled as M
`M1+M2`	Include M_1 and M_2
`M1-M2`	Include M_1 and leave out M_2 (`-1` deletes intercept term)
`M1:M2`	The cross-product of M_1 and M_2
`M1*M2`	`M1+M2+M1:M2`
`(M1+M2)^m`	M_1 and M_2 and all the powers and interaction terms up to order m.
`poly(M,n)`	Orthogonal polynomial of order n
`I()`	Remove the special meaning of operators

it very flexible. `rcs` in Design is the transformation for a restricted cubic spline. By default it takes 5 knots, but you can give it the number of knots or their position if you desire. `lsp(age,75)` fits age as a linear spline with a knot at 75 years of age (i.e., a bilinear relationship). For `lsp` you need to give it the position of the knots.

Transformations which involve the use of `*`, `^`, and `|` which have special meaning in this context, need to be enclosed within the function `I()`.

9.2 Purposes and Capabilities of Design

Harrell's Design library supports biostatistical and epidemiologic modeling, testing, estimation, validation, graphics, prediction, and typesetting. The name "Design" comes from the fact that this library works by storing enhanced model design attributes in the fit. These attributes are ones needed to generate the design matrix in the first place. Design consists of about 200 functions that assist and streamline modeling and also contains new function for binary and ordinal logistic regression models and the Buckley–James censored least squares multiple linear regression model, and implements penalized maximum likelihood estimation (shrinkage) for logistic and ordinary linear models. Design works with almost any regression model, but it was especially written to work with the models mentioned below. To use Design, you should have already installed and attached Hmisc. To access Design you need to put in the search list the directory where the functions are stored. You must force Design to be placed in front of other libraries, as Design overrides a few system–provided functions (`model.frame.default` and `Surv` being two of them):

```
> library(Design,T)
```

The Design library implements the following statistical methods.

1. Ordinary linear regression models
2. Binary and ordinal logistic models (proportional odds and continuation ratio models)
3. Cox model
4. Parametric survival models in the accelerated failure time class
5. Buckley–James distribution–free regression model for right–censored responses
6. Bootstrap model validation to obtain unbiased estimates of model performance without requiring a separate validation sample
7. Automatic Wald tests of all effects in the model, e.g., tests of nonlinearity of main effects when the variable does not interact with other variables, tests of nonlinearity of interaction effects, tests for whether a predictor is important, either as a main effect or as an effect modifier
8. Graphical depictions of model estimates (effect plots, odds/hazard ratio plots, nomograms that allow model predictions to be obtained manually even when there are nonlinear effects and interactions in the model)
9. Various smoothed residual plots, including some new residual plots for verifying ordinal logistic model assumptions
10. Composing S functions to evaluate the linear predictor ($X\hat{\beta}$), hazard function, survival function, and quantile functions analytically from the fitted model
11. Typesetting of fitted model using $\LaTeX$
12. Robust covariance matrix estimation (Huber or bootstrap)
13. Cubic regression splines with linear tail restrictions
14. Tensor splines (formed by taking cross–product of all spline terms of each variable)
15. Interactions restricted to not be doubly nonlinear
16. Penalized maximum likelihood estimation for ordinary linear regression and logistic regression models. Different parts of the model may be penalized by different amounts, e.g., you may want to penalize interaction or nonlinear effects more than main effects or linear effects

17. Estimation of hazard or odds ratios in presence of nonlinearity and interaction
18. Sensitivity analysis for an unmeasured binary confounder

Many of the functions in Design are organized into groups in the following tables.

Table 9.2: *Special fitting functions*

Function	Purpose	Related S Functions
`ols`	Ordinary and penalized least squares linear model	`lm`
`lrm`	Binary and ordinal logistic regression model	`glm`
	Has options for penalized maximum likelihood estimation	
`psm`	Accelerated failure time parametric survival models	`survreg`
`cph`	Cox proportional hazards regression	`coxph`
`bj`	Buckley–James least squares model for censored data	`survreg`

The following functions have special meaning when using Design.

Table 9.3: *Functions for transforming predictor variables in models*

Function	Purpose	Related S Functions
`asis`	No post–transformation (seldom used explicitly)	
`rcs`	Restricted cubic splines	`ns`
`pol`	Polynomial using standard notation	`poly`
`lsp`	Linear spline	
`catg`	Categorical predictor (seldom)	`factor`
`scored`	Ordinal categorical variables	`ordered`
`matrx`	Keep variables as group for `anova` and `fastbw` (seldom)	`matrix`
`strat`	Non-modeled stratification factors (used for `cph` only)	`strata`

Table 9.4: *Generic Functions and Methods*

Function	Purpose	Related Functions
`print`	Print parameters and statistics of fit	
`coef`	Fitted regression coefficients	
`formula`	Formula used in the fit	
`specs`	Detailed specifications of fit (e.g., knot locations)	
`robcov`	Robust covariance matrix estimates	
`bootcov`	Bootstrap covariance matrix estimates and bootstrap distributions of estimates	
`pentrace`	Find optimum penalty factors by tracing effective AIC for a grid of penalties	
`effective.df`	Print effective d.f. for each type of variable in model, for penalized fit or `pentrace` result	
`rm.impute`	Impute repeated measures data with non–random dropout	`transcan`, `fit.mult.impute`
`summary`	Summary of effects of predictors	
`plot.summary`	Plot continuously shaded confidence bars for results of `summary`	
`anova`	Wald tests of most meaningful hypotheses	
`plot.anova`	Graphical depiction of anova	
`contrast`	General contrasts, C.L., tests	
`plot`	Plot effects of predictors	
`gendata`	Easily generate data with predictor combinations	
`predict`	Obtain predicted values or design matrix	
`fastbw`	Fast backward step–down variable selection	`step`
`residuals`	(or `resid`) Residuals, influence statistics from fit	
`sensuc`	Sensitivity analysis for unmeasured confounders	
`which.influence`	Which observations are overly influential	`residuals`
`latex`	LaTeX representation of fitted model	`display`
`Dialog`	Create a menu to enter predictor values and obtain predicted values from fit	`Function.Design` `nomogram.Design`
`Function`	S function analytic representation of $X\hat{\beta}$ from a fitted regression model	`Function.transcan` `Function.areg.boot`
`Hazard`	S function analytic representation of a fitted hazard function (for `psm`)	
`Survival`	S function analytic representation of fitted survival function (for `psm,cph`)	
`Quantile`	S function analytic representation of fitted function for quantiles of survival time (for `psm, cph`)	
`Mean`	S function analytic representation of fitted function for mean survival time	

Table 9.5: *Generic Functions and Methods*

Function	Purpose	Related S Functions
`nomogram`	Draws a nomogram for the fitted model	`latex, plot`
`survest`	Estimate survival probabilities (`psm, cph`)	`survfit`
`survplot`	Plot survival curves (`psm, cph`)	`plot.survfit`
`validate`	Validate indexes of model fit using resampling	
`calibrate`	Resampling estimation of model's calibration curve	`val.prob`
`vif`	Variance inflation factors for fitted model	
`naresid`	Bring elements corresponding to missing data back into predictions and residuals	
`naprint`	Print summary of missing values	

The following list of topics in the online help (for Windows) for Design will also assist in understanding the components of this library.

Add to Existing Plot
Bootstrap
Categorical Data
Character Data Operations
Data Manipulation
Grouping Observations
High-Level Plots
Interfaces to Other Languages
Linear Algebra
Logistic Regression Model
Mathematical Operations
Matrices and Arrays
Methods and Generic Functions
Nonparametric Statistics
Overview
Predictive Accuracy
Printing
Regression
Regression and Classification Trees
Robust/Resistant Techniques
Sampling
Smoothing Operations
Statistical Inference
Statistical Models
Survival Analysis
Utilities
Validation of Prediction Models

See [13] for an overview of survival modeling and validation of survival models using Design. See [14] for a comprehensive case study of ordinal logistic modeling using Design. These papers also have lots of references.

9.2.1 Differences Between `lm` (Builtin) and `Design`'s `ols` Function

dummy variables `ols` uses traditional dummy variable coding.

`NA`**'s** `ols` deletes observations containing `NA`'s for variables in the model. For `lm` you have to specify `na.action=na.omit`. The `resid`, `fitted`, and `predict` (with `type='fitted'`) functions, when used with objects created by `lm`, will not hold places for `NA`'s that were removed during the fitting process. `ols` holds places for `NA`'s so that, for example, residuals can be plotted against variables in the original dataset (without having to remove observations from the variables).

$\hat{\beta}$ The two functions compute identical coefficient and standard error estimates, assuming that ordinary dummy variable coding was used with `factor` variables in `lm` formulas.

`print` `print`'ing an `lm` object results in an abbreviated summary of the model; `ols` prints model summary statistics (including the likelihood ratio χ^2) as well as all coefficients, standard errors, t statistics, and P–values based on the t distribution. `ols` also prints the adjusted R^2 and a summary of how many `NA`'s were due to each variable in the model.

`summary` `summary` for an `lm` object prints output similar to what `print` for an `ols` object prints. `summary` for an `ols` object prints estimates of effects of variables in the model (e.g., inter–quartile range differences in $\hat{Y}$).

`anova` For both `lm` and `ols` F–tests are done by default and there is an option to use χ^2 tests. Unless `ssType=3` is specified to `anova` for `lm`, `anova` prints sequential tests. `anova` for `ols` always prints partial test statistics. So by default, `anova.lm` only prints partial F statistic for the final predictor in the model, and it never tests for linearity for variables that are expanded using polynomials or splines. General partial tests are obtained for `lm` using e.g. `anova(fit.reduced,fit.full)`. `anova` for `ols` prints all partial tests and tests of linearity. When interactions are present, it also prints meaningful "total effect" test statistics (main effects + interaction effects combined) whereas `anova` for `lm` prints meaningless main effect tests. `anova` for `ols` also prints global (over all predictors) tests of linearity and additivity, and pooled tests involving multiple predictors can be easily specified (e.g., `anova(f, sys.bp, dias.bp)`).

`plot` `plot` for `lm` plots regression diagnostics. `plot` for `ols` plots effects of predictors. You can obtain diagnostic plots for an `ols` fit using `plot.lm(fit)`.

other functions `ols` fits can be used with all the other methods in `Design` such as `nomogram`, `validate`, and `calibrate`.

9.3 Examples of the Use of Design

9.3.1 Examples with Graphical Output

The first series of examples we will consider are based on binary logistic analyses of diagnostic data from the Duke Cardiovascular Disease Databank. We consider how age, sex, and serum cholesterol

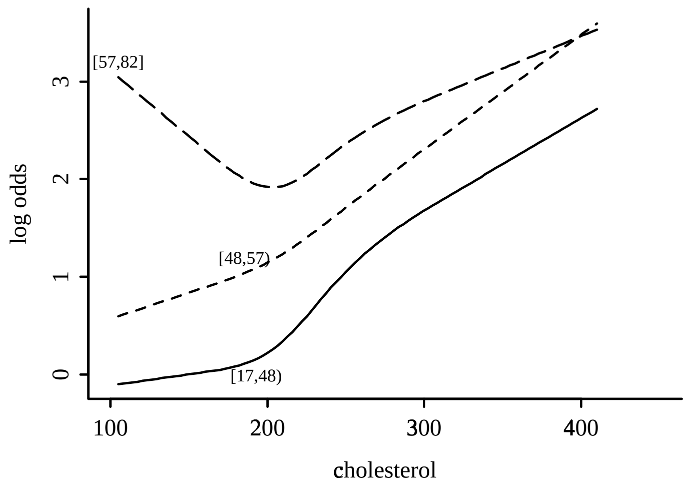

Figure 9.1: *Log odds of significant coronary artery disease modeling age with two dummy variables*

level relate to the probability that a patient will be found to have significant coronary artery disease by cardiac catherization (arteriography). First, to understand interactions involving age we perform an inefficient analysis in which age is stratified into tertiles. We allow for two two–way interactions but not for interaction between `sex` and `cholesterol`. We assume that the relationship between `cholesterol` and log odds of disease is smooth, by fitting a restricted cubic spline function with 4 knots. We plot the fitted model with respect to `cholesterol` and `age.tertile` by placing `cholesterol` on the x–axis and making separate curves for each age tertile. The `sex` variable is set to its reference value. In the notation `cholesterol=NA`, `NA` is a keyword which causes default ranges computed by `datadist` to be used. We could have given ranges explicitly, e.g.,`cholesterol=seq(100,400,by=5)`. The graph appears in Figure 9.1.

```
library(Design, T)
age.tertile ← cut2(age, g=3)
dd ← datadist(age, sex, cholesterol, age.tertile)
options(datadist='dd')
fit ← lrm(sigdz ∼ age.tertile * (sex +
          rcs(cholesterol, 4)))
plot(fit, cholesterol=NA, age.tertile=NA,
     conf.int=F)
```

Next we obtain Wald tests of all meaningful hypotheses which can be inferred from the design:

```
anova(fit)
```

The table below was actually obtained by typing `latex(anova(fit))`. Next we model `age` more properly as a continuous variable (using a restricted cubic spline with 4 default knot locations), allowing for a general interaction surface (tensor spline) between the two continuous predictors. The surface is plotted in Figure 9.2 using default ranges, and the portion of the `anova` table corresponding to interactions is printed.

Wald Statistics

Factor	χ^2	d.f.	P
age.tertile (Main+Interactions)	112.62	10	0.0000
All Interactions	22.37	8	0.0043
sex (Main+Interactions)	328.90	3	0.0000
All Interactions	9.61	2	0.0082
cholesterol (Main+Interactions)	94.01	9	0.0000
All Interactions	10.03	6	0.1234
Nonlinear (Main+Interactions)	10.30	6	0.1124
age.tertile * sex	9.61	2	0.0082
age.tertile * cholesterol	10.03	6	0.1232
Nonlinear Interaction : $f(A,B)$ vs. AB	2.40	4	0.6635
TOTAL NONLINEAR	10.30	6	0.1124
TOTAL INTERACTION	22.37	8	0.0043
TOTAL NONLINEAR+INTERACTION	30.12	10	0.0008
TOTAL	404.94	14	0.0000

```
fit ← lrm(sigdz ~ rcs(age,4) * (sex +
          rcs(cholesterol,4)))
plot(fit, cholesterol=NA, age=NA)
anova(fit)
```

You may want to override the 3–dimensional display method used by the plot.Design function. For example, we can produce an “image” plot where for color plots the third dimension is depicted using colors of the heat spectrum and for black and white plots it is depicted using gray scale. This is done using

```
plot(fit, cholesterol=NA, age=NA, method='image')
```

Wald Statistics

Factor	χ^2	d.f.	P
age * cholesterol	12.95	9	0.1649
Nonlinear Interaction : $f(A,B)$ vs. AB	7.27	8	0.5078
$f(A,B)$ vs. $Af(B)+Bg(A)$	5.41	4	0.2480
Nonlinear Interaction in age vs. $Af(B)$	6.44	6	0.3753
Nonlinear Interaction in cholesterol vs. $Bg(A)$	6.27	6	0.3931

The next model restricts the interaction between age and cholesterol to not be doubly nonlinear. The plot is in Figure 9.3.

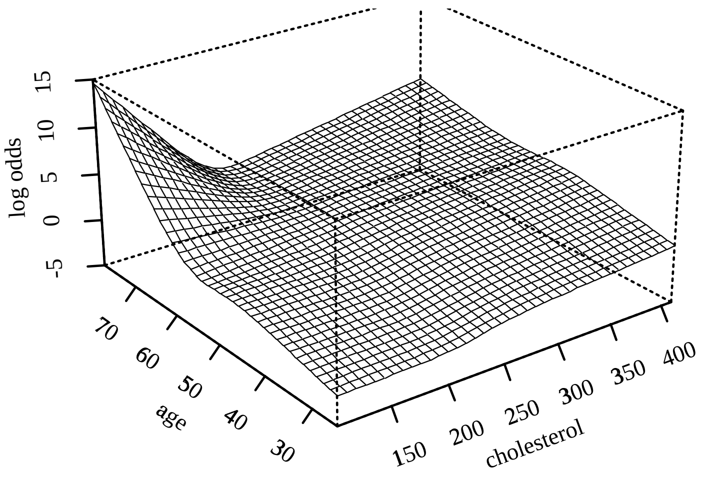

Figure 9.2: *Restricted cubic spline surface in two variables, each with $k = 4$ knots*

```
fit2 ← lrm(sigdz ~ rcs(age,4) * sex +
           rcs(cholesterol,4) +
           rcs(age,4) %ia% rcs(cholesterol,4))
plot(fit2, cholesterol=NA, age=NA)
anova(fit2)
```

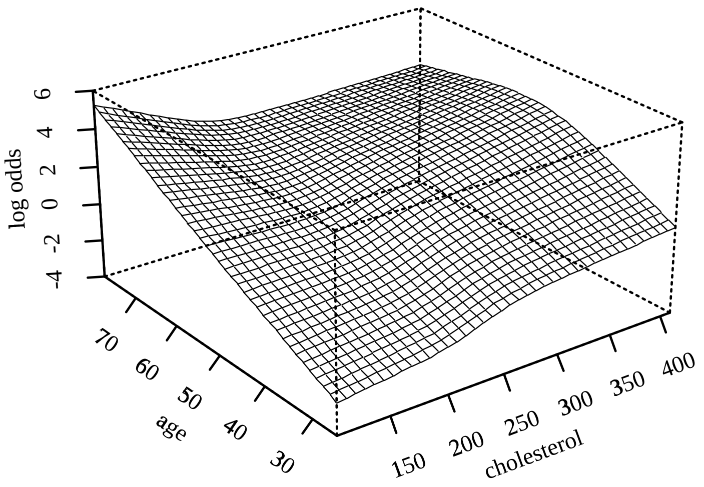

Figure 9.3: *Restricted cubic spline fit with age × spline(cholesterol) and cholesterol × spline(age)*

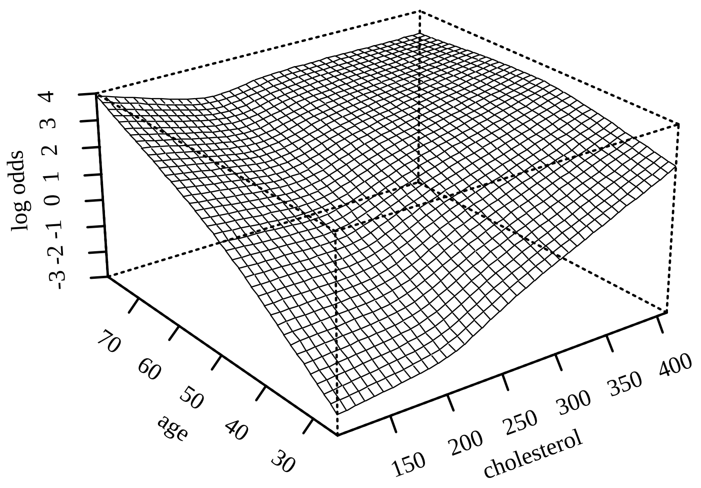

Figure 9.4: *Spline fit with non-linear effects of cholesterol and age and a simple product interaction*

Wald Statistics

Factor	χ^2	d.f.	P
age * cholesterol	10.83	5	0.0548
Nonlinear Interaction : $f(A,B)$ vs. AB	3.12	4	0.5372
Nonlinear Interaction in age vs. $Af(B)$	1.60	2	0.4496
Nonlinear Interaction in cholesterol vs. $Bg(A)$	1.64	2	0.4399

Finally, fit a model in which the interaction between `age` and `cholesterol` is restricted to be linear in both variables (simple product form interaction). The graphical output is in Figure 9.4.

```
fit3 ← lrm(sigdz ~ rcs(age,4) * sex +
           rcs(cholesterol,4) +
           age %ia% cholesterol)
plot(fit3, cholesterol=NA, age=NA)
```

Predictions from this fit can be compared with the first model (Figure 9.1) in which `age` was categorized if we ask for predictions to be made at the mean `age` within each tertile of `age`. See Figure 9.5 for the result.

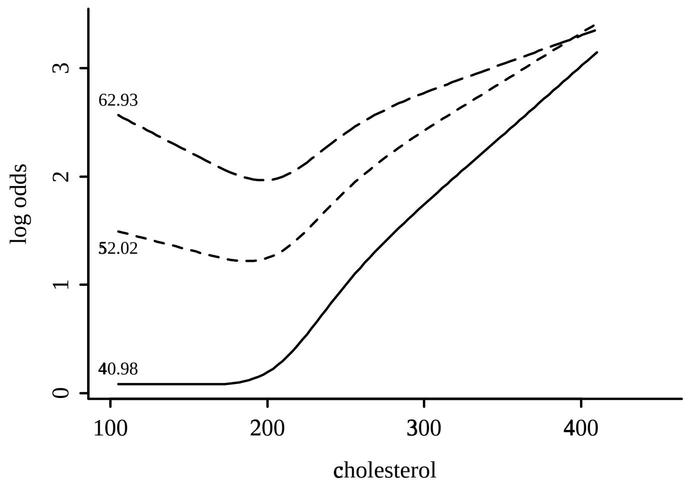

Figure 9.5: *Predictions from linear interaction model with mean age in tertiles indicated*

```
mean.age ← tapply(age, age.tertile, mean)  # add ,na.rm=T if NAs exist
plot(fit3, cholesterol=NA, age=mean.age,
     sex="male", conf.int=F)
```

Now summarize the effects of variables from this fit. The default inter–quartile–range odds ratios are used for continuous variables. Because of the presence of interactions it is important to note the settings of interacting variables when interpreting these odds ratios. These settings are listed at the end to the output from the `summary` (actually `summary.Design`) function.

```
summary(fit3)
```

```
            Factor Low High Diff. Effect S.E. Lower 0.95 Upper 0.95
 age                46  59  13     0.91  0.18  0.55       1.27
  Odds Ratio        46  59  13     2.48    NA  1.73       3.55
 cholesterol       196 259  63     0.75  0.14  0.49       1.02
  Odds Ratio       196 259  63     2.13    NA  1.63       2.78
 sex - female:male   1   2  NA    -2.43  0.15 -2.72      -2.14
  Odds Ratio         1   2  NA     0.09    NA  0.07       0.12

Adjusted to: age=52 sex=male cholesterol=224
```

This summary can also be passed to a plot method, whose results are shown in Figure 9.6. A log odds ratio scale is used.

```
plot(summary(fit3), log=T)
```

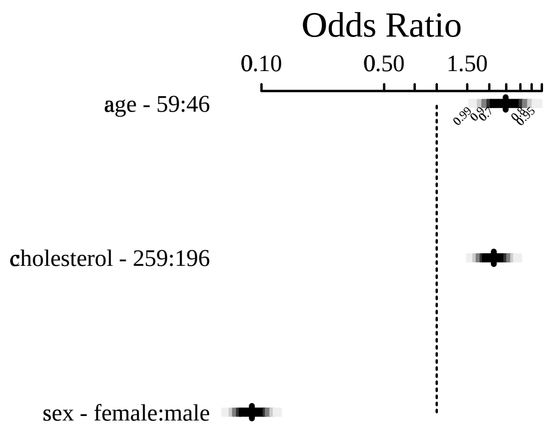

Figure 9.6: *Summary of model using odds ratios and inter–quartile–range odds ratios*

Next consider a simple binary logistic model fitted to a small sample. Eighty bootstrap samples are used to compute the optimism in various indexes of model performance, and optimism is subtracted to obtain bias–corrected (overfitting corrected) estimates. This simple dataset is available on the UVa web page.

```
f ← lrm(response ∼ age + sex, x=T, y=T)
validate(f, B=80)
```

Index	Original Sample	Training Sample	Test Sample	Optimism	Corrected Index
D_{xy}	0.70	0.70	0.67	0.03	0.67
R^2	0.34	0.35	0.32	0.03	0.31
Intercept	0.00	0.00	0.00	0.00	0.00
Slope	0.00	0.00	0.92	0.08	0.92
E_{max}	0.00	0.00	0.02	0.02	0.02
D	0.39	0.41	0.36	0.05	0.34
U	-0.05	-0.05	0.01	-0.06	0.01
Q	0.44	0.46	0.35	0.11	0.33

We can also validate a model obtained by step–down variable selection if we remember to include all candidate predictors in the fit being validated.

```
validate(f, B=80, bw=T, rule="p",
         sls=.1, type="individual")
```

Index	Original Sample	Training Sample	Test Sample	Optimism	Corrected Index
D_{xy}	0.70	0.69	0.65	0.04	0.66
R^2	0.34	0.35	0.31	0.04	0.30
Intercept	0.00	0.00	0.00	0.00	0.00
Slope	1.00	1.00	0.90	0.10	0.90
E_{max}	0.00	0.00	0.02	0.02	0.02
D	0.39	0.41	0.35	0.06	0.33
U	-0.05	-0.05	0.01	-0.06	0.01
Q	0.44	0.46	0.34	0.12	0.32

```
  Factors Retained in Backwards Elimination

 age sex
 *   *
 *   *
 *   *
     *
 *   *
 . . .
 *   *
 *   *
     *
 *   *
 *   *
 *

Frequencies of Numbers of Factors Retained

 1  2
10 70
```

Next turn to Cox survival modeling in a hypothetical dataset. In the following example we do not assume linearity in `age`, proportional hazards for `sex`, or additivity for `age` and `sex`. Figure 9.7 shows the model's estimates of 3–year survival probability after using the log–log transformation.

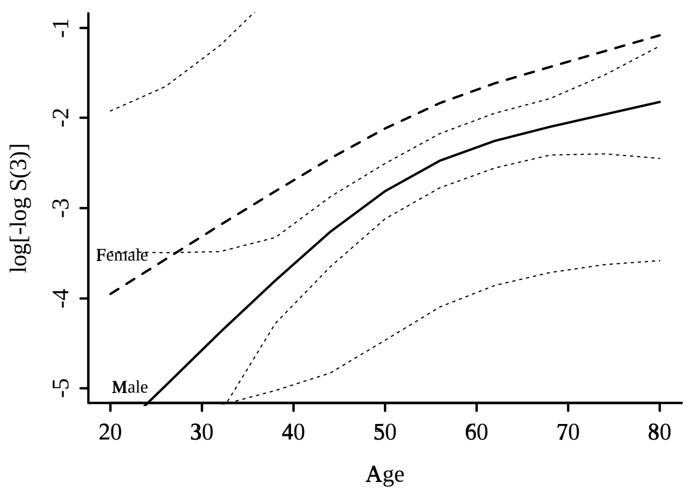

Figure 9.7: *Cox PH model stratified on sex, with interaction between age spline and sex*

```
f ← cph(Srv ∼ rcs(age,4) * strat(sex), surv=T)
plot(f, age=NA, sex=NA, time=3, loglog=T)
```

This model can be depicted with a nomogram. First we invoke the `Survival` function to compose an S function that computes survival probabilities as needed. Then we create special cases of this function to compute 3–year survival probabilities for each of the two `sex` strata. The two functions are needed because we are not assuming proportional hazards for `sex`; separate transformations of time are thus needed to compute survival probabilities. After deriving survival probability prediction functions, the `Quantile` function is used to compose a function to compute quantiles of survival times on demand. Then special cases are computed as before. The `nomogram` function is used to draw the nomogram (Figure 9.8), adding axes corresponding to the special functions just created.

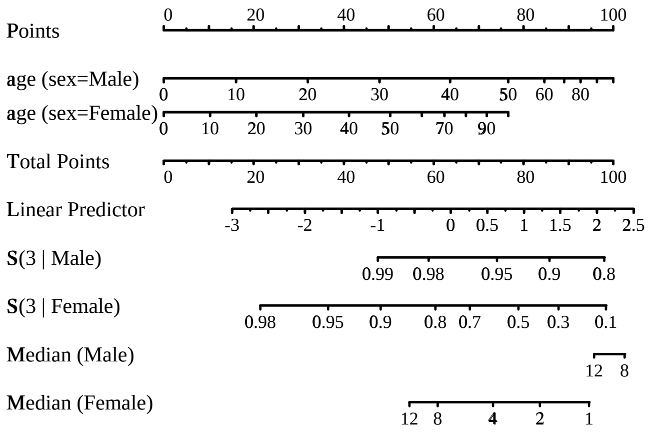

Figure 9.8: *Nomogram from fitted Cox model*

```
surv ← Survival(f)
surv.f ← function(lp) surv(3, lp, stratum="sex=Female")
surv.m ← function(lp) surv(3, lp, stratum="sex=Male")
quant ← Quantile(f)
med.f ← function(lp) quant(.5, lp, stratum="sex=Female")
med.m ← function(lp) quant(.5, lp, stratum="sex=Male")
at.surv ← c(.01,.05,seq(.1,.9,by=.1),.95,.98,.99,.999)
at.med ← c(0,.5,1,1.5,seq(2,14,by=2))
nomogram(f, conf.int=F,
         fun=list(surv.m,surv.f,med.m,med.f),
         funlabel=c("S(3 | Male)","S(3 | Female)",
                    "Median (Male)","Median (Female)"),
         fun.at=list(at.surv,at.surv,at.med,at.med))
```

In the following example we assume proportional hazards for all variables and add another continuous variable to the model. This results in a nomogram (Figure 9.9) which actually requires some manual additions by the user.

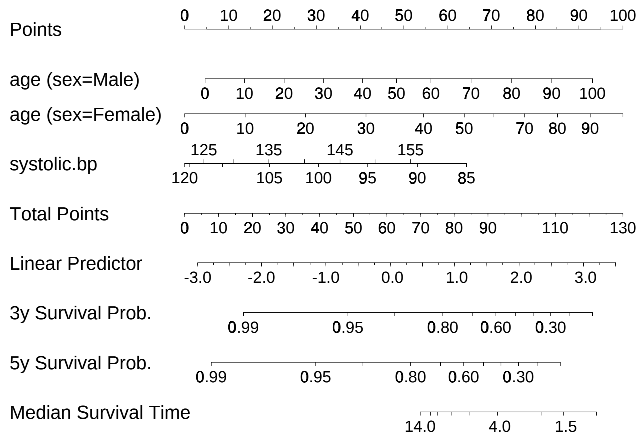

Figure 9.9: *Nomogram from fitted Cox model*

```
f ← cph(Srv ∼ rcs(age,4)*sex + rcs(systolic.bp,4), surv=T)
survfun ← Survival(f)
surv3 ← function(lp) survfun(3, lp)
surv5 ← function(lp) survfun(5, lp)
quant ← Quantile(f)
med ← function(lp) quant(.5, lp)
at.surv ← c(seq(.1,.9,by=.1),.95,.99)
at.med ← c(0,.5,1,1.5,seq(2,14,by=2))
nomogram(f, conf.int=F, fun=list(surv3, surv5, med),
         funlabel=c("3y Survival Prob.",
                    "5y Survival Prob.",
                    "Median Survival Time"),
         fun.at=list(at.surv, at.surv, at.med))
```

Now use the `latex` function to typeset the fitted model. The particular `latex` method for `cph` fits also prints a table of underlying survival estimates to complete the model specification.

```
options(digits=3)
latex(f)
```

$$\text{Prob}\{T \geq t \mid \text{sex} = i\} = S_i(t)^{e^{X\beta}}, \quad \text{where}$$

$$\begin{array}{l} X\hat{\beta} = \\ \quad -3.93 \\ \quad +9.86\times10^{-2}\text{age} - 2.91\times10^{-5}(\text{age} - 30.7)_+^3 \\ \quad +8.72\times10^{-6}(\text{age} - 45.4)_+^3 \\ \quad +6.22\times10^{-5}(\text{age} - 54.8)_+^3 - 4.19\times10^{-5}(\text{age} - 69.6)_+^3 \\ \quad +\{\text{Female}\}[-3.50\times10^{-2}\text{age} + 1.90\times10^{-5}(\text{age} - 30.7)_+^3 \\ \quad -1.76\times10^{-5}(\text{age} - 45.4)_+^3 \\ \quad -2.10\times10^{-5}(\text{age} - 54.8)_+^3 + 1.97\times10^{-5}(\text{age} - 69.6)_+^3] \end{array}$$

and $\{c\} = 1$ if subject is in group c, 0 otherwise,
$(x)_+ = x$ if $x > 0$, 0 otherwise.

t	$S_{Male}(t)$	$S_{Female}(t)$
0	1.000	1.000
1	0.992	0.901
2	0.980	0.815
3	0.973	0.759
4	0.966	0.679
5	0.963	0.612
6	0.955	0.556
7	0.947	0.478
8	0.938	0.437
9	0.932	0.390
10	0.920	0.354
11	0.909	0.322
12	0.909	0.287
13	0.909	0.240
14	0.882	0.240

9.3.2 Binary Logistic Modeling with the Prostate Data Frame

Consider the strange task of predicting the probability of cardiovascular death (vs. alive or death due to other causes) for men with prostate cancer, allowing time until death or censoring to be a predictor variable (!).

```
> library(Design, T)       # make Design functions and datasets available
> attach(prostate)
> cvd ← status %in%  c("dead - heart or vascular","dead - cerebrovascular")
> # Note: %in% is in Hmisc - makes using the match function easier
> table(cvd)
```

```
 FALSE TRUE
   375  127
> f ← lrm(cvd ~ rx+rcs(dtime,5)+age+hx+bp)
> f

Logistic Regression Model

lrm(formula = cvd ~ rx + rcs(dtime, 5) + age + hx + bp)


Frequencies of Responses
 FALSE TRUE
   374  127


Frequencies of Missing Values Due to Each Variable
 cvd rx dtime age hx bp
   0  0     0   1  0  0

 Obs Max Deriv Model L.R. d.f. P     C   Dxy Gamma Tau-a   R2 Brier
 501     2e-05      131.2   10 0 0.811 0.622 0.623 0.236 0.34 0.145

                       Coef    S.E. Wald Z      P
         Intercept -3.01327 1.41243 -2.13  0.0329
rx=0.2 mg estrogen -0.42659 0.34083 -1.25  0.2107
rx=1.0 mg estrogen -0.16740 0.34176 -0.49  0.6243
rx=5.0 mg estrogen  0.36948 0.32082  1.15  0.2495
             dtime -0.02632 0.03855 -0.68  0.4947
            dtime'  0.35472 0.25948  1.37  0.1716
           dtime'' -1.13796 0.74752 -1.52  0.1279
          dtime'''  0.78195 0.99670  0.78  0.4327
               age  0.02417 0.01829  1.32  0.1862
                hx  1.25773 0.24352  5.16  0.0000
                bp  0.17881 0.09702  1.84  0.0653
```

The fitted model object `f`, is a list. Let us take a look at its components.

```
> names(f)
 [1] "freq"              "stats"             "fail"
 [4] "coefficients"      "var"               "u"
 [7] "deviance"          "est"               "non.slopes"
[10] "linear.predictors" "call"              "scale.pred"
[13] "terms"             "assign"            "na.action"
[16] "fail"
```

Most of them are technical and needed for other functions to make calculations but a few have an immediately recognizable meaning like `coefficients`. One could look at them by doing something like `coeff ← f$coefficients`. The preferred method however is to use functions like `coef` and `predict`. `formula` can also be useful to know what model we fitted without having to print all coefficients.

There are many other arguments to `lrm`. Among them are the data set to be used, subset of observations to select, what to do with missing values, and whether to keep or not the design matrix and dependent variable. Look at the help files for this and other modeling functions.

Next we want to do some testing. The `anova` function applied to an `lrm` object performs a Wald test on any variable given, or all variables if no variable is given.

```
> anova(f)
              Wald Statistics          Response: cvd

 Factor     Chi-Square d.f.      P
 rx          6.14       3   0.1049
 dtime      29.03       4   0.0000
  Nonlinear 25.08       3   0.0000
 age         1.75       1   0.1862
 hx         26.67       1   0.0000
 bp          3.40       1   0.0653
 TOTAL      64.07      10   0.0000
```

If you really want to do a stepwise variable selection, the function to use is `fastbw`.

```
> fastbw(f)

 Deleted Chi-Sq d.f.      P Residual d.f.      P   AIC
 age     1.75   1    0.1862 1.75     1    0.1862 -0.25

Approximate Estimates after Deleting Factors

                      Coef    S.E.  Wald Z         P
         Intercept -1.22902 0.41606 -2.9539 3.138e-03
rx=0.2 mg estrogen -0.45556 0.34013 -1.3394 1.805e-01
rx=1.0 mg estrogen -0.15662 0.34167 -0.4584 6.467e-01
rx=5.0 mg estrogen  0.38155 0.32069  1.1898 2.341e-01
             dtime -0.02854 0.03852 -0.7410 4.587e-01
            dtime'  0.35037 0.25946  1.3504 1.769e-01
           dtime'' -1.11290 0.74728 -1.4893 1.364e-01
          dtime'''  0.73557 0.99608  0.7385 4.602e-01
                hx  1.29353 0.24201  5.3449 9.048e-08
                bp  0.17860 0.09702  1.8408 6.565e-02

Factors in Final Model

[1] rx    dtime hx    bp
```

After you run `fastbw` you get an estimate of the coefficients after deleting factors. The arguments to `fastbw` are

```
  fastbw(fit, rule="aic", type="residual", sls=.05, aics=0, eps=1E-9)
```

The stopping rule can be `"aic"` for Akaike's information criteria or `"p"` for p-values. `type` is the type of statistic on which to base the stopping rule. `type` can be `"residual"` for pooled residual

χ^2, or `"individual"` for Wald χ^2 statistics of individual variables. `sls` and `aics` are cut-off values to decide when a variable is dropped from the model.

After using `fastbw` we may decide to refit the model dropping some variable and also on only a subset of the observations. Instead of retyping the `lrm` expression we can use the function `update`.

```
> f1 ← update(f,.~.-age,subset=dtime>20)
```

The arguments to update are the fitted object, the formula suitably modified, and perhaps other arguments. In the formula, we use a `"."` to represent the expressions that were present before and add or substract terms.

9.3.3 Troubleshooting Problems with `factor` Predictors

Here is an example of a problem you may encounter when using a modeling function.

```
> attach(resuse.dframe)
> m ← ols(log(billing) ~ dzgroup)
Error in lm.fit.qr(x, y, qr = ..1): computed fit is singular, rank 8
Dumped
>
> table(dzgroup)
 0:HELP only pts 1:ARF/MOSF 2:COPD 3:CHF 4:Cirrhosis 5:Coma 6:Colon Cancer
               0       1513    458   726         296    247           269
 7:Lung Cancer 8:MOSF w/Malig
           459            333
```

The problem here is that the *factor* dzgroup has HELP as a possible level, but there are no patients in that category. This happens when you have a `factor` or `category` variable and there are no observations for a particular level of the variable. If importing data from SAS and there is an unused SAS PROC FORMAT VALUE label, `sas.get` will create a level for the factor anyway and since there will be no observations the resulting design matrix will be singular because one of the dummy variables is always one. The easiest way out of this problem is to run the `factor` variable through the method for subscripting factor variables as described in Section 3.4.

```
> dzgroup ← dzgroup[]   # use dzgroup[,drop=T] if Hmisc not in effect
> table(dzgroup)
 1:ARF/MOSF 2:COPD 3:CHF 4:Cirrhosis 5:Coma 6:Colon Cancer 7:Lung Cancer
       1513    458   726         296    247           269           459
 8:MOSF w/Malig
            333
```

Now the new version of `dzgroup` will replace the old one in any subsequent calculations.

Another problem that may arise is when you want to collapse a few levels of a factor into a single level. To do this one can redefine the levels of the factor. The `%in%` operator can help here; see Sections 3.4 and 4.4 for other examples. Let us look at the variable `group2` in the data frame.

```
> group ← group2
> table(group)
 1:surgery 2:cardiology 3:oncology 4:pulmonary/MICU 5:medicine 6:medicine 6C
       692          784        757              886        970            70
 7:medicine 8B 8:medicine 9B 9:medical house staff 10:surgical house staff
            50            82                     0                       0
```

We would like to collapse levels 6,7 and 8 into level 5. We redefine the levels attribute of `group` this way.

```
> levels(group)[levels(group) %in% c("6:medicine 6C","7:medicine 8B",
+ "8:medicine 9B")] ← "5:medicine"
> # or levels(group) ← list(medicine=c('5:medicine','6:medicine 6C',
> # '7:medicine 8B','8:medicine 9B'))
> table(group)
 1:surgery 2:cardiology 3:oncology 4:pulmonary/MICU 5:medicine
       692          784        757              886       1172
 9:medical house staff 10:surgical house staff
                     0                       0
> # Now delete unused levels
> group ← group[]   $ group[,drop=T] if Hmisc not in effect
> table(group)
    1:surgery 2:cardiology 3:oncology 4:pulmonary/MICU 5:medicine
 10       692          784        757              886       1172
```

Notice that the the values of medicine 6C, 8B, 9B etc have been correctly collapsed into medicine.

9.3.4 A Comprehensive Hypothetical Example

As another example of using many of the Design functions (as well as the `describe` and `impute` functions in Hmisc), suppose that a categorical variable `treat` has values `"a"`, `"b"`, and `"c"`, an ordinal variable `num.diseases` has values 0,1,2,3,4, and that there are two continuous variables, `age` and `cholesterol`. `age` is fitted with a restricted cubic spline, while `cholesterol` is transformed using the transformation `log(cholesterol+10)`. Cholesterol is missing on three subjects, and we impute these using the overall median cholesterol. We wish to allow for interaction between `treat` and `cholesterol`. The following S program will fit a logistic model, test all effects in the design, estimate effects, and plot estimated transformations. The fit for `num.diseases` really considers the variable to be a 5-level categorical variable. The only difference is that a 3 d.f. test of linearity is done to assess whether the variable can be re-modeled "asis". Here we also show statements to store predictor characteristics from `datadist`.

```
library(Design, T)
ddist ← datadist(cholesterol, treat, num.disease, age)
# Could have used ddist ← datadist(data.frame.name)
options(datadist="ddist")        # defines data dist. to Design
cholesterol ← impute(cholesterol)
fit ← lrm(y ∼ treat + scored(num.diseases) + rcs(age) +
                 log(cholesterol+10) + treat:log(cholesterol+10))
describe(y ∼ treat + scored(num.diseases) + rcs(age))
# or use describe(formula(fit)) for all variables used in fit
# describe function (in Hmisc) gets simple statistics on variables
#fit ← robcov(fit)              # Would make all statistics which follow
                                # use a robust covariance matrix
                                # would need x=T, y=T in lrm()
specs(fit)                      # Describe the design characteristics
anova(fit)
anova(fit, treat, cholesterol)   # Test these 2 by themselves
plot(anova(fit))                 # Summarize anova graphically
summary(fit)                     # Estimate effects using default ranges
plot(summary(fit))               # Graphical display of effects with C.L.
summary(fit, treat="b", age=60)  # Specify reference cell and adjustment val
summary(fit, age=c(50,70))       # Estimate effect of increasing age from
                                 # 50 to 70
summary(fit, age=c(50,60,70))    # Increase age from 50 to 70, adjust to
                                 # 60 when estimating effects of other factors
# If had not defined datadist, would have to define ranges for all var.
plot(fit, age=seq(20,80,length=100), treat=NA, conf.int=F)
                                 # Plot relationship between age and log
                                 # odds, separate curve for each treat,
                                 # no C.I.
plot(fit, age=NA, cholesterol=NA)# 3-dimensional perspective plot for age,
                                 # cholesterol, and log odds using default
                                 # ranges for both variables
plot(fit, num.diseases=NA, fun=function(x) 1/(1+exp(-x)) ,
     ylab="Prob", conf.int=.9)   # Plot estimated probabilities instead of
                                 # log odds
# Again, if no datadist were defined, would have to tell plot all limits

# Estimate and test treatment (b-a) effect averaged over 3 cholesterols
contrast(fit, list(treat='b', cholesterol=c(150,200,250)),
              list(treat='a', cholesterol=c(150,200,250)),
         type='average')

logit ← predict(fit, expand.grid(treat="b",num.dis=1:3,age=c(20,40,60),
                 cholesterol=seq(100,300,length=10)))
# Could also obtain list of predictor settings interactively
logit ← predict(fit, gendata(fit, nobs=12))

# Since age doesn't interact with anything, we can quickly and
# interactively try various transformations of age, taking the spline
# function of age as the gold standard. We are seeking a linearizing
# transformation.

ag ← 10:80
logit ← predict(fit, expand.grid(treat="a", num.dis=0, age=ag,
                 cholesterol=median(cholesterol)),
                 type="terms")[,"age"]
# Note: if age interacted with anything, this would be the age
#       "main effect" ignoring interaction terms
# Could also use
#    logit ← plot(f, age=ag, ...)$x.xbeta[,2]
# which allows evaluation of the shape for any level of interacting
# factors.  When age does not interact with anything, the result from
# predict(f, ..., type="terms") would equal the result from
# plot if all other terms were ignored
# Could also specify
#    logit ← predict(fit, gendata(fit, age=ag, cholesterol=...))
# Un-mentioned variables set to reference values

plot(ag^.5, logit)               # try square root vs. spline transform.
plot(ag^1.5, logit)              # try 1.5 power

latex(fit)                       # invokes latex.lrm, creates fit.tex
# Draw a nomogram for the model fit
nomogram(fit)

# Compose S function to evaluate linear predictors analytically
g <- Function(fit)
g(treat='b', cholesterol=260, age=50)
# Letting num.diseases default to reference value
```

The following is a typical sequence of steps that would be used with Design in conjunction with the Hmisc transcan function to do single imputation of all `NA`s in the predictors[1], fit a model, do backward stepdown to reduce the number of predictors in the model (with all the severe problems this can entail), and use the bootstrap to validate this stepwise model, repeating the variable selection for each re-sample. Here we take a short cut as the imputation is not repeated within the bootstrap. In what follows we (atypically) have only 3 candidate predictors. In practice be sure to have the validate and calibrate functions operate on a model fit that contains all predictors that were involved in previous analyses that used the response variable. Here the imputation is necessary because backward stepdown would otherwise delete observations missing on any candidate variable.

```
xt <- transcan(~ x1 + x2 + x3, imputed=T)
impute(xt)  # imputes any NAs in x1, x2, x3
# Now fit original full model on filled-in data
f <- lrm(y ~ x1 + rcs(x2,4) + x3, x=T, y=T)  # x,y allow boot.
fastbw(f)   # derive stepdown model (using default stopping rule)
validate(f, B=100, bw=T)          # repeats fastbw 100 times
cal <- calibrate(f, B=100, bw=T)  # also repeats fastbw
plot(cal)
```

See Section 13.2 for a much more comprehensive example of the use of Design.

9.3.5 Using Design and Interactive Graphics to Generate Flexible Functions

Sometimes one wishes to simulate data from a complex non–monotonic regression relationships. In this example we open an empty plot, draw a curve using mouse clicks, fit the function using least squares via a spline function, and create an S function representing a close approximation to the manually drawn function. This latter function can then be used inside a simulation loop to create a population predictor effect, for example. This example also shows how restricted cubic splines are fitted. You may have to specify knot locations yourself to fit the tails of the curve adequately if you don't click the mouse very rapidly there.

[1] Multiple imputation would be better but would be harder to do in the context of bootstrap model validation.

```
plot(0,0, xlim=c(0,1), ylim=c(0,1))  # open empty graph
z ← locator(type='b')                # return points until right mouse button
                                     # clicked, drawing pts and lines
x1 ← z$x                             # pull off x-coordinates
y  ← z$y                             # pull off y-coordinates
w  ← datadist(x1)
options(datadist='w')
h  ← ols(y ~ rcs(x1,6))              # least squares fit
plot(h, add=T, conf.int=F, col=2)    # show fitted curve
hf ← Function(h)                     # represent fit as an S function
xx ← seq(0,1,length=100)             # grid of points to evaluate
lines(xx, hf(xx), lwd=2, col=2)      # re-draw fitted curve
```

9.4 Checklist of Problems to Avoid When Using Design

1. Don't have a formula like `y ~ age + age^2`. In S you need to connect related variables using a function which produces a matrix, such as `pol` or `rcs`. This allows effect estimates (e.g., hazard ratios) to be computed as well as multiple d.f. tests of association.

2. Don't use `poly` or `strata` inside formulas used in Design. Use `pol` and `strat` instead.

3. Almost never code your own dummy variables or interaction variables in S. Let S do this automatically. Otherwise, `anova` and other functions can't do their job.

4. Almost never transform predictors outside of the model formula, as then plots of predicted values vs. predictor values, and other displays, would not be made on the original scale. Use instead something like `y ~ log(cell.count+1)`, which will allow `cell.count` to appear on x–axes. You can get fancier, e.g., `y ~ rcs(log(cell.count+1),4)` to fit a restricted cubic spline with 4 knots in `log(cell.count+1)`. For more complex transformations do something like

```
f ← function(x) {
   ... various 'if' statements, etc.
   log(pmin(x,50000)+1)
}
fit1 ← lrm(death ∼ f(cell.count))
fit2 ← lrm(death ∼ rcs(f(cell.count),4))
```

5. Don't put `$` inside variable names used in formulas. Either attach data frames or use `data=`.

6. Don't forget to use `datadist` and `options(datadist=...)`. Try to use it at the top of your program so that all model fits can automatically take advantage if its distributional summaries for the predictors.

7. Don't `validate` or `calibrate` models which were reduced by dropping "insignificant" predictors. Proper bootstrap or cross–validation must repeat any variable selection steps for each re–sample. Therefore, `validate` or `calibrate` models which contain all candidate predictors, and if you must reduce models, specify the option `bw=T` along with any non–default stopping rules when you run `validate` or `calibrate`.

8. Dropping of "insignificant" predictors ruins much of the usual statistical inference for regression models (confidence limits, standard errors, P–values, χ^2, ordinary indexes of model performance) and it also results in models which will have worse predictive discrimination.

9. Make sure you include the `T` in `library(Design, T)`, and do `library(Design, T)` after `library(Hmisc, T)`

9.5 Describing Representation of Subjects

The Hmisc `dataRep` function is useful for describing how well a new subject was represented in a dataset used to develop a predictive model. This can supplement confidence intervals in guarding against over–interpretation when extrapolation is done.

Chapter 10

Principles of Graph Construction

The ability to construct clear and informative graphs is related to the ability to understand the data. There are many excellent texts on statistical graphics (many of which are listed at the end of this chapter). Some of the best are Cleveland's 1994 book *The Elements of Graphing Data* and the books by Tufte. The suggestions for making good statistical graphics outlined here are heavily influenced by Cleveland's books, and quotes below are from his 1994 book.

10.1 Graphical Perception

- Goals in communicating information: reader perception of data values and of data patterns. Both accuracy and speed are important.
- Pattern perception is done by

  detection : recognition of geometry encoding physical values

  assembly : grouping of detected symbol elements

  estimation : assessment of relative magnitudes of two physical values
- For estimation, many graphics involve discrimination, ranking, and estimation of ratios
- Humans are not good at estimating differences without directly seeing differences (especially for steep curves)
- Humans do not naturally order color hues
- Only a limited number of hues can be discriminated in one graphic
- Weber's law: The probability of a human detecting a difference in two lines is related to the ratio of the two line lengths

- This is why grid lines and frames improve perception and is related to the benefits of having multiple graphs on a common scale.
  - eye can see ratios of filled or of unfilled areas, whichever is most extreme
- For categorical displays, sorting categories by order of values attached to categories can improve accuracy of perception. Watch out for over-interpretation of extremes though.
- The aspect ratio (height/width) does not have to be unity. Using an aspect ratio such that the average absolute curve angle is 45° results in better perception of shapes and differences (banking to 45°).
- Optical illusions can be caused by:
  - hues, e.g., red is emotional. A red area may be perceived as larger.
  - shading; larger regions appear to be darker
  - orientation of pie chart with respect to the horizon
- Humans are bad at perceiving relative angles (the principal perception task used in a pie chart)
- Here is a hierarchy of human graphical perception abilities:
  1. Position along a common scale (most accurate task)
  2. Position along identical nonaligned scales
  3. Length
  4. Angle and slope
  5. Area
  6. Volume
  7. Color: hue (red, green, blue, etc.), saturation (pale/deep), and lightness
     - Hue can give good discrimination but poor ordering

10.2 General Suggestions

- Exclude unneeded dimensions (e.g. width, depth of bars)
- "Make the data stand out. Avoid Superfluity"; Decrease ink to information ratio
- "There are some who argue that a graph is a success only if the important information in the data can be seen in a few seconds. . . . Many useful graphs require careful, detailed study."
- When actual data points need to be shown and they are too numerous, consider showing a random sample of the data.
- Omit "chartjunk"
- Keep continuous variables continuous; avoid grouping them into intervals. Grouping may be necessary for some tables but not for graphs.
- Beware of subsetting the data finer than the sample size can support; conditioning on many variables simultaneously (instead of multivariable modeling) can result in very imprecise estimates

10.3 Tufte on "Chartjunk"

> Chartjunk does not achieve the goals of its propagators. The overwhelming fact of data graphics is that they stand or fall on their content, gracefully displayed. Graphics do not become attractive and interesting through the addition of ornamental hatching and false perspective to a few bars. Chartjunk can turn bores into disasters, but it can never rescue a thin data set. The best designs . . . are *intriguing and curiosity-provoking*, drawing the viewer into the wonder of the data, sometimes by narrative power, sometimes by immense detail, and sometimes by elegant presentation of simple but interesting data. But no information, no sense of discovery, no wonder, no substance is generated by chartjunk.

— Tufte p. 121, 1983

10.4 Tufte's Views on Graphical Excellence

"Excellence in statistical graphics consists of complex ideas communicated with clarity, precision, and efficiency. Graphical displays should

- show the data
- induce the viewer to think about the substance rather than about methodology, graphic design, the technology of graphic production, or something else
- avoid distorting what the data have to say
- present many numbers in a small space
- make large data sets coherent
- encourage the eye to compare different pieces of data
- reveal the data at several levels of detail, from a broad overview to the fine structure
- serve a reasonably clear purpose: description, exploration, tabulation, or decoration
- be closely integrated with the statistical and verbal descriptions of a data set."

10.5 Formatting

- Tick Marks should point outward
- x- and y-axes should intersect to the left of the lowest x value and below the lowest y value, to keep values from being hidden by axes
- Minimize the use of remote legends. Curves can be labeled at points of maximum separation (see the Hmisc `labcurve` function).

10.6 Color, Symbols, and Line Styles

- Some symbols (especially letters and solids) can be hard to discern
- Use hues if needed to add another dimension of information, but try not to exceed 3 different hues. Instead, use different saturations in each of the three different hues.
- Make notations and symbols in the plots as consistent as possible with other parts, like tables and texts
- Different dashing patterns are hard to read especially when curves inter-twine or when step functions are being displayed
- An effective coding scheme for two lines is to use a thin black line and a thick gray scale line

10.7 Scaling

- Consider the inclusion of 0 in your axis. Many times it is essential to include 0 to tell the full story. Often the inclusion of zero is unnecessary.
- Use a log scale when it is important to understand percent change of multiplicative factors or to cure skewness toward large values
- Humans have difficulty judging steep slopes; bank to 45°, i.e., choose the aspect ratio so that average absolute angle in curves is 45°.

10.8 Displaying Estimates Stratified by Categories

- Perception of relative lengths is most accurate — areas of pie slices are difficult to discern
- Bar charts have many problems:
  - High ink to information ratio
  - Error bars cause perception errors
  - Can only show one-sided confidence intervals well
  - Thick bars reduce the number of categories that can be shown
  - Labels on vertical bar charts are difficult to read
- Dot plots are almost always better
- Consider multi-panel side-by-side displays for comparing several contrasting or similar cases. Make sure the scales in both x and y axes are the same across different panels.
- Consider ordering categories by values represented, for more accurate perception

10.9 Displaying Distribution Characteristics

- When only summary or representative values are shown, try to show their confidence bounds or distributional properties, e.g., error bars for confidence bounds or box plot
- It is better to show confidence limits than to show ± 1 standard error
- Often it is better still to show variability of *raw* values (quartiles as in a box plot so as to not assume normality, or S.D.)
- For a quick comparison of distributions of a continuous variable against many categories, try box plots.
- When comparing two or three groups, overlaid empirical distribution function plots may be best, as these show all aspects of the distribution of a continuous variable.

10.10 Showing Differences

- Often the only way to perceive differences accurately is to actually compute differences; then plot them
- It is not a waste of space to show stratified estimates and differences between them on the same page using multiple panels
- This also addresses the problem that confidence limits for differences cannot be easily derived from intervals for individual estimates; differences can easily be significant even when individual confidence intervals overlap.
- Humans can't judge differences between steep curves; one needs to actually compute differences and plot them.

10.11 Choosing the Best Graph Type

The recommendations that follow are good on the average, but be sure to think about alternatives for your particular data set. For nonparametric trend lines, it is advisable to add a "rug" plot to show the density of the data used to make the nonparametric regression estimate. Alternatively, use the bootstrap to derive nonparametric confidence bands for the nonparametric smoother.

10.11.1 Single Categorical Variable

Use a dot plot or horizontal bar chart to show the proportion corresponding to each category. Second choices for values are percentages and frequencies. The total sample size and number of missing values should be displayed somewhere on the page. If there are many categories and they are not naturally ordered, you may want to order them by the relative frequency to help the reader estimate values.

10.11.2 Single Continuous Numeric Variable

An empirical cumulative distribution function, optionally showing selected quantiles, conveys the most information and requires no grouping of the variable. A box plot will show selected quantiles effectively, and box plots are especially useful when stratifying by multiple categories of another variable. Histograms are also possible.

10.11.3 Categorical Response Variable vs. Categorical Ind. Var.

This is essentially a frequency table. It can also be depicted graphically (Section 6.3).

10.11.4 Categorical Response vs. a Continuous Ind. Var.

Choose one or more categories and use a nonparametric smoother to relate the independent variable to the proportion of subjects in the categories of interest. Show a rug plot on the x-axis.

10.11.5 Continuous Response Variable vs. Categorical Ind. Var.

If there are only two or three categories, superimposed empirical cumulative distribution plots with selected quantiles can be quite effective. Also consider box plots, or a dot plot with error bars, to depict the median and outer quartiles. Occasionally, a back-to-back histogram can be effective for two groups (see the Hmisc `histbackback` function).

10.11.6 Continuous Response vs. Continuous Ind. Var.

A nonparametric smoother is often ideal. You can add rug plots for the x- and y-axes, and if the sample size is not too large, plot the raw data. If you don't trust nonparametric smoothers, group the x-variable into intervals having a given number of observations, and for each x-interval plot characteristics (3 quartiles or mean $\pm$ 2 SD, for example) vs. the mean x in the interval. This is done automatically with the Hmisc `xYplot` function with the `methods='quantile'` option.

10.12 Conditioning Variables

You can condition (stratify) on one or more variables by making separate pages by strata, by making separate panels within a page, and by superposing groups of points (using different symbols or colors) or curves within a panel. The actual method of stratifying on the conditional variable(s) depends on the type of variables.

Categorical variable(s) : The only choice to make in conditioning (stratifying) on categorical variables is whether to combine any low-frequency categories. If you decide to combine them on the basis of relative frequencies you can use the `combine.levels` function in Hmisc.

Continuous numeric variable(s) : Unfortunately, to condition on a continuous variable without the use of a parametric statistical model, one must split the variable into intervals. The first choice is whether the intervals of the numeric variable should be overlapping or non-overlapping. For the former the built-in `equal.count` function can be used for a paneling or grouping variable in trellis graphics (these overlapping intervals are called "shingles" in trellis).

For non-overlapping intervals the Hmisc `cut2` function is a good choice because of its many options and compact labeling.

`hesweb1.med.virginia.edu/biostat/teaching/statcomp` has more information on statistical graphics and links to pertinent sites.

Bibliography

[1] C. F. Alzola and F. E. Harrell. *An Introduction to* S *and the* `Hmisc` *and* `Design` *Libraries.* Available from `hesweb1.med.virginia.edu/biostat/s/doc/splus.pdf`.

[2] F. J. Anscombe. Graphs in statistical analysis. *American Statistician*, 27:17–21, 1973.

[3] J. Bertin. *Graphics and Graphic Information-Processing.* de Gruyter, Berlin, 1981.

[4] D. B. Carr and S. M. Nusser. Converting tables to plots: A challenge from Iowa State. *Statistical Computing and Graphics Newsletter, ASA*, December 1995.

[5] W. S. Cleveland. Graphs in scientific publications (c/r: 85v39 p238-239). *American Statistician*, 38:261–269, 1984.

[6] W. S. Cleveland. *Visualizing Data.* Hobart Press, Summit, NJ, 1993.

[7] W. S. Cleveland. *The Elements of Graphing Data.* Hobart Press, Summit, NJ, 1994.

[8] W. S. Cleveland and R. McGill. A color-caused optical illusion on a statistical graph. *American Statistician*, 37:101–105, 1983.

[9] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79:531–554, 1984.

[10] A. Gelman, C. Pasarica, and R. Dodhia. Let's practice what we preach: Turning tables into graphs. *The American Statistician*, 56:121–130, 2002.

[11] X. Li, J. Buechner, P. Tarwater, and A. Muñoz. A diamond-shaped equiponderant graphical display of the effects of two categorical predictors on continuous outcomes. *The American Statistician*, 57:193–199, 2003.

[12] F. E. Harrell. *Regression Modeling Strategies.* New York: Springer, 2001.

[13] G. T. Henry. *Graphing Data.* Sage, Newbury Park, CA, 1995.

[14] D. McNeil. On graphing paired data. *American Statistician*, 46:307–311, 1992.

[15] S. M. Powsner and E. R. Tufte. Graphical summary of patient status. *Lancet*, 344:386–389, 1994.

[16] P. R. Rosenbaum. Exploratory plots for paired data. *American Statistician*, 43:108–109, 1989.

[17] P. D. Sasieni and P. Royston. Dotplots. *Applied Statistics*, 45:219–234, 1996.

[18] P. A. Singer and A. R. Feinstein. Graphical display of categorical data. *Journal of Clinical Epidemiology*, 46:231–236, 1993.

[19] E. R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[20] E. R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, Connecticut, 1990.

[21] E. R. Tufte. *Visual Explanations.* Graphics Press, Cheshire, CT, 1997.

[22] H. Wainer. How to display data badly. *American Statistician*, 38:137, 1984.

[23] H. Wainer. Three graphic memorials. *Chance*, 7:52–55, 1994.

[24] H. Wainer. Depicting error. *American Statistician*, 50:101–111, 1996.

[25] A. Wallgren, B. Wallgren, R. Persson, U. Jorner, and J. Haaland. *Graphing Statistics & Data.* Sage Publications, Thousand Oaks, 1996.

[26] L. Wilkinson. *The Grammar of Graphics.* Springer, New York, 1999.

Chapter 11

Graphics in S

11.1 Overview

S has a large variety of plotting routines. In order to be able to display a plot one needs to open a special window for this purpose as described in Section 12.2. For example, UNIX users might open a graphics window using `X11()` or `motif()`, and Windows 3.3 users usually use `win.graph()`, while 4.x or later Windows users let the system open a graph sheet.

We will begin this chapter by covering some of the lower–level plotting functions, then we will move up to the higher–level multi–way `trellis` graphics (which generalize the `coplot` function). All of the basic graphs and many `trellis`–type graphs can be produced in S-PLUS 4.x or later using dialog boxes, and graphs can be edited and annotated interactively using point–and–click. All this is usually done by clicking on a data frame in the left pane of the `Object Browser` to get a list of the frame's variables in the right pane. Then one or more variable names are highlighted (using left click or control–left–click) and a graph type is clicked on a 2D or 3D graphics palette. However, with our slant of being able to reproduce analyses when data are updated, we will present only the command interface in this chapter.

In 4.x or later you can edit an S-PLUS graph, whether it was produced by a dialog or by commands, using the S-PLUS graphics editor, or you can even edit the graph in Microsoft Word or Powerpoint when the editing process requires S-PLUS to manipulate graphics objects. This is done through dynamic object linking in Windows 95. Also take a look at `Metafile Companion` for editing Windows metafiles, as described briefly in Section 1.9.

The typical plotting command is of the form `plot(fun1(var1),fun2(var2),...)`. This will plot a transformation (`fun2`) of `var2` on the y–axis vs. the transformation `fun1` of `var1` on the x–axis. The `...` represent graphical parameters to pass to `plot` to control different aspects of the plot such as plot size, labeling of axes, position of the plot on the paper, orientation, limits for the axes, line type, number of plots to a page, etc.

Graphical parameters can be passed to `plot` as part of its arguments, or be defined beforehand through the `par(...)` function. In this latter case, they remain in effect for the duration of the

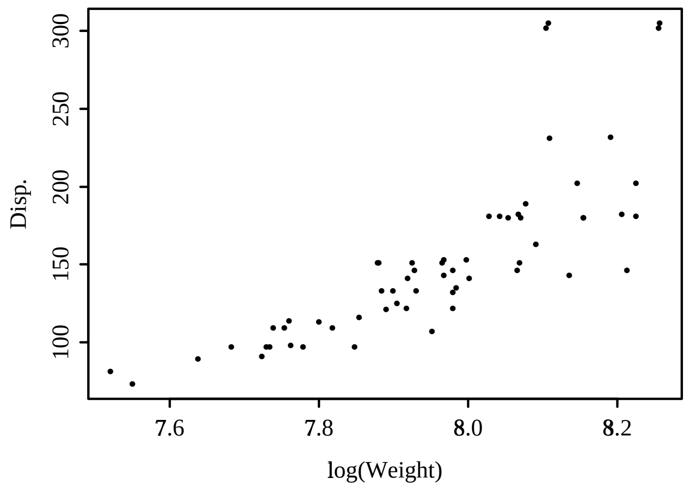

Figure 11.1: *Basic Plot*

S session, while in the former they are only active for that particular `plot` command. There are many graphical parameters and we can classify them in basically four groups: parameters affecting graphical elements (lines, points, polygons and so on), parameters affecting axes and tick marks, parameters affecting margins and parameters affecting a multiple figure layout. Let's do some examples using the car.test.frame dataframe that is supplied with S-Plus.

```
> attach(car.test.frame)
> names(car.test.frame)
[1] "Price"      "Country"     "Reliability" "Mileage"     "Type"
[6] "Weight"     "Disp."       "HP"
> plot(log(Weight),Disp.)
```

The results are shown in Figure 11.1.

Axes are labeled with variable names and they are scaled to include all observations in the data frame. We will improve it a little bit by adding titles, a smooth fitting line and axis labels

```
> plot(log(Weight),Disp.,xlab="Log of Weight",
+      ylab="Displacement",
+      main="Displacement vs log of Weight")
> lines(supsmu(log(Weight),Disp.))
```

If at this point we wanted to print the plot, there is an option in the graphics window to do so (in UNIX, click the right mouse button on `Graph` and choose `Print` from the menu). You can also make a copy of the plot in a smaller window or resize it to your preferences). For Windows 4.x and later you use the `Print Graph Sheet Page` command in the `File` menu. In the example above, the arguments for labels and titles could have been given in a different command `title(main,sub,xlab,ylab)`. The command `lines(x,y)` adds lines with coordinates determined by the vectors `x` and `y` (it could also be a matrix with two columns). Here, the role of `x` and `y` is played by the pair of vectors returned

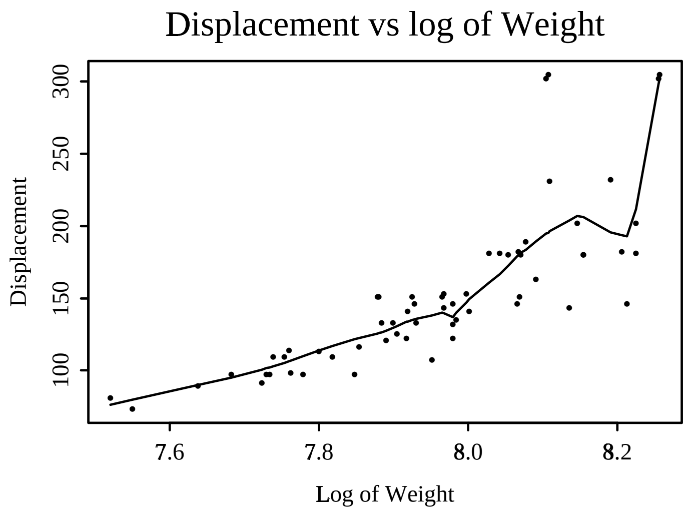

Figure 11.2: *Basic Plot with Labels and Title*

by `supsmu` which determine a non-parametric smooth fit. `lowess` is another function which does smooth fitting note that you must remove NAs yourself to use it. A function similar in purpose to `lines` is `points`. The results are shown in Figure 11.2.

Let us look at the distribution of mileage by type of car. Let us try

```
> plot(Type,Mileage,boxmeans=T,rotate=T)
```

Here, again we see how S is smart enough to recognize that `Type` is a factor variable and does a boxplot. The arguments `boxmeans` and `rotate` display the mean `Mileage` by `Type` and rotate labels on the x-axis. However, to do boxplots, the `boxplot` function is preferred due to its greater flexibility.

```
> boxplot(split(Mileage,Type),varwidth=T,notch=T)
```

The `split` function here is needed to classify `Mileage` by `Type`. The argument `varwidth` specifies that the box widths is proportional to the square root of the number of observation in each box. The `notch` arguments provide notches that can be used for a rough significance test at the 5% level.

Let us examine how `plot` behaves on fitted models. (`plot` acts differently on models not fitted with a `Design` function).

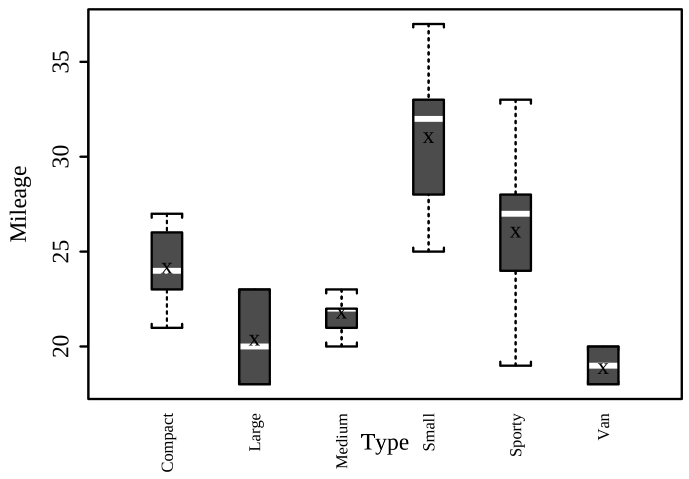

Figure 11.3: *Plotting a Factor*

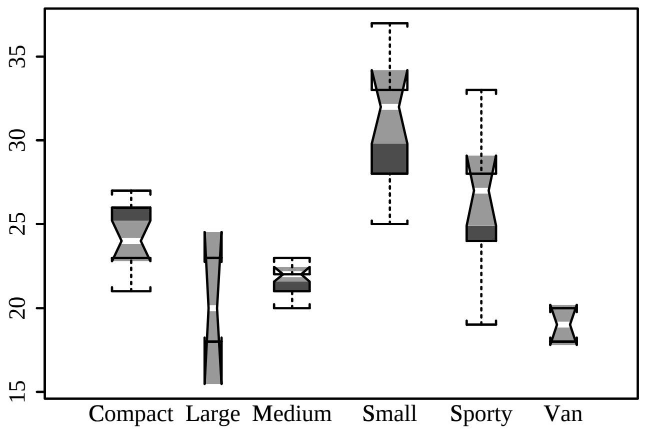

Figure 11.4: *Example of Boxplot*

```
> f ← ols(Mileage ~ Weight+Type+Disp.,x=T)
> f

Least Squares Regression Model

ols(formula = Mileage  ~  Weight + Type + Disp.)

n=60   p=7

Residuals:
    Min     1Q   Median   3Q   Max
 -5.515 -1.141 -0.05707 1.54 4.715

Coefficients:
               Value Std. Error  t value Pr(>|t|)
  Intercept  33.9640   4.1053     8.2733   0.0000
     Weight  -0.0017   0.0018    -0.9152   0.3643
 Type=Large   2.6627   1.8435     1.4444   0.1546
Type=Medium  -0.4514   0.9673    -0.4666   0.6427
 Type=Small   4.3597   1.1194     3.8948   0.0003
Type=Sporty   2.6860   0.9985     2.6900   0.0096
   Type=Van  -3.2339   1.4875    -2.1740   0.0343
      Disp.  -0.0361   0.0124    -2.9100   0.0053

Residual standard error: 2.238 on 52 degrees of freedom
Multiple R-Squared: 0.8077   Adjusted R-Squared: 0.7818
```

`plot` can be applied to fitted models to display how the response function behaves as the predictors in the model vary. When using `plot` with Design for this purpose we need to tell the function how to adjust the predictors that are not being plotted. This can be done in two ways: by passing them explicitly to `plot` as arguments, or by means of the `datadist` function. `datadist` takes a dataframe or a list of variable names and returns an object of class “datadist” with information that helps `plot` determine the limits of the variables being plotted, and adjustments for other variables in the model. We then set the `options(datadist=...)` parameter which instructs S where to point to find the limits for the variables. For example

```
> dd ← datadist(Type,Weight,Disp.)
> options(datadist="dd")
> dd
                   Type  Weight  Disp.
     Low:effect         2571.25 113.75
      Adjust to Compact 2885.00 144.50
    High:effect         3231.25 180.00
 Low:prediction Compact 2165.25  90.90
High:prediction Van     3735.00 302.00
            Low Compact 1845.00  73.00
           High Van     3855.00 305.00

Values:
```

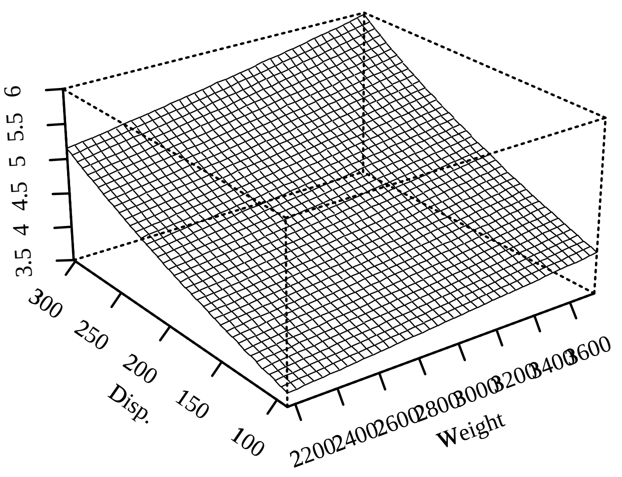

Figure 11.5: *Example of Plot on a Fitted Model*

```
Type : Compact Large Medium Small Sporty Van
```

This is saying that if we want to plot the predicted values from `f` as a function of `Weight` and `Disp.`, they will range from 2165.25 to 3735.00 and 90.90 to 302.00 respectively, while the `Type` factor will be adjusted to the value `Compact`. To specify that we want `plot` to cover the full range determined by `datadist` we use `NA` by convention

```
> plot(f,Weight=NA,Disp.=NA,fun=function(x) 100/x)
```

In this example we chose to represent a transformation of the response (`100/x`) by defining it in line through the `fun=` argument. This is a feature common to many S functions. If we wanted to use a factor rather than a continuous variable as a predictor we would have obtained one curve for each level of the factor plus confidence intervals.

We can override the values chosen by `datadist` as limits and adjustment

```
> plot(f,Disp.=seq(150,250,by=5),Type=NA,Weight=2800,conf.int=F)
```

How does `plot` manage to behave so differently depending on the kind of arguments we give it? The answer is in the `class` attribute of its main argument. If we try to look at the `plot` function itself we get

```
> plot
function(x, ...)
UseMethod("plot")
```

What this means is that when we type `plot(x)`, `plot` looks at the `class` attribute of `x`, say `z`, and then calls the function `plot.z` which will produce the appropriate plot.

```
> class(f)
```

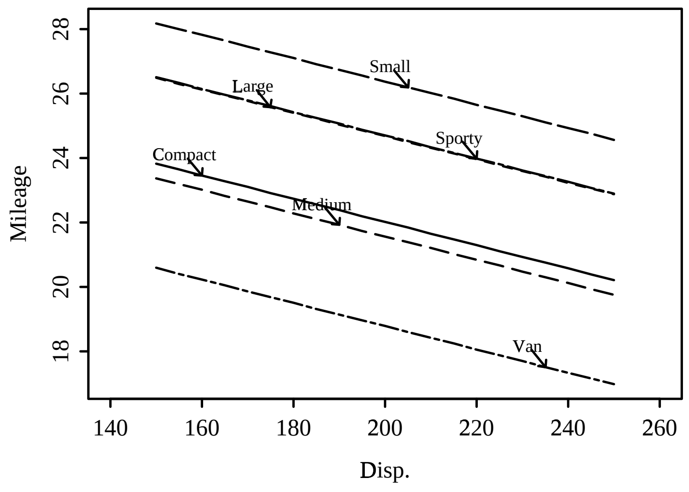

Figure 11.6: *Overriding datadist Values*

```
[1] "ols"    "Design" "lm"
> args(plot.ols)
Error: Object "plot.ols" not found
Dumped
> args(plot.Design)
function(fit, ..., xlim, ylim, fun, xlab, ylab, conf.int = 0.95, add = F,
        label.curves = T, eye, lty, col = 1, adj.zero = F, ref.zero = F,
        adj.subtitle, cex.adj = 1, non.slopes, time, loglog = F, val.lev = F,
        digits = 4, cex.label = 0.75)
> class(site)
[1] "factor"
> args(plot.factor)
function(x, y = NULL, ..., style = "box", rotate = sum(nchar(xalabs)) > 80,
        boxmeans = F, character, xlab = fn, ylab = yname, ylim = ymm, ask = T,
        data = NULL)
```

This is showing that an object can have more than one class attribute and `plot` will look at all of them starting from the left until it finds a function of the form `plot.z`. This behavior is not restricted to `plot`. Many other functions such as `print, summary` and `anova` also act this way. They are called *generic* functions and *methods* can be written for them, meaning special functions to handle special objects. One consequence of this software design is that when we look up help, we want to make sure that we are looking for help for the correct function. For example `help(anova)` will give us help for the S `anova` function, while `help(anova.Design)` will give us help on Design's `anova` function.

Two other plotting functions that we want to look at are `qqnorm` and `coplot`. `qqnorm` (complemented by `qqline`) does a normal probability plot. In our example we may want to check if the residuals from the fitted model are normally distributed. We could extract the residuals by using `resid` and then type

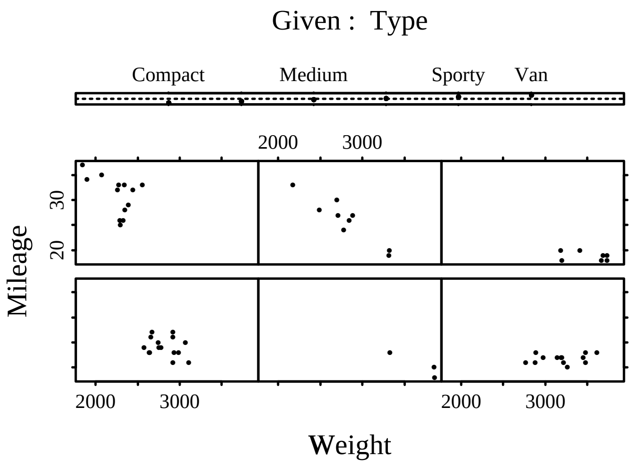

Figure 11.7: *Example of Co-Plot*

```
> qqnorm(resid(f))
> qqline(resid(f))
```

There are functions to do quantile-quantile plots for other distributions (See [7, section 5.5.4.1]).

If we want to see how `Mileage` depends on `Weight` across the different types we can do a *conditioning plot* or *coplot*.

```
coplot(y ~ x|z, given.values=z, panel=panel.smooth)
```

gives a scatterplot of `y` vs `x` conditioning on the values of `z`. `z` could be a factor or a series of overlapping (or not) intervals. In this latter case, if `z` is divided into, say, `m` intervals, then `m` plots of `y` vs `x` are done with the variables restricted to those intervals. When `z` is a factor we get one plot for each level of the the factor.

```
> coplot(Mileage ~ Weight|Type)
```

The labeling here is a little unusual since it starts from left to right and from bottom to top. The idea is that it should be read like any other graph, with the origin in the bottom left of the page and values on the x-axis increasing to the right, and values on the y-axis increasing upwards. The key on how to read it is given by the top panel. If `z` is a continuous variable the function `co.intervals` could be used to construct the intervals. It doesn't matter what function we use to construct them as long as the end result is a matrix with two columns and the interval extremes as rows.

There are many other plotting functions. Two that are worth exploring are `pairs` and `interaction.plot`.

11.2 Adding Text or Legends and Identifying Observations

In the example above, we fitted a least squares model to `Mileage`. We specified `x=T` to store the design matrix along with the fit. Using this information we can now use the function `which.influence` to extract influential observations

```
> w ← which.influence(f,cutoff=.5)
> w
$Intercept:
[1] "Ford Festiva 4"       "Honda Civic CRX Si 4"

$Weight:
[1] "Ford Festiva 4"

$Type:
[1] "Ford Festiva 4"
```

w is a list with one component for each factor with influential observations according to a criteria defined by the cutoff= argument. Each component lists the observations that unduly affect that particular coefficient. We would like to refit the model dropping this observations. This is a good time to use unlist.

```
> wu ← unlist(w,use.names=F)
> wu
[1] "Ford Festiva 4"       "Honda Civic CRX Si 4" "Ford Festiva 4"
[4] "Ford Festiva 4"
> wu ← sort(unique(wu))
> wu
[1] "Ford Festiva 4"
> wuc ← match(wu,row.names(car.test.frame))
```

unlist makes the list w into a vector but one with repeated components and not sorted. This is solved by using unique and sort. Then we use match to get a vector with the indexes of car.test.frame corresponding to the influential observations. We can use them to extract those elements not in wu.

```
> fu ← update(f,. ~ .,subset=row.names(car.test.frame)[-wuc],x=F)
> fu

Least Squares Regression Model

ols(formula = Mileage  ~  Weight + Disp. + Type, subset = row.names(
        car.test.frame)[ - wuc], x = F)

n=58   p=7

Residuals:
    Min     1Q Median    3Q   Max
 -5.067 -1.216  0.103 1.188 4.245

Coefficients:
               Value Std. Error  t value Pr(>|t|)
  Intercept  28.0726   4.1988     6.6859   0.0000
     Weight   0.0007   0.0018     0.3771   0.7077
      Disp.  -0.0421   0.0117    -3.6100   0.0007
 Type=Large   1.4479   1.7522     0.8263   0.4125
```

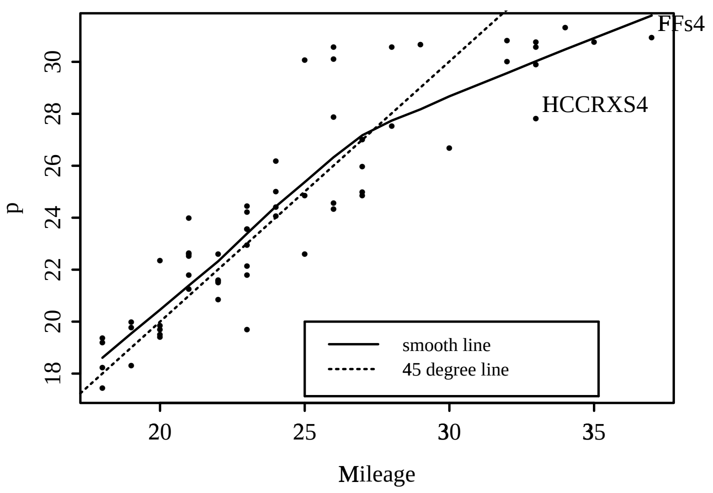

Figure 11.8: *Identifying Observations*

```
Type=Medium  -1.1343   0.9186    -1.2347   0.2227
 Type=Small   4.9799   1.0654     4.6739   0.0000
Type=Sporty   2.3026   0.9553     2.4104   0.0197
   Type=Van  -4.7508   1.4495    -3.2775   0.0019

Residual standard error: 2.067 on 50 degrees of freedom
Multiple R-Squared: 0.8096   Adjusted R-Squared: 0.7829

> p ← predict(fu,newdata=car.test.frame)
> plot(Mileage,p)
> identify(Mileage,p,label=abbreviate(row.names(car.test.frame)))
```

We can try to look for the influential observations in the plot of observed vs predicted. When we type `identify(...)` we are calling an interactive procedure; now S expects us to point at the points in the graph and click on the left mouse button. It will then label the point with the vector given to the `label=` argument. (See fig 9). Here we used `abbreviate` to produce a shortened version of the label. When we don't want to label any more points, we just position the cursor anywhere on the graphics window and click on the middle mouse button. S will then return a vector with the indexes of those observations labeled, which we can use to check the data.

The legend in the plot was obtained using a combination of `legend` and `locator`.

```
> ss ← lowess(Mileage,p,iter=0)
> plot(Mileage,p)
> lines(ss,lty=1)
> abline(0,1,lty=2)
> legend(locator(1),c("smooth line","45 degree line"),lty=1:2)
```

The arguments to `legend` are pretty easy to interpret here, except perhaps for `locator(1)`. `locator(n)` is just a drawing function which will connect n points with lines as you draw them on the screen by clicking the mouse. `legend` interprets `locator(1)` to mean to position 1 point (the top left of the box) in the place where you click the mouse. Alternatively, a vector of coordinates could have been used.

The `legend` function has largely be obsoleted by the `key` function which is more versatile. When identifying separate curves you may also want to let Hmisc's `labcurve` function call `key` for you.

11.3 Hmisc and Design High–Level Plotting Functions

Table 11.1 summarizes high–level plotting commands from Hmisc, Design, and standard S.

The Hmisc function `scat1d` has several options for showing the density of the raw data through a rug plot on one of the four axes or along a user–specified curve. `scat1d` is especially good at showing the data density for very large datasets, as it will draw a random sub–segment of each whisker or "strand" of the rug. It has an argument which allows you to place the rug plot along a curve rather than on an axis. For extremely large datasets, you may want to use the `histSpike` function instead (see below). By default, if $n \geq 2000$, `scat1d` calls `histSpike` automatically.

The `datadensity` function is a generalization of `scat1d` which calls `scat1d` for each continuous variable in a data frame and draws frequency bar plots for categorical variables. `datadensity` places each variable on a separate axis and writes the number of missing values to the right of each axis if there are any `NA`s. `datadensity` is a good tool for initial data inspections (see Section 3.6). Figure 11.9 shows the result of the commands

```
> datadensity(prostate)    # prostate is on Web page
```

`datadensity` accepts a `group` argument which will cause tick marks for individual raw data points to be distinguished by color. This can be used to identify associations between the `group` variable and the other variables.

`histSpike` draws a "spike" histogram using by default 100 bins. This is especially useful for large datasets as such histograms can have high resolution. `histSpike` can also draw a kernel density plot. A useful feature of `histSpike` is that like `scat1d` it can be used to enhance an existing plot (e.g., scatter diagram) with histograms or density plots showing the marginal distribution of one of the variables plotted. These plots point inward from any of the 4 axis lines. Unlike `scat1d` you have to add the argument `add=T` to do this.

The `ecdf` function in the Hmisc library draws empirical cumulative distributions for either individual variables or for all variables in a data frame. For the latter case, a suitable matrix of plots is set up automatically. `ecdf` also accepts a `group` variable, so it can automatically draw multiple cdfs based on stratifying the data on categorical variables. Curves are labeled using `labcurve`. Here are some examples.

```
ecdf(age, group=sex)
ecdf(age, group=interaction(sex,race))
```

Empirical distribution functions have two advantages over histograms: they do not require the choice of bins, and you can overlay several distributions on the same plot. There is a `trellis` version of `ecdf` (see Section 11.4). It is easy to have `ecdf` call `scat1d` to add a rug plot to the cumulative distribution plot, or to call `histSpike` to add a histogram or density plot to the plot. For example, you can type `ecdf(age, datadensity='rug')`.

Table 11.1: *Non–`trellis` High Level Plotting Functions*

Function	Description
`barplot`	vertical or horizontal bar graph
`bpplot`	box–percentile plots (Hmisc)
`boxplot`	side-by-side boxplots
`contour`	contour plot
`coplot`	separate plots of different ranges
`datadensity`	multivariable version of Hmisc's `scat1d`
	displays data density for all variables in a data frame
`dotchart`	displays values based on position of dots
`ecdf`	empirical distribution function plot (Hmisc)
`faces`	Chernoff faces for multivariate data
`hist`	histogram
`hist.data.frame`	histrogram of all variables in a data frame (Hmisc)
`histSpike`	high–resolution "spike" histograms and density plots
`labcurve`	draw and label curves or label existing curves (Hmisc)
`nomogram`	nomograms (Design)
`pairs`	all possible pairs of scatterplots
`persp`	3-D perspective plots of grids
`pie`	pie charts
`plclust`	plots of cluster trees from hclust
`plot`	scatterplot or line plot
`plot.anova.Design`	Dot chart of `anova` table (Design)
`plot.Design`	family of functions for fitted objects
`plot.summary.Design`	plots effect ratios and CIs (Design)
`plot.summary.formula`	plotting functions for `summary.formula` function (Hmisc)
`plsmo`	plot smoothed nonparametric estimates (Hmisc)
`qqnorm`	normal probability plot
`qqplot`	quantile-quantile plot
`scat1d`	add data density (rug plot) to plot (Hmisc enhancement of `rug`)
`survplot`	survival plots (Design)
`symbol.freq`	diagram of frequency table (Hmisc)
`tsplot`	time series plots
`usa`	map of the US

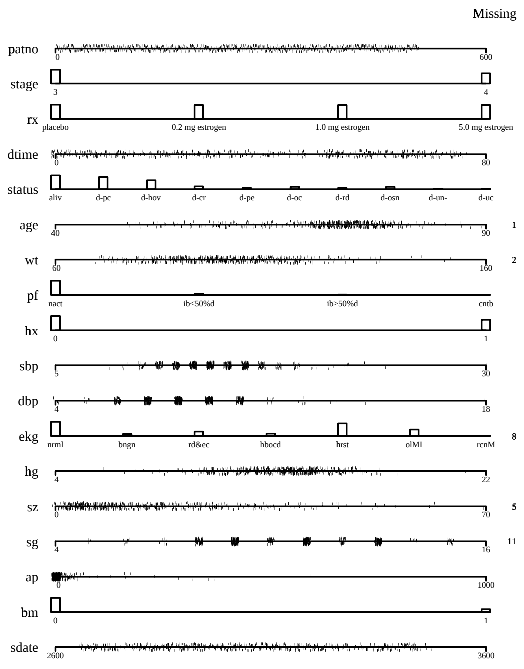

Figure 11.9: `datadensity` *plot for the* `prostate` *data frame*

The builtin function `cdf.compare` draws an empirical CDF alongside a theoretical one. See also the `trellis stripplot` function discussed in Section 11.4.

There are several builtin ways to draw box and whisker plots in S. These plots provide useful overall summaries but they are not especially sensitive to the behavior of the tails of the distributions. A continuous version of the box plot called the box–percentile plot, was developed by Esty and Banfield. Banfield's `bpplot` function, with slight modifications, is included in Hmisc. To quote from Banfield's help file for `bpplot`: "Box-percentile plots are similiar to boxplots, except box–percentile plots supply more information about the univariate distributions. At any height the width of the irregular 'box' is proportional to the percentile of that height, up the the 50th percentile, and above the 50th percentile the width is proportional to 100 minus the percentile. Thus, the width at any given height is proportional to the percent of observations that are more extreme in that direction. As in boxplots, the median, 25th and 75th percentiles are marked with line segments across the box".

The leftmost arguments to `bpplot` may be a sequence of not necessarily equal–length vectors or a list containing the same. The latter is often produced by the `split` function in order to stratify on a grouping variable.

Here is an example of a box–percentile plot showing the distribution of ages in the `titanic` data frame, stratified by passenger class. We add a group representing the overall age distribution and a hypothetical group having a normal distribution with the same mean and variance of the overall age distribution. To omit these extra groups use the simple command `bpplot(split(age,pclass), xlab='Passenger Class', ylab='Years')`.

```
w ← split(age, pclass)
w$Overall ← age
a ← age[!is.na(age)]
w$Normal  ← rnorm(2000, mean(a), sqrt(var(a)))
bpplot(w, xlab='Passenger Class', ylab='Years', srtx=30)
# labels are rotated 30 degrees
```

The result is shown in Figure 11.10.

Hmisc's `labcurve` function will automatically label a set of existing curves on the current plot, or draw and label curves, if you tell it the coordinates of all of the curves. `labcurve` has many options. By default it will label curves at the points for which they are maximally separated. You can usually get away without a legend if you do this. If you want a legend, `labcurve` will position the legend at the most empty area of the plot, and it is easier to use than the `key` or `legend` commands in many cases. Here is an example where `labcurve` draws and labels the curves. The lines are distinguished by different colors and styles.

```
labcurve(list(Female=list(ages.f,height.f,col=2),
              Male  =list(ages.m,height.m,col=3,lty=2)),
         xlab='Age', ylab='Height', pl=T)
# add ,keys=c('f','m') to label curves with single letters
```

The `plsmo` function plots smoothed estimates of `x` vs. `y`, handling missing data for `lowess` or `supsmu`, and adding axis labels. It optionally suppresses plotting extrapolated estimates. An optional `group` variable can be specified to compute and plot the smooth curves by levels of `group`. When `group` is present, the `datadensity` option will draw tick marks showing the location of the raw `x`–values, separately for each curve. `plsmo` also has an option to plot connected points for raw

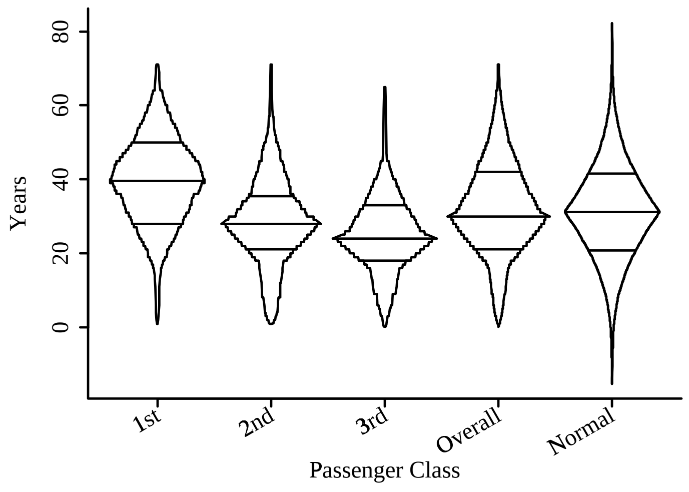

Figure 11.10: *Box–percentile plot showing the distribution of ages of passengers on the Titanic stratified by passenger class. More than half of the passengers are omitted from this plot due to missing ages. The rightmost part of the plot shows the box–percentile plot for a normal distribution having mean and standard deviation equal to that of the ages in the data.*

data, with no smoothing. Here is an example in which the log odds of smoothed estimates of the probability of death vs. age is plotted, stratified by sex.

```
plsmo(age, death, group=sex, datadensity=T, fun=plogis)
```

See also the Hmisc `trellis panel` function `panel.plsmo` described in Section 11.4.

11.4 `trellis` Graphics

S-Plus also comes with a library of advanced graphics functions called `trellis`. In R, this library is called `lattice`. The library name comes from the fact that when you are displaying multiple graphics panels after conditioning on other variables, the resulting display looks like a garden trellisor lattice. `trellis` has these advantages over S's older graphical functions:

1. `trellis` uses better defaults for fonts, colors, and point symbols.
2. It uses S symbolic formulas for specification of the main and conditioning variables.
3. Related to the last point, you can condition on one variable or on the cross–classifications of any number of "given" variables.
4. Some new graphics types are implemented, including some for 3–D plots.
5. Graphical parameters such as the aspect ratio are chosen for improved graphical perception.

Function	Purpose	Formula Argument
`barchart`	Bar chart	`y ~ x | g1*g2`
`bwplot`	Box and whisker plot	`y ~ x | g1*g2`
`densityplot`	Probability density plot	`~ x | g1*g2`
`dotplot`	Dot plot	`y ~ x | g1*g2`
`Dotplot`	Hmisc generalization of `dotplot`	`y ~ x | g1*g2`
`ecdf`	Hmisc ECDF plot	`~ x | g1*g2, groups=g3`
`histogram`	Histogram	`~ x | g1*g2`
`parallel`	Parallel coordinate plot	`~ x | g1*g2`
`panel.bpplot`	enhanced box plots and box–%-tile plots with `bwplot`	
`panel.plsmo`	Hmisc `panel` function for `xyplot`	`y ~ x | g1*g2, groups=g3`
`splom`	Multi–panel scatterplot matrices	`~ x | g1*g2`
`stripplot`	One–dimensional scatter plot	`y ~ x | g1*g2`
`xyplot`	Conditioning plots/scatter plots	`y ~ x | g1*g2`
`xYplot`	Hmisc generalization of `xyplot` for multi–column `y`	`Cbind(y,y2,y3) ~ x | g1*g2, groups=g3`
`setTrellis`	Hmisc `trellis` setup	
`trellis.strip.blank`	Hmisc function to set `trellis` to use blank background for panel titles	

Here is a list of some of the most commonly used `trellis` functions.

Type `?trellis` to see several other `trellis` functions. In the formulas, `g1,g2,g3` are categorical "given" (conditioning) variables. There may be more than two of them. If only one is given, the `*`s are omitted. `y` is a `factor` variable (except for `xyplot` for which it is numeric), and `x` is a numeric variable. For `splom`, `x` is a matrix or a data frame. `parallel` is useful for representing multivariate data. For it, `x` is a numeric matrix whose columns represent the multivariate response. See Section 11.4.1 for a generalization of `xyplot` in Hmisc.

You can use `trellis`'s `shingle` function to make overlapping intervals of a continuous variable, to use it as a conditioning or `y` variable. The `equal.count` function is convenient for this. For example, the `g1, g2` variables above could be of the form `equal.count(z)` where `z` is a numeric continuous variable. Optional arguments to `equal.count` are `number` (number of intervals) and `overlap` (degree of overlap between intervals). Defaults are 6 and .5, respectively.

When plotting a factor variable, particularly when making dot plots, one frequently wants to control the ordering of factor levels in constructing an axis or in arranging panels. The `reorder.factor` function is useful for this. As an example consider two simple vectors. For generality, we create a factor variable whose levels are not in alphabetic order.

```
> a ← c(1, 3, 2.5, 2.2)
> b ← factor(c('a','c','b','d'), c('d','c','b','a'))
levels(b)
[1] "d" "c" "b" "a"

> dotplot(b ~ a)   # y-axis is a b c d (from bottom to top)

> # Now re-order levels of b to be in order of a
> b ← reorder.factor(b, a)
> b
[1] a c b d

> levels(b)
[1] "a" "d" "b" "c"

> dotplot(b ~ a)  # y-axis is a d b c (bottom to top)
```

This places the dots in ascending order from bottom to top. To put them into descending order, use instead

```
> b ← reorder.factor(b, -a)
```

You can also order factor levels by another variable, and if the data are not already grouped, by the value of a summary statistic computed after grouping. `reorder.factor` creates an ordered variable and trellis functions respect the order of the levels of such variables (see the `ordered` function).

Most of the `trellis` functions accept a `groups` argument that is used in conjunction with the `panel.superpose` function to plot different classes of points in the same plot (superposition), distinguishing them by different symbols or colors. For example, to plot `age` vs. `height` on one plot using different color symbols for females and males we might use the command

```
xyplot(height ~ age, groups=sex, panel=panel.superpose)
```

This can be enhanced by allowing the user to control the plotting symbols and other characteristics. In what follows we plot males using an x (`pch=2`) and females using a triangle (`pch=4`). We use the `key` option to place a legend on top of the plot.

```
s ← trellis.par.get('superpose.symbol')
s$pch[1:2] ← c(2,4)   # replace first 2 elements with 2,4; ignore others
trellis.par.set('superpose.symbol',s)   # replace trellis default pch
xyplot(height ~ age, groups=sex, panel=panel.superpose,
       key=list(text=list(c('female','male')), points=Rows(s,1:2)))
```

If you want the key to have one row, specify `columns=2` inside the `key` list.

By combining `trellis`' `xyplot` function with Hmisc's `panel.plsmo` function, more flexibility is obtained and keys can be created to define multiple groups of data points on one `trellis` panel. By default, both raw data and nonparametric trend estimates are graphed. The previous example could be done using

```
xyplot(height ~ age, groups=sex, panel=panel.plsmo, type='p')
Key()
```

Here `type='p'` caused only points to be drawn. The default, `type='b'` causes both raw data points and `lowess`–smoothed trend lines to be drawn, and the different curves are labeled by the group names where the curves are furthest apart. Specify `type='l'` to omit the raw data points.

Here are some other examples:

```
# Plot points and smooth trend line (add type='l' to suppress points)
xyplot(blood.pressure ~ age, panel=panel.plsmo)

# Do this for multiple panels
xyplot(blood.pressure ~ age | sex, panel=panel.plsmo)

# Do this for subgroups of points on each panel, show the data
# density on each curve, and draw a key at the default location
xyplot(blood.pressure ~ age | sex, groups=race, panel=panel.plsmo,
       datadensity=T)
Key()     # Use Key(locator(1)) to position key with mouse
```

The `Key` function is created by `panel.plsmo` when a `groups` variable is present. `Key` calls the builtin `key` function with suitable arguments for drawing the key, remembering what was specified to `xyplot` in the form of symbols and colors.

Here are some other `trellis` examples.

```
trellis.device()     # not needed for 4.x or later
bwplot(sex ~ age | pclass, data=titanic)
bwplot(sex ~ age | pclass*survived, data=titanic)
bwplot(sex ~ age | pclass, panel=panel.bpplot, data=titanic)
# Uses Hmisc's panel.bpplot function for drawing more versatile box plots
# Result is in Figure 11.11

ecdf(~ age | pclass, groups=sex, q=.5, label.curve=F,
     col=c(1,.4))  # use gray scale for one of the sexes, show medians
# Hmisc ecdf.formula function - Figure 11.12
```

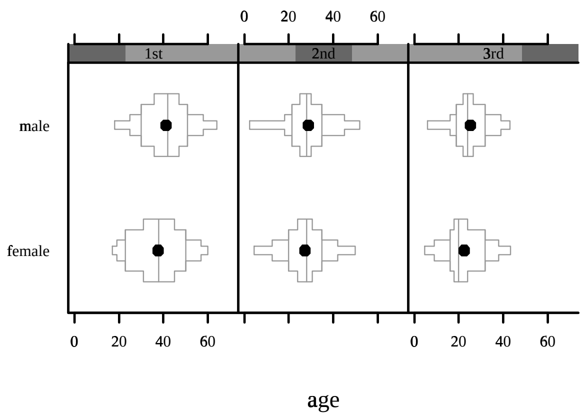

Figure 11.11: *Extended box plot for `titanic` data. Shown are the median, mean (solid dot), and quantile intervals containing 0.25, 0.5, 0.75, and 0.9 of the age distribution.*

```
# Add datadensity='rug', 'hist', or 'density' to augment CDF with
# density information

library(Design,T); attach(prostate)  # prostate is on web page
x ← nomiss(cbind(age,wt,sbp,dbp,hg,sz))
# Using nomiss in Hmisc - splom has a bug in handling missing values
splom(∼ x)
```

`trellis` shades panels which label the current level of conditioning variables. To instead use white backgrounds on these title panels, use the following commands:

```
s.b ← trellis.par.get("strip.background")
s.b$col ← 0
trellis.par.set("strip.background", s.b)
s.s ← trellis.par.get("strip.shingle")
s.s$col ← 0
trellis.par.set("strip.shingle", s.s)
```

This can be also be done by using the command `trellis.strip.blank()`, which calls a little function in Hmisc. `trellis.strip.blank` must be called before the graphics device is opened. Another way to remove shading from the strip is to add the following argument to a top–level `trellis` function:

```
strip=function(...) strip.default(..., style=1)
```

The help file for `strip.default` documents the various values of `style`:

1. the full strip label is colored in background color and the text string for the current factor level is centered in it

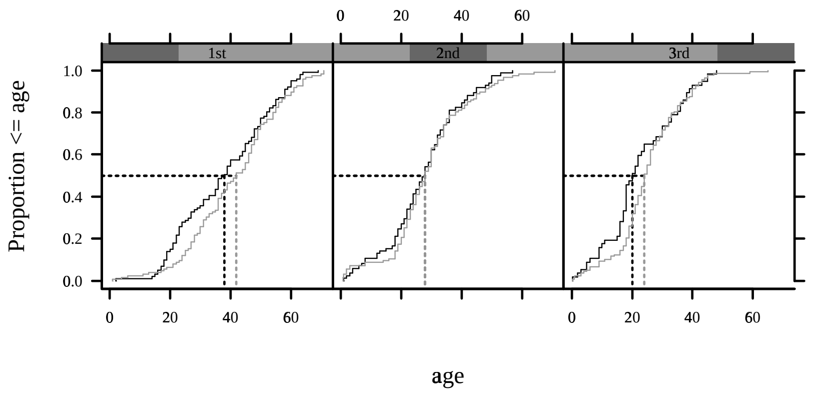

Figure 11.12: *Multi–panel `trellis` graph produced by the Hmisc `ecdf` function.*

2. all the factor levels are spread across the strip with the current level is drawn atop a colored rectangle
3. identical to style 1 but a portion of the strip is highlighted (as in a shingle) to indicate the position of the current level
4. like 2 except the entire strip label is colored in background color
5. like 1 but the current factor level is positioned left-to-right across the strip
6. like 5 but the string adjustment varies from left-justified to right-justified as the string moves left-to-right

The Hmisc `setTrellis` function by default calls `trellis.strip.blank` and sets the line thickness for dot plot reference lines to 1 instead of the usual default of 2. See also the `setps` Hmisc function.

There are many standard options to high-level `trellis` functions (a good reference is Kraus & Olson Section 6.4). If you are making a multi-panel graph that is 2 rows $\times$ 2 columns with one of the four panels unused, you can specify that all 3 panels are to be put in a row. Add `layout=c(1,3)` to put then in one row or `layout=c(3,1)` to put the three panels in one column. You can specify a logical vector `skip` to control where unused panels appear. To make a 2×2 layout with the upper left panel being blank, specify `layout=c(2,2), skip=c(F,F,T,F)`; note the numbering from the origin of the lower left panel. You can arrange and number panels from the upper left by using the `as.table=T` argument.

Trellis graphs are not actually drawn until a `print` function is executed, either explicitly or implicitly. This can be used to great advantage in composing a multi-graph display of different types of Trellis graphs. In the following example we make four graphs and do not draw them immediately. The `print.trellis` is invoked to put the four graphs in the desired locations on the page.

```
plot1 ← xYplot(...)
plot2 ← Dotplot(...)
plot3 ← histogram(...)
plot4 ← xyplot(...)
print(plot1, split=c(1,1,2,2), more=T)
print(plot2, split=c(1,2,2,2), more=T)
print(plot3, split=c(2,1,2,2), more=T)
print(plot4, split=c(2,2,2,2), more=F)
```

The first two arguments to `split` specify the column and row number (from the lower left) the current graph should occupy. The last two arguments specify the overall number of columns and rows that are to be set aside.

To have finer control of positioning of sub-graphs you can use the `position` argument which contains fractional values. For details see the help file for `print.trellis`.

To specify axis details, use `scales=`. For example, to specify that 10 tick marks are to appear on the x-axis and 5 on the y-axis, use `scales=list(x=list(tick.number=10),y=list(tick.number=5))`.

Specify `aspect='xy'` to bank panels to 45°.

See the `trellis.args` help file for more information about general `trellis` arguments including `main, sub, page, xlim, ylim, xlab, ylab`. To graphically display all the current `trellis` settings, issue the `show.settings()` command.

When you want to display the data density of one variable stratified by one or more other variables and you don't like cumulative distribution functions or multiple histograms, the `trellis stripplot` function may be of interest, as a generalization of Hmisc's `datadensity` function (Section 11.3). By default, `stripplot` makes a separate horizontal band for each level of the stratification variable and plots small circles at each actual data point (with optional jittering). You can give `stripplot` a `panel` argument to specify other representations. Here are some examples:

```
trellis.device()
# Separate strips of circles by treatment
stripplot(treatment ~ y, subset=!is.na(y))

# Instead, use a rug plot via scat1d
stripplot(treatment ~ y,
          panel=function(x,y,...) scat1d(x,y=y),
          subset=!is.na(y))

# Substitute an estimated density plot for individual points
g ← function(x, y, ...) {
  for(yy in unique(y)) {
    d ← density(x[y==yy], na.rm=T)
    lines(d$x, d$y + yy)
  }
}
stripplot(treatment ~ y, panel=g)
```

A much easier approach to producing the previous graph is to use the `trellis densityplot` function as shown below. A second example uses the `prostate` data frame that is in the Design library.

```
densityplot(∼ y | treatment)
densityplot(∼hg | rx*factor(stage), width=3, data=prostate)
```

See Section 6.1 for an example where a frequency table is constructed for two categorical variables, row percents are computed, and these are plotted using `dotplot`.

You can find out more about `trellis` by visiting MathSoft's Web page `http://www.mathsoft.com/splus.html`.

11.4.1 Multiple Response Variables and Error Bars

The `trellis` `xyplot` function is quite flexible as long as you are plotting a single response variable. Trellis in general requires the response variable to be univariate so there is no opportunity to add, for example, a systolic blood pressure time trend to a diastolic pressure trend on the same panels. There is also no way with `xyplot` of plotting "error bars" such as mean ± 2 standard errors, or the median and outer quartiles. Hmisc's `xYplot` function uses a trick to get around this problem. The trick is that the analyst specifies multiple response variables, but all but the first are converted to become attributes of the first variable, using an auxiliary function `Cbind`. Then a new panel function `panel.xYplot` fetches these attributes as needed.

`xYplot` other advantages over `xyplot`.

1. It automatically goes into "superpose" mode when a `groups` variable is present. No `panel` = `panel.superpose` needs to be specified.

2. `xYplot` produces a function `Key` that makes it easy to plot a key for how the `groups` variable is denoted in the plot.

3. `xYplot` can use the Hmisc `labcurve` function to automatically label multiple curves generated by the `groups` variable.

4. `xYplot` uses variable labels, when they are present, to label axes.

5. `xYplot` can aggregate raw data automatically, given a function that produces a 3-number summary. Numeric x-variables can be collapsed into intervals containing a pre-specified number of observations, and represented by the mean x within the intervals.

Here are some examples taken from the help file. The first several examples draw error bars, then other examples show how to plot multiple curves generated by the multiple response variables, using e.g. `method='band'`. For any plot, you can control whether lines or points are plotted through the use of `type='l'`, `'p'`, or `'b'` (both).

```
# First generate combinations of some variable values
dfr ← expand.grid(month=1:12, continent=c('Europe','USA'),
                  sex=c('female','male'))
attach(dfr)   # to get access to 3 new variables
set.seed(13)  # so values can be replicated
# Add a response variable (monthly mean) to the predictor settings
# using an assumed linear regression model
y ← month/10 + 1*(sex=='female') + 2*(continent=='Europe') +
     runif(48,-.15,.15)
```

```
lower ← y - runif(48,.05,.15)   # Generate hypothetical monthly ranges
upper ← y + runif(48,.05,.15)

# Show mean and range at each month, for one panel
xYplot(Cbind(y,lower,upper) ~ month,subset=sex=='male' & continent=='USA')
# add ,label.curves=F to suppress use of labcurve to label curves where farthest apart

# Now make a panel for each continent, for males
xYplot(Cbind(y,lower,upper) ~ month|continent,subset=sex=='male')

# Make a panel for each continent; within each panel, separate sex
# groups; use Key to automatically place a key
xYplot(Cbind(y,lower,upper) ~ month|continent,groups=sex); Key()

# Separate sex groups within a single panel
xYplot(Cbind(y,lower,upper) ~ month,groups=sex,subset=continent=='Europe')

# Same as above but automatically position labels for sex groups
xYplot(Cbind(y,lower,upper) ~ month,groups=sex,subset=continent=='Europe',keys='lines')
# keys='lines' causes labcurve to draw a legend where the panel is most empty

# Draw 3 lines for the three variables
xYplot(Cbind(y,lower,upper) ~ month,groups=sex,subset=continent=='Europe',method='bands')

# Show error bars once again, but only the upper part
xYplot(Cbind(y,lower,upper) ~ month,groups=sex,subset=continent=='Europe',method='upper')

# Now use a label for y, and alternate using only upper or lower bars
# so bars for different groups don't run into each other
label(y) ← 'Quality of Life Score'
# can also specify Cbind('Quality of Life Score'=y,lower,upper)
xYplot(Cbind(y,lower,upper) ~ month,groups=sex,subset=continent=='Europe',method='alt bars',
       offset=.4)   # offset passed to labcurve to label .4 y units away from curve
```

11.4.2 Multiple x–axis Variables and Error Bars in Dot Plots

The Hmisc `Dotplot` function has three of the four advantages over the builtin `trellis` function `dotplot` that `xYplot` has over `xyplot`. But instead of generalizing the function to allow multiple y–axis variables, `Dotplot` generalizes `dotplot` to allow for several x–axis variables. The main usage of this is displaying confidence or quantile intervals on the horizontal reference lines, in addition to showing point estimates. For example, to turn the last `xYplot` example into a dot plot, use

```
Dotplot(month ~ Cbind(y,lower,upper), groups=sex,
        subset=continent=='Europe')
                   # Cbind(y,lower,upper)|sex may work better
Key()              # Key is generated by Dotplot for the sex variable
```

This will produce a solid line[1] connecting the `lower` and `upper` values for each `month`, with a solid dot indicating the value of `y`. To further emphasize the range rather than the point estimate, specify

[1]The line style is taken from `trellis.par.get('plot.line')`.

`pch=3` (plus sign) as the plotting symbol.

Like `dotplot`, panel variables can be used with `Dotplot` when the formula contains a `|`. `Dotplot` does not fully handle both superposition (using `groups`) and multiple x–variables, as it currently does not use different colors or other line styles to distinguish the low–high line segments for the superposed groups.

See Section 4.7 for examples where `Dotplot` is used for profiling multiple groups, based on means, confidence limits, and ranks.

11.4.3 Using `summarize` with `trellis`

The Hmisc `summarize` function creates a new data frame containing multi–way descriptive statistics. This data frame is suitable for use by all trellis functions. In the next example we generate a dataset containing 24 `month` × `year` combinations with 100 observations per combination. Then we compute 24 medians and 0.025 and 0.975 quantiles to show the center and 0.95 coverage intervals for each stratum.

```
set.seed(111)   # so we can replicate the example
dfr ← expand.grid(month=1:12, year=c(1997,1998), reps=1:100)
attach(dfr)
y ← abs(month-6.5) + 2*runif(length(month)) + year-1997
s ← summarize(y, llist(month,year), smedian.hilow, conf.int=.5)
# To plot only the median we can use any trellis function, e.g.:
xyplot(y ∼ month, groups=year, panel=panel.superpose, data=s)

# But now show all 3 values for each stratum
xYplot(Cbind(y,Lower,Upper) ∼ month, groups=year, data=s,
       keys='lines', method='alt')
# Can also show 3 quantiles
s ← summarize(y, llist(month,year), quantile, probs=c(.5,.25,.75),
               stat.name=c('y','Q1','Q3'))
xYplot(Cbind(y, Q1, Q3) ∼ month, groups=year, data=s, keys='lines')
# To display means and bootstrapped nonparametric confidence intervals use:
s ← summarize(y, llist(month,year), smean.cl.boot)
s
```

```
 month year    y Lower Upper
     1 1997 6.55  6.44  6.67
     1 1998 7.51  7.40  7.62
     2 1997 5.58  5.47  5.69
     2 1998 6.44  6.33  6.55
     3 1997 4.53  4.42  4.67
     3 1998 5.47  5.37  5.58
     4 1997 3.36  3.26  3.46
     4 1998 4.59  4.49  4.69
     5 1997 2.48  2.36  2.60
     5 1998 3.31  3.22  3.41
     6 1997 1.58  1.47  1.69
     6 1998 2.50  2.38  2.60
     7 1997 1.39  1.28  1.51
     7 1998 2.47  2.36  2.58
     8 1997 2.54  2.43  2.64
     8 1998 3.43  3.32  3.55
     9 1997 3.52  3.42  3.63
     9 1998 4.56  4.45  4.67
    10 1997 4.50  4.39  4.63
    10 1998 5.52  5.41  5.62
    11 1997 5.49  5.37  5.61
    11 1998 6.44  6.34  6.56
    12 1997 6.51  6.39  6.64
    12 1998 7.47  7.37  7.58
xYplot(Cbind(y, Lower, Upper) ~ month | year, data=s)
```

To convert this to a dot plot, use

```
Dotplot(month ~ Cbind(y, Lower, Upper) | year, data=s, pch=3)
```

The combination of `summarize` and `trellis` graphics is also useful for showing empirical results when the number of points is too large to make an interpretable scatterplot (especially when stratified by categories of a third variable). In what follows we compute the three quartiles of `height` stratified by `age` and `sex` simultaneously. We either round `age` to the nearest year, or group it into deciles, to have sufficient sample sizes in each `age` × `sex` stratum. For grouping `age` into deciles, we use a feature of the Hmisc `cut2` function in which the levels for the decile groups are the mean values of `age` within each decile group. We have to go to extra trouble to convert the `factor` variable created by `cut2` to a numeric variable.

```
ageg ← round(ageg)  # or:
ageg ← as.numeric(as.character(cut2(age,g=10,levels.mean=T)))
# Also see the m= argument to cut2
s ← summarize(height, llist(sex,age=ageg), smedian.hilow,
               conf.int=.5)  # 3 quartiles named height,Lower,Upper
xYplot(Cbind(height,Lower,Upper) ~ age, groups=sex, method='bands',
       data=s)
```

This process has been automated in `xYplot`:

```
xYplot(height ~ age, groups=sex, method='quantiles')
```

Here `method='quantiles'` runs `cut2` and `summarize` on the raw data to produce the summarized data. By default, this will, for each sex, group age into intervals containing `min(40, n/4)` observations where `n` is the number of observations in that sex group. You can override this using the `nx` argument to `xYplot`. You can also specify the vector of 3 quantiles to compute, in a `probs` argument. The central quantile needs to be listed first. `method` can also be the name of a function that returns a matrix containing, in order, a measure of central tendence and some sort of limits. For example:

```
xYplot(y ~ month | year, nx=F, method=smean.cl.boot)
```

displays the mean `y` and bootstrap confidence limits, stratified by unique values of `month` (with no intervals, because `nx=F` was specified). You can specify instead `nx=m` where m is the number of observations to achieve in each automatically-created x interval.

See Section 6.1 for an example where row percentages are computed from a frequency table, and then displayed using `trellis` graphics. Section 4.7 has other examples for `summarize` and `Dotplot`.

If the data frame is organized so that the multiple variables to plot are in separate rows, the Hmisc `reShape` function may be useful for reorganizing the data for plotting. Here is an example where for each `Department` there is a row for each of three salary levels: low, middle, and high. A variable named `type` tells what each row pertains to. `reShape` will create a matrix with columns named `low,middle,high`. Here we assume the `levels` of `type` are in the order `middle, low, high`. indexhlabel

```
label(salary) ← 'Salary, $'
a ← reShape(salary, id=Department, colvar=type)
dept ← dimnames(a)[[1]]
Dotplot(dept ~ Cbind(a))
```

Note that when there is a single argument to `Cbind` and that argument is a matrix, `Cbind` will pull off the first column as the main variable and "hide" the other columns as an attribute to the main variable.

11.4.4 A Summary of Functions for Aggregating Data for Plotting

Various functions in S can be used to compute aggregate statistics with stratification that can be passed to various plotting routines. A summary of these functions is below.

`tapply`: This function will stratify a single variable by one or a list of stratification variables. When you stratify by more than one variable, the result is a matrix which is generally difficult to plot directly. The Hmisc `reShape` function can be used to re–shape the result into a data frame for plotting. When you stratify by a single variable, `tapply` creates a vector of summary statistics suitable for making a simple dot or bar plot without conditioning.

`aggregate`: The function takes as input a vector or a data frame and a `by` list of one or more stratification variables (see p. 88). It is handy to enclose the `by` variables in the `llist` function. You can summarize many variables at once but only a single number such as the mean is computed for each one. `aggregate` does not preserve numeric stratification variables — it transforms them into factors which are not suitable for certain graphics. The result of `aggregate` is a data frame for printing or plotting.

`summary.formula`: This Hmisc function by default will compute separate summaries for each of the stratification variables. It can also do cross–classifications when `method='cross'`. You can summarize the response variable using multiple statistics (e.g., mean and median) and if you specify a `fun` function that can deal specially with matrices, you can summarize multiple–column response variables. `summary.formula` creates special objects and has special plotting methods (e.g., `plot.summary.formula.response`) for plotting those objects. In general you don't plot the results of `summary.formula` using one of the trellis functions.

`summarize`: This Hmisc function has a similar purpose as `aggregate` but with some differences. It will summarize only a single response variable but the `FUN` function can summarize it with many statistics. Thus you can compute multiple quantiles or upper and lower limits for error bars. `summarize` will not convert numeric stratifiers to factors, so `summarize` is suitable for summarizing data for `xyplot` or `xYplot` when the stratification variable needs to be on the x–axis. `summarize` only does cross–classification. It will not do separate stratifications as the `summary.formula` function does. Unlike `summary.data.frame`, `summarize` creates an ordinary data frame suitable for any use in S, especially for passing as a `data` argument to trellis graphics functions. You can also easily use the GUI to graph this data frame.

`method=function` with `xYplot`: This automatically aggregates data to be plotted when central tendency and upper and lower bands are of interest.

Chapter 12

Controlling Graphics Details

S has the capabilities to produce very complex and detailed graphical summaries. Loosely speaking, there are three levels of complexity in the elements comprising a plot. The first level consists of commands that can produce a plot by themselves. They set up a coordinate system for us and automatically determine the size of the plot, margins, orientation, font, plotting characters and the box surrounding the plot. They are called *high level* plotting functions. They can be described as functions that will produce results with a single call to them. Examples of high level plotting functions include `plot`, `hist` and `boxplot`. The next level allows to add detail to the plot by including other elements such as lines, symbols, legends, draw polygons, etc. These kind of functions are called *low level* plotting functions. They are functions whose output is added to a currently active graphics device. Finally, the greatest detail and control over your graphics can be exercised through the use of graphics parameters. Some of them can only be used in high level plotting functions; they are called *high level parameters.* Others can be used in high level functions or through the function `par`; these are classified as *general parameters.* There are also *layout parameters* which can only be changed through `par` because they change the overall layout of plots or figures. The last category of parameters are *information parameters* which cannot be changed but can be queried through `par`.

Table 12.1 summarizes low–level plotting commands for taking charge of details of how plots are drawn.

The undocumented Hmisc function `pstamp` uses the `stamp` function to date/time stamp an existing plot or multi–image plot. Under UNIX, `pstamp` can optionally stamp the plot with the current project directory name. Additional user–specified text (the first argument) can be specified; this is used as a prefix to the stamp. Unlike `stamp`, `pstamp` uses very small letters so as to not obstruct the rest of the graph.

12.1 Graphics Parameters

The function `par()` with no arguments returns a list. We list below the names of all the parameters in alphabetical order. In order for `par()` to work, a graphics device should be active.

Table 12.1: *Low Level Plotting Functions*

Function	Description
`abline`	add straight line to plot
`arrows`	draw arrow
`axes`	add axis label
`axis`	add custom axis
`box`	add box to plot
`character.table`	show special text symbols (Hmisc)
`frame`	advance to next figure
`labelclust`	add labels to cluster plot
`legend`	add legend to plot
`lines`	add lines
`minor.tick`	add minor tick marks (Hmisc)
`mtext`	add text in margins
`mtitle`	add titles and subtitles to a multiple image plots (Hmisc)
`perspp`	project points on perspective plots
`points`	add points
`polygon`	draw and shade polygons
`pstamp`	date/time stamp current plot (Hmisc enhancement of `stamp`)
`qqline`	draw median line on `qqnorm` plot
`rug`	add data-based marks to an axis
`segments`	draw disconnected line segments
`show.pch`	show plotting characters (Hmisc)
`stamp`	add a time stamp to a plot
`symbols`	draw symbols on a plot
`text`	add text
`title`	add title or axis labels

```
> names(par())
 [1] "1em"  "adj"  "ask"  "bty"  "cex"  "cin"  "col"  "cra"  "crt"  "csi"
[11] "cxy"  "din"  "err"  "exp"  "fig"  "fin"  "font" "frm"  "fty"  "lab"
[21] "las"  "lty"  "lwd"  "mai"  "mar"  "mex"  "mfg"  "mgp"  "new"  "oma"
[31] "omd"  "omi"  "pch"  "pin"  "plt"  "pty"  "rsz"  "smo"  "srt"  "tck"
[41] "uin"  "usr"  "xaxp" "xaxs" "xaxt" "xpd"  "yaxp" "yaxs" "yaxt"
```

To change one or more of the parameters we pass them to `par` as arguments with their new values. For instance, to change the default plotting symbol from `"*"` to `"+"` and setup a matrix of plots with two rows and three columns, we would type

```
> par(mfrow=c(2,3),pch=3)
```

The statements above, did not only change the value of `mfrow` and `pch` but also returned *invisibly* a list containing the original values of the parameters that we changed. Thus, if we were going to assign the statement, we would get a list with these original values. This can be useful to restore the parameters to its previous values.

```
> par.old ← par(mfrow=c(2,3),pch=3)
> par.old
$mfrow:
[1] 1 1

$pch:
[1] "*"

> par(par.old)
> par()$mfrow
NULL
> par()$pch
[1] "*"
```

The value of some parameters may change when you change another. The parameter `cex` for example, which controls character expansion relative to the device size, is related to the `mfrow` parameter. When you change `mfrow`, `cex` will be set automatically, so that the character size is not too big for the number of plots in the screen. You can still change `cex` to be whatever you like. You just have to do it after you set `mfrow` and in a different `par`. The reason for doing it in a different `par` is that `cex` is a general graphics parameter and `mfrow` is a layout parameter. S sets general parameters first and then layout parameters.

```
> par(mfrow=c(3,2))
> par(cex=0.75)  # NOT par(mfrow=c(3,2),cex=0.75)
```

12.1.1 The Graphics Region

To understand what each parameter does it is necessary to visualize how S divides up the device surface. We have an *outer margin*, inside of which we find the *figure region.* This region contains one or more *plot areas* surrounded by a *margin.*

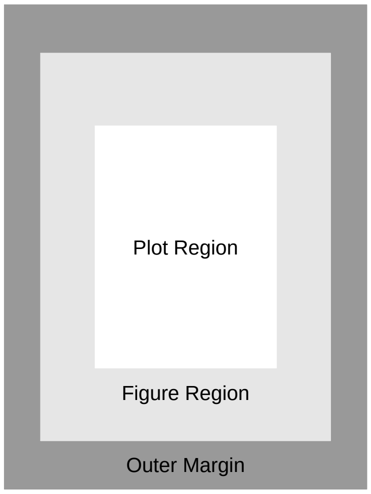

Figure 12.1: *Plot Region*

By default the device is initialized with zero area in the outer margin. Typically the axis line is drawn in the border between the plot region and the margin. If we change the size of one of the regions, the others are adjusted automatically.

12.1.2 Controlling Text and Margins

The parameters to control the size of the outer margin are `oma`,`omi` and `omd`. For the figure margin we use `mar` and `mai`. Related to all of them is `mex`. The parameters `fig` and `fin` control the physical size of the figure region, while `plt` and `pin` do likewise for the plot region. Here is a description of all of them

```
fig=c(x1,x2,y1,y2)
       coordinates of the current figure region  expressed  as  a
       fraction  of  the  device  surface.   This is dependent on
       mfrow and mfcol.

fin=c(w,h)
       width and height of figure in inches.

mai=c(xbot,xlef,xtop,xrig)
       margin size specified in inches.  Values given for bottom,
       left, top, and right margins in that order.

mar=c(xbot,xlef,xtop,xrig)
       lines of margin on each side of plot.  Margin  coordinates
```

```
        range  from  0  at the edge of the box outward in units of
        mex sized characters.  If the margin is respecified by mai
        or  mar,  the plot region is re-created to provide the ap-
        propriate sized margins within the  figure.   The  default
        value is c(5,4,4,2)+.1.  Problems with lines not appearing
        on some devices  might  be  remedied  by  specifying  non-
        integer values in mar.

mex=x   the coordinate unit for  addressing  locations  in  the
        margin  is  expressed  in terms of mex. Margin coordinates
        are measured in terms of characters of size cex  equal  to
        mex.  mex does not change the font size - it merely states
        which font is to be used to measure the margins.

oma=c(xbot,xlef,xtop,xrig)
        outer margin lines of  text.   oma  provides  the  maximum
        value  for  outer  margin  coordinates on each of the four
        sides of the multiple figure region.  oma  causes  recrea-
        tion of the current figure within the confines of the new-
        ly specified outer margins.  The default is rep(0,4).  See
        mtext to create titles in the outer margin.

omd=c(x1,x2,y1,y2)
        coordinates of the outer margin region as  a  fraction  of
        the device surface.

omi=c(xbot,xlef,xtop,xrig)
        size of outer margins in inches.

pin=c(w,h)
        width and height of plot, measured in inches.

plt=c(x1,x2,y1,y2)
        the coordinates of the plot region measured as a  fraction
        of the figure region.
```

By default we start with zero area for the outer margin. We can change it by changing `oma`, `omd` or `omi`. It is easiest to work with `oma` since it measures relative sizes; namely, the number of lines of text that we want to have in the outer margin. The height of the lines is measured in units of `mex` which is just the size of the default font. Thus if `oma` is `c(0,0,5,0)` and `mex` has the default value of one, it means that we are leaving room for zero lines of text in the bottom, right and left margin and 5 lines of text of the default size at the top. This does not mean that the text itself that we type has to be of the default size. The size of text is determined by the value of `cex` (character expansion). If `cex` equals 2.5, then we can only fit two lines of text at the top. Of course changing `oma` changes some of the other parameters. The physical size of the figure and plot regions will be reduced. However the space for margins in the figure region will remain the same.

Let us look at an example. We list below the default values of the parameters mentioned above.

```
> par(Cs(fig,fin,mai,mar,mex,cex,oma,omd,omi,pin,plt))
$fig:
```

```
[1] 0 1 0 1

$fin:
[1] 8.00 6.32

$mai:
[1] 0.714 0.574 0.574 0.294

$mar:
[1] 5.1 4.1 4.1 2.1

$mex:
[1] 1

$cex:
[1] 1

$oma:
[1] 0 0 0 0

$omd:
[1] 0 1 0 1

$omi:
[1] 0 0 0 0

$pin:
[1] 7.132 5.032

$plt:
[1] 0.0717500 0.9632500 0.1129747 0.9091772
```

Now, let us change the value of `oma` to allow space for 5 lines of text in the default size of the default font (`mex=1`).

```
> par(oma=c(0,0,5,0))
> par(Cs(fig,fin,mai,mar,mex,cex,oma,omd,omi,pin,plt))
$fig:
[1] 0.0000000 1.0000000 0.0000000 0.8892405

$fin:
[1] 8.00 5.62

$mai:
[1] 0.714 0.574 0.574 0.294

$mar:
[1] 5.1 4.1 4.1 2.1

$mex:
```

```
[1] 1

$cex:
[1] 1

$oma:
[1] 0 0 5 0

$omd:
[1] 0.0000000 1.0000000 0.0000000 0.8892405

$omi:
[1] 0.0 0.0 0.7 0.0

$pin:
[1] 7.132 4.332

$plt:
[1] 0.0717500 0.9632500 0.1270463 0.8978648
```

We see that all the parameters have changed with the exception of `mai`, `mar`, `mex` and `cex`. Let us now change `mar` and `cex` to allow for only one line of text of size 2.5 in the top margin figure.

```
> par(mar=c(0,0,2.5,0))
> par(cex=2.5)
> mtext("A Title in the Figure Margin",side=3)
> mtext("A Title in the Outer Margin",side=3,outer=T,line=2.5)
> mtext("Another Title in the Outer Margin",side=3,outer=T)
> text(0.5,0.5,"This is the qPlot Region",cex=1)
> box()
```

Look at the help file for an explanation on the use of `mtext`. Also notice that we used `text` with a pair of coordinates despite the fact that there was no plot in the graphics surface. The reason we could do this is that S sets up a coordinate system as soon as you open a graphics device. The default coordinates are `c(0,1,0,1)` as determined by`par("usr")`. If we were going to issue a high level plotting command this coordinates would be set according to the range of your data.

One other layout command that we can look at is `pty`. The value of `pty` is a character string: `"s"` for a square plotting region, and `"m"` for a maximal region.

12.1.3 Controlling Plotting Symbols

We can specify the type of line to be used and its width with the parameters `lty` and `lwd`. The plotting symbol used by default is a ".". We can change it to any character we want by specifying `pch="c"`, where "c" is any character. We may also use a number from 0 to 18 instead of a character to obtain a variety of symbols. Other numbers up to 252 yield characters that are font and device dependent as explained in the section of the help file below. If we use a character as a plotting symbol its size will be determined by the value of `cex` just as in the case of text. If we use a special symbol (by using the `pch=n` form) the size of the symbol will be given by `mkh`. The value of `mkh` is a non-negative number giving the height in inches of the symbol. A value of zero for `mkh` (the default) means that the symbol will be of approximately the same size as a capital letter according to `cex`.

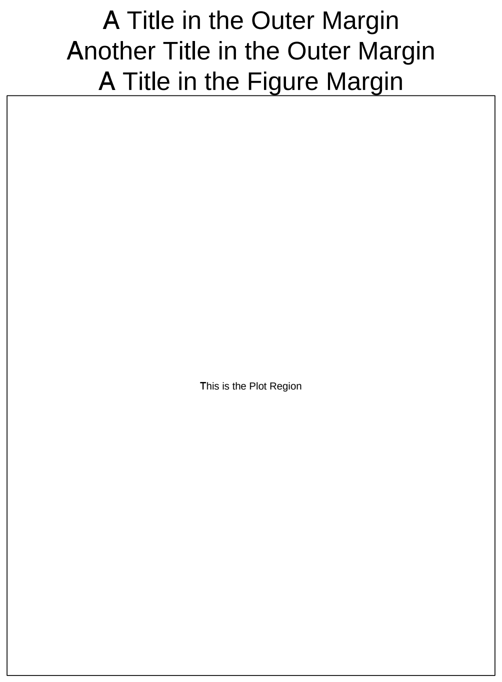
A Title in the Outer Margin
Another Title in the Outer Margin
A Title in the Figure Margin
This is the Plot Region

Figure 12.2: *Handling text in margins*

```
lty=x     line type, device dependent.  Normally type 1 is solid,
       2  and  up  are dotted or dashed.  A few devices have only
       one line type.

lwd=x     line width, device dependent.  Width 1 is the  standard
       width  for  the  device.   Many devices cannot change line
       width.
mkh=x     height in inches of mark  symbols  drawn  when  pch  is
       given  as a number.  The default value of 0 means that the
       cex parameter controls the size of symbols when pch  is  a
       number  (the symbol is approximately the size of a capital
       letter in this case).

pch="c"   the character to be used for plotting points.   If  pch
       is a period, a centered plotting dot is used.

pch=n     the number of a plotting symbol to be drawn when  plot-
       ting  points.   Basic  marks are: square (0); octagon (1);
       triangle (2); cross (3); X (4); diamond (5)  and  inverted
       triangle  (6).  To  get superimposed versions of the above
       use the following arithmetic(!): 7==0+4;  8==3+4;  9==3+5;
       10==1+3; 11==2+6; 12==0+3; 13==1+4; 14==0+2.  Filled marks
       are square (15), octagon (16), triangle (17), and  diamond
       (18).   Use the mkh graphics parameter to control the size
       of these marks.  See the EXAMPLES section for a display of
       the  plotting  symbols.  Using the numbers 32 through 126
       for pch yields the 95 ASCII characters from space  through
       tilde  (see the SPLUS data set font).  The numbers between
       161 and 252 yield characters, accents, ligatures, or noth-
       ing, depending on the font (which is device dependent).
```

You may use the code below to produce a graph of the different plotting symbols and line types, then print a copy to have as a reference. Then you could do something similar to look at the effects of changing the `mkh` and `lwd` parameters.

```
> # A comprehensive pch table can be obtained using the Hmisc show.pch function
> par(usr=c(-1,19,0,1))
> for(i in 0:18)
> points(i,.5,pch=i)
> text(i,.35,i)
> title('Samples of "pch=" Parameter')
> box()
> par(usr=c(-1,11,0,11))
> for(i in 1:10)
> abline(h=i,lty=i)
> text(5,i+.5,paste("lty=",i,sep=""))
> box()
> title('Examples of "lty"')
```

The Hmisc `character.table` function written by Pierre Joyet shows the numeric equivalents of all latin characters, facilitating the use of special characters in graph titles and other character

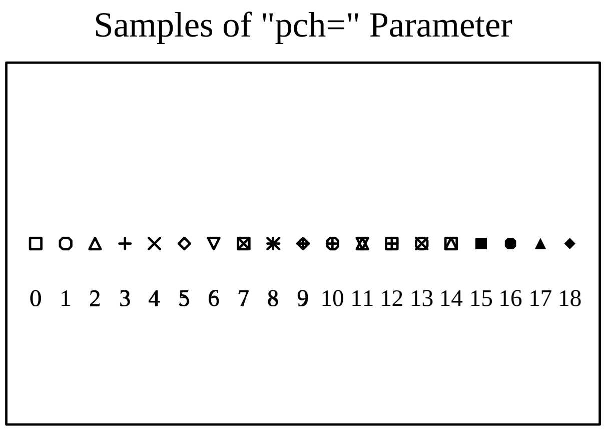

Figure 12.3: *Plotting Symbols*

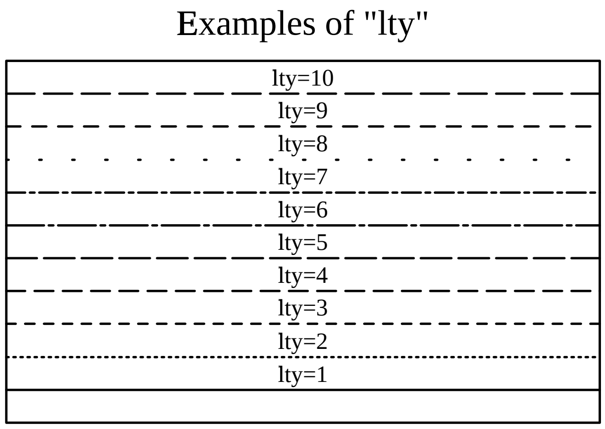

Figure 12.4: *Different Types of Lines*

strings. Typing `character.table(8)` for example will show the characters in font 8. From that you will see that ρ is equivalent to 162. So `title('\162', font=8)` will write a title of ρ.

12.1.4 Multiple Plots

To construct a plot with several figures in it we use the parameters `mfrow=c(m,n)` or `mfcol=c(m,n)`. These set up a matrix of plots with `m` rows and `n` columns and the plots are drawn row-by-row or column-by-column. Setting one implies setting the other, to the same value. To know the order in which to do the plots, S looks at `fty` which will have the value `"r"` (rows) or `"c"` (columns) depending on which parameter was set. If the number of rows or columns is greater than 2 then `cex` and `mex` are set to 0.5.

12.1.5 Skipping Over Plots

We will now examine the function `frame`. We can use it to cause the graphics driver to advance to the next frame, that is, the next plot. If we have only one plot per page, the command `frame()` will erase the current plot (because it is moving to the next frame). In a multiple figure layout, a call to `frame` will move to the next figure. This provides an alternative way to skip over one or more plots in the layout. Two successive calls to `frame` will skip over the next figure.

The parameter `new` on the other hand, is a logical parameter whose purpose is to determine whether or not a high level function will move to the next figure or overlay the current one. If `new=T` it is assumed there are no plots in the current figure and therefore the canvas *will not* be erased when we call a high level function. If `new=F` then a call to a high level function will cause the graphics device to move to the next figure in order to avoid overwriting the current one. After executing a high level graphics command `new` is immediately set to `F`. This the normal situation; there are no plots in the current figure, we do a plot (`new` is set to `F`), add lines, text or whatever is necessary; once we are finished with this particular plot we want to move to a new one. Since `new` is `F` we only need to call the high level function and it will start a new plot without overwriting the current one. In some circumstances we may want to overlay the results of two high level plotting functions. In this case we type `par(new=T)` before executing the second one and its output will be overlayed on top of the output of the first function. We will have more examples about this later.

A side effect of the way `new` operates is that when setting up the device layout, if there is more than one plot per page, the current figure is set to be the last one in the layout and `new` is set to `F`. This way, calling a high level plotting command will cause the device to move to the next figure in the layout, i.e. the first one, correctly producing the plot. The problem is that if we try to execute a *low level* plotting command, the results will apply to the last figure in the layout which is probably not what we intended.

12.1.6 A More Flexible Layout

Other parameters that change when setting `mfrow` or `mfcol` are `fig` and `mfg`. Notice that the `mf*` parameters divide the screen in regions of *the same size.* Additional flexibility is possible by using `mfg` and `fig`. In fact, setting one of `mfrow` or `mfcol` changes both of these. `mfg` allows under certain conditions to expand a particular plot to fill a whole row or a whole column. `fig` is much more flexible and permits creative arrangements of the different plots.

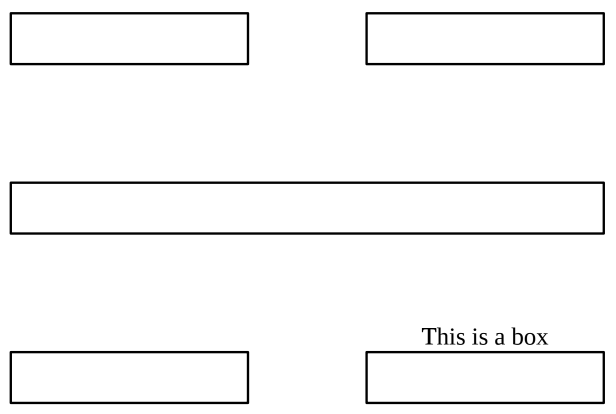

Figure 12.5: *Flexible layout using `mfg`*

The form of `mfg` is `c(i,j,m,n)`, where `i` and `j` denote the row and column of the *current* figure in the multiple figure layout and `m` and `n` are the number of rows and columns. Thus, `mfg` can be used to make a specific figure active. The way to use `mfg` is to plan our layout and then after each plot change its value to make the next figure span a different region. For example

```
> par(mfg=c(1,1,3,2))
> box()
> par(mfg=c(1,2,3,2))
> box()
> par(mfg=c(2,1,3,1))
> box()
> par(mfg=c(3,1,3,2))
> box()
> par(mfg=c(3,2,3,2))
> box()
> title("This is a box")
```

The `fig` parameter allows greater flexibility than `mfg`. Simply set the coordinates of the figure region as a fraction of the device surface before each plot. The example above could have been

```
> par(fig=c(0,.5,.66,1))
> par(new=F)
> box()
> frame()
> par(fig=c(0,.5,.66,1))
> box()
> par(fig=c(.5,1,.66,1))
> box()
> par(fig=c(0,1,.33,.66))
> box()
> par(fig=c(0,0.5,0,.33))
> box()
> par(fig=c(0.5,1,0,.33))
> box()
> title("This is a box")
```

It is clear that by varying the parameters in `fig` we can obtain a much more flexible layout than by using `mfg` alone. A combination of both is most efficient, but setting `fig` will leave `mfg` unchanged. Also notice that the `title` function only affects the current figure. If we want to use an overall title we need to use `mtext`. There is an Hmisc function called `mtitle` which can also write an overall title. Look at it to see how it uses `mtext`.

12.1.7 Controlling Axes

With one exception, the parameters used to control the axes are all general parameters. That is, they can be changed as part of the argument to a high level plotting function, or through `par`. The exception is `axes` which is a high level parameter and can only be changed in the call to a high level plotting function. Using said parameters we can control four aspects of the axes: whether or not to draw an axis at all, the axis style (meaning, in general, the range), the style and positioning of labels, and length and position of tick marks.

We will now examine `axes`:

```
axes=L    if FALSE, suppresses all axis plotting (x, y  axes  and
       box).  Useful  to  make a high-level plotting routine gen-
       erate only the plot portion of the figure.
```

If we choose to set this parameter to `F`, we may add a custom axis later by means of the `axis` function. It is also possible to eliminate the plotting of only one axis by using `xaxt` or `yaxt`. Setting either of these to `"n"` will produce that result. Other possibilities for these parameters are `"s"` (standard axis), `"t"` (time) and `"l"` (logarithmic).

The axis labels are modified through the parameters `mgp`, `exp`, `lab` and `las`. These are described below.

```
exp=x     if exp=0, then axis labels in exponential notation have
       the "e" and the exponent on a newline.  If exp is equal to
       1, then such numbers are written all on  one  line.   When
       exp=2  (the default), then numbers are written in the form
       2*10^6.

lab=c(x,y,llen)
       desired number of tick intervals on the x and y  axes  and
       the  length  of  labels  on  both  axes.   The  default is
       c(5,5,7).

las=x     style of axis labels.  0 = always parallel to axis (the
       default),  1 = always horizontal, 2 = always perpendicular
       to axis.
```

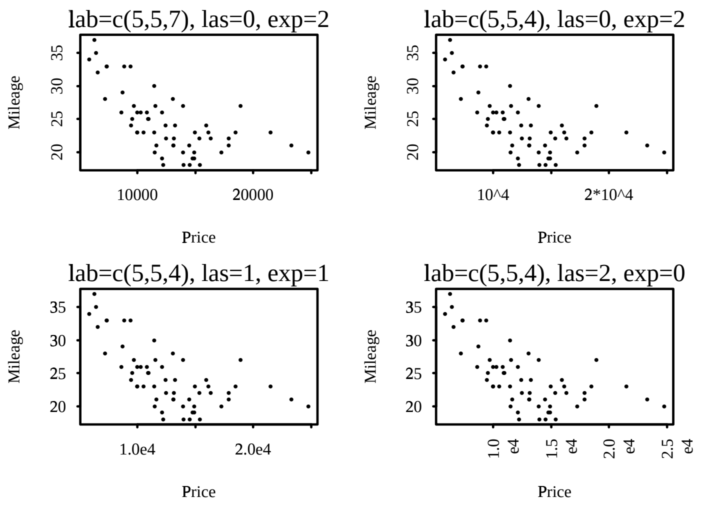

Figure 12.6: *Controlling Axis Labels Style*

```
mgp=c(x1,x2,x3)
       margin line for the axis title, axis labels, and axis line
       in  units  of  mex  (see below).  The default is c(3,1,0).
       Larger numbers are farther from the plot region,  negative
       numbers are inside the plot region.
```

The next example from the `car.test.frame` dataframe illustrates the use of `lab`, `las` and `exp`.

```
> attach(car.test.frame)
> par(mfrow=c(2,2))
> plot(Price,Mileage,main="lab=c(5,5,7), las=0, exp=2")
> plot(Price,Mileage,main="lab=c(5,5,4), las=0, exp=2",
+      lab=c(5,5,4),las=0)
> plot(Price,Mileage,main="lab=c(5,5,4), las=1, exp=1",
+      lab=c(5,5,4),las=1,exp=1)
> plot(Price,Mileage,main="lab=c(5,5,4), las=2, exp=0",
+      lab=c(5,5,4),las=2,exp=0)
```

The mode of axis interval calculation can be controlled individually for the `x` and `y` axis by means of `xaxs` and `yaxs`. The value for these parameters can be any of `"r"`, `"i"`, `"e"`, `"s"` or `"d"`. A description of what they mean follows

```
xaxs="c"    style of axis interval calculation.  The  styles  "s"
       and  "e"  set up standard and extended axes, where numeric
       axis labels are more extreme than any  data  values.   Ex-
       tended  axes  may  be  extended another character width so
       that no data points lie very near the axis  limit.   Style
       "i"  creates  an axis labeled internal to the data values.
```

```
          This style wastes no space, yet still gives pretty labels.
          Style  "r"  extends  the data range by 4% on each end, and
          then labels the axis internally.  This  ensures  that  all
          plots  take  up  a  fixed  percent of the plot region, yet
          keeps points away from the axes.  Style "d"  is  a  direct
          axis,  and  axis parameters will not be changed by further
          high-level plotting routines.  This is used  to  "lock-in"
          an axis from one plot to the next.  The default is "r".

   yaxs="c"    see xaxs.
```

The most useful of these is style "d" which comes in handy when we want to overlay plots.

We already mentioned one of the parameters that control the number of tick marks, `lab`. They can also be controlled individually for each axis using `xaxp` and `yaxp`, but for this we need to set `axes` to `F` or `xaxt` or `yaxt` to `"n"` and then add the axes using `axis`. The length of tick marks is determined by `tck` which can also be used to put grids on the plot.

```
   tck=x     the length of tick marks as a fraction of  the  smaller
          of the width or height of the plotting region if less than
          one-half. When tck is more than one-half,  the  ticks  are
          drawn across that fraction of the side; thus if tck equals
          1, grid lines are drawn.  If tck is  negative,  ticks  are
          drawn outside of the plot region. The default is -.02.

   xaxp=c(ul,uh,n)
          coordinates of lower tick mark ul, upper tick mark uh, and
          number of intervals n within the range from ul to uh.  For
          log axes if uh>1, uh is the number of decades covered plus
          2  and  n is the number of tick marks per decade (n may be
          1, 2, or 9); if uh==1 then n is the upper tick mark.  xaxp
          and  yaxp  are set by high level plotting functions, based
          on the current values in lab (and log)  and  used  by  the
          axis  function  (which  is  called implicitly by most high
          level plotting  functions  unless  you  use  the  argument
          axes=F).

   yaxp=c(ul,uh,n)     see xaxp.
```

Here's an example of different `tck` values.

```
> par(mfrow=c(2,2))
> plot(x,y,main="tck=-0.02")
> plot(x,y,main="tck=0.05",tck=0.05)
> plot(x,y,main="tck=1",tck=1)
> plot(x,y,yaxt="n",main="Different tick marks for each axis")
> axis(2,tck=1,lty=2)
```

The `minor.tick` function in Hmisc makes it easy to add tick marks for minor axis subdivisions.

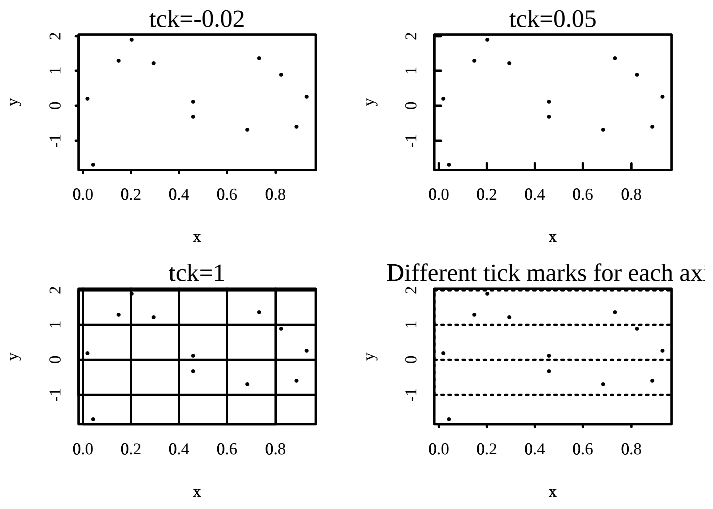

Figure 12.7: *Examples of tick marks*

12.1.8 Overlaying Figures

If we have a plot and we want to overlay different graphics elements on top of it, usually, the simplest way to do it is to use some low-level graphics functions such as `lines` or `points`. Other possibilities include `arrows`, `symbols`, `abline`, `segments`, `matlines`, `matpoints`, and, in the case of time-series plots, `tslines` and `tspoints`. These functions may not work if the new plot is on a different scale than the existing one. We have two main methods of dealing with this situation. One is to use a combination of the function `axis` and the parameter `new`, and the other is to use the function `subplot`. The latter one has other uses as well.

Let us examine in more detail the `axis` function. A section of the help file follows

```
Add an Axis to the Current Plot


DESCRIPTION:
       Adds an axis to the current plot.  The  side,  positioning
       of tick marks, labels and other options can be specified.

USAGE:
       axis(side, at=<<see below>>, labels=T, ticks=T, distn=NULL, line=0,
             pos=<<see below>>, outer=F)

REQUIRED ARGUMENTS:
side:     a number representing the side of the plot for the axis
       (1 for bottom, 2 for left, 3 for top, and 4 for right).

OPTIONAL ARGUMENTS:
at:    vector of positions at which the  ticks  and  tick  labels
```

```
         will  be  plotted.   If  side  is 1 or 3, at represents x-
         coordinates.  If  side  is  2  or  4,  at  represents  y-
         coordinates.  If  at  is  omitted,  the  current axis (as
         specified by the xaxp or yaxp parameters, see par) will be
         plotted.
  labels:   if labels is logical, it specifies whether  or  not  to
         plot  tick  labels.   Otherwise,  labels  must be the same
         length as at, and label[i] is plotted at coordinate at[i].
  ticks:    if TRUE, tick marks and the axis line will be plotted.
  distn:    character string describing the distribution  used  for
         transforming   the   axis   labels.  The  only  choice  is
         distn="normal", in which case values of at are assumed  to
         be probability levels, and the labels are actually plotted
         at qnorm(at).  This also implies a reasonable default  set
         of  values  for the at argument.  By default the values in
         at are used as the labels.

  ...

         Graphical parameters may also be supplied as arguments  to
         this function (see par).  However, arguments to title such
         as xlab and ylab are not allowed.  For string rotation use
         the  las  graphical  parameter: 0 = always parallel to the
         axis (the default), 1 = always horizontal to the axis, 2 =
         always  perpendicular  to  the  axis.   The  srt graphical
         parameter may be ignored for Windows.
```

This is a low-level plotting function (it adds an axis to the existing plot), and graphics parameters can be part of its argument. The only required argument is `side`, which indicates where the axis is going to be drawn, following the usual convention to denote the axes numbers. Most commonly, the `at` argument is used to specify the position of the tick marks, and the `labels` argument determines how they are going to be labeled. The labels justification and style are given by the parameters `srt` and `adj`. However, if `at` is not specified, then the values of `las` gives the orientation of labels. They will be centered at the tick mark if parallel to the axis, and right or left justified if perpendicular to it; left justified if inside the plot, right justified if outside, as determined by `mgp`. In this case `srt` and `adj` are ignored. Example

```
> fahrenheit  ←  c(25, 28, 37, 49, 59, 69, 73, 71, 63, 52, 42, 29)
> plot(fahrenheit, axes=F, pch=12, xlab="", ylab="Fahrenheit",
+      sub="Monthly Mean Temperatures for Hartford, Conn.")
> axis(2)
> axis(1, at=1:12, labels=month.abb)
> celsius  ←  pretty((range(fahrenheit)-32)*5/9)
> axis(side=4, at=celsius*9/5+32, lab=celsius, srt=90)
```

Notice that there is no box and the axis style is somewhat different from what we are used to see. The reason for the absence of a box is that we set `axes=F` in the call to plot, which not only suppressed the axes but also the box. If we wanted to have a box we may easily do so by typing `box()`. On the other hand, we may want to have the old axis style and no box. One quick and easy fix is to look at the values of `usr` and draw vertical lines using `abline`.

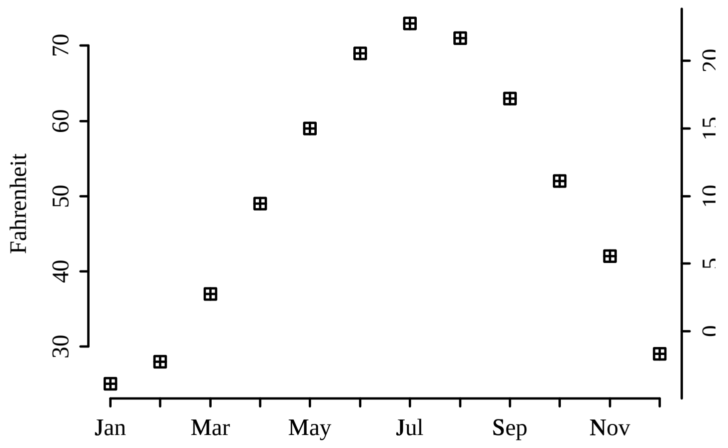

Figure 12.8: *Use of axis*

```
> par("usr")
[1]  0.56 12.44 23.08 74.92
> abline(v=0.56)
> abline(h=23.08)
```

If we want to be purists however, and get the right axes style from the beginning, we would just suppress the plotting of the x-axis by setting `xaxs` to `"n"` and change the box style with `bty="l"`. The calls to `axis` as shown will give us the correct results.

```
> fahrenheit  ←  c(25, 28, 37, 49, 59, 69, 73, 71, 63, 52, 42, 29)
> plot(fahrenheit, xaxt="n", pch=12, xlab="", ylab="Fahrenheit",
+      sub="Monthly Mean Temperatures for Hartford, Conn.",bty="l")
> axis(1, at=1:12, labels=month.abb)
> celsius  ←  pretty((range(fahrenheit)-32)*5/9)
> axis(side=4, at=celsius*9/5+32, lab=celsius, srt=90)
```

Let us now use `axis` in combination with `xaxs` will also be handy here. We begin by changing the right margin to add space for an axis title there.

```
> par(mar=c(5,4,4,5)+.1)
> tsplot(hstart,ylab="Housing Starts")
```

Next we set `par(new=T)` in order not to erase the plot with a new call to `tsplot` and also `par(xaxs="d")` to retain the x-axis from the previous plot.

```
> par(new=T,xaxs="d")
> tsplot(ship,axes=F,lty=2)
> axis(side=4)
> mtext(side=4,line=3.8,"Manufacturing (millions of dollars)")
```

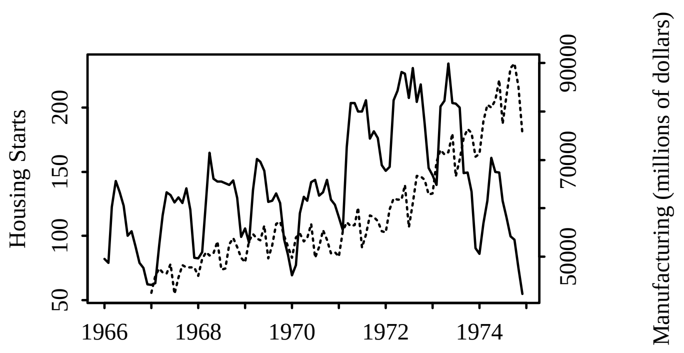

Figure 12.9: *Overlaying high-level plots*

The basic form of the `subplot` function is `subplot(fun,x,y,size=c(1,1))`. `fun` is any plotting routine that we want executed; `x` and `y` are the (user) coordinates of the current figure where the new plot will be positioned, and `size` is the size in inches of the new plot. `subplot` returns the values of the graphics parameters that were in effect for the subplot. For example, we could fit a least squares model in the `car.test.frame` and add a boxplot of the distribution of the predictors.

```
> attach(car.test.frame)
> dd ← datadist(car.test.frame);options(datadist="dd")
> f ← ols(Mileage ∼ Type+Disp.)
> plot(f,Disp.=NA,Type=NA,conf.int=F)
> subplot(plot(Type,Disp.,rotate=T,xlab="",cex=.8),c(55,100),c(15,20))
```

This plot allows to look at the distribution of `Disp.` by `Type` at the same time that examine their combined effect on `Mileage`.

More elaborate plots are possible using `subplot`. The next example adds graphical estimates of the density of `Price` and `Mileage` on top of a plot of `Mileage` vs `Price`.

```
> par(usr=c(0,1,0,1))
> o.par ← subplot(plot(Price,Mileage,log="x"),x=c(0,.85),y=c(0,.85))
> # Save the parameters from subplot
> o.usr ← o.par$usr # Save specially the user coordinates
> o.usr
[1]  3.743326  4.418767 17.240000 37.759998
> den.p ← density(Price,width=3000)
> # density returns a list with an estimate of the density of Price.
> den.m ← density(Mileage,width=10) # ditto for Mileage
> subplot(fun=par(usr=c(o.usr[1:2],0,1.04*max(den.p$y)),xaxt="l");lines(den.p);
+ box(),x=c(0,.85),y=c(.85,1))
```

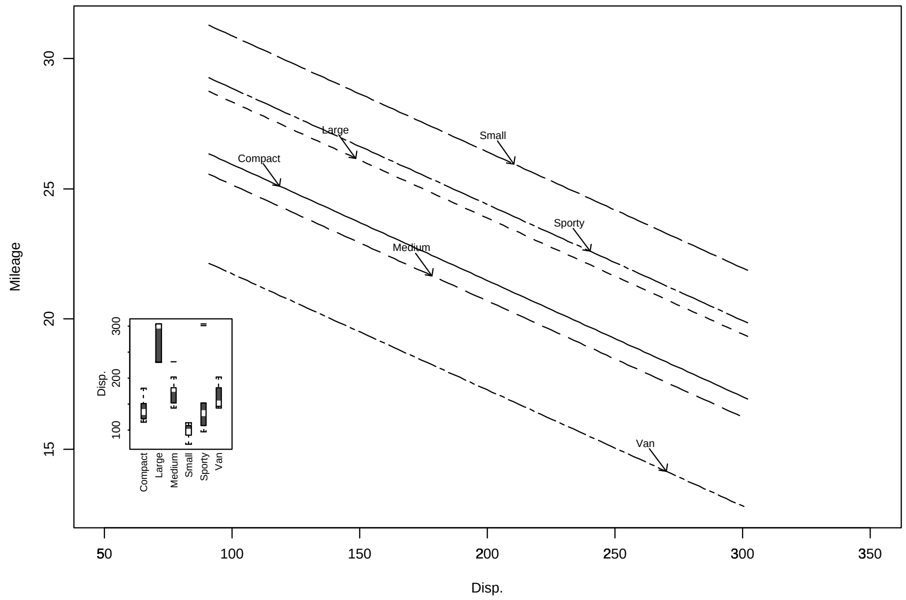

Figure 12.10: *Example of subplot*

```
> # Define the function using a logarithmic axes and plot the density
> subplot(x=c(.85,1),y=c(0,.85),fun=par(usr=c(0,1.04*max(den.m$y),o.usr[3:4]));
+ lines(den.m$y,den.m$x);box()) # same here
```

In this example we have created a plot of `Mileage` vs `Price` using a logarithmic axis for the x-axis. We stored the value of the graphics parameters and later we stored the user coordinates of this plot (which was possible because we used `subplot` rather than `plot`). Then we created two lists with density estimates for `Price` and `Mileage` respectively. The next step is to plot these densities using `subplot`. Note that in the `fun` argument to `subplot` we are defining a function which changes the values of some parameters in the plot that `subplot` is going to create. The user coordinates *y-axis range* in the density plot of `Price` is is 4% bigger than the maximum value of the density estimate, and the *x-axis range* takes its values from the user coordinates from the first subplot. The `xaxt` parameter was set to `"l"` since the first plot also had a logarithmic x-axis. Something similar was done for the last subplot except that the roles of x and y had to be reversed.

12.2 Specifying a Graphical Output Device

Under UNIX, the `printgraph` function can be used to obtain a printed copy of the plot on the screen. In its simplest form, `printgraph()`, will send a copy of the plot on the screen to the default printer. This is equivalent to clicking on the print button of your current graphics device (usually `openlook`, `motif`, or `win.graph`). `printgraph` does not offer many options to control the printed output, and thus it does not provide any significant advantages over just clicking on the print button. In either UNIX or Windows you can use the `dev.print` function to print the current plot window or `dev.copy` to copy the current plot to a graphics file.

To effectively exercise control over aspects of your plot such as fonts and pointsize, you need to use a function appropriate to your printer or graphics editor. If you have a postscript printer, the

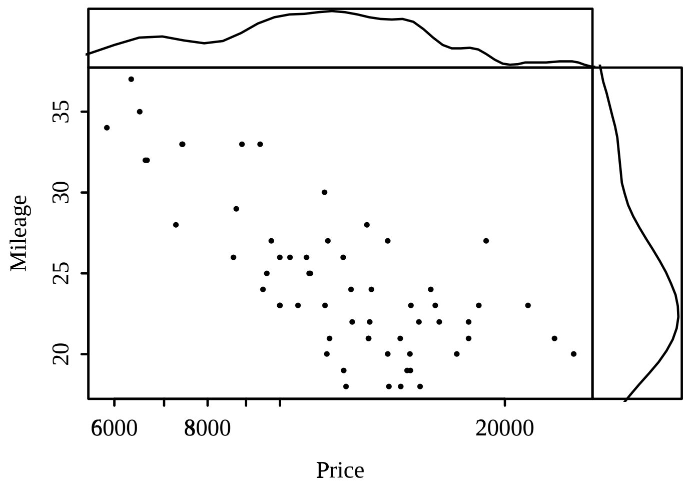

Figure 12.11: *Another subplot example*

function to use is `postscript`. For HP LaserJet printers you can use `hplj`, while for plotters using the HP-GL command set, the function to use is `hpgl`. To make a Windows metafile suitable for inclusion in Word or PowerPoint, you can use the `format="placeable metafile"` parameter with `win.printer`. Check the help for `Devices` for many more possibilities.

12.2.1 Opening Graphics Windows

In UNIX you usually open a graphics window with one of the following commands: `openlook()`, `motif()`, or `X11()`. In version 3.3 for Windows you can use `win.graph` or `win.slide` (see below), or `gs.slide` to set nice defaults for graph sheets in version 4.x.

12.2.2 The `postscript`, `ps.slide`, `setps`, `setpdf` Functions

There are many functions for specifying how graphics output can be stored in specially formatted graphics files. One of the most important functions is `postscript`, which works for both UNIX and Windows, although the `onefile` and `print.it` options do not apply to Windows.

```
> args(postscript)
function(file = NULL, width = -1, height = -1, append = F, onefile = T,
        print.it = NULL, ...)
```

When we type `postscript(...)`, we open a graphics file in the same way that typing `openlook` opens a graphics device on the screen. The difference is that we will not be able to see the results until we close the `postscript` device and send the resulting postscript file to a printer or view the file with a postscript previewer. For this reason, it is advisable to create our plot in an `openlook`, `motif`, or `win.graph` device, and when we are satisfied with the results open a `postscript` device and repeat exactly the commands we used to get the plot in `openlook`. When we close the`postscript` device by typing `dev.off()` we will have a postscript file with our plot, or we may choose to send the output directly to the printer without saving the postscript file. It helps to keep all your plotting commands in a scratch file, that you may then copy and paste to your S-Plus session window. For example, Fig. 12.10 was produced with the following commands.

```
postscript("/users/cfa/Sclass/subplot1.ps",
           width=4/0.727,height=4,hor=F,
pointsize=6)
par(bty="o")
dd ← datadist(car.test.frame)
options(datadist="dd")
f ← ols(Mileage ~ Type+Disp.)
plot(f,Disp.=NA,Type=NA,conf.int=F)
subplot(plot(Type,Disp.,rotate=T,xlab="",cex=.8),c(60,100),c(15,20))
dev.off()
```

Once you create a postscript graphics file you can preview it using Ghostview or other postscript previewers in UNIX or Windows.

Notice that the example uses an extra argument, `pointsize` that is not present in the list of arguments to `postscript`. The `...` imply that other arguments are accepted. In UNIX (only), a list of all such arguments (including fonts), is available by typing `ps.options()`.

The `onefile=T` argument means that successive plots will be acumulated in one file until we turn the device off. This may not be very useful if we want to incorporate the plots into a document. Setting `onefile` to `F` produces some peculiar results. In this case, each call to a high level plotting function will result in S sending all the plotting commands entered so far to the postscript file overwriting what was in it. To avoid this problem we must turn the device off before calling another high level plotting command. Another way around it is to omit the `file` argument and set `print.it` to `F`.

```
> postscript(onefile=F,print.it=F)
> plot(corn.rain)
> plot(corn.yield)
Starting to make postscript file.
Generated postscript file "ps.out.0001.ps".
> plot(corn.rain,corn.yield)
Starting to make postscript file.
Generated postscript file "ps.out.0002.ps".
> dev.off()
Starting to make postscript file.
Generated postscript file "ps.out.0003.ps".
```

The second call to `plot` closed the first postscript file; the third call closed the second file; to end with the third plot we had to close the device. This is just a way to produce several files with a single call to `postscript`. The reason for setting `print.it` to `F` is that, otherwise, S-Plus would have printed the `ps.out.*` files and then deleted them. We could also have used a naming convention for the files.

```
> postscript(onefile=F,print.it=F,tempfile="corn.###.ps")
```

The `###` refer to a sequential number for the file name.

The Hmisc `ps.slide` function for UNIX or Windows uses nice defaults to make four types of common postscript images, as controlled by an argument named `type`. Specify `type=1` to make nice fullsize graphs or `type=3` for making 5 × 7" landscape graphs using 14–point type (useful for submitting to journals). `type=2` (the default) is for color 35mm slides. Use `type=4` to make nice black and white overhead projection transparancies (portrait mode). For example, use the following code for making a 5 × 7" black and white graph with nice fonts.

```
ps.slide('myplot', type=3)   # makes myplot.ps
plot(x, y)
dev.off()
```

See the online help for `ps.slide` for more options.

The Hmisc `setps` function makes small postscript figures suitable for papers. `setps` was used to produce the small graphs in this document[1]. Here is an example.

```
setps(myplot)   # makes myplot.ps   Note absence of quotes
plot(....)
topdf()         # converts myplot.ps to myplot.pdf using Ghostscript
                # topdf is created by setps
dev.off()
```

`setps` also has an option to set up for `trellis` postscript graphics, and for converting postscript to `.pdf`. There is also an Hmisc function `setpdf` for setting up `.pdf` files, although creating a postscript file and converting it to `.pdf` often works better (as in the above example).

Windows users can easily incorporate postscript graphics into Microsoft Word and other applications (even though such graphics will not display on the screen) as long as they have a postscript printer.

12.2.3 The `win.slide` and `gs.slide` Functions

The basic plotting devices for Windows version 3.3 are `win.graph` and `win.printer`. To use nicer defaults for presentations and publications you can use the `win.slide` function in Hmisc. `win.slide` works similarly to `ps.slide` but draws graphs in the graphics window or writes a Windows metafile. If the file name is `''`, the graph is sent directly to the printer. The default value for `type` is 3 for `win.slide`. For S-Plus version 4.x for Windows, you can use the Hmisc `gs.slide` function to set up nice defaults for graph sheets. When you copy graph sheets that have been produced with `gs.slide` in effect to the clipboard and paste the graphs into Microsoft applications, the results will be more pleasing than when using the default graphical parameters.

12.2.4 Inserting S Graphics into Microsoft Office Documents

In S-Plus 2000 and S-Plus 6, copying and pasting a graph sheet page into Microsoft Word, Powerpoint, etc. does not reliably render a graph. A more reliable approach is to do `File ... Export Graph` to explicitly export the graph into a Windows metafile. In Word you can insert the graph using `Insert ... Picture ... From File`. It is important not to resize or otherwise edit the

[1] `setps` was changed to use a Helvetica font by default after these graphics were created.

graph using the mouse (for example by dragging a corner of the graph) as the graph will often become corrupted. To resize, right click and select a new size. To use a command to create a wmf file, use `win.printer`. In S-Plus 6.0 (even under UNIX/Linux) you can use the `wmf.graph` function.

Julian Wells has provided some valuable pointers for setting up graphics in Word:

> Word 97 has good tools for explicit formatting of pictures: select the picture, then choose Format → Picture (may show up as Format → Object; this seems to depend on exactly what option one chose when inserting the picture with Paste → Special). This brings up a tabbed dialogue box:
>
> **The Picture tab** allows you to crop the picture (so one can trim unwanted margins: there are also control here for colour, brightness and contrast, but I assume that scientific work will normally be black-and-white only)
>
> **The Size tab** allows one to change the size; the important thing here is that one has options to size relative to the original, and to maintain the original aspect ratio (the alternatives should be viewed very critically, in my opinion – not keeping the aspect ratio will distort your favourite type face, for a start).
>
> All this can be programmed simply by recording a macro, of course, so repetitive work is no problem.
>
> The key thing in lining up the charts neatly is a good understanding of the use of the Word ruler and/or the Format → Paragraph dialogue in line formatting (e.g. margins and indenting). I also find it very helpful to use table cells as containers for graphic material and captions – just about essential if one wants them in a multi-column layout.sent you.

Windows metafiles under the best of circumstances often do not render the graphic very well. The most beautiful graphs will be produced by outputting a postscript file for S and using `Insert ... Picture` in Word. This will put up a blank box on the screen but will print perfectly well to a postscript printer. If you want to be able to preview the graph on the screen (which is essential for Powerpoint presentations), have S-Plus export the graph with a `TIFF` preview image. This will greatly enlarge the size of the graphics file, however.

`Pstoedit` is a useful program for converting postscript graphics files to a variety of formats (including windows metafiles) for editing and for importing into Micro Office. The Windows version of the program includes an Office graphics import plug-in for importing postscript graphics. `pstoedit` may be obtained at `http://www.geocities.com/SiliconValley/Network/1958/pstoedit`.

Chapter 13

Managing Batch Analyses, and Writing Your Own Functions

13.1 Using S-Plus in Batch Mode

As an exploratory tool it is best to use S-Plus interactively. Interactive use is also valuable for debugging a large S-Plus program. But it is useful to ultimately create a file which will have the main elements of the analysis and which we can submit at any point to obtain a final report. That way analyses can easily be updated when new data arrive, when data are corrected, or when additional analyses are desired.

In what follows, S-language source files have extension `.s`. If using the Windows S-Plus script editor you might use its default suffix of `.ssc`. You can submit source files from inside S-Plus to get the results on the screen, or use a batch command which will save the printed results to a file. See Section 1.5 for related material.

13.1.1 Batch Jobs in UNIX

Typing the following command at the UNIX prompt will cause S-Plus to be run in batch mode in the background (omit the `&` to run it in the foreground):

```
Splus <file.s >file.lst &
```

This will run the S program `file.s` and produce the output file `file.out`.The command `Splus BATCH file.s file.lst &` is supposed to work but some users may have to prefix the command with `nohup`. It may be advantageous to define a UNIX shell program called `Bs` (in for example `/usr/local/bin`) to run S-Plus jobs in the background at a low priority, causing input commands to be interspersed with the printed output. The shell (`csh`) program is defined as follows:

```
/bin/nice -5 Splus BATCH $1.s $1.lst
echo "echo "‘pwd‘"/$1.lst;tail "’$1’ ‘pwd‘"/$1.lst" > $HOME/.lj
chmod a+x $HOME/.lj
```

Then you can initiate the job by typing `Bs file` at the UNIX prompt (note that the `.s` and `&` are implied automatically). `Bs` causes an executable program `.lj` to be placed in your root directory. When you enter the command `.lj` from any directory (assuming your home directory is in the directory path), the tail of the `.lst` file for the last job submitted will be printed. That makes it easy to monitor the job's progress.

Another useful script is `Bsw`, which causes the system to wait until the S-Plus job is finished before running the next program. In some occasions you may have several files that need to be run in a given order; rather than doing one at a time you may create an executable file with a `Bsw` for all the files that need to be created and then have control returned to you after that file is executed. As an example suppose that we had edited a file `create.all.s` with all the S-Plus jobs that need to be submitted. This master file might contain the following.

```
    Bsw create.file1.s
    Bsw create.file2.s
             .
             .

    Make it into an executable file (file owner only)

biostat3{cfa}: chmod u+x create.all.s

    Now run it

biostat3{cfa}: create.all.s &
```

`Bsw` is defined as follows:

```
echo "echo "‘pwd‘"/$1.lst;tail "’$1’ ‘pwd‘"/$1.lst" > $HOME/.lj
chmod +x $HOME/.lj
/bin/nice -5 echo ’options(echo=T)’ | cat - $1.s | Splus 1>$1.lst 2>&1
```

The following is an example of a file you could submit.

```
store()
library(Design,T)
postscript(’/tmp/plot’)
....
plot(fit)

# Define a function to print a heading with surrounding blank lines
note ← function(text) cat(’\n’, text, ’\n\n’)
note ← function(string) invisible(cat(’\n’,string,’\n\n’))
note("time to dnr among pts with preference to forgo CPR")
print(f)
print(anova(f))
```

We can run this code from inside S-Plus by typing `source("filename.s")` or `src(filename)`, or in batch mode. Notice that we don't need quotes nor the `".s"` extension with `src`. Apart from that the only difference is that `src` "remembers" the last file executed successfully, so, if you want to re-submit it, you need only type `src()`.

13.1.2 Batch Jobs in Windows

As mentioned in Chapter 1, it is a good idea to make a shortcut to S-Plus in each project directory. This will allow S-Plus to be run interactively from that project directory. You can make a second shortcut for running S-Plus in batch mode. For example, to run the program `filename.s` and create an output file `filename.lst`, use the `Properties` option in a shortcut to define a command such as `c:\splus45\cmd\splus.exe S_PROJ=. /BATCH filename.s filename.lst filename.-log`. Also, have the `Start in` box set to the project directory. Unfortunately, you'll have to edit the `Properties` or make a new shortcut for each source file to be used as the basis for a batch job.

To have your commands appear in the `.lst` file, put the command `options(echo=T)` at the start of the program or use the `S_FIRST` option below.

Assuming you are currently in the correct project directory, you can run S-Plus from the DOS prompt using a command such as

```
start splus /BATCH my.s my.lst my.err S_PROJ=. S_FIRST='options(echo=T)'
```

To make DOS wait until S-Plus finishes before running the next command, insert `/w` after `start`. To do the same thing in the Cygnus `bash` command shell under the Cygnus `cygwin32` system, use a command like

```
/sp2000/cmd/splus.exe /BATCH my.s my.lst my.err S_PROJ=. S_FIRST='options(echo=T)'
```

Add & after the command to cause the system not to wait until S-Plus finishes to go on to the next command.

You can run S-Plus batch jobs from the Windows `Start` ...`Run` command by specifying a command such as `splus /BATCH \mydir\prog.s \mydir\prog.lst S_PROJ=\mydir`.

Using any of these approaches you will see a progress dialog. Add `/BATCH_PROMPTS min` to iconify this dialog.

The authoritative reference to the S-Plus 2000 command line is *S-Plus 2000 Programmer's Guide, 1999*, Chapter 19, which is available on the online help.

13.2 Managing S-Plus Non-Interactive Programs

As just discussed, an S-Plus program may easily be run in batch mode, producing a single `.lst` file containing the output. The batch program may also produce one or more plot files. However, in an analysis project it is frequently the case that some components of the analysis actually get completed, and if these require significant computer time, it is advantageous to not repeat those sub-analyses every time the program is run. Many analysts break down components of the analyses into a long array of batch programs which often contain highly repetitive setup code (e.g., recoding variables, extracting subsets of observations to process). Multiple programs can cause bookkeeping problems, so it is often better to keep related analysis steps in a single batch file. One can comment out sections of code that do not need to be executed again, it requires a significant amount of editing to comment and un-comment large sections of code. A better approach is to make `if` statements apply to blocks of code:

```
f ← ols(y ~ rcs(x1,4), x=T, y=T)
if(F) {
  anova(f)
  summary(f)
  validate(f)
  }
```

This can be improved upon by making the program more self-documenting:

```
create ← F
fitmod ← F
valmod ← T
if(create) {
  df ← sas.get(...)
  df.desc describe(df)
  ddist ← datadist(df)
}
if(fitmod) {
  fit ← lrm(death ∼ age*sex, x=T, y=T)
  print(fit)
  print(anova(fit))
}
if(valmod) {
  val ← validate(fit)
  print(val)
}
```

There are two disadvantages to the last two examples. First, you must explicitly `print` objects; typing `fit` instead of `print(fit)`, for example, will not cause the object `fit` to print. This is because when you put a series of commands inside `{}`, the last object listed before the closing brace is "returned" by the nested expression, and only this last object is printed automatically. The second disadvantage is that the `.lst` file that was created when earlier parts of the program were executed will be overwritten by the new output.

The `do` function in Hmisc was created to facilitate conditional execution of parts of the analysis depending on what needs to be re-run. `do` makes it easy to write different parts of the analysis to different `.lst` files, and similarly it can segment plot output files. The first argument to `do` is a logical value. When this value is `T`, the second argument, which is an expression of arbitrary length, is executed. Otherwise this second argument is ignored. The second argument must be enclosed in `{}`; that is how it can contain multiple S-Plus statements. There are several optional arguments to `do`:

`device`: This specifies a function that sets up the graphics device if any graphics are being done. `do` expects one of the following to be specified (in quotes): `'postscript'`, `'ps'`, `'ps.slide'`, `'win.slide'`, `'win.printer'`. `device` may also be specified through a system option called `do.device`, e.g., `options(do.device = 'ps.slide')`.

`file`: The name of the output file for this section of the program. It is automatically suffixed by `.lst`. `file` can be a special keyword `'condition'`, in which case the `.lst` file will be the name of the first argument to `do`. `file` may also be specified through a system option called `do.file`.

`append`: Set this to `T` to have `.lst` output appended to an existing file.

`multiplot`: Set this to `T` if you are using Windows and you wish each plot to go in a separate graphics file, with the files numbered sequentially.

... Any number of other arguments may be specified, which are passed to the plotting device function. For example, if using `device='ps.slide'`

Besides the system options mentioned above, selected system options work with `do`.

`do.prefix`: A character string to prepend before `.lst` file names.

`do.echo`: Type `options(do.echo=F)` to prevent S-Plus commands from being interspersed with function output in the `.lst` file.

`do.comments`: Type `options(do.comments=T)` to include comments in the `.lst` file.

Here is an annotated example showing how to use this function. This example also shows how the Design library was used.

```
options(digits=3, datadist='ddist', continue='\n +',
        do.device='postscript', do.file='condition')
# do.file='condition' outputs do(condition,...) to condition.lst
# 'postscript' makes do(condition, ...) output postscript graphics
# into condition.ps
# May want to use do.device='win.slide' for Windows - uses nice defaults
# Add for example options(do.prefix='model') if you want .ps and .lst
# file names created by do() to be prefixed by 'model.' before any
# prefix normally created by do().

# Here is the analysis program, model.s

create              ← F
descriptives        ← F
ordinality          ← T
cluster.fit.impute  ← F
full.model          ← F
find.penalty        ← F
check.residuals     ← F
separate.binary.fits← F
validate.mod        ← T
simplify.nomogram   ← T


do(create, {
  tami.chf ← sas.get('/users/jdl/projects/COConnor','frankchf',recode=T,
                     id='studyno', ifs='if chfdev>.')

  hist.data.frame(tami.chf, n.unique=2)
  pstamp()   # pstamp in Hmisc date-time stamps plots
  par(mfrow=c(4,5))
  plot(tami.chf[-c(2,4,5,9,11,12,16,18,19,21,22,24,25,26,27,28,32,36,37,38)],
       ask=F)  # omit binary and ID vars
  pstamp()

  desc.tami ← describe(tami.chf)
  ddist     ← datadist(tami.chf)
})

store()
attach(tami.chf)

table(chfdev)
table(newpe)
y ← score.binary(chfdev, newpe, death)
table(y)
yn ← score.binary(chfdev, newpe, death, retfactor=F)
table(y, yn)

# Could also use:
# y ← ifelse(death, 3, ifelse(newpe,2, ifelse(chfdev, 1, 0))) or
# y ← 0; y[chfdev==1] ← 1; y[newpe==1] ← 2; y[death==1] ← 3


do(ordinality, {

  y.nodeath ← score.binary(chfdev, newpe)
  summary(death ∼ y.nodeath)

  par(mfrow=c(2,2))
  # The following is new:
  plot.xmean.ordinaly(y ∼ age + efpre + izpre + numdz, cr=T)
  pstamp('Figure 1')
})


do(descriptives, {

  y3 ← cbind('>=CHF'=yn>=1, '>=PE'=yn>=2, Death=yn==3)

  # Stratify separately on each predictor variable, computing all
  # cumulative probabilities.  For continuous variables, quartiles are used.
  s ← summary(y3 ∼ age + cptrttim + diabp + sysbp + drug + efpre +
              eversmk + htn + hxdiab + hxsmk5 + izpre + miloc +
              nonizpre + numdz + ptca + pulse + pvd + race +
              ralesyn + s3 + sex, nmin=15)
  # Default summary function (fun=) is mean of each column of Y
  # Add ,fun='%' after nmin=15 to compute % instead of fractions
  w ← latex(s, longtable=T, title='descriptives')  #makes descriptives.tex
  # Instead, summarize using logits of cumulative probs.
  # qlogis is S's built-in log(p/(1-p)) function

  g ← function(y3) qlogis(c('Logit >=CHF'=mean(y3[,1]),
                            'Logit >=PE' =mean(y3[,2]),
                            'Logit Death'=mean(y3[,3])))

  s ← update(s, nmin=15, fun=g)
  # update -> do same summary as created previous s but change options

  # Make dot plots for first 11 of 21 predictor vars (too many for 1 page)
  plot(s[1:11,],  which=1:3, xlab=xl ← 'Log Odds of Cumulative Probability',
       cex.labels=.6, pch=c(5,10,183),
       main=d ← 'Examining Proportional Odds Assumption')
  pstamp('Figure 2')    # date-time stamps lower right corner
  # Make dot plots for remaining variables
  plot(s[11:21,], which=1:3, xlab=xl, cex.labels=.6,
       pch=c(5,10,183), main=d)
  pstamp('Figure 3')

  logit ← function(p) ifelse(p==0 | p==1, NA, log(p/(1-p)))
  # qlogis did not work for next function - some conditional
  # probabilities were 0 and qlogis returned -infinity
  # logit is actually pre-defined (in /suserlib/.Data)

  h ← function(y) logit(c(mean(y==0), mean(y[y>=1]==1), mean(y[y>=2]==2)))

  s ← update(s, yn~., fun=h)  # same as previous s but with new fun
  plot(s[1:11,],  which=1:3, xlab=xl ← 'Log Odds of Conditional Probability',
       cex.labels=.6, pch=c(5,10,183),
       main=d ← 'Examining Continuation Ratio Assumption')
  pstamp('Figure 4a')
  plot(s[11:21,], which=1:3, xlab=xl, cex.labels=.6,
       pch=c(5,10,183), main=d)
  pstamp('Figure 4b')
})


do(cluster.fit.impute, {

  par(mfrow=c(2,1))
  plot(naclus(tami.chf))
  pstamp('Figure 5')
  # Do hierarchical clustering based on a similarity matrix of
  # squared Spearman correlations
  vclus ← varclus(~.-studyno-race-tami-chfgrp-chf-timtohf-timtodth,
                  data=tami.chf, sim='spearman')
  store(vclus)
  plot(vclus)
  pstamp('Figure 6')
  # Have transcan develop customized regression to predict each
  # predictor for all the other predictors.  transcan will also
  # impute missing values.  Use trantab=T so that fitted transformations
  # can be easily evaluated for future data

  trans ← transcan(∼age + cptrttim + diabp + efpre + eversmk + hbeta +
                   hcablock + htn + hxdiab + hxsmk5 + izpre + miloc +
                   murmur + nonizpre + numdz + ptca + pulse + pvd +
                   ralesyn + s3 + sex + sysbp + timi90,
                   imputed=T, shrink=T, trantab=T, eps=.5, pl=F)
  store(trans)
  par(mfrow=c(6,4))
  plot(trans)
  pstamp('Figure 7')
})

# Note how the following union of conditions makes it clear which
# parts of the analysis depend on the use of imputed values

do.impute.reduce ← full.model | find.penalty | check.residuals |
                   separate.binary.fits | simplify.nomogram

do(do.impute.reduce,
{

  # Imputation and data reduction needs to be done before any multivariable
  # model fits
  impute(trans)  # imputes all variables, here putting them in .Data.tempxxxx
  # To only impute certain ones, do e.g. numdz ← impute(trans, numdz)

  describe(efpre)
  describe(efpre[is.imputed(efpre)])


  numdz ← round(numdz) # some imputed values were fractional
                       # since didn't tell transcan that
                       # numdz was categorical
  drug     ← impute(drug) # replace 3 missing with most frequent (t-PA)

  table(race)
  # Combined last 5 levels of race
  levels(race) ← levels(race)[c(1,2,7,7,7,7,7)]
  # or levels(race) <- list(other=levels[3:7])
  table(race)

  sex ← factor(sex, labels=c('male','female'))  # was a numeric var

  map ← (2*diabp+sysbp)/3
  label(map) ← 'Mean Arterial Blood Pressure'
  table(is.imputed(diabp),is.imputed(sysbp))
  nrisk ← htn + pvd + hbeta + hcablock + murmur
  label(nrisk) ← 'Number of Misc. Risk Factors'

  s3.rales ← s3 | ralesyn
  label(s3.rales) ← 'S3 Heart Sound or Rales'

  ddist ← datadist(ddist, map, race, nrisk, s3.rales, sex)

})


do(full.model,
{

  f ← lrm(y ∼ rcs(age,3) + sex + race + rcs(map,4) + rcs(pulse,4) +
          pol(timi90,2)  + rcs(izpre,4) + rcs(nonizpre,3) + rcs(efpre,3) +
          miloc + hxsmk5 + s3.rales + rcs(cptrttim,4) + ptca + drug +
          hxdiab + scored(numdz) + nrisk, x=T, y=T)
  f$stats


  prlatex(latex(f, caption='Full Unpenalized Nonlinear Model'))
  an ← anova(f)
  an
  plot(an)
  title('Strength of Predictors of Ordinal Response')
  pstamp('Figure 8')

  par(mar=c(5,4,4,5)+.1)
  plot(f, efpre=NA, ref.zero=T, ylim=c(-1.5,1.5))
  abline(h=0, lty=2)
  abline(v=medef ← ddist$limits$efpre[2], lty=2)
  axis(4, log(at ←  c(.25,.5,.75,1,1.25,1.5,1.75,2,2.5,3,3.5,4,4.5)),
       labels=format(at), srt=90)
  mtext('Odds Ratio', side=4, line=3)
  text(medef, -1.3, 'Median efpre', adj=0, srt=90)
  title('Effect of efpre - Relative Log Odds')
  title(sub='plot(f, efpre=NA, ref.zero=T)', adj=0)
  pstamp('Figure 8a')

  par(mfrow=c(4,5), mar=c(5,4,4,1)+.1)
  plot(f, ref.zero=T, ylim=c(-1.5,1.5))  # no variables mentioned -> plot all
  # ref.zero=T -> Subtract a constant from X beta  before  plotting  so
  # that the reference value of the x-variable yields y=0.
  pstamp('Figure 8b')

  store(f, 'fit.full')
})

options(do.file='condition')    # now make do() store results in sep. files
```

```
do(find.penalty, {

  # First try penalizing all parameters (except the intercept)
  # Try the following vector of penalty factors:
  pens ← c(1:10,20,40,80,160,320,640,1280,2500)
  pentrace(fit.full, pens)
  # Best penalty was 40, with 19.98 effective d.f. (AIC=147.1)
  # (Started with 34 d.f. in the unpenalized fit, AIC=133.5)

  # Now try penalizing only parameters associated with nonlinear effects
  pentrace(fit.full, pens, penalize=2)

  # Penalized model most likely to cross-validate the best is a model
  # with infinite penalty for the nonlinear terms (i.e., all betas for
  # nonlinear terms shrunk to zero).
  # This is consistent with the Wald statistics from
  # the combined nonlinear terms being 15.94 with 14 d.f., i.e., the
  # chi-sq is less than 28 -> further justification for using a linear
  # model.  The model with no nonlinear terms has 20 d.f. and the
  # effective AIC in a 20.4 d.f. penalized model is 145, almost as good
  # as the 19.98 d.f. fully penalized nonlinear model.

  # Let's also fit a linear model and see if it could be improved on
  # by penalizing the linear effects.  Let's cheat a little and use
  # prior knowledge on the ejection fraction transformation.

  f ← lrm(y ∼ age + sex + race + map + pulse +
           timi90  + izpre + nonizpre + pmin(efpre,60) +
           miloc + hxsmk5 + s3.rales + cptrttim + ptca + drug +
           hxdiab + numdz + nrisk, x=T, y=T)
  f$stats

  prlatex(latex(f, caption='Full Unpenalized Linear Model'))
  anova(f)
  store(f, 'fit.full.linear')

  pt ← pentrace(f, pens)
  pt

  # We see that further shrinkage yields AIC=153 (with penalty=80, df=12.6)
  # Fit and save this penalized linear model

  f ← update(f, penalty=pt$penalty, x=T, y=T)
  prlatex(latex(f, caption='Full Penalized Linear Model'))
  anova(f)
  store(f, 'fit.full.linear.penalized')

})

do(check.residuals,
{

  # Fit reduced model and check residuals on most important variables,
  # to look for non-proportional odds.  Also take a look at residuals
  # from building block models used in continuation ratio model.

  # Fast backward step-down, using default statistic (AIC) but
  # Compute it on individual variables instead of using residual chi-square
  # aics=0 means delete variables with AIC<2, AIC = chisq-2 x d.f.
  # Override: do put map in model, and add hxsmk5 because of what was
  # found later in the program.

  fastbw(fit.full, type='individual', aics=2)

  # Use untransformed predictors  since we want to estimate transformations
  # using partial residuals

  f ← lrm(y ~ age + map + efpre + ptca + hxsmk5, x=T, y=T)

  par(mfrow=c(2,3),oma=c(3,0,3,0))
  resid(f, 'score.binary', pl=T)
  mtitle('Binary Logistic Model Score Residuals\nFrom Ordinal Model Fit',
         ll='Figure 9')
  # overall title+stamp, plus Figure # in lower left corner (ll)

  par(mfrow=c(2,3),oma=c(3,0,3,0))
  resid(f, 'partial', pl=T)
  mtitle('Binary Logistic Model Partial Residuals\nFrom Ordinal Model Fit',
         ll='Figure 10')

  # Compute global goodness-of-fit statistics
  # le Cessie - van Houwelingen - Hosmer (see residuals.lrm for refs)
  resid(f, 'gof')

  # Plot smoothed partial residuals for binary models that would be
  # components of a forward continuation ratio model (temporarily
  #combining 2 CHF categories).

  fit.none ← lrm(yn==0 ~ age + map + efpre + ptca + hxsmk5, x=T, y=T)
  fit.chf  ← update(fit.none, yn==1 | yn==2 ~ ., subset=yn>=1)
  fit.none$stats; fit.chf$stats

  # The plot.lrm.partial function computes partial residuals for a
  # sequence of binary logistic fits, and draws smoothed (lowess)
  # partial residual plots for each predictor (all fits) on one graph.
  # This is repeated for all predictors.  See online help for residuals.lrm

  par(mfrow=c(2,3), oma=c(3,0,3,0))
  plot.lrm.partial(fit.none, fit.chf)
  mtitle('Partial Residuals for 2 Binary Model Fits:\nY=0 and Y=1 or 2 | Y>0',
         ll='Figure 11')

  # Do the same thing for backward continuation ratio models

  fit.death ← lrm(yn==3 ∼ age + map + efpre + ptca + hxsmk5, x=T, y=T)
  fit.chf   ← update(fit.death, yn==1 | yn==2 ∼ ., subset=yn<3)
  fit.death$stats; fit.chf$stats

  par(mfrow=c(2,3), oma=c(3,0,3,0))
  plot.lrm.partial(fit.death, fit.chf)
  mtitle('Partial Residuals for 2 Binary Model Fits:\nY=3 and Y=1 or 2 | Y<3',
         ll='Figure 12')

})


do(separate.binary.fits,
{

  # Find list of variables that are important from either of 2
  # dichotomizations.  Use linear models.

  # First, model for any CHF or death
  f.any ← update(fit.full.linear, yn>0 ∼ ., x=F, y=F)
  fastbw(f.any, type='individual', aics=2)

  # Now model for death
  f.death ← update(f.any, yn==3 ∼ ., x=T, y=T)
  f.death
  fastbw(f.death, type='individual', aics=2)

  # Demonstrate that if we wanted to develop a separate model for death,
  # significant shrinkage is needed since there are only 89 events

  pt ← pentrace(f.death, c(1,2,4,8,16,32,64,128,256))
  pt
  xbeta.orig ← predict(f.death)
  f.death ← update(f.death, penalty=pt$penalty, x=F, y=F)
  # pt$penalty is best penalty as determined by pentrace (here, 32,
  # resulting in 11.7 effective d.f. - we started with 20 d.f.)
  f.death$stats

  # Compare predictive discrimination of customized death model
  # with ordinal model when asked to predict dead vs. alive

  somers2(predict(fit.full.linear.penalized), yn==3)

  plot(xbeta.orig, predict(f.death), ylab='Shrunken X*Beta', pch=202)
  # 202=small open circles (degree sign) on postscript printers
  abline(a=0, b=1, lwd=3)
```

```
title('Effect of Shrinkage on Linear Predictors\nIn Model for Death')
pstamp('Figure 13')

f.any ← lrm(yn>0 ~ age + map + pmin(efpre,60) + miloc + ptca +
             numdz + hxsmk5)
f.death ← update(f.any, yn==3 ~ .)

# Show side-by-side odds ratio charts.  The the default (inter-quartile-
# range odds ratios) for continuous variables

s.any   ← summary(f.any)
s.death ← summary(f.death)

par(mfrow=c(1,2), mgp=c(3,.4,0))
plot(s.any,   log=T, main='Binary Model for Y>0',
     at = at ← c(.25,.5,1,2,4,8))
plot(s.death, log=T, main='Binary Model for Y=3', at=at)
pstamp('Figure 14')


# Fit a reduced ordinal model and compare the predictions it
# gives for Prob{death} to the predictions from f.death

f ← update(f.death, y ~ .)
prob.death.ordinal ← predict(f, type='fitted')  #Computes all Prob(Y>=j)
prn(prob.death.ordinal[1:10,],
    'First 10 Rows of Reduced Ordinal Predictions')
prob.death.ordinal ← prob.death.ordinal[,3]

prob.death.customized ← predict(f.death, type='fitted')
describe(prob.death.ordinal - prob.death.customized)
par(mfrow=c(1,1))
plot(prob.death.ordinal, prob.death.customized, pch=202, log='xy',
     xlim=c(.001,.5),ylim=c(.001,.5))
abline(a=0, b=1, lwd=3)
scat1d(prob.death.ordinal)
scat1d(prob.death.customized, side=4)
title('Predicting Prob{Death} From\nCustomized and Proportional Odds Model')


# Form intervals of predicted probability of death from the ordinal
# model, such that there are 100 pts in each interval
# The levels.mean option to cut2 forces intervals to be labeled
# with the mean value within the interval, rather than the
# interval endpoints.  This allows estimates to be positioned
# sensibly on the x-axis

pdo ← cut2(prob.death.ordinal, m=100, levels.mean=T)

# Now find 0.9 quantile of prob of death stratified by prob of
# death from ordinal model.  Do same for 0.1 quantile.
```

```
  upper ← tapply(prob.death.customized, pdo, quantile, probs=.9)
  lower ← tapply(prob.death.customized, pdo, quantile, probs=.1)
  # convert levels of stratification variable to numeric
  x ← as.numeric(levels(pdo))

  lines(x, upper, lty=2, lwd=3)   # lty=2: dotted   lwd=3: triple thickness
  lines(x, lower, lty=2, lwd=3)
  pstamp('Figure 15')
})


do(validate.mod, {
  # Bootstrap validation of various indexes of fit
  val ← validate(fit.full.linear.penalized, B=150)
  val
  store(val)

  # Bootstrap smooth (lowess) nonparametric calibration curve
  cal ← calibrate(fit.full.linear.penalized, B=150)
  store(cal)
  plot(cal)
  title('Bootstrap Calibration of Penalized Linear Model')
  pstamp('Figure 16')

  cal.unpen ← calibrate(fit.full.linear, B=150)
  store(cal.unpen)
  plot(cal.unpen)
  title('Bootstrap Calibration of Unpenalized Linear Model')
  pstamp('Figure 16b')
}, file='')   # file='' -> print output goes to model.lst


do(simplify.nomogram, {
  # Approximate final model's predictions (logits) from a sub-model
  # This is more stable than doing stepwise variable selection against
  # the output, and it automatically makes use of penalization

  # Get predicted logit from final model (using first intercept)
  plogit ← predict(fit.full.linear.penalized)
  # Add a random error to this so that stepwise variable selection works
  alogit ← plogit + rnorm(length(plogit), sd=.2)

  f ← ols(alogit ~ age + sex + race + map + pulse +
          timi90  + izpre + nonizpre + pmin(efpre,60) +
          miloc + hxsmk5 + s3.rales + cptrttim + ptca + drug +
          hxdiab + numdz + nrisk)
  fastbw(f, aics=10000)  # aics=10000: eventually delete all variables

  # Fit sub-model against final model's predicted logit, using last
  # few variables deleted by fastbw
  f ← ols(plogit ~ pmin(efpre,60) + age + ptca + izpre + numdz +
           map + miloc + hxdiab + s3.rales)
  f

  # Compare approximate predicted logits to full model logits
  describe(abs(predict(f) - plogit))

  store(f, 'fit.full.linear.penalized.approx')

  # Make a nomogram based on the approximate model, with axes for
  # reading off predictions for all levels of output severity

  intercepts ← fit.full.linear.penalized$coefficients[1:3]
  fun2 ← function(x) plogis(x-intercepts[1]+intercepts[2])
  fun3 ← function(x) plogis(x-intercepts[1]+intercepts[3])
  nomogram(f, fun=list('Prob(CHF or Death)'=plogis, 'Prob(PE or Death)'=fun2,
           'Prob(Death)'=fun3),
           fun.at=c(.01,.05,seq(.1,.9,by=.1),.95,.99),
           cex.var=.7, cex.axis=.75, lmgp=.2)
  pstamp('Figure 17')
}, file='')


print(scan('/users/feh/.lst', list(char.string='')), quote=F)
# .lst was created by do() if using UNIX (don't usually print this)
# can say .lst lpr to send all .lst files to printer, or .lst xless
# to view them all in windows.  Do .lst ls to see their names.
# Note: The very first time .lst is created, the system won't
# be able to find it in your root directory unless it is the current
# directory.  When you re-log in in the future, the system will
# note the existence of this executable file in your root area
# (you can issue the UNIX 'rehash' command to have .lst instantly
# available from any directory).
```

13.3 Reproducible Analysis

Common problems for an analyst are figuring out how she obtained a certain calculation, what subset of subjects was used to draw a graph, whether a regression analysis was run before or after a data error was corrected, and what sequence of menus was run to produce an analysis. These issues are especially important when formal inference is an important part of the analysis, and especially when results are submitted to a peer-reviewed journal or to a regulatory authority such as the FDA. We have worked with investigators who are completely unable to reproduce results they obtained using interactive software.

Even though interactive or exploratory analysis may be an ideal mode of operation for an FDA reviewer or anyone else wishing to review an analysis to check for robustness, interactive analysis does not lead to well-documented and easily re-run analyses. The best way to have reproducible analyses is to build a complete, cumulative script file as the analysis develops. This script file can be run as a batch file to produce a list or report file as well as several graphics files. Roger Koenker argues that the ultimate documentation of much of scientific research is the source code behind the production of calculations, plots, and tables, and such code (or a URL to it) should be included in many scientific publications.

But what if the analysis was split into several S-Plus jobs, and what if the analysis depended on an S-Plus data frame that was imported from a SAS dataset? If the SAS dataset were to be updated, how do we know what all needs to be re-run in S-Plus? What if the graphics needed to be run through a command-oriented conversion program before inclusion in a report or on a Web page and we get tired of running the conversion steps manually? A solution to this problem is the use of the (originally) UNIX `make` utility program, also available on Linux and Windows 95/98/NT/2000 from the free Cygnus `cygwin32` package from `sourceware.cygnus.com/cygwin`[1]. In a `Makefile` you specify file dependencies. `make` analyzes these dependencies and examines file dates to see which programs need to be run so that all files are up to date.

Often in S-Plus the final file to be produced is an object. Specifying an object name in `.Data` or `_Data` in your `Makefile` is no problem except that under Windows, S-Plus still translates long file names to legal DOS names, and you will usually not know (or care about) such names. Possible solutions to this problem include creating as the final object in a job an object with a plain 8-letter (or less) name, or creating an ASCII file in the project directory (not in `.Data`) of any name. These "last objects created" can be used as the principal object name in the `make` dependencies.

As a demonstration of `make` under Windows, suppose we have the following files in our project directory:

```
test.dat
-------
1 11
2 22
3 33

create.s
--------
dat <- read.table('test.dat',col.names=c('x','y'))

analyze.s
---------
library(Hmisc, T)
results <- describe(dat)
```

[1] You have to install the full `cygwin32` product to get `make`, i.e., `full.exe`, not `usertools.exe`. See `hesweb1.med.-virginia.edu/biostat/s/EmacsTeX/index.html` for tips on installing `cygwin32`.

```
Makefile
--------
Splus = /sp2000/cmd/splus.exe S_PROJ=. S_FIRST='options(echo=T)' /BATCH
DATA = ./_Data

.IGNORE:

.INIT:

all: $(DATA)/results $(DATA)/dat

$(DATA)/results: analyze.s $(DATA)/dat
	$(Splus) analyze.s analyze.lst analyze.err

$(DATA)/dat: create.s test.dat
	$(Splus) create.s create.lst create.err
```

Note that in `Makefile` the indented lines have a tab, not spaces, on the left. To run the `Makefile` type `make` at the `bash` prompt. This will cause objects to be created if they don't already exist, or cause them to be re-created if their ancestors are deleted or modified.

A very nice tutoral on `make` by Duncan Temple Lang and Steve Golovich may be found in in an issue of the Statistical Computing and Graphics Newsletter at `http://cm.bell-labs.com/cm/ms/who/cocteau/newsletter/issues/v92/v92.pdf`.

Perl is a powerful language that is useful for a huge variety of tasks (such as manipulating data files) including managing program execution. In the example below for Unix/Linux, a directory named `input_dir` contains a list of input files to process, in a text file called `List` (one file name per line). An application called `APP` is run on each file to create a corresponding output file in directory `output_dir`. `APP` is only run when the input file has changed since the output file was last created, or if the corresponding output file does not exist. There is one exception. One of the file names in `List` needs to be changed. Also, the name of the output file is actually contained in the first line of the individual input file, after "# ", with a suffix of `.ppp` added.

```
#!/usr/bin/perl
$input_dir  = "/home/mine/dir";
$output_dir = "/home/mine/otherdir";
open(LSTFILE,"$input_dir/List");
@files = <LSTFILE>;
foreach $file (@files) {
  chop $file;  # remove end of line character
  # Exception for input file name: use yyyy instead of xxx for one file
  $file =~ s/xxx/yyyy/;
  $outfile = `head -1 $input_dir/$file`;  # Get 1st line
  # Note: command enclosed in back quotes: run UNIX command, return result
  chop $outfile;
  $outfile = substr($outfile, 2, 100);    # Remove leading "# "
  $infile_age = (-M "$input_dir/$file");  # -M: modification age in days
  $destfile = "$output_dir/$outfile.ppp";
  if(fileolder($destfile,$infile_age)) {
    print "Converting $file \tto\t$destfile\n";
    system("cd $output_dir; APP < $input_dir/$file > $outfile.ppp;");
    }
  }

# Define a function that returns true if a file exists and is
# older than $age, or if the file does not exist
sub fileolder { #(filename, age)
  my($file, $age) = @_;
  (! ((-f $file) && ((-M $file) < $age)))
}
```

hesweb1.med.virginia.edu/biostat/s/LiveDoc.html has pointers to useful information about reproducible analysis.

13.4 Reproducible Reports

As it is relatively easy to specify S-Plus commands to produce ASCII files containing LaTeX code, especially for tables, and it is very easy to produce postscript or pdf graphics files in an S-Plus program, running a master LaTeX document containing `\input` statements that include LaTeX code fragments or graphics files through the LaTeX compiler will update the entire report, including cross-references, the table of contents, the index, etc. This LaTeX step could easily be added to a `Makefile` such as the one above. See hesweb1.med.virginia.edu/biostat/s/LiveDoc.html for detailed documentation for using S-Plus and LaTeX to produce statistical graphical and tabular reports.

To assist in documenting how graphics are produced in a report, you can include the S-Plus code in the LaTeX document after putting special comments in the code, e.g.

```
\begin{Example}  # Example environment is in S.sty
                 # LaTeX style from UVa Web page
setps(fig1)      # setps is in Hmisc
plot(...)        # Figure \ref{fig1}
dev.off()
\end{Example}
```

When the code is listed in the document, the actual figure number will be inserted in the S-Plus comment.

13.5 Writing Your Own Functions

13.5.1 Some Programming Commands

We will describe here the commands for loops and conditional execution of statements. In general, for dealing with structures such as matrices and vectors, it is preferable to use vectorized arithmetic and indexing rather than loops, but sometimes they are necessary.

The commands and their syntax are

if(*cond*) *expr*	Evaluates *cond*; if `T` evaluates *expr*
if(*cond*) *expr1* **else** *expr2*	Evaluates *cond*; if `T` evaluates *expr1*; if `F` evaluates *expr2*
ifelse(*cond,expr1,expr2*)	This is a *vectorized* version of **if .. else ...** It evaluates *cond* and returns elements of *expr1* for `TRUE` elements, and elements of *expr2* for `FALSE` elements.
switch(*expr, ...*)	The result of *expr* must be character or numeric, it is compared to rest of the arguments and returns the first one that matches exactly.
for(*name* **in** *expr1*) *expr2*	Evaluates *expr2* for each name in *expr1*

13.5.2 Creating a New Function

One of the best features of S-Plus is perhaps the capability of writing your own functions. Most functions are written in the S language and you can look at them by just typing the function name

```
> sqrt
function(x)
x^0.5
```

Other functions are written in C or Fortran and you don't have easy access to them. However you can write a function that interfaces to a C or Fortran subroutine by using the functions `.C` and `.Fortran`. The general form of writing a function is

```
f ← function(x,y,z,...)   {

        S statements
        . . .
  }
```

We can either create a text file with this code and submit it through a batch command such as `Splus < filename.s` or `Bs filename` in UNIX, or use the `source` or `src` functions or paste the function definition if operating in an interactive session. The function will then reside in your `.Data` directory unless you have something else attached in position one of the search list (same rules as with other objects apply).

The `bpower` function in Hmisc, which approximates the power of a two-sample binomial test, is a good example of a simple function that has a few options for how the calculations are done.

```
bpower ← function(p1, p2, odds.ratio, percent.reduction, n, n1, n2, alpha =0.05)
{
  if(!missing(odds.ratio))
    p2 ← (p1 * odds.ratio)/(1 - p1 + p1 * odds.ratio)
  else if(!missing(percent.reduction))
    p2 ← p1 * (1 - percent.reduction/100)
  if(!missing(n)) {
    n1 ← n2 ← n/2
  }
  z ← qnorm(1 - alpha/2)
  q1 ← 1 - p1
```

```
    q2 ← 1 - p2
    pm ← (n1 * p1 + n2 * p2)/(n1 + n2)
    ds ← z * sqrt((1/n1 + 1/n2) * pm * (1 - pm))
    ex ← abs(p1 - p2)
    sd ← sqrt((p1 * q1)/n1 + (p2 * q2)/n2)
    c(Power = 1 - pnorm((ds - ex)/sd) + pnorm(( - ds - ex)/sd))
  }
```

Here is another simple function.

```
  > spearman ← function(x, y)
    {
          notna  ←  !is.na(x + y)  # exclude NAs
          c(rho = cor(rank(x[notna]), rank(y[notna])))
    }
```

This function just calculates the Pearson correlation on the ranks of `x` and `y` after excluding missing values (since `cor` does not accept missing values).

We could build on the existing functions and write our own versions of them. We could issue the command

```
  > my.matrix ← edit(matrix)
```

to change the default parameter `byrow` from `F` to `T`. `xedit` will make an "edit" window with the code for `matrix` to appear and make the changes there. It is possible to use other editors. Others may use the function `fix` which works in a very similar way.

13.6 Customizing Your Environment

You can customize your S-Plus environment a little by defining a `.First` function. Here is an example:

```
  > .First ← function() {
      attach("/support/1/s", pos = 4)  # get access to data frames
                                       # in another directory
      library(Hmisc,  T)
      library(Design, T)
      options(digits=4)  # default no. digits for printing
      invisible()        # makes .First not print anything
    }
```

The `.First` function contains commands that we want executed each time we start S-Plus. For UNIX S-Plus, If you want this function to be the same in all your subdirectories, you don't need to type it again. It is enough if you copy it to your `.Data` subdirectory. You can also have a `.Last` function as well. The `store` function in the Hmisc library creates a `.Last` function to delete temporary objects before S-Plus exits.

Bibliography

[1] R.A. Becker, J.M. Chambers, A.R. Wilks, *The New S Language*, Wadsworth & Brooks/Cole, 1988.

[2] J.M. Chambers, T.J. Hastie, *Statistical Models in S*, Wadsworth & Brooks/Cole, 1992. 171

[3] P. Spector, *An Introduction to S and S-Plus*, Duxbury Press, 1994. 90

[4] A. Krause and M. Olson, *The Basics of S and S-Plus*, New York: Springer–Verlag, Second Edition, 2000.

[5] L. Lam, *An Introduction to* S-Plus *for Windows*, Amsterdam: CANdiensten, 2000.

[6] MathSoft Data Analysis Products Division, S-Plus *Student Edition User's Guide*, Pacific Grove CA: Duxbury Press, 1999.

[7] MathSoft Data Analysis Products Division, *S-Plus 2000 User's Guide*, Seattle, 1999. 216

[8] MathSoft Data Analysis Products Division, *S-Plus 2000 Guide to Statistics*, Seattle, 1999.

[9] MathSoft Data Analysis Products Division, *S-Plus 2000 Programmer's Guide*, Seattle, 1999.

[10] Venables, William N. and Ripley, Brian D., *Modern Applied Statistics with S-Plus, Third Edition*, New York; Springer–Verlag, 1999.

[11] Venables, William N. and Ripley, Brian D., *S Programming*, New York; Springer–Verlag, 2000.

[12] Harrell, Frank E., Regression Modeling Strategies *with Applications to Linear Models, Logistic Regression, and Survival Analysis*, New York; Springer–Verlag, 2001.

[13] Harrell, Frank E., Lee, Kerry L., and Mark, Daniel B., Multivariable prognostic models: Issues in developing models, evaluating assumptions and adequacy, and measuring and reducing errors. *Statistics in Medicine* 15:361–387, 1996. 176

[14] Harrell, Frank E., *et al.*, Development of a clinical prediction model for an ordinal outcome. *Statistics in Medicine* 17:909-944. 176

[15] Faraway, JJ, The cost of data analysis. *Journal of Computation and Graphical Statistics* 1:213-229, 1992. 150, 154

[16] Breiman L, Friedman J: Estimating optimal transformations for multiple regression and correlation (with discussion). *Journal of the Americal Statistical Association* 80:580-619, 1985. 150

[17] Tibshirani, R, Estimating transformations for regression via additivity and variance stabilization. *Journal of the American Statistical Association* 83:394-405, 1989. 150

[18] Feng Z, McLerran D, Grizzle J, A comparison of statistical methods for clustered data analysis with Gaussian error. *Statistics in Medicine* 15:1793-1806, 1996. 159

[19] Tibshirani R, Knight K, Model search and inference by bootstrap "bumping". Technical Report, Department of Statistics, University of Toronto. (`http://www-stat.stanford.-edu/ tibs`). Presented at the Joint Statistical Meetings, Chicago, August 1996. 160

[20] Rosner, B, *Fundamentals of Biostatistics, Fourth Edition*, Belmont CA: Duxbury Press, 1995.

Index