Introduction to Lisp-Stat and its applications.

Angelo J. Canty

Department of Mathematics
Swiss Federal Institute of Technology
1015 Lausanne
Switzerland

June 2, 1998

Chapter 1

1.1 Introduction

`Lisp-Stat` is a statistical programming language based on the computer programming language Lisp. `Lisp-Stat` was developed at the University of Minnesota by Luke Tierney in the late 1980's. The primary reference is the book "`Lisp-Stat`: An Object-oriented Environment for Statistical Computing and Dynamic Graphics" (Tierney, 1990) on which much of this course will be based. Tierney has also published some further work on `Lisp-Stat` as it has developed and most of this can be found at the WWW site

`http://www.stat.umn.edu/~luke/xls/xlsinfo/xlsinfo.html`

Further material for this course was taken from short courses given by Luke Tierney and also a course given by Nancy Reid.

The Lisp language was chosen for this project as it lends itself very well to object-oriented programming which is important for interactive dynamic graphics. It also has the advantage that functions can be passed around in the same way as data. This is especially useful in statistics where we often want to examine density or distribution functions. Currently there is only one implemented version of `Lisp-Stat` and that is based on the Xlisp version of Lisp. This implementation is available, from the previously mentioned WWW page, for Unix systems running X11, Microsoft Windows and for the Macintosh operating systems.

There are two types of objects in Lisp: simple objects and compound objects. A simple object is a single object which may be an integer, floating-point number, a string (`"This is a string"`), a symbol such as a variable or function name or a logical (`t` is true, `nil` is false). The basic compound object in Lisp is a *list* which is surrounded by parentheses. All lists are treated in the same way regardless of their contents. That is all lists are evaluated unless the interpretor is specifically told not to evaluate the list. A list is evaluated by assuming that the first item in the list is a function call and the remaining items are the arguments. Each argument is evaluated in turn and the results passed to the function. The most common way of stopping execution of a list or other item is by *quoting* the list.

```
> (+ 1 2 3)
6
> '(+ 1 2 3)
(+ 1 2 3)
```

`Lisp-Stat` also has other types of compound items which are preceded by a special *read macro*. This allows objects to represent vectors or arrays. We shall see some of these later.

To start the `Xlispstat` interpreter on our system we use the command `xlispstat` from the Unix shell. We then see a short message and get the `Xlispstat` prompt

```
XLISP-PLUS version 2.1g
Portions Copyright (c) 1988, by David Betz.
Modified by Thomas Almy and others.
XLISP-STAT 2.1 Release 3.44 (Beta).
Copyright (c) 1989-1994, by Luke Tierney.
>
```

Once we have finished we leave the interpreter using the function

```
> (exit)
```

The function `help` allows us to access brief documentation associated with a function. When we do not know the exact name of the function then `help*` can be used to return the documentation associated with all functions whose name contains a given string.

1.2 Data Input

In order to analyze a dataset we must first input it to the interpreter. We will normally want to give the input dataset a name which can be used in many subsequent calculations. The macro `def` is used to do this. The simplest way of introducing data is to simply type it in as a list and store it using `def`. The following data record the length of throws in the shot-put at the English championships of 1991.

```
> (def shot-put '(17.79 17.21 16.47 16.27 15.53 14.92 14.72 14.63
14.23 13.70 14.52 14.46 13.22 11.75 16.31 15.81 14.98 14.30 14.05 13.66))
SHOT-PUT
```

Some things to note about this are that `def` does not work like a regular function since its first argument is not evaluated. The second argument is evaluated and so we need to quote the list. Also note that elements in a list are separated by one or more spaces.

Often we will have our data stored in some external file and wish to read this file into a variable in the interpreter. `read-data-file` takes a single argument which is a string giving the name of a data file. It then reads the elements in that file and returns a single list containing all of the data. `read-data-columns` is used for input which is input in columns from a file. As well as the string for the file name this function takes an integer argument `cols` giving the number of columns in the file. The function returns a list of `cols` lists. Suppose that we have a file called `datafile` which looks like

```
 1  2  3
 4  5  6
 7  8  9
10 11 12
```

We then wish to read this into a variable `junk`

```
> (def junk (read-data-file "datafile"))
JUNK
> junk
(1 2 3 4 5 6 7 8 9 10 11 12)
> (def junk (read-data-columns "datafile" 3))
JUNK
> junk
((1 4 7 10) (2 5 8 11) (3 6 9 12))
> (undef 'junk)
JUNK
```

The function `undef` removes the binding between a symbol name and its contents. When all bindings to a location are removed the location is marked for garbage collection. Note that `read-data-columns` does not count the number of columns in the data file. It simply reads the data as if they were all on one line and assigns them to sublists row-wise.

```
(def junk (read-data-columns "datafile" 2))
JUNK
> junk
((1 3 5 7 9 11) (2 4 6 8 10 12))
> (undef 'junk)
JUNK
```

1.2.1 Generating Data

A facility that is regularly useful in mathematical and statistical applications is to generate systematic data. `Lisp-Stat` provides two functions for this depending on the type of output expected. `iseq` generates a sequence of consecutive integers and `rseq` generates a sequence of equally spaced real numbers. `iseq` takes one or two integer arguments. If only one argument, n, is given then it returns a list of the integers $0, 1, 2, \ldots, n-1$. If two arguments m and n are given then `iseq` returns the list of integers with first element m and last element n. `rseq` takes three arguments; the first element of the sequence, the final element and the length of list to be returned.

Another useful function for generating data in a particular pattern is `repeat`. This takes a list of values and a pattern. The pattern can be an integer or a list of the same length as the list of values.

```
> (iseq 3)
(0 1 2)
> (repeat (iseq 3) 2)
(0 1 2 0 1 2)
> (repeat (iseq 3) '(1 0 3))
(0 2 2 2)
```

In statistics it is often useful to generate pseudo-random numbers. There are a number of functions available in `Lisp-Stat` to do this. The function `random` takes a single integer argument, m, and returns a random integer from the sequence $0, \ldots, m-1$. This function, as most `Lisp-Stat` functions, has been vectorized to work on a list of arguments. A related function is `sample` which returns a sample of given size from a list of numbers. The sample can be drawn with or without replacement. There are 10 other functions which return random numbers from a specified distribution. These functions are given in Table 1.1.

A recent innovation in `Lisp-Stat` is that it now allows the use of four different random number generators. The original generator implemented was Marsaglia's lagged Fibonacci generator. This generator is quite old and does not have a sufficiently long cycle for large simulations. The new default generator is L'Ecuyer's version of the Wichman-Hill generator. Two other generators are also available; Marsaglia's Super-Duper generator (as used in `S`) and a combine Tauseworthe generator of Tezuka and L'Ecuyer. To incorporate these four generators, the random-state contains information on the current generator and the current seed. Both the seed and the generator can be changed using the function `make-random-state`. This function can also return a copy of the current state which is stored in a variable called `*random-state*`. It is initialized and changed by reference to the system clock.

Table 1.1: *Functions to generate pseudo-random numbers from specified distributions.*

Name	Arguments	Description
`binomial-rand`	`k n p`	`k` draws from a Binomial(`n`,`p`).
`poisson-rand`	`n mu`	`n` draws from a Poisson(`mu`).
`beta-rand`	`n a b`	`n` Beta(`a`,`b`) random variates.
`cauchy-rand`	`n`	`n` standard Cauchy random variates.
`chisq-rand`	`n df`	`n` Chi-squared(`df`) random variates.
`f-rand`	`n df1 df2`	`n` F(`df1`,`df2`) random variates.
`gamma-rand`	`n a`	`n` Gamma(`a`,1) random variates.
`normal-rand`	`n`	`n` standard normal random variates.
`t-rand`	`n df`	`n` t(`df`) random variates.
`uniform-rand`	`n`	`n` uniform(0,1) random variates.

1.2.2 Sample Data sets

`Xlispstat` comes with some data sets and examples provided. There are special functions which allow us to use these. They are `load-data` and `load-example`. Both of these take a string which gives the name of a file in the Data or Examples directory for `Xlispstat` and executes the code in that file.

```
> (load-data "book")
; loading "/usr/local/lib/xlispstat/Data/book.lsp"
; loading "/usr/local/lib/xlispstat/Data/tutorial.lsp"
; loading "/usr/local/lib/xlispstat/Data/oxygen.lsp"
; loading "/usr/local/lib/xlispstat/Data/puromycin.lsp"
; loading "/usr/local/lib/xlispstat/Autoload/nonlin.fsl"
; loading "/usr/local/lib/xlispstat/Data/aircraft.lsp"
; loading "/usr/local/lib/xlispstat/Autoload/maximize.fsl"
; loading "/usr/local/lib/xlispstat/Data/leukemia.lsp"
; loading "/usr/local/lib/xlispstat/Autoload/bayes.fsl"
; loading "/usr/local/lib/xlispstat/Data/stackloss.lsp"
T
> (variables)
(ABRASION-LOSS ABSORPTION AGE AIR ALUMINUM CO CONC CPK DENSITY DEPTH ETHANOL
FAILURE-TIMES FLOW GAS1 GAS2 HARDNESS HC IRON LOSS MOIST OXYGEN PRECIPITATION
```

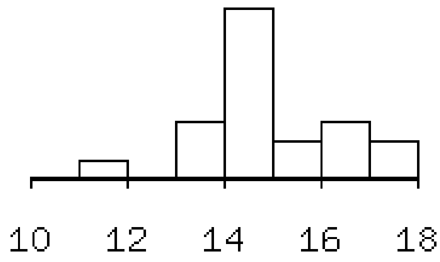

Figure 1.1: *Histogram of distances of shot-put throws.*

```
PURCHASES RATIO RURAL SEPARATION SHOT-PUT STRENGTH SUGAR TEMP TENSILE-STRENGTH
TIMES-POS TRANSFORMED-WBC-POS TRAVEL-SPACE URBAN VARIETY WATER WBC-POS X X1 X2
Y1 Y2 YIELD)
```

The function `variables` returns a list of all variables defined (using `def`) at the present time. Note that symbol names in lisp are not case sensitive.

1.2.3 Univariate Statistics

`Lisp-Stat` also has quite a number of statistical functions which are not in standard lisp implementations. These typically expect a list as their argument and interpret that list as a dataset.

```
> (mean shot-put)
14.9265
> (median shot-put)
14.675
> (standard-deviation shot-put)
1.43147
> (interquartile-range shot-put)
1.90
```

Simple univariate plots can be generated using `histogram` and `boxplot`

```
> (histogram shot-put)
#<Object: 269123968, prototype = HISTOGRAM-PROTO>
> (boxplot shot-put)
#<Object: 269197856, prototype = SCATTERPLOT-PROTO>
```

The output of these plots are given in Figure 1.1 and Figure 1.2. For now we need not worry about the returned values of these functions. Each plot opens a new window automatically. The window has a pull down menu which allows us to interact with the plot in various ways.

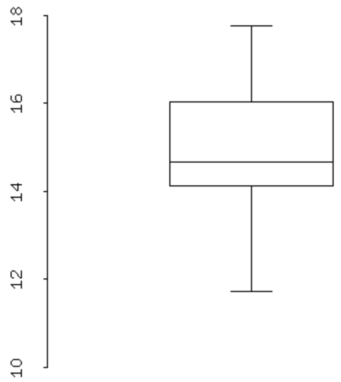

Figure 1.2: *Boxplot of distances of shot-put throws.*

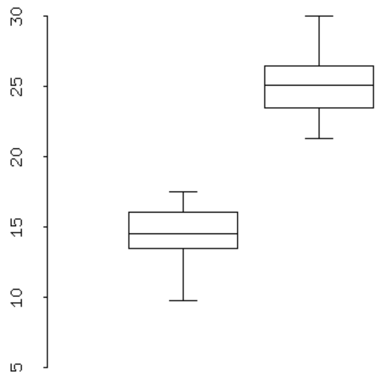

Figure 1.3: *Parallel boxplots of mileages per gallon of petrol in 1965 (left) and 1975 (right).*

`boxplot` can also be used to produce parallel boxplots of two or more datasets by supplying a list of lists. `gas1` and `gas2` are 50 independent measurements on mileage per gallon of petrol in 1965 and 1975. In Figure 1.3 we present parallel boxplots which clearly show the increase in efficiency in 1975 cars compared to those in 1965.

```
> (boxplot (list gas1 gas2))
#<Object: 269486048, prototype = SCATTERPLOT-PROTO>
```

1.2.4 Two-Dimensional Plots

The functions `plot-points` and `plot-lines` are used to create simple two-dimensional plots. The variables `air`, `temp`, `conc` and `loss` contain Brownlee's famous stack loss data which contains obser-

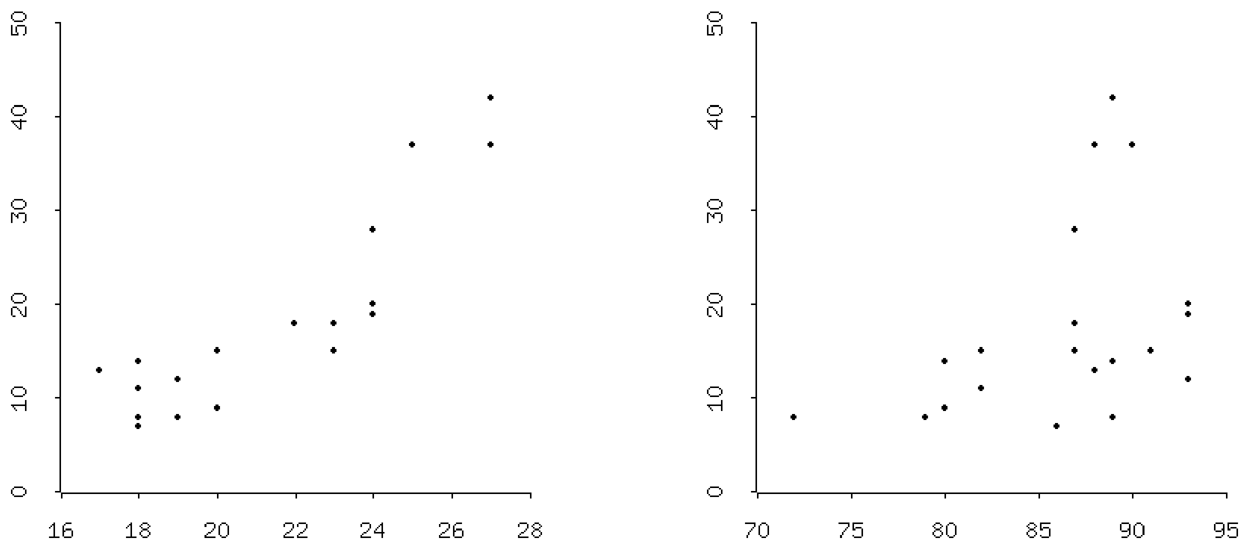

Figure 1.4: *Plots of* `loss` *against the predictor variables* `temp` *(left) and* `conc` *(right).*

vations for the oxidation of ammonia in the production of nitric acid. The predictor variables are the air flow, temperature of the cooling water, the percentage acid concentration over 50% multiplied by 10 and the dependent variable is the percentage of ammonia which escapes unabsorbed multiplied by 10. Plots of the dependent variable against two of the predictors are given in Figure 1.4.

```
> (plot-points temp loss)
#<Object: 269115616, prototype = SCATTERPLOT-PROTO>
> (plot-points conc loss)
#<Object: 269430944, prototype = SCATTERPLOT-PROTO>
```

These two plots can be *linked* by clicking on the appropriate entry in the menu for both plots. With this linking we can select plots in one plot and see them highlighted in both plots. The selection is done by marking an area of one plot using the mouse. The symbol and colour of the selected points can be changed using the menu. The slicer in the menu allows us to select points for ranges of another variable.

A very useful function is `plot-function` which allows us to plot a function over a given range. By default 50 equally-spaced points are used but the number can be altered using an optional argument. The following two code fragments produce the same plot which is shown in Figure 1.5.

```
> (plot-lines (rseq (- pi) pi 50) (sin (rseq (- pi) pi 50)))
#<Object: 269261200, prototype = SCATTERPLOT-PROTO>
```

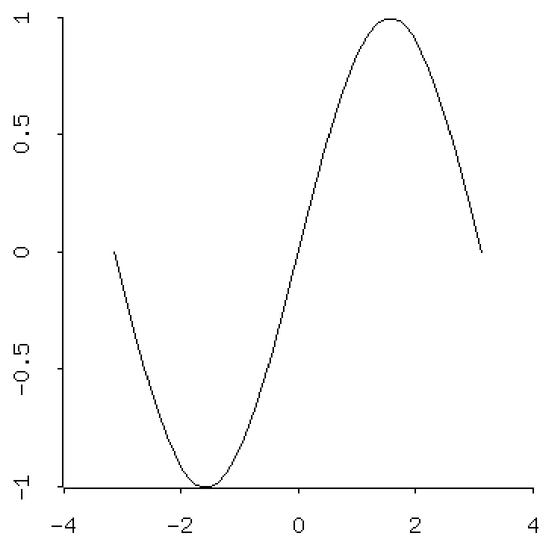

Figure 1.5: *Plot of the sin function between* $-\pi$ *and* π.

```
> (plot-function #'sin (- pi) pi)
#<Object: 269304656, prototype = SCATTERPLOT-PROTO>
```

1.2.5 Other Graphics

`Lisp-Stat` provides other graphics which can be used for plotting multi-dimensional data. The most widely used of these is the function `spin-plot` for three-dimensional plots. As with all `Lisp-Stat` graphics we can interact with the plot produced by `spin-plot` through menus nut there is an added feature of these plots in that there are buttons at the bottom of the plot which allow us to rotate the point cloud in various directions. This interaction with the plot allows us to examine the points for any relationships or clustering which may not be apparent from the usual viewpoint.

Another plot for multi-dimensional data is `scatterplot-matrix`. Figure 1.6 shows a scatterplot matrix for all four variables in the stack loss example.

```
> (scatterplot-matrix (list air temp conc loss)
     :variable-labels '("air" "temp" "conc" "loss"))
#<Object: 269546496, prototype = SCATMAT-PROTO>
```

The argument `variable-labels` is a *keyword* argument which is similar to an optional argument except that it must be proceeded by a colon (:). In a later chapter we will look at how functions are defined and describe in detail the types of arguments.

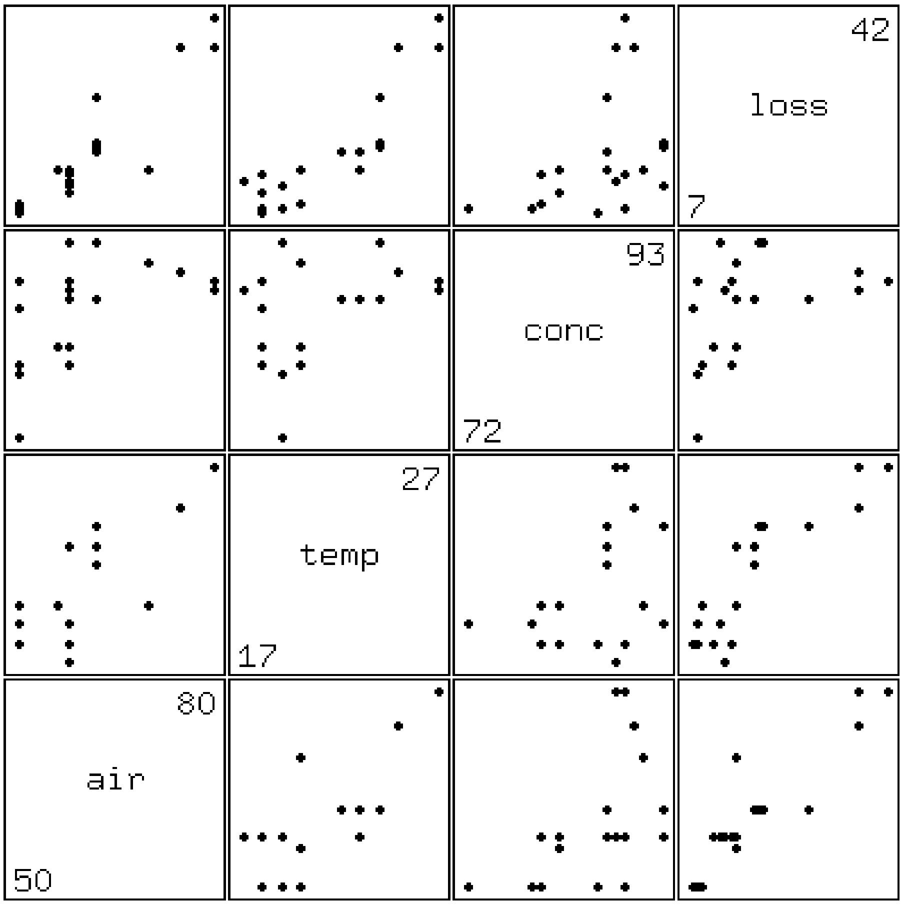

Figure 1.6: *Scatterplot matrix for the stack loss data.*

1.3 Interacting with Plots

In the preceding sections we have seen that we can interact with plots in a limited way through the menu in the plot window. There are many other ways that one can interact with an individual plot and these are based on the idea of object-oriented coding. In the `Xlispstat` code fragments you will have noticed that when make a plot there is a returned value which prints out as something like

```
#<Object: 269261200, prototype = SCATTERPLOT-PROTO>
```

This is a representation of a `plot object`. This object can be assigned to a variable name and thus we can interact with the plot. An object is a list of named components called slots which inherits from a given prototype. We can interact with an object by sending messages to it. A message is sent to an object using the `send` function. How a message is interpreted by an object depends on its prototype. One useful message is the `:help` message which lists all other messages which returns a list all the messages which can be sent to a given object. Let us return to the first scatter-plot we did for the stack loss data.

```
> (def plot1 (plot-points temp loss))
PLOT1
> (send plot1 :help)
loading in help file information - this will take a minute ...done
SCATTERPLOT-PROTO
Scatterplot
Help is available on the following:

ABLINE ACTIVATE ADD-FUNCTION ADD-LINES ADD-METHOD ADD-MOUSE-MODE
ADD-POINTS ADD-SLOT ADJUST-POINTS-IN-RECT ADJUST-SCREEN ADJUST-SCREEN-POINT
ADJUST-TO-DATA ALL-POINTS-SHOWING-P ALL-POINTS-UNMASKED-P ALLOCATE
ANY-POINTS-SELECTED-P APPLY-TRANSFORMATION BACK-COLOR BRUSH BUFFER-TO-SCREEN
CANVAS-HEIGHT CANVAS-TO-REAL CANVAS-TO-SCALED CANVAS-WIDTH CENTER
CHOOSE-MOUSE-MODE CLEAR CLEAR-LINES CLEAR-MASKS CLEAR-POINTS CLICK-RANGE
CLIP-RECT CLOSE CONTENT-ORIGIN CONTENT-RECT CONTENT-VARIABLES COPY-TO-CLIP
CURRENT-VARIABLES CURSOR CUT-TO-CLIP DELETE-DOCUMENTATION DELETE-METHOD
DELETE-MOUSE-MODE DELETE-SLOT DISPOSE DO-CLICK DO-IDLE DO-KEY DO-MOTION
DOC-TOPICS DOCUMENTATION DRAG-GREY-RECT DRAW-BITMAP DRAW-BRUSH DRAW-COLOR
DRAW-DATA-LINES DRAW-DATA-POINTS DRAW-LINE DRAW-MODE DRAW-POINT DRAW-STRING
DRAW-STRING-UP DRAW-SYMBOL DRAW-TEXT DRAW-TEXT-UP ERASE-ARC ERASE-BRUSH
ERASE-OVAL ERASE-POLY ERASE-RECT ERASE-SELECTION ERASE-WINDOW ......
NIL
```

Note that message names are always preceded by a colon. To get help on one of these we can use the `:help` message again this time giving another message as argument.

```
> (send plot1 :help :abline)
ABLINE
Message args: (a b)
Adds the graph of the line A + B x to the plot.
NIL
```

Now suppose we wish to fit a linear and non-linear fit to this data and compare them graphically. A linear regression returns the fitted line $\texttt{loss} = -41.91 + 2.817\texttt{temp}$. In the next chapter we shall see how a regression is fit in `Lisp-Stat` and details of the object returned. The fitted values of a

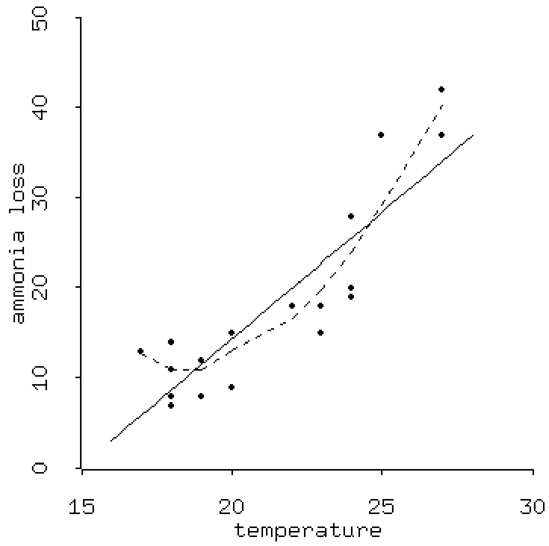

Figure 1.7: *Plot of ammonia loss against temperature for the stack loss data with fitted linear regression (solid) and lowess fit (dashed).*

non-linear fit are returned by the function `lowess` which uses a local smoother and robust weight function using the fraction `f` of the data around each point. The algorithm used is identical to that implemented in `S-Plus`. We shall also add labels to the axes to make the plot more readable. The message `:variable-label` takes two arguments; the first is a list of indices for the variables and the second is the list of corresponding labels. It is important to note that all indices in Lisp start at 0. The final plot is shown in Figure 1.7

```
> (send plot1 :abline -41.91 2.817)
NIL
> (send plot1 :add-lines (lowess temp loss :f 0.6) :type 'dashed)
NIL
> (send plot1 :variable-label '(0 1) '("temperature" "ammonia loss"))
("temperature" "ammonia loss")
```

In later chapters we shall see further ways of interacting with graphics and creating dynamic graphics.

Chapter 2

Statistics with Lisp-Stat

In this chapter we shall look at the statistical methods which are inbuilt to `Lisp-Stat`. It is important to note that `Lisp-Stat` is not designed as a statistical analysis package rather as a programming environment which allows the programmer to write procedures which they can use to analyze their data. This allows much greater flexibility but also means that the user must learn how to program in the language and, of course, know how to implement statistical procedures. As with other programming environments there is a growing user base and users frequently submit programs they have written to archives at `Statlib` or to the new archive maintained by Jan de Leeuw at `http://www.stat.ucla.edu/bookmarks/bookmarks.phtml?book=xls&parent=/` Unfortunately this latter archive is still in the process of getting started so there is not much there at present but that should change quite soon. If you need to do a certain type of analysis, then it is always a good idea to check to see if someone has already done the programming. We will not consider these user-provided procedures here but instead just look at the limited procedures which are built in to the package.

For the remainder of these notes we shall omit the value returned by any commands when it is not directly of interest.

2.1 Linear Regression

The basic function for doing a linear regression is `regression-model`. This function takes two required arguments, the matrix of predictor variables and the response vector. It also takes a number of keyword arguments to specify if an intercept should be included, if the result should be printed, weights for non-homogeneous variances and which observations should be included in finding the estimates. We can also specify names for the predictor and response variables as well as individual cases through keyword arguments. Regression is implemented using `Lisp-Stat`'s object oriented tools and so the returned value is an object to which messages can be sent if it is given a name. These messages allow us to look at various aspects of the fit either numerically or graphically.

```
> (def model1
       (regression-model (list air temp conc) loss
                         :response-name "Ammonia Loss"
                         :predictor-names
                           (list "Air Flow" "Water Temp." "Acid Conc.")))

Least Squares Estimates:

Constant                  -39.9197      (11.8960)
Air Flow                  0.715640      (0.134858)
Water Temp.                1.29529      (0.368024)
Acid Conc.               -0.152123      (0.156294)

R Squared:                0.913577
Sigma hat:                 3.24336
Number of cases:                21
Degrees of freedom:             17
```

The numbers in parentheses after the coefficient estimates are the standard errors. In order to use parts of the fitted model we need to know which messages to send to the regression object. As with plots this is done by sending the `help` message.

```
> (send model1 :help)
REGRESSION-MODEL-PROTO
Normal Linear Regression Model
Help is available on the following:

ADD-METHOD ADD-SLOT BASIS CASE-LABELS COEF-ESTIMATES COEF-STANDARD-ERRORS
COMPUTE COOKS-DISTANCES DELETE-DOCUMENTATION DELETE-METHOD DELETE-SLOT DF
DISPLAY DOC-TOPICS DOCUMENTATION EXTERNALLY-STUDENTIZED-RESIDUALS
FIT-VALUES GET-METHOD HAS-METHOD HAS-SLOT HELP INCLUDED INTERCEPT
INTERNAL-DOC ISNEW LEVERAGES METHOD-SELECTORS NEW NUM-CASES NUM-COEFS
NUM-INCLUDED OWN-METHODS OWN-SLOTS PARENTS PLOT-BAYES-RESIDUALS
PLOT-RESIDUALS PRECEDENCE-LIST PREDICTOR-NAMES PRINT PROTO R-SQUARED
RAW-RESIDUALS REPARENT RESIDUAL-SUM-OF-SQUARES RESIDUALS RESPONSE-NAME
RETYPE SAVE SHOW SIGMA-HAT SLOT-NAMES SLOT-VALUE STUDENTIZED-RESIDUALS
```

```
SUM-OF-SQUARES SWEEP-MATRIX TOTAL-SUM-OF-SQUARES WEIGHTS X X-MATRIX XTXINV Y
```

We can now send any of these messages to our object `model1` to better examine the fitted regression model. Let us first examine the t-statistics for the regression coefficients.

```
> (/ (send model1 :coef-estimates) (send model1 :coef-standard-errors))
(-3.3557 5.3066 3.5196 -0.9733)
> (- 1 (t-cdf (abs *) (send model1 :df))
(0.0019 2.9E-5 0.0013 0.1720)
```

The symbol `*`, when not the first element of a list, is shorthand for the result of the last expression. We see that the variable `conc` is not significant in the fit given the other variables.

As always plots of the output of a regression model are important. In the following we plot the residuals and the Cook's distances against the fitted values to examine leverage of individual points. By linking the two plots and selecting the `Show Labels` menu item on both plots we can identify individual points using the mouse.

```
> (def plot1 (send model1 :plot-residuals))
> (send plot1 :variable-label '(0 1) '("Fitted Values" "Residuals"))
("Fitted Values" "Residuals")
> (def plot2 (plot-points (send model1 :fit-values)
                          (send model1 :cooks-distances)))
> (send plot2 :variable-label '(0 1) '("Fitted Values" "Cook's Distances"))
```

There is one point which has a much higher Cook's distance than any of the others. By clicking on this point we see that it is observation 20 (starting from 0) Two other plots which may be useful are a normal probability plot of the studentized residuals and a plot Bayesian residual plot. This latter plot is described in Chaloner and Brant (1988) and is a Bayesian method of outlier detection in regression problems. It is basically a plot of the studentized residuals against the fitted values except that error bars are put on each point. These error bars represent ± 2 standard deviations of the posterior distribution of the realized errors based on an improper uniform prior on the regression coefficients. The observation 20 also appears as an outlier on these further plots. The four plots with this outlying observation selected are shown in Figure 2.1.

```
> (def plot3 (quantile-plot (send model1 :studentized-residuals)))
> (send plot3 :variable-label '(0 1) '("Normal Quantiles" "Studentized Residuals"))
> (send plot3 :abline 0 1)
NIL
> (def plot4 (send model1 :plot-bayes-residuals))
> (send plot4 :variable-label '(0 1) '("Fitted Values" "Studentized Residuals"))
```

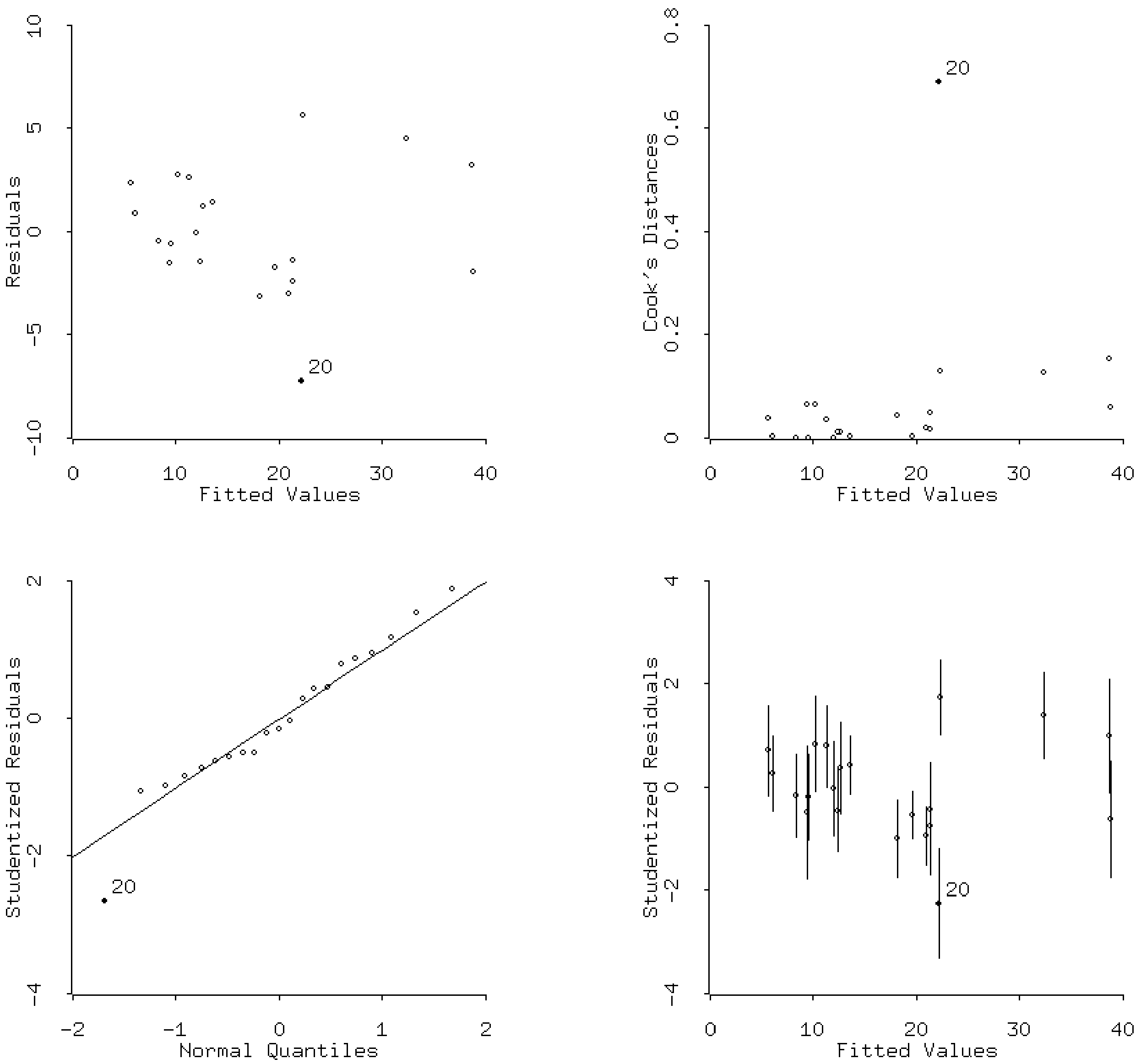

Figure 2.1: *Regression diagnostic plots for the stack loss data. The top two plots are of the raw residuals and Cook's distances against the fitted values. The lower two plots are a normal Q-Q plot of the studentized residuals with the line $y = x$ and a Bayesian residual plot. The highlighted point is a point of high influence.*

Now that we have identified observation 20 as an outlier we could refit the regression without that observation to further see the effect it has on the fit of the model. We do this by using the `included` keyword argument with a vector of logical values all of which are `t` except the last which is set to `nil`. The effect in general is to reduce the residual standard error and hence the standard errors of the coefficients. The actual values of the coefficients do not change very much.

```
> (def model2
       (regression-model (list air temp conc) loss
                         :response-name "Ammonia Loss"
                         :predictor-names
                            (list "Air Flow" "Water Temp." "Acid Conc.")
                         :included (append (repeat 't 20) (list nil))))
Least Squares Estimates:

Constant                  -43.7040      (9.49157)
Air Flow                  0.889108      (0.118848)
Water Temp.               0.816620      (0.325029)
Acid Conc.               -0.107141      (0.124541)

R Squared:                0.948795
Sigma hat:                 2.56920
Number of cases:                21
Number of cases used:           20
Degrees of freedom:             16
```

2.2 Generalized Linear Models

For many types of data a regular linear regression is not very useful. One reason for this may be that the error distribution is not thought to be normal. Generalized linear models (McCullagh and Nelder, 1989) can be used in this case. These are now becoming quite commonly used methods and have been implemented in many statistical packages. The original implementation of `Xlispstat` did not include any functions to fit generalized linear models but a limited selection of fits are now possible. These were first described in a University of Minnesota technical report by Luke Tierney in 1991 and have not changed much since then. They are still not part of the main `Xlispstat` system but they are distributed with it and can be loaded into the interpreter using

```
> (load "glim")
```

The functions available are `poissonreg-model`, `gammareg-model` and `binomialreg-model` and each returns an object similar to that returned by `regression-model`. The first two functions take the same two required arguments as `regression-model`, `binomialreg-model` also has a third required argument which specifies the denominator for each binomial. All keyword arguments which can

Table 2.1: *Cancer cell activity, LI and remission indicator, r for 27 patients. From Freeman (1987).*

LI	0.4	0.4	0.5	0.5	0.6	0.6	0.6	0.7	0.7	0.7	0.8	0.8	0.8	0.9
r	0	0	0	0	0	0	0	0	0	0	0	0	0	1
LI	1.0	1.0	1.0	1.1	1.1	1.2	1.3	1.4	1.6	1.7	1.9	1.9	1.9	
r	0	1	1	0	1	0	1	1	0	1	0	1	1	

be passed to `regression-model` can also be passed to these functions and in addition there are four keyword arguments specific to generalized linear models. `link` can be used to specify a link structure other than the default canonical link, `offset` is used to specify an offset, `pweights` should be used in place of `weights` to specify prior weights and `verbose` can be set to `nil` to prevent the default printing of the deviance for each iteration.

2.2.1 Example of a Binomial Regression

Table 2.1 gives $n = 27$ pairs of observations. LI is a measure of the activity of cancer cells and r is an indicator of whether or not the patient went into remission. We wish to fit a model to the relationship between cell activity and remission. A simple model for this problem is a binary logistic regression. This model simply says that the log-odds of remission is a linear function of the measure of cell activity.

The following code defines the data and fits a binomial regression model. Note that there is no need to specify the `link` argument in this case since we wish to use the canonical logit link for the binomial family.

```
> (def LI (repeat (rseq 0.4 1.9 16) '(2 2 3 3 3 1 3 2 1 1 1 0 1 1 0 3)))
> (def remission '(0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 1 0 1 0 1 1 0 1 0 1 1))
> (def cancer-mod (binomialreg-model LI remission (repeat 1 27)
                                     :predictor-names '("LI")))
Iteration 1: deviance = 26.5268
Iteration 2: deviance = 26.0798
Iteration 3: deviance = 26.0730

Weighted Least Squares Estimates:

Constant                    -3.77714      (1.37812)
```

```
LI                          2.89726       (1.18642)

Scale taken as:             1.00000
Deviance:                   26.0730
Number of cases:                 27
Degrees of freedom:              25
```

In this example we did not set the `print` argument to `nil` so a summary of the model is printed out. If we wished to print it out again at some later stage we use the `:display` message.

The fitted model corresponds to the equation

$$\log\left(\frac{p(LI)}{1-p(LI)}\right) = -3.78 + 2.90LI \tag{2.1}$$

where $p(LI)$ is the probability of a patient going into remission given that their measure of cell activity is the value LI. In the left panel of Figure 2.2 we fit the (jittered) data and two curves. The solid curve on the plot is the fitted curve as given in Equation (2.1). In the left panel we plot the Cook's distances against the fitted linear predictors. There is one point which is clearly very influential. On linking the two plots, choosing the `Show labels` menu item and selecting this influential point we see that it is observation 24 which is an individual with a very high level of LI who did not go into remission. To further examine the influence of this point we refit the model on the 26 points omitting this point. The fitted curve is shown as the dashed curve in the left panel of Figure 2.2.

```
> (def LI.jitter (+ LI (/ (- (uniform-rand 27)) 25)))
> (def plot1 (plot-points LI.jitter remission
                          :variable-labels '("LI (jittered)" "remission")))
> (send plot1 :add-lines LI (send cancer-mod :fit-probabilities))
> (def plot2 (plot-points (send cancer-mod :fit-values)
                          (send cancer-mod :cooks-distances)
                          :variable-labels
                            '("Linear Predictor" "Cook's Distance")))
> (def incl (combine (repeat t 24) nil (repeat t 2)))
> (def cancer-mod1 (binomialreg-model LI remission (repeat 1 27)
                                      :predictor-names '("LI")
                                      :included incl :print nil
                                      :verbose nil))
> (send cancer-mod1 :coef-estimates)
(-5.2151 4.5030)
```

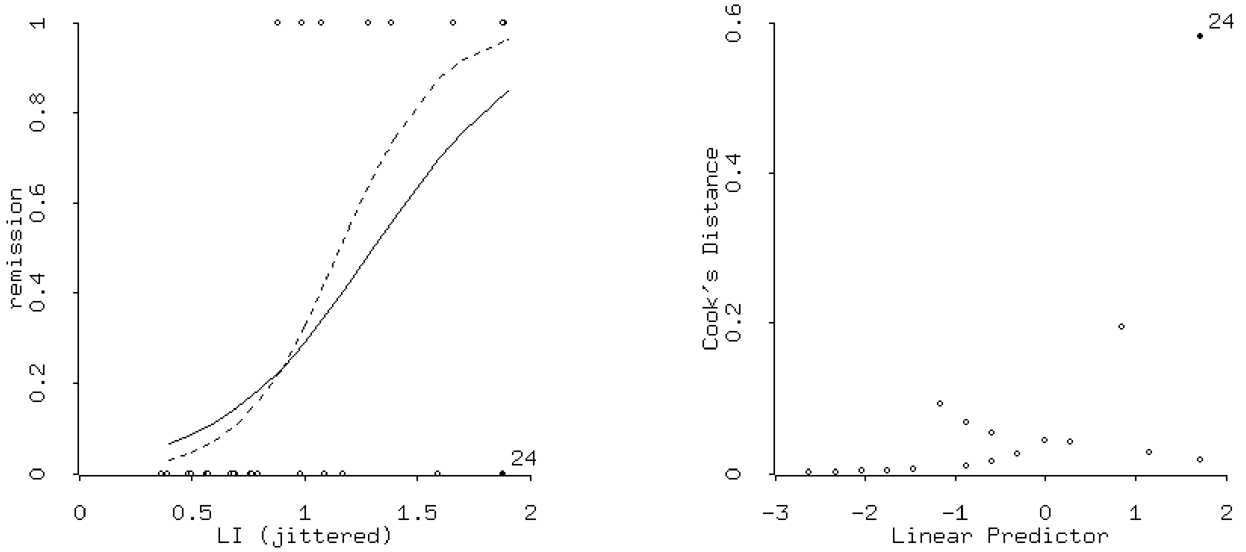

Figure 2.2: *Plots for a binomial model fitted to the cancer remission data. The left panel is a plot of the jittered data with the fitted curves for all of the data (solid) and excluding point 24 (dashed). The right panel is a plot of the Cook's statistics against the fitted linear predictors for the model on the entire data set.*

```
> (send cancer-mod1 :coef-standard-errors)
(1.8752 1.7629)
> (send plot1 :add-lines LI (send cancer-mod1 :fit-probabilities) :type 'dashed)
```

2.2.2 Example of a Poisson Model

Doll and Hill (1966) reported data collected, in 1961, on the death rates of British male doctors due to coronary heart disease. Their primary aim was to examine the relationship between smoking and this disease. We shall use the data as given in Breslow (1985) and shown in Table 2.2. A standard model is that the number of deaths follow a Poisson distribution with mean $n\lambda$ where n is the number of person-years and λ is the death rate due to coronary heart disease. We model this with a log link and offset $\log(n)$. Since the age is not recorded exactly but as a categorical variable we use the mid-point of the categories as the value for that category. We will compare two nested models here. The first model fits a different intercept for each age category.

$$\lambda = \exp(\alpha_{age} + \beta x) \tag{2.2}$$

where x is an indicator of smoking status. The second model fits a linear model to the mid-points

Table 2.2: *Deaths from coronary heart disease among British male doctors as reported in Breslow (1985)*

Age	Person-years		Coronary deaths	
	Non-smokers	Smokers	Non-smokers	Smokers
35–44	18790	52407	2	32
45–54	10673	43248	12	104
55–64	5710	28612	28	206
65–74	2585	12663	28	186
75–84	1462	5317	31	102

of the age categories.

$$\lambda = \exp(\alpha + \beta_1 age + \beta_2 x). \tag{2.3}$$

The only difference in the fitting of these two models is that the first requires `(indicators age)` whereas the second works directly on `age`.

```
> (def age (repeat (* 10 (iseq 4 8)) 2))
> (def smoke (repeat '(0 1) '(5 5)))
> (def person-years '(18790 10673 5710 2585 1462 52407 43248 28612 12663 5317))
> (def deaths '(2 12 28 28 31 32 104 206 186 102))
> (def breslow-mod1
       (poissonreg-model (append (indicators age) (list smoke))
                         deaths :offset (log person-years)
                         :predictor-names
                            (append (level-names age :prefix "Age")
                                    '("Smoke"))))
Weighted Least Squares Estimates:

Constant                  -7.91932      (0.191412)
Age(50)                    1.48400      (0.194793)
Age(60)                    2.62750      (0.183401)
Age(70)                    3.35048      (0.184475)
Age(80)                    3.70009      (0.191904)
Smoke                     0.354534      (0.107297)
```

```
Scale taken as:            1.00000
Deviance:                  12.1324
Number of cases:                10
Degrees of freedom:              4
> (def breslow-mod2
       (poissonreg-model (list age smoke)
                        deaths :offset (log person-years)
                        :predictor-names (list "Age" "Smoke")))
Weighted Least Squares Estimates:

Constant                 -10.6258       (0.209720)
Age                        8.358337E-2   (2.904497E-3)
Smoke                      0.406370      (0.107195)

Scale taken as:            1.00000
Deviance:                  69.1821
Number of cases:                10
Degrees of freedom:              7
```

The difference in deviances is 57.05 on 3 degrees of freedom which is clearly significant. This suggests that there is a non-linear relationship between the log of the Poisson mean and age and so the first model is to be preferred. Using this model the p-value for testing whether smoking has an effect on the death rate due to coronary heart disease is 1.49% which means that smokers have a significantly higher rate of mortality due to coronary heart disease than do non-smokers.

2.3 Non-linear Regression

The function `nreg-model` allows a nonlinear normal regression model to be fitted. The required arguments to this function are the mean function, the response vector and initial estimates for the parameters of the model. The predictor variable should be taken into account by the mean function which is a function of the parameters only. In the following example we shall define a function of the predictors which returns a function of the parameters. Because of the scoping rules of Lisp it is not a problem to do this. We shall come back to the specifics of the coding used here later.

The data which we shall use here are taken from Bates and Watts (1988) and concern the velocity of an enzymatic reaction and the substrate concentration. The actual interest was in the effect of a treatment of the enzyme with Puromycin but here we shall not consider this and simply look at

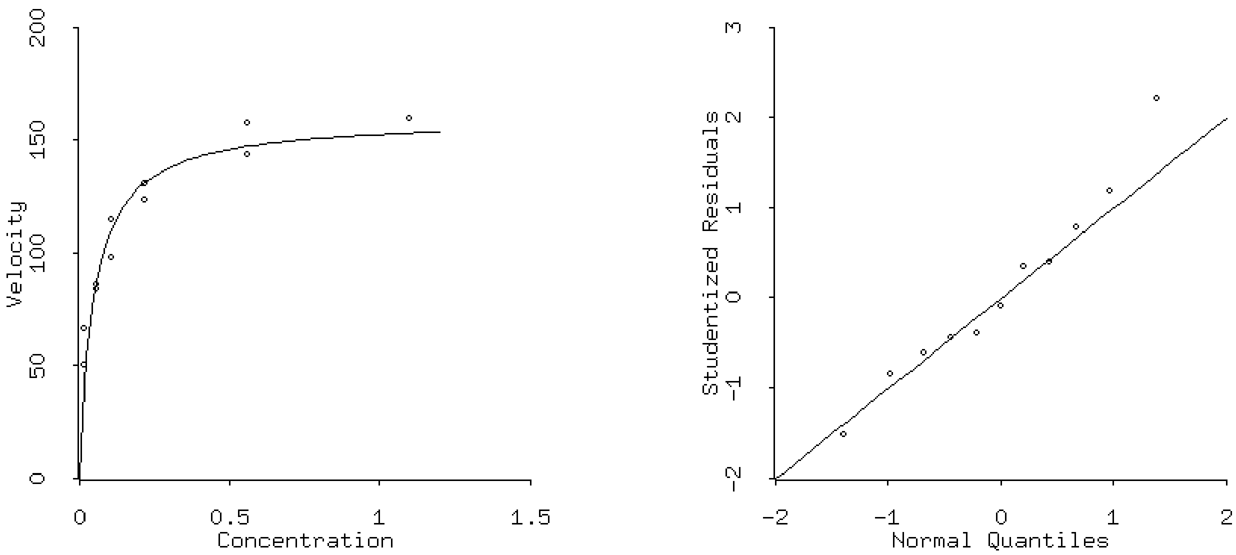

Figure 2.3: *Plots for a normal non-linear model fitted to the enzyme velocity data. The left panel is a plot of the data with the fitted curve. The right panel is a normal Q-Q plot of the studentized residuals with the line $y = x$.*

the untreated observations. A model which is often considered for these data is the mean velocity is related to the concentration through the Michaelis-Menton equation

$$\nu(x) = \frac{\theta_0 x}{\theta_1 + x}.$$

The parameter θ_0 is the asymptote of $\nu(x)$ as $x \longrightarrow \infty$ and the parameter θ_1 is defined as the minimum value of x for which $\nu(x) \geq \theta_0/2$. From the data plot in Figure 2.3 we estimate these values as roughly 160 and 0.1 respectively. The effect of not having good starting values will depend on the mean function being used. The model is fitted using a simple Gauss-Newton algorithm and numerically calculated derivatives. After fitting the model we plot the fitted curve on the left panel of Figure 2.3. The curve seems to describe the data quite well. The right panel contains a normal Q-Q plot of the residuals. Although not perfectly linear there does not seem to be too much evidence against normality in this case.

```
> (def conc '(0.02 0.02 0.06 0.06 0.11 0.11 0.22 0.22 0.56 0.56 1.1))
> (def velocity '(67 51 84 86 98 115 131 124 144 158 160))
> (def plot1 (plot-points conc velocity
                          :variable-labels '("Concentration" "Velocity")))
> (send plot1 :range 1 0 200)
> (send plot1 :y-axis t t 5)
```

```
> (defun michaelis (x)
     #'(lambda (theta) (/ (* (first theta) x) (+ (second theta) x))))
> (def michaelis-mod (nreg-model (michaelis conc) velocity '(160 0.1)))
Residual sum of squares: 6022.55
Residual sum of squares: 2551.03
Residual sum of squares: 939.589
Residual sum of squares: 862.452
Residual sum of squares: 859.672
Residual sum of squares: 859.606
Residual sum of squares: 859.604
Residual sum of squares: 859.604
Residual sum of squares: 859.604

Least Squares Estimates:

Parameter 0              160.280      (6.48024)
Parameter 1              0.04771      (0.00778)

R Squared:              0.935572
Sigma hat:               9.77300
Number of cases:              11
Degrees of freedom:            9
> (defun mich1 (x theta)
     (funcall (michaelis x) theta))
> (send plot1 :add-lines (rseq 0 1.2 50)
                         (mich1 (rseq 0 1.2 50)
                                (send michaelis-mod :coef-estimates)))
> (def plot2 (quantile-plot (send michaelis-mod :studentized-residuals)
                            :variable-labels
                              '("Normal Quantiles" "Studentized Residuals")))
> (send plot2 :abline 0 1)
```

Note that there are no functions for general non-linear regression with distributions other than normal. For the binomial, poisson and gamma this can be achieved using the generalized linear model functions described in the previous section. For a general non-linear function this will require defining the link structure of the generalized linear model. For details on how this may be done see Tierney (1991).

Table 2.3: *Coagulation time (seconds) for blood from 24 animals randomly allocated to four different diets.*

Diet	Coagulation times							
A	62	60	63	59				
B	63	67	71	64	65	66		
C	68	66	71	67	68	68		
D	56	62	60	61	63	64	63	59

2.4 Other Statistical Features

2.4.1 One-way Analysis of Variance

The function `oneway-model` will fit a one-way model to a set of data. The data should be entered as a list of lists with each sub-list containing the observations for one group. As an example consider the data in Table 2.3 taken from Box *et al.* (1978).

We fit the model as follows

```
> (def blood '((62 60 63 59) (63 67 71 64 65 66) (68 66 71 67 68 68)
             (56 62 60 61 63 64 63 59)))
> (def blood-model (oneway-model blood))
Least Squares Estimates:

Group 0                    61.0000      (1.18322)
Group 1                    66.0000      (0.966092)
Group 2                    68.0000      (0.966092)
Group 3                    61.0000      (0.836660)

R Squared:                0.670588
Sigma hat:                 2.36643
Number of cases:                24
Degrees of freedom:             20

Group Mean Square:         76.0000      (3.00000)
Error Mean Square:         5.60000      (20.0000)
```

The returned object inherits from `regression-model-proto` and has many of the same messages.

2.4.2 Approximate Bayesian Computations

The function `bayes-model` allows us to compute first and second order approximations to posterior moments and also a Laplace approximation to a univariate marginal posterior density. These approximations are based on Tierney and Kadane (1986) and Tierney *et al.* (1989) and assume that the posterior distribution is unimodal. The required arguments are the log posterior density and an initial estimate of the mode. The returned value is an object which can be sent messages to get the first order moment approximations (`:1stmoments`), slower but more accurate second order approximations (`:moments`), and a list of marginal density values which can be used to plot the marginal density (`margin1`). Further information and an example can be found in Section 2.8.3 of Tierney (1990).

2.4.3 Maximization

The function `newtonmax` maximizes a function of several variables using an algorithm based on Newton's method with backtracking as described in Dennis and Schnadel (1983). It takes a function and an initial estimate for the location of the maximum and returns the location of the maximum. If the keyword argument `:return-derivs t` is supplied then a list of the location, the maximum function value, the gradient and the Hessian matrix at the maximum are returned. The function normally uses numerical derivatives but this can be changed by having the input function return a list of the function value and the gradient or the function value, the gradient and the Hessian matrix.

The function `newtonmax` works well for functions which are twice continuously differentiable. If this is not true then the function `nelmeadmax` can be used. This implements the Nelder-Mead simplex method as described in Press *et al.* (1992). Since this method does not work with derivatives it is usually slower than `newtonmax` when both are possible.

Further information on these functions and an example can be found in Section 2.8.2 of Tierney (1990).

Chapter 3

Programming in Lisp-Stat

In the previous chapters we looked at some of the statistical and graphical functionality in `Lisp-Stat`. One of the advantages of the language, however, is the ease with which new functionality can be programmed by the user. In this chapter we shall briefly look at some of the basics of programming in `Lisp-Stat`.

3.1 Data Handling

As we have seen the basic compound data item in Lisp is the list which can be bound to a symbol using the macro `def`. Two or more lists can be combined using the functions `combine` and `append`. The difference between these two functions is that `combine` returns a list which contains only non-compound elements whereas `append` merges only the top level lists.

```
> (combine '((1 2) (3 4 5)) '(6 7))
(1 2 3 4 5 6 7)
> (append '((1 2) (3 4 5)) '(6 7))
((1 2) (3 4 5) 6 7)
```

We can access one or more individual components of a list using the function `select`. This takes two arguments; a list or a symbol which evaluates to a list and the index or list of indices of the elements required. Remember that all numbering in Lisp starts at 0 so the first element in a list `x` is extracted using `(select x 0)`. The first 10 elements of a list can be extracted using the functions `first`, `second`, ..., `tenth` which are simply shorthand for the corresponding calls to `select`. If we wish to modify one or more elements of a list then we combine the `select` function with a special form `setf`. A special form behaves differently to functions and macros in how its arguments are evaluated. In the case of `setf` the first argument is evaluated but instead of the value being returned the memory location of the value is returned. The second argument is evaluated as normal and the result stored in that memory location.

Some caution is needed when modifying lists which have been bound to variables. To understand this we need to look a little more closely into how Lisp does symbol binding. When a command such as `(def x '(1 2 3))` is executed, the list `(1 2 3)` is stored in three consecutive words in memory and the location returned. Then a new entry is added to the symbol table which relates the symbol `x` with the memory location. If a new assignment is made which binds a new symbol to one or more elements of `x` then the elements are not copied as in some languages instead only the symbol table is changed to associate the new variable name with the relevant memory location. This method clearly saves memory space but can cause unexpected results for the unwary since any modification to one of the symbols potentially affects the other also. If we wish to force the copying of the value of symbol then we should use the function `copy-list` when doing the assignment.

```
> (def x '(1 (2 3) (4 5)))
> (def y (select x 1))
> (def z (copy-list y))
> (setf (select y 0) -5)
> x
(1 (-5 3) (4 5))
> y
(-5 3)
> z
(2 3)
```

The function `select` works only with indices but we often want to extract elements satisfying a logical condition. The function `which` turns a list of logical elements into a list of indices that can then be used by select to extract or modify the required elements.

3.2 Defining Functions

Functions are defined using the macro `defun`. This takes as arguments a symbol which is to be the name of the function, a list of the formal parameters of the function, an optional documentation string and the main body of the function. The body of a function can be one or more `Lisp-Stat` expressions. The parameter list should initially list the names of the required parameters. Optional parameters can be specified in two ways. The preferred method is as keyword arguments. Keyword arguments are all those following `&key` in the argument list. To give them default values the arguments should be in the form of a list with two elements the first of which is the name and the second the default value if the argument is not supplied. Keyword arguments are specified in the call to a function by preceding the name with a colon. Another option is optional arguments. These are less flexible, however, since they cannot be referred to by name. This means that optional arguments must be supplied in the order given in the function definition. Optional arguments follow `&optional` in the formal argument list. Finally we can use `&rest` followed by a single symbol name. When the function is called any arguments which are not keyword arguments or already bound to the required or optional arguments will be put into a single list and this will be bound to the symbol. One, two or all three of these argument types can appear in a function definition. If more than one of them appears they must be in the order `&optional`, `&rest` `&key`.

3.3 Conditional Evaluation

There are three macros which can be used for conditional evaluation. The first is the simple `if` expression.

`(if` $<\textit{test}>$ $<\textit{consequent}>$ $<\textit{alternative}>$`)`

where $<\textit{test}>$ is an expression which returns `t` or `nil`. If $<\textit{test}>$ returns `t` then $<\textit{consequent}>$ is evaluated and its value returned otherwise $<\textit{alternative}>$ is evaluated and its value returned. Both $<\textit{consequent}>$ and $<\textit{alternative}>$ are any `Lisp-Stat` expressions. In particular one or both of them could be an `if` expression. Nested `if` expressions are often useful but they make code complex and difficult to read. An alternative is often to use the `cond` macro. The general form of a `cond` expression is

`(cond (`$<\textit{test}\ 1>$ $<\textit{expr}\ 1>$`)`
`      (`$<\textit{test}\ 2>$ $<\textit{expr}\ 2>$`)`
`      .......`
`      (`$<\textit{test}\ n>$ $<\textit{expr}\ n>$`))`

Each of the tests is evaluated in turn until one of them returns `t` and then the expression corresponding to that test is evaluated and its result returned. A simplification is possible when each of the tests is examining a single value of the same discrete variable In this case we can use the simpler construct of a `case` macro. The following two code fragments produce identical results.

```
(cond ((= (first x) 0) (sin (second x)))
      ((= (first x) 1) (cos (second x)))
      (t (tan (second x))))

(case (first x)
      (0 (sin (second x)))
```

```
    (1 (cos (second x)))
    (t (tan (second x))))
```

The `cond` macro however allows much more flexibility than `case`. Note that the final test in both of these macros can be `t`, this means that the final expression is always evaluated if none of the earlier tests evaluated to `t`.

3.4 Looping

Lisp has a very general looping structure using the special form `do`. This takes two arguments. The first is a list of triples each of which contain a name for a state variable, its initial value and an update to be evaluated at each iteration through the loop. The second argument is a list of two expressions; the first is a logical expression which is evaluated after each iteration and the second is the resulting expression which is evaluated and returned when the loop terminates.

```
(do ((<name 1>  <initial 1>  <update 1>)
      .......
     (<name n>  <initial n>  <update n>))
   (<test>  <result>))
```

Note that the bindings in a `do` loop are done in parallel, if we require that they are done in sequence then there is an alternative form `do*` which has identical syntax. The `do` loop actually constructs a new environment and all of the state variables are local to that environment so they do not affect any outside variables with the same name. We shall see other ways of defining local variables in the next section.

3.5 Local Variables and Functions

The macros `def` and `defun` create global bindings. That is the variables or functions are defined until they are explicitly undefined or the session is terminated. It is often useful however to have variables or functions which are available only within a piece of code. These are called local bindings. The parameters of a function are one example of *local variables which are bound to the values used in calling the function. We can create other local variables using the `let` macro.

```
(let ((<name 1>  <expr 1>)
      .......
     (<name n>  <expr n>))
  <body>)
```

All of the bindings are done in parallel and then the expressions in *<body>* are evaluated using these bindings. Any symbol used which is not one of the local bindings is bound to its value in the environment containing the `let` statement. Again there is a second form `let*` which causes the bindings to be done sequentially and so a `let*` binding is visible to subsequent bindings in the same `let*` expression.

It is often useful to change the values of local variables within the body of the local environment. `def` cannot be used for this as all of its bindings are global. The macro `setf` however can be used since it uses the bindings in the current environment. Of course we can change individual elements of a local variable using `setf` and `select` as described earlier.

Local functions are created in much the same fashion as local variables. The special form `flet` is used. It is very similar to `let` except that the bindings are now triples of a name, a parameter list and an expression or sequence of expressions making up the body of the local function. The local function definitions are again done in parallel so one local function cannot refer to itself or another local function. A form `labels` exists which allows the local functions to refer to previously defined local functions and also to themselves. It has the same structure as `flet` in all other respects.

Note that there is no environment in which both local variables and functions can be defined together. This can be accomplished by nesting one local environment within another such as making the body of a `flet` be a `let` expression.

3.6 Functions as Data

An advantage of Lisp for statistical and mathematical programming is the fact that function definitions can be passed as data. It is also not necessary that such function definitions have names since Lisp implements the ideas of *lambda calculus* which logicians have used for many years. Essentially lambda calculus is just a shorthand way of defining functions without giving them names. In Lisp the special form `lambda` is used to define such functions. These are called lambda expressions or anonymous functions. One aspect in which they are especially useful is in passing functions as parameters to other functions or as returning functions as the value of a function. We already saw an example of this in the section on non-linear regression.

```
(defun michaelis (x)
     #'(lambda (theta) (/ (* (first theta) x) (+ (second theta) x))))
(defun mich1 (x theta)
     (funcall (michaelis x) theta))
```

There are a number of points to note here. The form `#'func` is shorthand for `(function func)` and returns a function closure which is a combination of a function definition and the environment in which it was defined. This second part is necessary so that any free variables in the function can be correctly evaluated. In our case the function closure would include the binding of the parameter `x` to its supplied value. Secondly note that `(michaelis x)` has the required function as its *value* but not as its *function definition*. In fact the function closure is stored in the same way as the value of a variable as far as the interpreter is concerned. The function `funcall` evaluates its first argument and then applies it as a function to the rest of the arguments. `funcall` is also necessary when a function is passed as an argument to another function.

3.7 An Example

In this section we shall look at an example using many of the ideas presented so far in this chapter. Suppose we wish to be able to approximate a double integral using Simpson's method. First let us look at the one-dimensional Simpson's rule for integration of a function over a small interval

$$\int_a^{a+2h} f(x)dx \approx \frac{h}{3}\left[f(a) + 4f(a+h) + f(a+2h)\right]. \tag{3.1}$$

To apply this to larger intervals we divide the interval into n smaller intervals and apply (3.1) to each interval and sum the results. A function to do this could be defined as

```
(defun simpsons (f lower upper &optional (n 100) (epsilon 0.01))
  (let ((h (/ (- upper lower) (* 2 n))))
    (flet ((simp (a)
              (let ((f0 (funcall f a))
                    (f1 (funcall f (+ a h)))
                    (f2 (funcall f (+ a h h))))
                (* (/ h 3) (+ f0 (* 4 f1) f2)))))
      (do ((a lower (+ a h h))
           (int 0 (+ int (simp a))))
          ((>= a (- upper (* epsilon h))) int)))))
```

This requires a total of $3n$ function evaluations although it can be reduced to $2n+1$. In this function we defined h as a local variable and then within the body of that `let` expression we defined a local function which implements (3.1) for one interval. Within the local function, `simp`, `h` is a free variable so it is bound to its value in the environment in which `simp` was defined which in this case is the environment created by the `let` expression in whose body the `flet` expression occurs. We use a `do` expression to loop over the n intervals. We also use a tolerance of εh on the test for termination of the `do` expression. In general some allowance must be made for rounding error whenever tests are performed on real numbers. Since the tolerance is not a fixed constant but depends on the width of the intervals there is no problem for very small intervals.

One way to extend this idea to two-dimensional integration is to consider the inner integral as a function and so apply the method used above to get the outer integral. This means that we must evaluate the inner integral at a number of points and we do this by applying the one-dimensional Simpson's method again. We will take the basic structure which was used in the definition of `simpsons` and modify it for the two-dimensional case. The major part which needs modification are the three function evaluations in the local function `simp`. Instead of being simple function evaluations these now become calls to `simpsons`. One additional problem which must be taken into account is that the limits of integration for the inner integral can be functions of the variable in the outer integral or they can be numbers.

```
(defun simpsons2D (f lower.x upper.x lower.y upper.y
                   &optional (n 100) (epsilon 0.01))
   (let ((h (/ (- upper.x lower.x) (* 2 n))))
      (flet ((simp (a)
                (let ((f0 (simpsons #'(lambda (y) (funcall f a y))
                                    (if (numberp lower.y) lower.y
                                        (funcall lower.y a))
                                    (if (numberp upper.y) upper.y
                                        (funcall upper.y a))
                                    n epsilon))
                      (f1 (simpsons #'(lambda (y) (funcall f (+ a h) y))
                                    (if (numberp lower.y) lower.y
                                        (funcall lower.y (+ a h)))
                                    (if (numberp upper.y) upper.y
                                        (funcall upper.y (+ a h)))
                                    n epsilon))
                      (f2 (simpsons #'(lambda (y) (funcall f (+ a h h) y))
                                    (if (numberp lower.y) lower.y
                                        (funcall lower.y (+ a h h)))
                                    (if (numberp upper.y) upper.y
                                        (funcall upper.y (+ a h h)))
                             n epsilon)))
                   (* (/ h 3) (+ f0 (* 4 f1) f2)))))
        (do ((a lower.x (+ a h h))
             (int 0 (+ int (simp a))))
          ((>= a (- upper.x (* epsilon h))) int)))))
```

The function numberp returns t if its argument is a simple number and returns nil otherwise. This allows us to distinguish between numerical limits and limits for the inner integral which are functions. Note that we use lambda expressions to return the function as a function of y for fixed x and then these are passed to the one-dimensional Simpson's program defined earlier. We can now apply this to get approximations to the following integrals

$$\int_1^4 \int_{\sqrt{x}}^2 (x^2+y^2)dydx = \frac{1006}{105} \approx 9.5810 \int_{-1}^1 \int_{-\sqrt{1-x^2}}^{\sqrt{1-x^2}} \exp(-x^2-y^2)dydx = \pi(1-\mathrm{e}^{-1}) \approx 1.9859$$

We will also use the function time to see how long these examples take.

```
> (time (simpsons2D #'(lambda (x y) (+ (* x x) (* y y)))
                    1 4
                    #'(lambda (x) (sqrt x)) 2))
The evaluation took 5.66 seconds; 0.04 seconds in gc.
9.58095
> (time (simpsons2D #'(lambda (x y) (exp (- (+ (* x x) (* y y)))))
                    -1 1
                    #'(lambda (x) (if (>= x 1) 0 (- (sqrt (- 1 (* x x))))))
                    #'(lambda (x) (if (>= x 1) 0 (sqrt (- 1 (* x x)))))))
The evaluation took 6.43 seconds; 0.02 seconds in gc.
1.98567
```

The timings here are not too bad for an interpreted language. Remember that there are a total of 90,000 evaluations of the objective function used here. Of course the speed could be greatly increased if we did the code in a language such as C or Fortran. If we do not want to do this then there is an other option provided in Lisp-Stat which is to compile the functions into byte code which, while not as fast as native machine code, speeds up the execution of Xlispstat code by a factor of between 2 and 10 on average. The compile function is used to compile functions into byte code.

```
> (compile 'simpsons)
SIMPSONS
> (compile 'simpsons2D)
SIMPSONS2D
> (time (simpsons2D #'(lambda (x y) (+ (* x x) (* y y)))
                    1 4
                    #'(lambda (x) (sqrt x)) 2))
The evaluation took 2.94 seconds; 0.04 seconds in gc.
```

```
9.58095
> (time (simpsons2D #'(lambda (x y) (exp (- (+ (* x x) (* y y)))))
                    -1 1
                    #'(lambda (x) (if (>= x 1) 0 (- (sqrt (- 1 (* x x))))))
                    #'(lambda (x) (if (>= x 1) 0 (sqrt (- 1 (* x x)))))))
The evaluation took 3.75 seconds; 0.04 seconds in gc.
1.98567
```

3.8 Object-Oriented Programming

The type of programming which we have looked at thus far has been based on functions which return one or more numbers as results. Often, however, we would like a function to return a list with lots of different components and we would like the ability to perform various operations on this returned value. Another thing we would like is to be able to perform an action differently on lists representing different things. Although this can be achieved with regular functional programming it is very laborious. A better approach is object-oriented programming.

Let us consider the example of a linear regression model. We could write a function which does the least squares fits and returns a list of components such as the coefficient estimates, their covariance matrix, a list of residuals, the hat matrix etc. We could then write functions which take the relevant bits out of this list and do various things such as printing a summary or plotting residuals. This list quickly becomes out of hand and we would like to name the components in some way to ease the programming of new functions which work on lists like this. Now suppose we would like to define some generalized linear models. This would return a list with many of the same types of elements as the list for linear models, some of them would be treated in the same way as for linear models but some would be interpreted differently. It would be nice to be able to reuse the code already written for linear models and just add new functions for those bits that need changing. Object-oriented programming aims to make all of this possible in a relatively simple way.

Objects in `Lisp-Stat` are a special data structure with a number of named *slots* in which information is stored. Each object has a set of *messages* to which it can respond. The code that is executed by an object in response to a message is called a *method*. Objects are usually placed in a hierarchy with objects lower down the hierarchy inheriting methods and slots from its parent(s). In order to use object-oriented programming we need to be able to define new types of objects and new methods. In this section we shall see how this can be done in `Lisp-Stat`

The first concept which we will consider is that of a *prototype*. Basically this is a template which is used to form objects of a certain type. Objects which inherit from a prototype are called *instances* of that prototype. A new prototype is created using the macro `defproto` which has the form

`(defproto` <*name*> <*instance slots*> <*shared slots*> <*parents*> <*documentation*>`)`

<*name*> is just the name of the prototype, it is common practice for this name to end in `-proto`. <*instance slots*> is a list of the names of slots which are to be in each instance of this prototype. <*shared slots*> are slots which are only in the prototype. These are used to store information which is common to all instances of a prototype. <*parents*> is a list of prototypes from which the new prototype inherits. <*documentation*> is a string describing the prototype. Instances of a prototype are created by sending the prototype the `new` message.

We will usually define a number of methods for a prototype. These methods are defined using the `defmeth` macro. This is very similar to the function definition macro `defun` except that it has one extra argument.

`(defmeth` <*object*> <*name*> <*arguments*> <*documentation*> <*body*>`)`

The new argument is <*object*> which says which object this method is to be associated with. It will also be associated with any objects inheriting from this one unless redefined for an object lower down in the hierarchy. The name of a message must always start with a colon (:). The body of a method has exactly the same form as a functions body and the same scoping rules apply.

3.9 Data-set Objects Example

Let us consider an example for describing data sets. We will first define a very general prototype for all data sets and then create new prototypes which inherit from this one as we add specific data types into our system.

```
(defproto data-set-proto '(data title documentation) () ())
```

We did not specify any parents for this prototype so it is set to inherit from the root object `*object*`. This means that certain methods are already available. These enable us to add or delete slots or methods and to return the information contained in a slot as well as other basic functions which must be possible for all objects regardless of what they represent. The prototype is not very useful at present as it does not have any methods associated with it specifically for its role as a representation of a data set. Let us define some methods which print out the data and the documentation and produce boxplots of the data.

```
(defmeth data-set-proto :display ()
  (if (slot-value 'title) (princ (slot-value 'title)))
  (print x)
  (terpri))


(defmeth data-set-proto :datadoc ()
```

```
  (princ (slot-value 'documentation))
  (terpri))

(defmeth data-set-proto :boxplot () (boxplot (slot-value 'data)))
```

We could now define instances of this prototype and be able to print out the data set and the relevant documentation. For more elaborate functions on data sets however we need to know the type of a data set. What we do to a one-dimensional data set is different from what we do to a two-sample data set for example. Let us now define new prototypes which will do nice things for these cases.

```
(defproto one-sample-proto () () data-set-proto)
(defproto two-sample-proto () () data-set-proto)
```

We did not define any new slots for these particular data types but since they inherit from `data-set-proto` they will automatically have the slots defined for that prototype. There does not seem any need to change the `:datadoc` method and the but let us change the `:display` method for these objects. We shall also create summary methods which produce meaningful summaries of the data. The boxplot method does not need to be changed as it will automatically produce a single boxplot or two parallel boxplots in the two cases we are looking at.

```
(defmeth one-sample-proto :display ()
  (call-next-method)
  (format t "The sample size is ~d~%" (length (slot-value 'data))))

(defmeth two-sample-proto :display ()
  (if (slot-value 'title) (princ (slot-value 'title)))
  (let ((data (slot-value 'data)))
    (print (first data))
    (terpri)
    (format t "The first sample size is ~d~%" (length (first data)))
    (print (second data))
    (terpri)
    (format t "The second sample size is ~d~%" (length (second data)))))

(defmeth one-sample-proto :summary ()
  (format t "Summary for ~a~%"
        (if (slot-value 'title) (slot-value 'title) "a univariate sample"))
```

```
  (format t "The sample size is ~d~%" (length (slot-value 'data)))
  (format t "The mean is ~g~%" (mean (slot-value 'data)))
  (format t "The standard deviation is ~g~%"
        (standard-deviation (slot-value 'data))))


(defmeth two-sample-proto :summary ()
  (format t "Summary for ~a~%"
        (if (slot-value 'title) (slot-value 'title) "a two-sample dataset"))
  (format t "The sample sizes are ~d and ~d~%"
        (length (first (slot-value 'data)))
        (length (second (slot-value 'data))))
  (format t "The means are ~g and ~g~%"
        (mean (first (slot-value 'data)))
        (mean (second (slot-value 'data))))
  (format t "The standard deviations are ~g and ~g~%"
        (standard-deviation (first (slot-value 'data)))
        (standard-deviation (second (slot-value 'data)))))
```

The functions `print`, `princ` `terpri` and `format` are various ways that `Lisp` has for printing. See Section 4.1 of Tierney (1990) for more information on this aspect. The function `call-next-method` is often very useful when we just want to add something to a method which has already been defined for the parents of an object or prototype.

Now we have nice prototypes but we need to know how to create instances from these prototypes. This is done by sending the message `:new` to a prototype which executes the `:isnew` method. The basic `:isnew` allows you to initialize slots using keyword arguments but it does not force any slot to be initialised. In our case we would like to force the dataset slot to be filled and we would also like to ensure that the data is of the correct type.

```
(defun one-sample-p (data)
  (and data
       (listp data)
       (not (reduce #'(lambda (x y) (or x y)) (mapcar #'listp data)))))

(defun two-sample-p (data)
  (and (= (length data) 2)
       (one-sample-p (first data))
       (one-sample-p (second data))))
```

```
(defmeth one-sample-proto :isnew (data &key title doc)
  (if (one-sample-p data)
      (call-next-method :data data :title title :documentation doc)
      (print "Data is not a univariate sample.  Object not created")))

(defmeth two-sample-proto :isnew (data &key title doc)
  (if (two-sample-p data)
      (call-next-method :data data :title title :documentation doc)
      (print "Data is not a two-sample data set.  Object not created")))
```

We can use the default `:isnew` method once we have tested that data is of the correct type for our objects. `mapcar` applies a function to each element of a list in turn and returns a list of the results. `reduce` combines all elements of a list using a binary function between successive pairs. Note that we had to define an anonymous function to use `or` because `or` is a macro and `reduce`, `mapcar` and related functions work only with functions.

Finally we can now construct new instances of our two data types.

```
> (def shot-put1 (send one-sample-proto :new shot-put
                        :title "Distances of Shot-Put Throws"
                        :doc
  "   The data set records the distances of throws in the men's shot-put
   at the 1991 English athletics championships."))
> (def gasdata (send two-sample-proto :new (list gas1 gas2)))
> (send shot-put1 :summary)
Summary for Distances of Shot-Put Throws
The sample size is 20
The mean is 14.93
The standard deviation is 1.431
NIL
> (send gasdata :summary)
Summary for a two-sample dataset
The sample sizes are 50 and 50
The means are 14.63 and 25.16
The standard deviations are 1.829 and 2.191
NIL
```

If we want other abilities such as the making histograms, or conducting tests on our data sets we just need to define the appropriate methods. New data set types can be easily added by defining a new prototype which inherits from `data-set-proto` and adding any new slots or methods which are required for this new type.

Instead of sending the `new` message to a prototype, it is normal to define a constructor function which often just provide a wrapper in which the `new` message is sent to the appropriate prototype but they can also do much more. An example of this is the `regression-model` function which sends a `new` message to the prototype `regression-model-proto` fits the linear regression model and stores the results of this fit in the appropriate slots within the newly created object and then returns the object.

Chapter 4

Windows and Menus

The graphical interface in `Lisp-Stat` is implemented using the ideas of object oriented programming described in the previous chapter. In this chapter we shall look at this aspect more closely so that we can create graphics which are specialized to our own needs. There are two basic types of graphical interfaces and we shall look at these in turn.

4.1 Windows

All windows are objects which inherit from the prototype `window-proto`. This prototype contains slots detailing the title of a window, its size, its location on the screen as well as other information which is necessary for the creation of a window regardless of what is actually in the window. There are also methods defined for this prototype which allow us to change aspects of the window itself. Most actions however will depend on the contents of the window. We shall consider two prototypes which inherit from `window-proto`; `modal-dialog-proto` and `dialog-proto` which implement dialogs and `graph-proto` which implements data graphs.

4.1.1 Modal Dialogs

Modal dialogs are windows which appear on the screen to give the user some information and/or ask for user input. The `Lisp-Stat` interpreter will not continue until a modal dialog has been dismissed by clicking on the appropriate button. We will first look at some standard dialogs for which constructor functions are already available.

The simplest type of modal dialog is a window which displays a message to the user and has a single button labelled `OK` which dismisses the dialog when clicked. This type of dialog is created using the `message-dialog` function which takes a string as argument. It returns the value `nil`. A slightly more complicated dialog constructed using `ok-or-cancel-dialog` has a second button labelled `CANCEL`. Both buttons dismiss the dialog but the returned value is `t` if `OK` is selected and `nil` otherwise. This type of dialog is useful for non-fatal errors when the user can be told of the error and asked if they wish to continue the calculation or not.

There are two dialogs which allow the user to select one or more items from a list. These are constructed using `choose-item-dialog` and `choose-subset-dialog`. The difference is that the former allows the user to pick exactly one of the items in the list whereas the latter allow the user to pick any subset of the items in the list. Both take the same arguments that is a string for a message and a list of strings corresponding to the items in the list. A window is created with the supplied message and a radio button or check box for each element in the list labelled with the corresponding string. There are also `OK` and `CANCEL` buttons. The returned value is `nil` if `CANCEL` is clicked and an integer for `choose-item-dialog` or a list of integers for `choose-subset-dialog`. These integers correspond to the indices of the chosen items in the list of strings supplied.

Finally there are two dialogs which allow the user to type in the information required. They differ only in how the resulting value is interpreted. The constructor function `get-string-dialog` returns a string containing the user input and `get-value-dialog` returns the result of evaluating the input which it assumes is a Lisp expression. Again clicking the `CANCEL` button returns `nil` regardless of the input.

4.1.2 Modeless Dialogs

Modeless dialogs are dialogs which remain on the screen until dismissed but do not interupt input. The most useful type of modeless dialog is a `slider`. A slider contains a scroll bar which can be moved through a sequence of points. Each time the bar is moved the value at which it is stopped is passed to a function which is then executed. The two functions which construct sliders are `sequence-slider-dialog` and `interval-slider-dialog`. The former takes a sequence of points whereas the latter takes a range and a number of points. Usually a function will be specified using the keyword argument `action`. To see an example use `(load-example "bc-demo")` which shows a normal Q-Q plot of a data set and the slider allows us to change the power to be used in a Box-Cox transformation of this data. We could use this to graphically select a power transformation of the data which brings the data close to normality.

4.2 Creating Custom Dialogs

Although the dialogs described above are often sufficient we may want to create a custom dialog for a particular purpose. To do this we need to directly use the prototypes. The two prototypes which construct dialogs are `modal-dialog-proto` and `dialog-proto`. To create an instance of a dialog we should send one of these the `:new` message. There is one required argument which is a list of items describing how the dialog is laid out. All top level items in this list are displayed in a column. Lists within the item list represent rows, further nesting corresponds to sub-columns and sub-rows alternatively. A modal dialog initially acts in the same way as a modeless dialog. The difference is that a modal dialog can be sent the message `:modal-dialog` which causes all input to be sent to the dialog until it receives the message `:modal-dialog-return`.

The items which are contained in the item list are all objects which inherit from the prototype `dialog-item-proto`. In fact this prototype is never used directly. Inheriting from it are other prototypes which define specific items that one may find in a dialog. Some of these are

- `text-item-proto` used to display message strings,
- `button-item-proto` for clickable buttons,
- `modal-button-proto` is simplified for buttons which dismiss a modal dialog,
- `choice-item-proto` creates a column of radio buttons only one of which can be selected at a time,
- `toggle-item-proto` creates a single check box,
- `edit-text-item-proto` creates an editable text field which always returns a string. The string can be turned into an expression using the function `read-from-string` and the result can be passed to `eval` to create something like `get-value-dialog`.
- `sequence-scroll-item-proto` creates scroll bars which scroll through a sequence of numbers.
- `interval-scroll-item-proto` creates scroll bars which scroll through an interval.

The arguments which are necessary in creating instances of these prototypes vary. Essentially everything that is needed to describe the item and any action which it can cause needs to be specified. In order to create a dialog it is necessary to create instances of any items we want and then incorporate these into a list which is passed as argument in the `:new` message which is sent to the appropriate dialog prototype. This can be quite a lot of work. Usually a constructor function is defined for a custom dialog which is likely to be used more than once. The code for such functions is not difficult but for a complex dialog it can be error-prone.

4.3 Menus

Another way of asking for user input is through a menu. Menus are objects which inherit from `menu-proto`. The `:isnew` method for this prototype requires one argument, a string giving the name of the menu. Initially the menu is empty. We need to construct each of the items in the menu individually as objects inheriting from `menu-item-proto`. Each item must be given a name and usually we will also specify an action to be taken when the item is selected. This is done by including a function which takes no arguments as the `action` keyword argument to the `:isnew` method. Once the structure of the menu has been defined it should be sent two messages, `:install` makes the menu appear in a menu bar and `:enabled` with the argument `t` makes it ready to use. Each of the individual items must also be sent the message `:enabled` with the argument `t`. This allows us to have some items enabled and others disabled if we wish. There are no predefined constructor functions for menus as no two menus will be the same.

4.4 Data sets Example

Let us now return to the data set objects which we defined in Section 3.9. Instead of needing to remember the messages which can be sent to this type of object we may wish to install a menu from which we can choose which message should be sent. An easy way to do this would be to create a individual menu for each data set but this is not very satisfactory. We will choose an alternative which causes each menu item to create a modal dialog to get the name of the data set to which we wish to send a message and then send the correct message to the named object. First we define the menu

```
(def data-menu (send menu-proto :new "Data Menu"))
```

Next we define each of the items to appear in the menu. Each one is very similar since the only differences are the name of the item and the message which is sent to the object. We use `get-string-dialog` to get a string corresponding to the name of the data-set. The construct `(eval (read-from-string name))` returns the object whose name is represented by `name` and we can then send messages to this. Since the process of getting the object is identical for all items we define a function which returns the appropriate object for us and use this in each of the items.

```
(defun get-object ()
   (eval (read-from-string (get-string-dialog "Name of Data set"))))

(def doc-item (send menu-item-proto :new "Documentation"
                    :action #'(lambda ()
                                  (send (get-object) :datadoc)
                                  (top-level nil))))


(def summ-item (send menu-item-proto :new "Summary"
```

```
                     :action #'(lambda ()
                                  (send (get-object) :summary)
                                  (top-level nil))))

(def display-item (send menu-item-proto :new "Display"
                  :action #'(lambda ()
                               (send (get-object) :display)
                               (top-level nil))))

(def boxplot-item (send menu-item-proto :new "Boxplots"
                        :action #'(lambda ()
                                     (send (get-object) :boxplots))))
```

Now our menu is constructed so we send the following messages to install and enable it.

```
(send data-menu :append-items display-item summ-item boxplot-item doc-item)
(send data-menu :enabled t)
(send data-menu :install)
(send display-item :enabled t)
(send summ-item :enabled t)
(send boxplot-item :enabled t)
(send doc-item :enabled t)
```

The menu name `Data Menu` should now appear on the menu bar (the exact implementation of this is system dependent) and is ready for use. We can remove the menu from the menu bar using the `:remove` message. Note that the definition of `get-object` above will crash if the `CANCEL` button on the dialog is pressed. It should be made more secure by including some code to say what happens if the dialog returns `nil`.

4.5 Graphics Windows

The third type of graphical interface is a graphics window which is a display such as a plot. There are two levels at which we may wish to have the ability to work on a graphics window. The lower-level entails the ability to consider individual pixels or blocks or pixels in a graphics window. All graphics are implemented by changing the colour of individual pixels so this level allows us very great flexibility in the graphics we produce. The payment for this is that we need to program at a low level and so code can quickly become complex. We shall not consider this level but it is described in Chapter 8 of Tierney (1990) for those who are interested.

Statistical graphics are different to basic graphics in one major respect; a statistical graphic operates on data. The fact that a data set can be large and we usually want to draw the same thing for each data point at slightly different locations makes low level graphical programming very repetitive and error-prone. A higher level has been introduced in `Lisp-Stat` which builds on the basic graphical windows to allow statistical graphics to be produced relatively easily. We have already seen many of the constructor functions which are available and these will often be sufficient. Sometimes, however, we may want to produce special graphs for particular types of data. As with dialogs this entails us working at the level of the objects which underpin statistical graphics.

The basic prototype for statistical graphics is `graph-proto`. There is one required argument to the `:isnew` method for this prototype and that is the dimension of the space (number of variables) required. This remains fixed once the graph has been created and cannot be changed. There are a number of prototypes which inherit from `graph-proto` which implement different types of plots. For example `histogram-proto` has methods which do the correct things for a histogram. Each time we use a constructor function (such as `histogram`) the returned value is an instance of the associated prototype. Often it is useful to create a standard graphic but add or change methods. This would be done by using `defmeth` for the instance. If, however, we wish to create a new type of graphic we should create an instance of `graph-proto` and set up its methods. If the graphic is likely to be used regularly then we may want to define a new prototype, inheriting from `graph-proto`, and an associated constructor function.

Once an instance of `graph-proto` has been created we need to supply it with data. This is done using the `:addpoints` method. The method `:variable-label` can be used to give the variables names. A graph always starts by looking at a two-dimensional plot on the unit interval. This will usually mean that we need to send the plot the `:adjust-to-data` message which adjusts the range so that one should now be able to see a scatter-plot of the first two variables. For concreteness let us consider the stack loss data.

```
(def plot1 (send graph-proto :new 4))
(send plot1 :add-points (list air temp conc loss))
(send plot1 :variable-label (iseq 4)
      '("Air Pressure" "Water Temp." "Acid Conc." "Ammonia Loss"))
(send plot1 :adjust-to-data)
```

The plots now appear on the screen but it is still not very informative as there are no axes. These can be added using the messages `:x-axis` and `:y-axis`

```
(send plot1 :x-axis t t 5)
(send plot1 :y-axis t t 5)
```

The three optional arguments indicate whether the axis is shown, whether it is labelled and the number of tick marks on the axis. The default range for the axes is the exact range of the data. This can be changed with the `:range` method which takes three arguments; the index of the variable and the upper and lower end-points of the range. The function `get-nice-range` can sometimes be useful to determine a nice combination of range and tick-marks. Each time an `:adjust-to-data` message is sent the ranges are reset to the data ranges. We can write a new `:adjust-to-data` method which sets the ranges of the current variables to allow 5% at either end and chooses a nice range.

```
(defmeth plot1 :adjust-to-data (&key (draw t))
  (call-next-method :draw nil)
  (let* ((vars (send self :current-variables))
         (xpts (send self :point-coordinate (first vars)
                           (iseq (send self :num-points))))
         (xmin (min xpts))
         (xmax (max xpts))
         (skip (* 0.05 (- xmax xmin)))
         (xaxis (send self :x-axis))
         (newx (get-nice-range (- xmin skip) (+ xmax skip) (third xaxis)))
         (ypts (send self :point-coordinate (second vars)
                          (iseq (send self :num-points))))
         (ymin (min ypts))
         (ymax (max ypts))
         (yaxis (send self :y-axis))
         (newy (get-nice-range (- ymin skip) (+ ymax skip) (third yaxis))))
    (send self :range (first vars) (first newx) (second newx))
    (send self :x-axis (first xaxis) (second xaxis) (third newx))
    (send self :range (second vars) (first newy) (second newy))
    (send self :y-axis (first yaxis) (second yaxis) (third newy))))
```

Note the use of the symbol `self` in this method. This can only be used in methods and refers to the object to which the message was sent which caused this method to be executed. The `:current-variables` method returns the indices of the two variables currently in view. It can be given optional arguments to change the current variables.

There are other methods which act on the individual data points. We can change the label used for a point, change the plotting symbol used or the colour used to plot it. We can also add lines to the plot using the `:add-lines` method which takes a list of lists corresponding to the start points of consecutive line segments. As with points we can change the colour of these line segments individually and also the width of the segments.

Centring and fixed or variable scaling methods are available to aid in the application of transformations of the data. These are especially useful for multi-dimensional plots for which we would like to have the ability to rotate the points. Another useful aspect for rotatable plots are the symbols `dot1`, `dot2`, `dot3` and `dot4` which allow us to use different sizes of dots for closer and further away points. When transformations are used there are in effect two different coordinate systems which are affine transformations of each other. The real coordinate of the point is fixed being defined by the input variables. The transformed coordinate on the other hand changes depending on which view is currently shown in the window. `point-coordinate` and `point-transformed-coordinate` retrieve the coordinates of a point.

The details of how a plot react to the mouse can also be customized. The easiest way to do this is to add a new *mouse mode*. By default there are two mouse modes. `selecting mode` and `brushing mode` which can be changed using a dialog produced by the `Mouse Mode` menu item. We can add a new mouse mode using the `:add-mouse-mode` method. We should give this a symbol which will be used for the mouse mode. It can also be supplied by keywords which give the mode a title for use in the selection dialog, a method which should be executed when the mouse is clicked, a method for when the mouse is dragged and a cursor symbol for when the mouse mode is active. The writing of the methods requires that we interact directly with the canvas and this can only be done using the lower-level tools.

Each plot comes with a menu which has a number of items installed by default. The actual implementation is that a menu object is stored in one of the slots of the graph. We can access the menu by sending the `:menu` message to the plot. New items can be added using the basic tools described in the previous section. Items can inherit from `menu-item-proto` as before but it is more common to make them inherit from `graph-item-proto`. The `:isnew` method for this takes three arguments. The first is the item name, the second is the symbol for the plot object to which the menu is attached and the third is a message which is sent to the plot when the menu item is selected. The item is put into the menu by sending the `append-items` message to the menu returned by sending the `:menu` message to the plot. The `:new-menu` message can be sent to the plot to alter the title of the basic menu and which items are included by default.

Bibliography

Bates, D. M. and Watts, D. G. (1988) *Nonlinear Regression Analysis and its Applications.* New York: Wiley.

Box, G. E. P., Hunter, W. G. and Hunter, J. S. (1978) *Statistics for Experimenters.* New York: Wiley.

Breslow, N. (1985) Cohort analysis in epidemiology. In *A Celebration of Statistics*, eds A. C. Atkinson and S. E. Fienberg, pp. 109–143. New York: Springer.

Chaloner, K. and Brant, R. (1988) A Bayesian approach to outlier detection. *Biometrika* **75**, 651–660.

Dennis, J. E. and Schnadel, R. B. (1983) *Numerical Methods fof Unconstrained Optimization and Nonlinear Equations.* Englewood Cliffs, New Jersey: Prentice-Hall.

Doll, R. and Hill, A. B. (1966) Mortality of British doctors in relation to smoking: Observations on coronary thrombosis. In *Epidemiological Study of Cancer and Other Chronic Diseases*, ed. W. Haenszel, volume National Cancer Institute Monograph 19, pp. 205–268. Washington: Government Printing Office.

Freeman, D. H. (1987) *Applied Categorical Data Analysis.* New York: Marcel Dekker.

Press, W. H., B. P. Flannery, S. A. T. and Vetterling, W. T. (1992) *Numerical Recipes in C: The Art of Scientific Computing.* Second edition. Cambridge: Cambridge University Press.

Tierney, L. (1990) *Lisp-Stat: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics.* New York: John Wiley & Sons.

Tierney, L. (1991) Generalized linear models in Lisp-Stat. Technical Report 557, School of Statistics, University of Minnesota.

Tierney, L. and Kadane, J. B. (1986) Accurate approximations for posterior moments and marginal densities. *Journal of the American Statistical Association* **81**, 82–86.

Tierney, L., Kass, R. E. and Kadane, J. B. (1989) Approximate marginal distributions of nonlinear functions. *Biometrika* **76**, 425–433.