XLISP-STAT

A Statistical Environment Based on the XLISP Language

by
Luke Tierney

Technical Report No. 512
February 1988

School of Statistics
University of Minnesota

XLISP-STAT Version 1.0

Table of Contents

Preface

XLISP-STAT is a statistical environment built on top of the XLISP programming language. This document is intended to be a tutorial introduction to the basics of XLISP-STAT. The first three sections contain the information you will need to do elementary statistical calculations and plotting. The fourth section describes some additional features of the user interface that may be helpful. The remaining sections deal with more advanced topics, such as interactive plots, regression models, data editing and writing your own functions. All sections are organized around examples, and most contain some suggested exercises for the reader.

This document is not intended to be a complete manual. However, a list of many of the commands available is given in the appendix. Brief help messages on these commands are available through the interactive help facility described in Section 4 below.

XLISP itself is a full fledged programming language developed by David Betz and placed in the public domain. It is a dialect of Lisp, most closely related to the Common Lisp dialect. XLISP also contains some extensions to Lisp to support object oriented programming. These facilities are used in XLISP-STAT to implement the screen menus, plots and regression models. Several excellent books on Common Lisp are available. One example is Winston and Horn (1984). A book on XLISP itself has recently been published, but I have not had a chance to review it. Documentation on XLISP is included in the source code distribution; a copy is available from me. Betz has also written a tutorial introduction to XLISP for BYTE, Betz (1985). However, this deals with version 1.2 which differs somewhat from version 1.6, the basis of XLISP-STAT.

This document deals primarily with Version 1.0 of XLISP-STAT for the Apple Macintosh. XLISP-STAT is also available for 4.[23]BSD UNIX and for running under suntools on Sun 3 computers. The basic UNIX version supports non-interactive plotting for tektronix terminals via the gnuplot routines. When running on a Sun 3 console under suntools interactive graphics are also available.

The Macintosh version of XLISP-STAT was developed and compiled using the Lightspeed™ C compiler from Think Technologies, Inc. The Macintosh user interface is based on Paul DuBois' TransSkel and TransEdit libraries. Most floating point calculations in the Macintosh version are done with the Microsoft Fortran runtime library. Some of the linear algebra calculations are done with Fortran code from the LINPACK library. Regression computations are carried out using the sweep algorithm as described in Weisberg (1982).

This tutorial has borrowed several ideas from Gary Oehlert's "MacAnova User's Guide". Many of the on-line help entries have been adopted directly or with minor modifications from the Kyoto Common Lisp System. Most of the examples used in this tutorial have been taken from Devore and Peck (1986). Many of the functions added to XLISP-STAT were motivated by similar functions in the S statistical environment described in Becker and Chambers (1984).

The present version of XLISP-STAT seems to run fairly comfortably on a Mac Plus, but is a bit cramped on a 512K Mac. The program will occasionally bomb with an ID=28 if it gets into a recursion that is too deep for the Macintosh stack to handle. On a 512K Mac it may also bomb with and ID=15 if too much memory has been used for the segment loader to be able to bring in a required code segment.

Development of XLISP-STAT was supported in part by the grant of an Apple Macintosh computer and hard disk from the MinneMac Project at the University of Minnesota, by a single

quarter leave granted to the author by the University of Minnesota, and by grant DMS-8705646 from the National Science Foundation.

Using this Tutorial

The best way to learn about a new computer program is usually to use it. You will get most out of this tutorial if you read it at your computer and work through the examples yourself. To make this easier the named data sets used in this tutorial have been stored on the file `tutorial.lsp` in the **Data** folder of the Macintosh distribution disk. To read in this file select the **Load** item on the file menu. This will bring up an Open File dialog window. Use this dialog to open the **Data** folder on the distribution disk. Now select the file `tutorial.lsp` and press the **Open** button. The file will be loaded and some variables will be defined for you.

Why XLISP-STAT Exists

There are three primary reasons behind my decision to produce the XLISP-STAT environment. The first is to provide a vehicle for experimenting with dynamic graphics and for using dynamic graphics in instruction. Second, I wanted to be able to experiment with an environment supporting functional data, such as mean functions in nonlinear regression models and prior and likelihood functions in Bayesian analyses. Finally, I was interested in exploring the use of object oriented programming ideas for building and analyzing statistical models. I will discuss each of these points in a little more detail in the following paragraphs.

The development of high resolution graphical computer displays has made it possible to consider the use of dynamic graphics for understanding higher-dimensional structure. One of the earliest examples is the real time rotation of a three dimensional point cloud on a screen - an effort to use motion to recover a third dimension from a two dimensional display. Other techniques that have been developed include "brushing" a scatterplot - highlighting points in one plot and seeing where the corresponding points fall in other plots. A considerable amount of research has been done in this area, see for example the discussion in Becker and Cleveland (1987). However most of the software developed to date has been developed on specialized hardware, such as the Tty 5620 terminal or Lisp machines. As a result, very few statisticians have had an opportunity to experiment with dynamic graphics first hand, and still fewer have had access to an environment that would allow them to implement dynamic graphics ideas of their own. To the best of my knowledge there are at this point only two widely available software packages that incorporate dynamic graphics. They are MacSpin™ and the Data Desk™, both written for the Apple Macintosh. Both are fine programs, but both have their drawbacks (especially for use in instructional labs at an impoverished state institution). MacSpin only provides point cloud rotation and is not part of an integrated environment. To view a pair of covariates along with the residuals from a regression, for example, would require a second statistical program to calculate the regression residuals. The Data Desk does provide an integrated framework, but it does not allow a user to add new functions. Furthermore, many users have found its heavy reliance on mouse and menu operations to be rather restrictive. XLISP-STAT provides at least a partial solution to these problems. It allows the user to modify a scatter plot with Lisp level functions and provides hooks for attaching a user defined function to a mouse click within a scatter plot window. Alternatively, it is easy to add functions written in C to the program. Ideally this should not be necessary; but on the Macintosh it is often the only way to achieve reasonable execution speed.

An integrated environment for statistical calculations and graphics is essential for developing an understanding of the uses of dynamic graphics in statistics and for developing new graphical techniques. Such an environment must essentially be a programming language. Its basic data types must include types that allow groups of numbers - data sets - to be manipulated as entire objects. But in model-based analyses numerical data are only part of the information being used. The remainder is the model itself. Sometimes a model is easily characterized by specifying a set of numbers. A normal linear regression model with i. i. d. errors might be described by the number of covariates, the coefficients and the error variance. On the other hand, in many cases it is easier

to specify a model by specifying a function. To specify a normal nonlinear regression model, for example, one might specify the mean function. If our language is to allow us to specify this function within the language itself then the language must support a functional data type with full rights: It has to be possible to define functions that manipulate functions, return functions, apply functions to arguments, etc.. The choice I faced was to define a language from scratch or to use an existing language. Because of the complexity of issues involved in functional programming I decided to use a dialect of a well understood functional language, Lisp. The syntax of Lisp is somewhat unfamiliar to most users of statistical packages, but it is easy to learn and several good tutorials are available in local book stores. I considered the possibility of using Lisp to write a top level interface with a more "natural" syntax, but I did not see any way of doing this without complicating access to some of the more powerful features of Lisp or running into some of the pitfalls of functional programming. I therefore decided to retain the basic Lisp top level syntax. To make the manipulation of numerical data sets easier I have redefined the arithmetic operators and basic numerical functions to work on lists and arrays of data.

Having decided to use Lisp as the basis for my environment XLISP was a natural choice for several reasons. It has been placed in the public domain by its author, David Betz. It is small (for a Lisp system), its source code is available in C, and it is easily extensible. Finally, it includes support for object oriented programming in the form of a tree structured class system and message passing facility. Object oriented programming has received considerable attention in recent years and is particularly natural for use in describing and manipulating graphical objects. It may also be useful for the analysis of statistical data and models. A collection of data and assumptions may be represented as an "object". The model object can then be examined and modified by sending it messages. Many different kinds of models will answer similar questions, thus fitting naturally into an inheritance structure. XLISP-STAT's implementation of linear and nonlinear regression models as "objects", with nonlinear regression inheriting many of its methods from linear regression, is a first, primitive attempt to exploit this programming technique in statistical analysis.

Availability

Source code for XLISP-STAT for Sun 3, 4.[23]BSD UNIX and Macintosh versions and executables for the Macintosh are available free of charge for non-commercial use. (You should, however, be prepared to bear the cost of copying, e. g. by supplying a disk or tape and a stamped mailing envelope).

Disclaimer

XLISP-STAT is an experimental program. It has not been extensively tested. The University of Minnesota, the School of Statistics, the author of the statistical extensions and the author of XLISP take no responsibility for losses or damages resulting directly or indirectly from the use of this program.

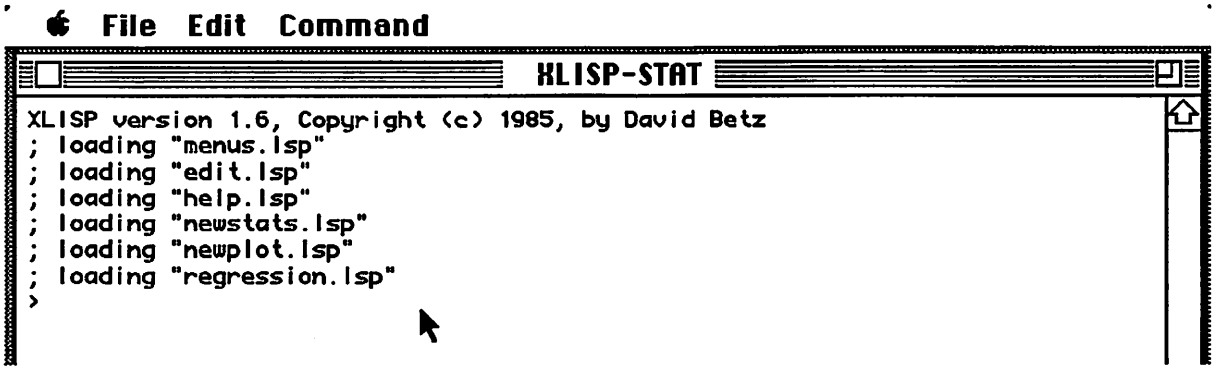
1. Starting and Finishing

You should have the program XLISP-STAT on a Macintosh disk. XLISP-STAT needs to have several files available for it to work properly. These files are:

init.lsp
menus.lsp
edit.lsp
help.lsp
newstats.lsp
newplot.lsp
regression.lsp
f77.rl
xlisp.help

Before starting XLISP-STAT you should make sure that these files are in the same folder as the XLISP-STAT application.

To start XLISP-STAT double click on its icon. The program will need a little time to start up and read in the files mentioned above. When XLISP-STAT is ready its command window will look something like this[1]:



The XLISP-STAT prompt is the final ">" in the window.

Any characters you type while the command window is active will be added to the line after the final prompt. When you hit a return, XLISP-STAT will try to interpret what you have typed and will print a response. For example, if you type a 1 and hit return then XLISP-STAT will respond by simply printing a 1 on the following line and then giving you a new prompt:

```
> 1
1
>
```

If you type an *expression* like `(+ 1 2)`, then XLISP-STAT will print the result of evaluating the expression and give you a new prompt:

```
> (+ 1 2)
```

[1]On a 512K Mac or in a small Switcher or MultiFinder partition a message warning about memory restrictions will be printed at this point.

```
3
>
```

As you have probably guessed, this expression means that the numbers 1 and 2 are to be added together. The next section will give more details on how XLISP-STAT expressions work. In this tutorial I will always show interactions with the program as I have done here: The ">" prompt will appear before lines you should type. XLISP-STAT will supply this prompt when it is ready; you should not type it yourself.

Now that you have seen how to start up XLISP-STAT it is a good idea to make sure you know how to get out. As with many Macintosh programs the easiest way to get out is to choose the **Quit** command from the **File** menu. You can also use the command key shortcut Command-Q, or you can type the expression

```
> (exit)
```

Any one of these methods should cause the program to exit and return you to the Finder.

2. Introduction to Basics

Before we can start to use XLISP-STAT for statistical work we need to learn a little about the kind of data XLISP-STAT uses and about how the XLISP-STAT evaluator (or interpreter) works.

2.1. Data

XLISP-STAT works with two kinds of data: simple data and compound data. Simple data, or atoms, are numbers

```
1          ; an integer
-3.14      ; a floating point number
```

logical values

```
T          ; true
nil        ; false
```

strings (always enclosed in double quotes)

```
"This is a string 1 2 3 4"
```

and symbols (used for naming things; see the following section)

```
x
x12
12x
this-is-a-symbol
```

Compound data are lists

```
(this is a list with 7 elements)
(+ 1 2 3)
(sqrt 2)
```

or vectors

```
#(this is a vector with 7 elements)
#(1 2 3)
```

Higher dimensional arrays are another form of compound data; they will be discussed below in Section 10.

All the examples given above can be typed directly into the command window as they are shown here. The next section describes what XLISP-STAT will do with these expressions.

2.2. The Evaluator

A session with XLISP-STAT basically consists of a conversation between you and the *evaluator*. The evaluator gives you a prompt, you type an expression, XLISP-STAT evaluates the expression, prints the result and gives you another prompt.[1]

[1]It is possible to make a finer distinction between the *top level*, which prints the prompt and then calls the *reader* to read your input, the *evaluator* to evaluate your input, ant the *printer* to print the results. For simplicity I will use *evaluator* to describe any one of these functions.

The basic rule to remember is that everything is evaluated. Numbers and strings evaluate to themselves:

```
> 1
1
> "Hello"
"Hello"
>
```

Lists are more complicated. Suppose you type the list `(+ 1 2 3)` at the evaluator. This list has four elements: the symbol + followed by the numbers 1, 2 and 3. Here is what happens:

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

A list is evaluated as a *function application.* The first element is a symbol representing a function (in this case the symbol + representing the addition function) and the remaining elements are the arguments. Thus the list in the example above is interpreted to mean "Apply the function + to the numbers 1, 2 and 3". Actually, the arguments to a function are always evaluated before the function is applied. In this example the arguments are all numbers and thus evaluate to themselves. On the other hand, consider

```
> (+ (* 2 3) 4)
10
>
```

The evaluator has to evaluate the first argument to the function + before it can apply the function.

Occasionally you may want to tell the evaluator <u>not</u> to evaluate something. For example, suppose we wanted to get the evaluator to simply print the list `(+ 1 2)` back to us, instead of evaluating it. To do this we need to *quote* our list:

```
> (quote (+ 1 2))
(+ 1 2)
>
```

`quote` is not a function. It does not obey the rules of function evaluation described above: it's argument is not evaluated. `quote` is called a *special form* - special because it has special rules for the treatment of its arguments. There are a few other special forms that we will need; I will introduce them as they are needed. Together with the basic evaluation rules described here these special forms make up the basics of the LISP language. The special form `quote` is used so often that a shorthand notation has been developed:

```
> '(+ 1 2)      ; single quote shorthand
```

a single quote before the expression you want to quote. This is equivalent to `(quote (+ 1 2))`. Note that there is no matching quote following the expression.

By the way, the semicolon "`;`" is the LISP comment character. Anything you type after a semicolon up to the next time you hit a return is ignored by the evaluator.

Exercises
For each of the following expressions try to predict what the evaluator will return. Then type them in, see what happens and try to explain any differences.

1) `(+ 3 5 6)`
2) `(+ (- 1 2) 3)`
3) `'(+ 3 5 6)`
4) `'( + (- 1 2) 3)`
5) `(+ (- (* 2 3) (/ 6 2)) 7)`

Remember, to quit from XLISP-STAT choose **Quit** from the **File** menu or type `(exit)`.

3. Elementary Statistical Operations

This section introduces some of the basic graphical and numerical statistical operations that are supported in XLISP-STAT.

3.1. First Steps

Statistical data usually consists of groups of numbers. Exercise 2.11 in Devore and Peck (1986) for example describes an experiment in which 22 consumers reported the number of times they had purchased a product during the previous 48 week period. The results are given as a table:

0	2	5	0	3	1	8	0	3	1	1
9	2	4	0	2	9	3	0	1	9	8

To examine this data in XLISP-STAT we represent it as a list of numbers using the `list` function:

```
> (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
(0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
>
```

Note that the numbers are separated by *white space* (spaces, tabs or even returns), not commas.

The `mean` function can be used to compute the average of a list of numbers. We can combine it with the list function to find the average number of purchases for our sample:

```
> (mean (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8))
3.227273
>
```

The median of these numbers can be computed as

```
> (median (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8))
2
>
```

It is of course a nuisance to have to type in the list of 22 numbers every time we want to compute a statistic for the sample. To avoid having to do this I will give this list a name using the `def` special form[1]:

```
> (def purchases (list 0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9
8))
(0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
>
```

[1] `def` acts like a special form, rather than a function, since its first argument is not evaluated (otherwise you would have to quote the symbol). Technically `def` is a macro, not a special form, but I will not worry about this distinction in this document. `def` is closely related to the standard LISP special forms `set` and `setq`. The advantage of using `def` is that it adds your variable name to a list of def'ed variables kept in the global variable *variables*. If you use setq there is no easy way to find variables you have defined, as opposed to ones that are predefined. `def` always affects top level symbol bindings, not local bindings. It can not be used in function definitions to change local bindings.

Now the *symbol* `purchases` has a value associated with it: its value is our list of 22 numbers. If you type the symbol `purchases` at the evaluator then it will find the value of this symbol and print that value:

```
> purchases
(0 2 5 0 3 1 8 0 3 1 1 9 2 4 0 2 9 3 0 1 9 8)
>
```

We can now easily compute various numerical descriptive statistics for this data set:

```
> (mean purchases)
3.227273
> (median purchases)
2
> (standard-deviation purchases)
3.279544
> (interquartile-range purchases)
3.5
>
```

XLISP-STAT also supports elementwise arithmetic operations on lists of numbers. For example, we can add 1 to each of the purchases:

```
> (+ 1 purchases)
(1 3 6 1 4 2 9 1 4 2 2 10 3 5 1 3 10 4 1 2 10 9)
>
```

and after adding 1 we can compute the natural logarithms of the results:

```
> (log (+ 1 purchases))
(0 1.098612 1.791759 0 1.386294 0.6931472 2.197225 0 1.386294
0.6931472 0.6931472 2.302585 1.098612 1.609438 0 1.098612 2.302585
1.386294 0 0.6931472 2.302585 2.197225)
>
```

Exercises

For each of the following expressions try to predict what the evaluator will return. Then type them in, see what happens and try to explain any differences.

```
1)    (mean (list 1 2 3))
2)    (+ (list 1 2 3) 4)
3)    (* (list 1 2 3) (list 4 5 6))
```

3.2. Summary Statistics and Plots

Table 10 on page 54 of Devore and Peck (1986) gives precipitation levels recorded during the month of March in the Minneapolis-St. Paul area over a 30 year period. Let's enter these data into XLISP-STAT with the name `precipitation`:

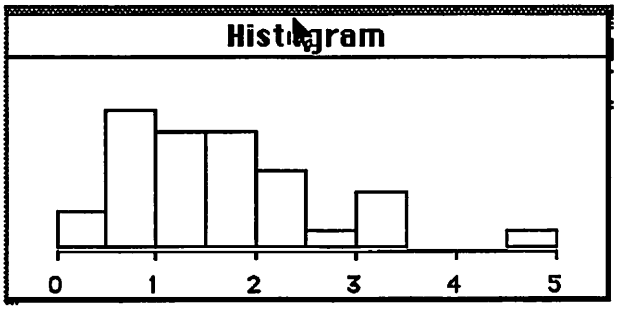
```
> (def precipitation
1>      (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20 3.30
2>            3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81 2.81 1.87
2>            1.18 1.35 4.75 2.48 .96 1.89 .90 2.05))
(0.77 1.74 0.81 1.2 1.95 1.2 0.47 1.43 3.37 2.2 3.3 3.09 1.51 2.1
0.52 1.62 1.31 0.32 0.59 0.81 2.81 1.87 1.18 1.35 4.75 2.48 0.96
1.89 0.9 2.05)
>
```

In typing this expression I have hit return a few times in order to make the typed expression easier to read. The evaluator responds by changing its prompt slightly: The numbers before the ">" represent the number of unmatched left parentheses.

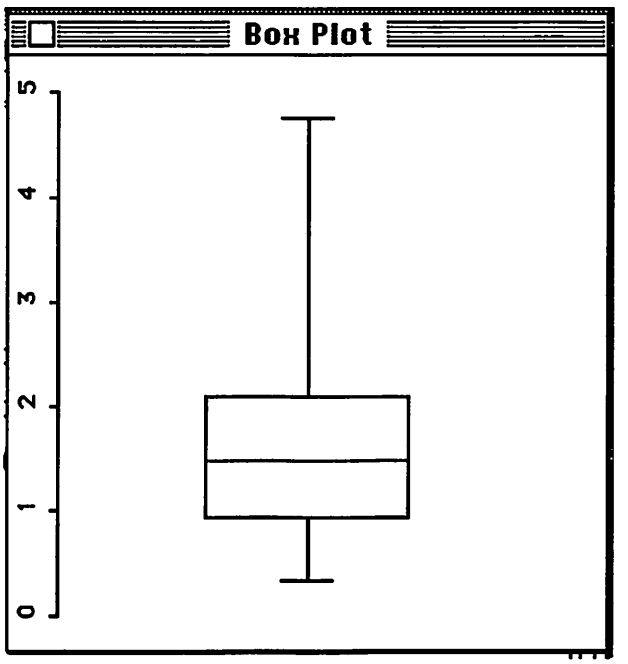
The `histogram` and `boxplot` functions can be used to obtain graphical representations of this data set:

```
> (histogram precipitation)
#<Object: #2>
> (boxplot precipitation)
#<Object: #2>
>
```

Each of these commands should cause a window with the appropriate graph to appear on your screen. The windows should look something like this



and this



Note that as each graph appears it becomes the active window. To get XLISP-STAT to accept further commands you have to click on the XLISP-STAT command window. You will have to click on the command window between the two commands shown here. The two functions return results that are printed something like this:

```
#<Object: #7>
```

These result will be used later to identify the window containing the plot. For the moment you can ignore them.

When you have several plot windows open you might want to close the command window so you can rearrange the plots more easily. You can do this by clicking in the command window's close box. You can later reopen the command window by selecting the **Show XLISP-STAT** item on the **Command** menu.

Here are some numerical summaries:

```
> (mean precipitation)
1.685
> (median precipitation)
1.47
> (standard-deviation precipitation)
1.0157
> (interquartile-range precipitation)
1.145
>
```

The distribution of this data set is somewhat skewed to the right. Notice the separation between the mean and the median. You might want to try a few simple transformations to see if you can symmetrize the data. Square root and log transformations can be computed using the expressions

```
      (sqrt precipitation)
```
and
```
      (log precipitation).
```

You should look at plots of the data to see if these transformations do indeed lead to a more symmetric shape. The means and medians of the transformed data are

```
> (mean (sqrt precipitation))
1.243006
> (median (sqrt precipitation))
1.212323
> (mean (log precipitation))
0.3405517
> (median (log precipitation))
0.384892
>
```

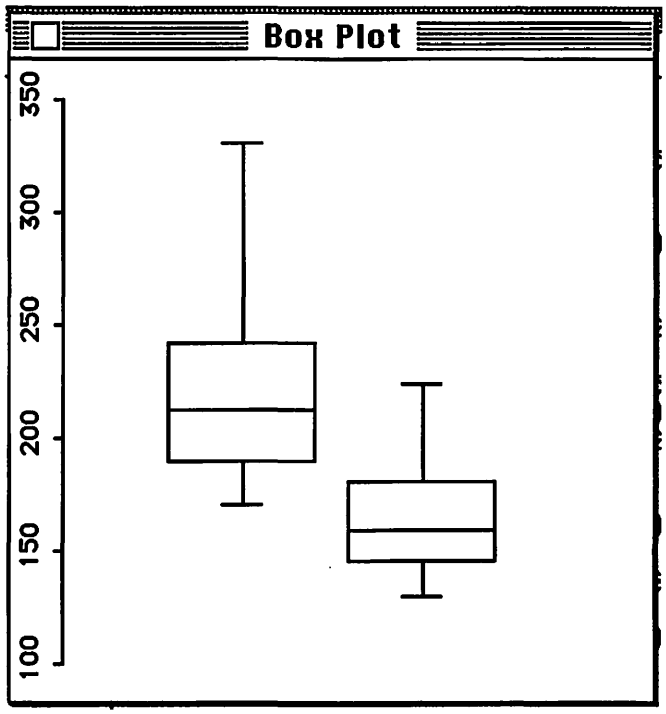
The boxplot function can also be used to produce parallel boxplots of two or more samples. It will do so if it is given a list of lists as its argument instead of a single list. As an example, let's use this function to compare serum total cholesterol values for samples of rural and urban Guatemalans (see Devore and Peck, 1986, p. 19, Example 3):

```
> (def urban (list 184 196 217 284 184 236 189 206 179 170 205 190
2>                  204 330 217 242 222 242 249 241))
(184 196 217 284 184 236 189 206 179 170 205 190 204 330 217 242
222 242 249 241)
> (def rural (list 166 146 144 204 158 143 158 180 223 194 194 175
2>                  171 155 143 145 131 181 148 144 220 129))
(166 146 144 204 158 143 158 180 223 194 194 175 171 155 143 145
131 181 148 144 220 129)
>
```

The parallel boxplot is obtained by

```
> (boxplot (list urban rural))
#<Object: #6>
>
```

and looks like this:



Exercises
The following exercises involve examples and problems from Devore and Peck (1986). The data sets are in files in the folder **Data** on the **XLISP-STAT** distribution disk and can be read in using the **Load** item in the **File** menu or using the `load` function (see Section 4.3 below). To use the **Load** item on the file menu select this item from the menu. This will bring up an Open File dialog window. Use this dialog to open the **Data** folder on the distribution disk. Now select one of the .lsp files (`car-prices.lsp` for the first exercise) and press the **Open** button. The file will be loaded and some variables will be defined for you. Loading file `car-prices.lsp` will define the single variable `car-prices`. Loading file `heating.lsp` will define two variables, `gas-heat` and `electric-heat`.

1) Devore and Peck (1986), p. 18, Example 2, give advertised prices for a sample of 50 used Japanese subcompact cars. Obtain some plots and summary statistics for this data. Experiment with some transformations of the data as well. The data set is called `car-prices` in the file `car-prices.lsp`. The prices are given in units of $1000; thus the price 2.39 represents $2390. The data have been sorted by their leading digit.

2) In Exercise 2.40 Devore and Peck (1986) give heating costs for a sample of apartments heated by gas and a sample of apartments heated by electricity. Obtain plots and summary statistics for these samples separately and look at a parallel box plot for the two samples. These data sets are called `gas-heat` and `electric-heat` in the file `heating.lsp`.

3.3. Two Dimensional Plots
Many single samples are actually collected over time. The precipitation data set used above is an example of this kind of data. In some cases it is reasonable to assume that the observations are independent of one another, but in other cases it is not. One way to check the data for some form of serial correlation or trend is to plot the observations against time, or against the order in which

they were obtained. I will use the `plot-points` function to produce a scatterplot of the precipitation data versus time. The `plot-points` function is called as

```
(plot-points x-variable y-variable)
```

Our `y-variable` will be precipitation, the variable we defined earlier. As our `x-variable` we would like to use a sequence of integers from 1 to 30. We could type these in ourselves, but there is an easier way. The function `iseq`, short for integer-sequence, generates a list of consecutive integers between two specified values. The general form for a call to this function is

```
(iseq start end).
```

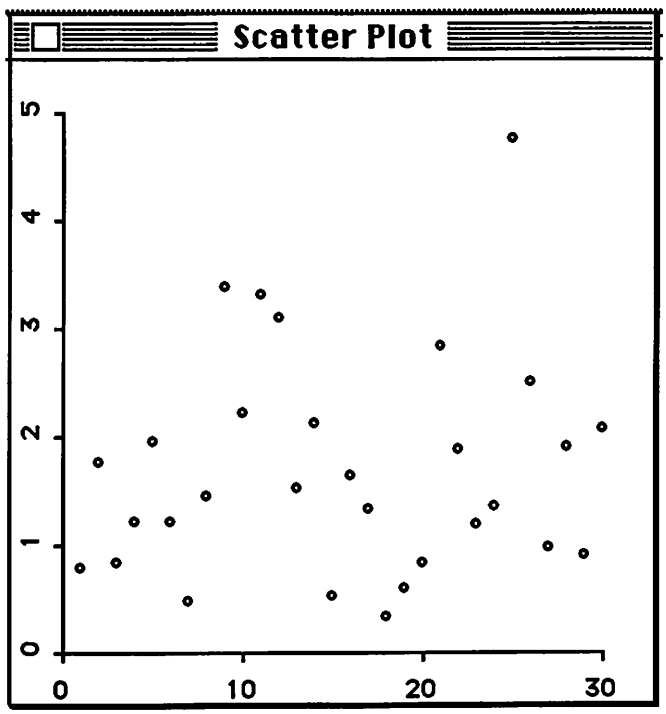
To generate the sequence we need we use

```
(iseq 1 30)
```

Thus to generate the scatter plot we type

```
> (plot-points (iseq 1 30) precipitation)
#<Object: #7>
>
```

and the result will look like this:



There does not appear to be much of a pattern to the data; an independence assumption may be reasonable.

Sometimes it is easier to see temporal patterns in a plot if the points are connected by lines. Try the above command with `plot-points` replaced by `plot-lines`.

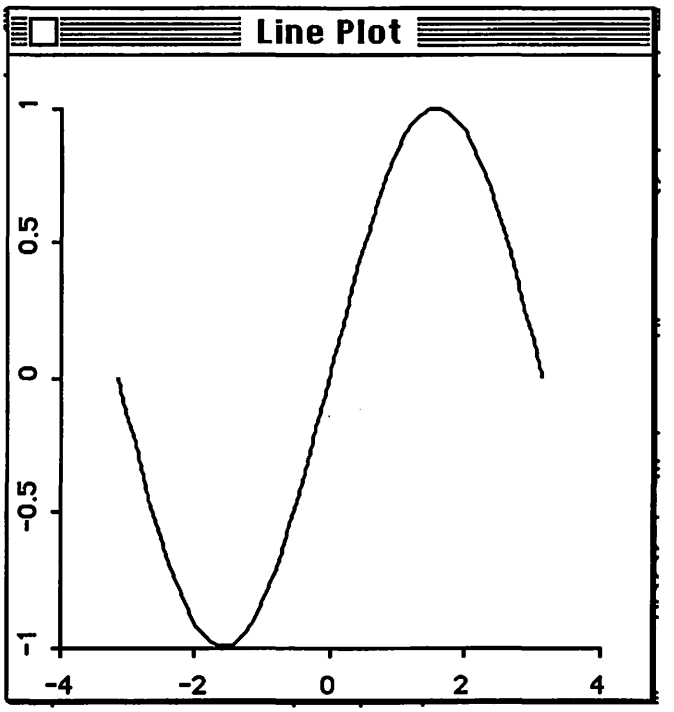
The plot-lines function can also be used to construct graphs of functions. Suppose you would like a plot of sin(x) from $-\pi$ to $+\pi$. The constant π is predefined as the variable `pi`. You can construct a list of n equally spaced real numbers between a and b using the expression

```
(rseq a b n).
```

Thus to draw the plot of sin(x) using 50 equally spaced points type

```
> (plot-lines (rseq (- pi) pi 50) (sin (rseq (- pi) pi 50)))
#<Object: #7>
>
```

The plot should look like this:

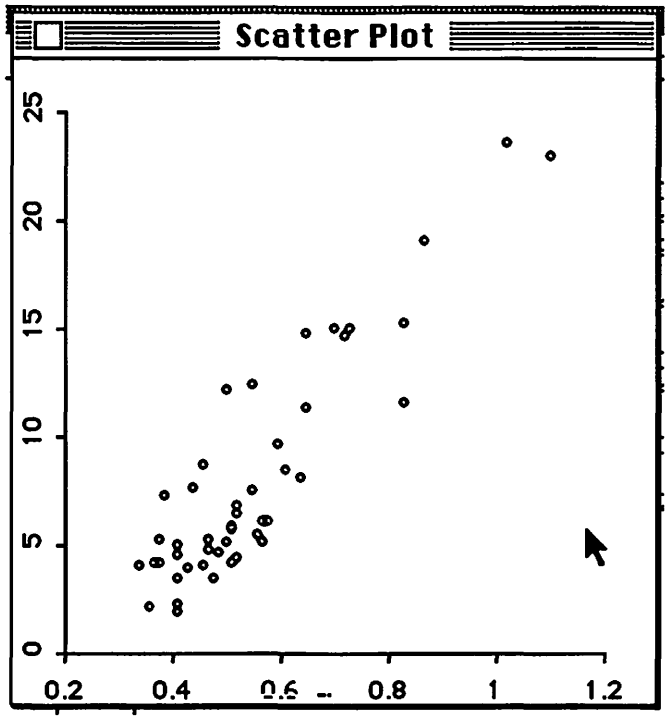


Scatterplots are of course particularly useful for examining the relationship between two numerical observations taken on the same subject. In Exercise 2.33 Devore and Peck (1986) give data for HC and CO emission recorded for 46 automobiles. The results can be placed in two variables, hc and co and these variable can then be plotted against one another with the plot-points function:

```
> (def hc (list .5 .46 .41 .44 .72 .83 .38 .60 .83 .34
2>              .37 .87 .65 .48 .51 .47 .56 .51 .57 .36
2>              .52 .58 .47 .65 .41 .39 .55 .64 .38 .50
2>              .73 .57 .41 1.02 1.10 .43 .41 .41 .52 .70
2>              .52 .51 .49 .61 .46 .55))
(0.5 0.46 0.41 ....
```

```
> (def co (list 5.01 8.60 4.95 7.51 14.59 11.53 5.21 9.62 15.13
2>              3.95 4.12 19.00 11.20 3.45 4.10 4.74 5.36 5.69
2>              6.02 2.03 6.78 6.02 5.22 14.67 4.42 7.24 12.30
2>              7.98 4.10 12.10 14.97 5.04 3.38 23.53 22.92 3.81
2>              1.85 2.26 4.29 14.93 6.35 5.79 4.62 8.43 3.99
2>              7.47))
(5.01 8.6 4.95 ....
> (plot-points hc co)
#<Object: #7>
>
```

Here is the resulting plot:



Exercises

1) Draw a graph of the function $f(x) = 2x + x^2$ between -2 and 3.

2) Devore and Peck (1986), Exercise 4.2 gives the age and CPK concentration, a measure of metabolic activity, recorded for 18 cross country skiers during a relay race. These data are in the variables `age` and `cpk` in the file `metabolism.lsp`. Plot the data and describe any relationship you observe between age and CPK concentration.

4. Some Useful Shortcuts

This section describes some additional features of XLISP-STAT that you may find useful.

4.1. Getting Help

On line help is available for many of the functions in XLISP-STAT. As an example, here is how you would get help for the function `median`.

```
> (help 'median)

Args: (x)
Returns the median of the elements of the list or array X.
NIL
```

Note the quote in front of `median`. `help` is itself a function, and its argument is the symbol representing the function `median`. To make sure `help` receives the symbol, not the value of the symbol, you need to quote the symbol.

If you are not sure about the name of a function you may still be able to get some help. Suppose you want to find out about functions related to the normal distribution. Most such functions will have `norm` as part of their name. The expression

```
(help* 'norm)
```

will print the help information for all symbols whose names contain the string "NORM":

```
> (help* 'norm)
-------------------------------------------------------------
NORM
Sorry, no help available on this item
-------------------------------------------------------------
NORMAL-QUANT

Args (p)
P is number-data with numbers between 0 and 1. Returns the P-th
quantile(s) of the standard normal distribution.
-------------------------------------------------------------
NORMAL-RAND

Args: (n)
Returns a list of N standard normal random numbers.
-------------------------------------------------------------
NORMAL-CDF

Args: (x)
X are number-data. Returns the values(s) of the standard normal
distribution function at X.
-------------------------------------------------------------
NIL
>
```

The symbol `norm` does not have a function definition and thus there is no help available. The term number-data refers to an item that is either a number or a compound data item whose

atoms are numbers. These help functions may take some time: information for many functions is contained in the file `xlisp.help` and has to be read in off the disk.

Let me briefly explain the notation used in the information printed by help to describe the arguments a function expects[1]. Most functions expect a fixed set of arguments, described in the help message by a line like

```
Args: (x y z)
```

Some functions can take one or more optional arguments. The arguments for such a function might be listed as

```
Args: (x &optional y (z t))
```

This means that `x` is required and `y` and `z` are optional. If the function is named `f`, it can be called as `(f x-val)`, `(f x-val y-val)` or `(f x-val y-val z-val)`. The list `(z t)` means that if `z` is not supplied it's default value is `T`. No explicit default value is specified for `y`; it's default value is therefore `NIL`. The arguments must be supplied in the order in which they are listed. Thus if you want to give the argument `z` you must also give a value for `y`.

Another form of optional argument is the *keyword* argument. The histogram function for example takes arguments

```
Args: (data &key (title "Histogram"))
```

The `data` argument is required, the `title` argument is an optional keyword argument. The default title is "Histogram". If you want to create a histogram of the `purchases` data set used in Section 3.1 with title "Purchases" use the expression

```
(histogram purchases :title "Purchases")
```

Thus to give a value for a keyword argument you give the keyword[2] for the argument, a symbol consisting of a colon followed by the argument name, and then the value for the argument. If a function can take several keyword arguments then these may be specified in any order, following the required and optional arguments.

Finally, some functions can take an arbitrary number of arguments. This is denoted by a line like

```
Args: (x &rest args)
```

The argument x is required, and zero or more additional arguments can be supplied.

4.2. More on the XLISP-STAT Top Level

After you have been working for a while you may want to find out what variables you have defined (using `def`). The function `variables` will produce a listing:

---

[1]The notation used corresponds to the specification of the argument lists in Common Lisp function definitions. XLISP's argument lists only allow &options, &rest and &aux arguments, not &key arguments. XLISP also does not allow default values to be specified for optional variables in the argument list. Keyword arguments to XLISP functions have to be implemented manually using getf on the remaining arguments, collected using &rest.

[2]Note that the keyword `:title` has not been quoted. Keyword symbols, symbols starting with a colon, are somewhat special. When a keyword symbol is created its value is set to itself. Thus a keyword symbol effectively evaluates to itself and does not need to be quoted.

```
> (variables)
CO
HC
RURAL
URBAN
PRECIPITATION
PURCHASES
NIL
>
```

If you are working on a 512K Macintosh you may occasionally want free up some space by getting rid of some variables you no longer need. You can do this using the undef function:

```
> (undef 'co)
CO
> (variables)
HC
RURAL
URBAN
PRECIPITATION
PURCHASES
NIL
>
```

Occasionally a calculation will take too long, or it will appear to have gotten stuck in some kind of loop. If you want to interrupt the calculation hold down the COMMAND key and the PERIOD. This should return you to the evaluator.

4.3. Loading Files

The data for the examples and exercises in this tutorial have been stored on files with names ending in ".lsp". On the XLISP-STAT distribution disk they can be found on the folder **Data**. Any variables you save (see the next sub section for details) will also be saved on files of this form. The data in these files can be read into XLISP-STAT with the load function. To load a file named "randu.lsp" type the expression

```
(load "randu.lsp")
```

or just

```
(load "randu")
```

If you give load a name that does not end in .lsp then load will add this suffix. Alternatively, you can use the **Load** command in the **File** menu.

4.4. Saving Your Work

If you want to record a session with XLISP-STAT you can do so using the dribble function. The expression

```
(dribble "myfile")
```

starts a recording. All expressions typed by you and all results typed by XLISP-STAT will be entered into the file named "myfile". The expression

```
(dribble)
```

stops the recording. Note that `(dribble "myfile")` starts a new file by the name "myfile". If you already have a file by that name its contents will be lost. Thus you can't use dribble to toggle recording to a single file on and off. This may be changed in a future version of XLISP-STAT.

`dribble` only records text that is typed, not plots. However, you can use the standard Macintosh shortcut COMMAND-SHIFT-3 to save a MacPaint image of the current screen. You can also choose the **Copy** command from the **Edit** menu while a plot window is the active window to save the contents of the plot window to the clip board. You can then open the scrap book from the apple menu and paste the plot into the scrap book.

Variables you define in XLISP-STAT only exist for the duration of the current session. If you quit from XLISP-STAT, or the program crashes, your data will be lost. To preserve your data you can use the save function. This function allows you to save one or more variable into a file. (Again a new file is created and any existing file by the same name is destroyed). To save the variable `precipitation` in a file called `"precipitation.lsp"` type

```
(save 'precipitation "precipitation")
```

Do not add the `".lsp"` suffix yourself; save will supply it. To save the two variables `precipitation` and `purchases` in the file `"examples.lsp"` type[1]

```
(save '(purchases precipitation) "examples")
```

The files `"precipitation.lsp"` and `"examples.lsp"` now contain a set of expression that, when read in with the load command, will recreate the variables precipitation and purchases. You can look at these files with an editor like MacWrite and you can prepare files with your own data by following these examples. For more information on data files see Section 11 below.

4.5. The XLISP-STAT Editor

The Macintosh version of XLISP-STAT includes a simple editor for preparing data files and function definitions. To edit an existing file select the **Open Edit** item on the **File** menu; to start a new file select **New Edit**. Other commands on the **File** menu can be used to save your file and to revert back to the saved version. The editor can only handle text files of less than 32K characters, does not include a search facility and does not handle tabs properly. It does, however, allow you to select a section of text representing one or more Lisp expressions and have these evaluated. To do this select the expressions you want to evaluate and then choose **Eval Selection** from the **Edit** menu. The returned values are not available, so this is only useful for producing side effects, such as defining functions.

[1] I have used a quoted list `'(purchases precipitation)` in this expression to pass the list of symbols to the `save` function. A longer alternative would be the expression `(list 'purchases 'precipitation)`.

5. More on Generating and Modifying Data

This section briefly summarizes some techniques for generating random and systematic data.

5.1. Generating Random Data

At present XLISP-STAT only has three functions for generating pseudo-random numbers. They are the functions `uniform-rand`, `normal-rand` and `cauchy-rand`. The expression

```
(uniform-rand 50)
```

for example will generate a list of 50 independent uniform random variables. The other two functions work similarly.

Cdf's for chi-squared, gamma, t, F and Beta distributions are available, but no quantile functions. These will hopefully be added later. At present there is also no way to reseed the random number generator. This may be changed in a future version of XLISP-STAT.

5.2. Generating Systematic Data

We have already used the functions iseq and rseq to generate equally spaced sequences of integers and real numbers. The function repeat is useful for generating sequences with a particular pattern. The general form of a call to repeat is

```
(repeat list pattern)
```

`pattern` must be either a single number or a list of numbers of the same length as `list`. If pattern is a single number then `repeat` simply repeats `list pattern` times. For example

```
> (repeat (list 1 2 3) 2)
(1 2 3 1 2 3)
```

If `pattern` is a list then each element of `list` is repeated the number of times indicated by the corresponding element of `pattern`. For example

```
> (repeat (list 1 2 3) (list 3 2 1))
(1 1 1 2 2 3)
```

In Section 6.2 below I generate the variables `density` and `variety` by typing them in directly. Using the `repeat` function we could have generated them like this:

```
(def density (repeat (repeat (list 1 2 3 4) (list 3 3 3 3)) 3))
(def variety (repeat (list 1 2 3) (list 12 12 12)))
```

5.3. Forming Subsets and Case Deletion

The `select` function allows you to select a single element or a group of elements from a list or vector. For example, if we define `x` by

```
(def x (list 3 7 5 9 12 3 14 2))
```

then `(select x i)` will return the i-th element of `x`. LISP, like the language C but in contrast to Fortran, numbers elements of list and vectors starting at zero. Thus the indices for the elements of `x` are 0, 1, 2, 3, 4, 5, 6, 7. So

```
> (select x 0)
3
> (select x 2)
5
```

To get a group of elements at once we can use a list of indices instead of a single index:

```
> (select x (list 0 2))
(3 5)
```

If you want to select all elements of x except element 2 you can use the expression `(remove 2 (iseq 0 7))` as the second argument to the function select:

```
> (remove 2 (iseq 0 7))
(0 1 3 4 5 6 7)
> (select x (remove 2 (iseq 0 7)))
(3 7 9 12 3 14 2)
```

Another approach is to use the logical function /= (meaning not equal) to tell you which indices are not equal to 2. The function `which` can then be used to return a list of all the indices for which the elements of it's argument are not nil:

```
> (/= 2 (iseq 0 7))
(T T NIL T T T T T)
> (which (/= 2 (iseq 0 7)))
(0 1 3 4 5 6 7)
> (select x (which (/= 2 (iseq 0 7))))
(3 7 9 12 3 14 2)
```

This approach is a little more cumbersome for deleting an element, but it is more general. The expression `(select x (which (< 3 x)))`, for example, returns all elements in x that are greater than 3:

```
> (select x (which (< 3 x)))
(7 5 9 12 14)
```

5.5. Combining Several Lists

At times you may want to combine several short lists into a single longer variable. This can be done using the append function. For example, if you have three variables x, y and z constructed by the expressions

```
(def x (list 1 2 3))
(def y (list 4))
(def z (list 5 6 7 8))
```

then the expression

```
(append x y z)
```

will return the list

```
(1 2 3 4 5 6 7 8).
```

5.6. Modifying Data

So far when I have asked you to type in a list of numbers I have been assuming that you will type the list correctly. If you made an error you had to retype the entire def expression. Since you can use cut and paste this is really not too serious. However it would be nice to be able to replace the values in a list after you have typed it in. The `setf` special form is used for this. Suppose you would like to change the 12 in the list `x` used in the previous subsection to 11. The expression

```
(setf (select x 4) 11)
```

will make this replacement:

```
> (setf (select x 4) 11)
11
> x
(3 7 5 9 11 3 14 2)
```

The general form of `setf` is

```
(setf form value)
```

where `form` is the expression you would use to select a single element or a group of elements from `x` and `value` is the value you would like that element to have, or the list of the values for the elements in the group. Thus the expression

```
(setf (select x (list 0 2)) (list 15 16))
```

changes the values of elements 0 and 2 to 15 and 16:

```
> (setf (select x (list 0 2)) (list 15 16))
(15 16)
> x
(15 7 16 9 11 3 14 2)
```

A note of caution is needed here. LISP symbols are merely pointers to different objects. When you assign a name to an object with the `def` command you are not producing a new object. Thus

```
(def x (list 1 2 3 4))
(def y x)
```

means that x and y are two different names for the same thing. As a result, if we change an element of (the object referred to by) x with `setf` then we are also changing the element of (the object referred to by) y, since both x and y refer to the same object. If you want to make a copy of x and store it in y before you make changes to x then you must do so explicitly using, say, the `copy-list` function. The expression

```
(def y (copy-list x))
```

will make a copy of x and set the value of y to that copy. Now x and y refer to different objects and changes to x will not affect y.

5.7. User Initialization File

After loading in all its program files and before giving you your first prompt XLISP-STAT looks to see if there is a file named statinit.lsp in the startup folder. If there is one it will be loaded. You can use this file to load any data sets you would like to have available or to define functions of your own. See Section 8 below for more information on defining functions.

6. More Elaborate Plots

The plots described so far were designed for exploring the distribution of a single variable and the relationship among two variables. This section describes some plots and techniques that are useful for exploring the relationship among three or more variables. The techniques and plots described in the first four subsections are simple, requiring only simple commands to the evaluator and mouse actions. The fifth subsection introduces the concept of a plot *object* and the idea of sending *messages* to an object from the command window. These ideas are used to add lines and curves to scatter plots, and the basic concepts of objects and messages will be used again in the next section on regression models. The final two subsections are somewhat more advanced. The sixth subsection shows how Lisp iteration can be used to construct a dynamic simulation - a plot movie, and the final subsection shows how to attach a function of your own to a mouse click in a scatter plot window.

6.1. Spinning Plots

If we are interested in exploring the relationship among three variables then it is natural to imagine constructing a three dimensional scatterplot of the data. Of course we can only see a two dimensional projection of the plot on a computer screen - any depth that you might be able to perceive by looking at a wire model of the data is lost. One approach to try to recover some of this depth perception is to rotate the points around some axis. The `spin-plot` function allows you to construct a rotatable three dimensional plot.

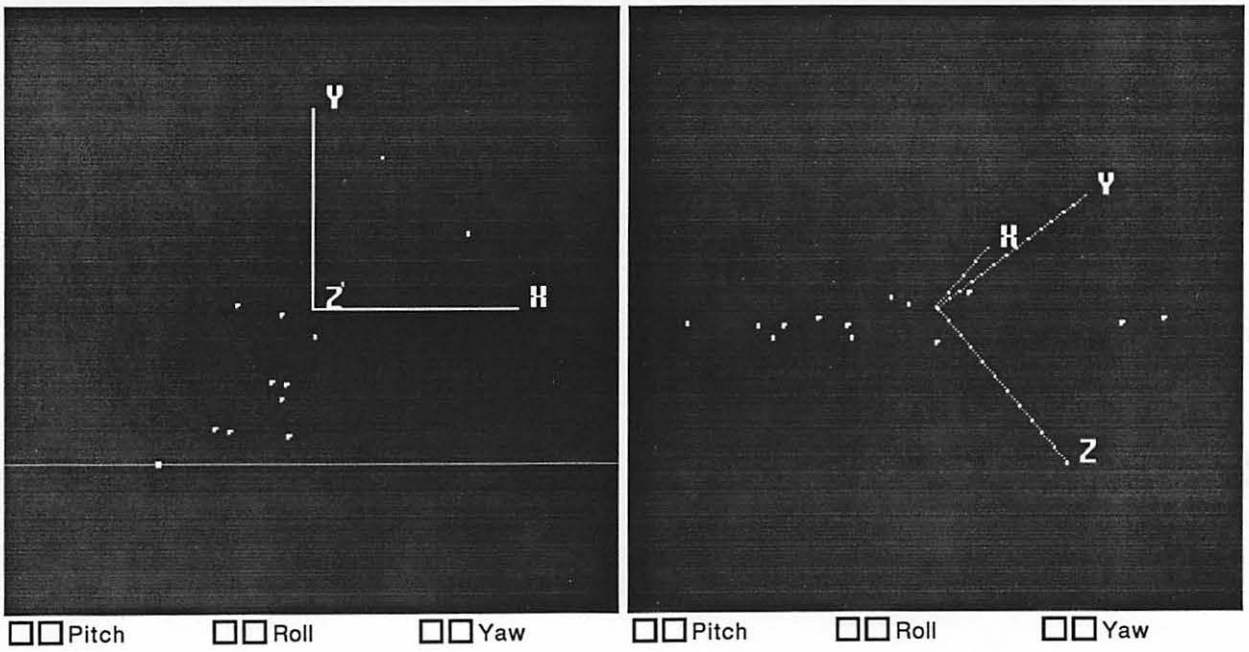
As an example let's look at the data of Example 6 on p. 509 of Devore and Peck (1986). The relationship between a phosphate absorption index and the amount of extractable iron and aluminum in a sediment is of interest. The data can be entered with the expressions

```
(def iron (list 61 175 111 124 130 173 169 169 160 224 257
                333 199))
(def aluminum (list 13 21 24 23 64 38 33 61 39 71 112 88 54))
(def absorption (list 4 18 14 18 26 26 21 30 28 36 65 62 40))
```

The expression

```
(spin-plot (list absorption iron aluminum))
```

produces the plot shown below on the left. The argument to `spin-plot` is a list of three lists or vectors, representing the X, Y and Z variables. You can rotate the plot by pointing at one of the **Pitch**, **Roll** or **Yaw** squares and pressing the mouse button. By rotating the plot you can see that the points seem to fall close to a plane. The plot on the right shows the data viewed along the plane. A linear model should describe this data quite well.



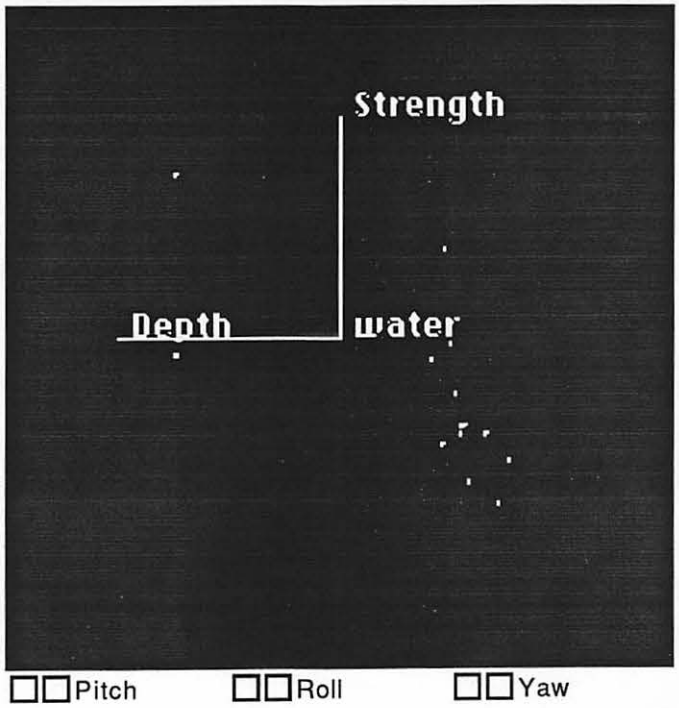
As a second example, with the data defined by

```
(def strength (list 14.7 48.0 25.6 10.0 16.0 16.8 20.7 38.8
                    16.9 27.0 16.0 24.9 7.3 12.8))
(def depth (list 8.9 36.6 36.8 6.1 6.9 6.9 7.3 8.4 6.5 8.0
                 4.5 9.9 2.9 2.0))
(def water (list 31.5 27.0 25.9 39.1 39.2 38.3 33.9 33.8
                 27.9 33.1 26.3 37.8 34.6 36.4))
```

(taken from Devore and Peck, 1986, problem 12.18) the expression

```
(spin-plot (list water depth strength)
               :variable-labels
                 (list "Water" "Depth" "Strength"))
```

produces a plot that can be rotated to look like this:



These data concern samples of thawed permafrost soil. `strength` is the shear strength, and `water` is the percentage water content. `depth` is the depth at which the sample was taken. The plot shows that a linear model will not fit well. Devore and Peck (1986) suggest fitting a model with quadratic terms to this data.

The function `spin-plot` also accepts the additional keyword argument scale. If scale is `T`, the default, then the data are centered at the midranges of the three variables, and all three variables are scaled to fit the plot. If scale is `NIL` the data are assumed to be scaled between -1 and 1, and the plot is rotated about the origin. Thus if you want to center your plot at the means of the variables and scale all observations by the same amount you can use the expression

```
(spin-plot (list (/ (- water (mean water)) 20)
                 (/ (- depth (mean depth) 20)
                 (/ (- strength (mean strength) 20))
           :scale nil)
```

Exercises

1) An Upstate New York business machine company used to include a random number generator called RANDU in its software library. This generator was supposed to produce numbers that behaved like i. i. d. uniform random variables. The data set `randu` in the file `randu.lsp` in the **Data** folder consists of a list of three lists of numbers. These lists are consecutive triples of numbers produced by RANDU. Use spin-plot to see if you can spot any unusual features in this sample.

6.2. Scatterplot Matrices

Another approach to graphing a set of variables is to look at a matrix of all possible pairwise scatterplots of the variables. The scatterplot-matrix function will produce such a plot. The data

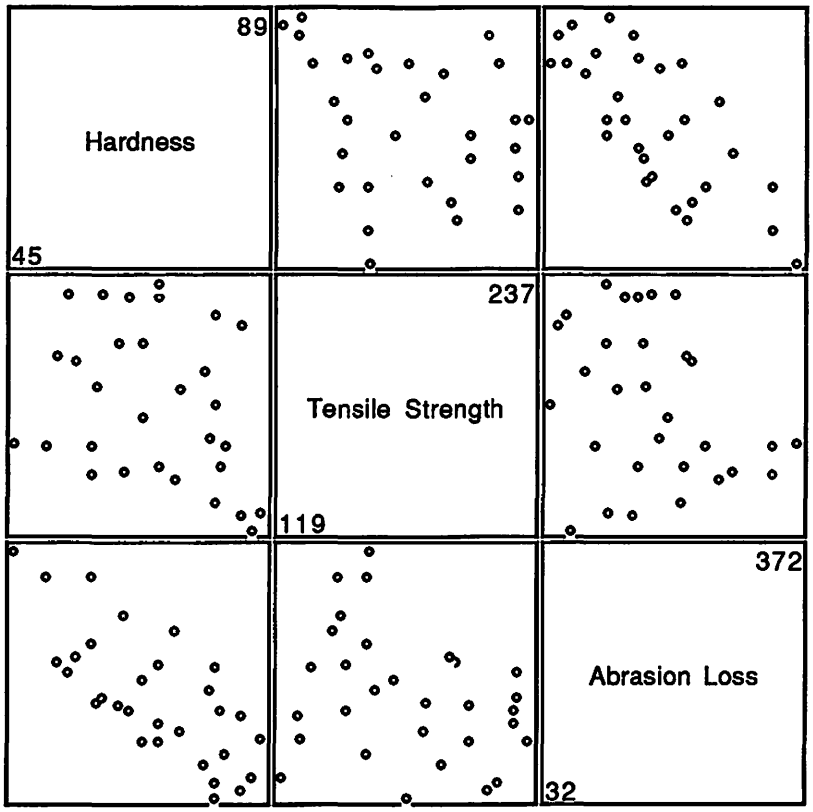
```
(def hardness (list 45 55 61 66 71 71 81 86 53 60 64 68 79 81 56
                    68 75 83 88 59 71 80 82 89 51 59 65 74 81 86))
```

```
(def tensile-strength (list 162 233 232 231 231 237 224 219 203
                            189 210 210 196 180 200 173 188 161
                            119 161 151 165 151 128 161 146 148
                            144 134 127))
(def abrasion-loss (list 372 206 175 154 136 112 55 45 221 166 164
                         113  82  32 228 196 128 97 64 249 219 186
                         155 114 341 340 284 267 215 148))
```

were produced in a study of the abrasion loss in rubber tyres and the expression

```
(scatterplot-matrix
  (list hardness tensile-strength abrasion-loss)
  :variable-labels
    (list "Hardness" "Tensile Strength" "Abrasion Loss"))
```

produces this scatterplot matrix:



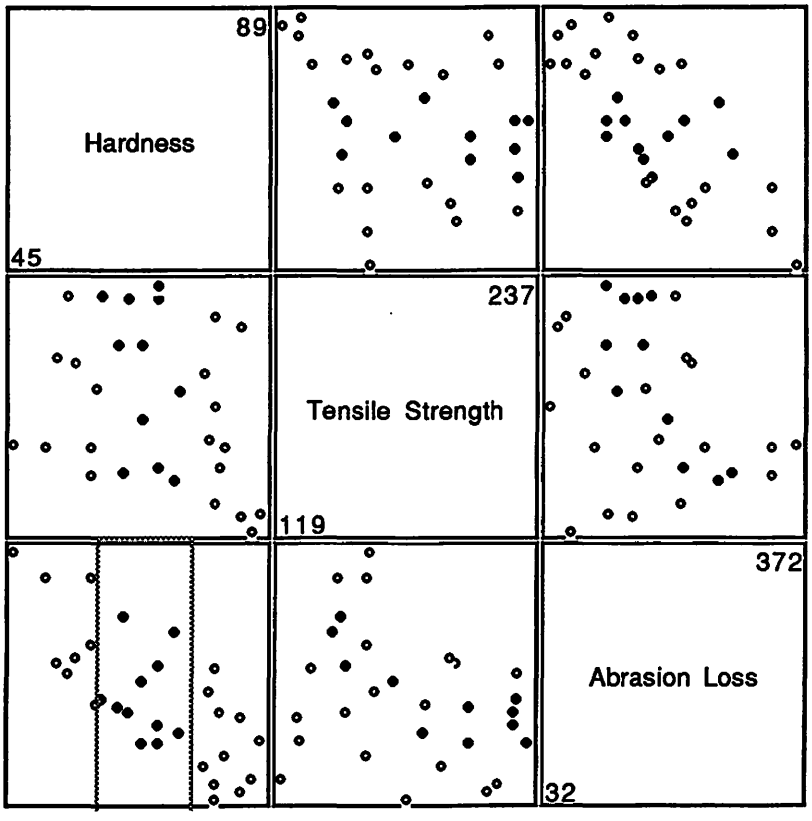
The plot of `abrasion-loss` against `tensile-strength` gives you an idea of the joint variation in these two variables. But `hardness` varies from point to point as well. To get an understanding of the relationship among all three variables it would be nice to be able to fix `hardness` at various levels and look at the way the plot of `abrasion-loss` against `tensile-strength` changes as you change these levels. You can do this kind of exploration in the scatterplot matrix by using the two highlighting techniques **Selecting** and **Brushing**.

Selecting: Your plot is in the selecting mode when the **Brush Plot** item in the **Scatmat** menu does not have a check mark in front of it. This is the default setting. In this mode select a point by clicking the mouse on top of it. To select a group of

points drag a selection rectangle around the group. If the group does not fit in a rectangle you can build up your selection by holding down the shift key as you click or drag. If you click without the shift key any existing selection will be unselected; when the shift key is down selected points remain selected.

Brushing: You can enter the brushing mode by choosing **Brush Plot** from the **Scatmat** menu. In this mode the **Brush Plot** item will be checked in the menu and a dashed rectangle, the brush, will be attached to your cursor. As you move the brush across the plot points in the brush will be highlighted, points outside of the brush will not be highlighted unless they are marked as selected. To select points in the brushing mode (make their highlighting permanent) hold the mouse button down as you move the brush.

In the plot below the points within the middle of the Hardness range have been highlighted using a long, thin brush (you can change the size of your brush using the **Resize Brush** command on the **Scatmat** menu). In the plot of Abrasion Loss against Tensile Strength you can see that the highlighted points seem to follow a curve. If you want to fit a model to this data this suggests fitting a model that accounts for this curvature.

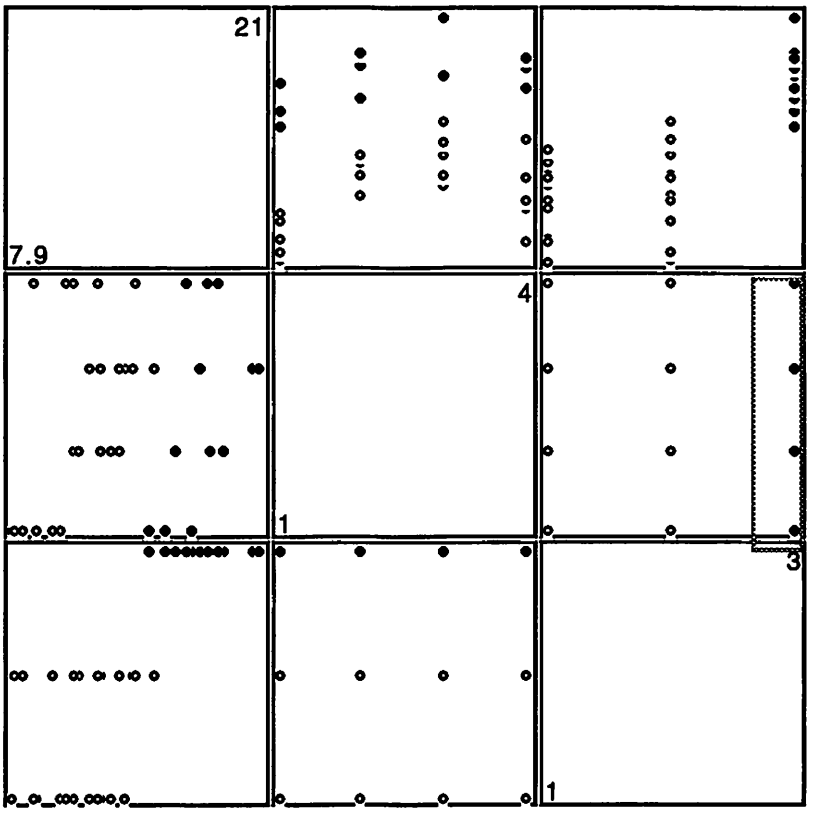


A scatterplot matrix is also useful for examining the relationship between a quantitative variable and several categorical variables. In the data

```
(def yield (list 7.9 9.2 10.5 11.2 12.8 13.3 12.1 12.6 14.0 9.1
                 10.8 12.5 8.1 8.6 10.1 11.5 12.7 13.7 13.7 14.4
                 15.5 11.3 12.5 14.5 15.3 16.1 17.5 16.6 18.5 19.2
                 18.0 20.8 21 17.2 18.4 18.9 ))
(def density (list 1 1 1 2 2 2 3 3 3 4 4 4 1 1 1 2 2 2 3 3 3 4 4 4
                   1 1 1 2 2 2 3 3 3 4 4 4))
```

```
(def variety (list 1 1 1  1 1 1  1 1 1  1 1 1  2 2 2  2 2 2  2 2 2
                   2 2 2  3 3 3  3 3 3  3 3 3  3 3 3  3 3 3))
```

from Devore and Peck (1986), p. 595, Example 14, the yield of tomato plants is recorded for an experiment run at four different planting densities and using three different varieties. In the plot below a long, thin brush has been used to highlight the points in the third variety. If there is no interaction between the varieties and the density then the shape of the highlighted points should move approximately in parallel as the brush is moved from one variety to another.



Like `spin-plot`, the function `scatterplot-matrix` also accepts the optional keyword argument scale.

Exercises

1) Devore and Peck (1986), Exercise 13.62, describe an experiment to determine the effect of oxygen concentration on fermentation end products. Four oxygen concentrations and two types of sugar were used. The data are

```
(def ethanol (list .59 .30 .25 .03 .44 .18 .13 .02 .22 .23 .07
                   .00 .12 .13 .00 .01))
(def oxygen (list 1 1 1 1 2 2 2 2 3 3 3 3 4 4 4 4))
(def sugar (list 1 1 2 2 1 1 2 2 1 1 2 2 1 1 2 2 ))
```

and are on file oxygen.lsp. Use a scatterplot matrix to examine these data.

2) Use scatterplot matrices to examine the data in the examples and exercises of Section 6.1 above.

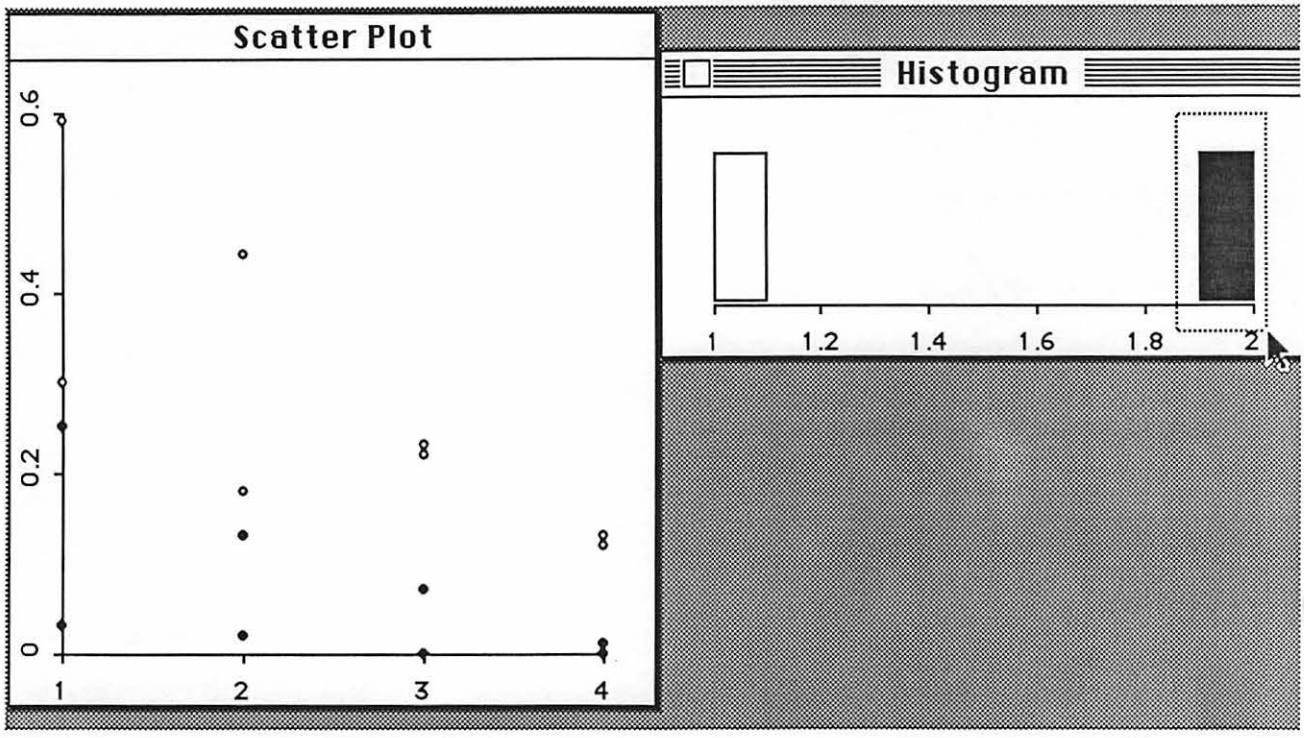
6.3. Interacting with Individual Plots

Rotating plots and scatterplot matrices are interactive plots. Simple scatter plots also allow some interaction: If you select the **Show Labels** option in the **Plot** menu a label will appear next to a highlighted point. You can use either the selecting or the brushing mode to highlight points. The default labels are of the form "Point 0", Point 1", ... (In LISP it is conventional to start numbering indices with 0, rather than 1.) Most plotting functions allow you to supply a list of case labels using the `:case-labels` keyword.

Another option, useful in viewing large data sets is to remove a subset of the points from your plot. This can be done by selecting the points you want to remove and then choosing **Remove Selection** from the plot menu. This option is available for histograms as well.

6.4. Linked Plots

When you brush or select in a scatterplot matrix you are looking at the interaction of a set of separate scatterplots.[1] You can construct your own set of interacting plots by choosing the **Link View** option from the menus of the plots you want to link. For example, using the data from Exercise 1 in 6.2 we can put `ethanol` and `oxygen` in a scatterplot and `sugar` in a histogram.If we link these two plots then selecting one of the two sugar groups in the histogram highlights the corresponding points in the scatterplot:



If you want to be able to select the points with a particular label you can use the `name-list` function to generate a window with a list of names in it. This window can be linked with any plot, and selecting a name in a `name-list` will then highlight the corresponding points in the linked plots. You can use the `name-list` function with a numerical argument; e. g.

[1] According to Stuetzle (1987) the idea to link several plots was first suggested by McDonald (1982).

```
(name-list 10)
```

will generate a list with the names "Point 0" , ..., "Point 9", or you can give it a list of case labels of your own.

Exercise
Try to use linked scatter plots and histograms on the data in the examples and exercises of Sections 6.1 and 6.2.

6.5. Modifying a Scatter Plot
After producing a scatterplot of a data set we might like to add a line, a regression line for example, to the plot. As an example, Devore and Peck (1986), p. 105, Example 2, describe a data set collected to examine the effect of bicycle lanes on drivers and bicyclists. The variables given by

```
(def travel-space
     (list 12.8 12.9 12.9 13.6 14.5 14.6 15.1 17.5
           19.5 20.8))
(def separation
     (list 5.5 6.2 6.3 7.0 7.8 8.3 7.1 10.0 10.8 11.0))
```

represent the distance between the cyclist and the roadway center line and the distance between the cyclist and a passing car, respectively, recorded in ten cases. A regression line fit to these data, with `separation` as the dependent variable, has a slope of 0.66 and an intercept of -2.18. Let's see how to add this line to a scatterplot of the data.

We can use the expression

```
(plot-points travel-space separation)
```

to produce a scatterplot of these points. To be able to add a line to the plot, however, we must be able to refer to it within XLISP-STAT. To accomplish this let's assign the result returned by the `point-plot` function to a symbol[1]:

```
(def myplot (plot-points travel-space separation))
```

The result returned by `plot-points` is an xlisp *object*.. To use an object you have to send it *messages*. This is done using a special syntax of the form

```
(object message argument1 ... )
```

I will use the expression

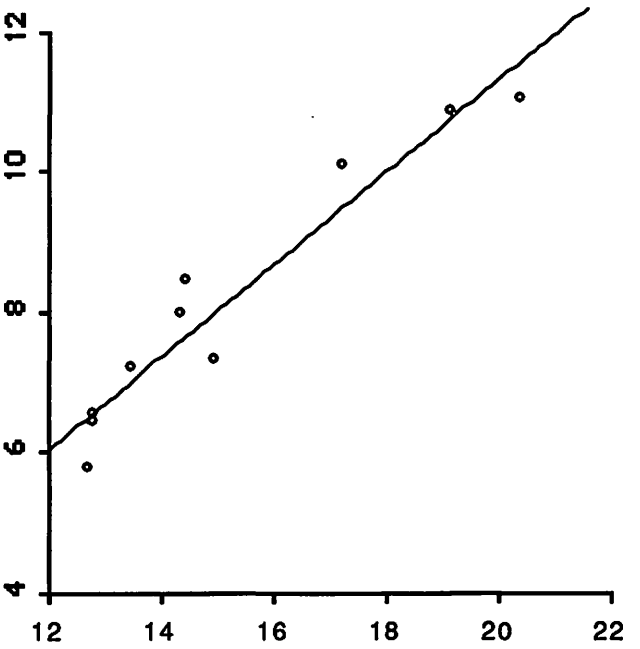
```
(myplot :abline -2.18 0.66)
```

to tell myplot to add the graph of a line a + b x, with a = -2.18 and b = 0.66, to itself. The message is `:abline`, the numbers -2.18 and 0.66 are the arguments. Messages are always Lisp keywords; that is, they are symbols that start with a colon. Before typing in this expression you might want to

---

[1]The result returned by `plot-points` is printed something like `#<Object: #5>`. This is not the *value* returned by the function, just its *printed representation* . There are several other data types that are printed this way; file streams, as returned by the openi and openo functions, are one example. For the most part you can ignore these printed results. There is one unfortunate feature, however: the form `#<...>` means that there is no printed form of this data type that the Lisp reader can understand. As a result, if you forget to give your plot a name you can't cut and paste the `#<Object: #5>` into a def expression - you have to redo the plot.

resize and rearrange the command window so you can see the plot and type commands at the same time. Once you have resized the command window you can easily switch between the small window and a full size window by clicking in the zoom box at the right end corner of the window title bar. The resulting plot looks like this:



Scatter plot objects understand a number of other messages. One other message is the :help message:

```
> (myplot :help)
Two Dimensional Plot

To get help on a particular topic for an object m type

     (m :help topic).

TOPIC should be a keyword (i. e. it should start with a colon).

Help is available on the following topics:
:CLASS
:SET-MOUSE
:ABLINE
:ADD-FUNCTION
:ADD-LINES
:ADD-POINTS
:CLEAR
:HELP
:CLASS
:SHOW-WINDOW
:UNLINK
:LINK
:COPY-TO-CLIP
:REDRAW
```

```
:REMOVE
:REFCON
:IS-SHOWING
:IS-HILITED
:IS-SELECTED
:HELP
```

The list of topics will be the same for all scatter plots but will be somewhat different for rotating plots, scatterplot matrices or histograms.[1]

The `:clear` message, as its name suggests, clears the plot and allows you to build up a new plot from scratch. Two other useful messages are `:add-points` and `:add-lines`. To find out how to use them we can use the help message with `:add-points` or `:add-lines` as arguments:

```
> (myplot :help :add-points)

Message args: (x y &key case-labels)
Adds points given by X and Y to the plot. CASE-LABELS, if
supplied, should be lists of character strings.
NIL

> (myplot :help :add-lines)

Message args: (x y)
Adds connected lines between points given by X and Y to the plot.
NIL
```

The plot produced above shows some curvature in the data. A regression of `separation` on a linear and a quadratic term in `travel-space` produces estimates of -16.41924 for the constant, 2.432667 as the coefficient of the linear term and -0.05339121 as the coefficient of the quadratic term. Let's use the `:clear`, `:add-points` and `:add-lines` messages to change myplot to show the data along with the fitted quadratic model. First we use the expressions

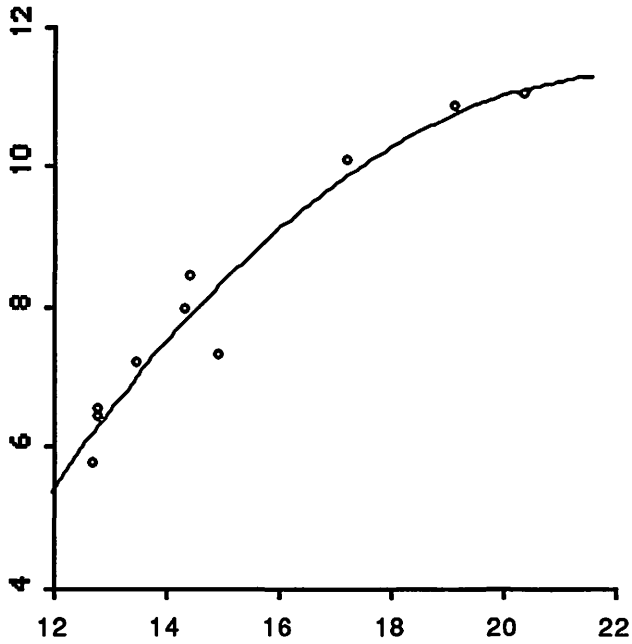
```
(def x (rseq 12 22 50))
(def y (+ -16.41924 (* 2.432667 x) (* -0.05339121 (* x x))))
```

to define x as a grid of 50 equally spaced points between 12 and 22 and y as the fitted response at these x values. Then the expressions

```
(myplot :clear)
(myplot :add-points travel-space separation)
(myplot :add-lines x y)
```

change `myplot` to look like this:

[1]Note that the `:help` message, like all other messages, can only be sent to an object you have created, not to the function you would use to create it. For example, `(plot-points :help)` will not work since `plot-points` is a function, not an object.



Of course we could have used `plot-points` to get a new plot and just modified that plot with `:add-lines`, but the approach used above allowed us to try out all three messages.

6.6. Dynamic Simulations

As another illustration of what you can do by modifying existing plots let's construct a dynamic simulation - a movie - to examine the variation in the shape of histograms of samples from a standard normal distribution. To start off, use the expression

```
(def h (histogram (normal-rand 20)))
```

to construct a single histogram and save its plot object as h. The `:clear` message is available for histograms as well. As you can see from its help message

```
> (h :help :clear)

Message args: (&optional draw t)
Clears the plot data. If DRAW is nil the plot is redrawn;
otherwise its current screen image remains unchanged.
NIL
```

it takes an optional argument. If this argument is nil then the plot will not actually be redrawn until some other event causes it to be redrawn. This is useful for dynamic simulations. Rearrange and resize your windows until you can see the histogram window even when the command window is the active window. Then type the expression[1]

---

[1] `dotimes` is one of several Lisp looping constructs. It is a special form with the syntax `(dotimes (var count) expr ....)`. The loop is repeated `count` times, with `i` bound to 0, 1, ..., `count` - 1. Other looping constructs are `dolist`, `do` and `do*`.

```
(dotimes (i 50)
         (h :clear nil)
         (h :add-points (normal-rand 20)))
```

This should produce a sequence of 50 histograms, each based on a sample of size 20. By giving the optional `nil` argument to the `:clear` message you have insured that each histogram stays on the screen until the next one is ready to be drawn. Try the example again without this argument and see what difference it makes.

Programmed dynamic simulations may provide another approach to viewing the relation among several variables. As a simple example, suppose we want to examine the relation between the variables `abrasion-loss` and `hardness` introduced in Section 6.2 above. Let's start with a histogram of `abrasion-loss` produced by the expression

```
(def h (histogram abrasion-loss))
```

The messages `:is-selected`, `:is-hilited` and `:is-showing` are particularly useful for dynamic simulations. Here is the help information for `:is-selected` in a histogram:

```
> (h :help :is-selected)

Message args: (point &optional set)
Without the SET argument returns T if POINT is selected, NIL
otherwise. If set is supplied POINT is selected if set is not NIL,
unselected if SET is NIL.
NIL
```

Thus you can use this message to determine whether a point is currently selected and also to set its selection state. Again rearrange the windows so you can see the histogram while typing in the command window and type the expression

```
(dolist (i (order hardness))
        (h :is-selected i t)
        (h :is-selected i nil))
```

The expression `(order hardness)` produces the list of indices of the ordered values of hardness. Thus the first element of the result is the index of the smallest element of hardness, the second element is the index of the second smallest element of hardness, etc.. The loop moves through each of these indices and selects and unselects the corresponding point.

The result on the screen is very similar to the result of brushing a `hardness` histogram linked to an `abrasion-loss` histogram from left to right. The drawback of this approach is that it is harder to write an expression than to use a mouse. On the other hand, when brushing with a mouse you tend to focus your attention on the plot you are brushing, rather than on the other linked plots. When you run a dynamic simulation you do not have to do anything while the simulation is running and can therefore concentrate fully on the results.

Like most Lisp systems XLISP-STAT is not ideally suited to real time simulations because of the need for *garbage collection*, to reclaim dynamically allocated storage. This is the reason that the simulations in this section will occasionally pause for a second or two. Nevertheless, in a simple simulation like the last one each iteration may still be too fast for you to be able to pick up any pattern. To slow things down you can add some extra busy work to each iteration. For example, you could use

```
(dolist (i (order hardness))
        (h :is-selected i t)
        (dotimes (i 100))
        (h :is-selected i nil))
```

in place of the previous expression.

6.7. Customizing Plot Actions

This section requires some more advanced Lisp concepts. It is placed here for reference and should probably be skipped by most readers.

All plot objects allow a user defined action to take place when the plot is redrawn. To install this action use the :set-redraw message. As a simple example, the expression

```
(myplot :set-redraw
        #'(lambda (plot) (princ "Plot has been redrawn")))
```

will print a message each time `myplot` is redrawn. A more interesting use of this feature might be to recalculate a regression model after a set of points have been removed from a related plot.

The scatter plot object is set up to allow you to replace or supplement the default action taken on a mouse click by an action of your own. To do this you use the message `:set-mouse`. As an example, the expression

```
(myplot :set-mouse #'(lambda (plot x y) (print (list x y))))
```

installs a mouse action that will print the coordinates at which the mouse was clicked. A more elaborate use of this feature might be to redraw the plot in a way that depends on where the click occurred. If the function installed with `:set-mouse` returns `NIL` the standard mouse action will be taken after the function is evaluated; otherwise no further action is taken by the plot.

7. Regression

Regression models have been implemented using XLISP's object and message sending facilities. These were introduced above in Section 6.5. You might want to review that section briefly before reading on.

Let's fit a simple regression model to the bicycle data of Section 6.5. The dependent variable is `separation` and the independent variable is `travel-space`. To form a regression model use the `regression-model` function:

```
> (regression-model travel-space separation)

Least Squares Estimates:

Constant               -2.182472   (1.056688)
Variable  0:            0.6603419   (0.06747931)

R Squared:              0.922901
Sigma hat:             0.5821083
Number of cases:              10
Degrees of freedom:            8

#<Object: #6>
```

The basic syntax for the regression-model function is

```
(regression-model x y)
```

For a simple regression x can be a single list or vector. For a multiple regression x can be a list of lists or vectors or a matrix. The `regression-model` function also takes three optional keyword arguments, `:intercept`, `:print`, and `:weights`. Both `:intercept` and `:print` are `T` by default. To get a model without an intercept use the expression

```
(regression-model x y :intercept nil)
```

To form a weighted regression model use the expression

```
(regression-model x y :weights w)
```

where `w` is a list or vector of weights the same length as `y`. The variances of the errors are assumed to be inversely proportional to the weights `w`.

The `regression-model` function prints a very simple summary of the fit model and returns a model object as its result. To be able to examine the model further assign the returned model object to a variable using an expression like[1]

```
(def bikes
  (regression-model travel-space separation :print nil))
```

[1]Recall from Section 6.5 that `#<Object: #5>` is the printed representation of the model object returned by `regression-model`. Unfortunately you can't cut and paste it into the `def`, but of course you can cut and paste the `regression-model` expression.

I have given the keyword argument `:print nil` to suppress the printing of the summary, since we have already seen it. To find out what messages are available use the `:help` message:

```
> (bikes :help)
Normal Linear Regression Model

To get help on a particular topic for an object m type

     (m :help topic).

TOPIC should be a keyword (i. e. it should start with a colon).

Help is available on the following topics:
:CLASS
:PLOT-BAYES-RESIDUALS
:PLOT-RESIDUALS
:COEF-STANDARD-ERRORS
:XTXINV
:COEF-ESTIMATES
:R-SQUARED
:SIGMA-HAT
:SUM-OF-SQUARES
:RESIDUALS
:FIT-VALUES
:LEVERAGES
:X-MATRIX
:DF
:NUM-COEFS
:NUM-CASES
:BASIS
:WEIGHTS
:INTERCEPT
:Y
:X
:DISPLAY
:COMPUTE
:SAVE
:HELP

NIL
>
```

Many of these messages are self explanatory, and many have already been used by the `:display` message, which `regression-model` sends to the new model to print the summary. As examples let's try the `:coef-estimates` and `:coef-standard-errors` messages[1]:

[1]Ordinarily the entries in the lists returned by these messages correspond simply to the intercept, if one is included in the model, followed by the independent variables as they were supplied to `regression-model`. However, if degeneracy is detected during computations some variables will not be used in the fit; they will be marked as aliased in the printed summary. The indices of the variables used can be obtained by the `:basis` message; the entries in the list returned by `:coef-estimates` correspond to the intercept, if appropriate, followed by the coefficients of the elements in the basis. The messages `:x-matrix` and `:xtxinv` are similar in that they use only the variables in the basis.

```
> (bikes :coef-estimates)
(-2.182472 0.6603419)
> (bikes :coef-standard-errors)
(1.056688 0.06747931)
>
```

The `:plot-residuals` message will produce a residual plot. To find out what residuals are plotted against let's look at the help information:

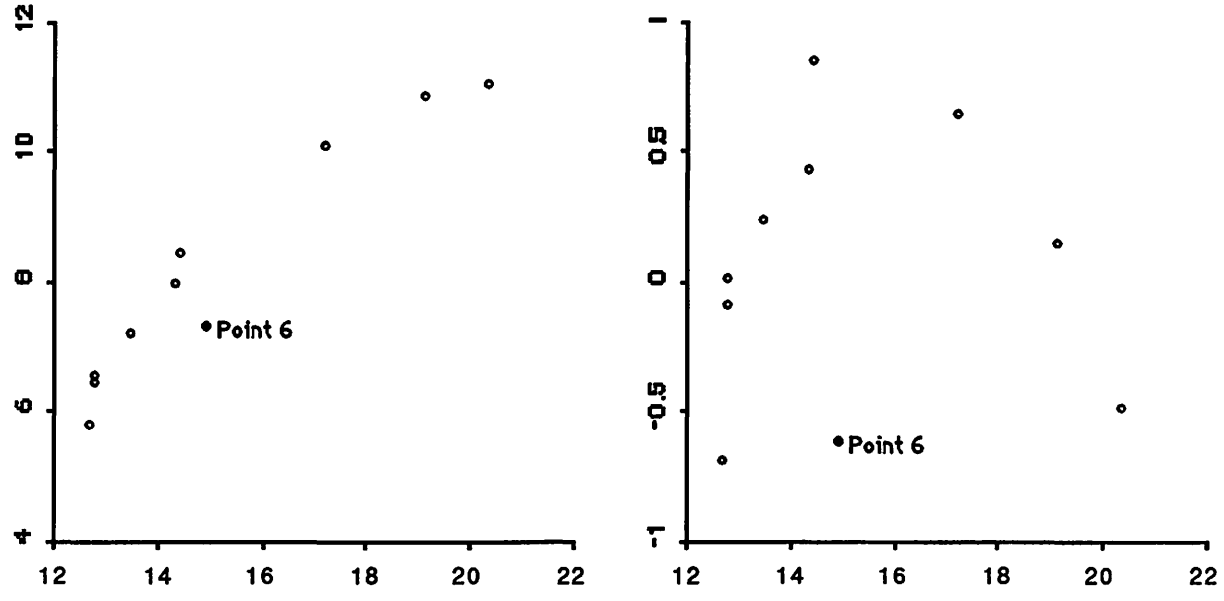
```
> (bikes :help :plot-residuals)

Message args: (&optional x-values)
Opens a window with a plot of the residuals. If X-VALUES are not
supplied the fitted values are used. The plot can be linked to
other plots with the link-views function. Returns a plot object.
NIL
```

Using the expressions

```
(plot-points travel-space separation)
(bikes :plot-residuals travel-space)
```

we can construct two plots of the data. By linking the plots we can use the mouse to identify points in both plots simultaneously. A point that stands out is observation 6 (starting the count at 0, as usual):



The plots both suggest that there is some curvature in the data; this curvature is particularly pronounced in the residual plot if you ignore observation 6 for the moment. To allow for this curvature we might try to fit a model with a quadratic term in `travel-space`:

```
> (def bikes2
1>      (regression-model (list travel-space (^ travel-space 2))
2>                        separation))

Least Squares Estimates:

Constant                -16.41924    (7.848271)
Variable  0:            2.432667     (0.9719628)
Variable  1:           -0.05339121    (0.02922567)

R Squared:              0.9477923
Sigma hat:              0.5120859
Number of cases:               10
Degrees of freedom:             7

#<Object: #9>
>
```

I have used the exponentiation function "^" to compute the square of `travel-space`. Since I am now forming a multiple regression model the first argument to `regression-model` is a list of the x variables.

You can proceed in many directions from this point. If you want to calculate Cook's distances for the observations you can first compute internally studentized residuals as

```
(def studres (/ (bikes2 :residuals)
                (* (bikes2 :sigma-hat)
                   (sqrt (- 1 (bikes2 :leverages))))))
```

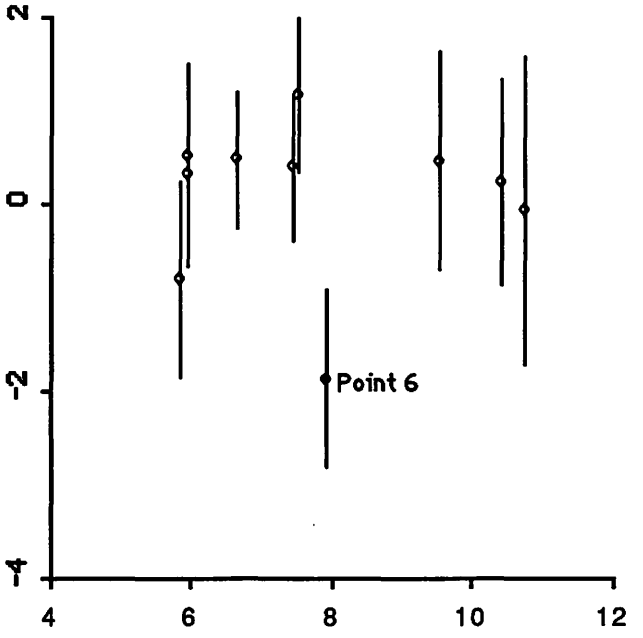
Then Cooks distances are obtained as[1]

```
> (* (^ studres 2) (/ (bikes2 :leverages) (- 1 (bikes2
:leverages)) 3))
(0.166673 0.00918596 0.03026801 0.01109897 0.009584418 0.1206654
0.581929 0.0460179 0.006404474 0.09400811)
```

The seventh entry - observation 6, counting from zero - clearly stands out.

Another approach to examining residuals for possible outliers is to use the Bayesian residual plot proposed by Chaloner and Brant (1987), which can be obtained using the message `:plot-bayes-residuals`. The expression `(bikes2 :plot-bayes-residuals)` produces the plot

[1]The / function is used here with three arguments. The first argument is divided by the second, and the result is then divided by the third. Thus the result of the expression (/ 6 3 2) is 1.



The bars represent mean ± 2SD of the posterior distribution of the actual realized errors, based on an improper uniform prior distribution on the regression coefficients. The y axis is in units of sigma-hat. Thus this plot suggests the probability that point 6 is three or more standard deviations from the mean is about 3%; the probability that it is at least two standard deviations from the mean is around 50%.

Exercises

1) Using the variables `absorption`, `iron` and `aluminum` introduced in Section 6.1 above construct and examine a model with `absorption` as the dependent variable.

2) Using the variables `abrasion-loss`, `tensile-strength` and `hardness` introduced in Section 6.2 above construct and examine a model with `abrasion-loss` as the dependent variable.

8. Defining Your Own Functions

You can use the XLISP language to define functions of your own. Many of the functions you have been using so far are written in this language. The special form used for defining functions is called `defun`. The simplest form of the defun syntax is

```
(defun fun args expression)
```

where fun is the symbol you want to use as the function name, args is the list of the symbols you want to use as arguments and expression is the body of the function. Suppose for example that you want to define a function to delete a case from a list. This function should take as its arguments the list and the index of the case you want to delete. The body of the function can be based on either of the two approaches described in Section 5.3 above. Here is one approach:

```
(defun delete-case (x i)
  (select x
          (remove i (iseq 0 (- (length x) 1))) ) )
```

I have used the function `length` in this definition to determine the length of the argument x. Note that none of the arguments to defun are quoted: defun is a special form that does not evaluate its arguments.

Unless the functions you define are very simple you will probably want to define them in a file and load the file into XLISP-STAT with the `load` command. You can put the functions in the statinit.lsp file or include in statinit.lsp a load command that will load another file. The version of XLISP-STAT for the Macintosh includes a simple editor that can be used from within XLISP-STAT. The editor is described briefly in Section 4.5 above.

You can also write functions that send messages to objects. Here is a function that takes two regression models, assumed to be nested, and computes the F statistic for comparing these models:

```
(defun f-statistic (m1 m2)
"
Args: (m1 m2)
Computes the F statistic for testing model m1 within model m2."
  (let ((ss1 (m1 :sum-of-squares))
        (df1 (m1 :df))
        (ss2 (m2 :sum-of-squares))
        (df2 (m2 :df)))
    (/ (/ (- ss1 ss2) (- df1 df2)) (/ ss2 df2))))
```

This definition uses the Lisp `let` construct to establish some local variable bindings. The string following the arguments is a documentation string. When a documentation string is present in a defun expression defun will install it so the help function will be able to retrieve it.

The discussion in this section has only scratched the surface of what you can do with functions in the XLISP language. To see more examples you can look at the files that are loaded when XLISP-STAT starts up. The first file to be loaded is init.lisp, and it forces several other files to be loaded as well. For more information on options of function definition, macros, etc see the XLISP documentation and the books on LISP mentioned in the references.

9. Adding Your Own Regression Methods

XLISP allows you to add your own methods to an existing class of objects and to define new classes of objects as refinements of existing ones. In this section I will only give some simple examples to show what is possible. For more details see Betz (1986) and look over the files regression.lsp and newplot.lsp.

In Section 8 we calculated Cook's distances for a regression model. If you find that you are doing this very frequently then you might want to define a `:cooks-distances` method. The general form of a method definition is

```
(object-class :answer :new-method arg-list body-list)
```

`object-class` is the class your objects belong to. In the case of regression models this class is called `regression-model-class`. The argument `:new-method` is the message you want to send to invoke your new method; in our case this would be `:cooks-distances`. The argument `arg-list` is the list of argument names for your method, and `body-list` is a list of expressions. When the message is received each of these expressions will be evaluated, in the order in which they appear. Note that the expression used to define a new method in fact sends a message to an object - a class object. As a result, its arguments will be evaluated. Thus you will usually have to quote `arg-list` and `body-list`.

Here is an expression that will install the `:cooks-distances` method:

```
(regression-model-class :answer :cooks-distances '() '(
"
Message args: ()
Returns Cooks distances for the model."
  (let* ((leverages (self :leverages))
         (studres (/ (self :residuals)
                     (* (self :sigma-hat)
                        (sqrt (- 1 leverages))))))
    (* (^ studres 2)
       (/ leverages (- 1 leverages) (self :num-coefs))))))
```

I have used a `let*` construct to define two local variables, `studres` and `leverages`. The `let*` construct differs from the `let` construct in that it assigns the variables sequentially rather than in parallel. Thus the definition of the second variable, `studres`, can use the first variable, `leverages`. The symbol `self` refers to the object that is receiving the message.

10. Matrices and Arrays

XLISP-STAT includes support for multidimensional arrays patterned after the Common Lisp standard described in detail in Steele (1984). The functions supported are listed in the appendix.

In addition to the standard Common Lisp array functions XLISP-STAT also includes a number of linear algebra functions such as `inverse`, `solve` and `transpose`. These functions are listed in the appendix as well.

At present XLISP-STAT does not provide any facility for nice printing of matrices and other arrays. An array is printed using the standard Common Lisp format. For example, a 2 by 3 matrix with rows (1 2 3) and (4 5 6) is printed as

```
#2A((1 2 3)(4 5 6))
```

The select function can be used to extract elements or sub arrays from an array. If `A` is a two dimensional array then the expression

```
(select a 0 1)
```

will return the element 1 of the row 0 of `A`. The expression

```
(select a (list 0 1) (list 0 1))
```

returns the upper left hand corner of `A`.

11. Reading Data Files

The data files we have used so far in this tutorial have been processed to contain XLISP-STAT expressions. XLISP-STAT also provides two functions for reading raw data files. The simpler of the two is `read-data-file`. The expression

```
(read-data-file file)
```

where `file` is a string representing the name of the data file, returns a list of all the items in the file. Items can be separated by any amount of white space, but not by commas or other punctuation marks. Items can be any valid Lisp expressions. In particular they can be numbers, strings or symbols. The list can then be manipulated into the appropriate form within XLISP-STAT.

The function `read-data-columns` is provided for reading data files in which each row represents a case and each column a variable. The expression

```
(read-data-columns file cols)
```

will return a list of `cols` lists, each representing a column of the `file`. Note that this function determines the column structure from the value of `cols`, not from the structure of the file. The entries of file can be as for `read-data-file`.

These two functions should be adequate for most purposes. If you have to read a file that does not fit into the form considered here you can use the raw file handling functions of XLISP

12. Nonlinear Regression

XLISP-STAT allows the construction of nonlinear, normal regression models. The present implementation is experimental (even more experimental than the rest of the program). The definitions needed for nonlinear regression are in the file `nonlin.lsp` on the distribution disk. This file is not loaded automatically at start up; you should load it now, using the **Load** item on the **File** menu or the `load` command, to carry out the calculations in this section.

As an example, suppose we have data sets x and y given by

```
(def x (list 2.000 2.000 0.667 0.667 0.400 0.400
             0.286 0.286 0.222 0.222 0.200 0.200))
(def y (list 0.0615 0.0527 0.0334 0.0258 0.0138 0.0258
             0.0129 0.0183 0.0083 0.0169 0.0129 0.0087))
```

and we would like to fit a normal nonlinear regression model with the Michaelis-Menten mean function

$$\eta(x) = \frac{\theta_1 x}{\theta_2 + x}$$

to this data. The function f defined by

```
(defun f (b) (/ (* (select b 0) x) (+ (select b 1) x)))
```

will compute the list of mean response values at the points in x for a parameter list b. Using these definitions, which are contained in the file `mikement.lsp` in the **Data** folder of the distribution disk, we can construct a nonlinear regression model using the `nreg-model` function:[1]

```
> (def mikement (nreg-model #'f y (list 1 1)))

Least Squares Estimates:

Parameter  0:           0.1056427   (0.01760005)
Parameter  1:             1.70269   (0.4757766)

R Squared:               0.9379065
Sigma hat:              0.004483935
Number of cases:                12
Degrees of freedom:             10

#<Object: #6>
>
```

The function `nreg-model` requires three arguments, a mean function, a list or vector of observed response values, and an initial guess for the parameter vector. In addition it can take two keyword arguments, `:epsilon` specifies a convergence criterion and `:count-limit` a limit on

[1]The expression `#'f` is short for `(function f)` and is the standard Lisp method for obtaining the function definition associated with the symbol f. In XLISP it is not really needed since XLISP symbols do not have distinct function and value cells.

the number of iterations. By default these values are .001 and 20, respectively. A simple, unmodified Gauss-Newton algorithm based on numerical derivatives is used for fitting the model.

To see how you can analyze the model further you can send `mikement` the `:help` message. The result is very similar to the help information for a linear regression model. The reason for this is simple. Nonlinear regression models have been implemented as model objects. The class structure in XLISP allows new classes to be defined as subclasses of old classes. The new class *inherits* all the methods defined for the old class. If some of these methods don't apply directly to the new class they can be redefined, and new methods that only make sense for the new class can be added. I have defined nreg-model-class, the class of nonlinear regression models, as a subclass of the linear regression model class. The internal data, the method for computing estimates and the method of computing fitted values have been modified, but the other methods remain unchanged. Once the model has been fit the Jacobian of the mean function at the estimated parameter values is used as the X matrix. In terms of this definition most of the methods for linear regression, the methods `:coef-standard-errors` and `:leverages` for example, still make sense, at least as first order asymptotic approximations.

13. One Way ANOVA

XLISP-STAT allows the construction of normal one way analysis of variance models. The definitions needed are in the file `oneway.lsp` on the distribution disk. This file is not loaded automatically at start up; you should load it now, using the **Load** item on the **File** menu or the `load` command, to carry out the calculations in this section.

As an example, suppose we would like to model the data on cholesterol level in rural and urban Guatemalans, examined in Section 3.2, as a one way ANOVA model. The boxplots we obtained in Section 3.2 showed that the samples were skewed and the center and spread of the urban sample were larger than the center and spread of the rural sample. To compensate for these facts I will use a normal ANOVA model for the logarithms of the data:

```
> (def cholesterol (oneway-model (list (log urban) (log rural))))

Least Squares Estimates:

Group  0:                5.377172    (0.03624821)
Group  1:                5.099592    (0.03456131)

R Squared:              0.4343646
Sigma hat:              0.1621069
Number of cases:               42
Degrees of freedom:            40

Group Mean Square:      0.8071994    (1)
Error Mean Square:     0.02627865    (40)

#<Object: #6>
```

The function `oneway-model` requires one argument, a list of the lists or vectors representing the samples for the different groups. The model cholesterol can respond to all regression messages as well as a few new ones. The new ones are

```
:BOXPLOTS
:ERROR-MEAN-SQUARE
:ERROR-DF
:GROUP-MEAN-SQUARE
:GROUP-DF
:STANDARD-DEVIATIONS
:GROUPED-DATA
```

References

Becker, Richard A., and Chambers, John M., (1984), "S: an interactive environment for data analysis and graphics," Wadsworth, Belmont, Ca.

Becker, Richard A., and William S. Cleveland, (1987), "Brushing Scatterplots", *Technometrics*, vol. 29, pp. 127-142.

Betz, David, (1985) "An XLISP Tutorial", *BYTE*, pp 221.

Betz, David, (1986), "XLISP: An experimental object-oriented programming language," Reference manual for Version 1.6.

Chaloner, Kathryn, and Brant, Rollin, (1987) "A Bayesian approach to outlier detection and residual analysis", Technical Report, School of Statistics, University of Minnesota.

Devore, J. and Peck, R., (1986), "Statistics, the exploration and analysis of data", West Publishing Co., St. Paul, Mn.

McDonald, J. A., (1982), "Interactive Graphics for Data Analysis," unpublished Ph. D. thesis, Stanford University, Department of Statistics.

Oehlert, Gary W., (1987), "MacAnova User's Guide," Technical Report 493, University of Minnesota.

Steele, Guy L., (1984), "Common Lisp: The Language", Digital Press.

Stuetzle, W., (1987), "Plot windows," *J. Amer. Statist. Assoc.*, vol. 82, pp. 466 - 475.

Weisberg, Sanford, (1982), "MULTREG Users Manual," Technical Report 298, University of Minnesota.

Winston, Patrick H. and Berthold K. P. Horn, (1984), "LISP", Addison-Wesley, New York.

Appendix: Selected Listing of XLISP-STAT Functions

A 1. Arithmetic and Logical Functions

*
**
+
-
/
/=
1+
1-
<
<=
=
>
>=
^
ABS
ACOS
ASIN
ATAN
CEILING
COS
EVENP
EXP
EXPT
FLOAT
FLOOR
LOG
LOG-GAMMA
MAX
MIN
MINUSP
ODDP
PLUSP
PROD
R-AND
R-NOT
R-OR
RANDOM
REM
ROUND
SIN
SQRT
TAN
TRUNCATE
ZEROP

A 2. Constructing and Modifying Compound Data and Variables

DEF
ISEQ
LIST
REPEAT
RSEQ
SELECT
UNDEF
VARIABLES
VECTOR
WHICH

A 3. Basic Statistical Functions

BETA-CDF
CAUCHY-CDF
CAUCHY-QUANT
CAUCHY-RAND
CHISQ-CDF
COVARIANCE-MATRIX
DIFFERENCE
F-CDF
FIVNUM
GAMMA-CDF
INTERQUARTILE-RANGE
MEAN
MEDIAN
NORMAL-CDF
NORMAL-QUANT
NORMAL-RAND
NREG-MODEL
QUANTILE
READ-DATA-COLUMNS
READ-DATA-FILE
REGRESSION-COEFFICIENTS
REGRESSION-MODEL
ROWS-TO-COLUMNS
SORT
STANDARD-DEVIATION
T-CDF

A 4. Plotting Functions

BOXPLOT
BOXPLOT-X
HISTOGRAM
LINK-VIEWS

NAME-LIST
PLOT-FUNCTION
PLOT-LINES
PLOT-POINTS
PROBABILITY-PLOT
QUANTILE-PLOT
SCATTERPLOT-MATRIX
SPIN-PLOT
UNLINK-VIEWS

A 5. Some Useful Array and Linear Algebra Functions

%*
ARRAY
ARRAY-DIMENSION
ARRAY-DIMENSIONS
ARRAY-IN-BOUNDS-P
ARRAY-RANK
ARRAY-ROW-MAJOR-INDEX
ARRAY-TOTAL-SIZE
ARRAYP
BIND-COLUMNS
BIND-ROWS
COLUMN-LIST
COPY-ARRAY
COPY-LIST
COPY-VECTOR
CROSS-PRODUCT
DETERMINANT
DIAGONAL
IDENTITY-MATRIX
INNER-PRODUCT
INVERSE
MAKE-LIST
MAKE-SWEEP-MATRIX
MAP-ELEMENTS
MAPARRAY
MATMULT
MATRIX
MATRIXP
OUTER-PRODUCT
ROW-LIST
SIZE
SOLVE
SPLIT-LIST
SUBARRAY
SUBLIST
SUM
SWEEP-OPERATOR
TRANSPOSE
VECTORP

A 6. System Functions

EXIT
HELP
HELP*
LOAD
SAVE
VARIABLES

A 7. Some Basic Lisp and XLISP Functions and Special Forms

Except where noted these functions should have a significant subset of their Common Lisp functionality as defined in Steele (1984).

ALLOC	(XLISP specific)	CONS	
AND		CONSP	
APPEND		DEBUG	(XLISP specific)
APPLY	(differs from Common Lisp)	DEFMACRO	
APROPOS		DEFUN	
APROPOS-LIST		DO	
AREF	(XLISP function modified)	DO*	
ASSOC		DOLIST	
ATOM		DOTIMES	
BOUNDP		ELT	
C[AD]R	up to four A's or D's are supported	EQ	
CAR		EQL	
CASE		EQUAL	
CDR		EXPAND	
CLOSE	(differs from Common Lisp)	FILEP	
COERCE	(differs from Common Lisp)	FIRST	
COND		FORMAT	

FUNCALL
FUNCTION
GC (XLISP specific)
GETF
GET-MACRO-CHARACTER
GO
IDENTITY
IF
LAMBDA
LAST
LENGTH
LET
LET*
LIST
LISTP
MAKE-ARRAY (XLISP function modified)
MAP
MAPC
MAPCAN
MAPCAR
MAPCON
MAPL
MAPLIST
MEM (XLISP specific)
MEMBER
NODEBUG (XLISP specific)
NOT
NTH
NTHCDR
NULL
NUMBERP
OBJECTP (XLISP specific)
OPENI (XLISP specific)
OPENO (XLISP specific)
OR
PRIN1
PRINC
PRINT
PROG
PROG*
PROG1
PROG2
PROGN
PROVIDE
QUOTE
READ (differs from Common Lisp)
REDUCE
REMOVE
REMOVE-IF
REMOVE-IF-NOT
REQUIRE
REST
RETURN
REVERSE
SECOND
SET
SET-MACRO-CHARACTER
SETF
SETQ
STRCAT (XLISP specific)
STRING (differs from Common Lisp)
STRINGP
SUBLIS
SUBST
SYMBOL-NAME
SYMBOL-PLIST
SYMBOL-VALUE
SYMBOLP
TERPRI
TIME
TYPE-OF (differs from Common Lisp)
UNLESS
UNWIND-PROTECT
WHEN
Y-OR-N-P

Index

XLISP-STAT
Quick Reference Card
(Apple Macintosh Version 1.0)

Expressions typed to the command window are evaluated. Numbers and strings evaluate to themselves. Lists are interpreted as function applications. For example,

(+ 1 2)

means apply the function + to the arguments 1 and 2.

Notation and Terminology

In the argument lists for the basic functions described below

[name]
means NAME is an optional argument,

[:print logical]
means print is an optional keyword argument.

A *sequence* is a list or a vector. *Compound-data* is a list or an array.

Special Characters

'

Single quote. Items following a single quote are not evaluated. 'x is short for (quote x).

;

Comment character. Text on a line following a semicolon is ignored.

Arithmetic Functions

(+ x ...)	addition
(- x ...)	subtracion

With one argument, negates it (elementwise)

(* x ...)	multiplication
(/ x ...)	division

With one argument, returns reciprocal(s).

(expt x y)	exponentiation
(** x y)	exponentiation
(^ x y)	exponentiation

raises x to the power y (element-wise).

(abs x)	absolute value
(exp x)	exponential

e raised to the power x (element-wise)

(log x)	natural logarithm
(sqrt x)	square root

(min x ...)	minimum
(max x ...)	maximum

minimum or maximum of all elements.

(sum x ...)	sum of elements
(prod x ...)	product of elements

(random x)
If X is an integer generates a uniformly distributed pseudo-random integer between zero (inclusive) and X (exclusive). If X is a list or an array, returns the list or array of the results of applying RANDOM to each element of X.

Relational Functions

(= x y)	equal?
(/= x y)	not equal?
(< x y)	increasing?
(<= x y)	nondecreasing?
(> x y)	decreasing?
(>= x y)	nonincreasing?

If x and y are numbers returns T if its arguments are numerically equal, numerically not equal, or in the specified order, respectively; otherwise NIL is returned. If one or both arguments is compound, returns the results of the elementwise comparison.

Defining and Constructing Variables

(def var form)
VAR is not evaluated and must be a symbol. Assigns the value of FORM to VAR and adds VAR to the list of def'ed variables.

(iseq n m)
List of consecutive integers from n to m.

(list ...)
Returns a list of its arguments

(repeat vals times)
Repeats VALS as specified by TIMES.
Examples:

(repeat 2 5)	returns (2 2 2 2 2)
(repeat '(1 2) 3)	returns (1 2 1 2 1 2)
(repeat '(4 5 6) '(1 2 3))	returns (4 5 5 6 6 6)

(rseq a b num)
List of NUM equally spaced points starting at A and ending at B.

(select x i ...)
X can be a sequence or an array. If X is a list and I is a single number then the appropriate element of X is returned. If X is a list and I is a list of numbers then the sublist of the corresponding elements is returned. SELECT can be used in setf.

(setf form val)
Modifies lists and arrays. FORM can be a *select* expression and VAL is the new result this expression should return.

(which x)
List of the indices where X is not NIL.

Probability Distributions

Cumulative Distribution Functions:
(beta-cdf x alpha beta)
(cauchy-cdf x)
(chisq-cdf x df)
(f-cdf x numerator-df denominator-df)
(gamma-cdf x alpha)
(normal-cdf x)
(t-cdf x df)
X is a number or a sequence of numbers. Additional parameters should either be single numbers or sequences the same length as X.

Quantile Functions:
(cauchy-quant p)
(normal-quant p)
P is a number between 0 and 1 or a sequence or array of such numbers.

Random Number Generators:

(cauchy-rand n)
(normal-rand n)
(uniform-rand n)

Generate a list of N numbers from the distribution.

Summary Statistics

(interquartile-range x)
(mean x)
(median x)

(quantile x p)

X is a list or array of numbers and P a number between 0 and 1 or a sequence of such numbers. Returns the p-th quantile(s) of X

(standard-deviation x)

Sorting Functions

(order x)
(rank x)
(sort x)

Plotting Functions

(boxplot data [:title string])

DATA is a sequence, a list of sequences or a matrix.

(histogram data [:title string])

DATA is a sequence.

(name-list names [:title string])

NAMES is a number or a list of strings.

(plot-lines x y [:title string [:axis-labels strings]])

X and Y are sequences of equal length.

(plot-points x y [:title string
[:axis-labels strings
[:case-labels strings]]])

X and Y are sequences of equal length.

(scatterplot-matrix data [:title string
[:variable-labels strings
[:case-labels strings
[:scale logical]]]])

DATA is a list of two or more sequences of equal length.

(spin-plot data [:title string
[:variable-labels strings
[:case-labels strings
[:scale logical]]]])

DATA is a list of three sequences of equal length.

Statistical Models

(nreg-model mean-function y
[:epsilon number
[:count-limit number
[:print logical]]])

Nonlinear regression model with normal errors. MEAN-FUNCTION takes parameter sequence and returns the mean sequence. Y is the observed response and THETA is an initial guess for the parameter sequence.

(oneway-model data [:print logical])

Analysis of variance model with normal errors. DATA is a list of sequences.

(regression-model x y [:intercept logical
[:print logical
[:weights sequence]]])

Linear regression model with normal errors. For a simple regression X is a sequence of numbers. For a multiple regression X is a list of the independent variables or the X matrix. The sequence Y is the dependent variable. If WEIGHTS is supplied it should be a sequence the same length as Y; error variances are assumed to be inversely proportional to WEIGHTS. Returns a regression model object. To examine the model further assign the result to a variable and send it messages.
Example:
(data are in file absorbtion.lsp in the **Data** folder)

```
> (def m (regression-model (list iron aluminum)
2>                         absorbtion))
> (m :help)
> (m :plot-residuals)
```

Objects

Plotting functions and model functions return *objects*. To examine these further assign them to a symbol and send them messages. To find out what messages an object can respond to send it the help message. For example, if myplot is a plot object use the expression (myplot :help).

Other Functions and Special Forms

(defun name lambda-list [doc] form ...)

Defines a function as the global definition of the symbol NAME.

(map-elements function data ...)

FUNCTION must take as many arguments as there are DATA arguments supplied. DATA arguments must either all be sequences or all be arrays of the same shape. Returns the result of applying FUNCTION elementwise to the DATA arguments.

System Functions

(dribble [string])

With an argument, dribble causes all further input and output to be saved in the file named STRING. Without an argument, stops dribbling.

(exit)

Quits from the program. You can also exit by selecting **Quit** from the **File** menu.

(help [symbol])

Prints the documentation associated with SYMBOL.

(help* string)

Prints the documentation associated with those symbols whose print names contain STRING as a substring. STRING may be a symbol, in which case the print-name of that symbol is used.

(load filename [verbose-flag [print-flag]])

Loads the file named by FILENAME into XLISP. You can also use **Load** on the**File** menu.

(save vars file-name-root)

VARS is a symbol or a list of symbols. FILE-NAME-ROOT is a string (or a symbol whose print name is used) not ending in .lsp. The VARS and their current values are written to the file FILE-NAME-ROOT.lsp in a form suitable for use with the load command.