# XLISP-STAT

A Statistical Environment Based on
the XLISP Language

(Version 2.0)

by
Luke Tierney

University of Minnesota
School of Statistics
Technical Report Number 528

July 1989

# 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. It is written primarily for the Apple Macintosh version, but most of the material applies to other versions as well; some points where other versions differ are outlined in an appendix. The first three sections contain the information you will need to do elementary statistical calculations and plotting. The fourth section introduces some additional methods for generating and modifying data. The fifth section describes some additional features of the Macintosh user interface that may be helpful. The remaining sections deal with more advanced topics, such as interactive plots, regression models, 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, documentation for many of the commands that are available is given in the appendix. Brief help messages for these and other commands are also available through the interactive help facility described in Section 5.1 below.

XLISP itself is a high-level programming language developed by David Betz and made available for unrestricted, non-commercial use. 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 have been modified 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 [22]. A book on XLISP itself has recently been published. Unfortunately it is based on XLISP 1.7, which differs significantly from XLISP 2.0, the basis of XLISP-STAT 2.0.

XLISP-STAT was originally developed for the Apple Macintosh. It is now also available for UNIX systems using the *X11* window system, for *Sun* workstations under the *SunView* window system, and, with only rudimentary graphics, for generic 4.[23]BSD UNIX systems. 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. Some of the linear algebra and probability functions are based on code given in Press, Flannery, Teukolsky and Vetterling [14]. Regression computations are carried out using the sweep algorithm as described in Weisberg [21].

This tutorial has borrowed several ideas from Gary Oehlert's *MacAnova User's Guide* [13]. 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 [11]. Many of the functions added to XLISP-STAT were motivated by similar functions in the *S* statistical environment [2, 3].

The present version of XLISP-STAT, Version 2.0, seems to run fairly comfortably on a Mac II or Mac Plus with 2MB of memory, but is a bit cramped with only 1MB. It will not run in less than 1Mb of memory. 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 1MB Mac it may also bomb with an 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 grants of an Apple Macintosh Plus computer and hard disk and a Macintosh II computer from the MinneMac Project at the University of Minnesota, by a single quarter leave granted to the author by the University of Minnesota, by grant DMS-8705646 from the National Science Foundation, and by a research contract with Bell Communications Research.

## 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. [1]

## 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 density 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 [4] and the papers reproduced in Cleveland and McGill[8]. 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. Several commercial packages for microcomputers now contain some form of dynamic graphics, but most do not allow users to customize their plots or develop functions for producing specialized plots, such as dynamic residual plots. XLISP-STAT provides at least a partial solution to these problems. It allows the user to modify a scatter plot with Lisp functions and provides means for modifying the way in which a plot responds to mouse actions. It is also possible to add functions written in C to the program. On the Macintosh this has to be done by adding to the source code. On some unix systems it is also possible to compile and dynamically load code written in C or FORTRAN.

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 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

[1] On a UNIX system you can use the function `load-data` to load the tutorial data. After starting up XLISP-STAT enter the expression `(load-data "tutorial")`, followed by a *return*.

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 made available for unrestricted, non-commercial use 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. 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 *X11*, *Sun*, 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, for example by supplying a disk or tape and a stamped mailing envelope. You can also obtain the source code and Macintosh executables by anonymous *ftp* over the internet from *umnstat.stat.umn.edu* (128.101.51.1).

A version for the Mac II requiring the MC68881 coprocessor is available as well. This version is compiled to access the coprocessor directly and will therefore not run on a machine without the coprocessor. Numerical computations with this version are about 7 to 10 times faster on a Mac II than without the direct coprocessor access.

## 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.

XLISP-STAT is an evolving system. Over time new features will be introduced, and existing features that do not work may be changed. Every effort will be made to keep XLISP-STAT consistent with the information in this tutorial, but if this is not possible the help information should give accurate information about the current use of a commend.

# 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 [2]:

**init.lsp**
**common.lsp**
**help.lsp**
**objects.lsp**
**menus.lsp**
**statistics.lsp**
**dialogs.lsp**
**graphics.lsp**
**graphics2.lsp**
**regression.lsp**
**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 the text in its command window will look something like this[3]:

```
XLISP version 2.0, Copyright (c) 1988, by David Betz
XLISP-STAT version 2.0 , Copyright (c) 1988, by Luke Tierney.
Several files will be loaded; this may take a few minutes.


; loading "init.lsp"
; loading "common.lsp"
; loading "help.lsp"
; loading "objects.lsp"
; loading "menus.lsp"
; loading "statistics.lsp"
; loading "dialogs.lsp"
; loading "graphics.lsp"
; loading "graphics2.lsp"
; loading "regression.lsp"
>
```

The final “>” in the window is the XLISP-STAT prompt. 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 give you a new prompt:

```
> 1
1
>
```

[2]The file `xlisp.help` is optional. It may be replaced by a reduced file `xlisp.help.small` or it may be omitted entirely. If it is not present interactive help will not be available.

[3]On a Macintosh with limited memory a dialog warning about memory restrictions may be appear at this point. On a Mac II it takes about a minute to load these files; on a Mac Plus or an SE it takes about 3.5 minutes.

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)
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. In later sections I will omit the new prompt following the result in order to save space.

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 *listener* and *evaluator* work.

## 2.1 Data

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

```
1                   ; an integer
-3.14               ; a floating point number
#C(0 1)             ; a complex number (the imaginary unit)
```

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 9.

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

## 2.2 The Listener and the Evaluator

A session with XLISP-STAT basically consists of a conversation between you and the *listener*. The listener is the window into which you type your commands. When it is ready to receive a command it gives you a prompt. At the prompt you can type in an expression. You can use the mouse or the *backspace* key to correct any mistakes you make while typing in your expression. When the expression is complete and you type a *return* the listener passes the expression on to the *evaluator*. The evaluator evaluates the expression and returns the result to the listener for printing. [4] The evaluator is the heart of the system.

[4] It is possible to make a finer distinction. The *reader* takes a string of characters from the listener and converts it into an expression. The *evaluator* evaluates the expression and the *printer* converts the result into another string of characters for the listener to print. For simplicity I will use *evaluator* to describe the combination of these functions.

The basic rule to remember in trying to understand how the evaluator works 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 listener. 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. 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 the previous 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 *not* to evaluate something. For example, suppose we wanted to get the evaluator to simply return 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: Its 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, a single quote before the expression you want to quote:

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

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)`
6. `'x`

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 available in XLISP-STAT.

## 3.1 First Steps

Statistical data usually consists of groups of numbers. Devore and Peck [11, Exercise 2.11] describe 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 [5]:

```
> (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))
PURCHASES
>
```

Now the symbol `purchases` has a value associated with it: Its value is our list of 22 numbers. If you give the symbol `purchases` to the evaluator then it will find the value of this symbol and return 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)
>
```

[5] `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 tutorial. `def` is closely related to the standard Lisp special forms `setf` and `setq`. The advantage of using `def` is that it adds your variable name to a list of `def`'ed variables that you can retrieve using the function `variables`. If you use `setf` or `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.

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))`
4. `(+ (list 1 2 3) (list 4 5))`

## 3.2 Summary Statistics and Plots

Devore and Peck [11, page 54, Table 10] give 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
       (list .77 1.74 .81 1.20 1.95 1.20 .47 1.43 3.37 2.20 3.30
             3.09 1.51 2.10 .52 1.62 1.31 .32 .59 .81 2.81 1.87
             1.18 1.35 4.75 2.48 .96 1.89 .90 2.05))
PRECIPITATION
>
```

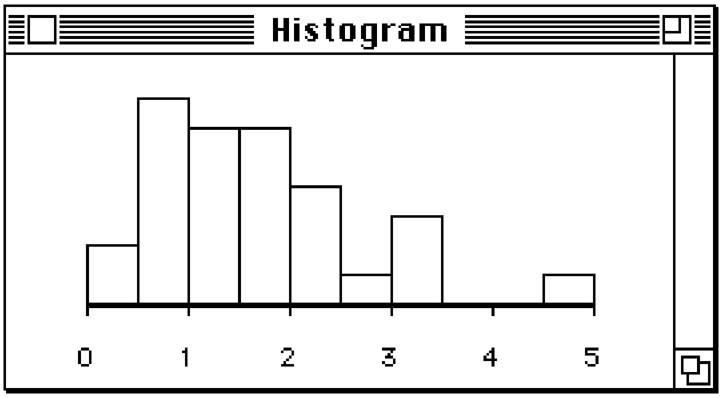


Figure 1: Histogram of precipitation levels.

In typing this expression I have hit the *return* and *tab* keys a few times in order to make the typed expression easier to read. The tab key indents the next line to a reasonable point to make the expression more readable.

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

```
> (histogram precipitation)
#<Object: 3564170, prototype = HISTOGRAM-PROTO>
> (boxplot precipitation)
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

Each of these commands should cause a window with the appropriate graph to appear on your screen. The windows should look something like Figures 1 and 2.

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 listener window. You will have to click on the listener window between the two commands shown here.

The two functions return results that are printed something like this:

```
#<Object: 3564170, prototype = HISTOGRAM-PROTO>
```

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 listener window so you can rearrange the plots more easily. You can do this by clicking in the listener window's close box. You can later re-open the listener 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
>
```

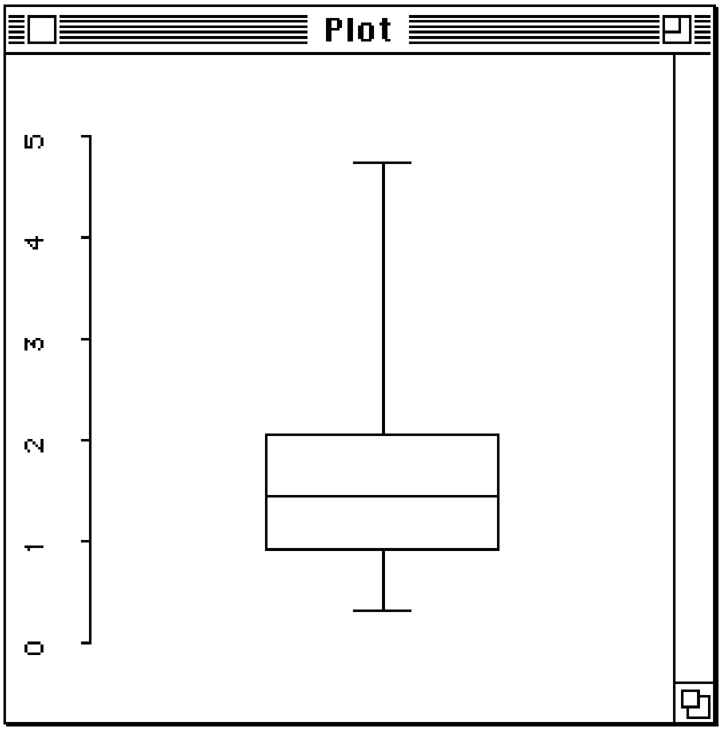


Figure 2: Boxplot of precipitation levels.

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 (Devore and Peck [11, page 19, Example 3]):

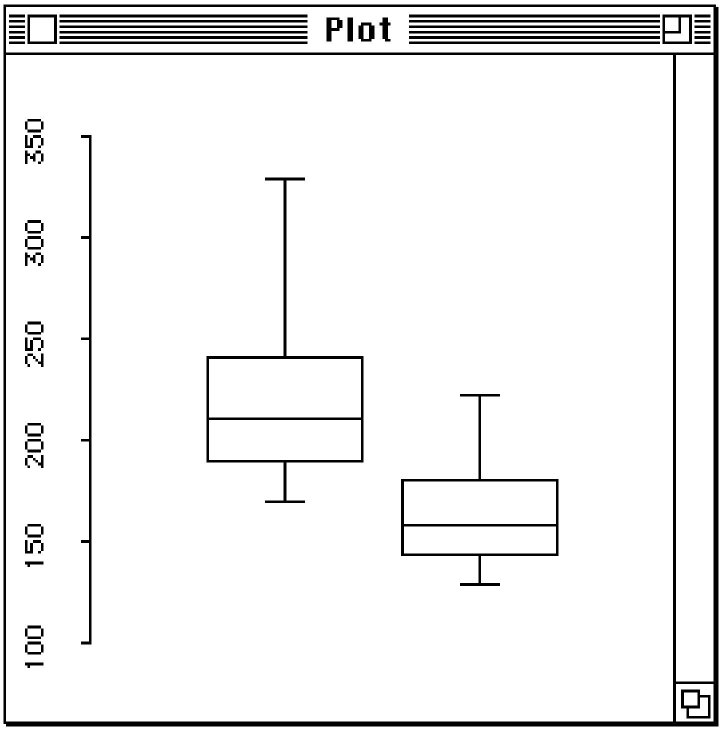


Figure 3: Parallel box plots of cholesterol levels for urban and rural guatemalans.

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

The parallel boxplot is obtained by

```
> (boxplot (list urban rural))
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

and is shown in Figure 3; the urban group is on the left.

## Exercises

The following exercises involve examples and problems from Devore and Peck [11]. 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 5.4 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`.[6]

1. Devore and Peck [11, page 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 [11] 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: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

and the result will look like Figure 4. 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 $\pi$ 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

[6] On UNIX systems use the function `load-data`. For example, evaluating the expression `(load-data "car-prices")` should load the file `car-prices.lsp`.

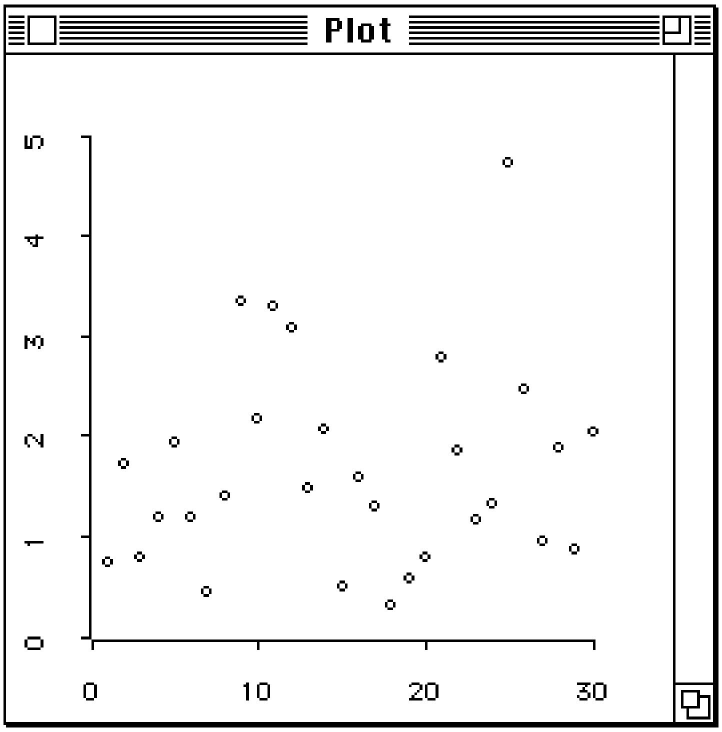


Figure 4: Scatterplot of precipitation levels against time.

```
> (plot-lines (rseq (- pi) pi 50) (sin (rseq (- pi) pi 50)))
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

The plot should look like Figure 5.

Scatterplots are of course particularly useful for examining the relationship between two numerical observations taken on the same subject. Devore and Peck [11, Exercise 2.33] 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 .37 .87
                .65 .48 .51 .47 .56 .51 .57 .36 .52 .58 .47 .65
                .41 .39 .55 .64 .38 .50 .73 .57 .41 1.02 1.10 .43
                .41 .41 .52 .70 .52 .51 .49 .61 .46 .55))
HC
> (def co (list 5.01 8.60 4.95 7.51 14.59 11.53 5.21 9.62 15.13
                3.95 4.12 19.00 11.20 3.45 4.10 4.74 5.36 5.69
                6.02 2.03 6.78 6.02 5.22 14.67 4.42 7.24 12.30
                7.98 4.10 12.10 14.97 5.04 3.38 23.53 22.92 3.81
                1.85 2.26 4.29 14.93 6.35 5.79 4.62 8.43 3.99 7.47))
CO
> (plot-points hc co)
#<Object: 3423466, prototype = SCATTERPLOT-PROTO>
>
```

The resulting plot is shown in Figure 6.

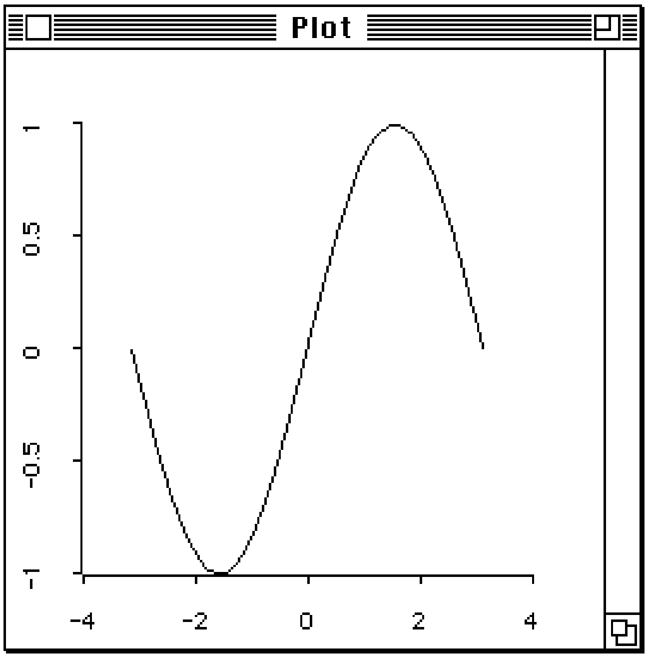


Figure 5: A plot of $\sin(x)$.

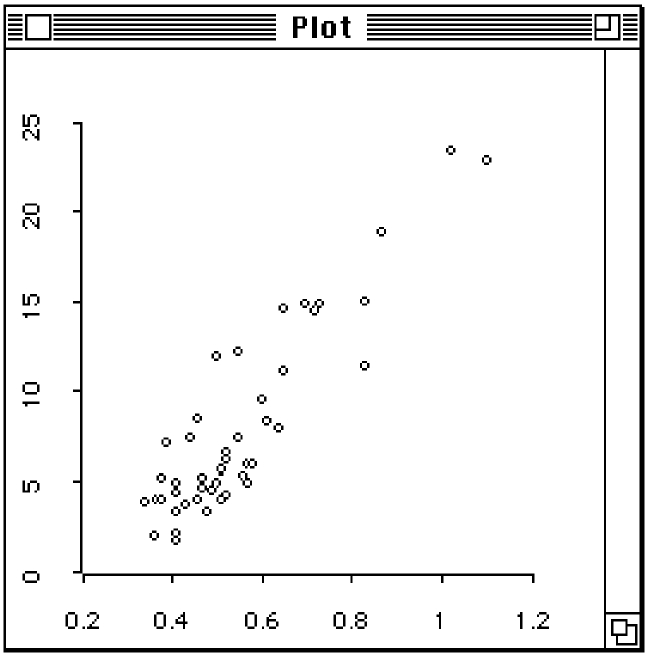


Figure 6: Plot of HC against CO.

### Exercises

1. Draw a graph of the function $f(x) = 2x + x^2$ between -2 and 3.
2. Devore and Peck [11, Exercise 4.2] give 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.

## 3.4 Plotting Functions

Plotting the sine function in the previous section was a bit cumbersome. As an alternative we can use the function `plot-function` to plot a function of one argument over a specified range. We can plot the sine function using the expression

```
(plot-function (function sin) (- pi) pi)
```

The expression `(function sin)` is needed to extract the function associated with the symbol `sin`. Just using `sin` will not work. The reason is that a symbol in Lisp can have both a *value*, perhaps set using `def`, and a *function definition* at the same time. [7] This may seem a bit cumbersome at first, but it has one great advantage: Typing an innocent expression like

```
(def list '(2 3 4))
```

will not destroy the `list` function.

Extracting a function definition from a symbol is done almost as often as quoting an expression, so again a simple shorthand notation is available. The expression

```
#'sin
```

is equivalent to the expression `(function sin)`. The short form `#'` is usually pronounced *sharp-quote*. Using this abbreviation the expression for producing the sine plot can be written as

```
(plot-function #'sin (- pi) pi).
```

[7] As an aside, a Lisp symbol can be thought of as a "thing" with four cells. These cells contain the symbol's print name, its value, its function definition, and its property list. Lisp symbols are thus much more like physical entities than variable identifiers in FORTRAN or C.

# 4 More on Generating and Modifying Data

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

## 4.1 Generating Random Data

XLISP-STAT has several functions for generating pseudo-random numbers. For example, the expression

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

will generate a list of 50 independent uniform random variables. The functions `normal-rand` and `cauchy-rand` work similarly. Other generating functions require additional arguments to specify distribution parameters. Here is a list of the functions available for dealing with probability distributions:

```
normal-cdf     normal-quant     normal-rand     normal-dens
cauchy-cdf     cauchy-quant     cauchy-rand     cauchy-dens
beta-cdf       beta-quant       beta-rand       beta-dens
gamma-cdf      gamma-quant      gamma-rand      gamma-dens
chisq-cdf      chisq-quant      chisq-rand      chisq-dens
t-cdf          t-quant          t-rand          t-dens
f-cdf          f-quant          f-rand          f-dens

binomial-cdf   binomial-quant   binomial-rand   binomial-pmf
poisson-cdf    poisson-quant    poisson-rand    poisson-pmf

bivnorm-cdf
```

More information on the required arguments is given in the appendix in Section C.3. The discrete quantile functions `binomial-quant` and `poisson-quant` return values of a left continuous inverse of the cdf. The pmf's for these distributions are only defined for integer arguments. The quantile functions and random variable generators for the beta, gamma, $\chi^2$, t and F distributions are presently calculated by inverting the cdf and may be a bit slow.

The state of the internal random number generator can be "randomly" reseeded, and the current value of the generator state can be saved. The mechanism used is the standard Common Lisp mechanism. The current random state is held in the variable `*random-state*`. The function `make-random-state` can be used to set and save the state. It takes an optional argument. If the argument is `NIL` or omitted `make-random-state` returns a copy of the current value of `*random-state*`. If the argument is a state object a copy of it is returned. If the argument is `t` a new, "randomly" initialized state object is produced and returned. [8]

## 4.2 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

[8] The generator used is Marsaglia's portable generator from the *Core Math Libraries* distributed by the National Bureau of Standards. A state object is a vector containing the state information of the generator. "Random" reseeding occurs off the system clock.

```
> (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)))
```

## 4.3 Forming Subsets and Deleting Cases

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 its 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 a single 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)
```

## 4.4 Combining Several Lists

At times you may want to combine several short lists into a single longer list. 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).
```

## 4.5 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 Section 4.3 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 labels for different items. When you assign a name to an item with the `def` command you are not producing a new item. 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 item referred to by) `x` with `setf` then we are also changing the element of (the item referred to by) `y`, since both `x` and `y` refer to the same item. 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 items and changes to `x` will not affect `y`.

# 5 Some Useful Shortcuts

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

## 5.1 Getting Help

On line help is available for many of the functions in XLISP-STAT [9]. As an example, here is how you would get help for the function `median`:

```
 > (help 'median)
MEDIAN                                                         [function-doc]
Args: (x)
Returns the median of the elements of 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)
------------------------------------------------------------------------------
Sorry, no help available on NORM
------------------------------------------------------------------------------
Sorry, no help available on NORM-2
------------------------------------------------------------------------------
Sorry, no help available on NORMAL
------------------------------------------------------------------------------
NORMAL-CDF                                                     [function-doc]
Args: (x)
Returns the value of the standard normal distribution function at X.
Vectorized.
------------------------------------------------------------------------------
NORMAL-DENS                                                    [function-doc]
Args: (x)
Returns the density at X of the standard normal distribution. Vectorized.
------------------------------------------------------------------------------
NORMAL-QUANT                                                   [function-doc]
Args (p)
Returns the P-th quantile of the standard normal distribution. Vectorized.
------------------------------------------------------------------------------
NORMAL-RAND                                                    [function-doc]
Args: (n)
Returns a list of N standard normal random numbers. Vectorized.
```

[9] The on line help file may not be available on a single disk version that includes a system file. Alternatively, there may be a small help file available that does not contain documentation for all functions.

```
-------------------------------------------------------------------------------
NIL
>
```

The symbols `norm`, `norm-2` and `normal` do not have function definitions and thus there is no help available for them. The term *vectorized* in a function's documentation means the function can be applied to arguments that are lists or arrays; the result will be a list or array of the results of applying the function to each element of its arguments. [10] A related term appearing in the documentation of some functions is *vector reducing*. A function is vector reducing if it is applied recursively to its arguments until a single number results. The functions `sum`, `prod`, `max` and `min` are vector reducing.

The first time a help function is used will take some time – the help file is scanned and the positions of all entries in the file are recorded. Subsequent calls will be faster.

Let me briefly explain the notation used in the information printed by `help` to describe the arguments a function expects [11]. 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 its default value is `T`. No explicit default value is specified for `y`; its 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 [12] 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.

In addition to providing information about functions `help` also gives information about data types and certain variables. For example,

[10] This process of applying a function elementwise to its arguments is called *mapping*.

[11] The notation used corresponds to the specification of the argument lists in Lisp function definitions. See Section 8 for more information on defining functions.

[12] 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.

```
> (help 'complex)
COMPLEX                                                        [function-doc]
Args: (realpart &optional (imagpart 0))
Returns a complex number with the given real and imaginary parts.
COMPLEX                                                            [type-doc]
A complex number
NIL
>
```

shows the function and type documentation for `complex`, and

```
> (help 'pi)
PI                                                             [variable-doc]
The floating-point number that is approximately equal to the ratio of the
circumference of a circle to its diameter.
NIL
>
```

shows the variable documentation for `pi`[13].

## 5.2 Listing and Undefining Variables

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:

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

If you are working on a 1Mb Macintosh you may occasionally want to 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
>
```

## 5.3 More on the XLISP-STAT Listener

Because of the large number of parentheses involved, Lisp expressions can be hard to read and type correctly. To make it easier to type readable, correct expressions the listener window on the Macintosh has the following features:

[13]Actually `pi` represents a constant, produced with `defconst`. Its value can't be changed by simple assignment.

- Typing a closing parenthesis flashes the matching opening parenthesis.
- You can move the cursor with the arrow keys, the mouse or the *backspace* key to any position in the current input expression, not just within the last line.
- If the current expression is more than one line long, hitting the *tab* key in any line but the first indents the line to its appropriate position according to (more or less) standard rules for Lisp code indentation.
- If the insertion point is at the end of the current expression hitting the *enter* key is equivalent to hitting a *return*. If the insertion point is not at the end of the expression, hitting *enter* moves the insertion point to the end of the expression.

These four features should make typing expressions correctly much easier. In particular, in translating mathematical formulas to Lisp it sometimes seems that you have to do things backwards. Using these features you can build up your expression from the inside out [14].

XLISP 2.0 provides a simple *command history* mechanism. The symbols `-`, `*`, `**`, `***`, `+`, `++`, and `+++` are used for this purpose. The top level reader binds these symbols as follows:

| | |
|---|---|
| `-` | the current input expression |
| `+` | the last expression read |
| `++` | the previous value of `+` |
| `+++` | the previous value of `++` |
| `*` | the result of the last evaluation |
| `**` | the previous value of `*` |
| `***` | the previous value of `**` |

The variables `*`, `**` and `***` are probably most useful. For example, if you construct a plot but forget to assign the resulting plot object to a variable you can recover it using one of the history variables:

```
> (histogram (normal-rand 50))
#<Object: 3701682, prototype = HISTOGRAM-PROTO>
> (def w *)
W
> w
#<Object: 3701682, prototype = HISTOGRAM-PROTO>
>
```

The symbol `W` now has the histogram object as its value and can be used to modify the plot, as described in Section 6.5 below.

Like most interactive systems, XLISP needs a system for dynamically managing memory. The system used by XLISP is to grab memory out of a fixed bin until the bin is exhausted. At that point the system pauses to reclaim memory that is no longer being used. This process, called *garbage collection*, will occasionally cause the system to freeze up for about a second. When the system garbage collects the Macintosh cursor changes to a trash bag.

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 listener. You must continue to hold down the key until the calculation stops.

[14] To support these features the listener checks the current expression each time you type a *return* to see if it has a complete expression. If so, the expression is passed to the reader and the evaluator. If not, you can continue typing. There are some heuristics involved here, and an expression with lots of quotes and comments may cause trouble, but it seems to work. Redefining the read table in major ways may not work as you might expect since some knowledge of standard Lisp syntax is built in to the listener.

## 5.4 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 in the folder `Data`. Any variables you save (see the next subsection 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. After loading a file using the **File** menu **Load** item the system does not print a prompt. Instead it prints a message indicating that the load is done:

```
> ; loading "temp.lsp"
; finished loading "temp.lsp"
```

## 5.5 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 on and off recording to a single file. You can also turn dribbling on and off using the **Dribble** item on the **Command** menu.

`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, or its command key shortcut COMMAND-C, 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 `savevar` function. This function allows you to save one or more variables 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

```
(savevar 'precipitation "precipitation")
```

Do not add the `.lsp` suffix yourself; `savevar` will supply it. To save the two variables `precipitation` and `purchases` in the file `examples.lsp` type [15].

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

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

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 or the XLISP-STAT editor and you can prepare files with your own data by following these examples.

## 5.6 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. As in the listener window, hitting the *tab* key in any line but the first of a multiline expression will indent the expression to a reasonable point. The editor also allows 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 variables or functions.

## 5.7 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.

## 5.8 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.

# 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 listener and mouse actions. The fifth subsection introduces the concept of a *plot object* and the idea of *sending messages* to an object from the listener 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 subsection shows how Lisp iteration can be used to construct a dynamic simulation – a plot movie.

## 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 a some data collected to examine the relationship between a phosphate absorption index and the amount of extractable iron and aluminum in a sediment (Devore and Peck [11, page 509, Example 6]). 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 on the left in Figure 7. The argument to `spin-plot` is a list of three lists or vectors, representing the $x$, $y$ and $z$ variables. The $z$ axis is initially pointing out of the screen. 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 of Figure 7 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))
```

(Devore and Peck[11, 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 produce the view in Figure 8. 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 [11] suggest fitting a model with quadratic terms to this data.

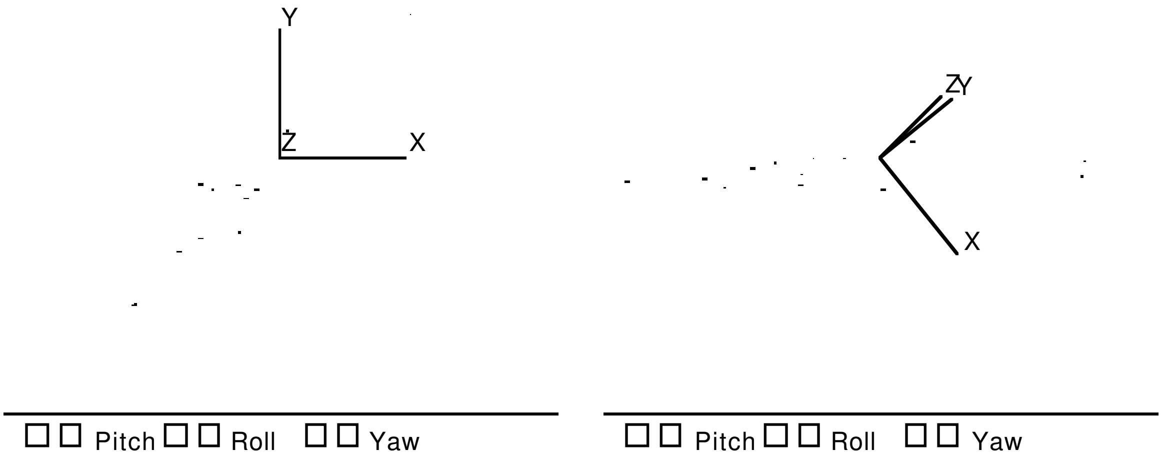


Figure 7: Two views of a rotatable plot of data on iron content, aluminum content and phosphate absorption in sediment samples.

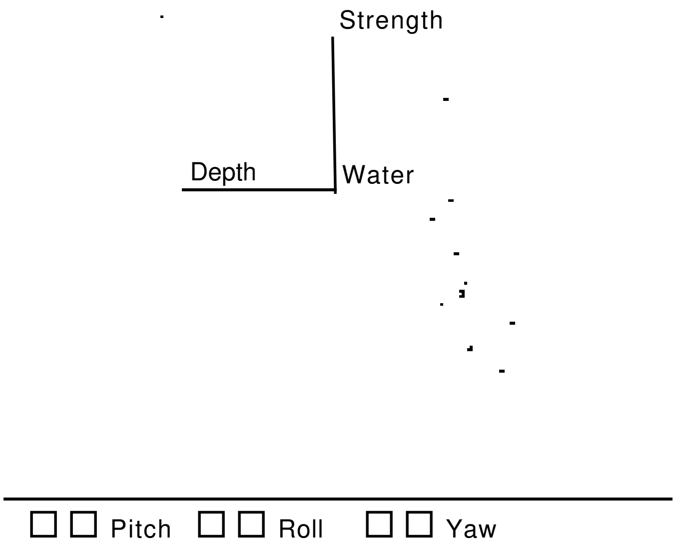


Figure 8: Rotatable plot of measurements on permafrost samples.

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)
```

Note that the `scale` keyword argument is given using the corresponding keyword symbol, the symbol `scale` preceded by a colon.

Rotation speed can be changed using the plot menu or the keyboard equivalents COMMAND-F for Faster and COMMAND-S for Slower.

Depth cuing and showing of the axes are controlled by items on the plot menu.

If you click the mouse in one of the `Pitch`, `Roll` or `Yaw` squares while holding down the *shift* key the plot will start to rotate and continue to rotate after the mouse button has been released.

### 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 tires and the expression

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

produces the scatterplot matrix in Figure 9. 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*.

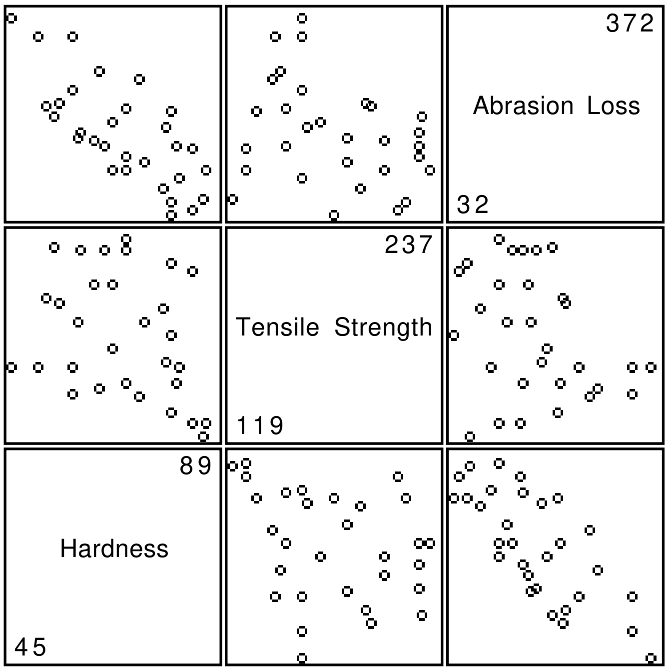


Figure 9: Scatterplot matrix of abrasion loss data.

*Selecting.* Your plot is in the selecting mode when the cursor is an arrow. This is the default setting. In this mode you can 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 **Mouse Mode...** from the **Scatmat** menu and selecting **Brushing** from the dialog box that is popped up. In this mode the cursor will look like a paint brush 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 in Figure 10 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))
```

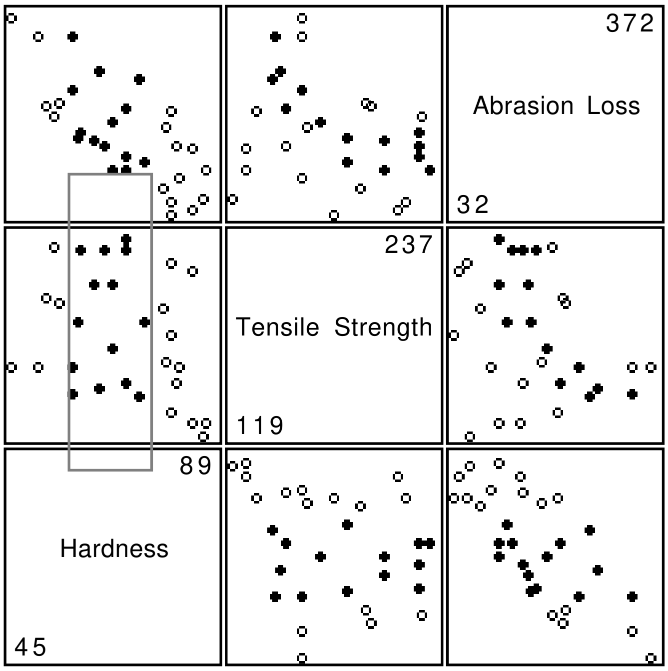


Figure 10: Scatterplot matrix with middle hardness values highlighted.

```
(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))
```

(Devore and Peck [11, page 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 in Figure 11 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 [11, 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 the `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.

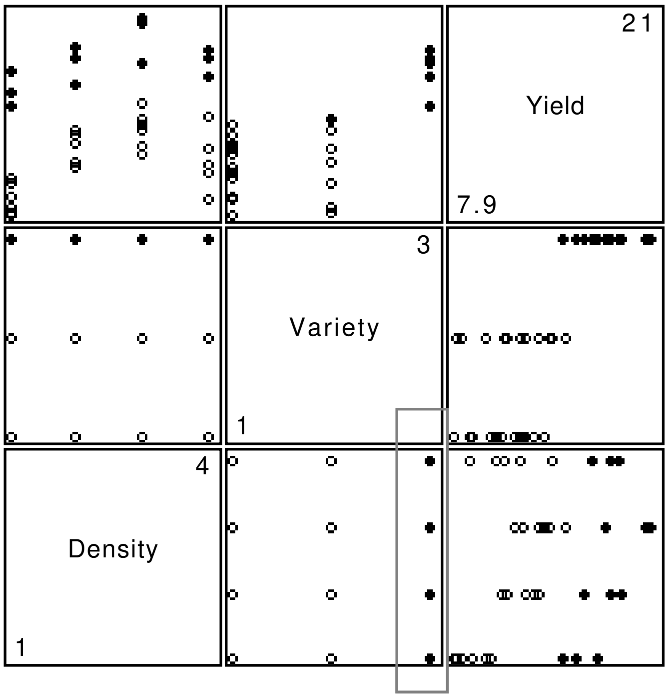


Figure 11: Scatterplot matrix for tomato yield data with points from the third variety highlighted.

## 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 "0", "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 `:point-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. The plot can then be rescaled using the **Rescale Plot** option. Alternatively, the **Focus on Selection** menu item removes all unselected points from the plot.

When a set of points is selected in a plot you can change the symbol used to display the points using the **Selection Symbol** item. On systems with color monitors you can set the color of selected points with the **Selection Color** item.

You can save the indices of the selected points in a variable by choosing the **Selection...** item in the plot menu. A dialog will ask you for a name for the selection. When no points are selected you can use the **Selection...** menu item to specify the indices of the points to select. A dialog will ask you for an expression for determining the selection to use. The expression can be any Lisp expression that evaluates to a list of indices.

## 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[16]. You can construct your own set of interacting plots by choosing the **Link**

[16]According to Stuetzle [16] the idea to link several plots was first suggested by McDonald [12].

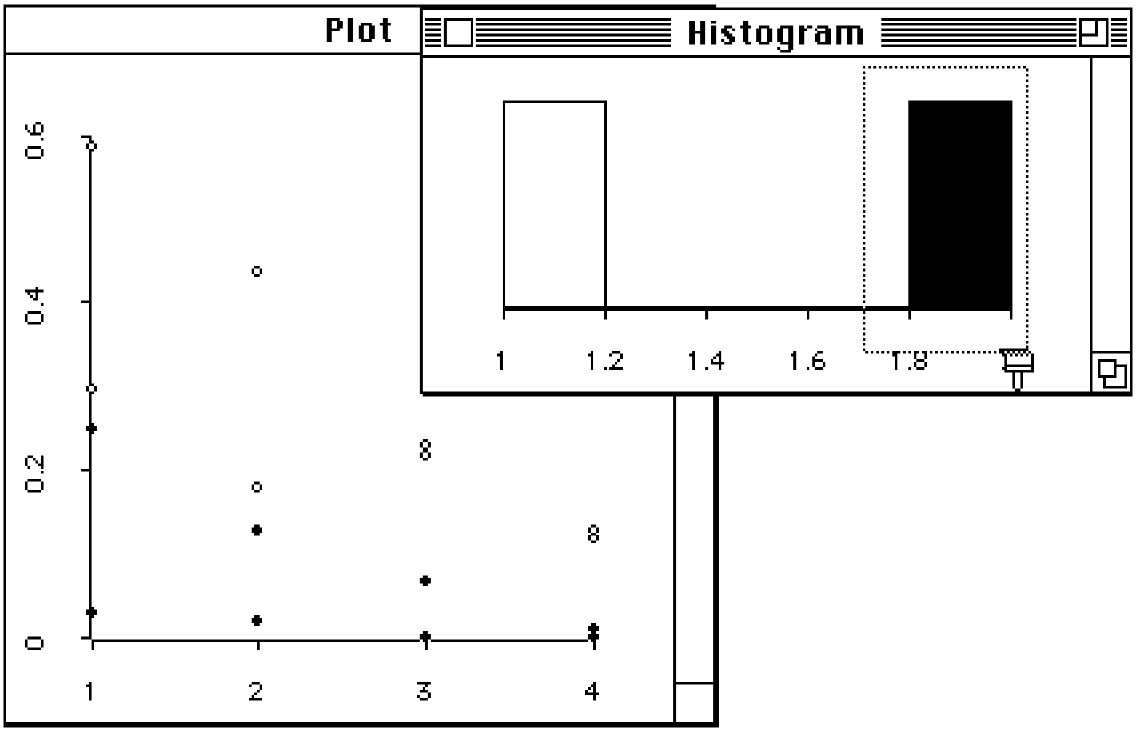


Figure 12: Scatterplot and histogram with points from one sugar group highlighted.

**View** option from the menus of the plots you want to link. For example, using the data from Exercise 1 in Section 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, as shown in Figure 12.

If you want to be able to select the points with particular labels 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.

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

will generate a list with the names "0" , . . . , "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 [11, page 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 `plot-points` function to a symbol [17]:

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

The result returned by plot-points is an XLISP-STAT *object* . To use an object you have to send it *messages*. This is done using the `send` function, as in the expression

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

I will use the expression

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

to tell `myplot` to add the graph of a line $a + bx$, with $a = -2.18$ and $b = 0.66$, to itself. The *message selector* is `:abline`, the numbers -2.18 and 0.66 are the arguments. The message consists of the selector and the arguments. Message selectors are always Lisp keywords; that is, they are symbols that start with a colon. Before typing in this expression you might want to resize and rearrange the listener window so you can see the plot and type commands at the same time. Once you have resized the listener window you can easily switch between the small window and a full size window by clicking in the zoom box at the right corner of the window title bar. The resulting plot is shown in Figure 13

Scatter plot objects understand a number of other messages. One other message is the `:help` message [18]:

```
> (send myplot :help)
> (send scatterplot-proto :help)
SCATTERPLOT-PROTO
Scatterplot.
Help is available on the following:

:ABLINE :ACTIVATE :ADD-FUNCTION :ADD-LINES :ADD-METHOD :ADD-MOUSE-MODE
:ADD-POINTS :ADD-SLOT :ADD-STRINGS :ADJUST-HILITE-STATE
:ADJUST-POINT-SCREEN-STATES :ADJUST-POINTS-IN-RECT :ADJUST-TO-DATA
:ALL-POINTS-SHOWING-P :ALL-POINTS-UNMASKED-P :ALLOCATE
:ANY-POINTS-SELECTED-P :APPLY-TRANSFORMATION :BACK-COLOR :BRUSH
:BUFFER-TO-SCREEN :CANVAS-HEIGHT :CANVAS-WIDTH :CLEAR :CLEAR-LINES
:CLEAR-MASKS :CLEAR-POINTS :CLEAR-STRINGS ....
```

[17] The result returned by `plot-points` is printed something like `#<Object: 2010278, prototype = SCATTERPLOT-PROTO>`. 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 `open` function, 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 result into a `def` expression – you have to redo the plot or use the history mechanism.

[18] To keep things simple I will use the term *message* to refer to a message corresponding to a message selector.

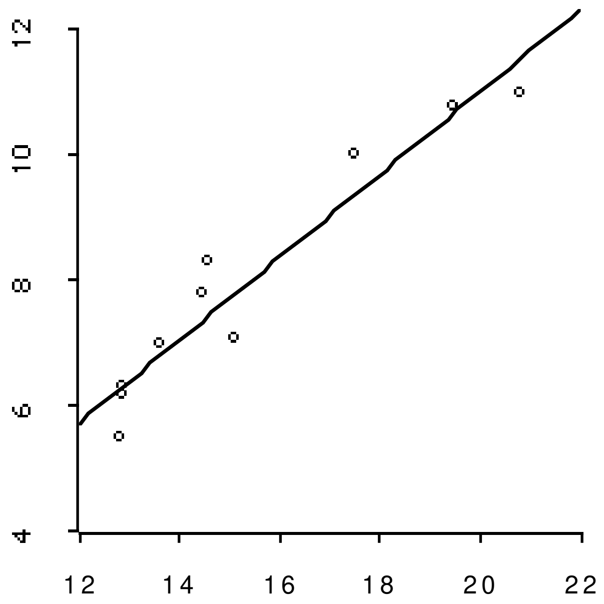


Figure 13: Scatterplot of bicycle data with fitted line.

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

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:

```
> (send myplot :help :add-points)
:ADD-POINTS
Method args: (points &key point-labels (draw t))
Adds points to plot. POINTS is a list of sequences, POINT-LABELS a list of
strings. If DRAW is true the new points are added to the screen.
NIL
> (send myplot :help :add-lines)
:ADD-LINES
Method args: (lines &key type (draw t))
Adds lines to plot. LINES is a list of sequences, the coordinates of the line
starts. TYPE is normal or dashed. If DRAW is true the new lines are added to the
screen.
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))
```

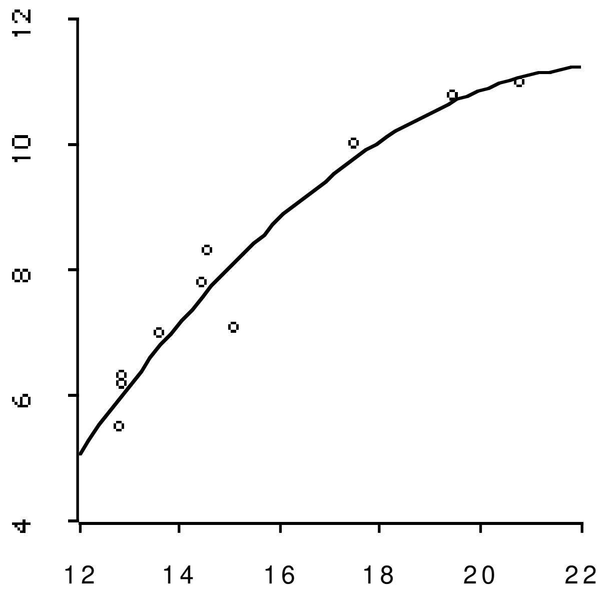


Figure 14: Scatterplot of bicycle data with fitted curve.

```
(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

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

change myplot to look like Figure 14. 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 here 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

```
> (send h :help :clear)
:CLEAR

Message args: (&key (draw t))
Clears the plot data. If DRAW is nil the plot is redrawn; otherwise its
```

```
current screen image remains unchanged.
NIL
>
```

the `:clear` message takes an optional keyword 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 listener window is the active window. Then type the expression [19]

```
(dotimes (i 50)
         (send h :clear :draw nil)
         (send 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 keyword argument `draw` with value `NIL` 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 `:point-selected` , `:point-hilited` and `:point-showing` are particularly useful for dynamic simulations. Here is the help information for `:point-selected` in a histogram:

```
> (send h :help :point-selected)
:POINT-SELECTED
Method args: (point &optional selected)
Sets or returns selection status (true or NIL) of POINT. Sends
:ADJUST-POINT-SCREEN-STATES message if states are set. Vectorized.
NIL
>
```

Thus you can use this message to determine whether a point is currently selected and also to select or unselect it. Again rearrange the windows so you can see the histogram while typing in the listener window and type the expression

```
(dolist (i (order hardness))
        (send h :point-selected i t)
        (send h :point-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.

[19] `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 `var` bound to 0, 1, ..., `count - 1`. Other looping constructs are `dolist`, `do` and `do*`.

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.

An intermediate solution is possible: You can set up a dialog window with a scroll bar that will run through the indices in the list `(order hardness)`, selecting the corresponding point as it is scrolled. An example in Section 8 will show you how to do this.

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))
        (send h :point-selected i t)
        (dotimes (i 100))
        (send h :point-selected i nil))
```

in place of the previous expression.

# 7 Regression

Regression models have been implemented using XLISP-STAT'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: 1966006, prototype = REGRESSION-MODEL-PROTO>
>
```

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 several optional keyword arguments. The most important ones are `: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`. In a weighted model 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 [20]

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

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:

[20] Recall from Section 6.5 that `#<Object: 1966006, prototype = REGRESSION-MODEL-PROTO>` 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 or use the history mechanism.

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

:ADD-METHOD :ADD-SLOT :BASIS :CASE-LABELS :COEF-ESTIMATES :COEF-STANDARD-ERRORS
:COMPUTE :COOKS-DISTANCES :DELETE-METHOD :DELETE-SLOT :DF :DISPLAY :DOC-TOPICS
:DOCUMENTATION :EXTERNALLY-STUDENTIZED-RESIDUALS :FIT-VALUES :GET-METHOD
:HAS-METHOD :HAS-SLOT :HELP :INCLUDED :INTERCEPT :INTERNAL-DOC :ISNEW
:LEVERAGES :METHOD-SELECTORS :NEW :NUM-CASES :NUM-COEFS :NUM-INCLUDED
:OWN-METHODS :OWN-SLOTS :PARENTS :PLOT-BAYES-RESIDUALS :PLOT-RESIDUALS
:PRECEDENCE-LIST :PREDICTOR-NAMES :PRINT :R-SQUARED :RAW-RESIDUALS
:RESIDUAL-SUM-OF-SQUARES :RESIDUALS :RESPONSE-NAME :RETYPE :SAVE :SHOW
:SIGMA-HAT :SLOT-NAMES :SLOT-VALUE :STUDENTIZED-RESIDUALS :SUM-OF-SQUARES
:SWEEP-MATRIX :TOTAL-SUM-OF-SQUARES :WEIGHTS :X :X-MATRIX :XTXINV :Y PROTO
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 [21]:

```
> (send bikes :coef-estimates)
(-2.182472 0.6603419)
> (send 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:

```
> (send bikes :help :plot-residuals)
: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)
(send bikes :plot-residuals travel-space)
```

we can construct two plots of the data as shown in Figure 15. 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).

[21] 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.

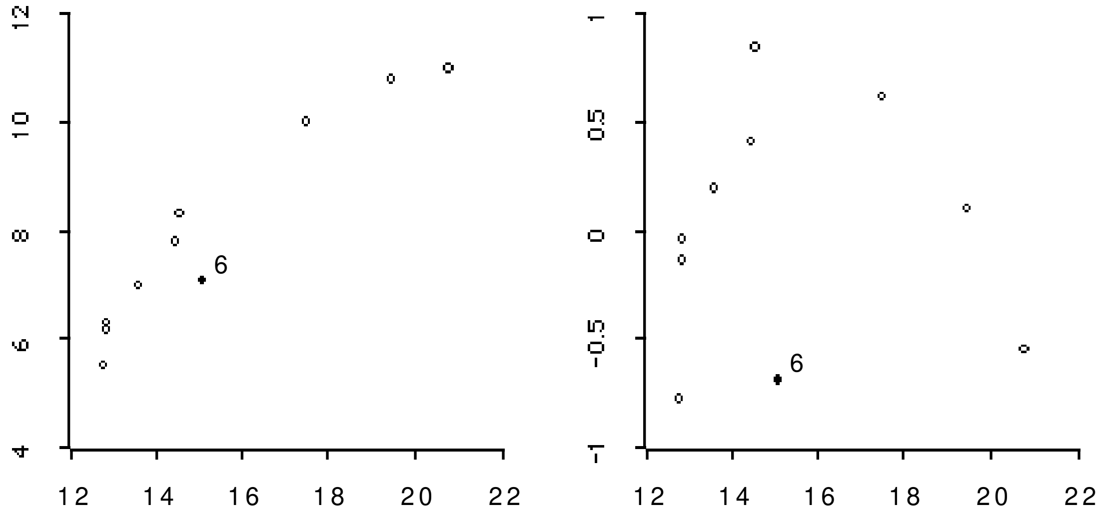


Figure 15: Linked raw data and residual plots for the bicycles example.

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  (regression-model (list travel-space (^ travel-space 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

BIKES2
>
```

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

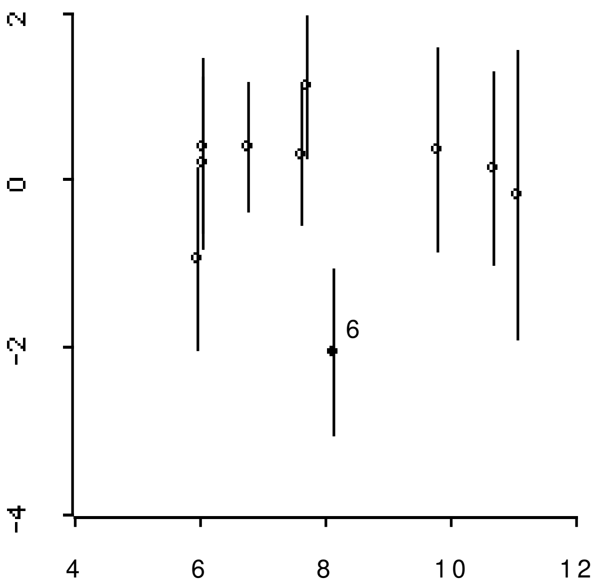


Figure 16: Bayes residual plot for bicycle data.

Then Cook's distances are obtained as [22]

```
> (* (^ studres 2)
     (/ (send bikes2 :leverages) (- 1 (send 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 [7], which can be obtained using the message `:plot-bayes-residuals`. The expression `(send bikes2 :plot-bayes-residuals)` produces the plot in Figure 16. The bars represent mean $\pm 2SD$ 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 $\hat{\sigma}$. 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%.

Several other methods are available for residual and case analysis. These include `:studentized-residuals` and `:cooks-distances`, which we could have used above instead of calculating these quantities from their definitions. Another useful message is `:included`, which can be used to change the cases to be used in estimating a model. Further details on these messages are available in their help information.

## 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.

[22] 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.

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 and Methods

This section gives a brief introduction to programming XLISP-STAT. The most basic programming operation is to define a new function. Closely related is the idea of defining a new method for an object. [23]

## 8.1 Defining 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 4.3 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 5.6 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 ((send ss1 (m1 :sum-of-squares))
        (send df1 (m1 :df))
        (send ss2 (m2 :sum-of-squares))
        (send df2 (m2 :df)))
    (/ (/ (- ss1 ss2) (- df1 df2)) (/ ss2 df2))))
```

This definition uses the Lisp `let` construct to establish some local *variable bindings*. The variables `ss1`, `df1`, `ss2` and `df2` are defined in terms of the two model arguments `m1` and `m2`, and are then used to compute the $F$ statistic. The string following the argument list 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.

[23] The discussion in this section only scratches 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. For more information on options of function definition, macros, etc. see the XLISP documentation and the books on Lisp mentioned in the references.

## 8.2 Anonymous Functions

Suppose you would like to plot the function $f(x) = 2x + x^2$ over the range $-2 \le x \le 3$. We can do this by first defining a function `f` and then using `plot-function`:

```
(defun f (x) (+ (* 2 x) (^ x 2)))
(plot-function #'f -2 3)
```

This is not too hard, but it nevertheless involves one unnecessary step: naming the function `f`. You probably won't need this function again; it is a throw-away function defined only to allow you to give it to `plot-function` as an argument. It would be nice if you could just give `plot-function` an expression that constructs the function you want. Here is such an expression:

```
(function (lambda (x) (+ (* 2 x) (^ x 2))))
```

There are two steps involved. The first is to specify the definition of your function. This is done using a *lambda expression*, in this case

```
(lambda (x) (+ (* 2 x) (^ x 2)))
```

A lambda expression is a list starting with the symbol `lambda`, followed by the list of arguments and the expressions making up the body of the function. The lambda expression is only a definition, it is not yet a function, a piece of code that can be applied to arguments. The special form `function` takes the lambda list and constructs such a function. The result can be saved in a variable or it can be passed on to another function as an argument. For our plotting problem you can use the single expression

```
(plot-function (function (lambda (x) (+ (* 2 x) (^ x 2)))) -2 3)
```

or, using the `#'` short form,

```
(plot-function #'(lambda (x) (+ (* 2 x) (^ x 2))) -2 3)
```

Since the function used in these expressions is never named it is sometimes called an *anonymous function*.

You can also construct a rotating plot of a function of two variables using the function `spin-function`. As an example, the expression

```
(spin-function #'(lambda (x y) (+ (^ x 2) (^ y 2))) -1 1 -1 1)
```

constructs a plot of the function $f(x, y) = x^2 + y^2$ over the range $-1 \le x \le 1, -1 \le y \le 1$ using a $6 \times 6$ grid. The number of grid points can be changed using the `:num-points` keyword. The result is shown in Figure 17. Again it convenient to use a lambda expression to specify the function to be plotted.

There are a number of other situations in which you might want to pass a function on as an argument without first going through the trouble of thinking up a name for the function and defining it using `defun`. A few additional examples are given in the next subsection.

## 8.3 Some Dynamic Simulations

In Section 6.6 we used a loop to control a dynamic simulation in which points in a histogram of one variable were selected and deselected in the order of a second variable. Let's look at how to run the same simulation using a *slider* to control the simulation.

A slider is a modeless dialog box containing a scroll bar and a value display field. As the scroll bar is moved the displayed value is changed and an action is taken. The action is defined by an *action function* given to the scroll bar, a function that is called with one value, the current slider

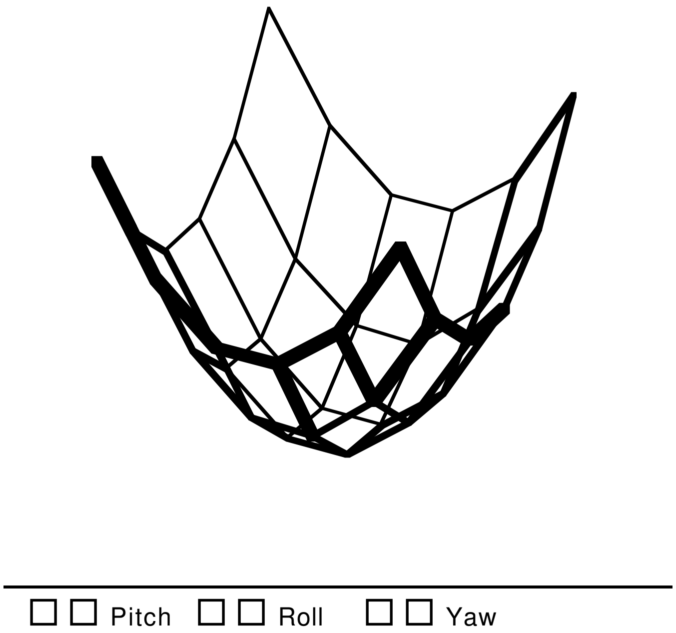


Figure 17: Rotatable plot of $f(x, y) = x^2 + y^2$.

value, each time the value is changed by the user. There are two kinds of sliders, sequence sliders and interval sliders. Sequence sliders take a sequence (a list or a vector) and scroll up and down the sequence. The displayed value is either the current sequence element or the corresponding element of a display sequence. An interval slider dialog takes an interval, divides it into a grid and scrolls through the grid. By default a grid of around 30 points is used; the exact number and the interval end points are adjusted to give nice printed values. The current interval point is displayed in the display field.

For our example let's use a sequence slider to scroll through the elements of the `hardness` list in order and select the corresponding element of `abrasion-loss`. The expression

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

sets up a histogram and saves its plot object in the variable `h`. The function `sequence-slider-dialog` takes a list or vector and opens a sequence slider to scroll through its argument. To do something useful with this dialog we need to give it an action function as the value of the keyword argument `:action`. The function should take one argument, the current value of the sequence controlled by the slider. The expression

```
(sequence-slider-dialog (order hardness) :action
                        #'(lambda (i)
                            (send h :unselect-all-points)
                            (send h :point-selected i t)))
```

sets up a slider for moving the selected point in the `abrasion-loss` histogram along the order of the `hardness` variable. The histogram and scroll bar are shown in Figure 18. The action function is called every time the slider is moved. It is called with the current element of the sequence `(order hardness)`, the index of the point to select. It clears all current selections and then selects the point specified in the call from the slider. The body of the function is almost identical to the body of the

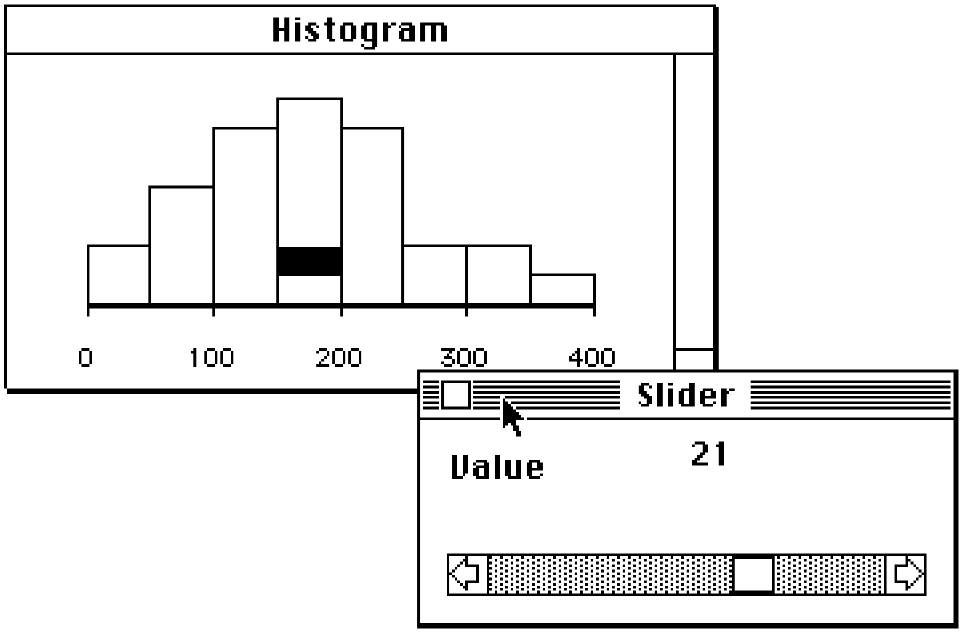


Figure 18: Slider-controlled animation of a histogram.

`dotimes` loop used in Section 6.6. The slider is thus an interactive, graphically controlled version of this loop.

As another example, suppose we would like to examine the effect of changing the exponent in a Box-Cox power transformation

$$h(x) = \begin{cases} \frac{x^\lambda - 1}{\lambda} & \text{if } \lambda \neq 0 \\ \log(x) & \text{otherwise} \end{cases}$$

on a probability plot. As a first step we might define a function to compute the power transformation and normalize the result to fall between zero and one:

```
(defun bc (x p)
  (let* ((bcx (if (< (abs p) .0001) (log x) (/ (^ x p) p)))
         (min (min bcx))
         (max (max bcx)))
    (/ (- bcx min) (- max min))))
```

This definition uses the `let*` form to establish some local variable bindings. The `let*` form is like the `let` form used above except that `let*` defines its variables sequentially, allowing a variable to be defined in terms of other variables already defined in the `let*` expression; `let` on the other hand creates its assignments in parallel. In this case the variables `min` and `max` are defined in terms of the variable `bcx`.

Next we need a set of positive numbers to transform. Let's use a sample of thirty observations from a $\chi^2_4$ distribution and order the observations:

```
(def x (sort-data (chisq-rand 30 4)))
```

The normal quantiles of the expected uniform order statistics are given by

```
(def r (normal-quant (/ (iseq 1 30) 31)))
```

and a probability plot of the untransformed data is constructed using

```
(def myplot (plot-points r (bc x 1)))
```

Since the power used is 1 the function `bc` just rescales the data.

There are several ways to set up a slider dialog to control the power parameter. The simplest approach is to use the function `interval-slider-dialog`:

```
(interval-slider-dialog (list -1 2)
                        :points 20
                        :action #'(lambda (p)
                                    (send myplot :clear nil)
                                    (send myplot :add-points r (bc x p))))
```

`interval-slider-dialog` takes a list of two numbers, the lower and upper bounds of an interval. The action function is called with the current position in the interval.

This approach works fine on a Mac II but may be a bit slow on a Mac Plus or a Mac SE. An alternative is to pre-compute the transformations for a list of powers and then use the pre-computed values in the display. For example, using the powers defined in

```
(def powers (rseq -1 2 16))
```

we can compute the transformed data for each power and save the result as the variable `xlist`:

```
(def xlist (mapcar #'(lambda (p) (bc x p)) powers))
```

The function `mapcar` is one of several *mapping functions* available in Lisp. The first argument to `mapcar` is a function. The second argument is a list. `mapcar` takes the function, applies it to each element of the list and returns the list of the results [24]. Now we can use a sequence slider to move up and down the transformed values in `xlist`:

```
(sequence-slider-dialog xlist
                        :display powers
                        :action #'(lambda (x)
                                    (send myplot :clear nil)
                                    (send myplot :add-points r x)))
```

Note that we are scrolling through a list of lists and the element passed to the action function is thus the list of current transformed values. We would not want to see these values in the display field on the slider, so I have used the keyword argument `:display` to specify an alternative display sequence, the powers used in the transformation.

## 8.4 Defining Methods

When a message is sent to an object the object system will use the object and the method selector to find the appropriate piece of code to execute. Different objects may thus respond differently to the same message. A linear regression model and a nonlinear regression model might both respond to a `:coef-estimates` message but they will execute different code to compute their responses.

The code used by an object to respond to a message is called a *method*. Objects are organized in a hierarchy in which objects *inherit* from other objects. If an object does not have a method of its own for responding to a message it will use a method inherited from one of its ancestors. The

[24] `mapcar` can be given several lists after the function. The function must take as many arguments as there are lists. `mapcar` will apply the function using the first element of each list, then using the second element, and so on, until one of the lists is exhausted, and return a list of the results.

`send` function will move up the *precedence list* of an object's ancestors until a method for a message is found.

Most of the objects encountered so far inherit directly from *prototype* objects. Scatterplots inherit from `scatterplot-proto`, histograms from `histogram-proto`, regression models from `regression-model-proto`. Prototypes are just like any other objects. They are essentially *typical* versions of a certain kind of object that define default behavior. Almost all methods are owned by prototypes. But any object can own a method, and in the process of debugging a new method it is often better to attach the method to a separate object constructed for that purpose instead of the prototype.

To add a method to an object you can use the `defmeth` macro. As an example, in Section 7 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:

```
(defmeth object :new-method arg-list body)
```

`object` is the object that is to own the new method. In the case of regression models this can be either a specific regression model or the prototype that all regression models inherit from, `regression-model-proto`. The argument `:new-method` is the message selector for your method; in our case this would be `:cooks-distances`. The argument `arg-list` is the list of argument names for your method, and `body` is one or more expressions making up the body of the method. When the method is used each of these expressions will be evaluated, in the order in which they appear.

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

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

The variable `self` refers to the object that is receiving the message. This definition is close to the definition of this method supplied in the file `regression.lsp`.

## 8.5 Plot Methods

All plot activities are accomplished by sending messages to plot objects. For example, every time a plot needs to be redrawn the system sends the plot the `:redraw` message. By defining a new method for this message you can change the way a plot is drawn. Similarly, when the mouse is moved or clicked in a plot the plot is sent the `:do-mouse` message. Menu items also send messages to their plots when they are selected. If you are interested in modifying plot behavior you may be able to get started by looking at the methods defined in the graphics files loaded on start up. Further details will be given in [17].

# 9 Matrices and Arrays

XLISP-STAT includes support for multidimensional arrays patterned after the Common Lisp standard described in detail in Steele [15]. The functions supported are listed in Section C.6 of 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`, `transpose`, `chol-decomp`, `lu-decomp` and `sv-decomp`. These functions are listed in the appendix as well.

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 prefix `#2A` indicates that this is a two-dimensional array. This form is not particularly readable, but it has the advantage that it can be pasted into expressions and will be read as an array by the XLISP reader.[25] For matrices you can use the function `print-matrix` to get a slightly more readable representation:

```
> (print-matrix '#2a((1 2 3)(4 5 6)))
#2a(
    (1 2 3)
    (4 5 6)
   )
NIL
>
```

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

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

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

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

returns the upper left hand corner of `A`.

[25] You should quote an array if you type it in using this form, as the value of an array is not defined.

# 10 Nonlinear Regression

XLISP-STAT allows the construction of nonlinear, normal regression models. The present implementation is experimental. 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, Bates and Watts [1, A1.3] describe an experiment on the relation between the velocity of an enzymatic reaction, $y$, and the substrate concentration, $x$. The data for an experiment in which the enzyme was treated with Puromycin are given by

```
(def x1 (list 0.02 0.02 0.06 0.06 .11 .11 .22 .22 .56 .56 1.1 1.1))
(def y1 (list 76 47 97 107 123 139 159 152 191 201 207 200))
```

The Michaelis-Menten function

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

often provides a good model for the dependence of velocity on substrate concentration. Assuming the Michaelis-Menten function as the mean velocity at a given concentration level the function `f1` defined by

```
(defun f1 (b) (/ (* (select b 0) x1) (+ (select b 1) x1)))
```

computes the list of mean response values at the points in `x1` for a parameter list `b`. Using these definitions, which are contained in the file `puromycin.lsp` in the `Data` folder of the distribution disk, we can construct a nonlinear regression model using the `nreg-model` function.

First we need initial estimates for the two model parameters. Examining the expression for the Michaelis-Menten model shows that as $x$ increases the function approaches an asymptote, $\theta_1$. The second parameter, $\theta_2$, can be interpreted as the value of $x$ at which the function has reached half its asymptotic value. Using these interpretations for the parameters and a plot constructed by the expression

```
(plot-points x1 y1)
```

shown in Figure 19 we can read off reasonable initial estimates of 200 for $\theta_1$ and 0.1 for $\theta_2$. The `nreg-model` function takes the mean function, the response vector and an initial estimate of the parameters, computes more accurate estimates using an iterative algorithm starting from the initial guess, and prints a summary of the results. It returns a nonlinear regression model object: [26]

```
> (def puromycin (nreg-model #'f1 y1 (list 200 .1)))
Residual sum of squares:    7964.19
Residual sum of squares:    1593.16
Residual sum of squares:    1201.03
Residual sum of squares:    1195.51
Residual sum of squares:    1195.45
Residual sum of squares:    1195.45
Residual sum of squares:    1195.45
Residual sum of squares:    1195.45

Least Squares Estimates:
```

[26] Recall that the expression `#'f1` is short for `(function f1)` and is used for obtaining the function definition associated with the symbol `f1`.

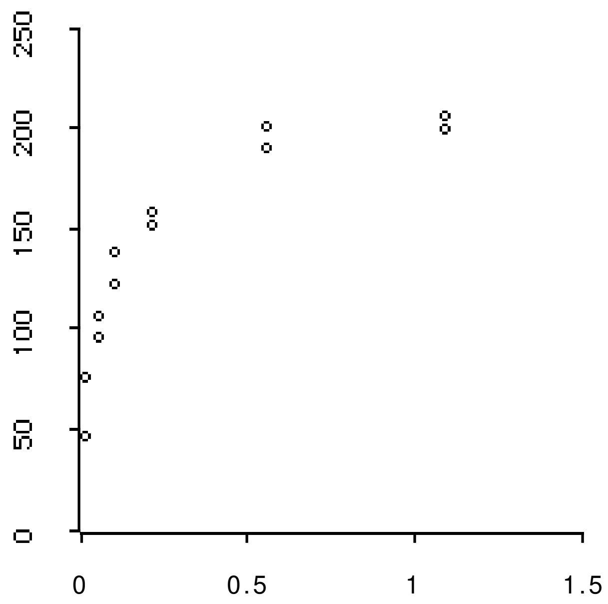


Figure 19: Plot of reaction velocity against substrate concentration for Puromycin experiment.

```
Parameter 0              212.684    (6.94715)
Parameter 1            0.0641213    (0.00828094)

R Squared:              0.961261
Sigma hat:               10.9337
Number of cases:              12
Degrees of freedom:           10

PUROMYCIN
>
```

The function `nreg-model` also takes several keyword arguments, including `:epsilon` to specify a convergence criterion and `:count-limit`, a limit on the number of iterations. By default these values are .0001 and 20, respectively. The algorithm for fitting the model is a simple Gauss-Newton algorithm with backtracking; derivatives are computed numerically.

To see how you can analyze the model further you can send `puromycin` the `:help` message. The result is very similar to the help information for a linear regression model. The reason is simple: nonlinear regression models have been implemented as objects, with the nonlinear regression model prototype `nreg-model-proto` inheriting from the linear regression model prototype. The internal data, the method for computing estimates, and the method of computing fitted values have been modified for nonlinear models, 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, such as the methods `:coef-standard-errors` and `:leverages`, still make sense, at least as first order asymptotic approximations.

In addition to the messages for linear regression models a nonlinear regression model can respond to the messages

```
:COUNT-LIMIT
:EPSILON
:MEAN-FUNCTION
:NEW-INITIAL-GUESS
:PARAMETER-NAMES
:THETA-HAT
:VERBOSE
```

## Exercises

1. Examine the residuals of the `puromycin` model.
2. The file `puromycin.lsp` also contains data `x2` and `y2` and mean function `f2` for an experiment run without the Puromycin treatment. Fit a model to this data and compare the results to the experiment with Puromycin.

## 11 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 levels 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 MeanSquare:      0.02627865   (40)

CHOLESTEROL
>
```

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-DF
:GROUP-MEAN-SQUARE
:GROUP-NAMES
:GROUP-SUM-OF-SQUARES
:GROUPED-DATA
:STANDARD-DEVIATIONS
```

# 12 Maximization and Maximum Likelihood Estimation

XLISP-STAT includes two functions for maximizing functions of several variables. The definitions needed are in the file `maximize.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. The material in this section is somewhat more advanced as it assumes you are familiar with the basic concepts of maximum likelihood estimation.

Two functions are available for maximization. The first is `newtonmax`, which takes a function and a list or vector representing an initial guess for the location of the maximum and attempts to find the maximum using an algorithm based on Newton's method with backtracking. The algorithm is based on the unconstrained minimization system described in Dennis and Schnabel [10].

As an example, Cox and Snell [9, Example T] describe data collected on times (in operating hours) between failures of air conditioning units on several aircraft. The data for one of the aircraft can be entered as

```
(def x (90 10 60 186 61 49 14 24 56 20 79 84 44 59 29 118 25 156 310 76
        26 44 23 62 130 208 70 101 208))
```

A simple model for these data might be to assume the times between failures are independent random variables with a common gamma distribution. The density of the gamma distribution can be written as

$$\frac{(\beta/\mu)(\beta x/\mu)^{\beta-1}e^{-\beta x/\mu}}{\Gamma(\beta)}$$

where $\mu$ is the mean time between failures and $\beta$ is the gamma shape parameter. The log likelihood for the sample is thus given by

$$n[\log(\beta) - \log(\mu) - \log(\Gamma(\beta))] + \sum_{i=1}^{n}(\beta - 1)\log(\beta x_i/\mu) - \sum_{i=1}^{n}\beta x_i/\mu.$$

We can define a Lisp function to evaluate this log likelihood by

```
(defun gllik (theta)
  (let* ((mu (select theta 0))
         (beta (select theta 1))
         (n (length x))
         (bym (* x (/ beta mu))))
    (+ (* n (- (log beta) (log mu) (log-gamma beta)))
       (sum (* (- beta 1) (log bym)))
       (sum (- bym)))))
```

This definition uses the function `log-gamma` to evaluate $\log(\Gamma(\beta))$. The data and function definition are contained in the file `aircraft.lsp` in the `Data` folder of the distribution disk.

Closed form maximum likelihood estimates are not available for the shape parameter of this distribution, but we can use `newtonmax` to compute estimates numerically. [27] We need an initial guess to use as a starting value in the maximization. To get initial estimates we can compute the mean and standard deviation of `x`

```
> (mean x)
83.5172
> (standard-deviation x)
70.8059
```

[27] The maximizing value for $\mu$ is always the sample mean. We could take advantage of this fact and reduce the problem to a one dimensional maximization problem, but it is simpler to just maximize over both parameters.

and use method of moments estimates $\hat{\mu} = 83.52$ and $\hat{\beta} = (\hat{\mu}/\hat{\sigma})^2$, calculated as

```
> (^ (/ (mean x) (standard-deviation x)) 2)
1.39128
```

Using these starting values we can now maximize the log likelihood function:

```
> (newtonmax #'gllik (list 83.5 1.4))
maximizing...
Iteration 0.
Criterion value = -155.603
Iteration 1.
Criterion value = -155.354
Iteration 2.
Criterion value = -155.347
Iteration 3.
Criterion value = -155.347
Reason for termination: gradient size is less than gradient tolerance.
(83.5173 1.67099)
```

Some status information is printed as the optimization proceeds. You can turn this off by supplying the keyword argument `:verbose` with value `NIL`.

You might want to check that the gradient of the function is indeed close to zero. If you do not have a closed form expression for the gradient you can use `numgrad` to calculate a numerical approximation. For our example,

```
> (numgrad #'gllik (list 83.5173 1.67099))
(-4.07269e-07 -1.25755e-05)
```

The elements of the gradient are indeed quite close to zero. You can also compute the second derivative, or Hessian, matrix using `numhess`. Approximate standard errors of the maximum likelihood estimates are given by the square roots of the diagonal entries of the inverse of the negative Hessian matrix at the maximum:

```
> (sqrt (diagonal (inverse (- (numhess #'gllik (list 83.5173 1.67099))))))
(11.9976 0.402648)
```

Instead of calculating the maximum using `newtonmax` and then calculating the derivatives separately you can have `newtonmax` return a list of the location of the maximum, the optimal function value, the gradient and the Hessian by supplying the keyword argument `:return-derivs` as `T`. [28]

Newton's method assumes a function is twice continuously differentiable. If your function is not smooth or you are having trouble with `newtonmax` for some other reason you might try a second maximization function, `nelmeadmax`. `nelmeadmax` takes a function and an initial guess and attempts to find the maximum using the Nelder-Mead simplex method as described in Press, Flannery, Teukolsky and Vetterling [14]. The initial guess can consist of a simplex, a list of $n+1$ points for an $n$-dimensional problem, or it can be a single point, represented by a list or vector of $n$ numbers. If you specify a single point you should also use the keyword argument `:size` to specify as a list or vector of length $n$ the size in each dimension of the initial simplex. This should represent the size of an initial range in which the algorithm is to start its search for a maximum. We can use this method in our gamma example:

[28] The function `newtonmax` ordinarily uses numerical derivatives in its computations. Occasionally this may not be accurate enough or may take too long. If you have an expression for computing the gradient or the Hessian then you can use these by having your function return a list of the function value and the gradient, or a list of the function value, the gradient and the Hessian matrix, instead of just returning the function value.

```
> (nelmeadmax #'gllik (list 83.5 1.4) :size (list 5 .2))
Value = -155.603
Value = -155.603
Value = -155.603
Value = -155.587
Value = -155.53
Value = -155.522
...
Value = -155.347
Value = -155.347
Value = -155.347
Value = -155.347
(83.5181 1.6709)
```

It takes somewhat longer than Newton's method but it does reach the same result.

## Exercises

1. The data set used in this example consists of sets of measurements for ten aircraft. Data for five of the aricraft are contained in the variable `failure-times` in the file `aircraft.lsp`. The calculations of this section used the data for the second aricraft. Examine the data for the remaining four aircraft.

2. Use maximum likelihood estimation to fit a Weibull model to the data used in this section.

# 13 Approximate Bayesian Computations

This section describes a set of tools for computing approximate posterior moments and marginal densities in XLISP-STAT. The definitions needed are in the file `bayes.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. The material in this section is somewhat more advanced as it assumes you are familiar with the basic concepts of Bayesian inference.

The functions described in this section can be used to compute first and second order approximations to posterior moments and saddlepoint-like approximations to one dimensional marginal posterior densities. The approximations, based primarily on the results in [18, 19, 20], assume the posterior density is smooth and dominated by a single mode. The implementation is experimental and may change in a number of ways in the near future.

Let's start with a simple example, a data set used to study the relation between survival time (in weeks) of leukemia patients and white blood cell count recorded for the patients at their entry into the study [9, Example U]. The data consists of two groups of patients classified as AG positive and AG negative. The data for the 17 AG positive patients, contained in the file `leukemia.lsp` in the `Data` folder on the distribution disk, can be entered as

```
(def wbc-pos (list 2300 750 4300 2600 6000 10500 10000 17000 5400 7000
                   9400 32000 35000 100000 100000 52000 100000))
(def times-pos (list 65 156 100 134 16 108 121 4 39 143 56 26 22 1 1 5 65))
```

A high white blood cell count indicates a more serious stage of the disease and thus a lower chance of survival.

A model often used for this data assumes the survival times are exponentially distributed with a mean that is log linear in the logarithm of the white blood cell count. For convenience I will scale the white blood cell counts by 10000. That is, the mean survival time for a patient with white blood cell count $\log(WBC_i)$ is

$$\mu_i = \theta_0 \exp\{-\theta_1 x_i\},$$

with $x_i = \log(WBC_i/10000)$. The log likelihood function is thus given by

$$\sum_{i=1}^{n} \theta_1 x_i - n\log(\theta_0) - \frac{1}{\theta_0}\sum_{i=1}^{n} y_i e^{\theta_1 x_i},$$

with $y_i$ representing the survival times. After computing the transformed $WBC$ variable as

```
(def transformed-wbc-pos (- (log wbc-pos) (log 10000)))
```

the log likelihood can be computed using the function

```
(defun llik-pos (theta)
  (let* ((x transformed-wbc-pos)
         (y times-pos)
         (theta0 (select theta 0))
         (theta1 (select theta 1))
         (t1x (* theta1 x)))
    (- (sum t1x)
       (* (length x) (log theta0))
       (/ (sum (* y (exp t1x)))
          theta0))))
```

I will look at this problem using a vague, improper prior distribution that is constant over the range $\theta_i > 0$; thus the log posterior density is equal to the log likelihood constructed above, up to an additive constant. The first step is to construct a Bayes model object using the function `bayes-model`. This function takes a function for computing the log posterior density and an initial guess for the posterior mode, computes the posterior mode by an iterative method starting with the initial guess, prints a short summary of the information in the posterior distribution, and returns a model object. We can use the function `llik-pos` to compute the log posterior density, so all we need is an initial estimate for the posterior mode. Since the model we are using assumes a linear relationship between the logarithm of the mean survival time and the transformed $WBC$ variable a linear regression of the logarithms of the survival times on `transformed-wbc-pos` should provide reasonable estimates. The linear regression gives

```
> (regression-model transformed-wbc-pos (log times-pos))

Least Squares Estimates:

Constant                  3.54234   (0.302699)
Variable 0              -0.817801   (0.214047)

R Squared:                 0.4932
Sigma hat:                1.23274
Number of cases:               17
Degrees of freedom:            15
```

so reasonable initial estimates of the mode are $\hat{\theta}_0 = \exp(3.5)$ and $\hat{\theta}_1 = 0.8$. Now we can use these estimates in the `bayes-model` function:

```
> (def lk (bayes-model #'llik-pos (list (exp 3.5) .8)))
maximizing...
Iteration 0.
Criterion value = -90.8662
Iteration 1.
Criterion value = -85.4065
Iteration 2.
Criterion value = -84.0944
Iteration 3.
Criterion value = -83.8882
Iteration 4.
Criterion value = -83.8774
Iteration 5.
Criterion value = -83.8774
Iteration 6.
Criterion value = -83.8774
Reason for termination: gradient size is less than gradient tolerance.


First Order Approximations to Posterior Moments:

Parameter 0                56.8489 (13.9713)
Parameter 1               0.481829 (0.179694)

#<Object: 1565592, prototype = BAYES-MODEL-PROTO>
>
```

It is possible to suppress the summary information by supplying NIL as the value of the :print keyword argument.

The summary printed by bayes-model gives first order approximations to the posterior means and standard deviations of the parameters. That is, the means are approximated by the elements of the posterior mode and the standard deviations by the square roots of the diagonal elements of the inverse of the negative Hessian matrix of the log posterior at the mode. These approximations can also be obtained from the model by sending it the :1stmoments message:

```
> (send lk :1stmoments)
((56.8489 0.481829) (13.9713 0.179694))
```

The result is a list of two lists, the means and the standard deviations. A slower but more accurate second order approximation is available as well. It can be obtained using the message :moments:

```
> (send lk :moments)
((65.3085 0.485295) (17.158 0.186587))
```

Both these messages allow you to compute moments for individual parameters or groups of parameters by specifying an individual parameter index or a list of indices:

```
> (send lk :moments 0)
((65.3085) (17.158))
> (send lk :moments (list 0 1))
((65.3085 0.485295) (17.158 0.186587))
```

The first and second order approximations to the moments of $\theta_0$ are somewhat different; in particular the mean appears to be somewhat larger than the mode. This suggests that the posterior distribution of this parameter is skewed to the right. We can confirm this by looking at a plot of the approximate marginal posterior density. The message :margin1 takes a parameter index, and a sequence of points at which to evaluate the density and returns as its value a list of the supplied sequence and the approximate density values at these points. This list can be given to plot-lines to produce a plot of the marginal density:

```
> (plot-lines (send lk :margin1 0 (rseq 30 120 30)))
#<Object: 1623804, prototype = SCATTERPLOT-PROTO>
```

The result is shown in Figure 20 and does indeed show some skewness to the right.

In addition to examining individual parameters it is also possible to look at the posterior distribution of smooth functions of the parameters.[29] For example, you might want to ask what information the data contains about the probability of a patient with $WBC = 50000$ surviving a year or more. This probability is given by

$$\frac{1}{\mu(x)} e^{-52/\mu(x)},$$

with

$$\mu(x) = \theta_0 e^{-\theta_1 x}$$

and $x = \log(5)$, and can be computed by the function

```
(defun lk-sprob (theta)
  (let* ((time 52.0)
         (x (log 5))
         (mu (* (select theta 0) (exp (- (* (select theta 1) x))))))
    (exp (- (/ time mu)))))
```

[29]The approximation methods assume these functions are twice continuously differentiable; thus they can not be indicator functions.

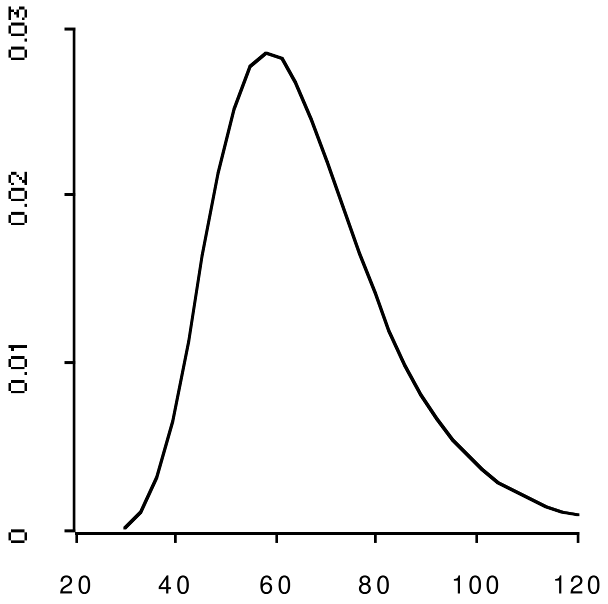


Figure 20: Plot of marginal posterior density for $\theta_0$.

This function can be given to the `:1stmoments`, `:moments` and `:margin1` methods to approximate the posterior moments and marginal posterior density of this function. For the moments the results are

```
> (send lk :1stmoments #'lk-sprob)
((0.137189) (0.0948248))
> (send lk :moments #'lk-sprob)
((0.184275) (0.111182))
```

with the difference again suggesting some positive skewness, and the marginal density produced by the expression

```
(plot-lines (send lk :margin1 #'lk-sprob (rseq .01 .8 30)))
```

is shown in Figure 21. Based on this picture the data suggests that this survival probability is almost certainly below 0.5, but it is difficult to make a more precise statement than that.

The functions described in this section are based on the optimization code described in the previous section. By default derivatives are computed numerically. If you can compute derivatives yourself you can have your log posterior function return a list of the function value and the gradient or a list of the function value, the gradient and the Hessian matrix.

## Exercises

1. To be able to think about prior distributions for the two parameters in this problem we need to try to understand what the parameters represent. The parameter $\theta_0$ is fairly easy to understand: it is the mean survival time for patients with $WBC = 10000$. The parameter $\theta_0$ is a little more difficult to think about. In represents the approximate percent difference in mean survival time for patients with $WBC$ differing by one percent. Because of the minus sign

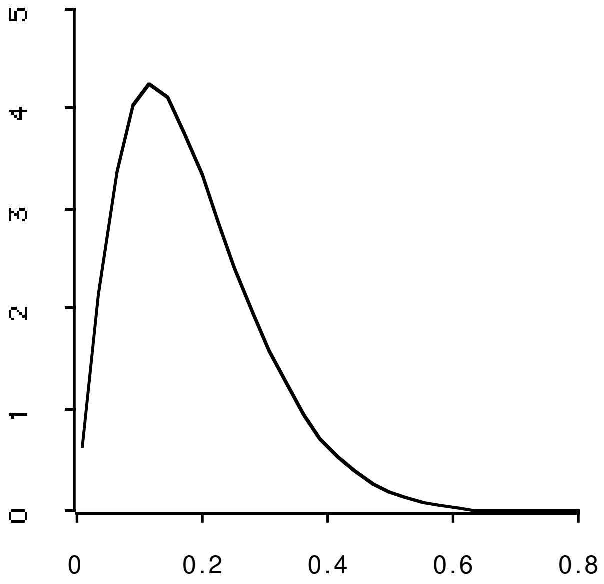


Figure 21: Plot of marginal posterior density of the one year survival probability of a patient with $WBC = 50000$.

in the mean relationship, and the expected inverse relation between $WBC$ and mean survival time, $\theta_1$ is expected to be positive.

Consider an informative prior distribution that assumes the two parameters *a priori* independent, takes $\log(\theta_0)$ to be normally distributed with mean $\log(52)$ and standard deviation $\log(2)$, and $\theta_1$ to be exponentially distributed with mean $\mu = 5$. This prior is designed to represent an opinion that mean survival time at $WBC = 10000$ should be around one year, but that guess could easily be off by a factor of two either way. The percentage change in the mean for a one percent change in $WBC$ should be on the order of one to ten or so. Examine the posterior distribution for this prior and compare your results to the results for the vague prior used above.

2. Construct and examine a posterior distribution for the parameters of the gamma model based on the aircraft data of Section 12.

# References


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

[2] Becker, Richard A., and Chambers, John M., (1984), *S: An Interactive Environment for Data Analysis and Graphics*, Belmont, Ca: Wadsworth.

[3] Becker, Richard A., Chambers, John M., and Wilks, Allan R., (1988), *The New S Language: A Programming Environment for Data Analysis and Graphics*, Pacific Grove, Ca: Wadsworth.

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

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

[6] Betz, David, (1988), "XLISP: An experimental object-oriented programming language," Reference manual for XLISP Version 2.0.

[7] Chaloner, Kathryn, and Brant, Rollin, (1988) "A Bayesian approach to outlier detection and residual analysis," *Biometrika*, vol. 75, pp. 651-660.

[8] Cleveland, W. S. and McGill, M. E., (1988) *Dynamic Graphics for Statistics*, Belmont, Ca.: Wadsworth.

[9] Cox, D. R. and Snell, E. J., (1981) *Applied Statistics: Principles and Examples*, London: Chapman and Hall.

[10] Dennis, J. E. and Schnabel, R. B., (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Englewood Cliffs, N.J.: Prentice-Hall.

[11] Devore, J. and Peck, R., (1986), *Statistics, the Exploration and Analysis of Data*, St. Paul, Mn: West Publishing Co.

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

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

[14] Press, Flannery, Teukolsky and Vetterling, (1988), *Numerical Recipes in C*, Cambridge: Cambridge University Press.

[15] Steele, Guy L., (1984), *Common Lisp: The Language*, Bedford, MA: Digital Press.

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

[17] Tierney, Luke, (1990) *LISP-STAT: Statistical Computing and Dynamic Graphics in Lisp.* Forthcoming.

[18] Tierney, L. and J. B. Kadane, (1986), "Accurate approximations for posterior moments and marginal densities," *J. Amer. Statist. Assoc.*, vol. 81, pp. 82-86.

[19] Tierney, Luke, Robert E. Kass, and Joseph B. Kadane, (1989), "Fully exponential Laplace approximations to expectations and variances of nonpositive functions," *J. Amer. Statist. Assoc.*, to appear.

[20] Tierney, L., Kass, R. E., and Kadane, J. B., (1989), "Approximate marginal densities for nonlinear functions," *Biometrika*, to appear.

[21] Weisberg, Sanford, (1982), "MULTREG Users Manual," Technical Report 298, School of Statistics, University of Minnesota.

[22] Winston, Patrick H. and Berthold K. P. Horn, (1988), *LISP*, 3rd Ed., New York: Addison-Wesley.

# A XLISP-STAT on UNIX Systems

This tutorial has dealt primarily with the Macintosh version of XLISP-STAT. XLISP-STAT is also available on UNIX systems. If it has been installed in a directory in your search path you should be able to start it up by typing

```
xlispstat
```

at the UNIX shell level. There are a few differences between the Macintosh and UNIX versions. On UNIX systems:

- UNIX versions of XLISP-STAT are designed to run on a standard terminal and therefore do not provide parenthesis matching or indentation support. If you use the *GNU* `emacs` editior you can obtain both these features by running XLISP-STAT from within `emacs`. Otherwise, for editing files with `vi` you can use the `-l` flag to get some Lisp editing support.
- To quit from the program type

  ```
  (exit)
  ```

  On most systems you can also quit by typing a *Control-D*.
- You can interrupt a calculation that is taking too long or was started in error by typing a *Control-C*.
- Data and example files are stored in the `Data` and `Examples` subdirectories of the library tree. The functions `load-data` and `load-examples` will look in these directories, so

  ```
  (load-data "tutorial")
  ```

  will load the data sets for the tutorial. Within XLISP-STAT the variable `*default-path*` shows the root directory of the library; you can look there if you want to examine the example files.
- The `require` function can be used to load program modules not loaded at startup. To load the nonlinear regression module, for example, use the expression

  ```
  (require "nonlin")
  ```

On basic UNIX systems the only graphics available are the functions `plot-points` and `plot-lines`. These functions assume you are using a *Tektronix* terminal or emulator.

## A.1 XLISP-STAT Under the *X11* Window System

Window based graphics are available in XLISP-STAT on a workstation running the *X11* window system. Graphics under *X11* are fairly similar to Macintosh graphics as documented in this tutorial. A few points to note:

- Plot menus are popped up using a menu button in the top right corner of a plot.
- In plot interaction you can use any of the mouse buttons. Normally clicking any button in a plot unselects all selected points. To extend a selection, or have a rotating plot continue rotating after the button is released, hold down the shift key as you press any mouse button.

- How plot windows are opened in response to a graphics command depends on the window manager you are using. Under `uwm`, the default window manager on many systems, a little corner will appear which you can use to choose the position of your plot window.
- Slider dialog items are the only items that assume a three button mouse. In the central part of the slider the right button increases and the left button decreases the slider value. The middle button drags the thumb indicator.
- Postscript images of plots can be saved by selecting the **Save to File** item on a plot menu. The postscript file can then be printed using a standard printing command.

### A.1.1 More Advanced *X11* Features

You can have XLISP-STAT use an alternate display for its graphics by setting the `DISPLAY` environment variable before starting XLISP-STAT. At present this is the only way to set an alternate display. You can specify alternate fonts and a few other options using the *X11* resource management facilities. Resources controlling appearance are

| | |
|---|---|
| `xlisp*menu*titles:` | *on* for a title on a menu, *off* otherwise |
| `xlisp*menu*font:` | |
| `xlisp*dialog*font:` | |
| `xlisp*graph*font:` | |

There are also a few experimental options controlling performance. These are

| | |
|---|---|
| `xlisp*graph*fastlines:` | *on* to use 0 width lines |
| `xlisp*graph*fastsymbols:` | *on* to use `DrawPoints` instead of bitmaps |
| `xlisp*graph*motionsync:` | *on* to use `XSync` during mouse motion |

By default all three options are are *on*. That seems to give the best performance on a *Sun 3/50*. It may not be the best choice on other workstations. You can also use the function `x11-options` to change these three options from within XLISP-STAT. The `fastlines` option will not take effect immediately when changed this way but will affect the next plot created. The other two options do take effect immediately.

## A.2 XLISP-STAT Under the *SunView* Window System

Window based graphics are also available when XLISP-STAT is run on a *Sun* console running `suntools`. Graphics under `suntools` work like graphics on the Macintosh with the following changes:

- To close or resize plots or dialogs use the frame menu or the standard `suntools` shortcuts (e. g. to resize a plot window drag the frame with the middle mouse button while holding down the *control* key).
- Plot menus are popped up by pressing the right mouse button in the interior of the plot. Check marks do not appear on menu items so it may not always be clear what state an item is in.
- Clicking and dragging on the Macintosh corresponds to clicking and dragging with the left mouse button. Shift-clicking or shift-dragging on the Macintosh (to extend a selection or cause a rotating plot to continue spinning when the button is released) corresponds to using the middle mouse button.
- Postscript images of plots can be saved by selecting the **Save to File** item on a plot menu.

## A.3 Running UNIX Commands from XLISP-STAT

The `system` function can be used to run UNIX commands from within XLISP-STAT. This function takes a shell command string as its argument and returns the shell exit code for the command. For example, you can print the date using the UNIX `date` command:

```
> (system "date")
Wed Jul 19 11:06:53 CDT 1989
0
>
```

The return value is 0, indicating successful completion of the UNIX command.

## A.4 Dynamic Loading and Foreign Function Calling

Some UNIX implementations of XLISP-STAT also provide a facility to allow you to use your own C functions of FORTRAN subroutines from within XLISP-STAT. The facility, patterned after the approach used in the New *S* language [3], consists of the function `dyn-load` for loading your code into a running XLISP-STAT process and the functions `call-cfun`, `call-fsub` and `call-lfun` for calling subroutines in the loaded code. The `dyn-load` function requires one argument, a string containing the name of a file to be linked and loaded. The file will be linked with standard C libraries before loading. If you need it to be linked with the standard FORTRAN libraries as well you can give the keyword argument `:fortran` the value `T`. Finally, if you need to link other libraries you can supply a string containing the library flags you would specify in a linking command as the value of the keyword argument `:libflags`. For example, to include the library `cmlib` use the string `"-lcmlib"`.[30]

The function `call-cfun` takes a string identifying the C function you want to call, followed by additional arguments for your C function. The additional arguments must be integers, sequences of integers, real numbers or sequences of real numbers. Pointers to `int` or `double` data containing these values will be passed to your routine. After your routine returns the contents of the data referred to by these pointers are copied into lists and `call-cfun` returns a list of these lists.

As an example, suppose the file `foo.c` contains the following C function:

```
foo(n, x, sum)
     int *n;
     double *x, *sum;
{
  int i;

  for (i = 0, *sum = 0.0; i < *n; i++) {
    *sum += x[i];
  }
}
```

After compiling the file to `foo.o` we can load it into XLISP-STAT using the expression

```
(dyn-load "foo.o")
```

We can then call the function `foo` using a list of real numbers as the second argument. The function `float` can be used to coerce numbers to reals:

```
> (call-cfun "foo" 5 (float (iseq 1 5)) 0.0)
((5) (1 2 3 4 5) (15))
```

[30] There may be slight differences in the implementation of `dyn-load` on different systems. The help information for this function should give information that is appropriate for your system.

The third argument to foo has been used to return the result.

The function call-fsub is used for calling FORTRAN subroutines that have been loaded dynamically. A FORTRAN subroutine analogous to the C function foo might be written as

```
      subroutine foo(n, x, sum)
      integer n
      double precision x(n), sum

      integer i

      sum = 0.0
      do 10 i = 1, n
         sum = sum + x(i)
 10   continue
      return
      end
```

After compiling and loading this routine it can be called using call-fsub:

```
> (call-fsub "foo" 5 (float (iseq 1 5)) 0.0)
((5) (1 2 3 4 5) (15))
```

Two C functions you may want to call from within your C functions are xscall_alloc and xscall_fail. The function xscall_alloc is like calloc, except it insures the allocated memory is garbage collected after the call to call-cfun returns. The function xscall_fail takes a character string as its argument. It prints the string and signals an error.

The function call-lfun can be used to call C functions written using the internal XLISP conventions for obtaining arguments and returning results. This allows you to accept any kinds of arguments. Unfortunately, the source code is the only documentation for the internal calling conventions.

A note of caution may be appropriate at this point. Dynamically loaded code contains only the error checking you build into it. If a function is not called with the proper arguments it will most likely cause XLISP-STAT to crash, losing any variables you have not saved.

At present the number of arguments you can pass to C functions or FORTRAN subroutines using call-cfun or call-fsubr is limited to 15.

If dynloading is not available on your system you can still recompile XLISP-STAT with files of your own added to the source code. The functions call-cfun, call-fsubr and call-lfun can be used to call your functions or subroutines in this case as well.

# B Graphical Interface Tools

One of the characteristic features of the Macintosh user interface is the use of menus and dialogs for interacting with the computer. XLISP-STAT allows you to construct your own menus and dialogs using Lisp commands. This appendix gives a very brief introduction to constructing menus and dialogs; further details will be given in [17]. A few of the explanations and examples in this appendix use Lisp concepts that have not been covered in this tutorial.

## B.1 Menus

As an illustration I will outline how to construct a menu for sending some simple messages to a regression model. I will make the convention that there is a *current regression model*, the value of the symbol `*current-model*`.

Menus are created by sending the `:new` message to the menu prototype, `menu-proto`. The method for this message takes a single argument, the menu title. We can use this message to set up our menu:

```
> (setf model-menu (send menu-proto :new "Model"))
#<Object: 4055334, prototype = MENU-PROTO>
```

Macintosh menus can be installed in and removed from the menu bar by sending them the `:install` and `:remove` messages:

```
> (send model-menu :install)
NIL
> (send model-menu :remove)
NIL
```

On other systems menus are *popped up*; this can be accomplished by sending the `:popup` message to a menu. This message requires two arguments, the $x$ and $y$ pixel coordinates of the left top corner of the menu, measured from the left top corner of the screen.

Initially the menu has no items in it. Items are created using the `menu-item-proto` prototype. The initialization method requires one argument, the item's title, and takes several keyword arguments, including

- `:action` – a function to be called when the item is selected
- `:enabled` – true by default
- `:key` – a character to serve as the keyboard equivalent
- `:mark` – `nil` (the default) or `t` to indicate a check mark.

Analogous messages are available for changing these values in existing menu items.

Suppose we would like to be able to use our menu to print a summary of the current model or obtain a residual plot. We can construct two menu items:

```
> (setf summary (send menu-item-proto :new "Summary" :action
                 #'(lambda () (send *current-model* :display))))
#<Object: 4034406, prototype = MENU-ITEM-PROTO>
> (setf plot (send menu-item-proto :new "Plot Residuals" :action
                 #'(lambda () (send *current-model* :plot-residuals))))
#<Object: 3868686, prototype = MENU-ITEM-PROTO>
```

Suppose we have assigned the `bikes2` model of Section 7 to `*current-model*`. You can force an item's action to be invoked by sending it the `:do-action` message from the listener:

```
> (send summary :do-action)

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

NIL
```

Ordinarily you will not send this message this way: the system sends this message to the menu item when you select the item from a menu.

To add these items to the menu use the `:append-items` message:

```
> (send model-menu :append-items summary plot)
NIL
```

You can also use the `:append-items` message to add items to a plot menu. The menu associated with a plot can be obtained by sending the plot the `:menu` message with no arguments.

You can enable and disable a menu item with the `:enabled` message:

```
> (send summary :enabled)
T
> (send summary :enabled nil)
NIL
> (send summary :enabled t)
T
```

## B.2 Dialogs

Dialogs are similar to menus in that they are based on a dialog prototype and dialog item prototypes. There are, however many more variations. Fortunately most dialogs you need fall into one of several categories and can be produced by custom dialog construction functions.

### B.2.1 Modal Dialogs

Modal dialogs are designed to ask specific questions and wait until they receive a response. All other interaction is disabled until the dialog is dismissed – they place the system in *dialog mode.* Six functions are available for producing some standard modal dialogs:

- `(message-dialog` <*string*>`)` – presents a message with an **OK** button; returns `nil` when the button is pressed.
- `(ok-or-cancel-dialog` <*string*>`)` – presents a message with an **OK** and a **Cancel** button; returns `t` or `NIL` according to the button pressed.
- `(choose-item-dialog` <*string*> <*string-list*>`)` – presents a heading and a set of radio buttons for choosing one of the strings. Returns the index of the selected string on **OK** or `nil` on **Cancel**. Example:

```
> (choose-item-dialog "Dependent variable:" '("X" "Y" "Z"))
1
```

- `(choose-subset-dialog <string> <string-list>)` – presents a heading and a set of check boxes for indicating which items to select. Returns a list of the list of selected indices on **OK** or `nil` on **Cancel**. Example:

```
> (choose-subset-dialog "Independent variables:" '("X" "Y" "Z"))
((0 2))
```

- `(get-string-dialog <prompt> [:initial <expr>])` – presents a dialog with a prompt, an editable text field, an **OK** and a **Cancel** button. The initial contents of the editable field is empty or the `princ` formated version of <*expr*>. The result is a string or `nil`. Example:

```
> (get-string-dialog "New variable label:" :initial "X")
"Tensile Strength"
```

- `(get-value-dialog <prompt> [:initial <expr>])` – like `get-string-dialog`, except
  - the initial value expression is converted to a string with `print` formatting
  - the result is interpreted as a lisp expression and is evaluated
  - the result is a list of the value, or `nil`

On the Macintosh there are two additional dialogs for dealing with files:

- `(open-file-dialog)` – presents a standard **Open File** dialog and returns a file name string or `nil`. Resets the working folder on **OK**.
- `(set-file-dialog prompt)` – presents a standard **Save File** dialog. Returns a file name string or `nil`. Resets the working folder on **OK**.

### B.2.2 Modeless Dialogs

Two functions for constructing custom modeless dialogs are available also. They are the functions `interval-slider-dialog` and `sequence-slider-dialog` introduced above in Section 8.

# C Selected Listing of XLISP-STAT Functions

## C.1 Arithmetic and Logical Functions

(+ &rest numbers) [Function]
Returns the sum of its arguments. With no args, returns 0. Vectorized.

(- number &rest more-numbers) [Function]
Subtracts the second and all subsequent NUMBERs from the first. With one arg, negates it. Vectorized.

(* &rest numbers) [Function]
Returns the product of its arguments. With no args, returns 1. Vectorized.

(/ number &rest more-numbers) [Function]
Divides the first NUMBER (element-wise) by each of the subsequent NUMBERS. With one arg, returns its reciprocal. Vectorized.

(^ base-number power-number) [Function]
Returns BASE-NUMBER raised to the power POWER-NUMBER. Vectorized.

(** base-number power-number) [Function]
Returns BASE-NUMBER raised to the power POWER-NUMBER. Vectorized.

(< &rest numbers) [Function]
Returns T if NUMBERS are in strictly increasing order; NIL otherwise. Vectorized.

(<= &rest numbers) [Function]
Returns T if NUMBERS are in nondecreasing order; NIL otherwise. Vectorized.

(= &rest numbers) [Function]
Returns T if NUMBERS are all equal; NIL otherwise. Vectorized.

(/= &rest numbers) [Function]
Returns T if NUMBERS no two adjacent numbers are equal; NIL otherwise. Vectorized.

(>= &rest numbers) [Function]
Returns T if NUMBERS are in nonincreasing order; NIL otherwise. Vectorized.

(> &rest numbers) [Function]
Returns T if NUMBERS are in strictly decreasing order; NIL otherwise. Vectorized.

(abs number) [Function]
Returns the absolute value or modulus of NUMBER. Vectorized.

(acos number) [Function]
Returns the arc cosine of NUMBER. Vectorized.

(asin number) [Function]
Returns the arc sine of NUMBER. Vectorized.

(atan number) [Function]
Returns the arc tangent of NUMBER. Vectorized.

(ceiling number) [Function]
Returns the smallest integer(s) not less than or NUMBER. Vectorized.

(complex realpart &optional (imagpart 0)) [Function]
Returns a complex number with the given real and imaginary parts.

(conjugate number) [Function]
Returns the complex conjugate of NUMBER.

(cos radians) [Function]
Returns the cosine of RADIANS. Vectorized.

(exp x) [Function]
Calculates e raised to the power x, where e is the base of natural logarithms. Vectorized.

(expt base-number power-number) [Function]
Returns BASE-NUMBER raised to the power POWER-NUMBER. Vectorized.

(float number) [Function]
Converts real number to a floating-point number. If NUMBER is already a float, FLOAT simply returns NUMBER. Vectorized.

(floor number) [Function]
Returns the largest integer( not larger than the NUMBER. Vectorized.

(imagpart number) [Function]
Extracts the imaginary part of NUMBER.

(log number) [Function]
Returns the natural logarithm(s) of NUMBER. Vectorized.

(log-gamma x) [Function]
Returns the log gamma function of X. Vectorized.

(max number &rest more-numbers) [Function]
Returns the greatest of its arguments. Vector reducing

(min number &rest more-numbers) [Function]
Returns the least of its arguments. Vector reducing

(phase number) [Function]
Returns the angle part of the polar representation of a complex number. For non-complex numbers, this is 0.

(pmax &rest items) [Function]
Parallel maximum of ITEMS. Vectorized.

(pmin &rest items) [Function]
Parallel minimum of ITEMS. Vectorized.

(prod &rest number-data) [Function]
Returns the product of all the elements of its arguments. Returns 1 if there are no arguments. Vector reducing.

(random number) [Function]
Generates a uniformly distributed pseudo-random number between zero (inclusive) and NUMBER (exclusive). Vectorized.

(realpart number) [Function]
Extracts the real part of NUMBER.

(rem x y) [Function]
Returns the remainder of dividing x by y. Vectorized.

(round number) [Function]
Rounds NUMBER to nearest integer. Vectorized.

(sin radians) [Function]
Returns the sine of RADIANS. Vectorized.

(sqrt number) [Function]
Returns the square root of NUMBER. Vectorized.

(sum &rest number-data) [Function]
Returns the sum of all the elements of its arguments. Returns 0 if there are no arguments. Vector reducing.

(tan radians) [Function]
Returns the tangent of RADIANS. Vectorized.

(truncate number) [Function]
Returns real NUMBER as an integer, rounded toward 0. Vectorized.

## C.2 Constructing and Modifying Compound Data and Variables

(def var form) [Macro]
VAR is not evaluated and must be a symbol. Assigns the value of FORM to VAR and adds VAR to the list *VARIABLES* of def'ed variables. Returns VAR. If VAR is already bound and the global variable *ASK-ON-REDEFINE* is not nil then you are asked if you want to redefine the variable.

(if-else first x y) [Function]
Takes simple or compound data items FIRST, X and Y and returns result of elementswise selecting from X if FIRST is not NIL and from Y otherwise.

(iseq n m) [Function]
Returns a list of consecutive integers from n to m. Examples:
(iseq 3 7) returns (3 4 5 6 7)
(iseq 3 -3) returns (3 2 1 0 -1 -2 -3)

(list &rest args) [Function]
Returns a list of its arguments

(repeat vals times) [Function]
Repeats VALS. If TIMES is a number and VALS is a non-null, non-array atom, a list of length TIMES with all elements eq to VALS is returned. If VALS is a list and TIMES is a number then VALS is appended TIMES times. If TIMES is a list of numbers then VALS must be a list of equal length and the simpler version of repeat is mapped down the two lists. 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)
(repeat '((4) (5 6)) '(2 3)) returns (4 4 5 6 5 6 5 6)

(rseq a b num) [Function]
Returns a list of NUM equally spaced points starting at A and ending at B.

(select a &rest indices) [Function]
A can be a list or an array. If A is a list and INDICES is a single number then the appropriate element of A is returned. If is a list and INDICES is a list of numbers then the sublist of the corresponding elements is returned. If A in an array then the number of INDICES must match the ARRAY-RANK of A. If each index is a number then the appropriate array element is returned. Otherwise the INDICES must all be lists of numbers and the corresponding submatrix of A is returned. SELECT can be used in setf.

(undef symbol) [Function]
If SYMBOL is a defined variable it is unbound and removed from the list of defined variables and returns SYMBOL.

(vector &rest items) [Function]
Returns a vector with ITEMS as elements.

(which x) [Function]
X is an array or a list. Returns a list of the indices where X is not NIL.

## C.3 Basic Statistical Functions

(bayes-model logpost mode &key scale data derivstep (verbose t) (quick t) (print t)) [Function]
LOGPOST computes the logposterior density. It should return the function, or a list of the function value and gradient, or a list of the function value, gradient and Hessian. MODE is an initial guess for the mode. SCALE and DERIVSTEP are used for numerical derivatives and scaling. VERBOSE controls printing of iteration information during optimization, PRINT controls printing of summary information. If QUICK is T the summary is based on first order approximations.

(beta-cdf x alpha beta) [Function]
Returns the value of the Beta(ALPHA, BETA) distribution function at X. Vectorized.

(beta-dens x alpha beta) [Function]
Returns the density at X of the Beta(ALPHA, BETA) distribution. Vectorized.

(beta-quant p alpha beta) [Function]
Returns the P-th quantile of the Beta(ALPHA, BETA) distribution. Vectorized.

(beta-rand n a b) [Function]
Returns a list of N beta(A, B) random variables. Vectorized.

(binomial-cdf x n p) [Function]
Returns value of the Binomial(N, P) distribution function at X. Vectorized.

(binomial-pmf k n p) [Function]
Returns value of the Binomial(N, P) pmf function at integer K. Vectorized.

(binomial-quant x n p) [Function]
Returns x-th quantile (left continuous inverse) of Binomial(N, P) cdf. Vectorized.

(binomial-rand k n p) [Function]
Returns list of K draws from the Binomial(N, P) distribution. Vectorized.

(bivnorm-cdf x y r) [Function]
Returns the value of the standard bivariate normal distribution function with correlation R at (X, Y). Vectorized.

(cauchy-cdf x) [Function]
Returns the value of the standard Cauchy distribution function at X. Vectorized.

(cauchy-dens x) [Function]
Returns the density at X of the standard Cauchy distribution. Vectorized.

(cauchy-quant p) [Function]
Returns the P-th quantile(s) of the standard Cauchy distribution. Vectorized.

(cauchy-rand n) [Function]
Returns a list of N standard Cauchy random numbers. Vectorized.

(chisq-cdf x df) [Function]
Returns the value of the Chi-Square(DF) distribution function at X. Vectorized.

(chisq-dens x alpha) [Function]
Returns the density at X of the Chi-Square(DF) distribution. Vectorized.

(chisq-quant p df) [Function]
Returns the P-th quantile of the Chi-Square(DF) distribution. Vectorized.

(chisq-rand n df) [Function]
Returns a list of N Chi-Square(DF) random variables. Vectorized.

(covariance-matrix &rest args) [Function]
Returns the sample covariance matrix of the data columns in ARGS. ARGS may consist of lists, vectors or matrices.

(difference x) [Function]
Returns differences for a sequence X.

(f-cdf x ndf ddf) [Function]
Returns the value of the F(NDF, DDF) distribution function at X. Vectorized.

(f-dens x ndf ddf) [Function]
Returns the density at X of the F(NDF, DDF) distribution. Vectorized.

(f-quant p ndf ddf) [Function]
Returns the P-th quantile of the F(NDF, DDF) distribution. Vectorized.

(f-rand n d) [Function]
Returns a list of N F(NDF, DDF) random variables. Vectorized.

(fivnum number-data) [Function]
Returns the five number summary (min, 1st quartile, medinan, 3rd quartile, max) of the elements X.

(gamma-cdf x alpha) [Function]
Returns the value of the Gamma(alpha, 1) distribution function at X. Vectorized.

(gamma-dens x alpha) [Function]
Returns the density at X of the Gamma(ALPHA, 1) distribution. Vectorized.

(gamma-quant p alpha) [Function]
Returns the P-th quantile of the Gamma(ALPHA, 1) distribution. Vectorized.

(gamma-rand n a) [Function]
Returns a list of N Gamma(A, 1) random variables. Vectorized.

(interquartile-range number-data) [Function]
Returns the interquartile range of the elements of X.

(mean x) [Function]
Returns the mean of the elements x. Vector reducing.

(median x) [Function]
Returns the median of the elements of X.

(newtonmax f start &key scale derivstep (verbose 1) return-derivs) [Function]
Maximizes F starting from START using Newton's method with backtracking. If RETURN-DERIVS is NIL returns location of maximum; otherwise returns list of location, unction value, gradient and hessian at maximum. SCALE should be a list of the typical magnitudes of the parameters. DERIVSTEP is used in numerical derivatives and VERBOSE controls printing of iteration information. COUNT-LIMIT limits the number of iterations

(nelmeadmax f start &key (size 1) (epsilon (sqrt machine-epsilon)) (count-limit 500) (verbose t) alpha beta gamma delta) [Function]
Maximizes F using the Nelder-Mead simplex method. START can be a starting simplex - a list of N+1 points, with N=dimension of problem, or a single point. If start is a single point you should give the size of the initial simplex as SIZE, a sequence of length N. Default is all 1's. EPSILON is the convergence tolerance. ALPHA-DELTA can be used to control the behavior of simplex algorithm.

(normal-cdf x) [Function]
Returns the value of the standard normal distribution function at X. Vectorized.

(normal-dens x) [Function]
Returns the density at X of the standard normal distribution. Vectorized.

(normal-quant p) [Function]
Returns the P-th quantile of the standard normal distribution. Vectorized.

(normal-rand n) [Function]
Returns a list of N standard normal random numbers. Vectorized.

(nreg-model mean-function y theta &key (epsilon 0.0001) (count-limit 20) (print t) parameter-names response-name case-labels weights included (vetbose print)) [Function]
Fits nonlinear regression model with MEAN-FUNCTION and response Y using initial parameter guess THETA. Returns model object.

(numgrad f x &optional scale derivstep) [Function]
Computes the numerical gradient of F at X.

(numhess f x &optional scale derivstep) [Function]
Computes the numerical Hessian matrix of F at X.

(oneway-model data &key (print t)) [Function]
DATA: list of compound-data Example:

(order x) [Function]
Returns a sequence of the indices of elements in the sequence of numbers or strings X in order.

(pmin &rest items) [Function]
Parallel minimum of ITEMS. Vectorized.

(pmax &rest items) [Function]
Parallel maximum of ITEMS. Vectorized.

(poisson-cdf x mu) [Function]
Returns value of the Poisson(MU) distribution function at X. Vectorized.

(poisson-pmf k mu) [Function]
Returns value of the Poisson(MU) pmf function at integer K. Vectorized.

(poisson-quant x mu) [Function]
Returns x-th quantile (left continuous inverse) of Poisson(MU) cdf. Vectorized.

(poisson-rand k mu) [Function]
Returns list of K draws from the Poisson(MU) distribution. Vectorized.

(quantile x p) [Function]
Returns the P-th quantile(s) of sequence X. P can be a number or a sequence.

(rank x) [Function]
Returns a sequence with the elements of the list or array of numbers or strings X replaced by their ranks.

(read-data-columns file cols) [Function]
Reads the data in FILE as COLS columns and returns a list of lists representing the columns.

(read-data-file file) [Function]
Returns a list of all lisp objects in FILE. FILE can be a string or a symbol, in which case the symbol'f print name is used.

(regression-model x y &key (intercept t) (print t) weights included predictor-names response-name case-labels) [Function]
X - list of independent variables or X matrix
Y - dependent variable
INTERCEPT - T to include (default), NIL for no intercept
PRINT - if not NIL print summary information
WEIGHTS - if supplied should be the same length as Y; error variances are
assumed to be inversely proportional to WEIGHTS
PREDICTOR-NAMES
RESPONSE-NAME
CASE-LABELS - sequences of strings or symbols
INCLUDED - if supplied should be the same length as Y, with elements nil
to skip a in computing estimates (but not in residual analysis)
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 sample data directory/folder):
(def m (regression-model (list iron aluminum) absorbtion))
(send m :help)
(send m :plot-residuals)

(sort-data sequence) [Function]
Returns a sequence with the numbers or strings in the sequence X in order.

(standard-deviation x) [Function]
Returns the standard deviation of the elements x. Vector reducing.

(t-cdf x df) [Function]
Returns the value of the T(DF) distribution function at X. Vectorized.

(t-dens x alpha) [Function]
Returns the density at X of the T(DF) distribution. Vectorized.

(t-quant p df) [Function]
Returns the P-th quantile of the T(DF) distribution. Vectorized.

(t-rand n d) [Function]
Returns a list of N T(DF) random variables. Vectorized.

(uniform-rand n) [Function]
Returns a list of N uniform random variables from the range (0, 1). Vectorized.

## C.4 Plotting Functions

(boxplot data &key (title "Box Plot")) [Function]
DATA is a sequence, a list of sequences or a matrix. Makes a boxplot of the sequence or a parallel box plot of the sequences in the list or the columns of the matrix.

(boxplot-x x data &key (title "Box Plot")) [Function]
DATA is a list of sequences or a matrix. X is a sequence with as many elements as DATA has elements or columns. Makes a parallel box plot of the sequences in the list or the columns of the matrix vs X.

(close-all-plots) [Function]
Close all plot windos.

(histogram data &key (title "Histogram")) [Function]
Opens a window with a histogram of DATA. TITLE is the window title. The number of bins used can be adjusted using the histogram menu. The histogram can be linked to other plots with the link-views command. Returns a plot object.

(link-views &rest plots) [Function]
Links the argument plots: any change in hiliting or visibility of points in the current plot is propagated to the other plots.

(name-list names &key (title "Name List")) [Function]
NAMES is a number or a list of character strings. Opens a window with a list of the supplied character strings or entries numbered from 0 to NAMES - 1. This display can be linked to plots with the link-views function. Returns a plot object.

(plot-function f xmin xmax &optional (num-points 50)) [Function]
Plots function F of one real variable over the range between xmin and xmax. The function is evaluated at NUM-POINTS points.

(plot-lines x y &key (title "Line Plot") variable-labels type width color) [Function]
Opens a window with a connected line plot of X vs Y, where X and Y are compound number-data. VARIABLE-LABELS, if supplied, should be lists of character strings. TITLE is the window title. The plot can be linked to other plots with the link-views command. Returns a plot object.

(plot-points x y &key (title "Scatter Plot") variable-labels point-labels symbol color) [Function]
Opens a window with a scatter plot of X vs Y, where X and Y are compound number-data. VARIABLE-LABELS and POINT-LABELS, if supplied, should be lists of character strings. TITLE is the window title. The plot can be linked to other plots with the link-views command. Returns a plot object.

(probability-plot data &key (distribution-function (function normal-cdf)) (title "Probability Plot") point-labels) [Function] (quantile-plot data &key (quantile-function (function normal-quant)) (title "Quantile Plot") point-labels) [Function]

(scatterplot-matrix data &key (title "Spinning Plot") variable-labels point-labels (scale t)) [Function]
DATA is a list of two or more compound number-data objects of equal length. Opens a window with a brushable scatter plot matrix of the elements of DATA. VARIABLE-LABELS and POINT-LABELS, if supplied, should be lists of character strings. TITLE is the window title. If scale is NIL data are assumed to be between -1 and 1.The plot can be linked to other plots with the link-views command. Returns a plot object.

(spin-function f xmin xmax ymin ymax &optional (num-points 6)) [Function]
Rotatable plot of function F of two real variables over the range between [xmin, xmax] x [ymin, ymax]. The function is evaluated at NUM-POINTS points.

(spin-plot data &key (title "Spinning Plot") variable-labels point-labels (scale t)) [Function]
DATA is a list of three compound number-data objects of equal length. Opens a window with a rotating plot of the three elements of DATA. VARIABLE-LABELS and POINT-LABELS, if supplied, should be lists of character strings. TITLE is the window title. If scale is NIL data are assumed to be between -1 and 1. The plot can be linked to other plots with the link-views command. Returns a plot object.

(unlink-views &rest plots) [Function]
Removes links to its arguments. With no arguments removes all links.

## C.5 Object Methods

### C.5.1 Regression Methods

:basis [Object Method]
Returns the indices of the variables used in fitting the model.

:coef-estimates [Object Method]
Returns the OLS (ordinary least squares) estimates of the regression coefficients. Entries beyond the intercept correspond to entries in basis.

:coef-standard-errors [Object Method]
Returns estimated standard errors of coefficients. Entries beyond the intercept correspond to entries in basis.

:cooks-distances [Object Method]
Computes Cook's distances.

:df [Object Method]
Returns the number of degrees of freedom in the model.

:display [Object Method]
Prints the least squares regression summary. Variables not used in the fit are marked as aliased.

:fit-values [Object Method]
Returns the fitted values for the model.

:included &optional new-included [Object Method]
With no argument, NIL means a case is not used in calculating estimates, and non-nil means it is used. NEW-INCLUDED is a sequence of length of y of nil and t to select cases. Estimates are recomputed.

:intercept &optional new-intercept [Object Method]
With no argument returns T if the model includes an intercept term, nil if not. With an argument NEW-INTERCEPT the model is changed to include or exclude an intercept, according to the value of NEW-INTERCEPT.

:leverages [Object Method]
Returns the diagonal elements of the hat matrix.

:num-cases [Object Method]
Returns the number of cases in the model.

:num-coefs [Object Method]
Returns the number of coefficients in the fit model (including the intercept if the model includes one).

:num-included [Object Method]
Returns the number of cases used in the computations.

:plot-bayes-residuals &optional x-values [Object Method]
Opens a window with a plot of the standardized residuals and two standard error bars for the posterior distribution of the actual deviations from the line. See Chaloner and Brant. 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.

:plot-residuals &optional x-values [Object Method]
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.

:predictor-names &optional (names nil set) [Object Method]
With no argument returns the predictor names. NAMES sets the names.

:r-squared [Object Method]
Returns the sample squared multiple correlation coefficient, R squared, for the regression.

:raw-residuals [Object Method]
Returns the raw residuals for a model.

:residuals [Object Method]
Returns the raw residuals for a model without weights. If the model includes weights the raw residuals times the square roots of the weights are returned.

:sigma-hat [Object Method]
Returns the estimated standard deviation of the deviations about the regression line.

:studentized-residuals [Object Method]
Computes the internally studentized residuals for included cases and externally studentized residuals for excluded cases.

:sum-of-squares [Object Method]
Returns the error sum of squares for the model.

:weights &optional new-w [Object Method]
With no argument returns the weight sequence as supplied to m; NIL means an unweighted model. NEW-W sets the weights sequence to NEW-W and recomputes the estimates.

:x-matrix [Object Method]
Returns the X matrix for the model, including a column of 1's, if appropriate. Columns of X matrix correspond to entries in basis.

:xtxinv [Object Method]
Returns $(X^TX)^{-1}$ or $(X^TWX)^{-1}$.

### C.5.2 General Plot Methods

:add-lines lines &key type (draw t) [Object Method]
Adds lines to plot. LINES is a list of sequences, the coordinates of the line starts. TYPE is normal or dashed. If DRAW is true the new lines are added to the screen.

:add-points points &key point-labels (draw t) [Object Method]
Adds points to plot. POINTS is a list of sequences, POINT-LABELS a list of strings. If DRAW is true the new points are added to the screen.

:adjust-to-data &key (draw t) [Object Method]
Sets ranges to the actual range of variables in the original coordinate system. If DRAW is true sends :RESIZE and :REDRAW messages.

:all-points-showing-p [Object Method]

:all-points-unmasked-p [Object Method]

:any-points-selected-p [Object Method]

:apply-transformation a &key draw [Object Method]
Applies matrix A to current transformation. If draw is true the :REDRAW-CONTENT message is sent.

:clear &key (draw t) [Object Method]
Clears the plot data. If DRAW is nil the plot is redrawn; otherwise its current screen image remains unchanged.

:clear-lines &key (draw t) [Object Method]
Removes all lines from the plot. If DRAW is true the :REDRAW-CONTENT message is sent.

:clear-points &key (draw t) [Object Method]
Removes all points from the plot. If DRAW is true the :REDRAW-CONTENT message is sent.

:clear-strings &key (draw t) [Object Method]
Removes all strings from the plot. If DRAW is true the :REDRAW-CONTENT message is sent.

:content-variables &optional xvar yvar [Object Method]
Sets or retrieves the indices of the current content variables.

:do-mouse x y type extend option [Object Method]
Sends appropriate action message for mouse mode to plot.

:drag-grey-rect x y width height [Object Method]
Drags grey rectangle starting at (LIST (- X WIDTH) (- Y HEIGHT) WIDTH HEIGHT) while mouse button is down. Returns the final rectangle. Should be called when the mouse is down.

:erase-selection [Object Method]
Sets selected points states to invisible and sends :ADJUST-SCREEN message.

:fixed-aspect &optional fixed [Object Method]
Sets or retrieves current size adjustment option (true or NIL).

:frame-location &optional left top [Object Method]
Moves window frame to (LEFT TOP) if supplied. Returns list of current left, top. Adjusts for the menu bar.

:frame-size &optional width height [Object Method]
Sets window frame width and size to WIDTH and SIZE if supplied. Returns list of current WIDTH and HEIGHT. Adjusts for the menu bar.

:idle-on &optional on [Object Method]
Sets or returns idling state. On means :do-idle method is sent each pass through the event loop.

:linked &optional on [Object Method]
Sets or retrieves plot's linking state.

:num-lines [Object Method]
Returns the number of line starts in the plot.

:num-points [Object Method]
Returns the number of points in the plot.

:num-strings [Object Method]
Returns the number of strings in the plot.

:num-variables [Object Method]
Returns the number of variables in the plot.

:point-coordinate var point &optional value [Object Method]
Sets or retrieves coordinate for variable VAR and point POINT in the original coordinate system. Vectorized.

:point-hilited point &optional hilited [Object Method]
Sets or returns highlighting status (true or NIL) of POINT. Sends :ADJUST-SCREEN message if states are set. Vectorized.

:point-label point &optional label [Object Method]
Sets or retrieves label of point POINT. Vectorized.

:point-selected point &optional selected [Object Method]
Sets or returns selection status (true or NIL) of POINT. Sends :ADJUST-SCREEN message if states are set. Vectorized.

:point-showing point &optional selected [Object Method]
Sets or returns visibility status (true or NIL) of POINT. Sends :ADJUST-SCREEN message if states are set. Vectorized.

:point-symbol point &optional symbol [Object Method]
Sets or retrieves symbol of point POINT. Vectorized.

:range index &optional low high [Object Method]
Sets or retrieves variable's original coordinate range. Vectorized.

:real-to-screen x y [Object Method]
Returns list of screen coordinates of point (X, Y), in the original coordinate system, based on current content variables.

:redraw [Object Method]
Redraws entire plot.

:redraw-content [Object Method]
Redraws plot's content.

:rotate-2 var1 var2 angle &key (draw t) [Object Method]
Rotates int the plane of variables with indices VAR1 and VAR2 by ANGLE, in radians. sends the :REDRAW-CONTENT message if DRWA is true.

:scale-to-range var low high &key (draw t) [Object Method]
Scales and shifts data to map visible range into specified range. Sends :RESIZE and :REDRAW messages if DRAW is true.

:scaled-range index &optional low high [Object Method]
Sets or retrieves variable's transformed coordinate range. Vectorized.

:screen-to-real x y [Object Method]
Returns list of real coordinates, in the original coordinate system, of screen point (X, Y), based on current content variables.

:selection [Object Method]
Return indices of current selection.

:show-all-points [Object Method]
Sets all point states to normal and sends :ADJUST-SCREEN message

:showing-labels &optional showing [Object Method]
Sets or retrieves current labeling state (true or NIL).

:title &optional string [Object Method]
Sets or retrieves window title.

:transformation &optional a &key (draw t) [Object Method]
Sets or retrieves transformation. A should be a matrix or NIL. If draw is true the :REDRAW-CONTENT message is sent.

:unselect-all-points &key (draw t) [Object Method]
Unselects all points. Sends :ADJUST-SCREEN message if DRAW is true.

:variable-label var &optional label [Object Method]
Sets or returns label for variable with index VAR. Vectorized.

:visible-range var [Object Method]
Returns list of min and max of variable VAR over visible, unmasked points, lines and strings. Vectorized.

:while-button-down fcn &optional (motion-only t) [Object Method]
Calls fcn repeatedly while mouse button is down. FCN should take two arguments, the current x and y coordinates of the mouse. Returns NIL. Should be called when button is already down.

:x-axis &optional showing labeled ticks [Object Method]
Sets or retrieves current acis label state. SHOWING and LABELED should be true or NIL; TICKS should be a number. All three should be supplied for setting a new state. A list of the three properties is returned.

:y-axis &optional showing labeled ticks [Object Method]
Sets or retrieves current acis label state. SHOWING and LABELED should be true or NIL; TICKS should be a number. All three should be supplied for setting a new state. A list of the three properties is returned.

### C.5.3 Histogram Methods

:add-points points (draw t) [Object Method]
Adds points to plot. POINTS is a sequence or a list of sequences. If DRAW is true the new points are added to the screen.

:num-bins &optional bins &key (draw t) [Object Method]
Sets or retrieves number of bins in the histogram. Sends :REDRAW-CONTENT message if DRAW is true.

### C.5.4 Name List Methods

:add-points points &key point-labels (draw t) [Object Method]
Adds points to plot. POINTS is a number or a list of sequences, POINT-LABELS a list of strings. If DRAW is true the new points are added to the screen.

### C.5.5 Scatterplot Methods

:abline a b [Object Method]
Adds the graph of the line A + B x to the plot.

:add-lines lines &key type (draw t) [Object Method]
Adds lines to plot. LINES is a list of sequences, the coordinates of the line starts. TYPE is normal or dashed. If DRAW is true the new lines are added to the screen.

:add-points points &key point-labels (draw t) [Object Method]
Adds points to plot. POINTS is a list of sequences, POINT-LABELS a list of strings. If DRAW is true the new points are added to the screen.

:add-strings locations strings [Object Method]
Adds strings to plot. LOCATIONS is a list of sequences, the coordinates of the strings. If DRAW is true the new lines are added to the screen.

### C.5.6 Spin Plot Methods

:abcplane a b c [Object Method]
Adds the graph of the plane A + B x + Cy to the plot.

:add-function [Object Method]
surface of function F over a NUM-POINTS by NUM-POINTS grid on the rectangle [xmin, xmax] x [ymin, ymax]. Passes other keywords to :add-surface method.

:add-surface [Object Method]
a grid surface using sequences X, Y with values in the matrix Z. Z should be (length X) by (length Y).

:angle &optional angle [Object Method]
Sets or retrieves current rotation angle, in radians.

:content-variables &optional xvar yvar [Object Method]
Sets or retrieves the indices of the current content variables.

:depth-cuing &optional cuing [Object Method]
Sets or retrieves depth cuing status (true or NIL).

:do-idle [Object Method]
Sends :ROTATE message.

:rotate [Object Method]
Rotates once in the current plane by the current angle.

:showing-axes &optional cuing [Object Method]
Sets or retrieves axis showing status (true or NIL).

## C.6 Some Useful Array and Linear Algebra Functions

(%* a b) [Function]
Returns the matrix product of matrices a and b. If a is a vector it is treated as a row vector; if b is a vector it is treated as a column vector.

(aref array &rest subscripts) [Function]
Returns the element of ARRAY specified by SUBSCRIPTS.

(array-dimension array) [Function]
Returns a list whose elements are the dimensions of ARRAY

(array-dimensions array) [Function]
Returns a list whose elements are the dimensions of ARRAY

(array-in-bounds-p array &rest subscripts) [Function]
Returns T if SUBSCRIPTS are valid subscripts for ARRAY; NIL otherwise.

(array-rank array) [Function]
Returns the number of dimensions of ARRAY.

(array-row-major-index array &rest subscripts) [Function]
Returns the index into the data vector of ARRAY for the element of ARRAY specified by SUBSCRIPTS.

(array-total-size array) [Function]
Returns the total number of elements of ARRAY.

(arrayp x) [Function]
Returns T if X is an array; NIL otherwise.

(bind-columns &rest args) [Function]
The ARGS can be matrices, vectors, or lists. Arguments are bound into a matrix along their columns. Example: (bind-columns #2a((1 2)(3 4)) #(5 6)) returns #2a((1 2 5)(3 4 6))

(bind-rows &rest args) [Function]
The ARGS can be matrices, vectors, or lists. Arguments are bound into a matrix along their rows. Example: (bind-rows #2a((1 2)(3 4)) #(5 6)) returns #2a((1 2)(3 4)(5 6))

(chol-decomp a) [Function]
Modified Cholesky decomposition. $A$ should be a square, symmetric matrix. Computes lower triangular matrix $L$ such that $LL^T = A + D$ where $D$ is a diagonal matrix. If $A$ is strictly positive definite $D$ will be zero. Otherwise $D$ is as small as possible to make $A + D$ numerically strictly positive definite. Returns a list $(L(maxD))$.

(column-list m) [Function]
Returns a list of the columns of M as vectors

(copy-array array) [Function]
Returns a copy of ARRAY with elements eq to the elements of ARRAY.

(copy-list list) [Function]
Returns a new copy of LIST.

(copy-vector vector) [Function]
Returns a copy of VECTOR with elements eq to the elements of VECTOR

(count-elements number &rest more-numbers) [Function]
Returns the number of its arguments. Vector reducing

(cross-product x) [Function]
If X is a matrix returns (matmult (transpose X) X). If X is a vector returns (inner-product X X).

(determinant m) [Function]
Returns the determinant of the square matrix M.

(diagonal x) [Function]
If X is a matrix, returns the diagonal of X. If X is a sequence, returns a diagonal matrix of rank (length X) with diagonal elements eq to the elements of X.

(identity-matrix n) [Function]
Returns the identity matrix of rank N.

(inner-product x y) [Function]
Returns inner product of sequences X and Y.

(inverse m) [Function]
Returns the inverse of the the square matrix M; signals an error if M is ill conditioned or singular

(lu-decomp a) [Function]
A is a square matrix of numbers (real or complex). Computes the LU decomposition of A and returns a list of the form (LU IV D FLAG), where LU is a matrix with the L part in the lower triangle, the U part in the upper triangle (the diagonal entries of L are taken to be 1), IV is a vector describing the row permutation used, D is 1 if the number of permutations is odd, -1 if even, and FLAG is T if A is numerically singular, NIL otherwise. Used bu LU-SOLVE.

(lu-solve lu b) [Function]
LU is the result of (LU-DECOMP A) for a square matrix A, B is a sequence. Returns the solution to the equation Ax = B. Signals an error if A is singular.

(make-list size &key (initial-element nil)) [Function]
Creates and returns a list containing SIZE elements, each of which is initialized to INITIAL-ELEMENT.

(make-sweep-matrix x y &optional weights) [Function]
X is a matrix, Y and WEIGHTS are sequences. Returns the sweep matrix for the (possibly weighted) regression of Y on X.

(map-elements function data &rest more-data) [Function]
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. The result is of the same type and shape as the first DATA argument, with elements the result of applying FUNCTION elementwise to the DATA arguments

(matmult a b) [Function]
Returns the matrix product of matrices a and b. If a is a vector it is treated as a row vector; if b is a vector it is treated as a column vector.

(matrix dim data) [Function]
returns a matrix of dimensions DIM initialized using sequence DATA in row major order.

(matrixp m) [Function]
Returns T if M is a matrix, NIL otherwise.

(outer-product x y &optional (fcn #'*)) [Function]
Returns the generalized outer product of x and y, using fcn. That is, the result is a matrix of dimension ((length x) (length y)) and the (i j) element of the result is computed as (apply fcn (aref x i) (aref y j)).

(permute-array a p) [Function]
Returns a copy of the array A permuted according to the permutation P.

(qr-decomp a) [Function]
A is a matrix of real numbers with at least as many rows as columns. Computes the QR factorization of A and returns the result in a list of the form (Q R).

(rcondest a) [Function]
Returns an estimate of the reciprocal of the L1 condition number of an upper triangular matrix a.

(row-list m) [Function]
Returns a list of the rows of M as vectors

(solve a b) [Function]
Solves A x = B using LU decomposition and backsolving. B can be a sequence or a matrix.

(split-list list cols) [Function]
Returns a list of COLS lists of equal length of the elements of LIST. Example: (split-list '(1 2 3 4 5 6) 2) returns ((1 2 3) (4 5 6))

(sum &rest number-data) [Function]
Returns the sum of all the elements of its arguments. Returns 0 if there are no arguments. Vector reducing.

(sv-decomp a) [Function]
A is a matrix of real numbers with at least as many rows as columns. Computes the singular value decomposition of A and returns a list of the form (U W V FLAG) where U and V are matrices whose columns are the left and right singular vectors of A and W is the sequence of singular values of A. FLAG is T if the algorithm converged, NIL otherwise.

(sweep-operator a indices &optional tolerances) [Function]
A is a matrix, INDICES a sequence of the column indices to be swept. Returns a list of the swept result and the list of the columns actually swept. (See MULTREG documentation.) If supplied, TOLERANCES should be a list of real numbers the same length as INDICES. An index will only be swept if its pivot element is larger than the corresponding element of TOLERANCES.

(transpose m) [Function]
Returns the transpose of the matrix M.

(vectorp m) [Function]
Returns T if M is a vector, NIL otherwise.

## C.7 System Functions

(alloc number) [Function]
Changes number of nodes to allocate in each segment to NUMBER. Returns old number of nodes to allocate.

(call-cfun cfun &rest args) [Function]
CFUN is a string naming a C function. The remaining arguments must be integers, sequences of integers, reals or sequences of reals. CFUN is called with the remaining arguments and a list of the lists of the values of the arguments after the call is returned. Arguments in the call will be pointers to ints or pointers to doubles. Not available on all implementations.

(call-fsub fsub &rest args) [Function]
FSUB is a string naming a FORTRAN subroutine. The remaining arguments must be integers, sequences of integers, reals or sequences of reals. FSUB is called with the remaining arguments and a list of the lists of the values of the arguments after the call is returned. Arguments in the call will be (arrays of) integers or double precision numbers. Not available on all implementations.

(call-lfun lfun &rest args) [Function]
LFUN is a C function written to conform to internal XLISP argument reading and value returning conventions. Applies LFUN to ARGS and returns the result.

(debug) [Function]
Enable breaking on error on.

(dyn-load file &key verbose libflags fortran) [Function]
Links the object file FILE with standard C libraries and loads into the running XLISP-STAT process. If FORTRAN is not NIL also searches standard FORTRAN libraries. LIBFLAGS can be a string used to specify additional libraries, for example

(exit) [Function]
Exits from XLISP.

(expand number) [Function]
Expand memory by adding NUMBER segments. Returns the number of segments.

(gc) [Function]
Forces garbage collection. Returns nil.

(help &optional symbol) [Function]
Prints the documentation associated with SYMBOL. With no argument, this function prints the greeting message to beginners.

(help* string) [Function]
Prints the documentation associated with those symbols whose print names contain STRING as substring. STRING may be a symbol, in which case the print-name of that symbol is used.

(load filename &key (verbose t) (print nil)) [Function]
Loads the file named by FILENAME into XLISP. Returns T if load succeeds, NIL if file does not exist.

(nodebug) [Function]
Disable breaking on error on.

(room) [Function]
Shows memory allocation statistics. Returns nil.

(save file) [Function]
Saves current memory image in FILE.wks. Does not work right with allocated objects.

(variables) [Function]
Prints the names of all def'ed variables

## C.8 Some Basic Lisp Functions, Macros and Special Forms

Except where noted these functions should have a significant subset of their Common Lisp functionality as defined in Steele [15].

(and {form}*) [Macro]
Evaluates FORMs in order from left to right. If any FORM evaluates to NIL, returns immediately with the value NIL. Else, returns the value of the last FORM.

(append &rest lists) [Function]
Constructs a new list by concatenating its arguments.

(apply function &rest args) [Function]
Conses all arguments but the last onto the last and applies FUNCTION to the resulting argument list. Last argument must be a list.

(apropos string) [Function]
Prints symbols whose print-names contain STRING as substring. If STRING is a symbol its print name is used.

(apropos-list string) [Function]
Returns, as a list, all symbols whose print-names contain STRING as substring. If STRING is a symbol its print name is used.

(assoc item alist &key (test (function eql)) test-not) [Function]
Returns the first pair in ALIST whose car is equal (in the sense of TEST) to ITEM.

(atom x) [Function]
Returns T if X is not a cons; NIL otherwise.

(boundp symbol) [Function]
Returns T if the global variable named by SYMBOL has a value; NIL otherwise.

(car list) [Function]
Returns the car of LIST. Returns NIL if LIST is NIL.

(case keyform (key | (key*) form*) *) [Function]
Evaluates KEYFORM and tries to find the KEY that is EQL to the value of KEYFORM. If one is found, then evaluates FORMs that follow the KEY and returns the value of the last FORM. If not, simply returns NIL.

(cdr list) [Function]
Returns the cdr of LIST. Returns NIL if LIST is NIL.

(close stream) [Function]
Close file stream STREAM.

(coerce x type) [Function]
Coerces X to an object of the type TYPE.

(cond (test form*) *) [Function]
Evaluates each TEST in order until one evaluates to a non-NIL value. Then evaluates the associated FORMs in order and returns the value of the last FORM. If no forms follow the TEST, then returns the value of the TEST. Returns NIL, if all TESTs evaluate to NIL.

(cons x y) [Function]
Returns a new cons (list node) whose car and cdr are X and Y, respectively.

(consp x) [Function]
Returns T if X is a cons; NIL otherwise.

(defmacro name defmacro-lambda-list [doc] {form}*) [Macro]
Defines a macro as the global definition of the symbol NAME. The complete syntax of a lambda-list is: (var* [&optional var*] [&rest var] [&aux var*]) The doc-string DOC, if supplied, is saved as a FUNCTION doc and can be retrieved by (documentation 'NAME 'function).

(defun name lambda-list [doc] {form}*) [Macro]
Defines a function as the global definition of the symbol NAME. The complete syntax of a lambda-list is: (var* [&optional var*] [&rest var] [&aux var*]) The doc-string DOC, if supplied, is saved as a FUNCTION doc and can be retrieved by (documentation 'NAME 'function).

(do ({(var [init [step]])}*) (endtest {result}*) {tag | statement}*) [Macro]
Creates a NIL block, binds each VAR to the value of the corresponding INIT, and then executes STATEMENTs repeatedly until ENDTEST is satisfied. After each iteration, assigns to each VAR the value of the corresponding STEP. When ENDTEST is satisfied, evaluates RESULTs as a PROGN and returns the value of the last RESULT (or NIL if no RESULTs are supplied). Performs variable bindings and assignments all at once, just like LET does.

(do* ({(var [init [step]])}*) (endtest {result}*) tag | statement*) [Macro]
Just like DO, but performs variable bindings and assignments in serial, just like LET* and SETQ do.

(dolist (var listform [result]) {tag | statement}*) [Macro]
Executes STATEMENTs, with VAR bound to each member of the list value of LISTFORM. Then returns the value of RESULT (which defaults to NIL).

(dotimes (var countform [result]) {tag | statement}*) [Macro]
Executes STATEMENTs, with VAR bound to each number between 0 (inclusive) and the value of COUNTFORM (exclusive). Then returns the value of RESULT (which defaults to NIL).

(elt a &rest indices) [Function]
A can be a list or an array. If A is a list and INDICES is a single number then the appropriate element of A is returned. If is a list and INDICES is a list of numbers then the sublist of the corresponding elements is returned. If A in an array then the number of INDICES must match the ARRAY-RANK of A. If each index is a number then the appropriate array element is returned. Otherwise the INDICES must all be lists of numbers and the corresponding submatrix of A is returned. ELT can be used in setf.

(eq x y) [Function]
Returns T if X and Y are the same identical object; NIL otherwise.

(eql x y) [Function]
Returns T if X and Y are EQ, or if they are numbers of the same type with the same value, or if they are identical strings. Returns NIL otherwise.

(equal x y) [Function]
Returns T if X and Y are EQL or if they are of the same type and corresponding components are EQUAL. Returns NIL otherwise. Arrays must be EQ to be EQUAL.

(equalp x y) [Function]
Returns T if (equal x y), or x, y are numbers and (= x y), or x and y are strings and (string-equal x y).

(first x) [Function]
Equivalent to (CAR X).

(format destination control &rest args) [Function]
Very basic implementation of Common Lisp format function. Only A, S, D, F, E, G, and G can take two.

(funcall function &rest arguments) [Function]
Applies FUNCTION to the ARGUMENTs

(function x) [Special Form]
or #'x If X is a lambda expression, creates and returns a lexical closure of X in the current lexical environment. If X is a symbol that names a function, returns that function.

(getf place indicator &optional default) [Function]
Returns property value of INDICATOR in PLACE, or DEFAULT if not found.

(identity x) [Function]
Simply returns X.

(if test then [else]) [Macro]
If TEST evaluates to non-NIL, then evaluates THEN and returns the result. If not, evaluates ELSE (which defaults to NIL) and returns the result.

(last list) [Function]
Returns the last cons in LIST

(length sequence) [Function]
Returns the length of SEQUENCE.

(let (var | (var [value]) *) form*) [Function]
Initializes VARs, binding them to the values of VALUEs (which defaults to NIL) all at once, then evaluates FORMs as a PROGN.

(let* (var | (var [value]) *) form*) [Function]
Initializes VARs, binding them to the values of VALUEs (which defaults to NIL) from left to right, then evaluates FORMs as a PROGN.

(listp x) [Function]
Returns T if X is either a cons or NIL; NIL otherwise.

(map result-type function sequence &rest more-sequences) [Function]
FUNCTION must take as many arguments as there are sequences provided. RESULT-TYPE must be the either the symbol VECTOR or the symbol LIST. The result is a sequence of the specified type such that the i-th element of the result is the result of applying FUNCTION to the i-th elements of the SEQUENCEs.

(mapc fun list &rest more-lists) [Function]
Applies FUN to successive cars of LISTs. Returns the first LIST.

(mapcar fun list &rest more-lists) [Function]
Applies FUN to successive cars of LISTs and returns the results as a list.

(mapl fun list &rest more-lists) [Function]
Applies FUN to successive cdrs of LISTs. Returns the first LIST.

(maplist fun list &rest more-lists) [Function]
Applies FUN to successive cdrs of LISTs and returns the results as a list.

(member item list &key (test (function eql)) test-not) [Function]
Returns the tail of LIST beginning with the first ITEM.

(not x) [Function]
Returns T if X is NIL; NIL otherwise.

(nth n list) [Function]
Returns the N-th element of LIST, where the car of LIST is the zero-th element.

(nthcdr n list) [Function]
Returns the result of performing the CDR operation N times on LIST.

(null x) [Function]
Returns T if X is NIL; NIL otherwise.

(numberp x) [Function]
Returns T if X is any kind of number; NIL otherwise.

(objectp x) [Function]
Returns T if X is an object, NIL otherwise.

(open fname &key (direction :input)) [Function]
Opens file named by string or symbol FNAME. DIRECTION is :INPUT or :OUTPUT.

(or {form}*) [Macro]
Evaluates FORMs in order from left to right. If any FORM evaluates to non-NIL, quits and returns that value. If the last FORM is reached, returns whatever value it returns.

(prin1 object &optional (stream *standard-output*)) [Function]
Prints OBJECT in the most readable representation. Returns OBJECT.

(princ object &optional (stream *standard-output*)) [Function]
Prints OBJECT without escape characters. Returns OBJECT.

(print object &optional (stream *standard-output*)) [Function]
Outputs a newline character, and then prints OBJECT in the most readable representation. Returns OBJECT.

(prog ({var | (var [init])}*) {tag | statement}*) [Macro]
Binds VARs in parallel, and then executes STATEMENTs.

(prog* ({var | (var [init])}*) {tag | statement}*) [Macro]
Binds VARs sequentially, and then executes STATEMENTs.

(prog1 first {form}*) [Macro]
Evaluates FIRST and FORMs in order, and returns the value of FIRST.

(prog2 first second {forms}*) [Macro]
Evaluates FIRST, SECOND, and FORMs in order, and returns the value of SECOND.

(progn {form}*) [Macro]
Evaluates FORMs in order, and returns whatever the last FORM returns.

(progv symbols values {form}*) [Macro]
Evaluates FORMs in order, with SYMBOLS dynamically bound to VALUES, and returns whatever the last FORM returns.

(provide name) [Function]
Adds NAME to the list of modules.

(quote x) [Special Form]
or 'x Returns X without evaluating it.

(read &optional (stream *standard-input*) (eof-error-p t) (eof-value nil) (recursivep nil)) [Function]
Reads and returns the next object from STREAM.

(reduce function sequence &key initial-value) [Function]
Combines all the elements of SEQUENCE using a binary operation FUNCTION. If INITIAL-VALUE is supplied it is logically placed before SEQUENCE.

(remove item list &key (test (function eql)) test-not) [Function]
Returns a copy of LIST with ITEM removed.

(remove-if test list) [Function]
Returns a copy of LIST with elements satisfying TEST removed.

(remove-if-not test list) [Function]
Returns a copy of LIST with elements not satisfying TEST removed.

(require name) [Function]
Loads module NAME, unless it has already been loaded. If PATH is supplied it is used as the file name; otherwise NAME is used. If file NAME is not in the current directory *default-path* is searched.

(rest x) [Function]
Equivalent to (CDR X).

(return [result]) [Macro]
Returns from the lexically surrounding PROG construct. The value of RESULT, which defaults to NIL, is returned as the value of the PROG construct.

(reverse list) [Function]
Returns a new list containing the same elements as LIST but in reverse order.

(second x) [Function]
Equivalent to (CAR (CDR X)).

(set symbol value) [Function]
Assigns the value of VALUE to the dynamic variable named by SYMBOL (i. e. it changes the global definition of SYMBOL), and returns the value assigned.

(setf {place newvalue}*) [Macro]
Replaces the value in PLACE with the value of NEWVALUE, from left to right. Returns the value of the last NEWVALUE. Each PLACE may be any one of the following: * A symbol that names a variable. * A function call form whose first element is the name of the following functions: nth aref subarray sublist select elt get symbol-value symbol-plist documentation slot-value c?r c??r c???r c????r where '?' stands for either 'a' or 'd'.

(setq {var form}*) [Macro]
VARs are not evaluated and must be symbols. Assigns the value of the first FORM to the first VAR, then assigns the value of the second FORM to the second VAR, and so on. Returns the last value assigned.

(string sym) [Function]
Returns print-name of SYM if SYM is a symbol, or SYM if SYM is a.

(stringp x) [Function]
Returns T if X is a string; NIL otherwise.

(sublis alist tree &key (test (function eql)) test-not) [Function]
Substitutes from ALIST for subtrees of TREE nondestructively.

(subst new old tree &key (test (function eql)) test-not) [Function]
Substitutes NEW for subtrees of TREE that match OLD.

(symbol-name symbol) [Function]
Returns the print name of the symbol SYMBOL.

(symbol-plist symbol) [Function]
Returns the property list of SYMBOL.

(symbol-value symbol) [Function]
Returns the current global value of the variable named by SYMBOL.

(symbolp x) [Function]
Returns T if X is a symbol; NIL otherwise.

(terpri &optional (stream *standard-output*)) [Function]
Outputs a newline character.

(time form) [Macro]
Form is evaluated and its result returned. In addition the time required for the evaluation is printed.

(type-of x) [Function]
Returns the type of X.

(unless test {form}*) [Macro]
If TEST evaluates to NIL evaluates FORMs as a PROGN. If not, returns NIL.

(unwind-protect protected-form {cleanup-form}*) [Macro]
Evaluates PROTECTED-FORM and returns whatever it returned. Guarantees that CLEANUP-FORMs be always evaluated before exiting from the UNWIND-PROTECT form.

(when test {form}*) [Macro]
If TEST evaluates to non-NIL evaluates FORMs as a PROGN. If not, returns NIL.

# Index