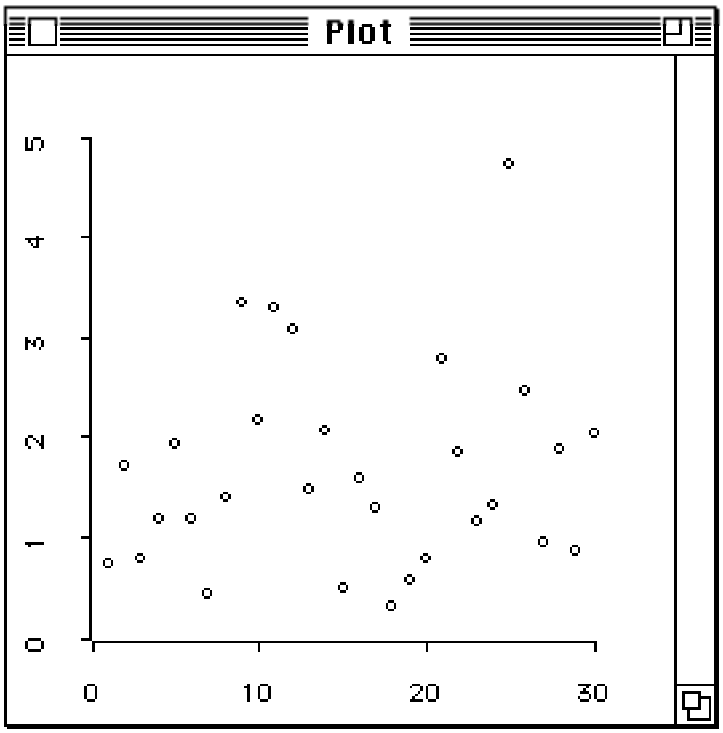

Figure 4: Scatterplot of precipitation levels against time.

```
> (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.

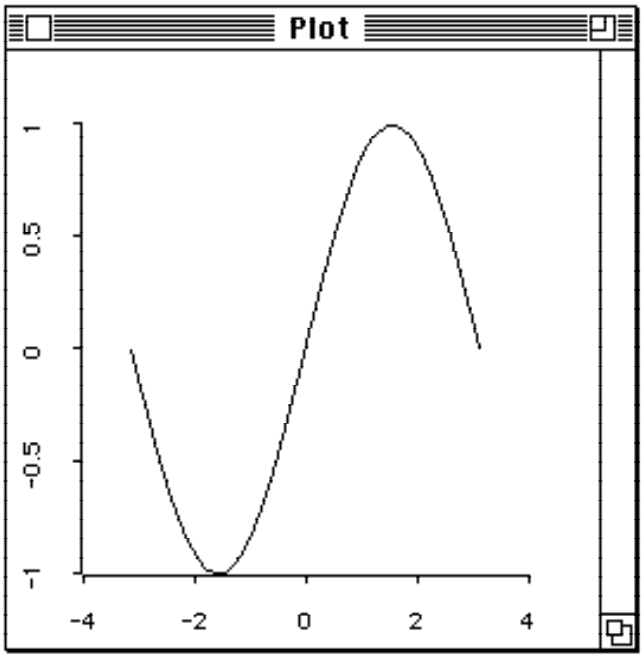

Figure 5: A plot of $\sin(x)$.

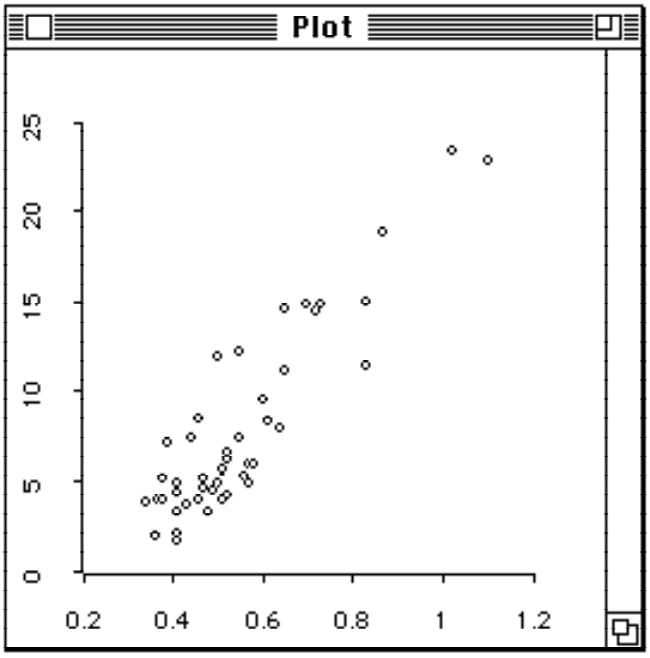

Figure 6: Plot of HC against CO.

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, χ^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.

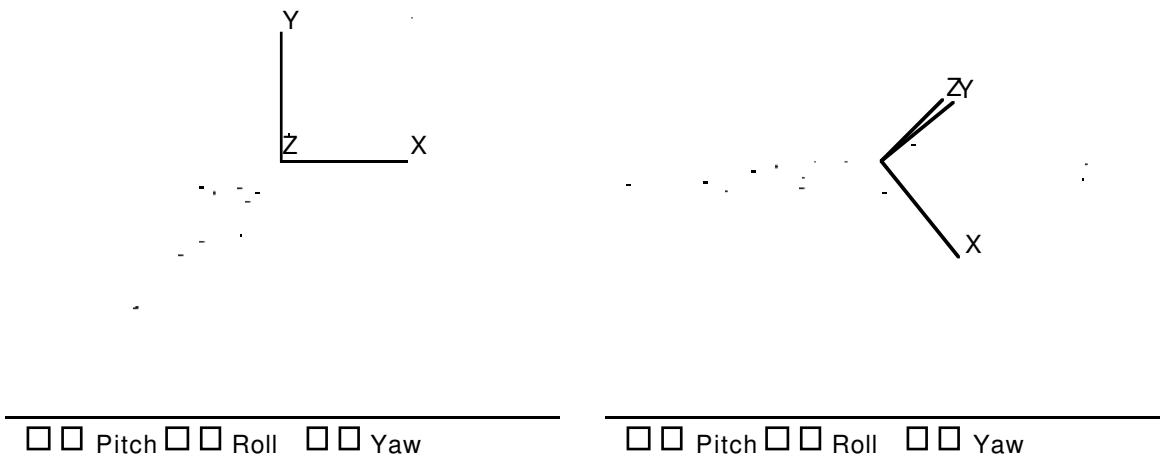

Figure 7: Two views of a rotatable plot of data on iron content, aluminum content and phosphate absorption in sediment samples.

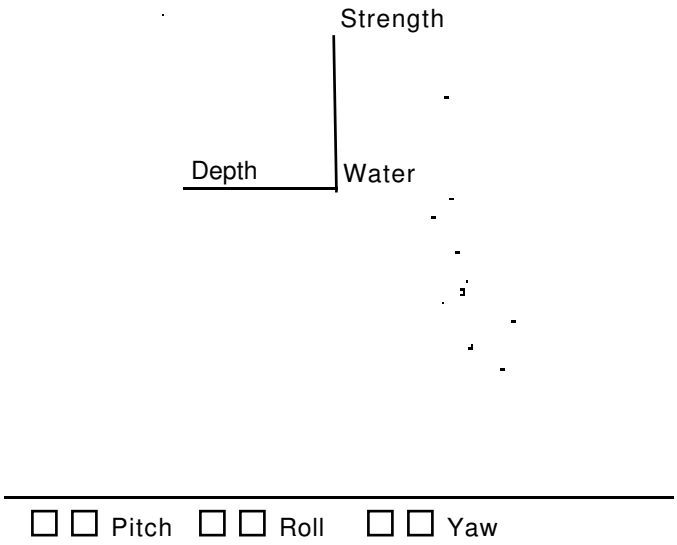

Figure 8: Rotatable plot of measurements on permafrost samples.

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*.

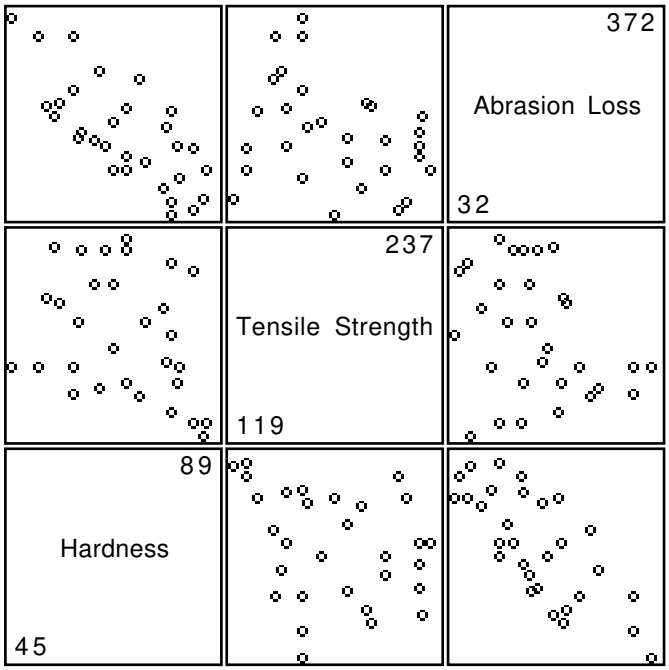

Figure 9: Scatterplot matrix of abrasion loss data.

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
```

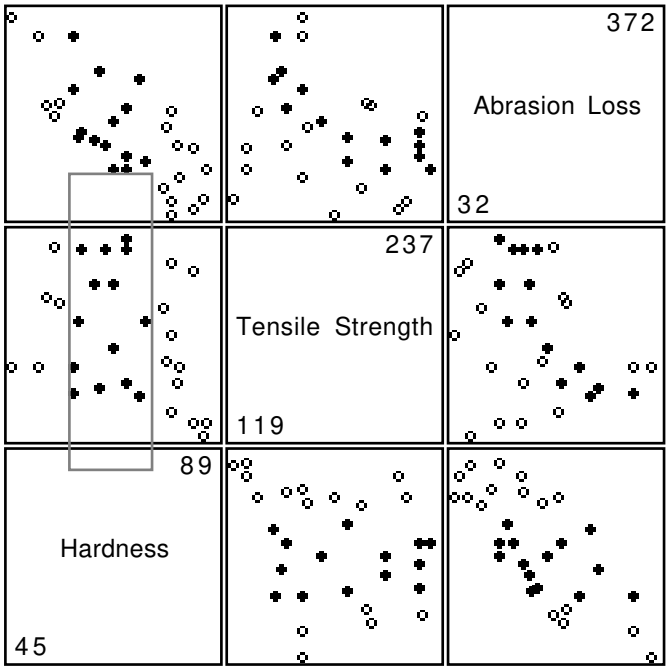

Figure 10: Scatterplot matrix with middle hardness values highlighted.

```
                  1 1 1 2 2 2 3 3 3 4 4 4))
(def variety (list 1 1 1  1 1 1  1 1 1  1 1 1  2 2 2  2 2 2  2 2 2
                   2 2 2  3 3 3  3 3 3  3 3 3  3 3 3))
```

(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.

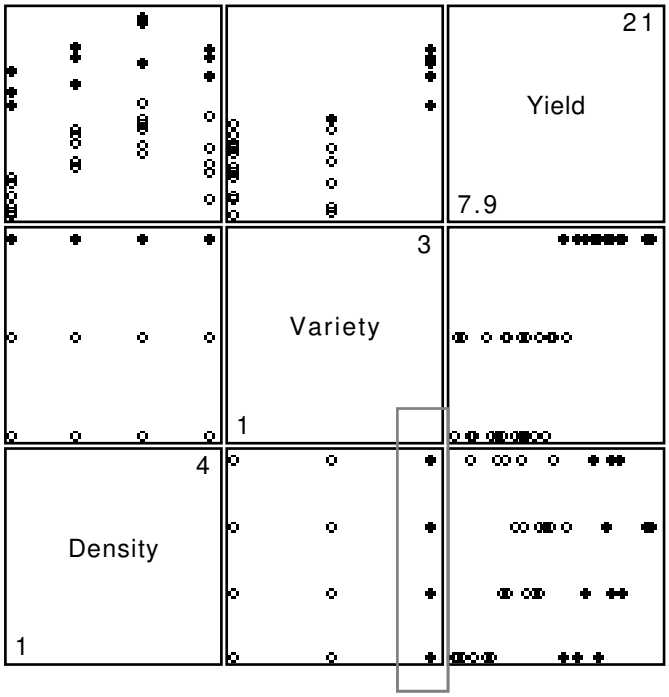

Figure 11: Scatterplot matrix for tomato yield data with points from the third variety highlighted.

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].

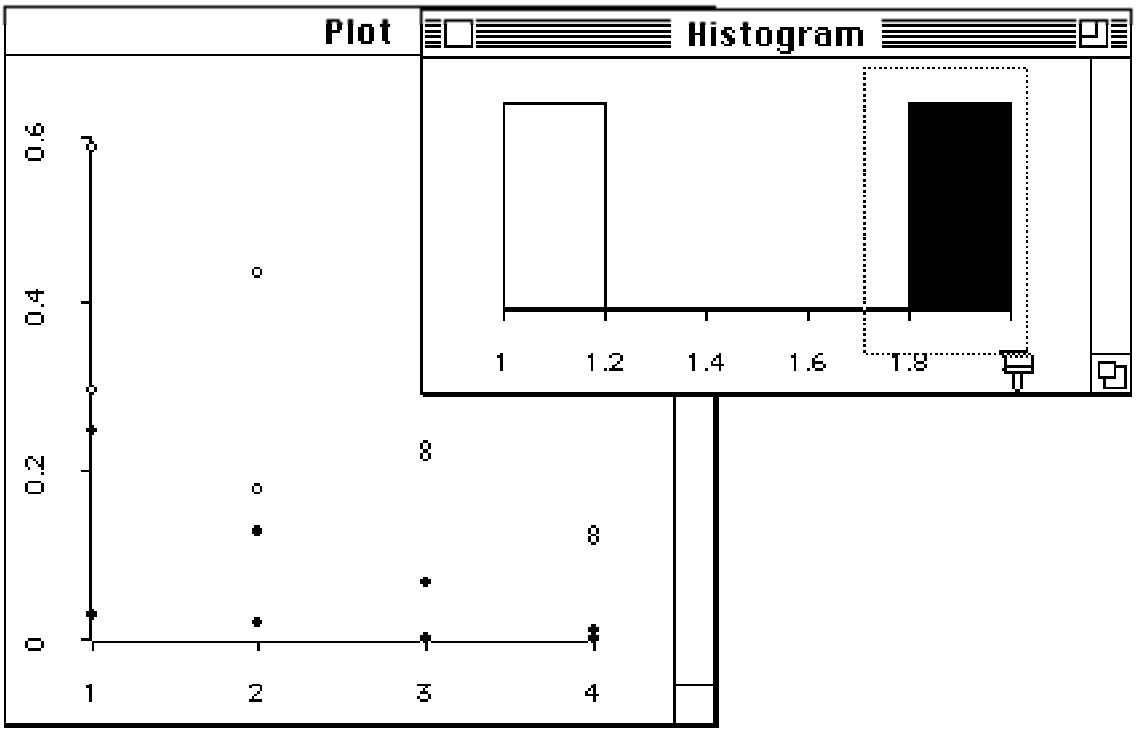

Figure 12: Scatterplot and histogram with points from one sugar group highlighted.

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.

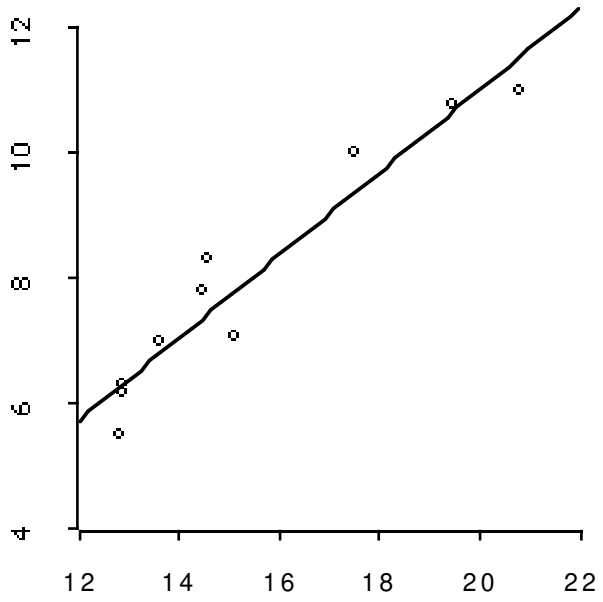

Figure 13: Scatterplot of bicycle data with fitted line.

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))
```

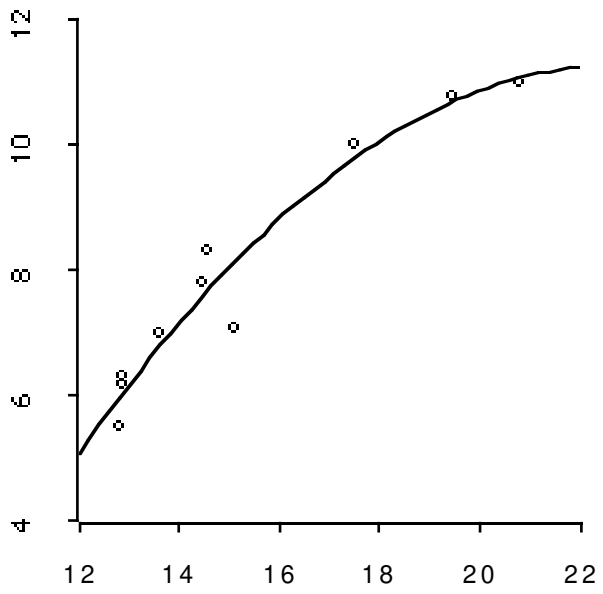

Figure 14: Scatterplot of bicycle data with fitted curve.

```
(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.

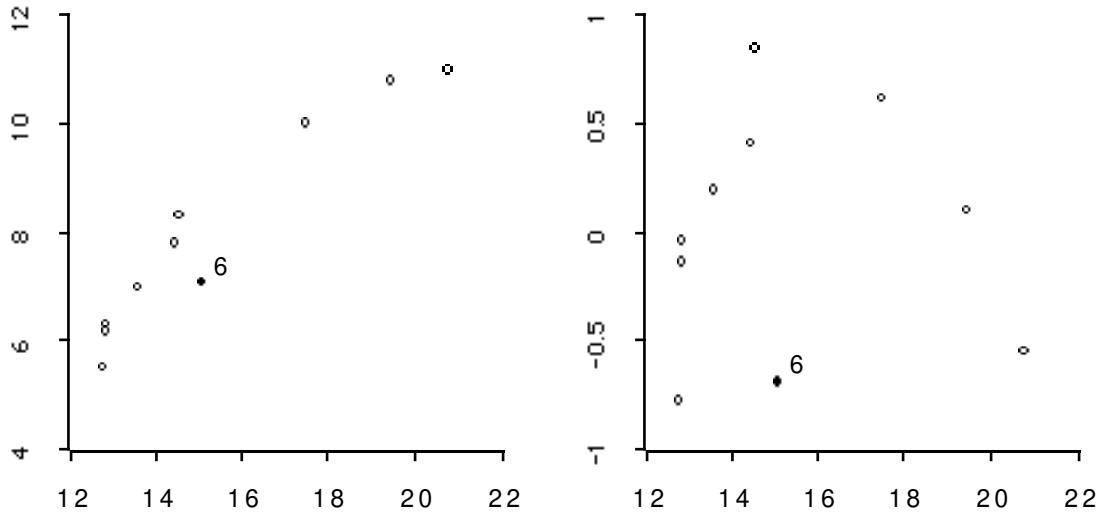

Figure 15: Linked raw data and residual plots for the bicycles example.

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))))))
```

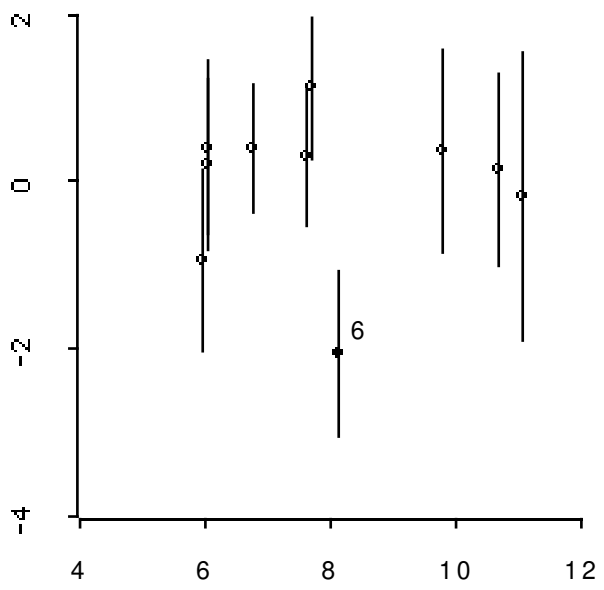

Figure 16: Bayes residual plot for bicycle data.

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 $-2 \leq x \leq 3$. 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 $-1 \leq x \leq 1, -1 \leq y \leq 1$ using a 6×6 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

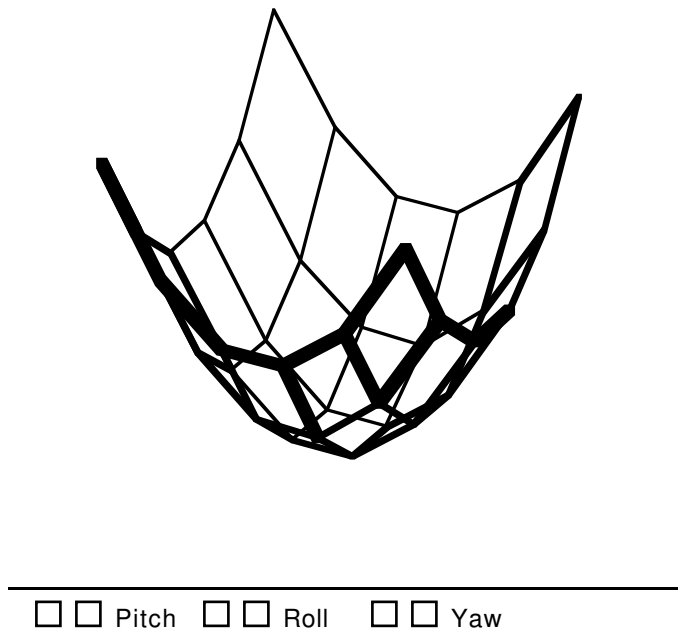

Figure 17: Rotatable plot of $f(x,y) = x^2 + y^2$.

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

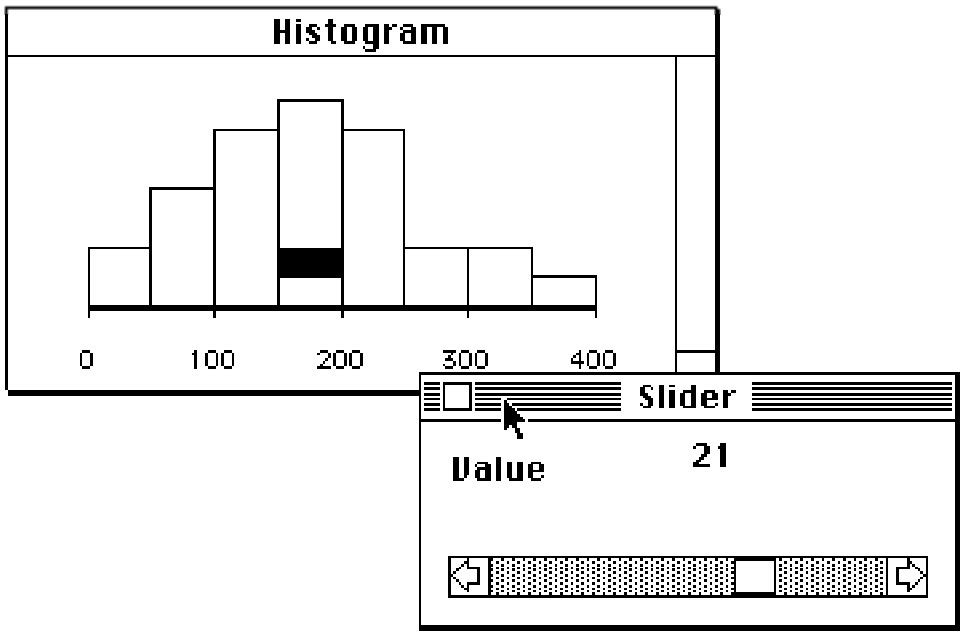

Figure 18: Slider-controlled animation of a histogram.

`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 χ^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, θ_1. The second parameter, θ_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 θ_1 and 0.1 for θ_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`.

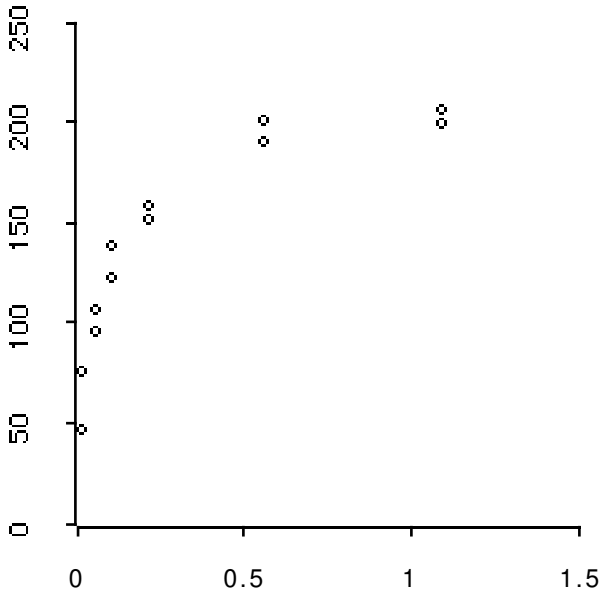

Figure 19: Plot of reaction velocity against substrate concentration for Puromycin experiment.

```
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 μ is the mean time between failures and β 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 μ 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 a4r(the)s22r(p331500000000100000100000100r(thmete0