A Tutorial Introduction

We'll start with an introduction to using Lisp-Stat as a statistical calculator and plotter.

We will see how to

- interact with the interpreter
- perform numerical operations
- modify data
- construct systematic and random data
- use built-in dynamic plots
- construct linear regression models
- define simple functions
- use these functions for
  - a simple animation
  - fitting nonlinear regression models
  - maximum likelihood estimation
  - approximate posterior computations

The Interpreter

Your interaction with Lisp-Stat is a conversation between you and the interpreter.

When the interpreter is ready to start the conversation, it gives you a prompt like

```
>
```

When you type in an expression and hit *return*, the interpreter evaluates the expression, prints the result, and gives a new prompt:

```
> 1
1
>
```

Operations on numbers are performed with *compound expressions*:

```
> (+ 1 2)
3
> (+ 1 2 3)
6
> (* (+ 2 3.7) 8.2)
46.74
```

The basic rule: everything is evaluated.

Numbers evaluate to themselves:

```
> 416
416
> 3.14
3.14
> -1
-1
```

Logical values and strings also evaluate to themselves:

```
> t                    ; true
T
> nil                  ; false
NIL
> "This is a string 1 2 3 4"
"This is a string 1 2 3 4"
```

The semicolon ";" is the Lisp comment character.

Symbols are evaluated by looking up their values, if they have one:

```
> pi
3.141593
> PI
3.141593
> x
error: unbound variable - X
```

Symbol names are *not* case-sensitive.

Compound expressions like `(+ 1 2)` are evaluated by

- looking up the function definition of the symbol `+`
- evaluating the argument expressions
- applying the function to the arguments

[Exception: If the function definition is a *special form*]

Compound expressions are evaluated recursively:

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

Operators like `+`, `*`, etc., are functions, like `exp` and `sqrt`

```
> (exp 1)
2.718282
> (sqrt 2)
1.414214
> (sqrt -1)
#C(0 1)
```

Numbers, strings, and symbols are *simple data*.

Several forms of *compound data* are available.

The most basic form of compound data is the list:

```
> (list 1 2 3)
(1 2 3)
> (list 1 "a string" (list 2 3))
(1 "a string" (2 3))
```

Other forms of compound data:

- vectors
- multi-dimensional arrays

Data sets can be named (symbols given values) using `def`:

```
> (def x (list 1 2 3))
X
> x
(1 2 3)
```

`def` is a *special form*.

Quoting an expression tells the interpreter *not* to evaluate it.

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

Lisp quotation is similar to English quotation:

Think about

> Say your name!

and

> Say "your name"!

`quote` is a special form.

Since `quote` is needed very frequently, there is a shorthand form:

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

Compound expressions are just lists.

Complicated expressions are easier to read when they are properly indented:

```
> (+ (- (* 3 5) (* (- 7 2) 6)) 3)
-12
> (+ (- (* 3 5)
        (* (- 7 2)
           6))
     3)
-12
```

The interpreter should help you with indentation, and with matching parentheses.

Exiting from Lisp-Stat:

- Choose **Quit** from the **File** menu
- Type `(exit)`

Elementary Computations and Graphs

One-Dimensional Summaries and Plots

A small data set:

Precipitation levels in inches recorded during the month of March in the Minneapolis-St. Paul area over a 30-year period:

0.77	1.74	0.81	1.20	1.95	1.20
0.47	1.43	3.37	2.20	3.00	3.09
1.51	2.10	0.52	1.62	1.31	0.32
0.59	0.81	2.81	1.87	1.18	1.35
4.75	2.48	0.96	1.89	0.90	2.05

Use `def` and `list` to enter the data:

```
(def precipitation
     (list .77 1.74 .81 ...))
```

or

```
(def precipitation '(.77 1.74 .81 ...))
```

We will see later how to read the data from a file.

Some summaries:

```
> (mean precipitation)
1.675

> (median precipitation)
1.47

> (standard-deviation precipitation)
1.0157

> (interquartile-range precipitation)
1.145
```

And some plots:

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

The results returned by these functions will be used later.

In Lisp-Stat, arithmetic operations are applied elementwise to compound data:

```
> (+ 1 precipitation)
(1.77 2.74 1.81 2.2 2.95 ...)
> (log precipitation)
(-0.2613648 0.5538851 -0.210721 ...)
> (sqrt precipitation)
(0.877496 1.31909 0.9 1.09545 ...)
```

The results can be passed to summary or plotting functions using compound expressions like

```
(mean (sqrt precipitation))
```

and

```
(histogram (sqrt precipitation))
```

The `boxplot` function produces a parallel boxplot when given a list of datasets:

```
(boxplot (list urban rural))
```

Two-Dimensional Plots

The `precipitation` data were collected over time.

It may be useful to look at a plot against time to see if there is any trend.

To construct a list of integers from 1 to 30 we use

```
> (iseq 1 30)
(1 2 ... 30)
```

A scatterplot of `precipitation` against time is produced by

```
(plot-points (iseq 1 30) precipitation)
```

Sometimes it is easier to see temporal patterns in a plot if the points are connected by lines:

```
(plot-lines (iseq 1 30) precipitation)
```

A connected lines plot is also useful for plotting functions.

As an example, let's plot the sine curve over the range $[-\pi, \pi]$.

A sequence of 50 equally spaced points between $-\pi$ and π is constructed by

```
(rseq (- pi) pi 50)
```

An expression for plotting $\sin(x)$ is

```
(plot-lines (rseq (- pi) pi 50)
            (sin (rseq (- pi) pi 50)))
```

You can simplify this expression by first defining a variable `x` as

```
(def x (rseq (- pi) pi 50))
```

and then constructing the plot with

```
(plot-lines x (sin x))
```

Scatterplots are of course particularly useful for bivariate data.

As an example, `abrasion-loss` contains the amount of rubber lost in an abrasion test for each of 30 specimens; `tensile-strength` contains the tensile strengths of the specimens.

A scatterplot of `abrasion-loss` against `tensile-strength` is produced by

```
(plot-points tensile-strength
             abrasion-loss)
```

Plotting Functions

A simpler way to plot a function like $\sin(x)$ is to use `plot-function`.

It expects a function, a lower limit, and an upper limit as arguments.

Unfortunately, just using

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

will not work:

```
> (plot-function sin (- pi) pi)
error: unbound variable - SIN
```

The reason is that symbols have separate values and function definitions.

This can be a bit of a nuisance.

But it means that you can't accidentally destroy the `list` function by defining a variable called `list`.

To get the *function definition* of `sin` we can use

```
(function sin)
```

or the shorthand form

```
#'sin
```

A plot of $\sin(x)$ between $-\pi$ and π is then produced by

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

or

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

More on the Interpreter

Saving Your Work

It is possible to

- save a transcript of your session with the `dribble` function
  (available in one of the menus on a Macintosh or the MS Windows version)
- save one or more defined variables to a file using the `savevar` function.
- save a copy of a plot
  - to the clipboard on a Macintosh or in Windows
  - to a postscript file in *X11*

A Command History Mechanism

There is a simple command history mechanism:

`-`	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 the most useful ones.

Getting Help

The `help` function provides a brief description of a function:

```
> (help 'median)
MEDIAN                                  [function-doc]
Args: (x)
Returns the median of the elements of X.
```

`help` is an ordinary function – the quote in front of `median` is essential.

`help*` gives help information about all functions with names containing its argument:

```
> (help* 'norm)
-------------------------------------------------
BIVNORM-CDF                          [function-doc]
Args: (x y r)
Returns the value of the standard bivariate
normal distribution function with correlation R
at (X, Y). Vectorized.
-------------------------------------------------
...
-------------------------------------------------
NORMAL-CDF                           [function-doc]
Args: (x)
Returns the value of the standard normal
distribution function at X. Vectorized.
-------------------------------------------------
...
```

The function **apropos** gives a listing of the symbols that contain its argument:

```
> (apropos 'norm)
NORMAL-QUANT
NORMAL-RAND
NORMAL-CDF
NORMAL-DENS
NORMAL
BIVNORM-CDF
NORM
```

Listing and Undefining Variables

`variables` gives a listing of variables created with `def`:

```
> (variables)
(PRECIPITATION RURAL URBAN ...)
```

To free up space, you may want to get rid of some variables:

```
> (undef 'rural)
RURAL
> (variables)
(PRECIPITATION URBAN ...)
```

Interrupting a Calculation

Occasionally you may need to interrupt a calculation.

Each system has its own method for doing this:

- On the Macintosh, *hold down* the **Command** key and the **Period** key.
- In MS Windows, hold down CONTROL-C
- On UNIX systems, use the standard interrupt, usually CONTROL-C.

Some Data-Handling Functions

Generating Systematic Data

We have already seen two functions,

- `iseq` for generating a sequence of consecutive integers
- `rseq` for generating equally spaced real values.

`iseq` can also be used with a single integer argument n; this produces a list of integers $0, 1, \ldots, n-1$:

```
> (iseq 10)
(0 1 2 3 4 5 6 7 8 9)
```

`repeat` is useful for generating sequences with a particular pattern:

```
> (repeat 2 3)
(2 2 2)

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

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

For example, if the data in the table

Density	Variety		
	1	2	3
1	9.2 12.4 5.0	8.9 9.2 6.0	16.3 15.2 9.4
2	12.4 14.5 8.6	12.7 14.0 12.3	18.2 18.0 16.9
3	12.9 16.4 12.1	14.6 16.0 14.7	20.8 20.6 18.7
4	10.9 14.3 9.2	12.6 13.0 13.0	18.3 16.0 13.0

are entered by rows

```
(def yield (list  9.2 12.4  5.0 ...))
```

then the density and variety levels are generated by

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

and

```
(def density (repeat (list 1 2 3 4)
                     (list 9 9 9 9)))
```

Generating Random Data

50 uniform variates are generated by

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

`normal-rand` and `cauchy-rand` are similar.

50 gamma variates with unit scale and exponent 4 are generated by

```
(gamma-rand 50 4)
```

`t-rand` and `chisq-rand` are similar.

50 beta variates with $\alpha = 3.5$ and $\beta = 7.2$ are generated by

```
(beta-rand 50 3.5 7.2)
```

`f-rand` is similar.

Binomial and Poisson variates are generated by

```
(binomial-rand 50 5 .3)
(poisson-rand 50 4.3)
```

The sample size arguments may also be lists of integers.

The result will then be a list of samples.

The function `sample` selects a random sample without replacement from a list:

```
> (sample (iseq 1 20) 5)
(20 11 3 14 10)
```

If `t` is given as an optional third argument, the sample is drawn with replacement:

```
> (sample (iseq 1 20) 5 t)
(2 14 14 11 18)
```

Any value other than `nil` can be given as the third argument.

[Logical operations usually interpret any value other than `nil` as true.]

Giving `nil` as the third argument is equivalent to omitting the argument.

Forming Subsets and Deleting Cases

The `select` function lets you to select one or more elements from a list:

```
(def x (list 3 7 5 9 12 3 14 2))
> (select x 0)
3
> (select x 2)
5
```

Lisp, like C but in contrast to FORTRAN, numbers elements of lists starting at zero.

To get a group of elements at once, use a list of indices:

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

To select all elements of `x` except the element with index 2, you can use

```
> (select x (remove 2 (iseq 8)))
(3 7 9 12 3 14 2)
```

or

```
> (select x (which (/= 2 (iseq 8))))
(3 7 9 12 3 14 2)
```

The `/=` function produces

```
> (/= 2 (iseq 8))
(T T NIL T T T T T)
```

and `which` turns this into indices of the `t` elements:

```
> (which (/= 2 (iseq 8)))
(0 1 3 4 5 6 7)
```

Combining Several Lists

`append` and `combine` allow you to merge several lists:

```
> (append (list 1 2 3) (list 4) (list 5 6 7 8))
(1 2 3 4 5 6 7 8)
> (combine (list 1 2 3)
           (list 4)
           (list 5 6 7 8))
(1 2 3 4 5 6 7 8)
```

`append` requires its arguments to be lists and only appends at one level.

`combine` allows simple data arguments, and works recursively through nested lists:

```
> (combine 1 2 (list 3 4))
(1 2 3 4)
> (append (list (list 1 2) 3) (list 4 5))
((1 2) 3 4 5)
> (combine (list (list 1 2) 3) (list 4 5))
(1 2 3 4 5)
```

Modifying Data

`setf` can be used to modify individual elements or sets of elements:

```
(def x (list 3 7 5 9 12 3 14 2))
X
> x
(3 7 5 9 12 3 14 2)
> (setf (select x 4) 11)
11
> x
(3 7 5 9 11 3 14 2)
> (setf (select x (list 0 2)) (list 15 16))
(15 16)
> x
(15 7 16 9 11 3 14 2)
```

`setf` *destructively modifies* a list, it does not copy it:

```
> (def x (list 1 2 3 4))
X
> (def y x)
Y
> (setf (select y 0) 'a)
A
> y
(A 2 3 4)
> x
(A 2 3 4)
```

The symbols `x` and `y` are two different names for the same list, and `setf` has changed that list.

To protect a list, you can copy it before making modifications:

```
> (def y (copy-list x))
Y
> (setf (select y 1) 'b)
B
> y
(A B 3 4)
> x
(A 2 3 4)
```

The general form of a `setf` call is

`(setf` <*form*> <*value*>`)`

`setf` can be used to modify other compound data, such as vectors and arrays.

Like `def`, `setf` can also be used to assign a value to a symbol:

```
> (setf z 3)
3
> z
3
```

There are three differences between `def` and `setf`:

- `def` returns the symbol, `setf` returns the value.
- `def` keeps track of the variables it creates; `variables` returns a list of them.
- `def` can only change global variable values, not local ones.

Reading Data Files

Two functions let you read data from a standard text file:

`(read-data-columns [`<*file*>` [`<*columns*>`]])`
`(read-data-file [`<*file*>`])`

The items in the file must be separated by white space (any number or combination of spaces, tabs or returns), not commas or other delimiters.

The arguments are optional:

- If <*file*> is omitted, a dialog is presented to let you select the file to read from.
- If `columns` is omitted, the number is determined from the first line of the file.

`read-data-columns` returns a list of lists representing the columns in the file.

`read-data-file` returns a list of the items in the file read in a row at a time.

Suppose you have a file `abrasion.dat` of the form

```
372     162     45
206     233     55
175     232     61
...   ...   ...
```

Then

```
> (read-data-columns "abrasion.dat" 3)
((372 206 175 ...)
 (162 233 232 ...)
 (45 55 61 ...))
> (read-data-file "abrasion.dat")
(372 162 45 206 233 55 ...)
```

The items in the file can be any items you would type into the interpreter (numbers, strings, symbols, etc.).

You can use these functions together with `select`:

```
> (def mydata
       (read-data-columns "abrasion.dat"))
> (def abr (select mydata 0))
ABR
> (def tens (select mydata 1))
TENS
> (def hard (select mydata 2))
HARD
> abr
(372 206 175 154 ...)
```

These functions should be adequate for most purposes.

If you need to read a file that is not of this form, you can use low-level file handling functions available in Lisp.

You can also use the `load` function or the **Load** menu command to load a file of Lisp expressions.

`load` requires the file name to have a `.lsp` extension.

Dynamic Graphs

The abrasion loss data set used earlier includes a second covariate, the hardness values of the rubber samples.

Two-dimensional static graphs alone are not adequate for examining the relationship among three variables.

Instead, we can use some dynamic graphs.

Dynamic graphs use motion and interaction to allow us to explore higher-dimensional aspects of a data set.

We will look at

- spinning plots
- scatterplot matrices
- brushing and selecting
- linking plots

Spinning Plots

Three variables can be displayed in a three-dimensional rotating scatterplot.

Rotation, together with depth cuing, allow your mind to form a three-dimensional image of the data.

A rotating plot of the abrasion loss data is constructed by

```
(spin-plot (list hardness
                 tensile-strength
                 abrasion-loss))
```

You can rotate the plot by placing the mouse cursor in one of the `Pitch`, `Roll`, or `Yaw` squares and pressing the mouse button.

If you press the mouse button using the *extend modifier*, then the plot continues to rotate after you release the mouse; it stops the next time you click in the button bar.

Every plot window provides a menu for communicating with the plot.

The menu on a rotating plot allows you to change the speed of rotation, or to choose whether to use depth cuing or whether to show the coordinate axes.

By default, the plot uses depth cuing: points closer to the viewer are drawn larger than points farther away.

The **Options** item in the menu lets you switch the background color from black to white; it also lets you change the type of scaling used.

`spin-plot` accepts several *keyword arguments*, including

`:title` – a title string for the plot

`:variable-labels` – a list of strings to use as axis labels

`:point-labels` – a list of strings to use as point labels

`:scale` – one of the symbols `variable`, `fixed`, or `nil`. (`nil` evaluates to itself, but the others need to be quoted).

A *keyword symbol* is a symbol starting with a colon.

Keyword symbols evaluate to themselves, so they do not need to be quoted.

Keyword arguments to a function must be given after all required (and optional) arguments.

Keyword arguments can be given in any order.

For example,

```
(spin-plot (list hardness
                 tensile-strength
                 abrasion-loss)
           :title "Abrasion Loss Data"
           :variable-labels
           (list "Hardness"
                 "Tensile Strength"
                 "Abrasion Loss"))
```

and

```
(spin-plot (list hardness
                 tensile-strength
                 abrasion-loss)
           :variable-labels
           (list "Hardness"
                 "Tensile Strength"
                 "Abrasion Loss")
           :title "Abrasion Loss Data")
```

are equivalent.

The center of rotation is the midrange of the data.

There are three scaling options:

`variable` (the default) – each variable is scaled by a different amount to make their ranges equal to $[-1, 1]$.

`fixed` – a common scale factor is applied to make the largest range equal to $[-1, 1]$

`nil` – no scaling is used; the variables are assumed to have been scaled to $[-1, 1]$.

You can center the variables at their means and scale them by a common factor with

```
(spin-plot
  (list (/ (- hardness (mean hardness))
           140)
        (/ (- tensile-strength
              (mean tensile-strength))
           140)
        (/ (- abrasion-loss
              (mean abrasion-loss))
           140))
  :scale nil)
```

Scatterplot Matrices

Another way to look at three (or more) variables is to use a scatterplot matrix of all pairwise scatterplots:

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

One way to use this plot is to approximately condition on one or more variables using *selecting* and *brushing*.

Two mouse modes are available:

Selecting. A plot is in selecting mode when the cursor is an arrow; this is the default.

In this mode you can

- select a point by clicking on it
- select a group of points by dragging a selection rectangle
- add to a selection by clicking or dragging with the extend modifier

Brushing. In this mode the cursor looks like a paint brush, and a dashed rectangle, the *brush*, is attached to it.

In this mode

- as you move the brush, points in it are highlighted
- this highlighting is transient; it is turned off when points move out of the brush
- you can select points by clicking and dragging

To change the mouse mode, choose **Mouse Mode** from the plot menu, and then select the mode you want from the dialog that is presented.

You can change the shape of the brush by selecting **Resize Brush** from the plot menu.

Brushing and selecting are available in all plots.

These operations can be useful in spinning plots to highlight features, for example by looking at the shape of a slice of the data.

It is possible to change the way these two standard modes behave and to define new mouse modes; we will see examples of this later on.

Interacting with Individual Plots

Plot menus contain several items that can be used together with selecting and brushing:

- If the **Labels** item is selected, point labels are shown next to selected or highlighted points.
- The selected points can be removed by choosing **Remove Selection**
- Unselected points can be removed by choosing **Focus on Selection**
- The plot can be rescaled after adding or removing points.
- You can change the color or the symbol used to draw a point.
- You can specify a set of indices to select or save the currently selected indices to a variable with the **Selection** item.

Depth cuing in spinning plots works by changing symbols, so it has to be turned off if you want to use different symbols.

Linked Plots

A scatterplot matrix links points in separate scatterplots.

You can link any plots by choosing **Link View** from the menus of the plots you want to link.

For example, you can link a histogram of `hardness` to a scatterplot of `abrasion-loss` against `tensile-strength`.

If you want to be able to select points with a particular label, you can use `name-list`.

```
(name-list 30)
```

You can also give `name-list` a list of strings.

Linking is based on the point index, so you have to use the same observation order in each plot.

Modifying a Scatterplot

After producing a plot, you can add more points or lines to it.

To do this, you need to save the *plot object* returned by plotting functions in a variable:

```
> (setf p (plot-points tensile-strength
                       abrasion-loss))
#<Object: 302592132, prototype = SCATTER...>
```

To use this object, you can send it *messages*.

This is done with expressions like

```
(send <object>
      <message selector>
      <argument 1> ...)
```

For example,

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

adds a regression line to the plot.

Plot objects understand a number of other messages.

The help message provides a (partial) listing:

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

:ABLINE :ACTIVATE :ADD-FUNCTION :ADD-LINES
...
```

The list of topics is the same for all scatterplots, but is somewhat different for rotating plots, scatterplot matrices, and histograms.

The `:clear` message clears the plot.

```
> (send p :help :clear)
:CLEAR
Message args: (&key (draw t))
Clears the plot data. If DRAW is true the plot
is redrawn; otherwise its current screen image
remains unchanged.
```

`:add-lines` and `:add-points` add new data:

```
> (send p :help :add-points)
:ADD-POINTS
Method args:
        (points &key point-labels (draw t))
or:     (x y  &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. For a 2D plot POINTS can be replaced
by two sequences X and Y.
```

The term *sequence* means a list or a vector.

A quadratic regression of **abrasion-loss** on **tensile-strength** produces the fit equation

$$Y = -280.1 + 5.986X + -0.01845X^2 + \epsilon$$

To put this curve on our plot we can use

```
> (send p :clear)
NIL
> (def x (rseq 100 250 50))
X
> (send p :add-lines x
                    (+ -280.1
                       (* x 5.986)
                       (* (^ x 2) -0.01845)))
NIL
> (send p :add-points tensile-strength
                      abrasion-loss)
```

Dynamic Simulations

We can also use plot modification to make a simple animation.

As an example, let's look at the variability in a histogram of 20 normal random variables.

Start by setting up a histogram:

```
> (setf h (histogram (normal-rand 20)))
#<Object: 303107988, prototype = HISTOGRAM...>
```

Then use a simple loop to replace the data:

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

The `:draw nil` insures that each histogram remains on the screen until it is replaced by the next one.

`dotimes` is one of several simple looping constructs available in Lisp; another is `dolist`.

If you have a very fast workstation or micro, this animation may move by too quickly.

One solution is to insert a pause of 10/60 of a second using

```
(pause 10)
```

So an alternate version of the loop is

```
(dolist (i (iseq 50))
  (send h :clear :draw nil)
  (pause 10)
  (send h :add-points (normal-rand 20)))
```

Regression

Regression models are implemented using objects and message-sending.

We can fit `abrasion-loss` to `tensile-strength` and `hardness`:

```
> (setf m (regression-model
           (list tensile-strength hardness)
           abrasion-loss))

Least Squares Estimates:

Constant                   885.537   (61.801)
Variable 0                -1.37537   (0.194465)
Variable 1                  -6.573   (0.583655)

R Squared:                0.840129
Sigma hat:                36.5186
Number of cases:               30
Degrees of freedom:            27

#<Object: 303170820, prototype = REGRESS...>
```

The `regression-model` function accepts a number of keyword arguments, including

`:print` – print summary or not

`:intercept` – include intercept term or not

`:weights` – optional weight vector

A range of messages are available for examining and modifying the model:

```
> (send m :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 ...
```

Some examples:

```
> (send m :coef-estimates)
(885.537 -1.37537 -6.573)
> (send m :coef-standard-errors)
(61.801 0.194465 0.583655)
> (send m :residuals)
(5.05705 2.43809 9.50074 19.9904 ...)
> (send m :fit-values)
(366.943 203.562 165.499 ...)
> (send m :plot-residuals)
#<Object: 1568394, prototype = SCATTER...>
> (send m :cooks-distances)
(0.00247693 0.000255889 0.00300101 ...)
> (plot-lines (iseq 1 30) *)
```

Defining Functions and Methods

No system can provide all tools any user might want.

Being able to define your own functions lets you

- provide short names for functions you use often
- provide simple commands to execute a long series of operations on different data sets
- develop new methods not provided in the system

Defining Functions

Functions are defined with the special form **defun**.

The simplest form of the **defun** syntax is

(defun <*name*> <*parameters*> <*body*>**)**

For example, a function to delete an observation can be defined as

```
> (defun delete-case (i x)
    (select x (remove i (iseq (length x)))))
DELETE-CASE
> (delete-case 2 (list 3 7 5 9 12 3 14 2))
(3 7 9 12 3 14 2)
```

None of the arguments to **defun** are quoted: **defun** is a special form that does not evaluate its arguments.

Functions can send messages to objects:

Suppose `m1` is a submodel of `m2`.

A function to compute the F statistic is

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

The `let` construct is used to create local variables that simplify the expression.

The documentation string is used by `help`:

```
> (help 'f-statistic)
F-STATISTIC                          [function-doc]
Args: (m1 m2)
Computes the F statistic for testing model m1
within model m2.
```

Functions as Arguments

Earlier we used the function definition of `sin` as an argument to `plot-function`.

We can use functions defined with `defun` as well:

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

Recall that `#'f` is short for `(function f)`, and is used for obtaining the function associated with the symbol `f`.

`spin-function` lets you construct a rotating plot of a function of two variables:

```
> (defun f (x y) (+ (^ x 2) (^ y 2)))
F
> (spin-function #'f -1 1 -1 1)
```

`contour-function` takes the same required arguments and produces a contour plot.

Graphical Animation Control

We have already seen how to construct some simple animations.

Using functions, we can provide graphical controls for animations.

As an example, let's look at the effect of the Box-Cox power transformation

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

on a normal probability plot of our `precipitation` variable.

A function to compute the transformation and normalize the result:

```
(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))))
```

`let*` is like `let`, but it establishes its bindings sequentially.

Next, sort the observations and compute the approximate expected normal order statistics:

```
(def x (sort-data precipitation))
(def nq (normal-quant (/ (iseq 1 30) 31)))
```

A probability plot of the untransformed data is constructed by

```
(setf p (plot-points nq (bc x 1)))
```

Since the power is 1, `bc` just rescales the data.

To change the power used in the graph, define

```
(defun change-power (r)
  (send p :clear :draw nil)
  (send p :add-points nq (bc x r)))
```

Evaluating

```
(change-power .5)
```

redraws the plot for a square root transformation.

We could use a loop to run through some powers, but it is nicer to use a slider dialog:

```
(sequence-slider-dialog (rseq -1 2 31)
                        :action #'change-power)
```

The action function `change-power` is called every time the slider is adjusted.

Defining Methods

Objects are arranged in a hierarchy.

When an object receives a message, it will use its own method to respond, if it has one.

If it does not have its own method, it asks its parents, and so on.

New methods are defined using `defmeth`.

For example, suppose `h` is a histogram created by

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

We can define a method for changing the sample by

```
(defmeth h :new-sample ()
  (send self :clear :draw nil)
  (pause 10)
  (send self :add-points (normal-rand 20)))
```

The variable `self` refers to the object receiving the message.

This special variable is needed because methods can be inherited.

The loop we used earlier can now be written as

```
(dotimes (i 50) (send h :new-sample))
```

Later on we will see how to override and augment standard methods.

More Models and Techniques

Nonlinear Regression

Suppose we record reaction rates `y` of a chemical reaction at various concentrations `x`:

```
(def x (list 0.02 0.02 0.06 0.06 0.11 0.11
             0.22 0.22 0.56 0.56 1.10 1.10))
(def y (list 76 47 97 107 123 139 159
               152 191 201 207 200))
```

A model often used for the mean reaction rate is the Michaelis-Menten model:

$$\eta(\theta) = \frac{\theta_0 x}{\theta_1 + x}$$

A lisp function to compute the mean response:

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

Initial estimates can be obtained from a plot

```
> (plot-points x y)
#<Object: ...>
```

The function **nreg-model** fits a nonlinear regression model:

```
> (setf m (nreg-model #'f y (list 200 .1)))
Residual sum of squares:   7964.185
Residual sum of squares:   1593.158
...
Residual sum of squares:   1195.449

Least Squares Estimates:

Parameter 0           212.6837 (6.947153)
Parameter 1         0.06412127 (0.0082809)

R Squared:           0.9612608
Sigma hat:            10.93366
Number of cases:            12
Degrees of freedom:         10
```

Several methods are available for examining the model:

```
> (send m :residuals)
(25.434 -3.56598 -5.81094 4.1891 ...)
> (send m :leverages)
(0.124852 0.124852 0.193059 0.193059 ...)
> (send m :cooks-distances)
(0.44106 0.0086702 0.041873 0.021761 ...)
```

Maximization and ML Estimation

The function **newtonmax** maximizes a function using Newton's method with backtracking.

As an example, times between failures on aircraft air-conditioning units are

```
(def x (list 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 assumes that the times are independent gamma variables with density

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

where μ is the mean time between failures and β is the gamma exponent.

A function to evaluate the log likelihood is

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

This definition uses the function `log-gamma` to evaluate $\log(\Gamma(\beta))$.

Initial estimates for μ and β are

```
> (mean x)
83.5172
> (^ (/ (mean x) (standard-deviation x)) 2)
1.39128
```

Using these starting values, we can 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)
```

You can use **numgrad** to check that the gradient is close to zero, and **numhess** to compute an approximate covariance matrix.

newtonmax will return the derivative information if it is given the keyword argument **:return-derivs** with value **t**.

If **newtonmax** does not converge, **nelmeadmax** may be able to locate the maximum.

Approximate Bayesian Computations

Suppose `times-pos` are survival times and `wbc-pos` white blood cell counts for AG-positive leukemia patients (Feigl & Zelen, 1965).

The log posterior density for an exponential regression model with a flat prior density is

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

It is computed by

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

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

`bayes-model` finds the posterior mode and prints a summary of first approximations to posterior means and standard deviations:

```
> (setf lk (bayes-model #'llik-pos '( 33 .8)))
maximizing...
......
Reason for termination: gradient size ...

First Order Approx. to Posterior Moments:

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

More accurate approximations are available:

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

Moments of functions of the parameters can also be approximated with `:moments`.

Marginal densities can be approximated using the `:margin1` message.

Generalized Linear Models

The generalized linear model system includes functions for fitting models with gamma, binomial, and poisson errors and various links.

New error structures and link functions can be added.

An Example:

```
> (def months-before (iseq 1 18))
MONTHS-BEFORE
> (def event-counts '(15 11 14 17 5 11 10 4
                      8 10 7 9 11 3 6 1 1 4))
EVENTS-RECALLED
> (def m (poissonreg-model months-before
                           event-counts))
Iteration 1: deviance = 26.3164
Iteration 2: deviance = 24.5804
Iteration 3: deviance = 24.5704
Iteration 4: deviance = 24.5704
```

```
Weighted Least Squares Estimates:

Constant               2.80316  (0.148162)
Variable 0          -0.0837691  (0.0167996)

Scale taken as:              1
Deviance:              24.5704
Number of cases:            18
Degrees of freedom:         16
```

`:residuals` and other methods are inherited:

```
> (send m :residuals)
(-0.0439191 -0.790305 ...)
> (send m :plot-residuals)
```