Statistical Computing and Dynamic Graphics Using Lisp-Stat

Luke Tierney
School of Statistics
University of Minnesota

February 1, 2021

Abstract

Lisp-Stat is an environment for general statistical computing and for using and developing dynamic graphical methods. The Lisp-Stat system contains a set of functions for performing basic statistical calculations and constructing dynamic statistical graphs such as linked scatterplots and rotatable three-dimensional plots. In addition, new numerical and graphical tools can be added to the system using the Lisp programming language and a simple object-oriented programming system.

The course will begin with an overview of the capabilities of the system and then introduce basic elements of Lisp programming and concepts of object-oriented programming. These ideas will then be used on several examples to illustrate how to add new graphical and numerical methods to the system. The examples may include a simple set of tools for fitting generalized linear models and an implementation of the Grand Tour for viewing data sets in four or more dimensions.

Several extended breaks will be scheduled and a small number of microcomputers or workstations will be made available during these breaks to allow participants to experiment with the ideas presented.

What is Lisp-Stat

Lisp-Stat is an extensible environment for statistical computing and dynamic graphics.

Some of its features are

- a variety of numerical statistical operations
- a variety of interactive and dynamic graphical methods
- a very high level programming language (Lisp) that can be used to
  - simplify combinations of calculations
  - adapt methods to specific problems
  - add new capabilities to the system
- an extensible graphics system through
  - access to graphical interface tools (Menus and Dialogs)
  - standard graphs as building blocks
  - use of object-oriented programming

Some things Lisp-Stat does not do:

- incorporate a large collection of tools for specialized analyses

  Such tools can be implemented within the system or, if C or Fortran implementations are already available, they can be accessed through dynamic or static loading.

- provide extensive support for presentation graphics

  Other systems on workstations and PC's already provide a wide range of tools for preparing presentation graphics.

Implementation and Portability:

- Lisp-Stat is a general specification
- XLISP-STAT is a first implementation
- XLISP-STAT runs on
  - Apple Macintosh
  - MS Windows
  - Commodore Amiga (J. Lindsey)
  - *Sun* workstations using *SunView*
  - BSD UNIX workstations running *X11*
- A Kyoto Common Lisp-based implementation may be available soon

Obtaining XLISP-STAT

XLISP-STAT source code and executables are available free of charge.

Source code for the UNIX, Macintosh, and MS Windows versions is available by anonymous *ftp* from `umnstat.stat.umn.edu` and by email from `statlib`.

> If you have access to the *internet* but have never used *ftp*, there is probably someone at your site who can help you.

Executables for Macintosh and MS Windows are also available for anonymous *ftp* from `umnstat.stat.umn.edu` and some other sites.

If you do not have access to *ftp*, you can get a copy of the Macintosh or MS Windows executables by sending disks and a self-addressed, stamped mailer to

Luke Tierney
School of Statistics
University of Minnesota
Minneapolis, MN 55455

The number of disks required is

Two 800K disks for the Macintosh version

One $3\frac{1}{4}$" high-density disk for the MS Windows version

Documentation

A tutorial introduction is available as a technical report:

Tierney, L., (1989), "XLISP-STAT: A statistical environment based on the XLISP language," U of M Tech. Rep. 528.

The LaTeX source for this report is available for *ftp* from `umnstat.stat.umn.edu` or by email from `statlib`.

More complete documentation is available as a book:

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

The technical report corresponds to Chapter 2 of the book.

Some Historical Notes

- Several extensible statistical environments have been based on high level languages:
  - The New *S* language
  - APL-based systems (Anscombe)
  - Tools for the *Gauss* system
- Several researchers have based systems on the Lisp language:
  - McDonald and Pedersen
  - Stuetzle
  - Oldford and Peters
  - Buja and Hurley
- A strong argument in favor of Lisp is that Lisp provides excellent support for *experimental programming*.

Course outline

1. Overview of Lisp-Stat
2. A Tutorial Introduction
3. Some Lisp Programming
4. Objects
5. Outline of the Graphics System
6. Some Dynamic Graphics Examples

Course Objectives

After completing the course, you should be able to

- use Lisp-Stat with your own data for basic statistical calculations and graphics.
- develop some simple numerical and graphical tools for your own applications
- understand the basics of Lisp programming and object-oriented programming

You should also be well prepared to learn more about Lisp-Stat by working through the book.

Overview of Lisp-Stat

Lisp-Stat allows you to enter data, transform data, and compute summary statistics:

```
> (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))
ABRASION-LOSS
> abrasion-loss
(372 206 175 154 136 112 55 45 ...)

> (log abrasion-loss)
(5.918894 5.327876 5.164786 ...)

> (mean abrasion-loss)
175.4667
> (standard-deviation abrasion-loss)
88.12755
```

You can construct a variety of interactive and dynamic graphs:

```
> (histogram hardness :title "Hardness")
#<OBJECT: ...>

> (plot-points
   tensile-strength abrasion-loss
   :Variable-labels (list "T" "A"))
#<OBJECT...>

> (spin-plot
   (list tensile-strength
         hardness
         abrasion-loss)
   :variable-labels (list "T" "H" "A"))
#<OBJECT: ...>
```

You can fit linear regression models:

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

Least Squares Estimates:

Constant              885.5374  (61.80104)
Variable  0:         -6.573002  (0.5836548)
Variable  1:         -1.375367  (0.1944645)

R Squared:             0.840129
Sigma hat:             36.51856
Number of cases:             30
Degrees of freedom:          27

#<Object: ...>
```

and nonlinear regression models:

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

> (def 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

M
```

You can also

- fit generalized linear models
- numerically maximize likelihood functions
- compute approximate posterior moments and marginal densities.

You can define functions of your own to implement new numerical methods or dynamic graphical ideas of your own.

We will look at several examples:

- estimating a survival curve
- Weibull regression
- dynamic power transformation
- variability in density estimation
- a regression sensitivity demonstration
- the Grand Tour

In most versions of XLISP-STAT it is also possible to incorporate and access routines written in C or FORTRAN.

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)
```

Some Lisp Programming

Conditional Evaluation and Predicates

The basic Lisp conditional evaluation construct is `cond`.

```
> (defun my-abs (x)
    (cond ((> x 0) x)
          ((= x 0) 0)
          ((< x 0) (- x))))
MY-ABS
> (my-abs -2)
2
```

Several simplified versions exist, including `if`, `unless` and `when`

```
(defun my-abs (x) (if (>= x 0) x (- x)))
```

Another simplified version of `cond` is `case`:

```
> (defun f (i)
    (case i
      (1 'one)
      (2 'two)
      (t (error "out of range"))))
> (f 1)
ONE
```

Logical expressions can be combined using **and**, **or**, and **not**.

```
> (defun in-range (x) (and (< 3 x) (< x 5)))
IN-RANGE
> (in-range 2)
NIL
> (in-range 4)
T

> (defun not-in-range (x)
    (or (>= 3 x) (>= x 5)))
NOT-IN-RANGE
> (not-in-range 2)
T

> (defun in-range (x) (< 3 x 5))
IN-RANGE
> (in-range 2)
NIL
> (in-range 4)
T
> (defun not-in-range (x) (not (in-range x)))
NOT-IN-RANGE
```

More on Functions

Function as Data

Suppose we want a function `num-deriv` to compute a numerical derivative.

If we define

```
(defun f (x) (+ x (^ x 2)))
```

then we want to get

```
> (num-deriv #'f 1)
3
```

Defining `num-deriv` as

```
(defun num-deriv (fun x)
  (let ((h 0.00001))
    (/ (- (fun (+ x h))
          (fun (- x h)))
       (* 2 h))))
```

will not work – our function is the *value* of `fun`, not its *function definition*:

```
> (num-deriv #'f 1)
error: unbound function - FUN
```

We need a function that calls the value of `fun` with an argument:

```
> (funcall #'+ 1 2)
3
```

A correct definition of `num-deriv` is

```
(defun num-deriv (fun x)
  (let ((h 0.00001))
    (/ (- (funcall fun (+ x h))
          (funcall fun (- x h)))
       (* 2 h))))
```

Another useful function is `apply`:

```
> (apply #'+ '(1 2 3))
6
> (apply #'+ 1 2 '(3 4))
10
> (apply #'+ 1 '(2 3))
6
```

Anonymous Functions

Defining and naming throw-away functions like `f` is awkward.

The same problem exists in mathematics.

Logicians developed the *lambda calculus*:

$$\lambda(x)(x + x^2)$$

is "the function that returns $x + x^2$ for the argument x."

Lisp uses this idea:

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

is a *lambda expression* for our function.

Lambda expressions are not yet Lisp functions.

To make them into functions, you need to use `function` or `#'`:

```
#'(lambda (x) (+ x (^ x 2)))
```

To take our derivative:

```
> (num-deriv #'(lambda (x) (+ x (^ x 2)))
             1)
3
```

To plot $2x + x^2$ over the range $[-2, 4]$,

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

Functions can also use lambda expressions to make new functions and return them as the value of the function.

We will see a few examples of this a bit later.

Local Variables and Environments

Variables and Scoping

A pairing of a variable symbol with a value is called a *binding*

Collections of bindings are called an *environment*.

Bindings can be global or they can be local to a group of expressions.

`let` and `let*` expressions and function definitions set up local bindings.

Consider the lambda expression

```
(lambda (x) (+ x a))
```

The meaning of `x` in the body is clear – it is bound to the calling argument.

The meaning of `a` is not so clear – it is a *free variable*.

We need a convention for determining the bindings of free variables.

This is the reason we need to use `function` on lambda expressions:

> Free variables in a function are bound to their values in the environment where the function is created

This is called the *lexical* or *static* scoping rule.

Other scoping rules are possible.

An example: making a derivative function:

```
> (defun make-num-deriv (fun)
    (let ((h 0.00001))
      #'(lambda (x)
        (/ (- (funcall fun (+ x h))
              (funcall fun (- x h)))
           (* 2 h)))))
MAKE-NUM-DERIV
> (setf f (make-num-deriv
           #'(lambda (x) (+ x (^ x 2)))))
#<Closure: #120cbe80>
> (funcall f 1)
3
> (funcall f 3)
7
```

Another example: making a normal log likelihood:

The log likelihood of a sample from a normal distribution is

$$-\frac{n}{2}\left[\log\sigma^2 + \frac{(\overline{x}-\mu)^2}{\sigma^2} + \frac{s^2}{\sigma^2}\right]$$

A function for evaluating this expression as a function of μ and σ^2 is returned by

```
(defun make-norm-log-lik (x)
  (let ((n (length x))
        (x-bar (mean x))
        (s-2 (^ (standard-deviation x) 2)))
    #'(lambda (mu sigma-2)
      (* -0.5
         n
         (+ (log sigma-2)
            (/ (^ (- x-bar mu) 2) sigma-2)
            (/ s-2 sigma-2))))))
```

The result returned by this function can be maximized, or it can be plotted with `spin-function` or `contour-function`.

Local Functions

It is also possible to set up local functions using `flet`:

```
> (defun f (x)
    (flet ((add-1 (x) (+ x 1)))
      (add-1 x)))
F
> (f 2)
3
> (add-1 2)
error: unbound function - ADD-1
```

`flet` sets up bindings in parallel, like `let`

`flet` cannot be used to define local recursive functions.

`labels` is like `flet` but allows mutually recursive function definitions.

Optional, Keyword and Rest Arguments

A number of functions used so far take optional arguments, keyword arguments, or variable numbers of arguments.

A function taking an optional argument is defined one of three ways:

```
(defun f (x &optional y) ...)
(defun f (x &optional (y 1)) ...)
(defun f (x &optional (y 1 z)) ...)
```

In the second and third forms, the default value is 1; in the first form it is `nil`

In the third form, `z` is `t` if the optional argument is supplied; otherwise `z` is `nil`.

We can add an optional argument for the step size to `num-deriv`:

```
(defun num-deriv (fun x &optional (h 0.00001))
  (/ (- (funcall fun (+ x h))
        (funcall fun (- x h)))
     (* 2 h)))
```

Keyword arguments are defined similarly to optional arguments:

```
(defun f (x &key y) ...)
(defun f (x &key (y 1)) ...)
(defun f (x &key (y 1 z)) ...)
```

The function is then called as

```
(f 1 :y 2)
```

Using a keyword argument in `num-deriv`:

```
(defun num-deriv (fun x &key (h 0.00001))
  (/ (- (funcall fun (+ x h))
        (funcall fun (- x h)))
     (* 2 h)))
```

With a keyword argument, `num-deriv` is called as

```
(num-deriv #'f 1 :h 0.001)
```

A function with a variable number of arguments is defined as

```
(defun f (x &rest y) ...)
```

All arguments beyond the first are made into a list and bound to `y`.

For example:

```
> (defun my-plus (&rest x) (apply #'+ x))
MY-PLUS
> (my-plus 1 2 3)
6
```

If more than one of these modifications is used, they must appear in the order `&optional`, `&rest`, `&key`.

There is an upper limit on the number of arguments a function can receive.

Mapping

Mapping is the process of applying a function elementwise to a list.

The primary mapping function is `mapcar`:

```
> (setf x (normal-rand '(2 3 2)))
((0.27397 3.5358) (-0.11065 1.2178 1.050)
 (0.78268 0.95955))
> (mapcar #'mean x)
(1.904895 0.71913 0.8711149)
```

Mapcar can take several lists as arguments:

```
> (mapcar #'+ '(1 2 3) '(4 5 6))
(5 7 9)
```

Using `mapcar`, we can define a simple numerical integrator for functions on $[0, 1]$:

```
> (defun integrate (f &optional (n 100))
    (let* ((x (rseq 0 1 n))
           (fv (mapcar f x)))
      (mean fv)))
INTEGRATE
> (integrate #'(lambda (x) (^ x 2)))
0.335017
```

More on Compound Data

Lists

Lists are the most important compound data type.

Lists can be empty:

```
> (list)
NIL
> '()
NIL
> ()
NIL
```

They can be used to represent sets:

```
> (union '(1 2 3) '(3 4 5))
(5 4 1 2 3)
> (intersection '(1 2 3) '(3 4 5))
(3)
> (set-difference '(1 2 3) '(3 4 5))
(2 1)
```

In addition to using `select`, you can get pieces of a list with

```
> (first '(1 2 3))
1
> (second '(1 2 3))
2
> (rest '(1 2 3))
(2 3)
```

Two other useful functions are `remove-duplicates`

```
> (remove-duplicates '(1 1 2 3 3))
(1 2 3)
```

and `count`:

```
> (count 2 '(1 2 3 4) :test #'=)
1
> (count 2 '(1 2 3 4) :test #'<=)
3
```

`remove-duplicates` also accepts a `:test` argument.

Vectors

Vectors are a second form of compound data.

A vector is constructed with the `vector` function

```
> (vector 1 2 3)
#(1 2 3)
```

or by typing its printed representation:

```
> (setf x '#(1 2 3))
#(1 2 3)
> x
#(1 2 3)
```

Elements of vectors can be extracted and changed with `select`:

```
> (select x 1)
2
> (setf (select x 1) 5)
5
> x
#(1 5 3)
```

Vectors can be copied with `copy-vector`.

Vectors are usually stored more efficiently than lists, and their elements can be accessed more rapidly.

But there are fewer functions for operating on vectors than on lists:

```
> (first x)
error: bad argument type - #(1 5 3)
> (rest x)
error: bad argument type - #(1 5 3)
```

Sequences

Lists, vectors, and strings are sequences.

Several functions operate on any sequence:

```
> (length '(1 2 3))
3
> (length '#(1 2 3))
3
> (length "abc")
3
> (select '(1 2 3) 0)
1
> (select '#(1 2 3) 0)
1
> (select "abc" 0)
#\a
```

Sequences can be coerced to different types with `coerce`:

```
> (coerce '(1 2 3) 'vector)
#(1 2 3)
> (coerce "abc" 'list)
(#\a #\b #\c)
```

Arrays

The `matrix` function constructs a two-dimensional array:

```
> (matrix '(2 3) '(1 2 3 4 5 6))
#2A((1 2 3) (4 5 6))
```

Again you can type the printed representation

```
> (setf m '#2A((1 2 3) (4 5 6)))
#2A((1 2 3) (4 5 6))
```

and `select` extracts and modifies elements:

```
> (select m 1 1)
5
> (select m 1 '(0 1))
#2A((4 5))
> (select m '(0 1) '(0 1))
#2A((1 2) (4 5))
> (setf (select m 1 1) 'a)
A
> m
#2A((1 2 3) (4 A 6))
```

Format

`format` is a very flexible output function.

It prints to *output streams* or to strings.

The default output stream is `*standard-output*`; it can be abbreviated to `t`:

```
> (format *standard-output* "Hello~%")
Hello
NIL
> (format t "Hello~%")
Hello
NIL
> (format nil "Hello")
"Hello"
```

`~%` is the *format directive* for a new line.

Other useful format directives are `~a` and `~s`:

```
> (format t "Examples: ~a ~s~%" '(1 2) '(3 4))
Examples: (1 2) (3 4)
NIL
> (format t "Examples: ~a ~s~%" "ab" "cd")
Examples: ab "cd"
```

These two directives differ in their handling of *escape characters*.

There are many other format directives.

Some Statistical Functions

Some Basic Functions

```
> (difference '(1 3 6 10))
(2 3 4)
> (pmax '(1 2 3) 2)
(2 2 3)
> (split-list '(1 2 3 4 5 6) 3)
((1 2 3) (4 5 6))
> (cumsum '(1 2 3 4))
(1 3 6 10)
> (accumulate #'* '(1 2 3 4))
(1 2 6 24)
```

Sorting Functions

```
> (sort-data '(14 10 12 11))
(10 11 12 14)
> (rank '(14 10 12 11))
(3 0 2 1)
> (order '(14 10 12 11))
(1 3 2 0)
```

Interpolation and Smoothing

```
(spline x y :xvals xv)
(lowess x y)
(kernel-smooth x y :width w)
(kernel-dens x)
```

Linear Algebra Functions

```
(identity-matrix 4)
(diagonal '(1 2 3))
(diagonal '#2a((1 2)(3 4)))
(transpose '#2a((1 2)(3 4)))
(transpose '((1 2)(3 4)))
(matmult a b)
(make-rotation '(1 0 0) '(0 1 0) 0.05)

(lu-decomp a)
(inverse a)
(determinant a)
(chol-decomp a)
(qr-decomp a)
(sv-decomp a)
```

Odds and Ends

Errors

The `error` functions signals an error:

```
> (error "bad value")
error: bad value
> (error "bad value: ~s" "A")
error: bad value: "A"
```

Debugging

Several debugging functions are available:

`debug/nodebug` – toggle debug mode; in debug mode, an error puts you into a break loop.

`break` – called within a function to enter a break loop

`baktrace` – prints traceback in a beak loop.

`step` – single steps through an evaluation.

Example

Estimating a Survival Function

Suppose the variable `times` contains survival times and `status` contains status values, with 1 representing death and 0 censoring.

To compute a Kaplan-Meier or Fleming-Harrington estimator, we first need the death times and the unique death times:

```
(setf dt-list
      (coerce (select times
                      (which (= 1 status)))
              'list))
(setf udt
      (sort-data
       (remove-duplicates dt-list :test #'=)))
```

Next, we need the number of deaths and the number at risk at each death time:

```
(setf d (mapcar #'(lambda (x)
                    (count x dt-list :test #'=))
                  udt))
(setf r (mapcar #'(lambda (x)
                    (count x times :test #'<=))
                  udt))
```

Using these values, we can compute the Kaplan-Meier estimator at the death times as

```
(setf km (accumulate #'* (/ (- r d) r)))
```

The Fleming-Harrington estimator is

```
(setf fh (exp (- (cumsum (/ d r)))))
```

Greenwood's formula for the variance is

```
(* (^ km 2) (cumsum (/ d r (pmax (- r d) 1))))
```

The `pmax` expression prevents a division by zero.

Tsiatis' formula leads to

```
(* (^ km 2) (cumsum (/ d (^ r 2))))
```

To construct a plot we need a function that builds the consecutive corners of a step function:

```
(defun make-steps (x y)
  (let* ((n (length x))
         (i (iseq (+ (* 2 n) 1))))
    (list (append '(0) (repeat x (repeat 2 n)))
          (select (repeat (append '(1) y)
                          (repeat 2 (+ n 1)))
                  i))))
```

Then

```
(plot-lines (make-steps udt km))
```

produces a plot of the Kaplan-Meier estimator.

Weibull Regression

Suppose `times` are survival times, `status` contains death/censoring indicators, and `x` contains a matrix of covariates, including a column of ones.

A Weibull model for these data has a log likelihood of the form

$$\sum s_i \log \alpha + \sum (s_i \log \mu_i - \mu_i)$$

where

$$\begin{aligned} \log \mu_i &= \alpha \log t_i + \eta_i \\ \eta_i &= x_i \beta \end{aligned}$$

and α is the Weibull exponent, β is a vector of parameters.

A function to compute this log likelihood is

```
(defun llw (x y s log-a b)
  (let* ((a (exp log-a))
         (eta (matmult x b))
         (log-mu (+ (* a (log y)) eta)))
    (+ (* (sum s) log-a)
       (sum (- (* s log-mu) (exp log-mu))))))
```

Reasonable initial estimates for the parameters might be $\alpha = 1$,

$$\beta_0 = \frac{\Sigma\, s_i}{\Sigma\, t_i}$$

for the constant term, and $\beta_i = 0$ for all other i.

For a single covariate:

```
> (newtonmax
   #'(lambda (theta)
     (llw x
          times
          status
          (first theta)
          (rest theta)))
   (list 0 (/ (sum status) (sum times)) 0))
maximizing...
Iteration 0.
Criterion value = -510.668
...
Iteration 8.
Criterion value = -47.0641
Reason for termination: gradient size ...
(0.311709 -4.80158 1.73087)
```

Objects

Some Background

Some basic features of object oriented programming include

- mutable state information
- messages and methods
- inheritance

Some people add other features, or place different emphasis on different features.

Origins of OOP

- Earliest example: *Simula*, a simulation language.
- First "pure" object oriented language: *Smalltalk*.
- Smalltalk was used for interacting with graphical objects, building graphical user interfaces.
- The Macintosh user interface inherits many features from the Smalltalk-based Xerox system.

Some Variations

- Languages/systems can be purely object oriented or just support object oriented programming.
- Two different approaches are used:
  - class/inheritance systems
  - prototype/delegation systems
- Most newer systems support some form of multiple inheritance.

Some OOP Systems

- *C++* from ATT
- *Objective C*, used in NeXT computers.
- *Smalltalk-80*
- *Object Pascal*, for programming the Macintosh.
- *Flavors*, on *Symbolics* Lisp machines
- *Lisp Object Oriented Programming System (LOOPS)*
- *Object Lisp*
- *Portable Common LOOPS*
- *CLOS*, the new Common Lisp standard
- *Common Objects*, from HP Labs.

References on OOP

Abelson, H. and Sussman, G. J., (1985), *Structure and Interpretation of Computer Programs*, Cambridge, Ma: MIT Press.

Winston, P. H. and Horn, B. K. P., (1988), *Lisp*, 3rd Edition, Reading, Ma: Addison Wesley.

Keene, S. E., (1989), *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Reading, Ma: Addison Wesley.

Proceedings of the ACM Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), in *ACM SIGPLAN Notices*, (1986-91).

Wegner, P., (1990), "Concepts and paradigms of object-oriented programming," *ACM SIGPLAN Messenger*, vol. 1.

Introduction

Suppose we would like to build an "intelligent" function for describing or plotting a data set.

Initially we are looking at three types of data:

- single samples
- multiple samples
- paired samples

A `describe-data` function might be written as

```
(defun describe-data (data)
  (cond
   ((single-sample-p data) ...)
   ((multiple-sample-p data) ...)
   ((paired-sample-p data) ...)
   (t (error "don't know how to ~
              describethisdataset")))))
```

Now suppose we come up with a fourth type, a simple time series.

We would have to edit this function, possibly breaking good code.

An alternative is to arrange for the appropriate code to be looked up in the data set itself – to dispatch on the data set type.

We would need some way of defining new data types and actions.

This is the basis of object oriented programming.

Some Terms

An *object* is a data structure that contains information in *slots*.

Objects respond to *messages* asking them to take certain actions.

A message is *sent* to an object using the function `send` in an expression like

`(send` <*object*> <*selector*> <*arg-1*> ... <*arg-n*>`)`

The <*selector*> is a keyword symbol used to identify the message.

As an example, the `histogram` function returns a histogram object:

```
> (setf p (histogram (normal-rand 20)))
#<Object: 1750540, prototype = HISTOGRAM-PROTO>
```

The object can be sent a message to add an additional set of points:

```
> (send p :add-points (normal-rand 20))
NIL
```

The *message selector* for the message is `:add-points`.

The *message argument* is the sample of normal variables generated by the expression `(normal-rand 20)`.

The *message* consists of the selector and the arguments.

The procedure called to respond to this message is the *method* for the message.

Inheritance and the Root Object

Objects are organized in an *inheritance hierarchy*.

At the top is the *root object*, the value of `*object*`.

The root object contains methods for standard messages, such as

`:own-slots` – list of slots in an object

`:add-slot` – insert a new slot

`:delete-slot` – delete a slot

`:slot-value` – access or modify a slot's value

Some examples:

```
> (send *object* :own-slots)
(DOCUMENTATION PROTO-NAME INSTANCE-SLOTS)

> (send *object* :slot-value 'proto-name)
*OBJECT*

> *object*
#<Object: 302253140, prototype = *OBJECT*>

> (send *object* :slot-value 'instance-slots)
NIL
```

Constructing New Objects

The root object is a *prototype* object.

Prototypes serve as templates for building new objects – *instances*.

The `:new` message is used to ask a prototype for a new instance.

Let's use an object to represent a data set:

```
> (setf x (send *object* :new))
#<Object: 1750672, prototype = *OBJECT*>
```

Initially, the object has no slots of its own:

```
> (send x :own-slots)
NIL
```

We can add a slot for some data and a slot for a title string:

```
> (send x :add-slot 'data (normal-rand 50))
(1.39981 -0.0601746 ....)
> (send x :add-slot 'title)
NIL
> (send x :own-slots)
(TITLE DATA)
> (send x :slot-value 'title)
NIL
> (send x :slot-value 'title "a data set")
"a data set"
> (send x :slot-value 'title)
"a data set"
```

Defining New Methods

The code used to respond to a message is called a *method*.

send searches the receiving object and its ancestors until it finds a method.

All messages used so far used methods in the root object.

New methods can be defined using **defmeth**.

The general form of a **defmeth** expression is:

(defmeth <*object*> <*selector*> <*parameters*>
<*body*>**)**

The argument <*object*> is evaluated, the others are not.

An example:

```
(defmeth x :describe (&optional (stream t))
  (let ((title (slot-value 'title))
        (data (slot-value 'data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The sample mean is ~a~%"
            (mean data))
    (format stream
            "The sample SD is ~a~%"
            (standard-deviation data))))
```

The definition looks like a `defun`, except for the object expression appearing as the first argument.

The new `:describe` method can be used like any other method:

```
> (send x :describe)
This is a data set
The sample mean is 0.127521
The sample SD is 1.0005
NIL
```

A few notes:

- Within a method, the variable `self` refers to the object receiving the message.
- `self` is needed since methods can be inherited.
- `self` is added to the environment for the function body before the other arguments.
- Within a method, the function `slot-value` can be used to access a slot.

  Using this function is more efficient than using the `:slot-value` message.
- The `slot-value` function can *only* be used in the body of a method.
- Within a method, `slot-value` can be used as a place form with `setf`.

It is good programming practice not to assume too much about the slots an object has.

For our data set object, we can write *accessor methods* for the title and data slots:

```
(defmeth x :title (&optional (title nil set))
  (if set (setf (slot-value 'title) title))
  (slot-value 'title))

(defmeth x :data (&optional (data nil set))
  (if set (setf (slot-value 'data) data))
  (slot-value 'data))
```

Using these accessors we can rewrite the `:describe` method:

```
(defmeth x :describe (&optional (stream t))
  (let ((title (send self :title))
        (data (send self :data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The sample mean is ~a~%"
            (mean data))
    (format stream
            "The sample SD is ~a~%"
     (standard-deviation data))))
```

Printing Objects

The printing system prints objects by sending them the `:print` message.

A method for `:print` must take an optional *stream* argument.

A simple `:print` method for our data set:

```
(defmeth x :print (&optional (stream t))
  (format stream "#<~a>" (send self :title)))
```

The result is

```
> x
#<a data set>
```

Help Information for Messages

If we change the definition of the `:describe` method to

```
(defmeth x :describe (&optional (stream t))
"Method args: (&optional (stream t))
Prints a simple description of the object
to STREAM."
  (let ((title (send self :title))
        (data (send self :data)))
    ...))
```

then the string is used as a documentation string:

```
> (send x :help :describe)
:DESCRIBE
Method args: (&optional (stream t))
Prints a simple description of the object
to STREAM.
NIL
```

Prototypes

Creating Simple Prototypes

Instead of repeating the process used so far for each new data set, we can construct a *dataset prototype* using `defproto`.

The simplest form of a `defproto` expression is

`(defproto` <*name*> <*instance slots*>`)`

`defproto` evaluates <*instance slots*> and

- constructs a new object, assigns it to the global variable <*name*>
- installs a slot `proto-name` with value `name`
- installs a slot `instance-slots` with the symbols from <*instance slots*>
- installs a slot with value nil for each symbol in the `instance-slots` list

Different instances created from a prototype usually differ only in the values of their instance slots.

We can define a prototype for our data set as

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

Prototype objects are like any other objects.

We can place values in their slots

```
(send data-set-proto :slot-value
      'title "a data set")
```

and we can define methods for them:

```
(defmeth data-set-proto
         :title (&optional (title nil set))
  (if set (setf (slot-value 'title) title))
  (slot-value 'title))

(defmeth data-set-proto :data ...)

(defmeth data-set-proto :describe ...)

(defmeth data-set-proto :print ...)

(defmeth data-set-proto :plot ()
  (histogram (send self :data)
             :title (send self :title)))
```

Creating Instances from Prototypes

An instance is created by sending a prototype the `:new` message.

The method for this message does the following:

- creates a new object that inherits from the prototype
- adds a slot for each symbol in the prototype's `instance-slots` list
- sets the value of each new slot to its value in the prototype
- sends the new object the `:isnew` message with the arguments given to `:new`
- returns the new object

The root object `:isnew` initialization method allows slots to be initialized by keyword arguments:

```
> (setf x (send data-set-proto :new
                :data (chisq-rand 20 5)))
#<a data set>
> (send x :title)
"a data set"
> (send x :data)
(10.7254 10.4314 2.32346 ...)
> (send x :describe)
This is a data set
The sample mean is 5.17844
The sample SD is 3.2129
```

Any data set needs its own data.

We can define an `:isnew` method that requires a data argument:

```
(defmeth data-set-proto
         :isnew (data &key title)
  (send self :data data)
  (if title (send self :title title)))
```

It is often useful to define an appropriate `:isnew` initialization method.

It is (almost) never necessary to define a new `:new` method.

Prototypes and Inheritance

Prototypes can inherit from other objects, usually other prototypes.

The full form of a `defproto` expression is

```
(defproto <name>
          <instance slots>
          <shared slots>
          <parents>
          <doc string>)
```

All arguments except the first are evaluated.

<instance slots> and *<shared slots>* should be `nil` or lists of symbols.

We won't be using shared slots.

<parents> should be a single object or a list of objects – the default is the root object.

The instance slots of the new prototype are the union of the *<instance slots>* in the call and the instance slots of all ancestors.

So you only need to specify *additional* slots you want.

An equally-spaced time series prototype might add `origin` and `spacing` slots:

```
(defproto time-series-proto
          '(origin spacing) () data-set-proto)
```

Accessor methods for the new slots:

```
(defmeth time-series-proto
         :origin (&optional (origin nil set))
  (if set (setf (slot-value 'origin) origin))
  (slot-value 'origin))

(defmeth time-series-proto
         :spacing (&optional (sp nil set))
  (if set (setf (slot-value 'spacing) sp))
  (slot-value 'spacing))
```

and default values for the `title` slot and the new slots can be set as

```
(send time-series-proto :title "a time series")
(send time-series-proto :origin 0)
(send time-series-proto :spacing 1)
```

An example – a short moving average series:

```
> (let* ((e (normal-rand 21))
         (i (iseq 1 20))
         (d (+ (select e i)
               (* 0.6 (select e (- i 1))))))
    (setf y (send time-series-proto :new d)))
#<a time series>
> (send y :describe)
This is a time series
The sample mean is 0.194134
The sample SD is 1.15485
```

Overriding and Modifying Inherited Methods

The inherited `:plot` and `:describe` methods are not optimal for a time series.

We can *override* the `:plot` method with the definition

```
(defmeth time-series-proto :plot ()
  (let* ((data (send self :data))
         (start (send self :origin))
         (step (send self :spacing))
         (n (length data)))
    (plot-points (+ start (* step (iseq n)))
                 data)))
```

Instead of overriding the `:describe` method completely, we can augment it:

```
(defmeth time-series-proto
         :describe (&optional (stream t))
  (let ((ac (autocor (send self :data))))
    (call-next-method stream)
    (format stream
            "The autocorrelation is ~a~%"
            ac)))
```

The `call-next-method` function calls the next method for the current selector

`(call-next-method` <*arg-1*> ... <*arg-n*>`)`

The `autocor` function can be defined as

```
(defun autocor (x)
  (let ((n (length x))
        (x (- x (mean x))))
    (/ (mean (* (select x (iseq 0 (- n 2)))
                (select x (iseq 1 (- n 1)))))
       (mean (* x x)))))
```

Some Cautions

- Unfortunately the object system does not allow you to hide slots and messages intended for internal use from public access.
- To protect yourself against breaking inherited methods you need to follow a few guidelines:
  - Don't use slot names you didn't add yourself.
  - Don't modify slots unless you are sure it is safe.
  - Don't define methods for messages unless you know this won't do any harm.

Examples

Survival Function Estimator

We can use an object to organize our tools for fitting a survival function.

Since we need the unique death times, death counts, and numbers at risk, we can define a prototype that adds slots to hold these values:

```
(defproto survival-proto
          '(death-times num-deaths num-at-risk)
          ()
          data-set-proto)

(send survival-proto :title
      "a survival data set")
```

Accessors for these three slots are defined as

```
(defmeth survival-proto :death-times ()
  (slot-value 'death-times))
(defmeth survival-proto :num-deaths ()
  (slot-value 'num-deaths))
(defmeth survival-proto :num-at-risk ()
  (slot-value 'num-at-risk))
```

The `data` slot can be used to hold both the times and the death indicators.

The accessor method can be defined as

```
(defmeth survival-proto :data
         (&optional times status)
  (when times
    (call-next-method (list times status))
    (let* ((i (which (= 1 status)))
           (dt (select times i))
           (dt-list (coerce dt 'list))
           (udt (sort-data (remove-duplicates
                            dt-list
                            :test #'=)))
           (d (mapcar
               #'(lambda (x)
                 (count x dt-list :test #'=))
               udt))
           (r (mapcar
               #'(lambda (x)
                 (count x times :test #'<=))
               udt)))
      (setf (slot-value 'death-times) udt)
      (setf (slot-value 'num-deaths) d)
      (setf (slot-value 'num-at-risk) r)))
  (slot-value 'data))
```

The `:isnew` method fills in the data:

```
(defmeth survival-proto :isnew
         (times status &optional title)
  (send self :data times status)
  (if title (send self :title title)))
```

A `:describe` method:

```
(defmeth survival-proto :describe
         (&optional (stream t))
  (let ((title (send self :title))
        (data (send self :data)))
    (format stream "This is ~a~%" title)
    (format stream
            "The mean time is ~a~%"
            (mean (first data)))
    (format stream
            "The number of failures is ~a~%"
            (sum (second data)))))
```

A `:plot` method:

```
(defmeth survival-proto :plot ()
  (let ((km (send self :km-estimator))
        (udt (send self :death-times)))
    (plot-lines (make-steps udt km))))
```

A method for the Kaplan-Meier estimator:

```
(defmeth survival-proto :km-estimator ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths)))
    (accumulate #'* (/ (- r d) r))))
```

Methods for the Fleming-Harrington estimator and standard errors based on Greenwood's formula and Tsiatis' formula are

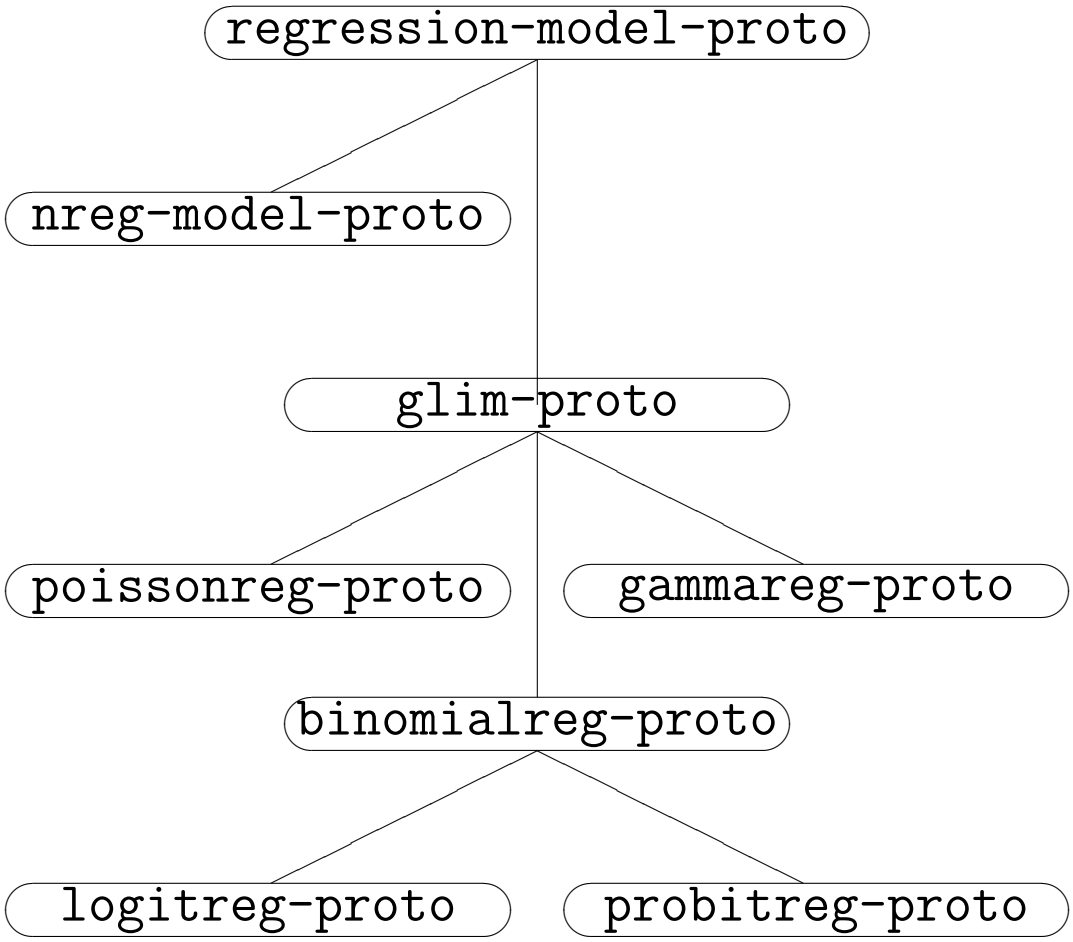
```
(defmeth survival-proto :fh-estimator ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths)))
    (exp (- (cumsum (/ d r))))))

(defmeth survival-proto :greenwood-se ()
  (let* ((r (send self :num-at-risk))
         (d (send self :num-deaths))
         (km (send self :km-estimator))
         (rd1 (pmax (- r d) 1)))
    (* km (sqrt (cumsum (/ d r rd1))))))

(defmeth survival-proto :tsiatis-se ()
  (let ((r (send self :num-at-risk))
        (d (send self :num-deaths))
        (km (send self :km-estimator)))
    (* km (sqrt (cumsum (/ d (^ r 2)))))))
```

Model Prototypes

Linear regression, nonlinear regression and GLIM model prototypes are arranged as:



OOP and User Interfaces

Users of a graphics workstation can take a limited set of actions.

They can

- hit keys
- move the pointer
- click the pointer

These actions can be viewed as messages that are being sent to objects on the screen.

Graphical user interfaces present users with a model based on a desk:

- Overlapping windows act like sheets of paper
- Different windows represent different software objects on the screen
- Each object may respond differently to key stroke or locator click messages

The *user interface conventions* of the window system determine what objects receive what messages.

Several variations are possible:

- On the Macintosh the front window receives all input.

  A window becomes the front window by clicking on it.

- Under *twm* in an *X11* system, the window containing the locator cursor receives all input.

Most window management systems are object-oriented in design.

The window system is responsible for converting user actions into messages.

The programmer is responsible for defining any methods objects need in order to respond to the user's actions.

Some other variations include:

- size and location of windows may have to be determined by the user
- resizing or moving windows from within a program may not be possible
- menus may be popped up or pulled down
- mouse buttons may mean different things in different settings
- mice have different numbers of buttons

Menus

Menus are created using the `menu-proto` prototype.

The initialization method requires a single argument, the menu title.

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

A menu bar is available for holding menus not associated with a graph.

The `:install` message installs a menu; `:remove` removes it:

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

Initially a menu contains no items.

Items are created using the `menu-item-proto` prototype.

The initialization method requires one argument, the item's title.

It also accepts several keyword arguments, including

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

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

Suppose we assign a data set object as the value of some global symbol:

```
(setf *current-data-set* x)
```

Two menu items for dealing with the current data set are:

```
> (setf desc
        (send menu-item-proto :new "Describe"
              :action
              #'(lambda ()
                  (send *current-data-set*
                        :describe))))
#<Object: 4034406, prototype = MENU-ITEM-PROTO>
> (setf plot
        (send menu-item-proto :new "Plot"
              :action
              #'(lambda ()
                  (send *current-data-set*
                        :plot))))
#<Object: 3868686, prototype = MENU-ITEM-PROTO>
```

You can force an item's action to be invoked by sending it the `:do-action` message:

```
> (send desc :do-action)
This is a data set
The sample mean is 4.165463
The sample SD is 1.921366
```

The system sends the `:do-action` message when you select the item in a menu.

The `:append-items` message adds the items to the menu:

```
> (send data-menu :append-items desc plot)
NIL
```

On the Macintosh, the `:key` message adds a keyboard equivalent to an item:

```
> (send desc :key #\D)
#\D
```

This message is ignored on most other systems.

You can also enable and disable an item with the `:enabled` message:

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

Before a menu is presented, the system sends each of its menu items the `:update` message.

You can define a method for this message that enables or disables the item, or adds a check mark, as appropriate.

Dialogs

Dialogs are similar to menus in that they are based on a dialog prototype and dialog item prototypes.

There are, however many more variations.

Fortunately most dialogs you need fall into one of several categories and can be produced by custom dialog construction functions.

Modal Dialogs

Modal dialogs are designed to ask specific questions and wait until they receive a response.

All other interaction is disabled until the dialog is dismissed – they place the system in dialog mode.

Six functions are available for producing some standard modal dialogs:

- `(message-dialog` <*string*>`)` – presents a message with an **OK** button; returns `nil` when the button is pressed.
- `(ok-or-cancel-dialog` <*string*>`)` – presents a message with an **OK** and a **Cancel** button; returns `t` or `nil` according to the button pressed.

- `(choose-item-dialog` *<string>* *<string-list>*`)` – presents a heading and a set of radio buttons for choosing one of the strings.

  Returns the index of the selected string on **OK** or `nil` on **Cancel**.

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

- `(choose-subset-dialog` *<string>* *<string-list>*`)` – presents a heading and a set of check boxes for indicating which items to select.

  Returns a list of the list of selected indices on **OK** or `nil` on **Cancel**.

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

- `(get-string-dialog` <*prompt*> `[:initial` <*expr*>`])`
  presents a dialog with a prompt, an editable text field, an **OK** and a **Cancel** button.

  The initial contents of the editable field is empty or the `~a`-formated version of <*expr*>.

  The result is a string or `nil`.

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

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

There are two dialogs for dealing with files:

- `(open-file-dialog)` – presents a standard **Open File** dialog and returns a file name string or `nil`.

  Resets the working folder on **OK**.

- `(set-file-dialog prompt)` – presents a standard **Save File** dialog.

  Returns a file name string or `nil`.

  Resets the working folder on **OK**.

In the *X11* versions of these are currently just `get-string-dialogs`.

The MS Windows versions are also not yet fully developed.

Modeless Dialogs

Two standard modeless slider dialogs are available:

- `sequence-slider-dialog` – a slider for scrolling through a sequence.

  An action function is called each time the slider is adjusted.

  The current sequence value or a value from a display sequence is displayed.

- `interval-slider-dialog` – similar to the sequence slider, except that a range to be divided up into a reasonable number of points is given.

  By default, the range and number of points are adjusted to produce nice printed values.

Custom Dialogs

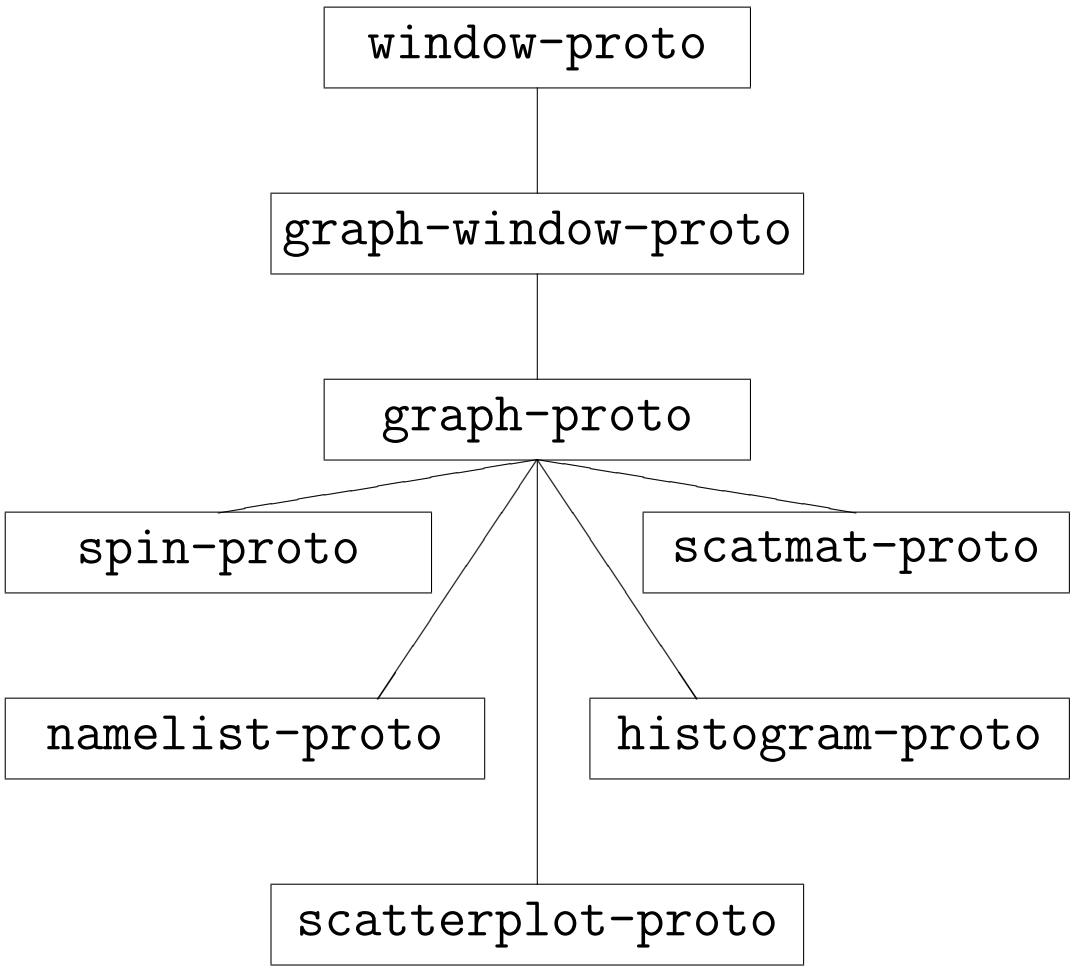
If the standard dialogs are not sufficient, you can construct custom modal or modeless dialogs using a variety of items:

- static and editable text fields
- radio button groups
- check boxes
- sliders
- scrollable lists
- push buttons

Outline of the Graphics System

Overview

The graphics window object tree:



All top-level windows share certain common features:

- a title
- a way to be moved
- a way to be resized

These common features are incorporated in the window prototype `window-proto`.

Both dialog and graphics windows inherit from the window prototype.

Graph Windows

Outline of the Drawing System

A graphics window is a view onto a *drawing canvas*.

The dimensions of the canvas can be *fixed* or *elastic*.

- If a dimension is fixed, it has a scroll bar.
- If it is elastic, it fills the window.

The name list window uses one fixed dimension; all other standard plots use elastic dimensions by default.

The canvas has a coordinate system.

- coordinate units are pixels
- the origin is the top-left corner
- the x coordinate increases from left to right
- the y coordinate increases from top to bottom

A number of drawing operations are available.

Drawable objects include

- rectangles
- ovals
- arcs
- polygons

These can be

- framed
- painted
- erased

Other drawables are

- symbols
- strings
- bitmaps

The precise effect of drawing operations depends on the *state* of the drawing system.

Drawing system state components include

- colors – foreground and background
- drawing mode – normal or XOR
- line type – dashed or solid
- pen width – an integer
- use color – on or off
- buffering – on or off

Animation Techniques

There are two basic animation methods

- *XOR drawing* – drawing "inverts" the colors on the screen
- *double buffering* – a picture is built up in a background buffer and then copied to the screen

Some tradeoffs:

- XOR drawing is usually faster
- XOR drawing automatically preserves the background
- XOR drawing distorts background and object during drawing
- XOR drawing inherently involves a certain amount of flicker
- Moving several objects by XOR causes distortion – only one can move at a time
- *Inverting* is not well-defined on color displays

Lisp-Stat uses

- XOR drawing for moving the brush rectangle
- double buffering for rotation

As an example, let's move a highlighted symbol down the diagonal of a window using both methods.

A new graphics window is constructed by

```
(setf w (send graph-window-proto :new))
```

Using XOR drawing:

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height))
      (mode (send w :draw-mode)))
  (send w :draw-mode 'xor)
  (dotimes (i (min width height))
    (send w :draw-symbol 'disk t i i)
    (pause 2)
    (send w :draw-symbol 'disk t i i))
  (send w :draw-mode mode))
```

Using double Buffering:

```
(let ((width (send w :canvas-width))
      (height (send w :canvas-height)))
  (dotimes (i (min width height))
    (send w :start-buffering)
    (send w :erase-window)
    (send w :draw-symbol 'disk t i i)
    (send w :buffer-to-screen)))
```

Handling Events

User actions produce various *events* that the system handles by sending messages to the appropriate objects.

There are several types of events:

- resize events
- exposure or redraw events
- mouse events – motion and click
- key events
- idle "events"

Resize and redraw events cause `:resize` and `:redraw` messages to be sent to the window object.

Mouse and key events produce `:do-click`, `:do-motion`, and `:do-key` messages.

In idle periods, the `:do-idle` message is sent.

The `:while-button-down` message can be used to follow the mouse inside a click
(for dragging, etc.)

Graphics Window Menus

Every graphics window can have a menu.

The user interface guidelines of the window system determine how the menu is presented:

- On the Macintosh, the menu is installed in the menu bar when the window is the front window.
- In MS Windows, the menu is installed in the application's menu bar when the window is the front window.
- In *SunView*, the menu is popped up when the right mouse button is pressed in the window.
- Under *X11*, the menu is popped up when the mouse is clicked in a **Menu** button at the top of the window.

The `:menu` message retrieves a graph window's menu or installs a new menu.

An Example

As a simple example to illustrate the handling of events, let's construct a window that shows a single highlighted symbol at its center.

The window is constructed by

```
(setf w (send graph-window-proto :new))
```

We can add slots for holding the coordinates of our point:

```
(send w :add-slot 'x
      (/ (send w :canvas-width) 2))
(send w :add-slot 'y
      (/ (send w :canvas-height) 2))

(defmeth w :x (&optional (val nil set))
  (if set (setf (slot-value 'x) (round val)))
  (slot-value 'x))

(defmeth w :y (&optional (val nil set))
  (if set (setf (slot-value 'y) (round val)))
  (slot-value 'y))
```

New coordinate values are rounded since drawing operations require integer arguments.

The `:resize` method positions the point at the center of the canvas:

```
(defmeth w :resize ()
  (let ((width (send self :canvas-width))
        (height (send self :canvas-height)))
    (send self :x (/ width 2))
    (send self :y (/ height 2))))
```

The `:redraw` method erases the window and redraws the symbol at the location specified by the coordinate values:

```
(defmeth w :redraw ()
  (let ((x (send self :x))
        (y (send self :y)))
    (send self :erase-window)
    (send self :draw-symbol 'disk t x y)))
```

The `:do-click` message positions the symbol at the click and then allows it to be dragged:

```
(defmeth w :do-click (x y m1 m2)
  (flet ((set-sym (x y)
           (send self :x x)
           (send self :y y)
           (send self :redraw)))
    (set-symbol x y)
    (send self :while-button-down #'set-sym)))
```

The `:do-idle` method can be used to move the symbol in a random walk:

```
(defmeth w :do-idle ()
  (let ((x (send self :x))
        (y (send self :y)))
    (case (random 4)
      (0 (send self :x (- x 5)))
      (1 (send self :x (+ x 5)))
      (2 (send self :y (- y 5)))
      (3 (send self :y (+ y 5))))
    (send self :redraw)))
```

We can turn the random walk on by typing

```
(send w :idle-on t)
```

and we can turn it off with

```
(send w :idle-on nil)
```

A better solution is to use a menu item:

```
(setf run-item
      (send menu-item-proto :new "Run"
            :action
            #'(lambda ()
              (send w :idle-on
                    (not (send w :idle-on))))))
```

To put a check mark on this item when the walk is running, define an `:update` method:

```
(defmeth run-item :update ()
   (send self :mark (send w :idle-on)))
```

It would also be nice to have a menu item for restarting the walk:

```
(setf restart-item
      (send menu-item-proto :new "Restart"
            :action
            #'(lambda () (send w :restart))))
```

The `:restart` method is defined as

```
(defmeth w :restart ()
  (let ((width (send self :canvas-width))
        (height (send self :canvas-height)))
    (send self :x (/ width 2))
    (send self :y (/ height 2))
    (send self :redraw)))
```

and a menu with the two items is installed by

```
(setf menu
      (send menu-proto :new "Random Walk"))
(send menu :append-items restart-item run-item)
(send w :menu menu)
```

There may be some flickering when the random walk is running.

This flickering can be eliminated by modifying the `:redraw` method to use double buffering:

```
(defmeth w :redraw ()
  (let ((x (round (send self :x)))
        (y (round (send self :y))))
    (send self :start-buffering)
    (send self :erase-window)
    (send self :draw-symbol 'disk t x y)
    (send self :buffer-to-screen)))
```

Statistical Graphics Windows

`graph-proto` is the statistical graphics prototype.

It inherits from `graph-window-proto`.

The graph prototype is responsible for managing the data used by all statistical graphs.

Variations in how these data are displayed are implemented in separate prototypes for the standard graphs.

The graph prototype is a view into m dimensional space.

It allows the display of both points and connected line segments

The default methods in this prototype implement a simple scatterplot of two of the m dimensions.

Many features of graphics windows are enhanced to simplify adding new features to graphs.

The graph prototype adds the following features:

- m-dimensional point and *line start* data
- affine transformations consisting of
  - centering and scaling
  - a linear transformation
- ranges for raw, scaled and canvas coordinates
- mouse modes for controlling interaction
- linking strategy
- window layout management
  - margin, content and aspect
  - background (axes)
  - overlays
  - content
- standard menus and menu items

Data and Axes

The `:isnew` method for the graph prototype requires one argument, the number of variables:

```
> (setf w (send graph-proto :new 4))
#<Object: 302823396, prototype = GRAPH-PROTO>
```

Using the stack loss data as an illustration, we can add data

```
> (send w :add-points
        (list air temp conc loss))
NIL
```

and adjust scaling to make the data visible:

```
> (send w :adjust-to-data)
NIL
```

We can also add line segments:

```
> (send w :add-lines (list air temp conc loss))
NIL
```

You can control whether axes are drawn with the `:x-axis` and `:y-axis` messages:

```
> (send w :x-axis t)
(T NIL 4)
```

The range shown can be accessed and changed:

```
> (send w :range 0)
(50 80)
> (send w :range 1)
(17 27)
> (send w :range 1 15 30)
(15 30)
```

The function `get-nice-range` helps choosing a range and the number of ticks:

```
> (get-nice-range 17 27 4)
(16 28 7)
```

To remove the axis:

```
> (send w :x-axis nil)
(NIL NIL 4)
```

Initially, the plot shows the first two variables:

```
> (send w :current-variables)
(0 1)
```

This can be changed:

```
> (send w :current-variables 2 3)
(2 3)
> (send w :current-variables 0 1)
(0 1)
```

Plot data can be cleared by several messages:

```
(send w :clear-points)
(send w :clear-lines)
(send w :clear)
```

If the `:draw` keyword argument is `nil` the plot is not redrawn.

The default value is `t`.

Scaling and Transformations

The scale type controls the action of the default `:adjust-to-data` method.

The initial scale type is `nil`:

```
> (send w :scale-type)
NIL
> (send w :range 0)
(50 80)
> (send w :scaled-range 0)
(50 80)
```

Two other scale types are `variable` and `fixed`.

For variable scaling:

```
> (send w :scale-type 'variable)
VARIABLE
> (send w :range 0)
(35 95)
> (send w :scaled-range 0)
(-2 2)
```

The `:scale`, `:center`, and `:adjust-to-data` messages let you build your own scale types.

Initially there is no transformation:

```
> (send w :transformation)
NIL
```

If the current variables are 0 and 1, a rotation can be applied to replace `air` by `conc` and `temp` by `loss`:

```
(send w :transformation '#2A((0  0 -1  0)
                             (0  0  0 -1)
                             (1  0  0  0)
                             (0  1  0  0)))
```

The transformation can be removed by

```
(send w :transformation nil)
```

A transformation can also be applied incrementally:

```
(let* ((c (cos (/ pi 20)))
       (s (sin (/ pi 20)))
       (m (+ (* c (identity-matrix 4))
             (* s '#2A((0  0 -1  0)
                      (0  0  0 -1)
                      (1  0  0  0)
                      (0  1  0  0))))))
  (dotimes (i 10)
    (send w :apply-transformation m)))
```

A simpler message allows rotation within coordinate planes:

```
(dotimes (i 10)
  (send w :rotate-2 0 2 (/ pi 20) :draw nil)
  (send w :rotate-2 1 3 (/ pi 20)))
```

Several messages are available for accessing data values in raw, scaled, and screen coordinates.

Other messages are available for converting among coordinate systems.

Mouse Events and Mouse Modes

The graph prototype organizes mouse interactions into *mouse modes*.

Each mouse mode includes

- a symbol for choosing the mode from a program
- a title string used in the mode selection dialog
- a cursor to visually identify the mode
- mode-specific click and motion messages

It should not be necessary to override a graph's `:do-click` or `:do-motion` methods.

Initially there are two mouse modes, `selecting` and `brushing`.

We can add a new mouse mode by

```
(send w :add-mouse-mode 'identify
      :title "Identify"
      :click :do-identify
      :cursor 'finger)
```

The `:do-identify` method can be defined as

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first
             (send self :points-in-rect
                   (- x 2) (- y 2) 4 4))))
    (if p
        (let ((mode (send self :draw-mode))
              (lbl (send self :point-label p)))
          (send self :draw-mode 'xor)
          (send self :draw-string lbl x y)
          (send self :while-button-down
                #'(lambda (x y) nil))
          (send self :draw-string lbl x y)
          (send self :draw-mode mode)))))
```

The button down action does nothing; it just waits.

An alternative is to allow the label to be dragged, perhaps to make it easier to read:

```
(defmeth w :do-identify (x y m1 m2)
  (let* ((cr (send self :click-range))
         (p (first
             (send self :points-in-rect
                   (- x 2) (- y 2) 4 4))))
    (if p
        (let ((mode (send self :draw-mode))
              (lbl (send self :point-label p)))
          (send self :draw-mode 'xor)
          (send self :draw-string lbl x y)
          (send self :while-button-down
                #'(lambda (new-x new-y)
                    (send self :draw-string lbl x y)
                    (setf x new-x)
                    (setf y new-y)
                    (send self :draw-string lbl x y)))
          (send self :draw-string lbl x y)
          (send self :draw-mode mode)))))
```

Standard Mouse Modes and Linking

The click and motion methods of the two standard modes use a number of messages.

In `selecting` mode, a click

- Sends `:unselect-all-points`, unless the extend modifier is used.
- Sends `:adjust-points-in-rect` with click x and y coordinates, width and height returned by `:click-range`, and the symbol `selected` as arguments.
- While the button is down, a dashed rectangle is stretched from the click to the mouse.

  When the button is released, `:adjust-points-in-rect` is sent with the rectangle coordinates and `selected` as arguments.

In `brushing` mode, a click

- Sends `:unselect-all-points` unless the extend modifier is used.
- While the mouse is dragged, sends `:adjust-points-in-rect` with the brush rectangle and `selected` as arguments.

In `brushing` mode, moving the mouse

- Sends `:adjust-points-in-rect` with the brush rectangle and `hilited` as arguments.

Points can be in four states:

`invisible`

`normal`

`hilited`

`selected`.

Linking is based on a *loose linking* model:

- points are related by index number
- only point states are adjusted

The system uses two messages to determine which plots are linked:

- `:links` returns a list of plots linked to the plot (possibly including the plot itself)
- `:linked` determines if the plot is linked, and turns linking on and off.

When a point's state is changed in a plot,

- Each linked plot (and the plot itself) is sent the `:adjust-screen-point` message with the index as argument.
- The action taken by the method for `:adjust-screen-point` may depend on both current and previous states.

Since it is not always feasible to redraw single points,

- the `:needs-adjusting` method can be used to check or set a flag
- the `:adjusting-screen` method can redraw the entire plot if the flag is set

The easiest, though not necessarily the most efficient, way to augment standard mouse modes is to define a new `:adjust-screen` method

Some useful messages:

For points specified by index:

`:point-showing`

`:point-hilited`

`:point-selected`

For sets of indices:

`:selection` or `:points-selected`

`:points-hilited`

`:points-showing`

Other operations:

`:erase-selection`

`:show-all-points`

`:focus-on-selection`

`:adjust-screen`

Some useful predicates:

`:any-points-selected-p`

`:all-points-showing-p`

Window Layout and Redrawing

The `:resize` method maintains a margin and a content rectangle

- The plot is surrounded by a margin, used by plot controls.
- The content rectangle can use a fixed or a variable aspect ratio.
- The size of the content depends on the aspect ratio and the axes.
- The plot can be covered by overlays, resized with `:resize-overlays`.

The aspect type used can be changed by

```
(send w :fixed-aspect t)
```

The `:redraw` method sends three messages:

- `:redraw-background` – erases the canvas and draws the axes
- `:redraw-overlays` – sends each overlay the `:redraw` message
- `:redraw-content` – redraws points, lines, etc.

Many methods, like `:rotate-2`, also send `:redraw-content`.

Plot overlays are useful for holding controls.

- Overlays inherit from `graph-overlay-proto`.
- Overlays are like transparent sheets of plastic.
- Overlays are drawn from the bottom up.
- Overlays can intercept mouse clicks.
- Clicks are processed from the top down:
  - Each overlay is sent the `:do-click` message until one returns a non-`nil` result.
  - Only if no overlay accepts a click is the click passed to the current mouse mode.

The controls of a rotating plot are implemented as an overlay.

Other examples of overlays are in `plotcontrols.lsp` in the **Examples** folder.

Menus and Menu Items

To help construct standard menus

- `:menu-title` returns the title to use
- `:menu-template` returns a list of items or symbols
- `:new-menu` constructs and installs the new menu

The `:isnew` method sends the plot the `:new-menu` message when it is created.

Standard items can be specified as symbols in the template:

- `color`
- `dash`
- `focus-on-selection`
- `link`
- `mouse`
- `options`
- `redraw`
- `erase-selection`
- `rescale`
- `save-image`
- `selection`
- `show-all`
- `showing-labels`
- `symbol`

Standard Statistical Graphs

Each of the standard plot prototypes needs only a few new methods.

The main additional or changes methods are:

`scatterplot-proto`:

Overrides: `:add-points`, `:add-lines`, `:adjust-to-data`.

New messages: `:add-boxplot`, `:add-function-contours`, `:add-surface-contour`, `:add-surface-contours`.

`scatmat-proto`:

Overrides: `:add-lines`, `:add-points`, `:adjust-points-in-rect`, `:adjust-screen-point`, `:do-click`, `:do-motion`, `:redraw-background`, `:redraw-content`, `:resize`.

`spin-proto`:

Overrides: `:adjust-to-data`, `:current-variables`, `:do-idle`, `:isnew`, `:resize`, `:redraw-content`.

New methods: `:abcplane`, `:add-function`, `:add-surface`, `:angle`, `:content-variables`, `:depth-cuing`, `:draw-axes`, `:rotate`, `:rotation-type`, `:showing-axes`

`histogram-proto`:

Overrides: `:add-points`, `:adjust-points-in-rect`, `:adjust-screen`, `:adjust-screen-point`, `:adjust-to-data`, `:clear-points`, `:drag-point`, `:isnew`, `:redraw-content`, `:resize`.

New methods: `:num-bins`, `:bin-counts`.

`name-list-proto`:

Overrides: `:add-points`, `:adjust-points-in-rect`, `:adjust-screen-point`, `:redraw-background`, `:redraw-content`.

Some Dynamic Graphics Examples

Some Background

Dynamic graphs usually have some of the following characteristics:

- high level of interaction
- motion
- more than two dimensional data
- require high performance graphics display hardware

Some of the basic techniques that appear to be useful are

- animation
  - controlled interactively
  - controlled by a program
- direct interaction and manipulation
  - linking and brushing
- rotation

Many ideas are described in the papers in the book edited by Cleveland and McGill (1988).

Research on the effective use of dynamic graphics is just beginning.

Some of the questions to keep in mind are

- How should dynamic graphs be used?
- What should you plot?
- How do you interpret dynamic images?
- What techniques are useful, and for which problems?

Even in simple two-dimensional graphics there are difficult open issues.

Answering these questions will take time.

We can make a start by learning how to try out some of the more promising dynamic graphical ideas, and variations on these ideas.

Some Examples of Animation

The basic idea in animation is to display a sequence of views rapidly enough and smoothly enough to

- create an illusion of motion
- allow the change in related elements to be tracked visually

The particular image is determined by the values of one or more parameters.

- For a small number of parameters, (1 or 2), the values can be controlled directly, e.g. using scroll bars.
- For more parameters, automated methods of exploring the parameter space are useful.

Dynamic Box-Cox Plots

Fowlkes's implementation of dynamic transformations in the late 1960's was one of the earliest examples of dynamic statistical graphics.

Start by defining a function to compute the transformation and scale the data to the unit interval:

```
(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))))
```

Next, construct an ordered sample and a set of approximate expected normal order statistics:

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

We can construct an initial plot as

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

To change the plot content to reflect a square-root transformation, we can use the `:clear` and `:add-points` messages:

```
(send p :clear)
(send p :add-points nq (bc x 0.5))
```

To achieve a smooth transition, you can give the `:clear` message a keyword argument `:draw` with value `nil`:

```
(send p :clear :draw nil)
(send p :add-points nq (bc x 0.0))
```

These steps can be built into a `:change-power` method:

```
(defmeth p :change-power (pow)
  (send self :clear :draw nil)
  (send self :add-points nq (bc x pow)))
```

To view a range of powers, you can use a `dolist` loop

```
(dolist (pow (rseq -1 2 20))
  (send p :change-power pow))
```

or you can use one of two *modeless* dialogs to get finer control over the animation:

```
(sequence-slider-dialog (rseq -1 2 21) :action
  #'(lambda (pow) (send p :change-power pow)))
```

or

```
(interval-slider-dialog '(-1 2)
  :points 20
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

To avoid losing color, symbol or state information in the points, we can change the coordinates of the points in the plot instead of replacing the points.

To preserve the original data ordering, we need to base the x values on ranks instead of ordering the data

```
(let ((ranks (rank precipitation)))
  (setf nq (normal-quant (/ (+ ranks 1) 31))))
```

and use these to construct our plot:

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

To change the coordinates, we need a list of the indices of all points.

To avoid generating this list every time, we can store it in a slot:

```
(send p :add-slot 'indices (iseq 30))
(defmeth p :indices () (slot-value 'indices))
```

The method for changing the power now becomes

```
(defmeth p :change-power (pow)
  (send self :point-coordinate
        1
        (send self :indices)
        (bc precipitation pow))
  (send self  :redraw-content))
```

and the slider is again set up by

```
(interval-slider-dialog '(-1 2)
  :points 20
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

The same idea can be used with a regression model.

To transform the dependent variable and scale the residuals, define the functions

```
(defun bcr (x p)
  (if (< (abs p) 0.0001)
      (log x)
      (/ (^ x p) p)))
```

and

```
(defun sc (x)
  (let ((min (min x))
        (max (max x)))
    (/ (- x min) (- max min))))
```

Let's use the stack loss data as an illustration.

A set of approximate expected normal order statistics and a set of indices are set up as

```
(setf nqr
      (let ((n (length loss)))
        (normal-quant (/ (iseq 1 n) (+ 1 n)))))

(setf idx (iseq (length loss)))
```

The regression model and initial plot are set up by

```
(setf m (regression-model (list air conc temp)
                          loss))

(setf p (let ((r (send m :residuals)))
          (plot-points (select nqr (rank r))
                       (sc r))))
```

The new `:change-power` method transforms the dependent variable in the regression model and then changes the plot using the new residuals:

```
(defmeth p :change-power (pow)
  (send m :y (bcr loss pow))
  (let* ((r (send m :residuals))
         (r-nqr (select nqr (rank r))))
    (send self :point-coordinate 0 idx r-nqr)
    (send self :point-coordinate 1 idx (sc r))
    (send self :redraw-content)))
```

Again, a slider for controlling the animation is set up by

```
(interval-slider-dialog '(-1 2)
  :action
  #'(lambda (pow) (send p :change-power pow)))
```

Since the point indices in the plot correspond to the indices in the regression, it is possible to track the positions of the residuals for groups of observations as the power is changed.

Density Estimation

Choosing a bandwidth for a kernel density estimator is a difficult problem.

It may help to have an animation that shows the effect of changes in the bandwidth on the estimate.

An initial plot can be set up as

```
(setf w (plot-lines (kernel-dens precipitation
                                 :width 1)))
```

The initial bandwidth can be stored in a slot

```
(send w :add-slot 'kernel-width 1)
```

and the accessor method for the slot can adjust the plot when the width is changed:

```
(defmeth w :kernel-width (&optional width)
  (when width
        (setf (slot-value 'kernel-width) width)
        (send self :set-lines))
  (slot-value 'kernel-width))
```

It may also be useful to store the data in a slot and provide an accessor method:

```
(send w :add-slot 'kernel-data precipitation)

(defmeth w :kernel-data ()
  (slot-value 'kernel-data))
```

The method for changing the plot is

```
(defmeth w :set-lines ()
  (let ((width (send self :kernel-width))
        (data (send self :kernel-data)))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (kernel-dens data :width width))))
```

and a slider for controlling the width is given by

```
(interval-slider-dialog '(.25 1.5)
  :action
  #'(lambda (s) (send w :kernel-width s)))
```

Assessing Variability by Animation

One way to get a feeling for the uncertainty in a density estimate is to re-sample the data with replacement, i.e. to look at bootstrap samples, and see how the estimate changes.

We can do this by defining a `:do-idle` method:

```
(defmeth w :do-idle ()
  (setf (slot-value 'kernel-data)
        (sample precipitation 30 :replace t))
  (send self :set-lines))
```

We can turn the animation on and off with the `:idle-on` message, but it is better to use a menu item.

We have already used such an item once; since we may need one again, it is probably worth constructing a prototype:

```
(defproto run-item-proto
          '(graph) () menu-item-proto)

(send run-item-proto :key #\R)

(defmeth run-item-proto :isnew (graph)
  (call-next-method "Run")
  (setf (slot-value 'graph) graph))

(defmeth run-item-proto :update ()
  (send self :mark
        (send (slot-value 'graph) :idle-on)))

(defmeth run-item-proto :do-action ()
  (let ((graph (slot-value 'graph)))
    (send graph :idle-on
          (not (send graph :idle-on)))))
```

We can now add a run item to the menu by

```
(send (send w :menu) :append-items
      (send run-item-proto :new w))
```

A problem with this version is that the x values at which the density estimate is evaluated change with the sample.

To avoid this, we can add another slot and accessor method,

```
(send w :add-slot
      'xvals
      (rseq (min precipitation)
            (max precipitation)
            30))

(defmeth w :xvals () (slot-value 'xvals))
```

and change the `:set-lines` method to use these values:

```
(defmeth w :set-lines ()
  (let ((width (send self :kernel-width))
        (xvals (send self :xvals)))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (kernel-dens (send self :kernel-data)
                       :width width
                       :xvals xvals))))
```

Another possible problem is that the curve may jump around too much to make it easy to watch.

We can reduce this jumping by changing one observation at a time instead of changing all at once:

```
(send w :slot-value
      'kernel-data (copy-list precipitation))

(defmeth w :do-idle ()
  (let ((d (slot-value 'kernel-data))
        (i (random 30))
        (j (random 30)))
    (setf (select d i)
          (select precipitation j))
    (send self :set-lines)))
```

A similar idea may be useful for estimated survival curves.

Suppose `s` is a survival distribution object.

A plot of the Fleming-Harrington estimator is

```
(setf p
      (let ((d (send s :num-deaths))
            (r (send s :num-at-risk))
            (udt (send s :death-times)))
        (plot-lines
         (make-steps
          udt
          (exp (- (cumsum (/ d r)))))))))
```

We can put a copy of the estimated hazard increments in a slot:

```
(send p :add-slot
      'del-hazard
      (copy-list (/ (send s :num-deaths)
                    (send s :num-at-risk))))

(defmeth p :del-hazard ()
  (slot-value 'del-hazard))
```

For certain prior distributions, a Bayesian analysis produces approximately a gamma posterior distribution for the hazard increments.

We can use this distribution to animate the plot:

```
(defmeth p :do-idle ()
  (let* ((udt (send s :death-times))
         (d (send s :num-deaths))
         (r (send s :num-at-risk))
         (dh (send self :del-hazard))
         (n (length dh))
         (i (random n))
         (di (select d i))
         (ri (select r i)))
    (setf (select dh i)
          (/ (first (gamma-rand 1 di)) ri))
    (send self :clear-lines :draw nil)
    (send self :add-lines
          (make-steps udt
                      (exp (- (cumsum dh)))))))
```

A run item lets us control the animation:

```
(send (send p :menu) :append-items
      (send run-item-proto :new p))
```

Another way to think of this animation is as a simulation from an approximation to the sampling distribution.

Modifying Mouse Responses

Modifying Selection and Brushing

We have already seen several examples of using the standard selecting and brushing modes.

The series of messages used to implement these modes lets us modify what happens when the brush is moved or points are selected.

The simplest way to build on the standard modes is to modify the `:adjust-screen` method.

This message is sent each time the set of selected or highlighted points is changed with the mouse.

As an example, if we are using selection or brushing to examine a bivariate conditional distribution, it may be useful to augment the bivariate plot with a smoother.

Suppose we have a histogram of **hardness** and a scatterplot of **abrasion-loss** against **tensile-strength** for the abrasion loss data

```
(setf h (histogram hardness))

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

The method

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send self :clear-lines :draw nil)
    (if (< 1 (length i))
        (let ((x (select tensile-strength i))
              (y (select abrasion-loss i)))
          (send self :add-lines
                (kernel-smooth x y)))
        (send self :redraw-content))))
```

adds a kernel smooth to the plot if at least two points are selected or hilited.

To have control over the smoother bandwidth, we can add a slot and accessor method, modify the `:adjust-screen` method, and add a slider:

```
(send p :add-slot 'kernel-width 50)

(defmeth p :kernel-width (&optional width)
  (when width
        (setf (slot-value 'kernel-width) width)
        (send self :adjust-screen))
  (slot-value 'kernel-width))

(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send self :clear-lines :draw nil)
    (if (< 1 (length i))
        (let ((x (select tensile-strength i))
              (y (select abrasion-loss i))
              (w (send self :kernel-width)))
          (send self :add-lines
                (kernel-smooth x y :width w)))
        (send self :redraw-content))))

(interval-slider-dialog '(20 100)
  :action #'(lambda (w) (send p :kernel-width w)))
```

A similar idea is useful for multivariate response data (e.g repeated measures or time series).

Suppose `p` is a plot of covariates and `q` is a plot of m responses, contained in the list `resp`, against their indices.

Then the method

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((i (union
            (send self :points-selected)
            (send self :points-hilited))))
      (send q :clear-lines :draw nil)
      (if i
          (flet ((ms (x) (mean (select x i))))
            (let ((y (mapcar #'ms resp))
                  (j (iseq 1 (length resp))))
              (send q :add-lines j y)))
          (send q :redraw-content))))
```

shows in `q` the average response profile for the highlighted and selected observations.

Another example was recently posted to `statlib` by Neely Atkinson, M. D. Anderson Cancer Center, Houston.

Here is a simplified version:

Suppose we have some censored survival data and a number of covariates.

To examine the relationship between the covariates and the times

- put the covariates in a plot, for example a scatterplot matrix
- in a second plot show the Kaplan-Meier estimate of the distribution of the survival times for points selected in the covariate plot.

To speed up performance, first define a faster Kaplan-Meier estimator that assumes the data are properly ordered:

```
(defun km (x s)
  (let ((n (length x)))
    (accumulate #'* (- 1 (/ s (iseq n 1))))))
```

Next, set up the covariate plot and a plot of the Kaplan-Meier estimator for all survival times:

```
(setf p (scatterplot-matrix (list x y z)))
(setf q (plot-lines
         (make-steps times (km times status))))
```

Finally, write an `:adjust-screen` method for `p` that changes the Kaplan-Meier estimator in `q` to use only the currently selected and highlighted points in `p`:

```
(defmeth p :adjust-screen ()
  (call-next-method)
  (let ((s (union
            (send self :points-selected)
            (send self :points-hilited))))
    (send q :clear-lines :draw nil)
    (if s
        (let* ((s (sort-data s))
               (tm (select times s))
               (st (select status s)))
          (send q :add-lines
                (make-steps tm (km tm st))))
        (send q :redraw-content))))
```

Using New Mouse Modes

We have already seen a few examples of useful new mouse modes.

As another example, let's make a plot for illustrating the sensitivity of least squares to points with high leverage.

Start with a plot of some simulated data:

```
(setf x (append (iseq 1 18) (list 30 40)))
(setf y (+ x (* 2 (normal-rand 20))))
(setf p (plot-points x y))
```

Next, add a new mouse mode:

```
(send p :add-mouse-mode 'point-moving
      :title "Point Moving"
      :cursor 'finger
      :click :do-point-moving)
```

The point moving method can be broken down into several pieces:

```
(defmeth p :do-point-moving (x y a b)
  (let ((p (send self :drag-point
                 x y :draw nil)))
    (if p (send self :set-regression-line))))

(defmeth p :set-regression-line ()
  (let ((coefs (send self :calculate-coefs)))
    (send self :clear-lines :draw nil)
    (send self :abline
          (select coefs 0)
          (select coefs 1))))

(defmeth p :calculate-coefs ()
  (let* ((i (iseq (send self :num-points)))
         (x (send self :point-coordinate 0 i))
         (y (send self :point-coordinate 1 i))
         (m (regression-model x y :print nil)))
    (send m :coef-estimates)))
```

If we want to change the fitting method, we only need to change the `:calculate-coefs` method.

Finally, add a regression line and put the plot into point moving mode

```
(send p :set-regression-line)

(send p :mouse-mode 'point-moving)
```

Several enhancements and variations are possible:

- The plot can be modified to use only visible points for fitting the line.
- Several fitting methods can be provided, with a dialog to choose among them.
- The same idea can be used to study the effect of outliers on smoothers.

As another example, we can set up a plot for obtaining graphical function input.

Start by setting up the plot, turning off the y axis, and adding a new mouse mode:

```
(setf p (plot-lines (rseq 0 1 50)
                    (repeat 0 50)))
(send p :y-axis nil)

(send p :add-mouse-mode 'drawing
      :title "Drawing"
      :cursor 'finger
      :click :mouse-drawing)
```

We can put the plot in our new mode with

```
(send p :mouse-mode 'drawing)
```

The drawing method is defined as

```
(defmeth p :mouse-drawing (x y m1 m2)
  (let* ((n (send self :num-lines))
         (rxy (send self :canvas-to-real x y))
         (rx (first rxy))
         (ry (second rxy))
         (old-i (max 0 (min (- n 1) (floor (* n rx)))))
         (old-y ry))
    (flet ((adjust (x y)
             (let* ((rxy (send self :canvas-to-real x y))
                    (rx (first rxy))
                    (ry (second rxy))
                    (new-i (max 0
                                (min (- n 1)
                                     (floor (* n rx)))))
                    (y ry))
               (dolist (i (iseq old-i new-i))
                 (let ((p (if (= old-i new-i)
                              1
                              (abs
                               (/ (- i old-i)
                                  (- new-i old-i))))))
                   (send self :linestart-coordinate 1 i
                         (+ (* p y) (* (- 1 p) old-y)))))
               (send self :redraw-content)
               (setf old-i new-i old-y y))))
      (adjust x y)
      (send self :while-button-down #'adjust))))
```

Some notes:

- `:canvas-to-real` converts the click coordinates to data coordinates.
- The `dolist` loop is needed to smooth out the curve if the mouse is moved rapidly.
- If the plot is to be normalized as a density, it can be adjusted before the `:redraw-content` message is sent.

A method for retrieving the lines is given by

```
(defmeth p :lines ()
  (let ((i (iseq 50)))
    (list
     (send self :linestart-coordinate 0 i)
     (send self :linestart-coordinate 1 i))))
```

Some Issues in Rotation

Statistical Issues

Rotation is a method for viewing a three dimensional set of data.

The objective of rotation is to create an illusion of 3D structure on a 2D screen.

Factors that can be used to produce or enhance such an illusion are

- stereo imaging
- perspective
- lighting
- motion parallax

Motion parallax is the principle driving rotation.

Scaling can have a significant effect on the shape of a point cloud.

Possible scaling strategies include

- fixed scaling – comparable scales
- variable scaling – non-comparable scales
- combinations

Several methods are available for enhancing the 3D illusion produced by rotation

- depth cuing
- framed box
- Rocking
- slicing

The nature of the rotation controls may also affect the illusion.

Some possible control strategies are

- rotation around screen axes: horizontal (Pitch), vertical (Yaw), out-of-screen (Roll)
- rotation around data axes: X, Y, Z
- direct manipulation (like a globe)

Rotation is still a fairly new technique; we are still learning how to use it effectively.

Some questions to keep in mind as you use rotation to explore your data sets:

- What types of phenomena are easy to detect?
- What is hard to detect?
- What enhancements aid in detection?

It may help to try rotation on some artificial structures and see if you can identify these structures.

Implementation Issues

The default scaling used by the rotating plot is `variable`.

You can change the scale type with the **Options** dialog or the `:scale-type` message.

You can implement non-standard scaling strategies by overriding the default `:adjust-to-data` method.

The standard rotating plot has controls for screen axis rotation.

You can use the `:transformation` and `:apply-transformation` methods to implement alternate control methods.

Plot overlays can be used to hold controls for these strategies.

Some Examples

By rocking the plot back and forth we can get the 3D illusion while keeping the view close to fixed.

A method for rocking the plot can be defines as

```
(defmeth spin-proto :rock-plot
         (&optional (a 0.15))
  (let* ((angle (send self :angle))
         (k (round (/ a angle))))
    (dotimes (i k)
      (send self :rotate-2 0 2 angle))
    (dotimes (i (* 2 k))
      (send self :rotate-2 0 2 (- angle)))
    (dotimes (i k)
      (send self :rotate-2 0 2 angle))))
```

A method for rotating around a specified data axis is

```
(defmeth spin-proto :data-rotate
         (axis &optional (angle pi))
  (let* ((alpha (send self :angle))
         (cols (column-list
                (send self :transformation)))
         (m (case axis
              (x (make-rotation (select cols 1)
                                (select cols 2)
                                alpha))
              (y (make-rotation (select cols 0)
                                (select cols 2)
                                alpha))
              (z (make-rotation (select cols 0)
                                (select cols 1)
                                alpha)))))
    (dotimes (i (floor (/ angle alpha)))
      (send self :apply-transformation m))))
```

To allow rotation to be done by direct manipulation, we can add a new mouse mode

```
(send spin-proto :add-mouse-mode 'hand-rotate
      :title "Hand Rotate"
      :cursor 'hand
      :click :do-hand-rotate)
```

and define the `:do-hand-rotate` method as

```
(defmeth spin-proto :do-hand-rotate (x y m1 m2)
  (flet ((calcsphere (x y)
           (let* ((norm-2 (+ (* x x) (* y y)))
                  (rad-2 (^ 1.7 2))
                  (z (if (< norm-2 rad-2) (sqrt (- rad-2 norm-2)) 0)))
             (if (< norm-2 rad-2)
                 (list x y z)
                 (let ((r (sqrt (max norm-2 rad-2))))
                   (list (/ x r) (/ y r) (/ z r))))))))
    (let* ((oldp (apply #'calcsphere
                        (send self :canvas-to-scaled x y)))
           (p oldp)
           (vars (send self :content-variables))
           (trans (identity-matrix (send self :num-variables))))
      (send self :idle-on nil)
      (send self :while-button-down
            #'(lambda (x y)
                (setf oldp p)
                (setf p (apply #'calcsphere
                               (send self :canvas-to-scaled x y)))
                (setf (select trans vars vars) (make-rotation oldp p))
                (when m1
                      (send self :slot-value 'rotation-type trans)
                      (send self :idle-on t))
                (send self :apply-transformation trans))))))
```

4D and Beyond

Introduction and Background

As with two and three dimensional plots, the objectives of higher dimensional graphics are, among others, to

- detect groupings or clusters
- detect lower dimensional structure
- detect other patterns

Higher dimensions introduce new difficulties:

- sparseness
- loss of intuition

A rough ranking of types of structures that can be detected might look like this:

- points (zero-dimensional structures) in 1D and 2D plots
- curves (one-dimensional structures) in 2D and 3D plots
- surfaces (two-dimensional structures) in 3D plots

Possible structures in four dimensions include

- separate point clusters
- curves
- 2D surfaces
- 3D surfaces

Useful general techniques include

- glyph augmentation
  - colors
  - symbol types
  - symbol sizes
- dimension reduction
  - projection
  - slicing, masking, conditioning

Some issues to keep in mind:

- using symbols or colors loses ordering
- projections to 2D will only become 1D if part of the structure is linear
- masking/conditioning requires large data sets

Some specific implementations:

- scatterplot matrix
- linked 2D plots
- 3D plot linked with a histogram
- plot interpolation
- grand tours and variants
- parallel coordinates

Plot Interpolation

Plot interpolation provides a way of examining the relation among 4 variables.

The idea is to take two scatterplots and rotate from one to the other.

(Interpolate using trigonometric interpolation.)

Watching the interpolation allows you to

- track groups of points from one plot to another
- identify clusters based on both location and velocity

A simple function to set up an interpolation plot for a data set with four variables:

```
(defun interp-plot (data &rest args)
  ;; Should check here that data is 4D
  (let ((w (apply #'plot-points data
                  :scale 'variable args))
        (m (matrix '(4 4) (repeat 0 16))))
    (flet ((interpolate (p)
             (let* ((a (* (/ pi 2) p))
                    (s (sin a))
                    (c (cos a)))
               (setf (select m 0 0) c)
               (setf (select m 0 2) s)
               (setf (select m 1 1) c)
               (setf (select m 1 3) s)
               (send w :transformation m))))
      (let ((s (interval-slider-dialog
                '(0 1)
                :points 25
                :action #'interpolate)))
        (send w :add-subordinate s)))
    w))
```

Some points:

- The definition allows keywords to be passed on to `plot-points`.
- The plot `w` and the matrix `m` are locked into the environment of `interpolate`.
- The slider is registered as a subordinate.

Some examples to try:

- Stack loss data
- Places rated data
- Iris data

The Grand Tour

The Grand Tour provides a way of examining "all" one- or two-dimensional projections of a higher-dimensional data set.

The projections are moved smoothly in a way that eventually brings them near every possible projection.

One way to construct a Grand Tour is to repeat the following steps:

- Choose two directions at random to define a rotation plane.
- Rotate in this plane by a specified angle a random number of times.

A simple Grand Tour function:

```
(defun tour-plot (&rest args)
  (let ((p (apply #'spin-plot args)))
    (send p :add-slot 'tour-count -1)
    (send p :add-slot 'tour-trans nil)
    (defmeth p :tour-step ()
      (when (< (slot-value 'tour-count) 0)
        (let ((vars (send self :num-variables))
              (angle (send self :angle)))
          (setf (slot-value 'tour-count)
                (random 20))
          (setf (slot-value 'tour-trans)
                (make-rotation
                 (sphere-rand vars)
                 (sphere-rand vars)
                 angle))))
      (send self :apply-transformation
            (slot-value 'tour-trans))
      (setf (slot-value 'tour-count)
            (- (slot-value 'tour-count) 1)))
    (defmeth p :do-idle ()
      (send self :tour-step))
    (send (send p :menu) :append-items
          (send run-item-proto :new p))
    p))
```

The `sphere-rand` function can be defined as

```
(defun sphere-rand (n)
  (let* ((z (normal-rand n))
         (r (sqrt (sum (^ z 2)))))
    (if (< 0 r)
        (/ z r)
        (repeat (/ (sqrt n)) n))))
```

Some examples to try the tour on:

- Diabetes data
- Iris data
- Stack loss data
- 4D structures

Some variations and additions:

- Make new rotation orthogonal to current viewing plane
- Controls for replaying parts of the tour
- Touring on only some of the variables (independent variables only)
- Constraints on the tour (Correlation Tour)
- Integration with slicing/conditioning
- Guided tours (Projection Pursuit – XGobi)

A similar idea can be used for examining functions.

As an example, suppose we want to determine if $f(x)$ looks like a multivariate standard normal density.

This is true if and only if $g(z) = f(zu)$ looks like a univariate standard normal density for any unit vector u.

By moving u through space with a series of random rotations, we can look at these univariate densities for a variety of different directions.

The normality checking plot can be implemented as a prototype:

```
(defproto ncheck-plot-proto
  '(function direction xvals
    tour-count tour-trans angle)
  ()
  scatterplot-proto)

(send ncheck-plot-proto :slot-value
      'tour-count -1)
(send ncheck-plot-proto :slot-value
      'angle 0.2)
```

The method for adjusting the image to the current state is

```
(defmeth ncheck-plot-proto :set-image ()
  (let* ((x (slot-value 'xvals))
         (f (slot-value 'function))
         (d (slot-value 'direction))
         (y (mapcar #'(lambda (x)
                        (funcall f (* x d)))
                    x)))
    (send self :clear-lines :draw nil)
    (send self :add-lines (spline x y))))
```

The method for moving the direction is similar to the tour method used earlier:

```
(defmeth ncheck-plot-proto :tour-step ()
  (when (< (slot-value 'tour-count) 0)
        (let* ((d (slot-value 'direction))
               (n (length d))
               (a (abs (slot-value 'angle))))
          (setf (slot-value 'tour-count)
                (random (floor (/ pi
                                  (* 2 a)))))
          (setf (slot-value 'tour-trans)
                (make-rotation d
                               (normal-rand n)
                               a))))
  (setf (slot-value 'direction)
        (matmult (slot-value 'tour-trans)
                 (slot-value 'direction)))
  (send self :set-image)
  (setf (slot-value 'tour-count)
        (- (slot-value 'tour-count) 1)))
```

Two additional methods:

```
(defmeth ncheck-plot-proto :do-idle ()
  (send self :tour-step))

(defmeth ncheck-plot-proto :menu-template ()
  (append
   (call-next-method)
   (list (send run-item-proto :new self))))
```

Finally, the initialization method sets up the plot:

```
(defmeth ncheck-plot-proto :isnew (f d)
  (setf (slot-value 'function) f)
  (setf (slot-value 'direction) d)
  (setf (slot-value 'xvals) (rseq -3 3 7))
  (call-next-method 2)
  (send self :range 0 -3 3)
  (send self :range 1 0 1.2)
  (send self :x-axis t nil 7)
  (send self :y-axis nil)
  (send self :set-image))
```

Some examples to try:

- independent gamma variates
- normal mixtures
- likelihood function
- posterior densities

Some possible enhancements:

- normal curve for comparison
- replay tools
- integration with transformations
- guided tour

Some References

Asimov, D., (1985), "The grand tour: a tool for viewing multidimensional data," *SIAM J. of Scient. and Statist. Comp.* 6, 128-143.

Becker, R. A. and Cleveland, W. S., (1987), "Brushing scatterplots," *Technometrics* 29, 127-142, reprinted in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Becker, R. A., Cleveland, W. S. and Weil, G., (1988), "The use of brushing and rotation for data analysis," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Bolorforush, M., and Wegman, E. J. (1988), "On some graphical representations of multivariate data," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 121-126.

Buja, A., Asimov, D., Hurley, C., and McDonald, J. A., (1988), "Elements of a viewing pipeline for data analysis," in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Donoho, A. W., Donoho, D. L. and Gasko, M., (1988), "MACSPIN: Dynamic Graphics on a Desktop Computer," n *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Fisherkeller, M. A., Friedman, J. H. and Tukey, J. W., (1974), "PRIM-9: An interactive multidimensional data display and analysis system," in *Data: Its Use, Organization and Management*, 140-145, New York: ACM, reprinted in *Dynamic Graphics for Statistics*, W. S. Cleveland and M. E. McGill (eds.), Belmont, Ca.: Wadsworth.

Inselberg, A., and Dimsdale, B. (1988), "Visualizing multi-dimensional geometry with parallel coordinates," *Computing Science and Statistics: Proceedings of the 20th Symposium on the Interface*, E. J. Wegman, D. T. Ganz, and J. J. Miller, editors, Alexandria, VA: ASA, 115-120.

Comments

Some statistical issues:

- Many good ideas for viewing higher-dimensional data are available
- There is room for many more ideas
- Eventually, it will be useful to learn to quantify the effectiveness of different viewing methods in different situations.
- It is also important to explore numerical enhancements as well as combinations of different graphical strategies.

Some computational issues:

- For point clouds, image rendering is quite fast, even on stock hardware.
- This is not true of lines and surfaces.
- The effective use of high performance 3D hardware is worth exploring.
- For data sets derived from models, computing speed can still be limiting
- Pre-computation can help, but limits interaction.

Final Notes

Where can you go from here?

Get a copy of the software for your computer and try it on your problems:

- try the standard tools
- try building tools of your own
- try graphics tailored to your problems

If you develop new tools and ideas that might be useful to others, please share them (e.g. by submitting them to `statlib` or the `stat-lisp-news` mailing list).

Electronic sources of information

There is a mailing list of users who help to answer questions, share ideas, etc.

To join the mailing list, send mail to:

`stat-lisp-news-request@umnstat.stat.umn.edu`

Once you are on the list, you can send mail to all members of the list by sending a message to

`stat-lisp-news@umnstat.stat.umn.edu`

The `statlib` archive contains contributed Lisp-Stat code, as well as the source code for XLISP-STAT.

To find out what is available from `statlib`, send mail to

`statlib@lib.stat.cmu.edu`

containing the single line

```
send index
```

`statlib` can also be accessed by *ftp*.

Where Lisp-Stat is headed

Lisp-Stat is an evolving system.

Plans for the near term are to

- improve and update current versions
- develop a Common Lisp version

Longer-term plans include

- improving the support for compound data objects
- some support for missing data conventions
- increased graphics capabilities
- support for constraints

Further development depends on the users of Lisp-Stat:

- If people find the system useful, new methods for handling a variety of problems will become available (e.g. through `statlib`).
- As new ideas are implemented, the need for new tools and primitives, or for modifications to existing ones, will become evident.

Some books on Lisp and Lisp programming:

Abelson, H. and Sussman, G. J. (1985), *Structure and Interpretation of Computer Programs*, New York: McGraw-Hill.

Franz Inc. (1988), *Common Lisp: The Reference*, Reading, MA: Addison-Wesley.

Steele, Guy L. (1990), *Common Lisp: The Language*, second edition, Bedford, MA: Digital Press.

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