HIGH QUALITY GRAPHICS FROM XLISP-STAT

JAN DE LEEUW AND FREDERIC UDINA

ABSTRACT. `Xlisp-Stat` is a statistical environment with very good tools for interactive and graphical statistical development. But it lacks good publication-quality graphical output. We present here an add-on to LISP-STAT graphic objects which produces files representing the graphic information in a `gnuplot` file. From these files, using `gnuplot`, high quality output can be obtained.

CONTENTS

Date. February 17, 1995.

Written in part while the first author was visiting the National Institute of Statistical Sciences, Research Triangle Park, NC.

1. Introduction

`Xlisp-Stat` [1] is a statistical environment based on the Lisp language. It includes object-oriented modules that provide ready-made tools for programming graphical and mouse-based user interfaces. The URL's

- `ftp://ftp.stat.ucla.edu/pub/lisp/xlisp/xlisp-stat/code`
- `http://euler.bd.psu.edu/lispstat/lispstat.html`
- `http://libiya.upf.es`

give examples of `Xlisp-Stat` programs that exploit these graphical capabilities [1].

In the X11 and Macintosh environment the plots produced by `Xlisp-Stat` are displayed on the screen as bitmaps. On the Mac the plot can be copied to the clipboard, and from there into various applications. The plots are saved as PICT files. In X11 the plots can be saved as Postscript files. The Postscript file and the PICT file are just wrappers around the bitmaps, however, which means that printed plots from `Xlisp-Stat` do not really look good. This is a major disadvantage compared to, for instance, S-plus, which produces real Postscript code. In this note we discuss some `Xlisp-Stat` functions that make it possible to produce beautiful plots on various devices (X11 screens, Tektronix emulators, HP printers, Postscript printers). We need, in addition to `Xlisp-Stat`, the `gnuplot` program [2].

`Gnuplot` is a command-driven interactive plotting program that can plot functions given their algebraic expression or data stored in files. `Gnuplot` can add plot titles, axis or point labels, etc, and has several lines and point styles to represent data. A powerful feature of `gnuplot` is the great variety of drivers it has for redirecting the final graph to the desired device (monitor or printer, for example). Because `gnuplot` is available on the Macintosh, on MS-Windows, and on UNIX, using it does not really restrict the generality of our approach. In the same way `gnuplot` is used in MatLab-like systems such as `octave`, `fudgit`, and `rlab`.

Our approach consists in defining some new methods for the graph-proto object, which create a series of files that can be used to reproduce the graph window content when loaded into `gnuplot`. Once in `gnuplot`, it is easy to translate the graph to the various desired formats.

In the next section we present a complete example of use of our transformation from `Xlisp-Stat` to postscript. Section 3 clarifies what the different graphic formats involved in the process are, and what we mean by "high quality" output. In Section 4 we discuss how the graphic content of a `Xlisp-Stat` windows is stored and how it can be retrieved. In Section 5, a full description of our methods is given and, finally, in Section 6 we give some additional technical information.

---

[1] `Xlisp-Stat` can be obtained by anonymous ftp from `ftp://umnstat.stat.umn.edu/pub/xlispstat` or from the UCLA ftp server. Relative to the `xlispstat/code` directory, the source for `gnuplot.lsp` is in `homegrown/filters/gnuplot` and the source of `histogram.lsp` is in `homegrown/plotting`.

2. An example

Running the following commands in `Xlisp-Stat` produces a graph object stored in the `aplot` variable.

```
(setq xx (uniform-rand 10))
(setq yy (normal-rand 10))

(setq aplot (plot-points xx yy))
(send aplot :title "scatter")

(setq regr-coefs (send (regression-model xx yy) :coef-estimates))

(send aplot :abline regr-coefs)
```

In Figure 1 a copy of the resulting window can be seen. The figure has been made using the standard `Xlisp-Stat` facilities, so it is a bitmap image of a window, a part of the screen.

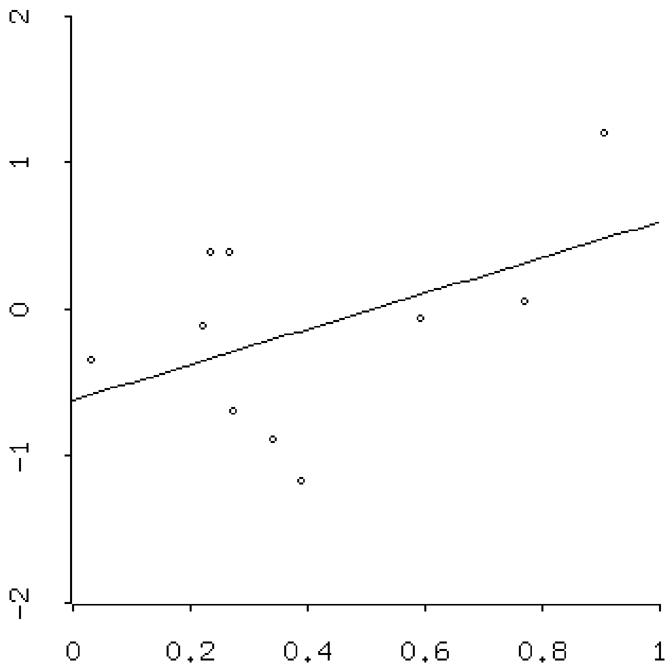

Figure 1. Plot saved from `Xlisp-Stat` as a bitmap

Running the following two `Xlisp-Stat` commands will load our programs and create the files for `gnuplot` reproducing the graphic content of the window.

```
(load "gnuplot")
(send aplot :to-gnuplot)
```

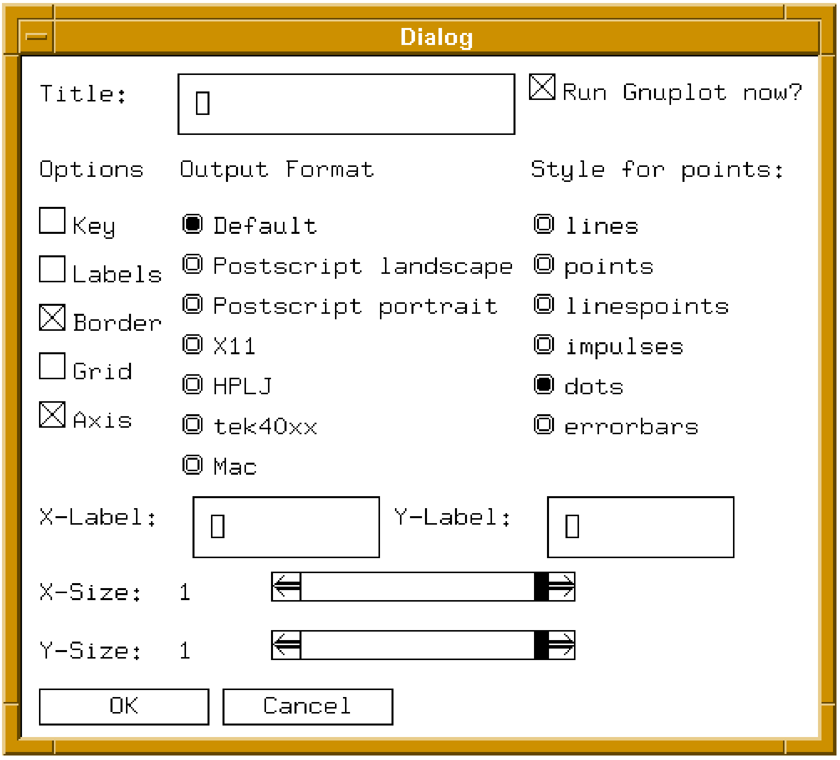

FIGURE 2. The Dialog for Driving `gnuplot` Conversion.

A dialog similar to Figure 2 will appear and, when the user clicks the OK button, the following files will be created (note that "scatter" is the name we have given to the object):

`scatter.gnu`: This is the main file to be loaded into `gnuplot`.
`scatter.pnt`: The coordinates of the 10 points contained in `aplot`
`scatter00.lin`: The coordinates of a series of points to describe the line.

Now, it is possible to start `gnuplot` and load the file. Doing this in Unix will look like:

```
$ gnuplot

        G N U P L O T
        unix version 3.5
        patchlevel 3.50.1.17, 27 Aug 93
```

```
        last modified Fri Aug 27 05:21:33 GMT 1993

        Copyright(C) 1986 - 1993   Colin Kelley, Thomas Williams

        Send comments and requests for help to info-gnuplot@dartmouth.edu
        Send bugs, suggestions and mods to bug-gnuplot@dartmouth.edu

Terminal type set to 'x11'
gnuplot> load scatter.gnu
```

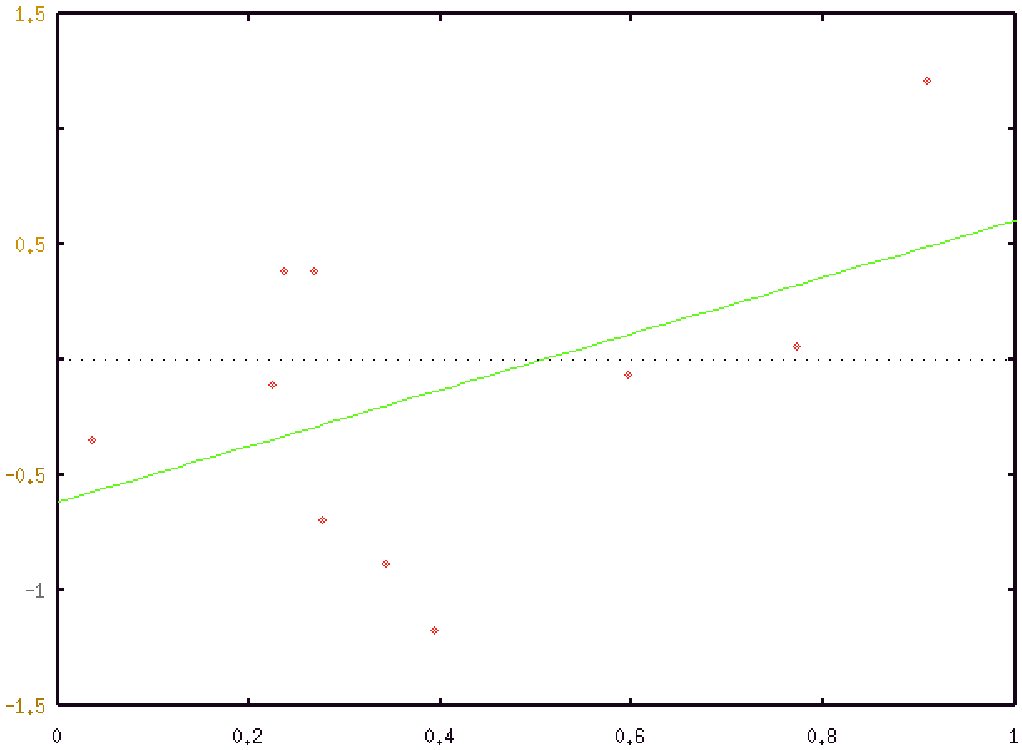

FIGURE 3. A gnuplot version of our plot, using the X11 driver

Figure 3 shows the window that gnuplot uses to display our plot. The figure is still a bitmap image of the window, so the quality improvement is small. In Figure 4, the Postscript output produced by gnuplot is shown, its “higher quality” is evident.

If Xlisp-Stat is running under UNIX and the box Run Gnuplot now? is checked, after creating the files gnuplot is automatically started to load the file scatter.gnu and it will produce the final file scatter.ps without further user action. For this it is not even necessary that X11 is present. It is probably possible to implement the same functionality in the Macintosh version, using Apple Events, and in the DOS/MSW version using DDE, but we have not experimented with this.

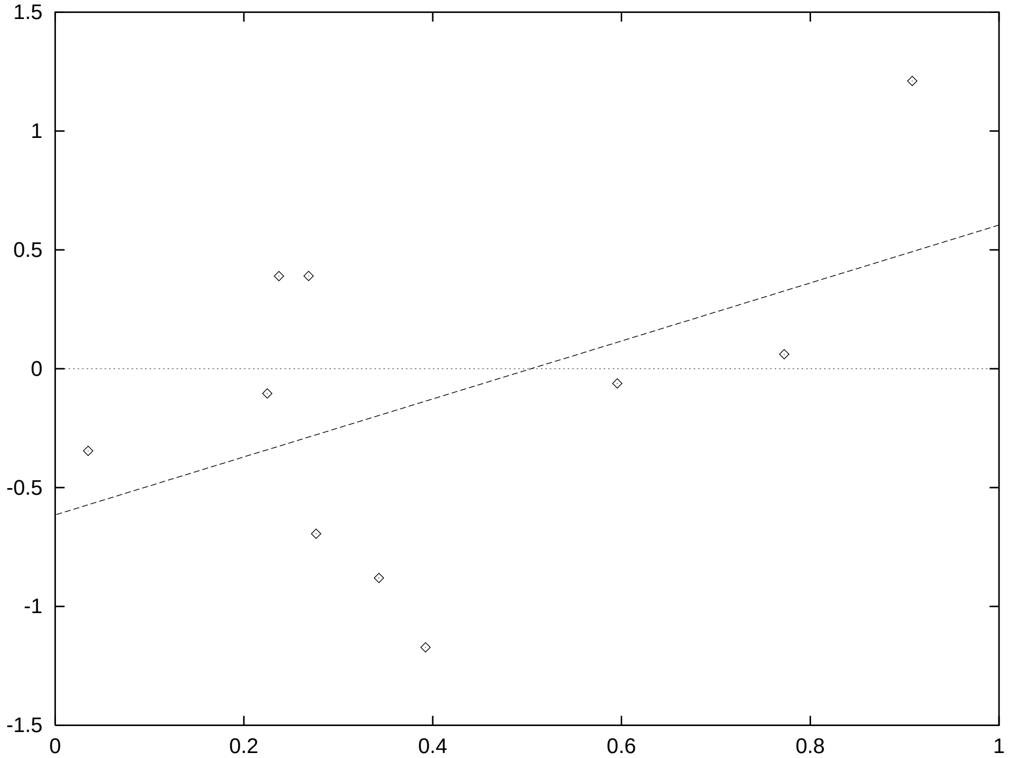

FIGURE 4. The final Postscript output obtained.

3. FORMATS OF GRAPHICAL OUTPUT

Let us explain a bit more in detail what the problem is. Postscript is a device-independent page description language. It codes a line by simply remembering the positions of the end points, and it then leaves it to the device driver to fill in the line. For a Postscript printer these means that the line is drawn with a resolution of 300 or 600 dots per inch. The same thing is true for text in the plot. Letters are defined by Postscript code, and the actual letter is scaled and constructed using the full resolution of the device.

For bitmaps the situation is much less favourable. An X11 bitmap, or a Macintosh screen image, has a resolution of 72 dpi. Bitmaps are very flexible, because we cannot only them to draw lines and splines and letters, but we can draw anything. A black and white screen of 1000 pixels in both directions has obviously 10^6 pixels, which means 2^{10^6} possible patterns. A lot of flexibility, but still only at 72 dpi. Thus if we scale a letter or a picture, we have to scale the bitmap, which involves complicated algorithms and unsatisfactory solutions. Also, although we can edit the bitmap in a program such as MacPaint or SuperPaint or XPaint, we cannot get past the 72 dpi boundary[2]. And if we paste a bitmap into a draw program such as MacDraw or idraw or xfig it just becomes one large object sitting in the drawing which cannot be edited further. Thus it would be nicer if `Xlisp-Stat` could make plots defined in terms of objects.

4. Graph windows in Xlisp-Stat

Most of the graphics shown by `Xlisp-Stat` are instances of the `graph-proto` prototype. These objects have methods to display points and lines. They have a slot named `:internals` where points and lines are stored in some (undocumented) internal format. Users and programmers store points into a graph object by means of the method `:add-point`. The method `:num-points` returns the number of points already stored in the object. The methods `:add-line` and `:num-lines` do the same with lines.

In our programs we extract the point and line coordinates using the methods already mentioned, and the `graph-proto` methods `:point-coordinate`, `:linestart-coordinate`, and `:linestart-next`. These are all standard and documented methods. This means that any data points and lines belonging to a graph object conforming to these specification will be translated by `:to-gnuplot` to `gnuplot`.

Graph objects have some other features not supported by `:to-gnuplot`. They can draw his points and data through a transformation involving more than two variables, or draw dashed lines or with a given color or even in diferent thickness. In some cases we don't translate these features to `gnuplot` because it doesn't support them, in other cases because our scope is limited in this first attempt.

`Xlisp-Stat` has also other graphic windows that are based on different kinds of objects. For example, histograms are displayed through descendants of the `histogram-proto` prototype. They handle data in a different way and `:to-gnuplot` doesn't support this. So it will not be possible to export histograms and some other kinds of `Xlisp-Stat` graphics to `gnuplot` in the same way. There is a simple way out of this quandry, however. The file `histogram.lsp` in Appendix B contains the code for a `my-histogram` prototype that inherits from the `scatterplot-proto`, and consequently can be used with `:to-gnuplot`. It is also able to make "hollow" histograms, and frequency polygons. The function `my-histogram` takes three arguments: the data, the binwidth, and the origin. Thus, in an `Xlisp-Stat` session in which gnuplot.lsp is already loaded, we can say

[2]This is not entirely true. Programs such as SuperPaint have an Autotrace option, which translates bitmaps into Bezier curves, and a Superbits option,, which makes it possible to edit bitmaps at 300 dpi. This is time consuming, and it requires considerable skill and patience.

```
(setq xx (normal-rand 100))
(setq hh (my-histogram 100 .2 0 :poly t))
(send hh :to-gnuplot))
```

This brings up the usual plots and dialogs, and ultimately produces the "before" and "after" pictures in Figures 5 and 6.

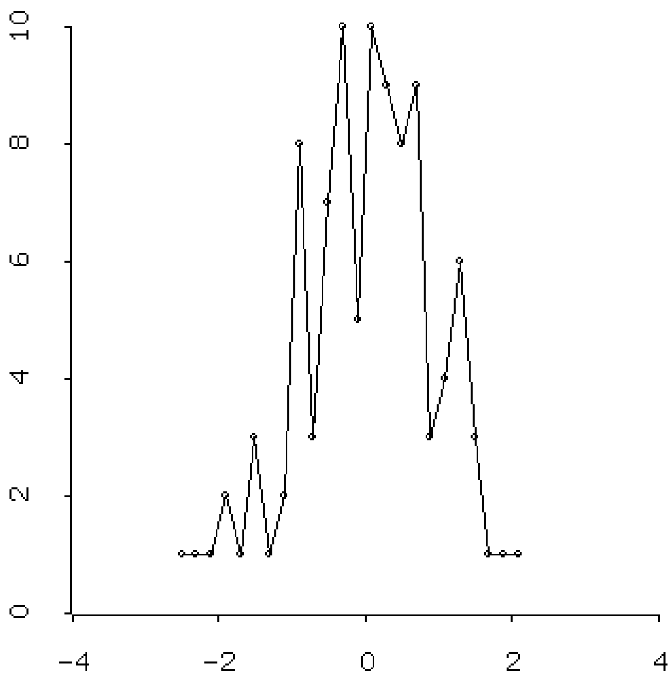

FIGURE 5. Frequency Polygon from `histogram.lsp`

5. GNUPLOT AND GNUPLOT DRIVERS

We will not give a full introduction here to `gnuplot`. For this we refer to [2], and to the on-line documentation included with `gnuplot` itself. We want to mention here just those aspects of `gnuplot` that you will need to use it from `Xlisp-Stat`.

`Gnuplot` has great flexibility in outputting graphics. The pieces of software used to direct the graphics to different devices are called *drivers*. There are about 60 drivers currently available in `gnuplot`, including Postscript format, Hewlett-Packard laser printers (PCL 5 page description language), X11 windows, VGA, Mac PICT, LaTeX, and so on. Before `gnuplot` can produce any output, a driver must be selected, and some output stream must be designated. For example, on a UNIX/X11 workstation the driver selected

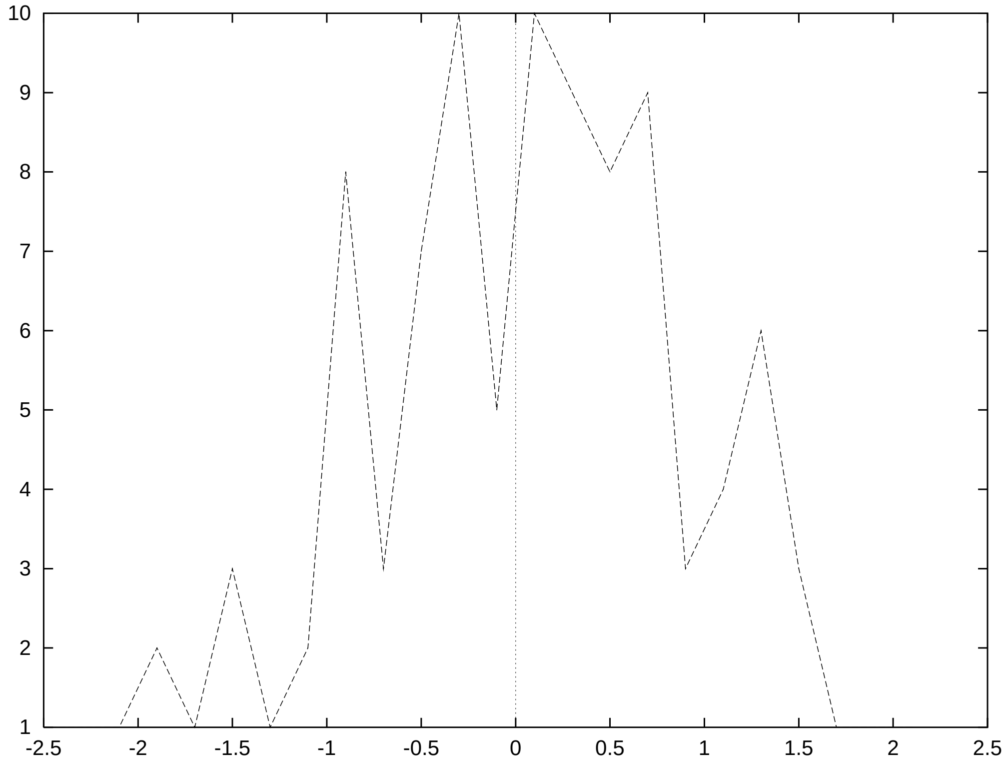

FIGURE 6. Frequency Polygon after Gnuplot

by default is `x11`, and the output stream is the `gnuplot` window. If the Postscript driver is selected, some file must be given as output stream for writing the representation of the graphic. This will be done automatically for you if you select the right format in the `gnuplot` dialog.

But `gnuplot` is not so flexible with regard to its input. The only way `gnuplot` can plot lines is by reading point coordinates from a file and then joining the resulting points with segments. In a file there can be several lines as different columns of data, but the x-coordinate of the points defining the lines must be the same. This puts severe constrains on the way we translate graphics from `Xlisp-Stat` to `gnuplot`. For each line that has a particular set of x-values we must define a separate file. If different lines have the same x-values, we put them all in the same file.

`Gnuplot` and `Xlisp-Stat` have different algorithms to compute default axis ranges or default tick mark positions. We leave it to `gnuplot` apply its own algorithms, so you probably will get different ranges and tick marks in the Postscript version of the graph. Compare Figures 5 and 6, for example. If you don't like them, you can edit the `gnuplot` file. The same can be said about line styles applied to different lines of your plot, and about other gadgets that gnuplot can draw for you.

6. Using the method :to-gnuplot

The simplest way to use the method :to-gnuplot is to send it to the graph object you want to export. See Section 2 for a complete example.

The graph-proto method :to-gnuplot has several optional key-type argument to add versatility to its use. The simplest way to use it has been shown in Section 2. The main key-arguments available are:

- :run-gnuplot: If followed by t (and the operating system in use supports it) gnuplot will be automatically started after graphics translation.
- :use-dialog: If followed by nil, the main dialog (see Figure 2) will not be used and values for gnuplot options will be taken by default.
- :ask: If followed by t, the user is asked about the filename to be used. If followed by nil, the filename is automatically generated from the window title.
- :temp-files-dir: Must be followed by a complete path name to be prepended to the filename that is generated from the name of the plot.
- :output-file: Must be followed by a filename. For output in files, in Postscript or HPLJ drivers, the filename extension will be appended automatically.

There are other key-arguments accepted by :to-gnuplot, mainly to decide the default values for gnuplot options. You might look at the gnuplot documentation to fully understand it.

Key	Default	Possible values	Description
:terminal	($\star$)	any of the 60 drivers	the driver to be used
:style	points	see gnuplot docs	the style for plotting points
:key	nil	t or nil	put a legend or not
:border	nil	t or nil	border the graph?
:data-labels	nil	t or nil	include point labels in graph?
:grid	nil	t or nil	draw a grid?
:axis	nil	t or nil	draw axis at 0?
:plot-title	""	any string	title for the gnuplot graph
:x-label	""	any string	label for x axis
:y-label	""	any string	label for y axis
:x-size	1	a 0 to 1 number	reduction to apply
:y-size	1	a 0 to 1 number	reduction to apply

($\star$) this depends on the system being used. As normally gnuplot sets the terminal when starts up, currently we leave the terminal unspecified as default.

Let's assume that `obj` is a graph-proto descendant, then the following will be legal calls to `:to-gnuplot`:

```
(send obj :to-gnuplot  :run-gnuplot t :use-dialog nil
  :terminal "postscript" :border nil :axis nil
          :output-file "myplot")

(send obj :to-gnuplot :terminal "" :style "dots"
  :temp-files-dir "/users/udina/xlisp/tmp/"
  :plot-title "This is my main graphic work")
```

The first example will create, with a single `Xlisp-Stat` command, the PostScript file containing the graphics from the `obj` object and will store it in the file `myplot.ps`. It will automatically run `gnuplot` and it will not bring-up the dialog, options will be taken from the default values.

The second one will put no `set term` and no `set output` lines in the gnu file it will create, because `:terminal` has been set to `""`. This is convenient if you want to run the `gnuplot` file in different systems.

When popping-up the dialog, or when the dialog is not in use, if some of these options are not given a value, the program will use the default values stored in the property list of the symbol `*gnuplot*`. After running the dialog, the values fixed by the user will be stored in the same place.

References

1. L. Tierney, *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*, Wiley, New York, NY, 1990.
2. T. Williams and C. Kelley, *GNUPLOT. An Interactive Plotting Program*, version 3.5 ed., 1993.

Appendix A. Code for gnuplot.lsp

```
(defmacro WHILE (cond &rest exprs)
  `(loop
   (unless ,cond (return))
    (progn ,@exprs)))

;;;to store and manage defaults we will use the property mechanism of lisp
; mainly to avoid use of so many global vars, and use of blind lists
;examples of use:
;(putprop '*gnuplot* "x11" 'terminal)
;(putprop '*gnuplot* nil 'key)
;(putprop '*gnuplot* t 'border)
;(get '*gnuplot* 'terminal)
;(symbol-plist '*gnuplot*) returns the whole list of pairs
;(putprop '*gnuplot* "vga" 'terminal)
;

;;;set *gnuplot* defaults
;this defaults will be used for setting the dialog
;and will be updated from the output of the dilaog

#+unix (putprop '*gnuplot* "x11" 'terminal)
#+macintosh (putprop '*gnuplot* "mac" 'terminal)
#+msdos (putprop '*gnuplot* "vga" 'terminal)

(putprop '*gnuplot* "" 'terminal)

#+unix (putprop '*gnuplot* t 'run-it?)
#+macintosh (putprop '*gnuplot* nil 'run-it?)
#+msdos (putprop '*gnuplot* nil 'run-it?)

(putprop '*gnuplot* "dots" 'style)
(putprop '*gnuplot* "" 'x-label)
(putprop '*gnuplot* "" 'y-label)
(putprop '*gnuplot* "" 'plot-title)
(putprop '*gnuplot* 1 'x-size)
(putprop '*gnuplot* 1 'y-size)
(putprop '*gnuplot* nil 'key)
(putprop '*gnuplot* nil 'data-labels)
(putprop '*gnuplot* t 'border)
(putprop '*gnuplot* nil 'grid)
(putprop '*gnuplot* t 'axis)

;;;this decides the default, 'auto or 'ask
;when 'auto, try to get a filename from the graph object title
;when 'ask, always ask the user about the filename
;(putprop '*gnuplot* 'auto 'filenaming)
```

```
(putprop '*gnuplot* 'ask 'filenaming)
#+msdos (putprop '*gnuplot* 'ask 'filenaming)

(defmeth graph-proto :filename (&key (num nil numset) (ask nil))
  "returns a default filename to use based on slot 'title
 but replacing non alfaanum chars with _
 Keys: If :NUM is given, appends it to the name, format 2 digits
       if :ASK is t, the user will be asked to confirm or modify
       the filename."
  (unless (send self :has-slot 'filename)
          (send self :add-slot 'filename nil))
  (if numset
      (let ((strnum (num-to-string num)))
        (if (< (length strnum) 2)
            (setf strnum (concatenate 'string (send self :filename)
                                     "0" strnum))
          (setf strnum (concatenate 'string (send self :filename)
                                   strnum))))
    (let* ((default (let* ((stringa (send self :title))
                           (newstr (coerce (repeat #\_ (length stringa)) 'string))
                           chari inti)
                      (dotimes (i (length stringa))
                               (setf inti (char-int (setf chari (elt stringa i))))
                               (when (or (char<= #\a chari #\z)
                                         (char<= #\A chari #\Z)
                                         (char<= #\0 chari #\9))
                                     (setf (elt newstr i) chari)))
                      newstr))
           #+macintosh (prmpt "Give a prefix for filenaming (<= 19 char)")
           #+unix (prmpt "Give a prefix for filenaming (<= 24 char)")
           #+msdos (prmpt "Give a prefix for filenaming (<= 6 char)")
           (current default))
      (if ask
          (progn
            (setf current (get-string-dialog  prmpt :initial default))
            (unless current (setf current default))
            (print (list "gorra" current)))
        (if (slot-value 'filename)
            (setq current (slot-value 'filename))
          (setq current (get-a-valid-gp-filename default))))
      (send self :slot-value 'filename current)
      current)))

(defmeth graph-proto :to-gnuplot (&key (use-dialog t)
                                       (run-gnuplot (get '*gnuplot* 'run-it?))
                                       (temp-files-dir "")
                                       (output-file "" fileset)
```

```
                                        (terminal (get '*gnuplot* 'terminal))
                                        (style    (get '*gnuplot* 'style))
                                        (x-label  (get '*gnuplot* 'x-label))
                                        (y-label  (get '*gnuplot* 'y-label))
                                        (plot-title (get '*gnuplot* 'plot-title))
                                        (x-size   (get '*gnuplot* 'x-size))
                                        (y-size   (get '*gnuplot* 'y-size))
                                        (key      (get '*gnuplot* 'key))
                                        (data-labels
                                                  (get '*gnuplot* 'data-labels))
                                        (border   (get '*gnuplot* 'border))
                                        (grid     (get '*gnuplot* 'grid))
                                        (axis     (get '*gnuplot* 'axis)))
"Args: (&key (use-dialog t) (run-gnuplot nil) ... )
constructs a gnuplot file representing self
from the info the object contains. On UNIX systems, it also runs
gnuplot if given the key option ':run-gnuplot t'.
Default is to display a dialog for choice of gnuplot options.
This can be overriden by ':use-dialog nil'"
(let* ((flprefix (case (get '*gnuplot* 'filenaming)
                       ('auto (send self :filename))
                       ('ask (send self :filename :ask t))
                       (nil (send self :filename :ask t))))
        (flnm (concatenate 'string temp-files-dir flprefix))
        (np (send self :num-points))
        x y
        (do-it t)
        (fnpoints (concatenate 'string flnm ".pnt"))
        (fngnu (concatenate 'string flnm ".gnu")))
   (when
    use-dialog
    (setq do-it (gnuplot-dialogue))
    (when do-it
         ;must reread all props from *gnuplot*
          (setq terminal (get '*gnuplot* 'terminal))
          (setq style    (get '*gnuplot* 'style))
          (setq x-label  (get '*gnuplot* 'x-label))
          (setq y-label  (get '*gnuplot* 'y-label))
          (setq plot-title (get '*gnuplot* 'plot-title))
          (setq x-size   (get '*gnuplot* 'x-size))
          (setq y-size   (get '*gnuplot* 'y-size))
          (setq key      (get '*gnuplot* 'key))
          (setq run-gnuplot      (get '*gnuplot* 'run-it?))
          (setq data-labels (get '*gnuplot* 'data-labels))
          (setq border   (get '*gnuplot* 'border))
          (setq grid     (get '*gnuplot* 'grid))
          (setq axis     (get '*gnuplot* 'axis))))
   (when
```

```
do-it
(with-open-file
 (f fngnu :direction :output)
 (format *standard-output* "~%Writing gnuplot file ~s" fngnu)
 (unless (string= terminal "")
         (format f "~%set term ~a" terminal))
 (if (or (string= terminal "postscript")
         (string= terminal "post portrait"))
     (format f "~%set output ~s"
             (concatenate 'string temp-files-dir
                          (if fileset output-file flnm) ".ps"))
   (if (string= terminal "hpljii")
       (format f "~%set output ~a"
               (concatenate 'string temp-files-dir
                            (if fileset output-file flnm) ".hpl"))
     (if (not (string= terminal ""))
         (format f "~%set output"))))
 (format f "~%set ~agrid" (if grid "" "no"))
 (format f "~%set ~akey" (if key "" "no"))
 (format f "~%set ~aborder" (if border "" "no"))
 (format f "~%set ~azeroaxis" (if axis "" "no"))
 (format f "~%set title ~s" plot-title)
 (format f "~%set xlabel ~s" x-label)
 (format f "~%set ylabel ~s" y-label)
 (format f "~%set data style ~a" style)
 (format f "~%set size ~f, ~f" x-size y-size)
 ;;now write (or erase) labels
 (format f "~%set nolabel")
 (if (and data-labels
          (> np 0))
     (let* ((xs (send self :point-coordinate 0 (iseq np)))
            (ys (send self :point-coordinate 1 (iseq np)))
            (labs (send self :point-label (iseq np))))
       (dotimes (i np)
                (format f "~%set label ~s at ~f, ~f"
                        (elt labs i) (elt xs i) (elt ys i)))))
 (unless (= 0 (+ np
                 (send self :num-lines)))
         (if (> (send self :num-lines) 0)
             (setf stringa (send self :write-lines-to-gnu temp-files-dir))
           (setf stringa ""))
         (format f "~%plot ")
         (if (> np 0)
             (progn
               (setf x (send self :point-coordinate 0 (iseq np)))
               (setf y (send self :point-coordinate 1 (iseq np)))
               (send self :write-points-to-gnu fnpoints)
               (format f " ~s" fnpoints))
```

```
                   (when (> (length stringa) 1)
                         (setf stringa (subseq stringa 1))))
                 (when (> (send self :num-lines) 0)
                       (format f stringa)))
         (if (string= terminal "postscript")
             (format f "~%quit")
           (if (string= terminal "X11")
               (format f "~%pause -1")
             (if (string= terminal "x11")
                 (format f "~%pause -1")
               (if (string= terminal "hpljii")
                   (format f "~%quit")
                 (if (string= terminal "tek40xx")
                     (format f "~%pause -1")
                   (if (string= terminal "mac")
                       (format f "~%")))))))
         (format f "~%")
         (format *standard-output* "~%")
         )
        (when run-gnuplot
              ;;(and (featurep 'UNIX)
              (format *standard-output*
                      "~%now running gnuplot on file ~s~%" fngnu)
              (system (concatenate 'string "gnuplot " fngnu "&"))))))

(defmeth graph-proto :write-points-to-gnu (&optional (filnm nil))
"this is method is used by :to-gnuplot method, it is not intended
 to be used by itself"
  (let* ((fn (if filnm filnm
                 (concatenate 'string (send self :filename) ".pnt")))
         (n (send self :num-points))
         (x (send self :point-coordinate 0 (iseq n)))
         (y (send self :point-coordinate 1 (iseq n))))
     (with-open-file
      (g  fn  :direction :output)
      (format *standard-output* "~%writing points to file ~s" fn)
      (dotimes (i n)
               (format g "~f~a~f~%" (elt x i) #\Space (elt y i))))))

(defmeth graph-proto :get-one-line (start)
"this method is used by :to-gnuplot method, it is not intended
 to be used by itself"
  (let* ((next start)
         (xcoords (list (send self :linestart-coordinate 0 start)))
         (ycoords (list (send self :linestart-coordinate 1 start))))
     (WHILE (setq next (send self :linestart-next next))
       (setq xcoords (cons (send self :linestart-coordinate 0 next) xcoords))
       (setq ycoords (cons (send self :linestart-coordinate 1 next) ycoords)))
```

```
    (list xcoords ycoords)))

(defmeth graph-proto :get-lines ()
"this method is used by :to-gnuplot method, it is not intended
 to be used by itself. Returns the lines of self in a list of lists format."
  (let ((first 0)
        (last (send self :num-lines))
        aline
        (resul nil))
    (WHILE (< first last)
           (setf aline (send self :get-one-line first))
           (setq resul (cons  aline resul))
           (setq first (+  first (length (first aline)))))
    resul))


(defmeth graph-proto :write-lines-to-gnu (&optional (directory ""))
"this method is used by :to-gnuplot method, it is not intended
 to be used by itself.
 creates files filenameNN.lin for gnuplot ploting the lines of self
 Returns a string for gnuplot to load the files created"
(let* ((lines (arrange-lines-for-gnu (send self :get-lines)))
       (num-file 0)
       filen
       (to-load "")
       num-cols)
  (if (> (length lines) 99)
      (message-dialog
       "This object has too much lines! ~%(more than 99) Trash it.")
    (dotimes
     (i (length lines))
     (setf filen (concatenate 'string
                              directory
                             (send self :filename :num num-file) ".lin"))
     (write-lines-one-gnu-file (nth i lines) filen)
     (setf num-cols (length (first (nth i lines))))
     (WHILE (> num-cols 1)
            (setf to-load
                  (concatenate 'string
                              (format nil ", ~s using 1:~a with lines"
                                      filen num-cols)
                               to-load))
            (setf num-cols (1- num-cols)))
     (setf num-file (1+ num-file))))
  to-load))

(defun gnuplot-dialogue ()
"Creates and display a dialog for the user choosing the desired
```

```
 gnuplot options to be passed to method :to-gnuplot.
 Resulting options are stored as props of the symbol *gnuplot*"
(let* ((cancel (send modal-button-proto :new "Cancel"))
       (title (send edit-text-item-proto :new
                    (get '*gnuplot* 'plot-title) :text-length 20))
       (key (send toggle-item-proto :new "Key"
                  :value (get '*gnuplot* 'key)))
       (run-it (send toggle-item-proto :new "Run Gnuplot now?"
                  :value (get '*gnuplot* 'run-it?)))
       (labels (send toggle-item-proto :new "Labels"
                     :value (get '*gnuplot* 'data-labels)))
       (border (send toggle-item-proto :new "Border"
                     :value (get '*gnuplot* 'border)))
       (grid (send toggle-item-proto :new "Grid"
                   :value (get '*gnuplot* 'grid)))
       (axis (send toggle-item-proto :new "Axis"
                   :value (get '*gnuplot* 'axis)))
;aixo no va be aixi
       (str-format (list "" "postscript" "post portrait"
                         "x11" "hpjii" "tek4010" "mac"))
       (lab-format (list "Default" "Postscript landscape" "Postscript portrait"
                         "X11" "HPLJ" "tek40xx" "Mac"))
       (format (send choice-item-proto :new lab-format
                     :value (position (get '*gnuplot* 'terminal)
                                      str-format :test #'string=)))
       (str-styles (list "lines" "points" "linespoints" "impulses"
                         "dots" "errorbars"))
       (style (send choice-item-proto :new str-styles
                    :value  (position (get '*gnuplot* 'style)
                                      str-styles :test #'string=)))
       (x-label (send edit-text-item-proto :new
                      (get '*gnuplot* 'x-label) :text-length 10))
       (y-label (send edit-text-item-proto :new
                      (get '*gnuplot* 'y-label) :text-length 10))
       (x-size (send text-item-proto :new "0.0" :text-length 4))
       (y-size (send text-item-proto :new "0.0" :text-length 4))
       (x-size-scroll (send interval-scroll-item-proto
                            :new (list 0 1) :text-item x-size))
       (y-size-scroll (send interval-scroll-item-proto
                            :new (list 0 1) :text-item y-size))
       (ok (send modal-button-proto :new "OK"
                 :action
                 #'(lambda ()
                     (putprop '*gnuplot* (send key :value) 'key)
                     (putprop '*gnuplot* (send run-it :value) 'run-it?)
                     (putprop '*gnuplot* (send labels :value) 'data-labels)
                     (putprop '*gnuplot* (send border :value) 'border)
                     (putprop '*gnuplot* (send grid :value) 'grid)
```

```
                     (putprop '*gnuplot* (send axis :value) 'axis)
                     (putprop '*gnuplot* (nth (send format :value)
                                              str-format)
                              'terminal)
                     (putprop '*gnuplot* (nth (send style :value)
                                              str-styles)
                              'style)
                     (putprop '*gnuplot* (send x-size-scroll :value) 'x-size)
                     (putprop '*gnuplot* (send y-size-scroll :value) 'y-size)
                     (putprop '*gnuplot* (send x-label :text) 'x-label)
                     (putprop '*gnuplot* (send y-label :text) 'y-label)
                     (putprop '*gnuplot* (send title :text) 'plot-title)
                     t)))
       (plot-dialog (send modal-dialog-proto :new
                          (list (list  (send text-item-proto :new "Title: ")
                                       title
                                       run-it)
                                (list
                                 (list (send text-item-proto :new "Options")
                                       key labels border grid axis)
                                 (list  (send text-item-proto :new "Output Format")
                                        format)
                                 (list  (send text-item-proto :new
                                              "Style for points:")
                                        style))
                                (list (send text-item-proto :new "X-Label:")
                                      x-label
                                (send text-item-proto :new "Y-Label:")
                                      y-label)
                                (list  (send text-item-proto :new "X-Size:")
                                       x-size x-size-scroll)
                                (list (send text-item-proto :new "Y-Size:")
                                      y-size y-size-scroll)
                                (list ok cancel)))))
  (send y-size-scroll :value (get '*gnuplot* 'x-size))
  (send x-size-scroll :value (get '*gnuplot* 'y-size))
  (send plot-dialog :modal-dialog)))

(defun lines-with-same-x-p (lin1 lin2)
"Args: line1 line2
 A line is a list of x's and y's, this function returns T if the x's are equal"
  (and (= (length (first lin1)) (length (first lin2)))
       (notany #'null (map 'list #'= (first lin1) (first lin2)))))

(defun arrange-lines-for-gnu (lines)
"given LINES, a list of lines, each one a list of x's and y's,
 returns a list of multi-lines, each one a list of x y1 y2 .. yn (n>0)"
(let* ((rest lines)
```

```
       output
       first current good bad)
  (WHILE (> (length rest) 0)
         (setf first (first rest))
         (setf current (cdr rest))
         (setf good first)
         (setf bad nil)
         (WHILE current ;scan all lines searching for lines that match first
                (if (lines-with-same-x-p first (first current))
                    (setf good
                          (reverse (cons (second (first current))
                                         (reverse good))))
                  (setf bad (cons (first current) bad)))
                (setf current (cdr current)))
         (setf output (cons (transpose good) output))
         (setf rest bad))
  output))

(defun write-lines-one-gnu-file (line filename)
"LINE is a list of list, each one is x y1 .. yn (n>0)
 FILENAME must be given also."
(with-open-file
 (f filename :direction :output)
 (format *STANDARD-OUTPUT* "~%writing lines to file ~s" filename)
 (dolist (item line)
         (format f "~f " (first item))
         (dotimes (i (1- (length item)))
                  (format f "~f " (nth (1+ i) item)))
         (format f "~%"))))


;;we deal here with problems with filename

(defun featurep (sym)
  (member sym *features*))


(defun valid-gp-filename (stringa)
  (when stringa
        (if (featurep 'macintosh)
            (< (length stringa) 20)
          (if (featurep 'msdos)
              (< (length stringa) 7)
            (if (featurep 'unix)
                (< (length stringa) 25)
              nil)))))

(defun get-a-valid-gp-filename (str)
```

```
"gives the user a chance to change the filename"
(if (valid-gp-filename str)
    str
  (let* (
         #+macintosh (prmpt "Give a prefix for filenaming (max 19 char)")
         #+unix (prmpt "Give a prefix for filenaming (max 24 char)")
         #+msdos (prmpt "Give a prefix for filenaming (max 6 char)")
         (stru (get-string-dialog  prmpt :initial str)))
    stru)))

;;;;;;;;;;for testing
(defun testit ()
  (setq plotgnu (plot-points (uniform-rand 5) (uniform-rand 5)))
  (send plotgnu :add-lines (list (uniform-rand 5) (uniform-rand 5)))
  (send plotgnu :add-lines (list (uniform-rand 5) (uniform-rand 5)))
  (send plotgnu :add-lines (list (uniform-rand 5) (uniform-rand 5)))
  (send (send plotgnu :menu)
        :append-items
        (send menu-item-proto :new "to-gnuplot"
                    :action (lambda () (send plotgnu :to-gnuplot))))
  (format t "~%stored in 'plotgnu'~%"))


;;;;;;;;;;;;;provide
;;;if problems with release 3 Comment out this line:
(provide "gnuplot")
```

Appendix B. Code for histogram.lsp

```
(defproto my-hist-proto () () scatterplot-proto)

(defun my-histogram
  (data h x0 &key (plot t) (hollow nil) (prob nil) (poly nil)
                          (title "Histogram"))
 (setf pp (if plot (send my-hist-proto :new 2 :show nil :title title)))
 (send pp :add-slot 'data data)
 (send pp :add-slot 'h h)
 (send pp :add-slot 'x0 x0)
 (send pp :add-slot 'plot plot)
 (send pp :add-slot 'hollow hollow)
 (send pp :add-slot 'prob prob)
 (send pp :add-slot 'title title)
 (send pp :add-slot 'poly poly)
 (let* (
        (slider (send menu-item-proto :new "Change-Bins"
                   :action #'(lambda () (send pp :create-slider))))
        (poly (send menu-item-proto :new "Polygon Frequencyiogram"
                   :action #'(lambda ()
                             (send pp :poly (not (send pp :poly)))
                             (send pp :main))))
        (hollow (send menu-item-proto :new "Hollow/Full Histogram"
                   :action #'(lambda ()
                             (send pp :hollow (not (send pp :hollow)))
                             (send pp :main))))
        (ppmenu (send menu-proto :new "Histogram"))
       )
   (send ppmenu :append-items slider poly hollow)
   (send pp :menu ppmenu)
   (send pp :main)
pp))


(defmeth my-hist-proto :create-slider ()
 (let (
       (ldata (length (remove-duplicates (send self :data))))
      )
  (interval-slider-dialog (list 0 ldata)
                          :points (* 10 ldata)
                          :action #'(lambda (x)
                            (send self :clear)
                            (send self :h x)
                            (send self :main)))
 )
)
```

```
(defmeth my-hist-proto :main ()
(let* (
       (data (send self :data))
       (h (send self :h))
       (x0 (send self :x0))
       (plot (send self :plot))
       (hollow (send self :hollow))
       (prob (send self :prob))
       (poly (send self :poly))
       (title (send self :title))
       (nn (length data))
       (ft (floor (/ (- (min data) x0) h)))
       (lt (ceiling (/ (- (max data) x0) h)))
       (bm (+ x0 (* h (iseq ft lt))))
       (nb (length bm))
       (n0 (repeat 0 nn))
       (n1 (repeat 1 nn))
       (cn (count-in-bins data h x0))
       (nh (* nn h))
       )
(cond (plot (cond (hollow (send self :draw-mode 'xor))
                  (t (send self :draw-mode 'normal)))
            (send self :clear)
            (dotimes (i (1- nb))
                   (let* (
                         (a (elt bm i))
                         (b (elt bm (1+ i)))
                         (c (elt cn i))
                         (d (if prob (/ c nh) c))
                         )

             (cond (poly (send self :draw-mode 'normal)
                     (send self :add-points (list (/ (+ a b) 2)) (list d)))
                   (t (send self :add-lines (list a a b b) (list 0 d d 0))))))
          (if poly (send self :add-lines
                             (send self :point-coordinate 0
                                    (send self :points-showing))
                             (send self :point-coordinate 1
                                    (send self :points-showing))))
          (send pp :adjust-to-data)
          (send pp :show-window t))
         (t (let (
                (midp (+ x0 (* h (rseq (- ft .5) (+ lt .5) (+ 2 (- lt ft))))))
                (est (/ cn (* nn h)))
                )
            (list midp (concatenate 'list (list 0) est (list 0))))))
))
```

```
(defun count-in-bins (data h x0)
(let* (
       (n (length data))
       (indx (floor (/ (- data x0) h)))
       (jndx (- indx (min indx)))
       (nbin (1+ (max jndx)))
       (bcnt (repeat 0 nbin))
      )
(mapcar #'(lambda (x) (setf (elt bcnt x) (1+ (elt bcnt x)))) jndx)
bcnt
))

(defmacro assessor (key slot prototype)
`(defmeth ,prototype ,key (&optional (content nil set))
    (if set (setf (slot-value ',slot) content))
    (slot-value ',slot)))

(assessor :data data my-hist-proto)
(assessor :h h my-hist-proto)
(assessor :x0 x0 my-hist-proto)
(assessor :plot plot my-hist-proto)
(assessor :poly poly my-hist-proto)
(assessor :hollow hollow my-hist-proto)
(assessor :title title my-hist-proto)
```

Interdivisional Program in Statistics, University Of California, Los Angeles, CA 90095
E-mail address: deleeuw@stat.ucla.edu

Departament d'Economía, Universitat Pompeu Fabra, Barcelona, Spain
E-mail address: udina@upf.es