GPLOT: A **DIMFILM** Based Graph Plotting and Drawing Program for CDC NOS 2.8

Nick Glazzard

Version 0.87
October 16, 2025

Contents

1 Introduction

GPLOT is an interactive graphics program based on U.L.C.C. **DIMFILM**, a substantial graphics library written and maintained by Dr. John Gilbert between 1972 and approximately 1995. The version of **DIMFILM** used is from somewhere around 1984 and is the second major version, which moved away from CDC Fortran to portable, standard Fortran-77 (necessitated by U.L.C.C.'s move to Amdahl IBM compatible machines).

GPLOT, in contrast, was deliberately written for the CDC `FTN5` compiler under NOS 2.8 and was not intended to be portable. It uses seven letter identifiers, dynamic memory allocation (via the Common Memory Manager), and makes some use of NOS system calls.

That said, it can be built and used on "Unix-like" systems (Debian Linux and macOS have been tested) where it behaves almost identically to the NOS version.

Originally, **GPLOT** was focussed on plotting graphs from tabulated data and could perhaps be thought of as a stripped down **Gnuplot**, though with far fewer capabilities. Recent versions have extended its capabilities in new directions so that it can be used for drawing block diagrams, tables and other technical drawings.

The primary 'interactive' output device for **GPLOT** is **GTerm** – a simple terminal emulator with colour graphics capabilities written in Python (see the **GTerm** project repository for more information). **GPLOT** also supports Tektronix 401x terminals (which `xterm` emulates, although René Richarz's excellent 4014 emulator is preferred).

Output in the form of graphics files which can be included in other documents is available in two formats: Encapsulated PostScript files and SVG files. The latter is used by the **PMDHTML** Markdown to HTML program and the **PLNOTES** short document maintenance scheme for NOS (which is not described further here) as well as for HTML documents in general. The former is useful for LaTeXand other "print" style document systems.

A major goal for **GPLOT** is that it should be *very easy to use* — as easy as other graph plotting programs on modern operating systems (such an **Gnuplot**, matplotlib, etc.) — although **GPLOT** is not directly comparable to those programs. In some aspects, it does much less than they do, but it also does things they do not do.

To build and install **GPLOT** on an emulated CYBER mainframe (with DtCyber), or on a "Unix-like" system, please refer to the main **GPLOT** Github project `README.md`. This describes the build procedures in a step-by-step fashion and also explains how a single source code base can be used fairly easily on both NOS and "Unix-like" systems.

GPLOT (with **DIMFILM**) is almost entirely self-contained – it has no dependencies beyond the operating system and language runtime libraries. By modern standards, it is very light weight – the entire source of GPLOT, DIMFILM, output device code and utility functions is less than 40,000 lines, including comments. It should be possible to port it to other systems with working Fortran-77 compilers and runtimes, provided they do not insist on six character variable and function names (although it would not be very hard to work around that with a simple Python program to transform

the source code). Many historic systems, as well as almost all modern systems, should be able to build and run it. That said, it will almost certainly not run on systems as small as PDP-11s.

1.1 About this document

This description of GPLOT is in three parts.

- **Part I** is a “traditional manual” for **GPLOT** that describes each of its commands in detail, but doesn’t help you decide which of them to use for a specific task.
- **Part II** is a tutorial (of sorts) that goes through various tasks that **GPLOT** can help with from plotting simple graphs to creating quite complicated diagrams.
- **Part III** is a collection of information of various kinds on various topics that fill in details (e.g. complete font tables) or describes tools that can be used with **GPLOT**.

Part I
GPLOT Manual

2 Running GPLOT

GPLOT needs two files: GPLOT (the pre-loaded binary executable) and DADIMFO (which contains the font data for **DIMFILM**). To run **GPLOT** on NOS, use:

```
/ATTACH,GPLOT.
/GPLOT.
```

GPLOT will ATTACH the font file (DADIMFO) itself. On NOS, options supplied on the "command line" should be comma separated, following the usual rules for CCL (Cyber Control Language). Note that arguments containing spaces need to be "quoted" by enclosing them in dollar signs (as per normal CCL practice).

To run **GPLOT** on a "Unix-like" system, use:

```
$ ugplot
```

On these systems, options should be space separated and arguments containing spaces should be enclosed in double quotation marks – the usual rules for the shell being used should be followed. Note, though, that the keyword=value format is used, as it is on NOS. ugplot must be used here – this is a shell script that establishes symbolic links as necessary to arrange that the font file is accessible, then runs **GPLOT**.

The following options are recognized.

- OBEY=FILE
  GPLOT will read commands from the specified LOCAL file with name FILE. The program exits when the file has been read.

- PARM=STRING
  Used with OBEY=FILE. STRING is passed as one or more parameters to the OBEY file. See 3.6 for more details. Under NOS, if STRING needs to contain spaces (i.e. there is more than one parameter), it should be enclosed in $ (or ") signs. If any single parameter in STRING itself needs to contain spaces, that parameter should be enclosed in single quotes.

- GET=YN
  Turn on automatic GETs of indirect PERMANENT files before READ and OBEY try to access the files they need. YN is YES or NO (or an abbreviation down to Y or N). This saves having to issue GET in **GPLOT** (or NOS before **GPLOT** is run) in order to make the files to be used LOCAL. On the other hand, if changes have been made to a LOCAL version of a PERMANENT file, these will be lost when **GPLOT** READs or OBEYs that file with 'auto GET' in effect. So be careful! This option is not relevant on "Unix-like" systems.

- SAVE=YN
  Turn on automatic SAVE or REPLACE of graphics output files as indirect PERMANENT files. This option is not relevant on "Unix-like" systems.

- DEBUG=YN
  Outputs debugging information (mostly to do with OBEY file arguments and parameter substitutions currently).

- QUIET=YN
  Displays the unabbreviated command and OBEY nesting level for every command executed. Note that this may upset some output devices (especially the Tektronix 401x).

- SLIDE=YN
  Tell **DIMFILM** to use a graph plotting layout suitable for slides. This is the default, as it seems better suited to most output devices. The alternative tends to have text and other features that look too small on the page or display. Some people might like that, though.

An example of a **GPLOT** control statement is:

```
GPLOT,OBEY=DOPLOT,PARM=$X Y 'A TITLE'$.
```

On a "Unix-like" system, this would be:

```
ugplot obey=doplot parm="x y 'a title'"
```

3 Commands

All commands can be abbreviated so long as they uniquely identify a command. (If the abbreviation is ambiguous, you will get a warning message and no command will be executed).

All commands are shown in UPPER CASE, as that is the primary (NORMAL) character mode for NOS. In NORMAL mode, lower case is equivalent to upper case, so lower case can be used if desired.

Warning

GPLOT should *not* be used in ASCII mode on NOS (in which lower case characters are *really* lower case), otherwise command names will not be recognised, amongst other problems. This does not mean that lower case text cannot be plotted, though – it can.

On "Unix-like" systems, lower case input can be used freely. Command names are converted to upper case internally. Strings preserve the case as entered. Also, EPS and SVG output files are given their usual extensions (extensions are alien to NOS and COS).

In order to simplify usage on "Unix-like" systems, **GPLOT** converts all file names to lower case on those systems. This may (rarely) be undesirable, but is thought to be helpful in general.

If a line starts with a hash character (#) or the two characters "C " (upper case C only), the entire line will be treated as a comment. Anything after a semi-colon (;) is treated as an inline comment, unless the semi-colon is in a double quoted string.

Commands are divided in to related groups below, but that is only for convenience in describing them.

3.1 Basic Graphics Commands

These are the lowest level facilities on which graph plotting is built.

- DEVICE NAME [OUTPUT-FILE [SIZES]] — Select the output device.
  The EPS and SVG devices need an output file name to write to. Note that this will be created as a LOCAL file and will need to be made permanent with SAVE or REPLACE on NOS unless the SAVE=YES option has been used. For the EPS and SVG devices, an additional SIZES specification may be given. See Section 7 for the available devices and their characteristics.

- CLEAR [NAME] — Clear the drawing area.
  The device display will be set to its default background colour (white, except on the Tektronix 401x, which is black). Any previously drawn material will be erased. If EPSCOL or SVG devices are being used, the NAME argument can be used to set the name to be used by the *next* output file(s). Note that the name given must be 4 or fewer characters on NOS, or 72 characters on "Unix-like" systems. See 7 for more information. Using CLEAR with a file name before outputting anything is a good way to create a specifically named output file. The file named in the DEVICE command will simply be deleted if it is found to be empty when the CLEAR command is encountered.

- BOUNDS XL XH YL YH — Set the plotting bounds.
  This establishes a user coordinate system with x_l, y_l at the bottom left and x_h, y_h at the top right. The MOVE, DRAW, PANE and BLANK commands use this coordinate system.

- PANE XL XH YL YH — Set the PANE (clipping area).
  All drawing with MOVE, DRAW and TEXT will be clipped to the area specified here. In contrast, all graph plotting commands will apply inside this area, so that graphs will be scaled to fit this region. This is a way to plot multiple graphs side by side.

- UNPANE — Stop using any pane.
  The full drawing area will be used after this.

- CANVAS XL XH YL YH — A convenience function that sets BOUNDS and PANE (to the same extents) in one command.

- OUTLINE object — Outline an area.
  The area is determined by object, which may be: PANE, BLANK, BOUNDS or DEVICE.

- BLANK XL XH YL YH — Set the blank area.
  No drawing with either the basic graphics or graph plotting routines will make a mark inside the specified area. This can be useful for reserving a space for a graph key, for example.

- UNBLANK — Stop using any blank area.
  The previous blank area is no longer protected after this.

- COLOUR R G B — Set the RGB colour to use for drawing.
  DIMFILM maintains a separate drawing colour and line width for three different classes of graphical objects: lines, including those on graph plots; text (character strings); and graph plot annotations. This COLOUR command will be applied to the colour/style group last selected with a CSGROUP command (all 3 groups being the default if CSGROUP hasn't been used).

- WIDTH WIDTH — Set the line width.
  The width is a multiple of a device dependent "base width" and not all devices can draw wide lines (currently, only the Tektronix device cannot). This WIDTH will apply to the currently selected colour/style group as explained for COLOUR.

- CSGROUP GROUPNAME — Set the current colour/style group.
  This chooses which class of graphical objects any subsequent COLOUR and WIDTH commands apply to. The GROUPNAME values are:

  - ALL – Apply COLOUR and WIDTH to all three colour/style groups.
  - GENERAL – Apply them to line drawing operations.
  - TEXT – Apply them to text drawing operations.
  - ANNOT – Apply them to graph plot annotation operations.

  As usual, these group names can be abbreviated, providing the abbreviation remains unique.

- STYLE STYLENAME — Set the line style.
  The following styles are defined: SOLID, DASH, DOT, DASHDOT. These are probably self explanatory! Note that this is not constrained to the current colour/style group.

- MOVE X Y — Move to position.
  The virtual pen is moved to (x, y) in BOUNDS coordinates.

- DRAW X Y — Draw to position.
  The virtual pen is lowered and moved to (x, y) in BOUNDS coordinates.

- TEXT "TEXT" — Draw text.
  The string TEXT is drawn at the current pen position. The string may be enclosed in double quotation marks to allow spaces in the string. The text will be drawn with the current font, which defaults to SANS.SIMPLEX. The string may contain many special formatting sequences. The most frequently useful are *L to switch to lower case and *U to switch back to upper. The full set of format controls are described in the **DIMFILM** manual, and a summary of most of them is given in Table 1. On systems other than NOS, lower case letters will be preserved.

- FILL — Fill the drawing area with the current colour.
  This erases all drawn graphics, of course. It differs from CLEAR in that CLEAR always fills with a fixed device dependent colour and empties any buffered graphics commands in the device (if any). FILL is only useful after a CLEAR and only on **GTerm**.

3.2 Further Drawing Commands

These add a small number of primitive shapes as well as more control over how text is drawn. There is also a command that draws an arbitrary polyline.

- CIRCLE X Y R – Draw a circle.
  Centre at (x, y) with radius r.

Sequence	Purpose
*U	Start upper case. This is the default state and $U is not useful.
*L ...$L	Start and end lower case.
*B	Backspace over last character.
*1 or *2 or *3	Select a font set.
*+ ...$+	Start and end super-script. Maximum of 2 levels.
*- ...$-	Start and end sub-script. Maximum of 2 levels.
*= ...$=	Start and end underlining.
*N	Reset sub- or super- script level to zero.
*O ...$O	Make sub- or super- scripts fully below or above previous character (for summations).
*, numerator $, denominator $.	Construct a fraction.
*:nn	Output current symbol font character with code number nn (00 to 96).
*::nn	Output current marker font character with code number nn (00 to 96).
*Vnn	Output current font set alphabetic character with code number nn (00 to 96).

Table 1: Special text formatting character sequences

- ARC X Y R A1 A2 – Draw an arc of a circle.
  Centre at (x, y) and with radius r, starting at angle a_1 and ending at angle a_2. The angles are measured in degrees counter-clockwise from the positive X axis.

- RECT X Y W H – Draw a rectangle.
  With bottom left at (x, y), width w, height h.

- CRECT X Y W H – Draw a rectangle.
  Centred on (x, y), width w, height h.

- FONT FONTNAME – Set the current font.
  The FONTNAME must be one of the alphabetic fonts listed by LISTFONT.

- SYMHT H – Set the height characters, symbols and markers.
  The height is given in user bounds units.

- SYMANG A – Set the angle at which text is drawn.
  The angle is in degrees measured counter-clockwise from the positive X axis.

- CTEXT W "TEXT" – Draw text centred at the current position.
  The text is scaled so its width is w in user bounds units. If w is 0, the text isn't scaled. Instead, it is drawn at the size set with SYMHT.

- MARKER NUMBER YES – Draw marker NUMBER at the current position.

- PATH TYPE — Use the contents of the X,Y arrays (e.g. as set by READ or the evaluator (see below) as the vertices of a polyline. If TYPE is C (closed), join the last coordinate pair to the first. If TYPE is O (open) do not do that.

3.3 Graph Axis Commands

Before plotting a graph, **GPLOT** needs to know the range and type of the axes on which to plot the graph. The default settings of XYAUTO, XLINEAR and YLINEAR mean that *none* of the commands below *need* to be used before plotting, but you may want to use them.

- XYAUTO — Find both axis ranges automatically.
  DIMFILM will examine the range of the data in use when an XYPOINT, XYLINE or XYHISTOGRAM command occurs and will set the axis ranges to match. Linear axes will be used.

- XYSAME — Keep the previous axis ranges.
  If a graph was plotted with XYAUTO on, and other graphs need to be plotted on the same axes, use XYSAME after the first graph has been drawn.

- XRANGE XLO YHI — Set the X axis range.
  The X axis will run from XLO to XHI.

- YRANGE YLO YHI — Set the Y axis range.
  The Y axis will run from YLO to YHI.

- XLINEAR — Use a linear X axis.
  This is the default.

- YLINEAR — Use a linear Y axis.
  This is the default.

- XLOG — Use a logarithmic X axis.

- YLOG — Use a logarithmic Y axis.

- INTVALUES Which — Try to use integer values for an axis/axes.
  Which may be: NONE, X, Y, or BOTH. It is not always possible to do this, because of the range of the axis values. If it is not possible, floating point values will be used with no error message given.

3.4 Graph Plotting Commands

These are the commands that get the data to be graphed and actually draw graphs. Before data can be plotted, it must be READ, so the graph drawing commands (XYPOINT, XYLINE and XYHISTOGRAM) will not work until a READ has got some data. The graphs that can be drawn depend on what data is READ — this can be two real numbers per point (for (x, y)), three – which adds information for a symmetric Y error bar, or four – which adds data needed for asymmetric Y error bars, or symmetric Y and X error bars. The graph drawing commands will always use all the data available, so if READ has got data for (x, y) and a symmetric Y error bar, for example, that is what XYPOINT and XYLINE will draw. To stop drawing unwanted items, simply re-read the data omitting the undesired items.

- MAXPOINTS NUMBER – Set maximum number of data points.
  Memory for data to be plotted is allocated dynamically (this could be done in most major Fortran implementations, even though it is completely non-standard in Fortran-77 and earlier — there is much, possibly deliberate, ignorance on the part of academics of what could actually be done in this language). Four real numbers are allocated for each point — two for an (x, y) coordinate and two for error bar data. On starting, **GPLOT** allocates space for 1000 points. This can be changed at any time (although any data READ will be thrown away if it is changed, of course). The number of data points is limited by the amount of central memory on the machine, and how much of that you are allowed to use by your account

settings. Under emulation, there is no reason not to use the maximum available — 262,144 words of which 131,072 can be used by any one job (minus a bit, of course!). **GPLOT** is, unfortunately, not small ... and will grow as more facilities are added. At present, the maximum number of points allowed is set to 10000 and this is close to the maximum possible. The limit of addressable central memory is the main limitation on what can actually be done under NOS. Although extended memory and 'out-of-core' solutions (where most of the data resides on disk) may be possible in some cases, that isn't always so (e.g. **DIMFILM** needs the data it plots to be in central memory), and these methods always come with a significant performance and complexity cost.

Note

On "Unix-like" systems, this command has no effect. The default value is always used regardless of what may be specified here.

- READ NAME XCOL YCOL [YECOL [XECOL]] — Read a data file using the specified 'columns'. The file NAME (which must be a LOCAL file – possibly obtained inside **GPLOT** using the GET command) should contain numbers (in the standard character code), with at least one per line, separated by spaces or commas. There may be any number of spaces preceding or separating each number (subject to line length limitations – the limit is 80). Blank lines are skipped and ignored. Lines with the character 'C' in column 1 are ignored (allowing comments). The numbers may be integers or any real number format (exponential or not), optionally preceded by a + or − sign. READ will try to interpret anything that *could* be a number *as* a number. It will stop reading a data item when a non-numeric character is found — but this is not treated as an error.
  Each space (or single comma) separated number on a line is termed a 'column', and these are numbered starting at 1. So column 2 is the second number on each line. Column numbers must be specified for XCOL and YCOL data to obtain at least an (x, y) coordinate for each point. As a special case, XCOL may be 0, in which case, the X coordinate is set to the point number (starting at 1). This allows a data file with a single data item per line to be processed.
  If a column number is supplied for YECOL, then that column's values will be used for symmetric Y error bars. If a column number is also specified for XECOL, then that data will be used as either a symmetric X error bar or as the upper part of an asymmetric Y error bar, depending on the setting made with the ASYMYERRORBARS command. The default is to interpret YECOL and XECOL as the lower and upper extents of asymmetric Y error bars, as this may be more frequently useful, There is currently no way to draw asymmetric X error bars.
  If NAME is HERE (i.e. the string HERE), then no data file is opened. Instead, READ tries to read data from the command input stream itself (INPUT or an OBEY file – see below). READ will stop reading data items when it finds a line that contains the string EOF as its only item. This facility is mostly to allow other programs to write a self contained OBEY file that executes a complete plotting task then exits **GPLOT**. A major anticipated application of this is plotting data from APL.

- MARKER NUMBER — Set the marker number to be used for points.
  This is a character number in Font 9001. Characters are associated with numbers 2 to 25, but with 5 and 11 omitted. Numbers 2, 3, 4, 8, 9, 19 and 20 are perhaps the most useful. Something may need to be done to improve this font or at least supply descriptive names for the markers!

- XYPOINT — Draw an XY graph with points.
  The points will be drawn with the last marker set with a MARKER command, or 3 (the default) if no MARKER command is used. Depending on the data items READ and the ASYMYERRORBARS setting, symmetric Y, asymmetric Y or symmetric Y and X error bars will also be drawn.

- XYLINE — Draw an XY graph with lines.
  By default, data points will be joined with straight lines (linear interpolation – of a kind) and in SOLID line style. The line style can be changed with the STYLE command. Cubic or quintic interpolation can be used between the data points to get smooth curves instead of straight lines. At least 3 points are required for cubic and 5 for quintic interpolation. If there are insufficient points, linear interpolation is used as a fall-back. The interpolation is as supplied by **DIMFILM** — as always, whether it is appropriate depends on the data and its desired interpretation. I'm not certain exactly how **DIMFILM** comes up with the interpolation, although it is visually fine.

- XYHISTOGRAM — Draw an XY histogram.
  Bars (of a type determined by HISTSTYLE — default: ABUT) are drawn centered on the X coordinate and extending vertically from the $X = 0$ axis to the Y coordinate. Error bar data (if any) is ignored.

- HISTSTYLE STYLENAME [WIDTH] — Sets the histogram bar style.
  STYLENAME is one of the following:

  - ABUT — Open rectangular bars are drawn, centred on an X coordinate, and with a width determined by the X location of the preceding point (apart from the first point, which uses the location of the second point), so that the bars touch one another along the X direction.
  - ABUT+SHADE — as ABUT, but the bars are filled with 'shading' — lines at 45 degrees spaced 0.02 of the X BOUNDS range apart.
  - LINES — Zero width lines are used for the bars.
  - WIDE — Bars of the specified WIDTH (in X axis units) are drawn without shading.
  - WIDE+SHADE — As WIDE, but with shading.

- INTERPOLATE STYLE [N] — Set the interpolation mode for XYLINE.
  STYLE may be: LINEAR, CUBIC or QUINTIC — which are hopefully self explanatory! As explained in XYLINE, either straight lines or smooth curves can join the points. If CUBIC or QUINTIC is used, the number of intermediate (linear) segments drawn for the curve between each point is controlled by N (default: 10).

- ASYMYERRORS MODE — Use asymmetric Y error bars if MODE is ON.
  If OFF, symmetric Y and X error bars are drawn. Error bars will only be drawn if error bar data has been READ, and both YECOL and XECOL must have been specified to get asymmetric Y error bars or symmetric Y and X error bars.

- USEKEY — Prepare to create a key for a graph.
  A separate panel will be created to the right of the graph plotting area containing a key or legend to identify lines or sets of points on a graph.

- ADDKEY — Add a key entry for the data plotted by the previous XYLINE or XYPOINT command.
- KEYS — Create the accumulated key and draw it.
- GRAPHMODE STATE — Automatically set bounds so that graphs occupy as much area on a device as possible.
  STATE must be either ON or OFF. The bounds default to 0 to 1 on both axes. All graph plotting takes place within the current pane, which is identical to the bounds unless specifically set. However, the bounds area will be centered in the output device drawable area in order to keep a unit step on both axes the same (so circles are circles and not ellipses when drawn). To fully use the device drawable area, the bounds must have the same aspect ratio as the device. GRAPHMODE ON arranges for this to happen internally.
- SUBFIGGRID NX NY IX IY [SHRINK] — Set up to draw a graph as part of a rectangular array of sub-figures.
  This allows a pane to be automatically set up so that the next graph is plotted as one of an array of NX by NY graphs at the IX,IY coordinate of that grid. As always in **GPLOT**, grid coordinates start at 1 and range up to NX or NY inclusive. The optional SHRINK argument is a floating point number less than 1 which shrinks the pane about its center to control the separation of the graphs.
- GRMOVE X Y — Move the "pen" to graph coordinates (x, y), setting the "current position".
- GRDRAW X Y — Draw from the current position to (x, y), adjusting the "current position".

3.5 Graph Annotation Commands

Graphs almost always need some annotation. Minimal annotation is: ticks on the axes, numeric values against those ticks, and a framing rectangle drawn around the graph plotting area. These are always drawn after each graph drawing command (XYPOINT, XYLINE, XYHISTOGRAM) unless ANNOTATE OFF is used. TITLE, XLABEL, YLABEL and GRID annotation elements are only drawn if these are explicitly specified.

- ANNOTATE ON or OFF — Turn annotation on or off.
  Typically, annotation is wanted whenever a single graph is drawn. However, after drawing one curve (etc.) with annotation, it might be desirable to turn off annotation when drawing the other curves on the same axes. It is certainly more efficient to do so, and some devices with some line width settings may show visible changes when items are drawn over existing identical elements (for whatever reason).
- TITLE "TEXT" — Set the title.
  This is drawn at the top of the graph, centred on the width of the graph drawing area. TEXT may be enclosed in double quotes to allow spaces and format commands (such as *L and *U) are understood.
- XLABEL "TEXT" — Set the X axis label.
  This is plotted below the X axis (at the bottom of the graph drawing area.
- YLABEL "TEXT" — Set the Y axis label.
  This is plotted (vertically) to the left of the Y axis.

- GRID Style — Draw a grid with lines at the major axis ticks for one or both axes.
  Style may be: NONE, X, Y, or BOTH, which are hopefully self explanatory. Note that the grid will only be drawn if ANNOTATE ON is in effect (it will not be drawn if RIGHTANNOT ON is in effect).

- GSTYLE Mode — Choose the overall graph drawing style.
  Mode may be: BOXED, AXES or OPEN. The default is BOXED, with axis annotations along the edges of a boxed region. The OPEN option disposes of the right and top edges. AXES draws only axis lines that pass through a point which can be specified but defaults to $(0, 0)$. This gives a dramatically different appearance. Note that the point the axes pass through must be in the range of graph values to be drawn, otherwise one or both of the axes will be omitted..

- AXCUT X Y — Set the coordinates through which axis lines will pass.
  If using GSTYLE AXES this must be used to position the axes in the coordinate ranges being used, otherwise no annotation will appear.

- RIGHTANNOT ON or OFF — Selects annotation of the right hand edge for GSTYLE BOXED.
  This will only be drawn if ANNOTATE OFF has been previously used to stop left edge annotation. Using this with changes to YRANGE allows data with different Y value ranges to be drawn on the same plot.

- RYLABEL "TEXT" — Sets the right hand edge label.
  This is useful if RIGHTANNOT ON is used.

See also GLABEL in Section 5.3.

3.6 GPLOT System Commands

These commands don't draw anything, but they show the state of **GPLOT**, interact usefully with NOS, and allow something like plotting macros (with parameters) to be used.

- OBEY NAME [PARAMETERS] — Start reading commands from file NAME.
  The file may contain any **GPLOT** command (one per line), including OBEY. OBEY files may be nested up to 5 deep (this could be increased, but 5 seems likely to be enough). If PARAMETERS is specified, it should be a single string without spaces or a string enclosed in double quotation marks. Each space separated word in the PARAMETERS string defines a single parameter, which are given numbers based on position starting at 1 on the left. Up to 9 parameters can usefully be set up in the PARAMETERS string. When reading commands from an OBEY file, **GPLOT** will apply a simple parameter substitution process: each space separated word of the form: $N, where N is 1 to 9, will be replaced by the Nth word of the PARAMETER string. It is possible to pass parameters containing spaces by enclosing them in single quotes in the PARAMETER string. For example:

  ```
  OBEY FRED "FIRST 'SECONDA SECONDB'"
  ```

  defines two actual parameters that will be substituted for the formal parameters $1 and $2 wherever these occur as space separated words in the file FRED. $2 will, in fact, be substituted as:

```
"SECONDA SECONDB"
```

in order to preserve the embedded space(s). This is useful for passing TITLE strings to OBEY files, for example.

- HELP [TOPIC] — Output help information, including the supported output devices.
  HELP alone outputs a list of **GPLOT** commands, alphabetically sorted. If TOPIC is the name of a command, detailed information on that command is displayed. HELP HELP briefly described the HELP command itself. If TOPIC is DEVICES, information on the available output devices is displayed.

- STATUS — Displays various **GPLOT** state information.
  This includes the number of points available, whether anything has been read, the BOUNDS, whether any PANE or BLANK is in effect, etc.

- LOGFILE NAME — Opens a command log file called NAME.
  This is a LOCAL file on NOS to which any interactive input from you, the user, is written. Commands executed from any OBEY files are not logged. This lets you log a session and repeat it (perhaps after editing) by executing the log file as an OBEY file. You will need to SAVE or REPLACE the file to make it PERMANENT if you want to keep it between jobs. When LOGFILE is executed, any previously open log file is closed and a new one is opened. If the new one has a different name from the old, the old one will not be destroyed.

- MEMTEST — Test dynamic memory is working.
  Which it will be! More usefully, it also generates test data in the form of a sine wave, which can then be plotted to evaluate how well an output device is working, or to get familiar with **GPLOT**.

- GET NAME — Get an indirect access PERMANENT file.
  This is useful to avoid having to exit **GPLOT** to make some data file LOCAL so it can be READ. There is no command as yet to ATTACH a direct access PERMANENT file, although it would be easy to add one.

- LISTFONT — Lists the available fonts.
  The names of all available fonts are shown, along with the type of font: ALPHABETIC, SYMBOL or MARKER.

- VERSION [N] — Print (if N is omitted) the current version of **GPLOT**. If N is supplied, it should be the number of the string register (1 to 9) in to which to put version information.

- WAIT — Wait for input from the user if the device is interactive (**GTerm** or Tektronix 401x). The user should respond with Y (enter) or just (enter) to continue. If N (enter) is used, the request for a response will be repeated. If Q (enter) is used, **GPLOT** will perform an EXIT command and terminate.

- RESET — Reset most of **GPLOT** state to the default values. This does not include any data that has been READ or the selected device.

- PREFIX PATH — Set a prefix string to be pre-pended to all input file names.
  This is ignored on NOS. On "Unix-like" systems, it allows a directory to be set in which to look for all input files. The format of this string is not examined by **GPLOT**, but currently a / character is added to the end of it. This behaviour may change in future versions, as this may be inappropriate on other operating systems to which **GPLOT** may be ported.

- EXIT — Exit **GPLOT**.
  This closes any log file, flushes graphics commands to the output device, and shuts down **DIMFILM**. Then it exits **GPLOT** — surprise!

4 Defining and Using Functions

The ability to define and use functions is based on an **RPN evaluator** for simplicity, where operands are held on a **stack**. However, in this case, the values on the stack are arrays, each containing up to MAXPOINTS elements. These stack entries may also be used to hold scalar values as needed by the RPN operators. The scalar values are held in the first element of the arrays.

4.1 Overview, storage and RPN evaluator commands

There are a number of **commands** associated with the RPN evaluator and these are like other **GPLOT** commands. The RPN evaluator itself deals with **operators** which act on **operands** with an entire thing to be evaluated defined as an **RPN-string** consisting of comma separated operands and operator names. The operand and operator name strings are the two types of **tokens** understood by the RPN evaluator.

The size of the stack can be set with the command:

- NSTACK n

where n is an integer greater than or equal to 4. The default is 8. The number of words used by the stack is NSTACK * MAXPOINTS and this must be compatible with the maximum field length. The first four stack entries are also used to store X and Y points and X and Y error bars for graph plotting, so NSTACK must be at least 4.

The stack entries are numbered 0, 1 ... NSTACK-1. Stack entry 0 is special:

- It can only be written to by the ERANGE command or the XLIN, XLOG and SETX RPN operators.

- Writing to it with the ERANGE command, or the XLIN or XLOG operators, sets the evaluation range and the number of active elements in the stack arrays – i.e. the array length, NELEM for evaluation purposes[1].

- Its contents can be accessed by the X RPN operator.

It is usual for the contents of stack level 0 to define the values for which the function will be evaluated, which is why its contents are accessed with the operator named X. Since the effective

[1] Also referred to as NEVAL internal to GPLOT.

array length must be known for the vast majority of operators to work, either the ERANGE command *must be used before* any RPN is evaluated or the XLIN or XLOG operators must be used before any other RPN operators. The contents of stack level 0 will be called the **X range array.**

Stack level 1 is also special in that it contains the Y values corresponding to the X values in the X range array when plotting graphs. This may be referred to as the **Y value array**. There is an RPN operator which explicitly sets this by copying the contents of the top-of-stack (TOS) array into it.

Three other groups of storage are also defined for use with function definition and evaluation:

- **Scalar Registers**: There are 9 registers (named 1 to 9) that can be used with STO and RCL commands and RPN operators to save scalar variables.
- **String Registers**: There are 9 registers (named 1 to 9) to hold strings of up to 80 characters. These can be set with the STRING command and accessed by RPN operators.
- **Procedure Registers**: These 9 registers (named 1 to 9) are used to hold RPN procedures as strings that can be 'called'. The contents can be set with the command PROC to a literal string, or loaded from a procedure library file using the LOADPROC command (see below).

A procedure is defined in the form of an **RPN-string** as a series of comma separated tokens:

`TOKEN,TOKEN,...,TOKEN`

A TOKEN is a numeric constant operand or an RPN operator. Unlike commands, RPN operator names cannot be abbreviated.

There are two commands which can cause an RPN-string to be evaluated. The main one is:

- `EVAL RPN-string ["arguments"]`

The RPN-string for EVAL can 'call' procedures stored in Procedure Registers (as RPN-strings) using the token @n where n is the Procedure Register number (1 to 9). The 'calling' of procedures in Procedure Registers is actually done by string substitution. There is no true call-return mechanism. If a stored procedure contains @n, it will also be substituted. The substitution continues until no @n sequences are found in the expanded procedure string.

There is an optional second argument to EVAL which can be used to pass parameters to a procedure, in a sense. If present, it should be a comma separated list of up to 9 numeric values (or $n for parameter substitution in scripts). These values are placed into successive Scalar Registers starting with register 1. Using these registers, along with String Registers, is how parameters can be given to RPN procedures. Inside those, the parameter values are accessed with the RCL operator.

A very useful variant on EVAL is:

- `ITEVAL start end step RPN-string ["arguments"]`

which will evaluate the RPN-string `max( int( (end - start + step) / step ), 0 )` times, as per a Fortran DO-loop. The value of the loop counter can be accessed using the token I in the RPN-string. The I token has the value 1 if used in an RPN-string evaluated via EVAL.

There are RPN operators which directly produce graphical output. It is sometimes useful to know the bounds of that output before actually drawing anything, and three commands are provided to do that: BBSTART, BBEND and BBSET. See the next Section for details.

4.1.1 Evaluator Related Command Descriptions

A full list of the commands provided for the RPN evaluator follows.

- ERANGE BASE START STOP NELEM — Set the contents of stack level 0 (or X) to a set of equally spaced values.
  This also sets the effective array length to NELEM. BASE is 1 for linearly spaced values or the base of the logarithm for logarithmically spaced values. START is the starting value for the range. For logarithmic ranges, the BASE is raised to START for the starting value. STOP is the ending value for he range (the range is inclusive). For logarithmic ranges, the BASE is raised to END for the ending value. NELEM is the number of steps to make between START and STOP.
  For example: ERANGE 2 1 8 8 results in 2, 4, 8, 16, 32, 64, 128 and 256 appearing in X.

- EVAL RPN-string ["arguments"] — See above.

- ITEVAL start end step RPN-string ["arguments"] — See above.

- PROC N RPN — Put an RPN string into Procedure Register N (1 to 9).

- LOADPROC N NAME — Copy a named procedure from the GPLPROC procedure library file to Procedure Register N. For more information on this, see Section 4.4.

- STRING N TEXT — Set the contents of String Register N to TEXT. The only way to pass strings to the RPN evaluator is through String Registers.

- STO N X — Put X into Scalar Register N. The only way to pass numeric parameters to the RPN evaluator is through Scalar Registers. The optional "arguments" string to EVAL and ITEVAL is a conveneient way of setting a group of these registers to pass parameters to a procedure.

- RCL N — Display the contents of Scalar Register N.

- ZEROVAL V — Set the value of divide-by-zero avoidance and some other tolerance tests. The default is 10^{-9}.

- BBSTART prevents RPN operators that would normally draw things from doing so, but instead has them accumulate bounding box information.

- BBEND restores normal operation of the graphics drawing RPN operators.

- BBSET sets the bounds to the accumulated bounding box and is equivalent to issuing a BOUNDS command with the appropriate arguments.

4.2 How the RPN evaluator works

The RPN evaluator's basic operation given an `RPN-string` is as follows:

1. Split `RPN-string` into tokens using commas as token separators.
2. Loop over the tokens.
3. For each token, first check if it could be an operand:
   (a) If it can be interpreted as a decimal number (integer, or real using floating point or scientific notation), treat it as an operand and push a constant array with every element set to this value on the stack (if there is room – if the stack would overflow abandon evaluation with an error message).
   (b) If the token is `X`, push the X range array on to the stack.
   (c) If the token is `PI`, push a constant array with every element set to π on to the stack.
   (d) If the token is `TWPI`, push a constant array with every element set to 2π on to the stack.
   (e) If the token is `PI/2`, push a constant array with every element set to $\frac{\pi}{2}$ on to the stack.
   (f) If the token is `E`, push a constant array with every element set to e on to the stack.
   (g) If the token is `I`, push a constant array with every element set to the iteration number on to the stack if the evaluation is being done from `ITEVAL` or 1 otherwise.
4. If it is not an operand, see if it is a known operator. If not, that is an error and evaluation is abandoned.
5. Given that the token is a known operator, the number of operands it needs is known. If there are insufficient operands on the stack, abandon evaluation with stack underflow.
6. Try to execute the operator. It is applied to all elements of its operand arrays to produce one or more results, which are usually also naturally arrays, but if the result is naturally scalar, all elements of the output array are set to that value. The operands are popped off the stack and the results are pushed on to it.
7. In the rare case that an operator has more results than operands, a check is made for stack overflow. If that would happen, evaluation is abandoned.
8. Some operators can also fail with some operands which are not within the domain of the operator or would result in divide by zero. If these conditions occur, evaluation is abandoned with a domain error or divide by zero error. The tolerance for possible divide by zero conditions can be set with the `ZEROVAL` command before evaluation.
9. Some operators cause graphics output as a 'side effect' . . . albeit a rather critical one!

The RPN evaluator, while useful, has a number of fundamental limitations.

- There are no jumps, conditional branches or looping constructs. This makes it less powerful than the RPN in programmable calculators. (There is conditional value selection operating on array elements, but that is a different thing.) The only form of looping available is through `ITEVAL`, which is useful, but limited.

- The command line (and all other input lines in **GPLOT**) is limited to 80 characters. Any lines longer than this are truncated at 80 characters. This means the length of an RPN string must be significantly less than this. For many problems, the limited size of procedures this implies makes it impossible to use the RPN evaluator to solve them. Using procedures stored in the GPLPROC procedure library is a useful way of working around this, as explained below. Using this, the maximum size of a procedure is 696 characters, which is not huge, but enough for many tasks.

4.3 Operands and Operators

In the following descriptions of the available operators, the symbol `A` is an array of `NELEM` values existing as a operand on the stack. The symbol `C` is a scalar value (a simple constant) existing as an operand on the stack and actually stored in the first element of the `NELEM` length array used for all operands. The convention used for describing Forth operators is used:
`( O1 O2 ...- R1 R2 ...)`
where `O1` (etc.) are the operands that must be on the stack before the operator is used (and it consumes them), and `R1` (etc.) are the values left on the stack by the operator. The `-` signifies the action of the operator. `TOS` is short for Top Of Stack.

4.3.1 Constants

Most of these tokens set the top-of-stack (TOS) to an arbitrary value or to one of several standard constants, all of which need no further description. There are two operators in this group and these are:

- `I` : This is the iteration number if the `RPN-string` is being evaluated from `ITEVAL` or the constant 1 otherwise.
- `IDX` : This fills the TOS array with the index of its array elements – `1 ...MAXSTACK` – so that `A[I]=I`.

Token	Actions	Description
<DIGITS>	(- C1)	SET TOS ARRAY TO A LITERAL CONSTANT.
X	(- A1)	SET TOS TO X RANGE ARRAY.
PI	(- C1)	SET TOS TO PI.
E	(- C1)	SET TOS TO E.
TWPI	(- C1)	SET TOS TO 2 PI.
PI/2	(- C1)	SET TOS TO PI / 2.
I	(- C1)	SET TOS TO ITERATION NUMBER FROM ITEVAL. 1 IF IN EVAL.
IDX	(- A1)	SET TOS TO THE ARRAY ELEMENT INDEX.

4.3.2 Stack Manipulation

These operators provide the essential stack operations needed by all RPN evaluators. They also provide:

- `SETX` and `SETY` : Easy ways of setting the stack levels used as X (also evaluation range) and Y values for graph plotting from the TOS values.

- XLIN and XLOG : Sets the X (evaluation range) stack level to equally spaced values in a linear or logarithmic space.

- I0IJ and I1IJ : Convert a 1D, 1 based, array index to a pair of 2D array indices for a 2D array with a given first dimension. The first of these generates 0 based 2D array indices and the second generates 1 based indices. In all cases, the specified first dimension of the 2D array must be > 0!

Token	Actions	Description
SWAP or S	(A1 A2 - A2 A1)	SWAP OR EXCHANGE TOP 2 STACK ARRAYS.
DUP or &	(A1 - A1 A1)	DUPLICATE TOP OF STACK (TOS)
POP	(A1 -)	POP TOP OF STACK
G	(...AN ...C1 - AN)	GET STACK LEVEL AT DEPTH C1 INTO TOS
CL	(-)	CLEAR STACK
SETX	(A1 - A1)	OVERWRITE RANGE / GRAPH X VALUES WITH TOS: X = A1
SETY	(A1 - A1)	OVERWRITE GRAPH Y VALUES WITH STACK TOP VALUES: Y = A1
XLIN	(C1 C2 C3 -)	X = C1 TO C2 IN C3 LINEAR STEPS
XLOG	(C1 C2 C3 C4 -)	X = C1**C2 TO C1**C3 IN C4 STEPS
EL	(A1 C2 - A1 C2)	C2 = A1[C2]
I0IJ	(C1 C2 - C1' C2')	C1'=MOD(C1,C2), C2'=(C1/C2) : 1D TO 2D INDICES, 0 BASED.
I1IJ	(C1 C2 - C1' C2')	C1'=MOD(C1,C2)+1, C2'=(C1/C2)+1 : 1D TO 2D IDX, 1 BASED.

4.3.3 Basic Arithmetic

These operators provide the arithmetic operations of a basic RPN calculator, but operating on arrays.

Token	Actions	Description
+	(A1 A2 - A1)	ADD ARRAY ELEMENTS: A1 = A1 + A2
-	(A1 A2 - A1)	SUBTRACT ARRAY ELEMENTS: A1 = A1 - A2
R-	(A1 A2 - A1)	REVERSE SUBTRACT: A1 = A2 - A1
*	(A1 A2 - A1)	MULTIPLY ARRAY ELEMENTS: A1 = A1 * A2
**	(A1 A2 - A1)	EXPONENTIATION: A1 = A1 ** A2
/	(A1 A2 - A1)	DIVIDE ARRAY ELEMENTS: A1 = A1 / A2
R/	(A1 A2 - A1)	REVERSE DIVIDE: A1 = A2 / A1
RCP	(A1 - A1)	RECIPROCAL A1 = 1.0 / A1
CHS	(A1 - A1)	A1 = -A1
ABS	(A1 - A1)	A1 = ABS(A1)

4.3.4 Math Functions

These operators provide all the standard functions that would be found on a scientific calculator, but operating on arrays.

Token	Actions	Description
SIN	(A1 - A1)	A1 = SIN(A1)
COS	(A1 - A1)	A1 = COS(A1)
TAN	(A1 - A1)	A1 = TAN(A1)
ASIN	(A1 - A1)	A1 = ARCSIN(A1)
ACOS	(A1 - A1)	A1 = ARCCOS(A1)
ATAN	(A1 - A1)	A1 = ARCTAN(A1)
SINH	(A1 - A1)	A1 = SINH(A1), DOMAIN |X| < 742.36
COSH	(A1 - A1)	A1 = COSH(A1), DOMAIN |X| < 742.36
TANH	(A1 - A1)	A1 = TANH(A1), DOMAIN |X| < 742.36
SQRT	(A1 - A1)	A1 = SQRT(A1)
LOG	(A1 - A1)	BASE E LOGARITHM: A1 = LN(A1)
LG10	(A1 - A1)	BASE 10 LOGARITHM: A1 = LOG10(A1)
LOG2	(A1 - A1)	BASE 2 LOGARITHM: A1 = LOG2(A1)
EXP	(A1 - A1)	A1 = E ** A1 DOMAIN: -675.81 TO 741.66
RAND	(A1 - A1)	UNIFORMLY DISTRIBUTED RANDOMS [0:A1): A1 = A1 * RANDOM
SEED	(C1 -)	SET RANDOM NUMBER SEED TO A1[1]
GCD	(C1 C2 - C1)	FIND GREATEST COMMON DIVISOR OF C1, C2

4.3.5 Number Range Related Functions

Token	Actions	Description
MIN	(A1 A2 - A1)	A1 = MIN(A1,A2)
MAX	(A1 A2 - A1)	A1 = MAX(A1,A2)
MOD	(A1 A2 - A1)	A1 = MOD(A1,A2) OR A1 = A1 % A2
SIGN	(A1 - A1)	A1 = (A1 < 0) ? -1 : 1

4.3.6 Conditionals

There functions compare arrays of numbers and set a result array to 0 or 1 (false or true) depending on the result of that comparison. There is also a SEL operator which uses the result of a comparison to select elements from one array or another. There is currently no way to conditionally execute code in the RPN-string, so they do not provide anything like IF ...THEN ... functionality.

Token	Actions	Description
ODD	(A1 - A1)	A1 = 1 WHERE INT(A1) IS ODD ELSE 0
GT	(A1 A2 - A1 A2 A3)	A3 = (A1 > A2) ? 1 : 0 ;
LT	(A1 A2 - A1 A2 A3)	A3 = (A1 < A2) ? 1 : 0 ;
LE	(A1 A2 - A1 A2 A3)	A3 = (A1 <= A2) ? 1 : 0 ;
GE	(A1 A2 - A1 A2 A3)	A3 = (A1 >= A2) ? 1 : 0 ;
EQ	(A1 A2 - A1 A2 A3)	A3 = (A1 == A2) ? 1 : 0 ;
NE	(A1 A2 - A1 A2 A3)	A3 = (A1 != A2) ? 1 : 0 ;
NOT	(A1 - A1)	A1 = (A1 == 0) ? 1 : 0 ;
SEL	(A1 A2 A3 - A1 A2 A3)	A3 = (A3 == 0) ? A1 : A2 ;

4.3.7 Number Type Conversions

These operators can the integer and fractional parts of a number and convert a number to the 'nearest' integer in various ways. Numbers are always kept as floating point internally, though.

Token	Actions	Description
INT	(A1 - A1)	TRUNCATION A1 = INT(A1)
FRAC	(A1 - A1)	FRACTIONAL PART A1 = FRAC(A1)
FLR	(A1 - A1)	A1 = FLOOR(A1)
CEIL	(A1 - A1)	A1 = CEILING(A1)

4.3.8 Graphics

These operators provide a pretty complete set drawing functions. If the BBSTART command has been given, they do not actually draw anything. Instead, they accumulate a bounding box, which can be used by issuing a BBSET command, which sets the plotting bounds to the accumulated bounding box. Using the BBEND command restores the normal drawing functions of these operators.

Token	Actions	Description
M	(C1 C2 -)	MOVE TO (C1,C2)
D	(C1 C2 -)	DRAW TO (C1,C2)
C	(C1 C2 C3 -)	DRAW CIRCLE, CENTER C1,C2 RADIUS C3.
A	(C1 C2 C3 C4 C5 -)	DRAW ARC, CENTER C1,C2 RADIUS C3, ANGLES C4 TO C5.
BOX	(C1 C2 C3 C4 -)	DRAW RECTANGLE, BOTTOM LEFT C1,C2 WIDTH C3 HEIGHT C4.
HDSH	(A1 - A1)	DRAW RELATIVE (1,0) IF A1[I] > 0 ELSE MOVE
VDSH	(A1 - A1)	DRAW RELATIVE (0,1) IF A1[I] > 0 ELSE MOVE
PTHO	(A1 A2 -)	DRAW POLYLINE X=A1,Y=A2, OPEN
PTHC	(A1 A2 -)	DRAW POLYLINE X=A1,Y=A2, CLOSED
T	(C1 -)	DRAW CONTENTS OF STRING REGISTER C1 AS TEXT.
TVF	(C1 C2 - C1)	DRAW C1 AS TEXT USING FORMAT STRING IN C2.
TVI	(C1 C2 - C1)	DRAW INT(C1) AS TEXT USING FORMAT STRING IN C2.
TS	(C1 -)	DRAW SYMBOL WITH CODE NUMBER C1
TM	(C1 -)	DRAW MARKER WITH CODE NUMBER C1
TC	(C1 -)	DRAW CHARACTER WITH CODE C1 FROM CURRENT ALPHABET
TH	(C1 -)	SET TEXT/MARKER/SYMBOL HEIGHT
TA	(C1 -)	SET TEXT DRAWING ANGLE IN DEGREES CCW
TSC	(-)	START TEXT CONTINUATION
TEC	(-)	END TEXT CONTINUATION
TLEN	(C1 - C1)	LENGTH IN BOUNDS UNITS AT HEIGHT=1 OF STRING IN REGISTER C1.
ROT	(A1 A2 C3 - A1 A2)	ROTATE X=A1,Y=A2 BY C3 RADIANS.
TRN	(A1 A2 C3 C4 - A1 A2)	TRANSLATE X=A1, Y=A2 BY C3,C4.
SCL	(A1 A2 C3 C4 - A1 A2)	SCALE X=A1, Y=A2 BY C3,C4.
R2D	(C1 - C1)	RADIANS TO DEGREES.
D2R	(C1 - C1)	DEGREES TO RADIANS.
LAB	(C1 C2 C3 C4 C5 -)	LABEL AT (C1,C2), LEN C3, ANG C4, SR C5

4.3.9 Output and Memory Functions

As with a calculator, the RPN evaluator provides a small number of ‘memories’ (9) to which values can be stored and then retrieved.

These operators also give a way of examining arrays on the stack or the whole stack, which can be very useful for debugging.

Token	Actions	Description
STO or =	(A1 C2 - A1)	STORE A1[1] (A.K.A. C1) IN REGISTER C2
RCL or #	(C1 - C1)	RECALL CONST VALUE FROM REGISTER INT(C1)
P	(-)	PRINT TOS ARRAY FIRST AND LAST ELEMENTS
PE	(A1 C2 C3 - A1)	PRINT ELEMENTS C2 TO C3 OF A1 IN FREE FORMAT
PC	(C1 - C1)	PRINT C1 (TOS A[1]) IN FREE FORMAT.
DUMP	(-)	PRINT ALL STACK LEVELS FIRST AND LAST ELEMENTS

4.4 The GPLPROC Procedure Library

The file GPLPROC can be used to define named procedures for the RPN evaluator to use. This file is automatically made LOCAL on NOS whenever LOADPROC is used.

The format of named procedure definitions in GPLPROC is very simple.

```
  PROCNAME [nlines]
  RPN-string
  . . .
  RPN-string
```

For example:

```
C AN UNROTATED ELLIPSE,  "XC YC A B"
ELLIPSE
CL,0,TWPI,360,XLIN,X,&,COS,3,#,*,1,#,+,S,SIN,4,#,*,2,#,+,PTHC
```

defines a single line procedure called ELLIPSE, which can be loaded into a Procedure Register with:

```
LOADPROC 1 ELLIPSE
```

would load ELLIPSE into Procedure Register 1, and

```
EVAL @1 "1.0,2.0,2.0,0.5"
```

will then draw an ellipse.

As with scripts (obey files), full line comments can be inserted by starting the line with "#" or "C ".

If the procedure name is followed by the optional nlines value (an integer in the range 2 to 9), procedures longer than (at the very most) 80 characters can be defined. This is implemented by putting consecutive lines into successive Procedure Registers. All lines except the last have ,@n added to their ends automatically, where n is the procedure line number plus 1. This has three consequences:

- The procedure substitution mechanism will "chain" the Procedure Register contents together into one RPN string.
- All lines (except the last) in such a procedure definition may have no more than 77 characters.
- The first nlines Procedure Register contents will be overwritten.

An example of a multi-line procedure definition is:

```
C ANGLE PAIR "X Y LENGTH ROT1-DEGREES ROT2-DEGREES"
ANGLEPAIR 4
CL,0,1,2,XLIN,X,0,3,#,DUP,SCL,4,#,D2R,ROT,1,#,2,#,TRN,PTHO
4,#,9,=,5,#,4,=
CL,0,1,2,XLIN,X,0,3,#,DUP,SCL,4,#,D2R,ROT,1,#,2,#,TRN,PTHO
9,#,4,=,CL,1,#,2,#,3,#,2,/,4,#,5,#,A
```

There is no specific limit to the number of procedures that can be defined in GPLPROC, although every LOADPROC just searches the file sequentially from the start for the name, so it will get slow eventually. There is also no limit on the length of procedure names, except for the universal 80 character line length restriction.

5 Higher Level Drawing Commands

While **DIMFILM** provides a comprehensive set of graph plotting facilities (many – but not all – of which can be accessed via **GPLOT**), its general purpose drawing facilities are somewhat limited, at least by today's standards. To address this, **GPLOT** has added some "higher level" features, which ultimately use **DIMFILM**'s functionality, but may make some tasks much easier. Rather than try to provide very general extensions, such as a stack of transformation matrices as in **PostScript**, the **GPLOT** extensions are aimed at specific common tasks.

These higher level drawing commands are based on three new components:

- **Boxed text**. These provide a simple way of creating a rectangle containing text (one or more lines) with several options. Such things are the basic building blocks of block diagrams and similar things. They can also be used to create tables.
- **Decorated lines**. These are polylines, the ends of which can be "decorated" with arrow heads or things we call "skips" which can give the impression of one line "jumping over" another. Additionally, the midpoint of a line segment can contain a very short textual label (typically one or two characters) which will be drawn inside a circle.
- **Labels**. These are single lines of text, enclosed in a box, from which an arrow points to a specified location. The length of the arrow, and the direction (given as an angle in degrees counter-clockwise around the "pointed at" position from the positive X axis) are also specified.

5.1 Boxed text

These commands draw a boxed text and establish its parameters:

- BOXTEXT [X Y] — This draws a boxed text object at the location (x, y) (if supplied) or at the last incremented boxed text position (see BOXPDELTAS). TEXT is the text to be placed in the box. This may have up to 5 lines with lines separated by back-slash characters. All **DIMFILM** string markup applies. The appearance of the text and various other options are controlled by the following BOXP... parameter setting commands.

- BOXPSIZE WIDTH HEIGHT BOTLEFT — This sets the size of the (outer) box. All text will be drawn within this box and it may, under some circumstances, be clipped against it. WIDTH and HEIGHT are the dimensions of the box in BOUNDS units. If BOTLEFT is YES, the box coordinates will be used for the bottom left point. If BOTLEFT is NO, they will be used as the centre point.

- BOXPBOX WHICH — Controls which box outlines are drawn. WHICH may be:

  - NONE — No outlines are drawn.
  - OUTER — The outer box, with dimensions specified with BOXPSIZE, will be drawn.
  - INNER — The inner box, tightly surrounding the text, will be drawn. This area is protected from hatching (if any), whether the outline is drawn or not.
  - BOTH — Both box outlines are drawn.

- BOXPTEXT WIDTHFRAC MODE FLUSHLEFT — Controls how the text is drawn. MODE may be:

  - FIX — In this case, the text is not scaled to fit the box directly. Instead, a SYMHT is calculated so that WIDTHFRAC lines will fit inside the box. If any line is too long with this SYMHT, it will be clipped at the box edges. All boxed text objects with the same height and WIDTHFRAC will have a consistent text size in a diagram. WIDTHFRAC will typically be an integer, 1 or greater.
  - SCALE — In this case, the text is scaled so that the longest line occupied WIDTHFRAC of the box width. In this case, WIDTHFRAC will typically be a floating point number less than 1 and greater than 0. There may be no consistency between text sizes in different boxed texts in a diagram with the SCALE option.

  If FLUSHLEFT is YES, the text will be left justified in the inner box. If NO, the text will be centred in the inner box.

- BOXPHATCH SPACING IANGLE MODE — This controls how the boxed text is hatched (or not). MODE may be:

  - NONE — No hatching will be drawn.
  - VERT — Hatch lines will be draw joining the bootom and top edges of the box.
  - HORIZ — Hatch lines will be drawn joining the left and right edges of the box.
  - BOTH — Both HORIZ and VERT hatching will be drawn.

  SPACING is the distance between hatch lines along the edges in BOUNDS units. IANGLE is the number of spacings on the opposite edge that the end point of a hatch line will be displaced, controlling the angle at which it is drawn.

- BOXPDELTAS DX DY — Set the auto-increment of boxed text positions after each boxed text is drawn. If a BOXTEXT is specified without an explicit position, the result of adding these deltas to the last BOXTEXT position will be used as the new position. This is very useful for constructing tables, which can easily be built from abutting boxed text objects.

5.2 Decorated lines

These commands draw, or set the parameters of, a decorated polyline:

- LINE LINESPEC — This defines and draws a decorated line. LINESPEC is a string (not quoted, as it cannot contain spaces) which describes the line using a series of POINTSPECs separated by > characters (which may be read as "go to" characters here). I.e.
  LINE POINTSPEC>POINTSPEC...POINTSPEC
  A POINTSPEC consists of a single character TYPE, then an X coordinate followed by a Y coordinate, optionally followed by a very short ANNOTATION string. X, Y and ANNOTATION are separated by commas. I.e.
  <TYPE>X,Y[,ANNOTATION]
  The possible values for TYPE are:
  - P — Plain. This point will be undecorated.
  - A — Arrow. This point will be decorated with an arrow head. If this is the first POINTSPEC in a LINESPEC, the arrow will point out from the start of the polyline If it is the last POINTSPEC, it will point out from the end of the polyline. If used on intermediate POINTSPECs, an arrow will be drawn at both the end of one line segment and the start of the next, with the arrows pointing towards each other (which is rarely useful).
  - S — Skip. This is (more or less) a quarter-circle. If this is the first POINTSPEC in a LINESPEC, the quarter-circle will start at the top of a circle and fall down to the line segment. If this is the last POINTSPEC in a LINESPEC, it will rise up from the line segment to the top of the quarter-circle. Using S on intermediate POINTSPECs will cause a half-circle to be drawn.
- ARROWPARM TYPE SIZE SHARPNESS BARBEDNESS — Sets how arrow heads are to be drawn. TYPE may be:
  - OPEN
  - CLOSED
  - BARBED

  These are probably self-explanatory. SIZE is the tip-to-base length in BOUNDS units. SHARPNESS control how sharp the arrow head looks in the range 0.5 (very broad) to 5 (very sharp) with 2 being a good default. The BARBEDNESS controls how deep the barb is, using a fraction of the tip-to-base distance. 0.3 is a good default. Note that these parameters also control the appearence of arrow heads in labels too.
- LINEPARM SKIPSCALE ANNOTSCALE — Sets the size at which skips and mid-line annotations are drawn. SKIPSCALE is half the length of a skip in BOUNDS units. That is, two abutting skips will cover this distance. ANNOTSCALE is twice the diameter of the annotation enclosing circle in BOUNDS units.

5.3 Labels

These let you point at something with a short message. These commands draw labels:

- ALABEL X Y LENGTH ANGLE TEXT — Add a label pointing at BOUNDS coordinates (x, y). LENGTH is the distance from the point (x, y) to the centre of the box containing the label text in BOUNDS units. ANGLE is the angle the arrow line makes with the +X axis in degrees. TEXT is a (short) label string, enclosed in double quotes if it contains spaces. If it is entirely spaces, the arrow is drawn without any box at its "far" end. All **DIMFILM** "string markup" can be used.

- GLABEL GX GY LENGTH ANGLE TEXT — Add a label pointing at graph coordinates (x, y). This is useful for labelling points on graphs. Note that LENGTH is in BOUNDS units. It is not possible to use "graph lengths" as the axis ranges may be wildly different or log axes may be in effect. The arguments are the same as for ALABEL apart from the meaning of the coordinates.

6 Defining and drawing L-systems

Leveraging the functionality added for general purpose function evaluation described above, a quite specialised but fun facility for defining and drawing L-systems is also available. Lindenmayer (L) systems were invented by the theoretical biologist Aristid Lindenmayer in the 1960's to describe the growth and development of multicellular organisms such as algae and plants. They can also be used to draw a variety of fractal curves.

L-systems are somewhat abstract and hard to grasp, but the graphical results make them worth persevering with. They are based on formal systems in which there is a predefined alphabet of allowed symbols, some of which can be interpreted as drawing commands. Others are place holders which have no associated graphical interpretation. Once an L-system is defined, it can be *developed* over a number of iterations. Once fully developed, it will be drawn by executing the drawing commands associated with certain symbols. The allowed symbols are arranged as strings, and the development of an L-system proceeds by a series of string manipulations – primarily string substitution.

An L-system is defined by four things:

- An axiom string. This is the starting point from which the final string to be drawn will be developed over a given number of iterations. In **GPLOT**, this is stored in string register 1.

- A set of production rules. These are used to rewrite the string in each iteration. There can be up to 8 rules, stored in string registers 2 to 9. These begin with the symbol to be substituted, followed by a colon, then the symbols which are to be substituted in place of that symbol each time the symbol is encountered in the current string (initially the axiom string, then the result of the previous application of the production rules).

- A turning angle, used when interpreting the final string as graphics commands.

- An initial angle, stored in memory register 3, as well as the initial (x, y) starting point for drawing being in memory registers 1 and 2.

The **GPLOT** L-system facility uses the following alphabet:
`F, A, B, C, D, E, M, X, Y, +, -, [, ]`

of which A, B, C, D, E, F, M, X, Y may be substituted with another string defined in a production rule. The graphics state used when drawing an L-system string consists of the current (x, y) position and a current drawing angle along which a virtual pen can move. The symbols with a graphical interpretation are:

- F – draw a unit length line segment at the current drawing angle.
- D – as F but two different substitutable draw symbols are needed.
- M – move by unit length at the current drawing angle.
- + – turn counter-clockwise by the turning angle.
- - – turn clockwise by the turning angle.
- [– push the current position and angle on to a stack.
-] – pop a position and angle off the stack and make them current.

An example of defining an L-system as an OBEY file is:

```
STRING 1 X
STRING 2 F:FF
STRING 3 X:F-[[X]+X]+F[+FX]-X
STO 3 90
LSYSTEM 2 $1 22.5
```

The (single) parameter that must be passed to this is the number of iterations to perform before drawing the result. This example draws something that looks remarkably similar to a straw-like plant.

7 Output Devices

GPLOT supports these devices:

- GTERM — **GTerm** colour graphics terminal.
  This is a Python program I wrote to allow APL to be used with its proper character set, as well as to function as a better alternative to an xterm emulation of a Tektronix 401x for vector graphics output (e.g. that from **GPLOT**). Unlike Tek 401x emulations, it supports colour, anti-aliasing[2] and anything displayed can be saved as SVG. **GTerm** runs on Linux and macOS, but not on any version of Windows.

- TEK4K — Tektronix 4014 graphics terminal emulation.
  This should work with any Tek 401x emulator. It has been tested with xterm and with René Richarz's excellent 4014 emulator. The latter has much higher quality graphics than xterm and reproduces other features of the Tektronix DVST displays for a very realistic experience.

[2] René Richarz's 4014 emulator is nicely anti-aliased, but xterm is not.

- EPSCOL NAME [SIZE] — Output to a colour EPS file.
  Output goes to the LOCAL file NAME. As of V0.58, 'real' EPS is generated directly in NOS, using 6/12 character encoding which supports all printable ASCII characters. SIZE is of the form: W,H+X+Y where W is the width, H the height and X and Y are offsets from the bottom left – the units are inches in all cases.

- SVG NAME [SIZE] — Output SVG (Scalable Vector Graphics) to a file. This can be included in HTML files, and this format is used by the PLNOTES application on NOS to insert graphs and graphics into documents. SIZE is of the form: W,H which are the width and height in "pixels" (although, this does not affect the "resolution", just the size browsers and other software assign to it).

There is also an EPSBIN device, which supports only black and white (i.e. binary) output colours, but this is no longer thought to be useful and will not be described further.

The length of NAME must be kept to 4 (!) characters on NOS and COS. The full output file name is formed from the name supplied here, followed by 3 leading zero padded decimal digits (starting at 001). With a 4 character NAME, this makes 7 characters in total, which is the maximum NOS and COS file name length. If NAME is longer than 4 characters, it is truncated to 4 by **GPLOT**.

On Unix, the length of NAME may be up 72 characters (before truncation by **GPLOT**).

At the end of each frame (i.e. when a CLEAR command is used or when **GPLOT** is exited), the frame number part of the name is incremented after the output file has been written. So, if NAME is EPXA then a series of output files may be generated on NOS and COS: EPXA001, EPXA002 ...

On Unix systems, the name may be more descriptive, use mixed case and will also be automatically given the appropriate extension. However, the output file names will be translated to all lower case by **GPLOT**. E.g. If NAME is ExampleA, the output files may be named examplea001.eps, examplea002.eps ....

8 Building and Maintaining GPLOT and DIMFILM

Please see the main README.md for the **GPLOT** project on Github for detailed information on how to build and install **GPLOT** on both NOS and "Unix".

Part II

GPLOT Tutorial and Examples

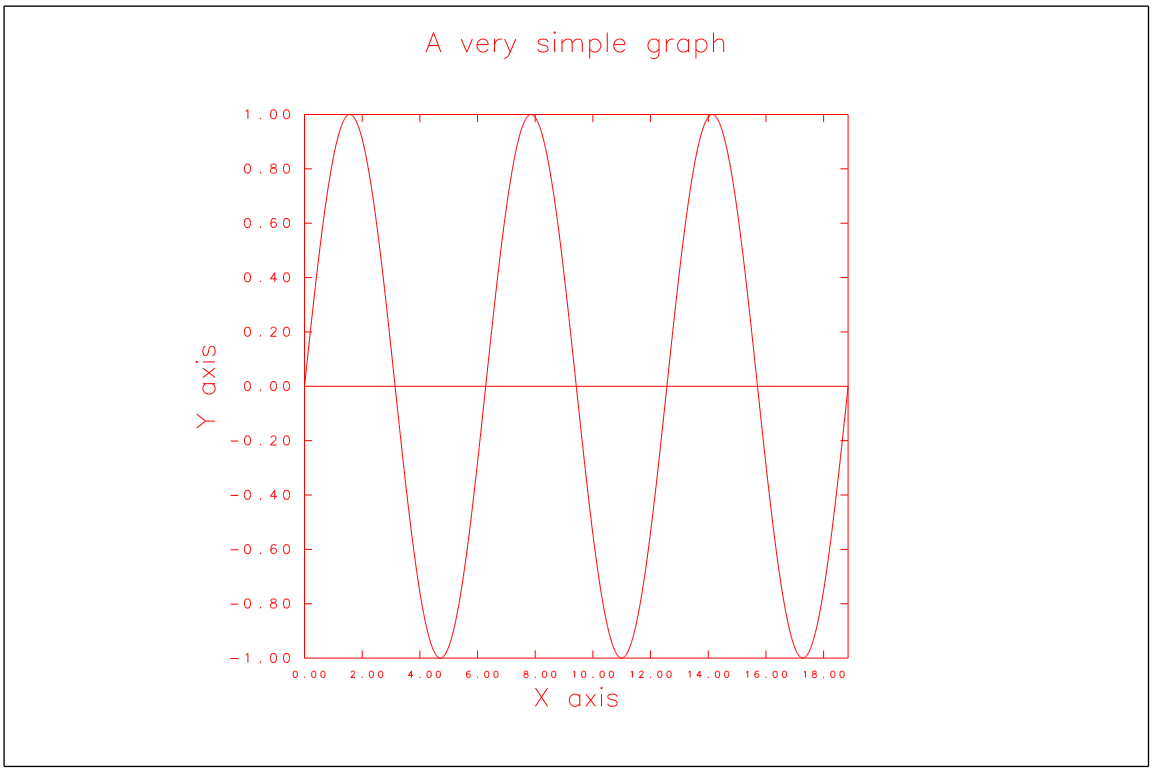


Figure 1: A very simple graph

9 Plotting graphs

9.1 The simplest graph

Let's start by plotting a very simple graph – Figure 1. Normally, data to be plotted exists in disk files, but, in this first example, the data will be internally generated by the `MEMTEST` command, which dynamically allocates data arrays on NOS (static arrays are used on "Unix-like" systems) and fills X,Y values with a sine wave.

The script file (obey file) to generate this contains:

```
# SIMPLEST GRAPH
#
RESET
MEMTEST
TITLE "A*L VERY SIMPLE GRAPH"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
```

Although very minimal, there are a few points worth noting.

- For maximum NOS / "Unix" compatibility, it is best to use only UPPER CASE in obey files. If you use lower case, it will work on both systems, but the results may be different. NOS (in `NORMAL` mode) will internally convert lower case to upper case. So "A very simple graph"

will appear exactly as that on "Unix", but as "A VERY SIMPLE GRAPH" on NOS. Using upper case and **DIMFILM** "string markup" is portable and handles subscripts, superscripts, fractions and so on too. Note that **GPLOT** should *not* be used in `ASCII` mode on NOS! **GPLOT** does not attempt to deal with 6/12 Display Code, but will convert it (mostly!) to upper case. However, internal arrays are not sized for 6/12 representations, so there will be problems (mainly strings being unexpectedly truncated rather than crahses, though).

- The `RESET` command re-establishes a default state. If multiple obey files are used in one **GPLOT** session without using `RESET` at the top of each file, the results will be unpredictable (the state at the end on one obey file will be inherited by the next).
- This script is device independent. It can be used "as is" with any of the supported devices. To generate SVG from it, we can use:

```
/ GPLOT or $ UGPLOT
? dev svg sveg 1200,800
? prefix obey-files
? obey obgraf1
? ex
```

  which will create an SVG file called `sveg001` (`sveg001.svg` on "Unix") using a "resolution" of 1200 by 800 pixels (with SVG, this is really a "size" rather than a resolution, as the vector data can be scaled without losing detail).
- The various annotations associated with a graph (title, axis labels, etc.) should appear before the command which plots the data (`XYLINE` here).
- The ranges of the X and Y axes are automatically determined from the supplied data.
- The graph is drawn in a square area in the center of the output device "canvas". When plotting graphs, it is often better to fill the output canvas.
- Everything is drawn in a single colour (red).

We will address these last two deficiencies in the next example.

9.2 Bounds, panes, devices and coordinate systems in DIMFILM

It is perhaps worth understanding why the graph appears in a centered square area on the device even at this stage.

DIMFILM lets the user define a "world coordinate system" or "user coordinates". These "bounds" are defined by four numbers, `xlo` to `xhi` defining a range of X coordinates, and `ylo` to `yhi` defining a range of Y coordinates. This coordinate system is intended to be "square" in that a unit step in X should have the same length when drawn by the output device as a unit step in Y. The output devices each have their own ranges of valid coordinates. In the case of the interactive devices (**GTerm** and the Tektronix 4014), these ranges are fixed (not strictly true for **GTerm**). The valid range of coordinates can be set by the user for the "file" devices, SVG and EPS, by defining

the “paper size” when the device is opened. The coordinate systems of these devices are also defined to be “square”.

DIMFILM also has the concept of a “pane”, which is initially identical to the “bounds”. The “pane” is a rectangular subregion of the bounds, specified by four numbers in bounds coordinates.

For general purpose drawing, you want to have a “square” coordinate system, so that circles when drawn are circles and not ellipses.

For plotting graphs of data, though, “squareness” is not usually important. **DIMFILM** determines the locations and sizes of the various elements of a graph in terms of fractions of the pane. The graph will be placed inside the pane, squashed and stretched as may be to fill it.

The default initial bounds (and pane) are set to 0 to 1 by 0 to 1, which is a square region. To maintain “squareness” while maximising the use of the display area, the bounds region must cover either the height or the width of the display (or both).

To make the situation clearer, we can draw the limits of the device (which we have set to a 1200 by 800 “pixel” region when we opened the SVG device) and the outline of the bounds (and pane) region by adding these commands:

```
OUTLINE DEVICE
OUTLINE BOUNDS
```

(see Figure 2).

The only way to use the full area of the display is for the bounds to have the same *aspect ratio* as the display. That is:

$$\frac{xhi - xlo}{yhi - ylo} = \frac{width\_display}{height\_display}$$

9.3 NOS, Unix-like systems and compatibility

Before moving on, it may be worth discussing how **GPLOT** tries to maximise compatability between these two very different operating systems.

Perhaps the most obvious thing when glancing at the first example is that the script is entirely UPPER CASE. This need not be so, and the following will work on both “Unix-like” systems and NOS (although after transfer to NOS without using 6/12 coding - which should not be used - it will appear in upper case there).

```
# Simplest graph.
#
reset
memtest
title "A very simple graph"
xlabel "X axis"
ylabel "Y axis"
```

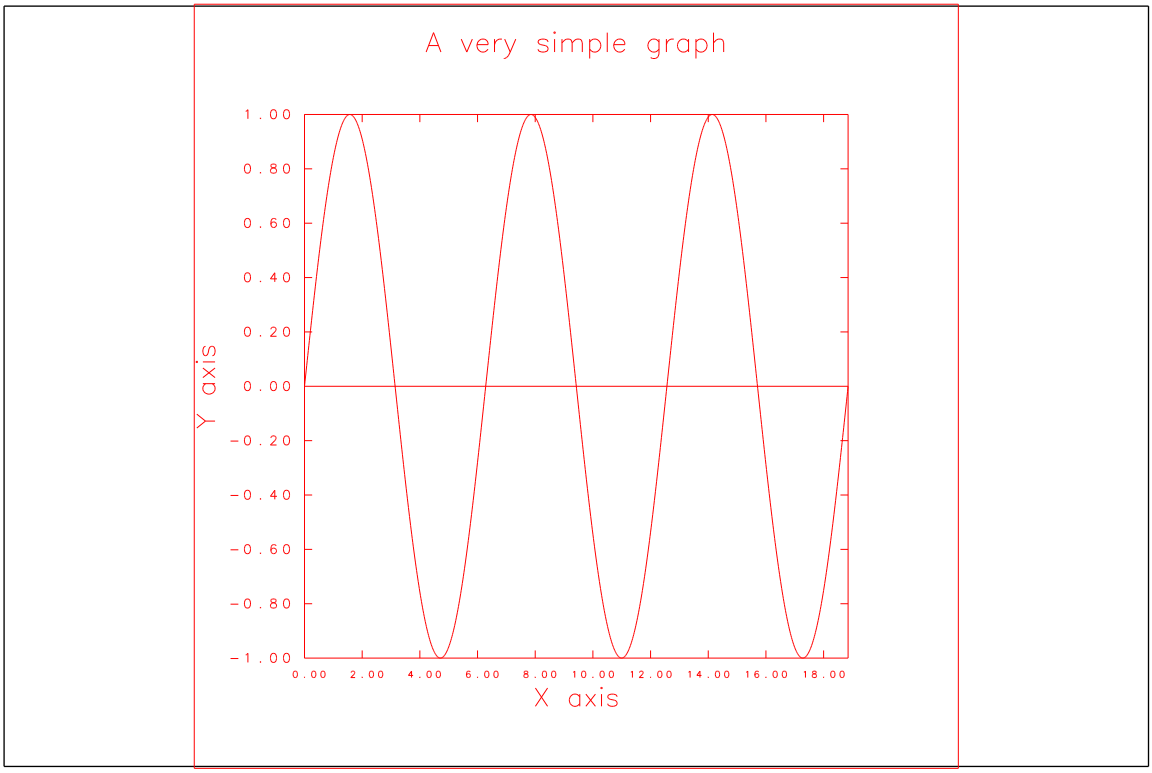


Figure 2: A simple graph showing the bounds

```
xyline
outline dev
```

On “Unix-like” systems, the output will be identical to the first script’s output. However, on NOS, the title and labels will appear in upper case.

Using **DIMFILM’s** “string markup sequences” (the `*L` here) allows lower case to be plotted on NOS.

If you really don’t like using upper case for some reason, you can use lower case and “string markup” and get the same result as using upper case on both systems.

GPLOT also tries to use heuristics to make file name related matters transparent, although there are limits to this considering each NOS account has a flat file system with maximum 7 letter/digit upper case only file names!

For NOS / “Unix” compatibilty, you will have to stick to 7 character file names. However, you can specify a path for “Unix” that is prefixed to each file name using the `PREFIX` command. This prefix is not used for device output files, though, so they will be created in the current working directory.

If you don’t care about NOS compatibility, you can use long file names, so long as the entire path name does not exceed 72 characters.

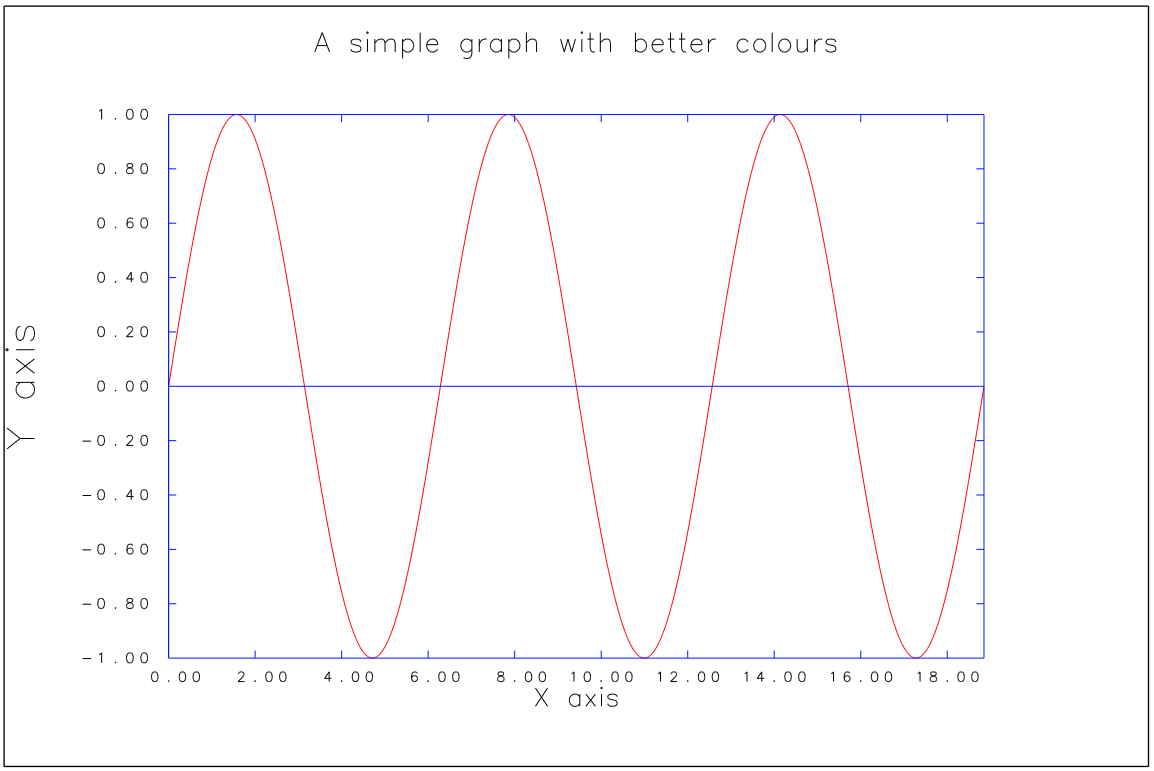


Figure 3: Better colours, filling the bounds

Note that NOS does not have file name extensions. In contrast, "Unix-like" systems pretty much rely on having file name extensions. EPS output files will be given `.eps` extensions on "Unix" and SVG files `.svg`. These should not be added to the file names you type in or put in scripts.

To help identify what files contain on NOS, I use the first two characters as a sort of "extension" – e.g. `OB` for obey files – but this is just a personal convention which NOS knows nothing about.

GPLOT converts all supplied file names (including any `PREFIX`) to lower case on "Unix". You cannot use upper or mixed case file names in **GPLOT** on "Unix". This applies to all input and output files. Very rarely, this will be undesirable, but it almost always simplifies matters. On NOS, all file names are upper case and all lower case is converted to upper case by NOS.

9.4 Using different colours and filling the canvas

Here is the same graph as above, but with different classes of things in different colours and filling the output canvas (Figure 3).

The code for this is:

```
# USING COLOUR/STYLE GROUPS
#
RESET
GRAPHMODE ON
```

```
MEMTEST
CSGROUP GENERAL
COL 1 0 0
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
TITLE "A*L SIMPLE GRAPH WITH BETTER COLOURS"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
OUTLINE DEV
```

To use the full area of the device, the command:

```
GRAPHMODE ON
```

was added. This just sets the bounds to have the same aspect ratio as the display device internally. In this case, to match the aspect ratio of 1200 by 800 (3:2), the bounds will be set to 0 to 1.5 by 0 to 1.

Note that

```
GRAPHMODE OFF
```

resets the bounds to 0 to 1 by 0 to 1 and if bounds are explicitly specified, they just supersede the ones set by GRAPHMODE.

In **GPLOT/DIMFILM** there are 3 "different classes of things", namely:

- Text (TEXT): this refers to all types of "strings".
- Annotation (ANNOT): this refers to the various things that annotate graphs specifically.
- Other (GENERAL): All elements not in the above classes. This includes all lines and points.

All of them are members of the class ALL.

These 3 classes are called "colour/style groups" and the current class is selected with the CSGROUP command. Following this, COLOUR and WIDTH commands set the colour (as normalised RGB) and line width for that CSGROUP only. Perhaps oddly, the line style (solid, dashed, etc.) is *not* set independently for different groups. The current line style applies to all groups.

Note that any command can be abbreviated – so long as the abbreviation remains unique, it will work. (Note that this is *not* true of evaluator operators, as we will see).

9.5 Adding a grid or graticule

In order to better see the values on plotted graphs, it is often useful to have a grid or graticule over which the graphs are plotted (as with traditional graph paper). This is easily added with the GRID command:

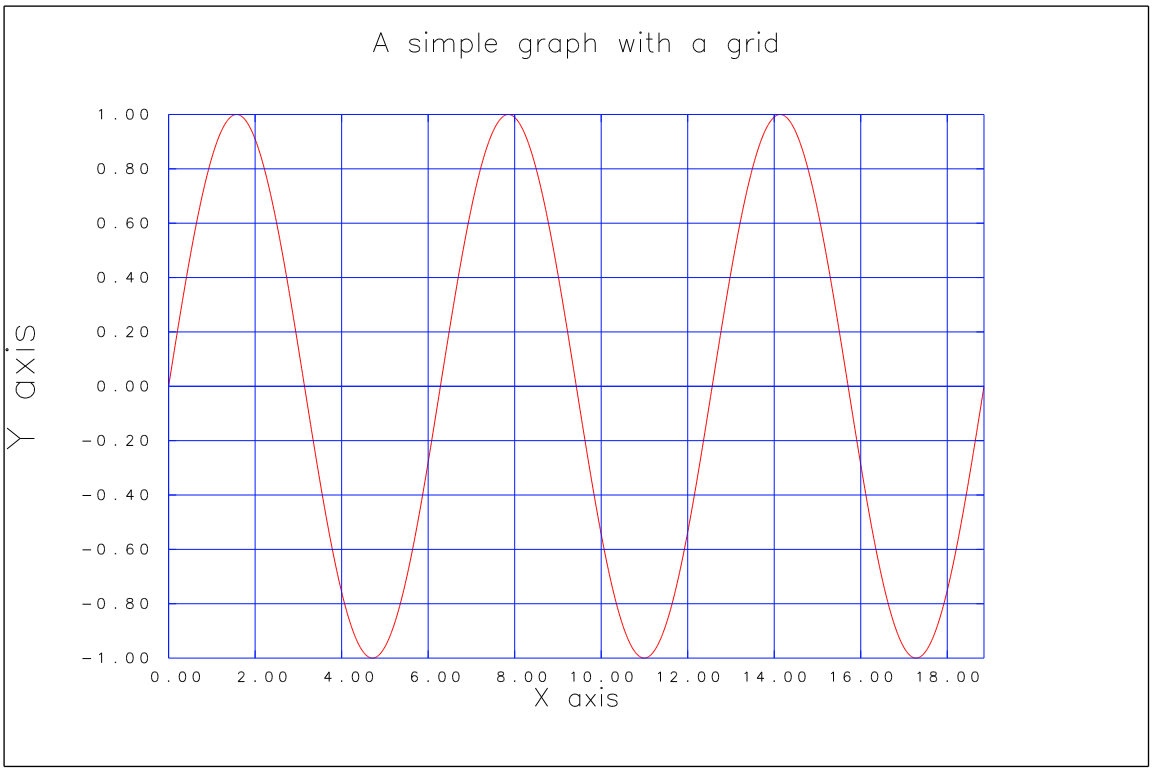


Figure 4: Adding a grid

```
# ADD A GRID
#
RESET
GRAPHMODE ON
MEMTEST
CSGROUP GENERAL
COL 1 0 0
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
TITLE "A*L SIMPLE GRAPH WITH A GRID"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
GRID BOTH
XYLINE
OUTLINE DEV
```

resulting in Figure 4.

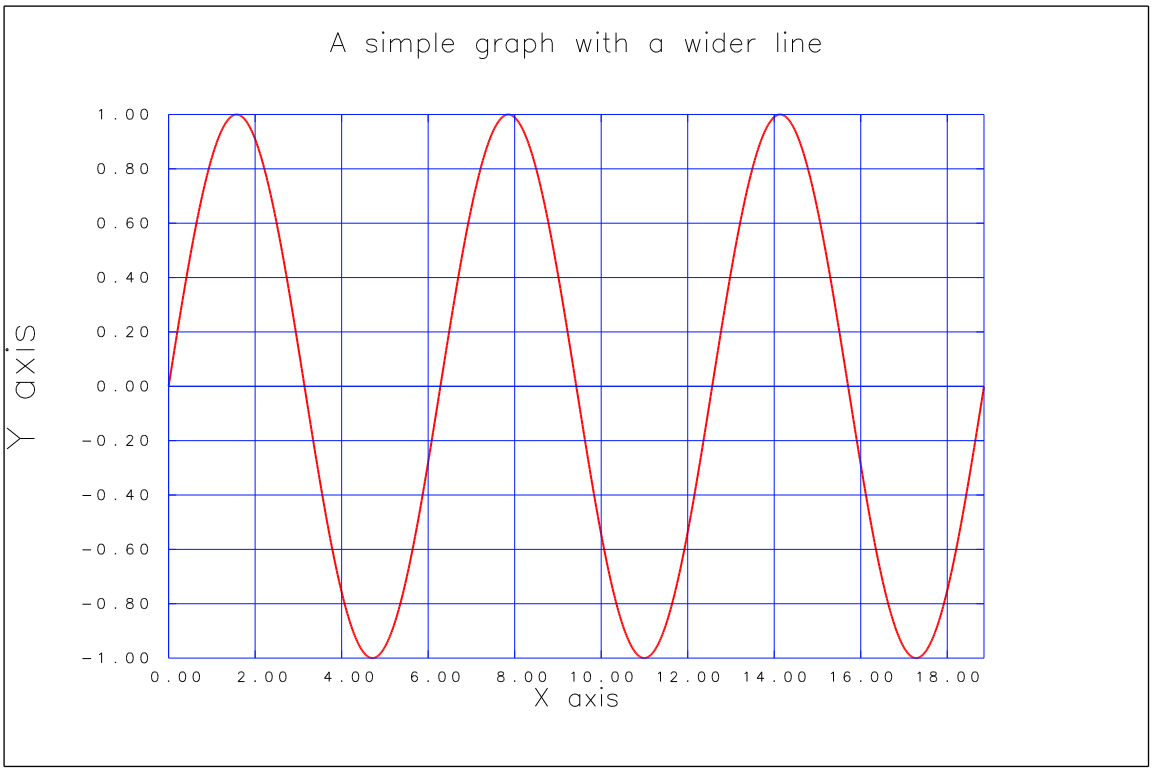


Figure 5: Adjusting line width

9.6 Changing the width of lines

As with colour, the width of lines can easily be set independently for each colour/style group with the `WIDTH` command. As an example, see Figure 5.

```
# GRAPH PLOT WITH WIDER LINE
#
RESET
MEMTEST
GRAPHMODE ON
CSGROUP GENERAL
COL 1 0 0
WIDTH 3
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
TITLE "A*L SIMPLE GRAPH WITH A WIDER LINE"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
GRID BOTH
XYLINE
OUTLINE DEV
```

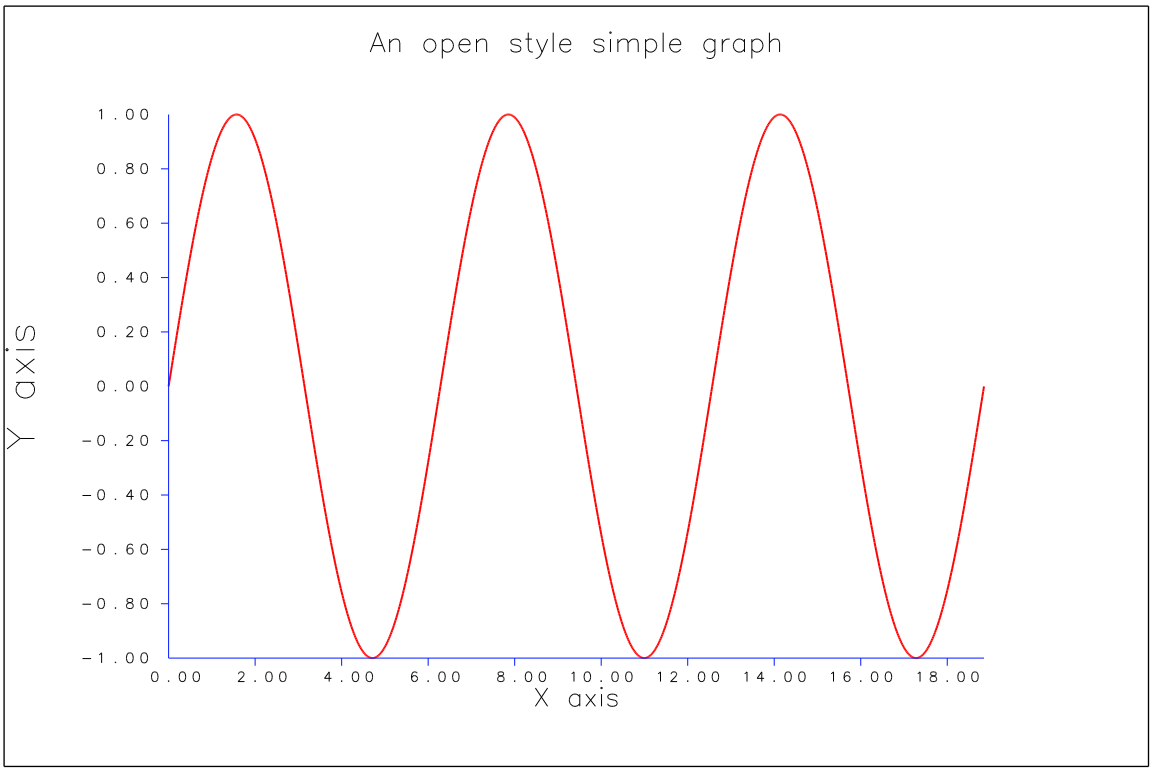


Figure 6: Open style graph

9.7 Different graph styles

The graphs above are all drawn with the default “framed” graph style. An alternative is the “open” style, shown in Figure 6.

```
# OPEN STYLE GRAPH
#
RESET
MEMTEST
GRAPHMODE ON
CSGROUP GENERAL
COL 1 0 0
WIDTH 3
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
GSTYLE OPEN
TITLE "A*LN OPEN STYLE SIMPLE GRAPH"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
OUTLINE DEV
```

Another alternative is the “axes” style, as shown in Figure 7.

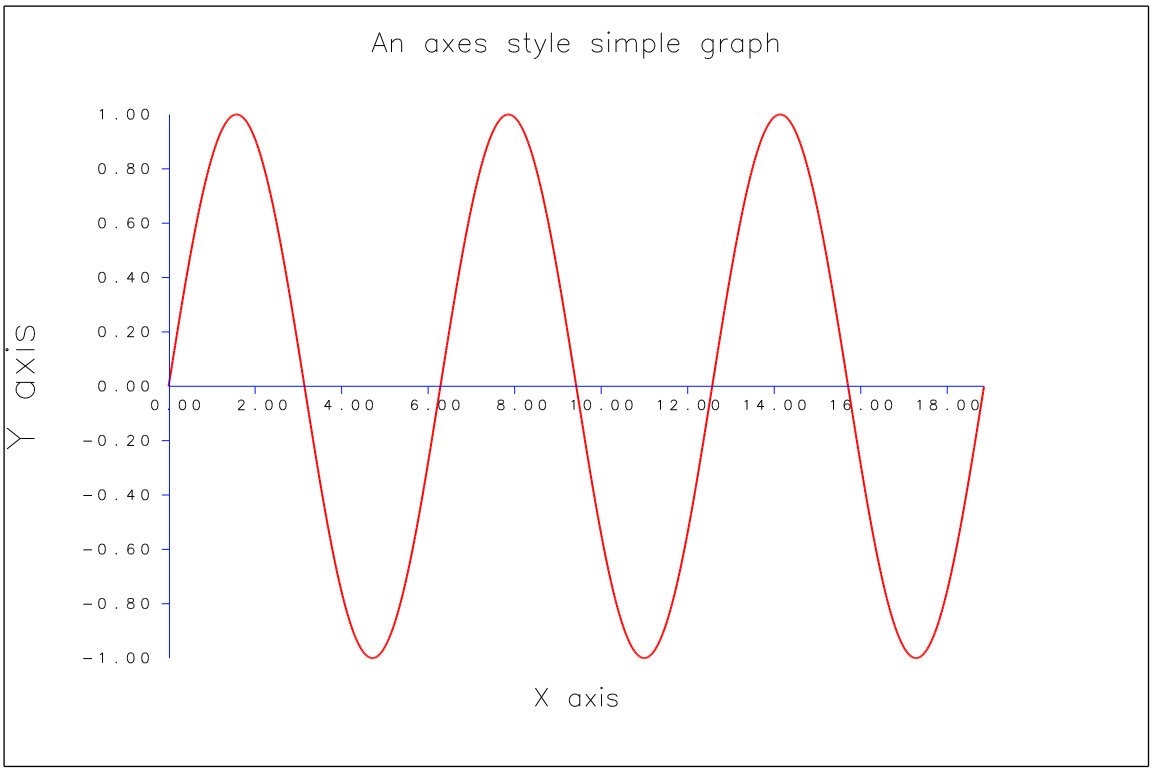


Figure 7: Axes style graph

```
# AXES STYLE GRAPH
#
RESET
MEMTEST
CSGROUP GENERAL
COL 1 0 0
WIDTH 3
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
AXCUT 0.02 0
GSTYLE AXES
TITLE "A*LN AXES STYLE SIMPLE GRAPH"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
OUTLINE DEV
```

To which a grid can be added in the obvious way (Figure 8).

```
# AXES STYLE GRAPH WITH GRID
#
RESET
GRAPHMODE ON
MEMTEST
```

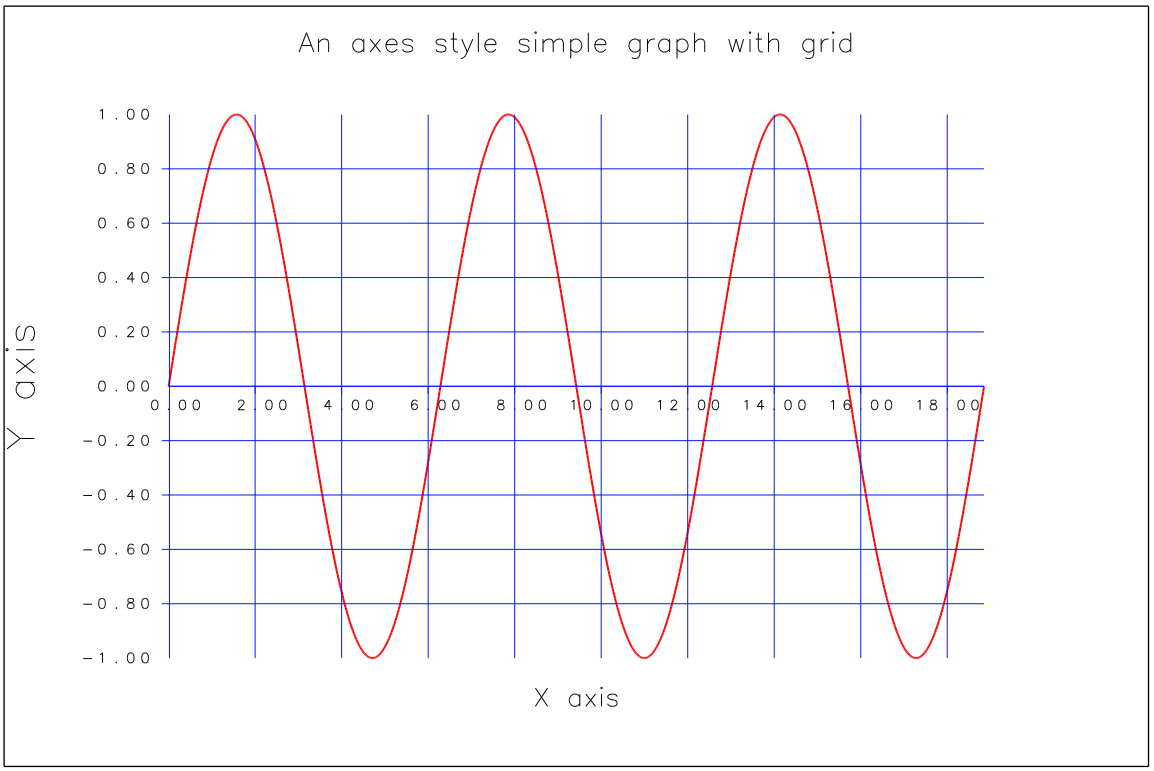


Figure 8: Adding a grid

```
CSGROUP GENERAL
COL 1 0 0
WIDTH 3
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0.1 1
AXCUT 0.02 0
GSTYLE AXES
GRID BOTH
TITLE "A*LN AXES STYLE SIMPLE GRAPH WITH GRID"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
OUTLINE DEV
```

9.8 Multiple graphs on the same axes and “HERE” data

To plot more than one graph on the same axes is quite straightforward. First, note that we haven’t explicitly specified the ranges of the axes in any of the graphs so far. Instead, the system has looked at the data and automatically found reasonable ranges for the axes. This is very often the best approach.

To plot more than one graph and keep automatically determining the axis ranges can be done

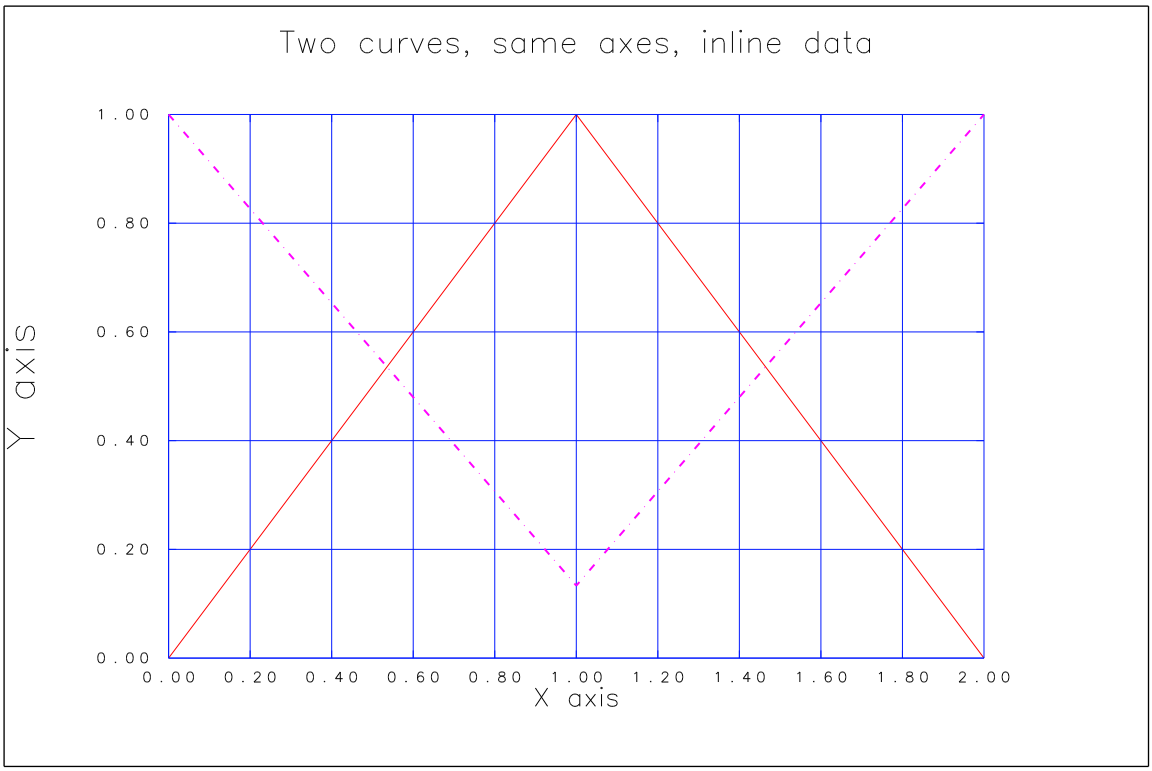


Figure 9: Multiple graphs, same axes

with the with two commands: `XYSAME` and `ANNOT OFF`. The second prevents annotation being drawn multiple times and the first keeps the axis ranges automatically determined from the first data plotted.

GPLOT also lets you specify data "inline" in a script file. Here is a very simple example (Figure 9) and the script that generated it.

```
# 2 CURVE GRAPH WITH INLINE DATA
#
RESET
#
GRAPHMODE ON
GSTYLE BOXED
#
CSGROUP GENERAL
COL 1 0 0
WIDTH 1
STYLE SOLID
#
CSG TEXT
COL 0 0 0
STYLE SOLID
#
CSG ANNOT
COL 0 0.1 1
STYLE SOLID
#
```

```
#--- DEFINE SOME DATA INLINE
READ HERE 1 2
0 0
1 1
2 0
EOF
#--- DRAW THAT WITH AUTO DETERMINED AXIS RANGES
GRID BOTH
TITLE "T*LWO CURVES, SAME AXES, INLINE DATA"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
#
#--- TURN OFF ANNOTATION AND KEEP AUTO DET RANGES FOR NEXT CURVE
ANNOT OFF
XYSAME
#
#--- DEFINE SOME MORE DATA INLINE
READ HERE 1 2
0 1
1 0.1333
2 1
EOF
#
#--- DRAW THAT IN A DIFFERENT COLOUR AND WIDTH.
CSG GEN
COL 1 0 1
WIDTH 3
STYLE DASHDOT
XYLINE
OUTLINE DEVICE
```

Note the READ command. This is described further in the section on plotting data in files. For inline data, the keyword HERE is used with it, and the data follows immediately. That data is terminated by EOF. The numeric arguments to READ are “column numbers” and will almost always be 1 2 for inline data.

9.9 Multiple graphs and keys

GPLOT provides a straightforward (if inflexible) way of adding a key or legend to help identify the meaning of the lines (or points) on graphs.

Figure 10 is an example of three curves with a key and the generating script. The data is again defined by “HERE” data.

```
# 3 CURVE GRAPH WITH INLINE DATA AND KEYS
#
RESET
GRAPHMODE ON
#
GSTYLE BOXED
#
CSGROUP GENERAL
COL 1 0 0
```

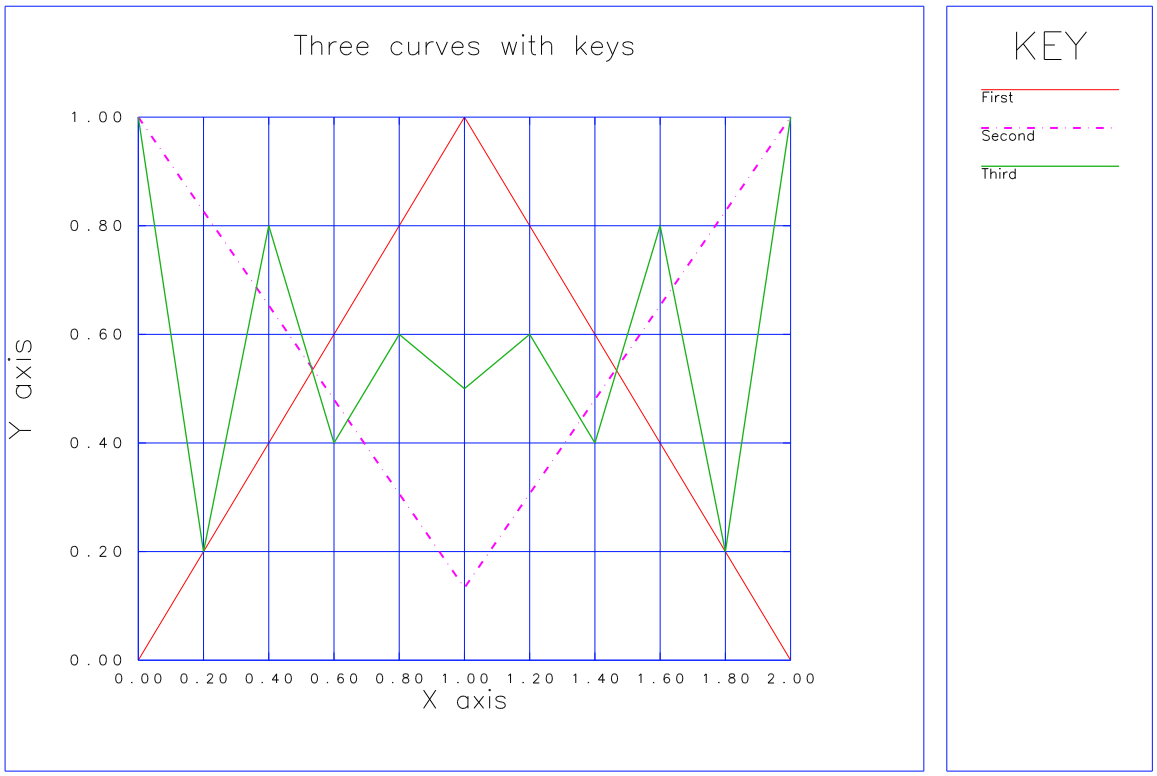


Figure 10: Multiple graphs with a key

```
WIDTH 1
STYLE SOLID
#
CSG TEXT
COL 0 0 0
STYLE SOLID
#
CSG ANNOT
COL 0 0.1 1
STYLE SOLID
#
# --- DEFINE SOME DATA INLINE
READ HERE 1 2
0 0
1 1
2 0
EOF
#
# --- WE WILL ADD KEYS.
USEKEY
#
# --- DRAW THAT WITH AUTO DETERMINED AXIS RANGES
GRID BOTH
TITLE "T*LHREE CURVES WITH KEYS"
XLABEL "X*L AXIS"
YLABEL "Y*L AXIS"
XYLINE
```

```
ADDKEY "F*LIRST"
#
# --- TURN OFF ANNOTATION AND KEEP AUTO DET RANGES FOR NEXT CURVE
ANNOT OFF
XYSAME
#
# --- DEFINE SOME MORE DATA INLINE
READ HERE 1 2
0 1
1 0.1333
2 1
EOF
#
# --- DRAW THAT IN A DIFFERENT COLOUR AND WIDTH.
CSG GEN
COL 1 0 1
WIDTH 3
STYLE DASHDOT
XYLINE
ADDKEY "S*LECOND"
#
# --- DEFINE YET MORE DATA INLINE
READ HERE 1 2
0 1
0.2 0.2
0.4 0.8
0.6 0.4
0.8 0.6
1.0 0.5
1.2 0.6
1.4 0.4
1.6 0.8
1.8 0.2
2 1
EOF
#
# --- DRAW THAT IN A DIFFERENT COLOUR AND WIDTH.
CSG GEN
COL 0.1 0.7 0.1
WIDTH 2
STYLE SOLID
XYLINE
ADDKEY "T*LHIRD"
#
# --- DRAW THE LEGEND.
KEYS
OUTLINE DEV
```

This allows adding keys for up to 20 curves (I find 10 curves on one graph is as much as I can cope with). The maximum length of the key text is also quite limited at a maximum of 15 characters (including any markup sequences).

The layout currently cannot be changed. The key/legend area is always positioned to the right of the graph. This may be better than trying to put the key on top of the graph, as it can be impossible to find a place to put that without obscuring part of the data (at least, I've often found

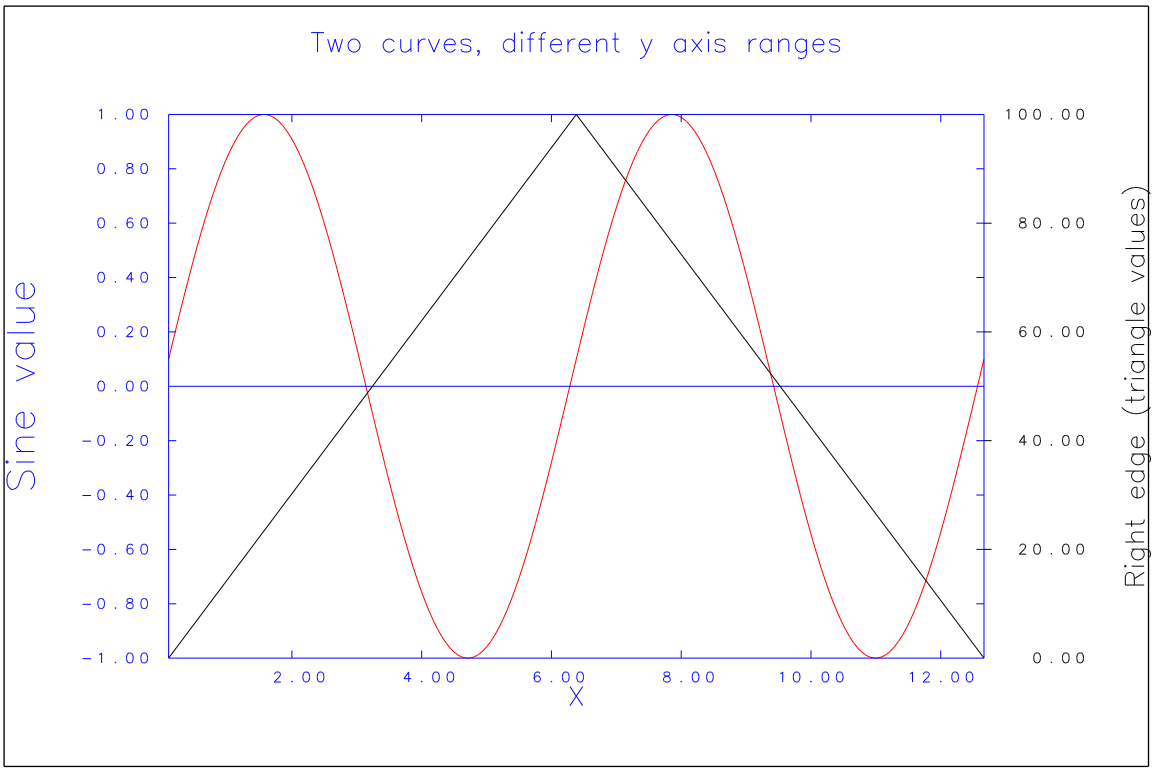


Figure 11: Common X axis, two different Y ranges

that is the case over the decades). That approach could certainly be added, though.

9.10 Multiple curves with two different Y axis ranges

Sometimes it is useful to plot two (or more) curves with the same X axis but, because they have significantly different Y ranges, two distinct Y axes. This can be done by having one Y axis on the left of a graph and another on the right.

A simple example is shown in Figure 11.

```
# TWO CURVES, DIFFERENT AXIS RANGES
#
RESET
GRAPHMODE ON
CSG ANNOT
COL 0 0 1
CSG TEXT
COL 0 0 1
#
# --- GENERATE A SINE WAVE.
EVAL 0.1,TWPI,2,*,0.1,+,201,XLIN,X,SIN
#
# --- PLOT IT WITH AUTO AXIS RANGES.
XLAB "X"
YLABEL "S*LINE VALUE"
```

```
TITLE "T*LWO CURVES, DIFFERENT Y AXIS RANGES"
XYL
ANNOT OFF
#
# --- DEFINE A TRIANGLE WAVE SHAPE.
READ HERE 1 2
0 0
5 100
10 0
EOF
#
# --- PLOT THAT WITH AUTO AXIS, PUT VALUES ON RIGHT
CSG ALL
COLOUR 0 0 0
RIGHTANNOT ON
RYLABEL "R*LIGHT EDGE (TRIANGLE VALUES)"
XYL
OUTLINE DEV
```

This example includes the first use of the evaluator so far (to create the sine wave curve). It is not essential to this example, though.

Note that the axis ranges are still automatically determined (`XYSAME` is not used for obvious reasons). The `ANNOT OFF` command is need though, before the `RIGHTANNOT` command is used.

It is possible to create a key for such a graph too (Figure 12).

```
# TWO CURVES, DIFFERENT AXIS RANGES, KEY
#
RESET
GRAPHMODE ON
CSG ANNOT
COL 0 0 1
CSG TEXT
COL 0 0 1
#
# --- GENERATE A SINE WAVE.
EVAL 0.1,TWPI,2,*,0.1,+,201,XLIN,X,SIN
#
# --- WE WILL USE KEYS.
USEKEY
#
# --- PLOT IT WITH AUTO AXIS RANGES.
XLAB "X"
YLABEL "S*LINE VALUE"
TITLE "T*LWO CURVES, DIFFERENT Y AXIS RANGES"
XYL
ADDKEY "S*LINE"
ANNOT OFF
#
# --- DEFINE A TRIANGLE WAVE SHAPE.
READ HERE 1 2
0 0
5 100
10 0
EOF
```

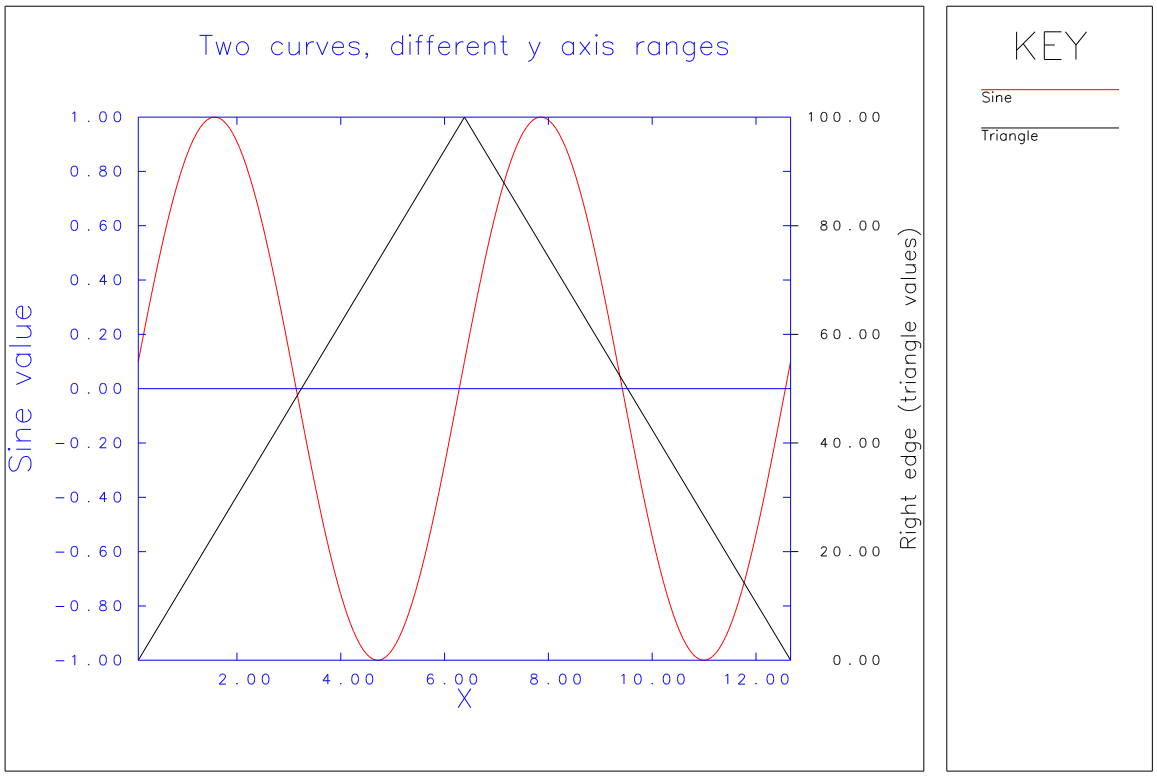


Figure 12: Two different Y ranges and a key

```
#
# --- PLOT THAT WITH AUTO AXIS, PUT VALUES ON RIGHT
CSG ALL
COLOUR 0 0 0
RIGHTANNOT ON
RYLABEL "R*LIGHT EDGE (TRIANGLE VALUES)"
XYL
ADDKEY "T*LRIANGLE"
#
# --- DRAW LEGEND.
KEYS
OUTLINE DEV
```

9.11 Plotting data stored in a disk file

So far, we have plotted data that has been generated by **GPLOT** or stored as “HERE” data in the command stream. Very often, the data to be plotted will be in disk files, though.

GPLOT provides a very simple, albeit basic, way of reading such data and plotting it. Given a file containing the same number of numeric items on each line, with those items separated by spaces or a comma (and perhaps spaces), **GPLOT** can treat the n-th number on each line as an entry in a column of numbers. The columns are numbered 1, 2, 3, etc.

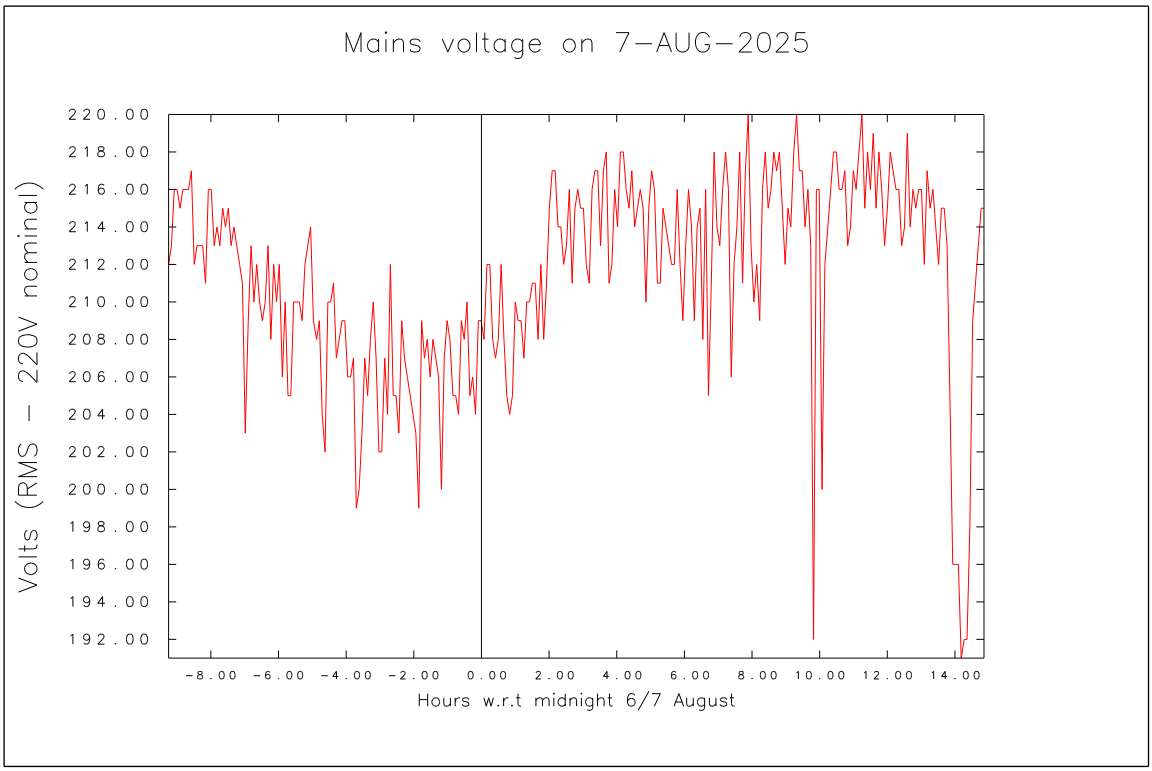


Figure 13: Measured data from a file

Any line that begins with # or C (upper case C, not lower case) will be skipped as a comment. Any entirely blank lines will also be skipped.

The READ command is followed by the name of the data file to be read then the column numbers for the X and Y coordinates.

Figure 13 is an example of some real measured data stored in a file (DAEG1) and plotted.

```
# READ AN X,Y DATA FILE AND PLOT THE CONTENTS
#
RESET
GRAPHMODE ON
#
# --- READ THE DATA FILE, FIRST 2 COLUMNS, SPACE OR COMMA SEP.
GET DAEG1
READ DAEG1 1 2
#
# --- DRAW THE GRAPH
CSG ANNOT
COL 0 0 0
CSG TEXT
COL 0 0 0
TITLE "M*LAINS VOLTAGE ON 7-*UAUG*L-2025"
XLABEL "H*LOURS W.R.T MIDNIGHT 6/7 *UA*LUGUST"
YLABEL "V*LOLTS (*URMS - 220V *LNOMINAL)"
XYLINE
#
```

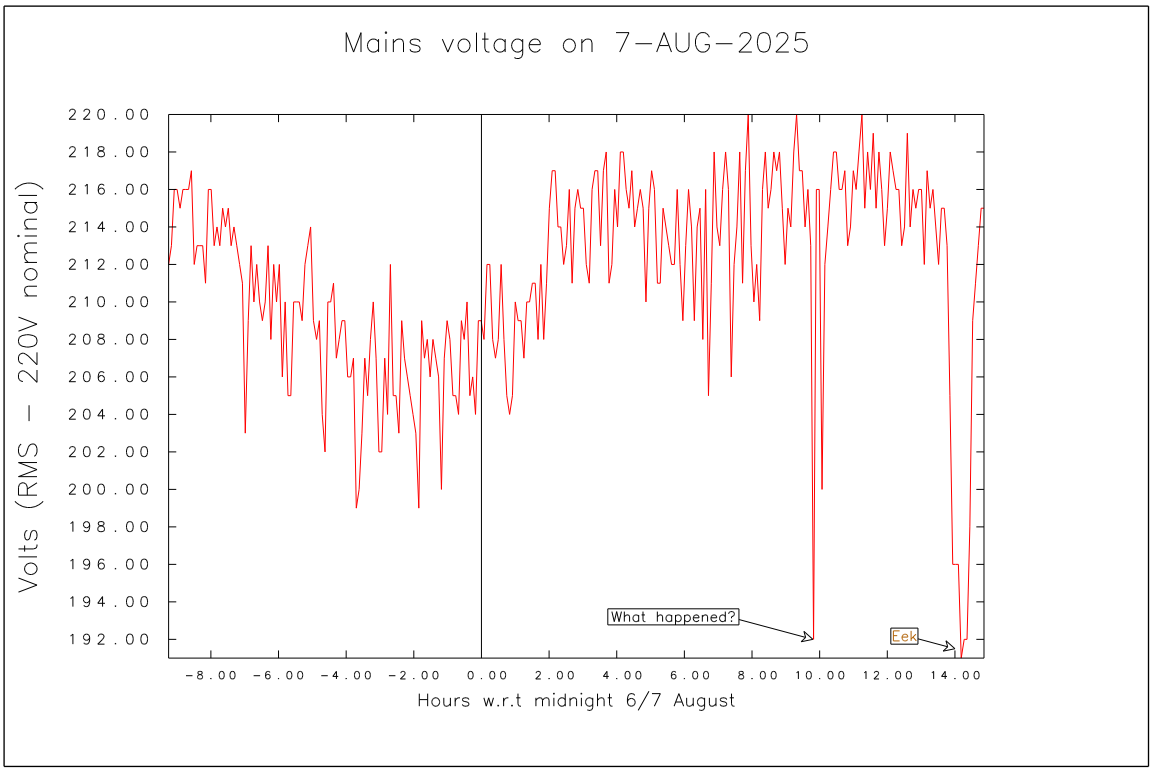


Figure 14: Labelling points on a graph

```
OUTLINE DEV
```

The data file will usually have at least two columns. However, a single column of data can be plotted. That data will become the Y values and the X values will be the line number (starting at 0). Use a 0 column number to do this. For example:

```
READ DAEG1 0 2
```

9.12 Labelling points on a graph

It is sometimes useful to point out something on a graph. Figure 14 is an example of this.

This is quite easy to do, in this case by adding the following commands between `XYLINE` and `OUTLINE`:

```
#
# --- ADD A LABEL OR TWO
CSG ALL
COL 0 0 0
GLABEL 9.8 192.0 0.1 165 "W*LHAT HAPPENED?"
CSG TEXT
COL 0.7 0.384 0.025
GLABEL 14 191.5 0.05 165 "E*LEK"
```

The GLABEL command's first two arguments are the graph coordinates to which the label's arrow should point. The third argument is the length of the arrow in *bounds coordinates* (which will be 0 to 1 by 0 to 1 by default, or 0 to (device aspect ratio) by 0 to 1 if GRAPHMODE ON is in effect). The fourth argument is the angle of the arrow around the location it is pointing at, counter-clockwise from +X in degrees. The final argument is the text for the label, which will be drawn in a box at the blunt end of the arrow.

9.13 Data with uncertainties (error bars) in Y

If your data has an uncertainty measure associated with it, that can be plotted as an error bar on the graph. The meaning of this value is entirely up to you, of course. For common meanings see this Wikipedia article.

Using a concocted data file (DAEG2) which looks like this:

```
0.0,0.0,0.0
0.1,0.01,0.004
0.2,0.04,0.01
0.3,0.09,0.041
0.4,0.16,0.083
0.5,0.25,0.013
0.6,0.36,0.022
0.7,0.49,0.033
0.8,0.64,0.054
0.9,0.81,0.065
1.0,1.0,0.1
```

(note the use of comma separated instead of blank separated values here), the following script will plot a graph with error bar data taken from column 3:

```
# READ AN X,Y,E DATA FILE AND PLOT THE CONTENTS
#
RESET
GRAPHMODE ON
#
# --- READ THE DATA FILE, FIRST 3 COLUMNS. THIS ONE USES COMMAS.
GET DAEG2
READ DAEG2 1 2 3
#
# --- DRAW THE GRAPH
CSG ANNOT
COL 0 0 0
CSG TEXT
COL 0 0 0
CSG GEN
COL 0.8 0.1 0.1
TITLE "L*LOOKS LIKE X*+2$+ WITH SYMMETRICAL ERRORS"
XLABEL "X"
YLABEL "Y"
XYLINE
OUTLINE DEV
```

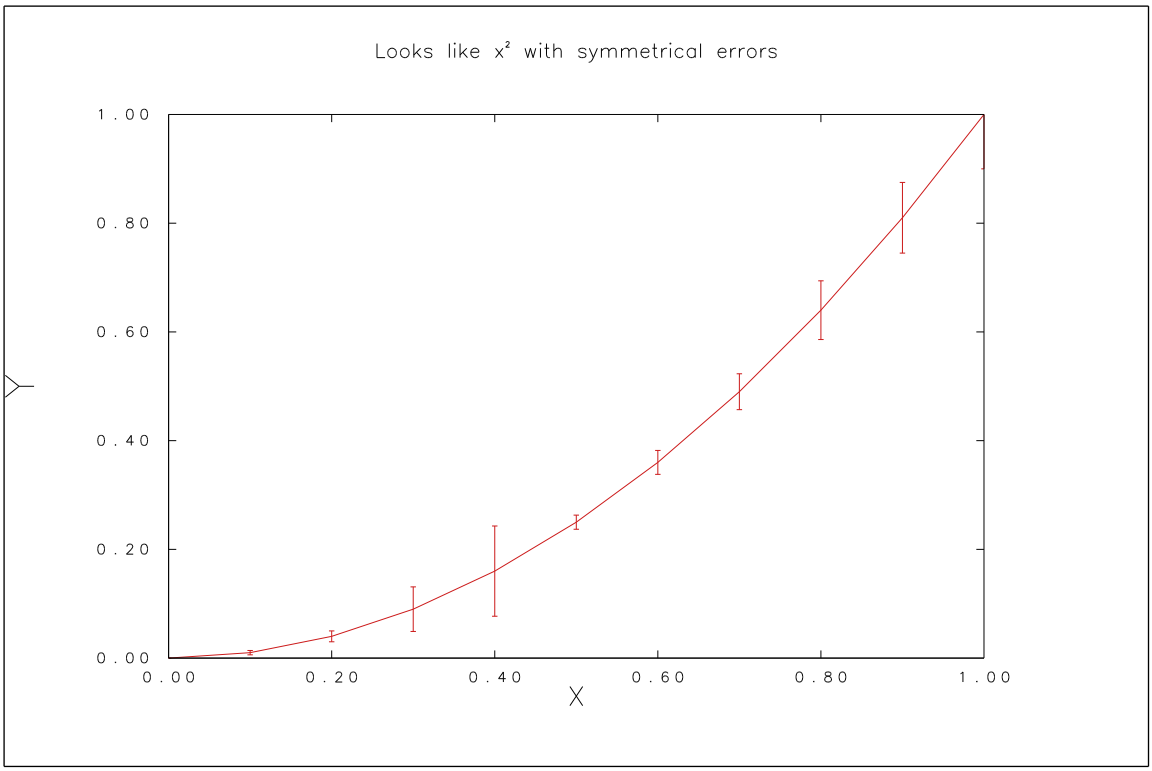


Figure 15: Symmetric Y error bars

This results in Figure 15.

The column given as the third argument to `READ` supplies the delta from the values taken from the column given as the second argument to `READ` at which error bar limits will be drawn above and below the value.

9.14 Data with asymmetric uncertainties in Y

It is possible that your data has potential deviations that are larger in the negative direction than in the positive direction, perhaps because the underlying data distribution is not normal, but skewed, or perhaps the device making measurements has errors that are proportional to the magnitude of the measured values. To visualise this, you need asymmetric error bars in Y.

To illustrate this, we have concocted another data file (`DAEG3`):

```
0.0,0.0,0.0,0.0,0.0
0.1,0.01,0.004,0.006,0.002
0.2,0.04,0.01,0.12,0.033
0.3,0.09,0.041,0.032,0.092
0.4,0.16,0.083,0.043,0.161
0.5,0.25,0.013,0.09,0.251
0.6,0.36,0.022,0.022,0.3617
0.7,0.49,0.033,0.021,0.5
0.8,0.64,0.054,0.066,0.666
```

```
0.9,0.81,0.065,0.032,0.814
1.0,1.0,0.1,0.05,0.997
```

Note this has 5 columns but here we will use the first four.

To plot this with asymmetrical error bars only requires a change to READ. The XYLINE (or XYPOINT) command will plot whatever the last READ has read.

Note the command: ASYMYERRORS YES which explicitly turns on this behaviour. This is not needed, however, as it is the default.

```
# READ AN X,Y,EU,EL DATA FILE AND PLOT THE CONTENTS
#
RESET
GRAPHMODE ON
#
# --- READ THE DATA FILE, FIRST 4 COLUMNS. THIS ONE USES COMMAS.
ASYMYERRORS YES
GET DAEG3
READ DAEG3 1 2 3 4
#
# --- DRAW THE GRAPH
CSG ANNOT
COL 0 0 0
CSG TEXT
COL 0 0 0
CSG GEN
COL 0.8 0.1 0.1
TITLE "L*LOOKS LIKE X*+2$+ WITH ASYMMETRICAL ERRORS"
XLABEL "X"
YLABEL "Y"
XYLINE
OUTLINE DEV
```

This results in Figure 16.

Note that the column specified as the 3rd argument to READ is the delta from the 2nd argument column value to the *upper* limit of the error bar, and the column specified as the 4th argument is the (negative) delta to the *lower* limit of the error bar.

It is easy to plot points on top of this. This might be useful when the points are derived from real measurements and a curve has been fitted to these measurements. As an example of this, we take the point Y coordinates from the 5th column of DAEG3 using this script:

```
# READ AN X,Y,EU,EL DATA FILE AND PLOT THE CONTENTS
#
RESET
GRAPHMODE ON
#
# --- READ THE DATA FILE, FIRST 3 COLUMNS. THIS ONE USES COMMAS.
ASYMYERRORS YES
GET DAEG3
READ DAEG3 1 2 3 4
#
```

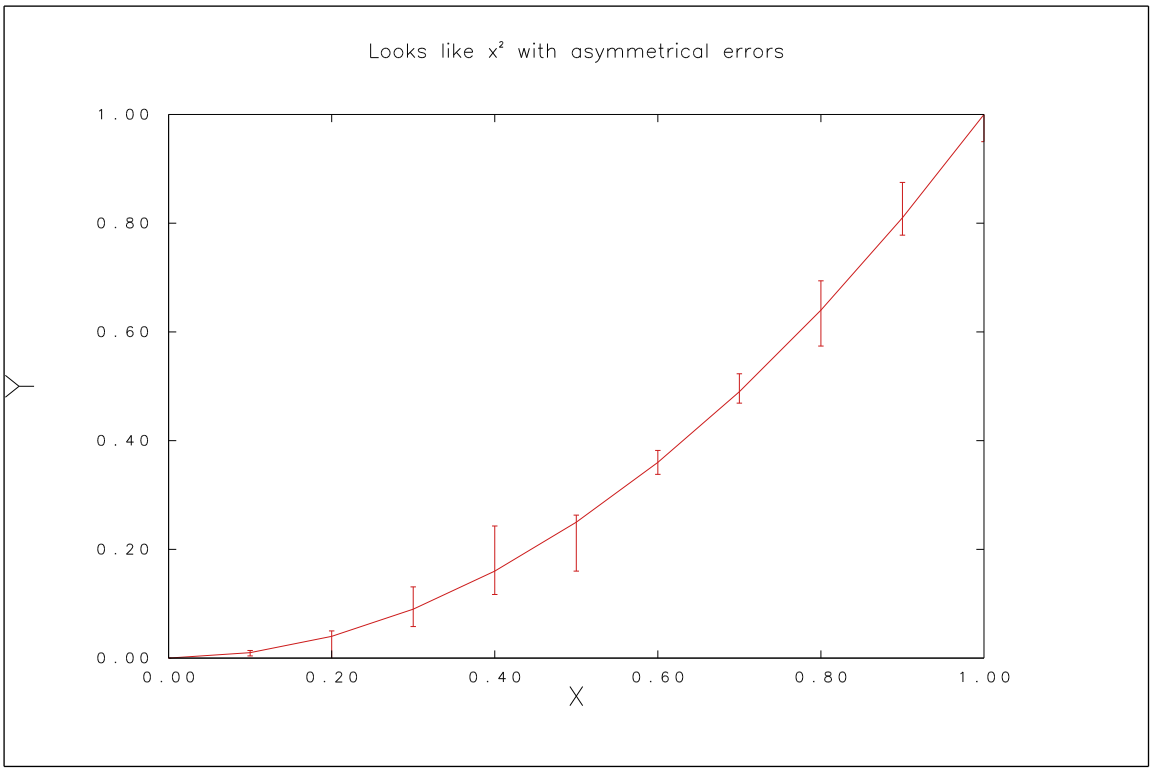


Figure 16:

```
# --- DRAW THE GRAPH
CSG ANNOT
COL 0 0 0
CSG TEXT
COL 0 0 0
CSG GEN
COL 0.8 0.1 0.1
TITLE "*LX*+2$+ WITH ASYMMETRICAL ERRORS AND POINTS"
XLABEL "X"
YLABEL "Y"
XYLINE
#
# --- PLOT POINTS
XYSAME
ANNOT OFF
CSG GEN
COL 0 0 1
SYMHT 0.04
READ DAEG3 1 5
XYPOINT
#
OUTLINE DEV
```

This results in Figure 17.

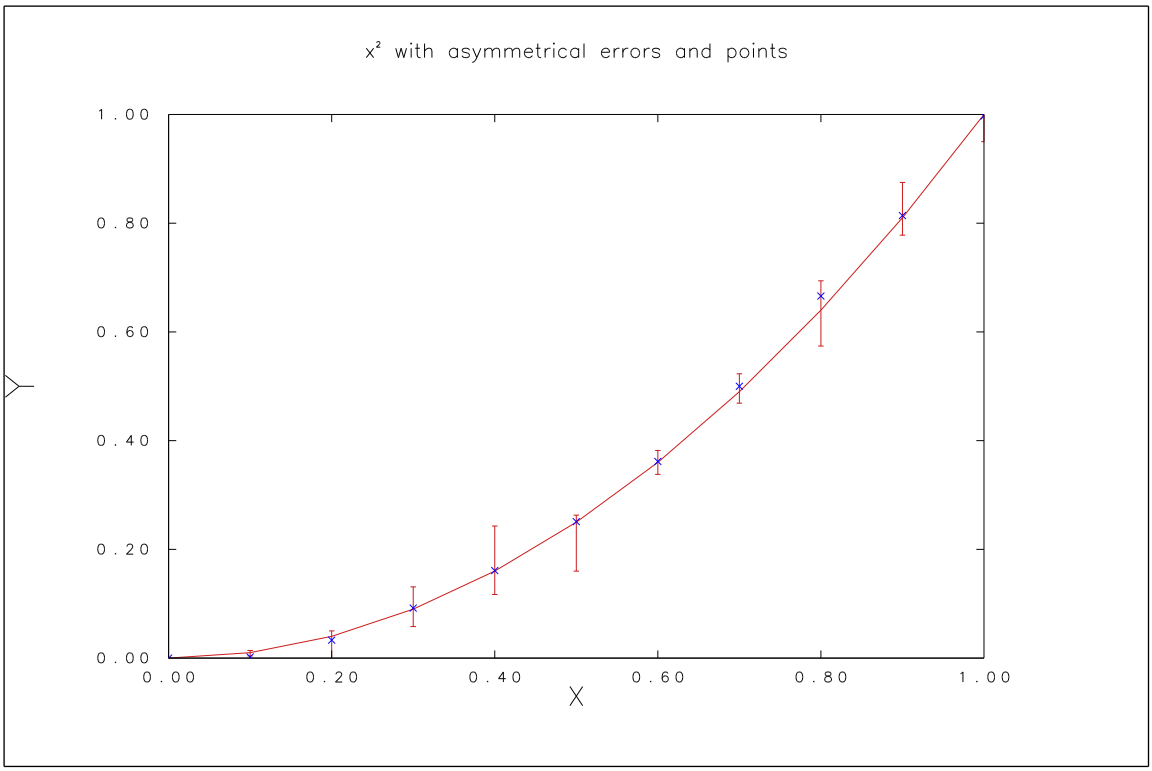


Figure 17: gr18001

9.15 Data with symmetric uncertainties in Y and X

It is also easy to plot data with uncertainty estimates in both axes (see Figure 18). This requires the same 4 data columns as used above, but with the command:

```
ASYMYERRORS NO
```

Note that this *must* appear *before* the `READ` command, as it causes data to be rearranged internally immediately after it is read. In this case, the 3rd column is interpreted as symmetric Y error and the 4th as symmetric X error.

9.16 Interpolating the data points

The supplied data can be interpolated for the purposes of plotting "smooth" curves through, rather than straight lines between, the supplied data. This is generally frowned upon in science, but **GPLOT/DIMFILM** can do it!

In addition to `LINEAR` interpolation (which is usually pointless, as the appearance of the graph does not change), a cubic (3rd order) or quintic (5th order) polynomial can be drawn through the data points. At least 3 points must be supplied for cubic, and 5 for quintic, interpolation. The data must be sorted in X. The number of points to be found between each supplied point must be

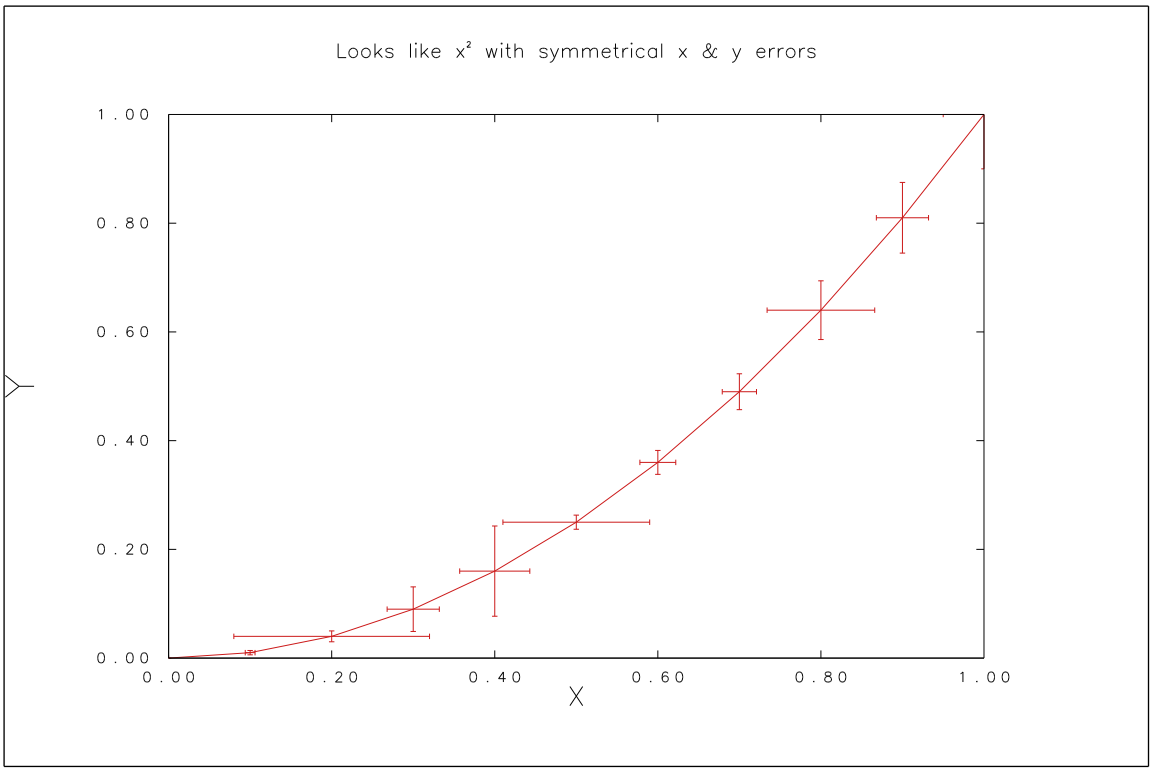


Figure 18: Y and X error bars

specified. If the data is not reasonably “well behaved”, there can be numerical issues.

To demonstrate this capability, Figure 19 shows a very simple example.

This is created by the following script:

```
# INTERPOLATION
#
RESET
GRAPHMODE ON
#
# --- SOME DATA - VERY UNDERSAMPLED SIN(X)
READ HERE 1 2
0.00000   0.00000
0.78540   0.70711
1.57080   1.00000
2.35619   0.70711
3.14159   0.00000
3.92699  -0.70711
4.71239  -1.00000
5.49779  -0.70711
6.28319   0.00000
EOF
#
# --- SETUP THE GRAPH
USEKEY
CSG ANNOT
```

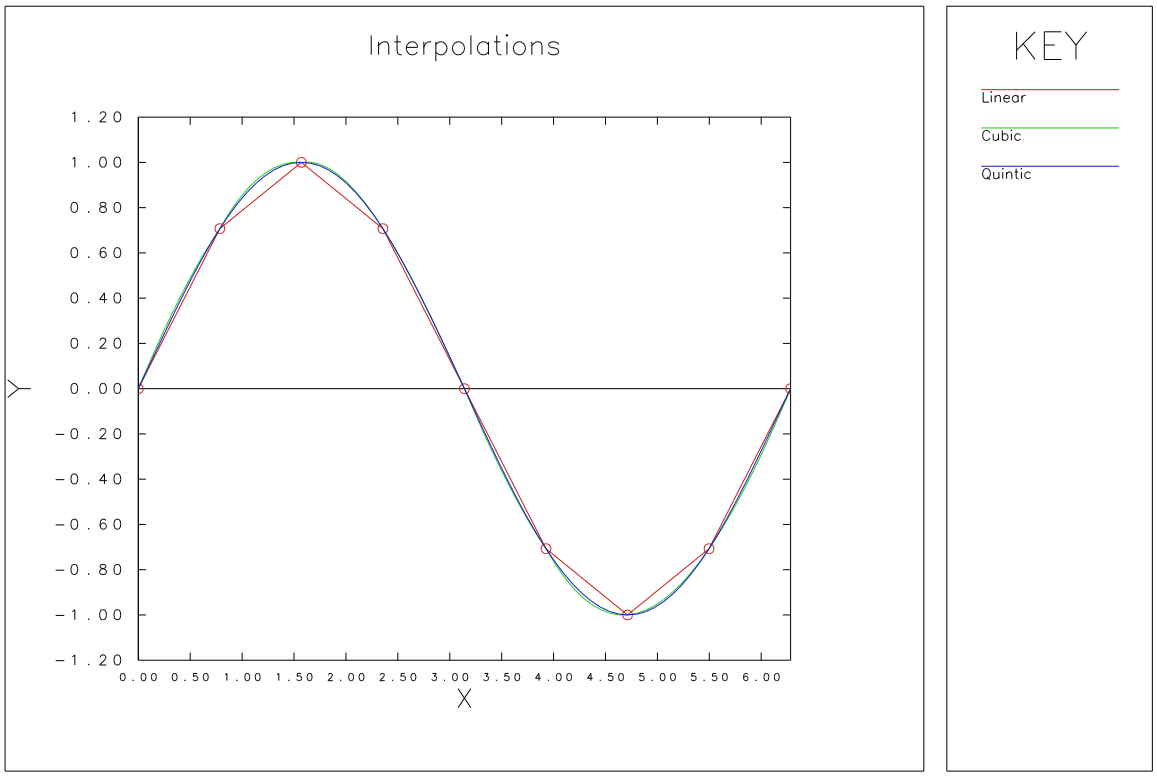


Figure 19: Smooth interpolated curves through data points

```
COL 0 0 0
CSG TEXT
COL 0 0 0
CSG GEN
COL 0.8 0.1 0.1
YRANGE -1.2 1.2
TITLE "I*LNTERPOLATIONS"
XLABEL "X"
YLABEL "Y"
#
# --- LINEAR INTERPOLATION, 9 INTERMEDIATES.
INTERPOLATE LINEAR 9
XYLINE
ADDKEY "L*LINEAR"
#
# --- POINTS BEING INTERPOLATED.
XYSAME
MARKER 20
XYPOINT
#
# --- CUBIC
COL 0.1 0.8 0.1
INTERPOLATE CUBIC 9
XYLINE
ADDKEY "C*LUBIC"
#
# --- QUINTIC
```

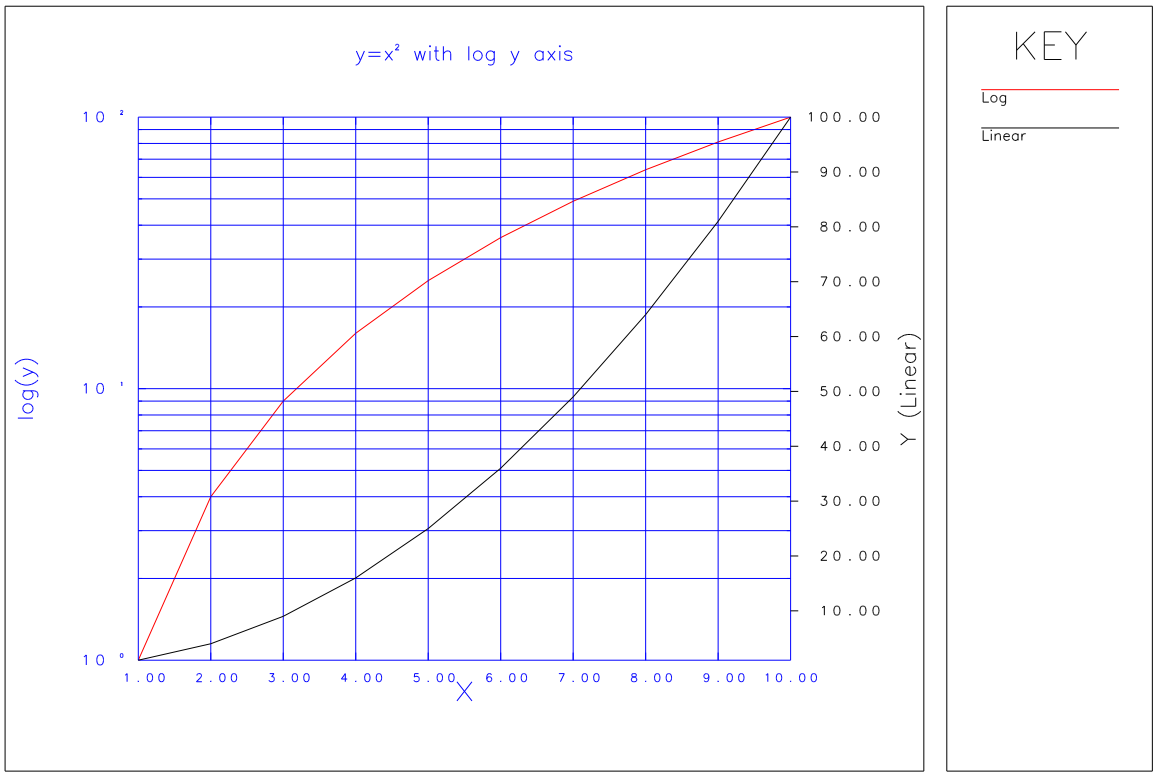


Figure 20: Log Y, linear X, $y = x^2$

```
COL 0.1 0.1 0.8
INTERPOLATE QUINTIC 9
XYLINE
ADDKEY "Q*LUINTIC"
#
# --- FINISH THE GRAPH
KEYS
```

9.17 Log Linear Plots

It is possible to use a logarithmic Y axis and linear X axis. An example of this is shown in Figure 20.

Well, if we were hoping for a straight line with log Y for this function, we are going to be disappointed! Here is the script fot this:

```
# LOG Y AXIS
#
RESET
GRAPHMODE ON
CSG ANNOT
COL 0 0 1
CSG TEXT
COL 0 0 1
```

```
#
# --- SQUARES OF SOME NUMBERS.
READ HERE 1 2
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
EOF
#
# --- WE WILL USE KEYS.
USEKEY
#
# --- PLOT IT WITH LOG Y AXIS
XLAB "*LX"
YLABEL "*LLOG(Y)"
TITLE "*LY=X*+2$+ WITH LOG Y AXIS"
GRID BOTH
YLOG
XYLINE
ADDKEY "L*LOG"
ANNOT OFF
#
# --- PLOT IT WITH LINEAR Y AXIS, VALUES ON RIGHT
CSG ALL
COLOUR 0 0 0
RIGHTANNOT ON
RYLABEL "Y (L*LINEAR)"
YLIN
XYL
ADDKEY "L*LINEAR"
#
# --- DRAW LEGEND.
KEYS
#
# OUTLINE DEV
```

Let's try a power function instead (Figure 21).

Ah hah! This time we are in luck – a straight line using a log-linear plot!

9.18 Log Log Plots

It is also easy to plot data with both log Y and log X axes. This script will do that with the same data as shown in the first log-linear plot (but with the values scaled up to see what happens):

```
# LOG X AND Y AXIS
#
RESET
```

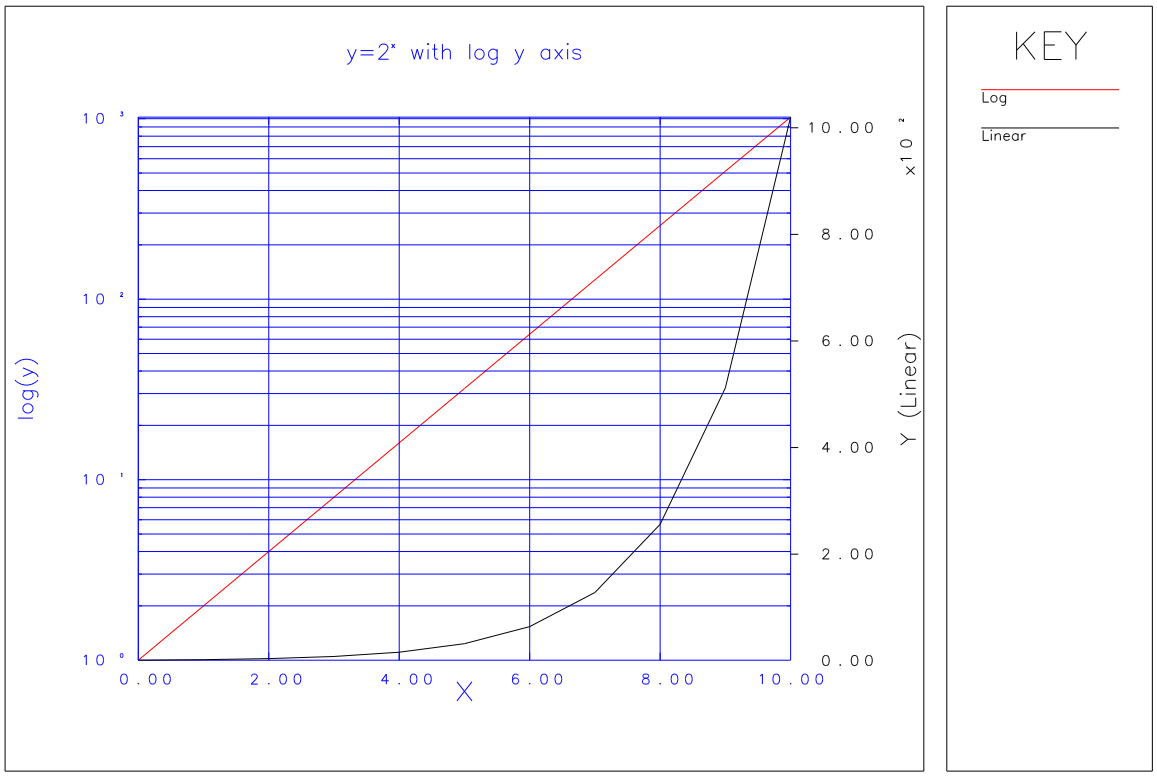


Figure 21: Log Y, linear X, $y = 2^x$

```
GRAPHMODE ON
#
CSG ANNOT
COL 0 0 1
CSG TEXT
COL 0 0 1
#
# --- SQUARES OF SOME NUMBERS.
READ HERE 1 2
1.0E7 1
2.0E7 4
3.0E7 9
4.0E7 16
5.0E7 25
6.0E7 36
7.0E7 49
8.0E7 64
9.0E7 81
1.0E8 100
2.0E8 400
3.0E8 900
4.0E8 1600
5.0E8 2500
6.0E8 3600
7.0E8 4900
8.0E8 6400
9.0E8 8100
```

```
1.0E9 10000
EOF
#
# --- WE WILL USE KEYS.
USEKEY
#
# --- PLOT IT WITH LOG X AND Y AXES
XLAB "*LLOG(X)"
YLABEL "*LLOG(Y)"
TITLE "*LY=(*,X$,10*+7$+$.)*0*+2$+$* WITH LOG X AND Y AXES"
GRID BOTH
XLOG
YLOG
XYLINE
ADDKEY "L*LOG"
#
# --- PLOT IT WITH LINEAR Y AXIS, VALUES ON RIGHT
ANNOT OFF
CSG ALL
COLOUR 0 0 0
RIGHTANNOT ON
RYLABEL "Y (L*LINEAR)"
YLIN
XYL
ADDKEY "L*LINEAR"
#
# --- DRAW LEGEND.
KEYS
```

And we now have a straight line on the log-log plot for this function, as shown in Figure 22.

Note that one of the main reasons people plot experimental data with log-linear or log-log axes is in the hope of seeing something like a straight line when they do so, which gives them a clue to the form of a mathematical model which might "explain" the experimental data (or, at least, approximate it).

9.19 Histograms and multiple plots in one figure

Histograms are another common type of plot and **GPLOT** can draw them in various styles. There are no facilities for binning raw data so that histograms can be drawn, however. Other software must prepare counts of data points that fall into bins of specified ranges. All **GPLOT** deals with are the counts and bin numbers for each count.

As a minimal example, here is a single column of counts that can be represented by a histogram (file DAEG4):

```
100
123
90
20
10
1
200
```

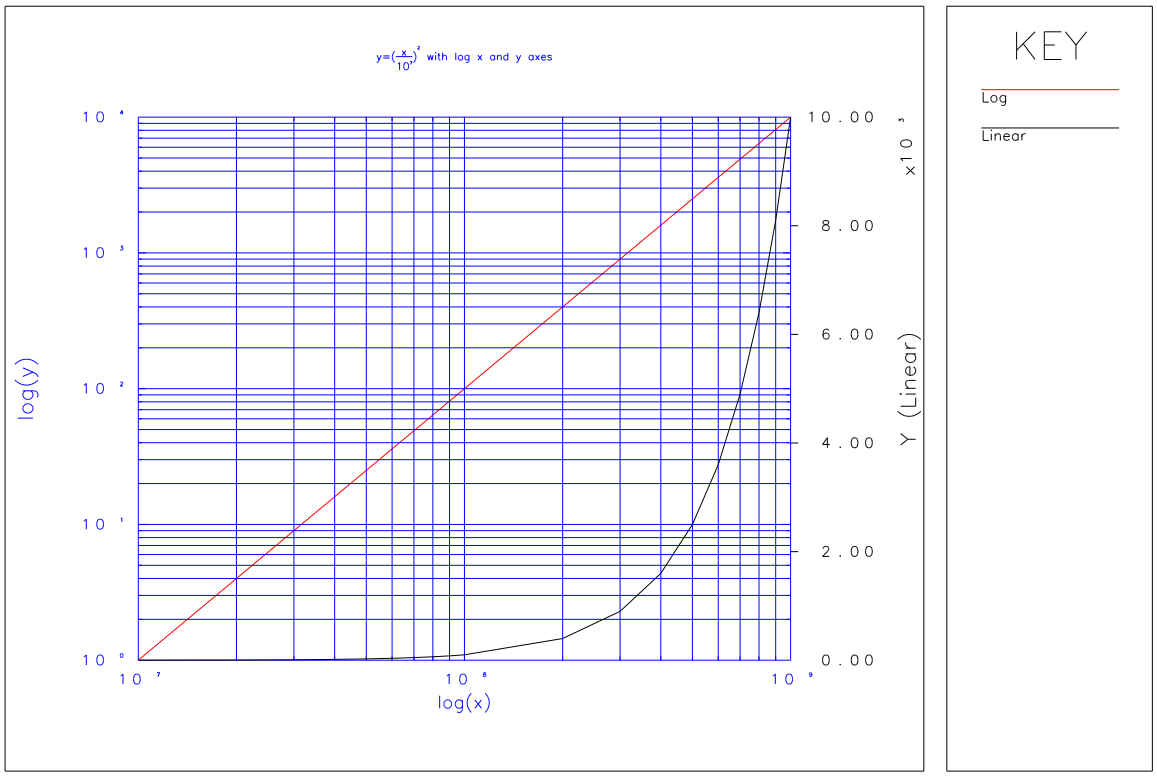


Figure 22: $y = \left(\frac{x}{10^7}\right)^2$

```
233
129
43
77
6
2
9
64
99
12
18
111
87
```

The following script draws histogram plots of this data in four different styles:

```
# READ A COUNTS ONLY DATA FILE AND MAKE HISTOGRAMS.
#
RESET
#
# --- READ THE DATA FILE, FIRST COLUMN ONLY.
GET DAEG4
READ DAEG4 0 1
#
# --- SET COLOURS
CSG ANNOT
```

```
COL 0 0 0
CSG TEXT
COL 0 0 0
CSG GEN
COL 0.8 0.1 0.1
#
# --- SET THE AXIS RANGES AND USE INT LABELS.
XRANGE -1 20
YRANGE 0 240
AXCUT NO
INTVALUES BOTH
#
# --- SET AXIS LABELS.
XLABEL "C*LLASS NUMBER"
YLABEL "C*LOUNT"
#
# --- ABUT STYLE IN BOTTOM LEFT.
PANE 0 0.5 0 0.5
HISTSTYLE ABUT
TITLE "C*LOUNTS HISTOGRAM (ABUTTING BARS)"
XYHIST
OUTLINE PANE
#
# --- SHADED ABUT STYLE IN BOTTOM RIGHT.
PANE 0.5 1 0 0.5
HISTSTYLE ABUT+SHADE
TITLE "C*LOUNTS HISTOGRAM (ABUTTING SHADED BARS)"
XYHIST
OUTLINE PANE
#
# --- SPECIFIED WIDTH BARS IN TOP LEFT.
PANE 0 0.5 0.5 1
HISTSTYLE WIDE 0.25
TITLE "C*LOUNTS HISTOGRAM (SPECIFIED WIDTH BARS)"
XYHIST
OUTLINE PANE
#
# --- THIN LINES IN TOP RIGHT.
PANE 0.5 1 0.5 1
HISTSTYLE LINES
TITLE "C*LOUNTS HISTOGRAM (IMPULSES)"
XYHIST
OUTLINE PANE
#
OUTLINE DEV
```

The result is shown in Figure 23.

In addition to showing how to draw histogram plots, this example shows a number of other new features.

- We explicitly specify X and Y axis ranges rather than letting the software choose. This is done with the `XRANGE` and `YRANGE` commands.
- We prevent lines being drawn for the axes – vertical and horizontal lines that pass through (0,0) (or another point specified with `AXCUT`) – by using: `AXCUT NO`

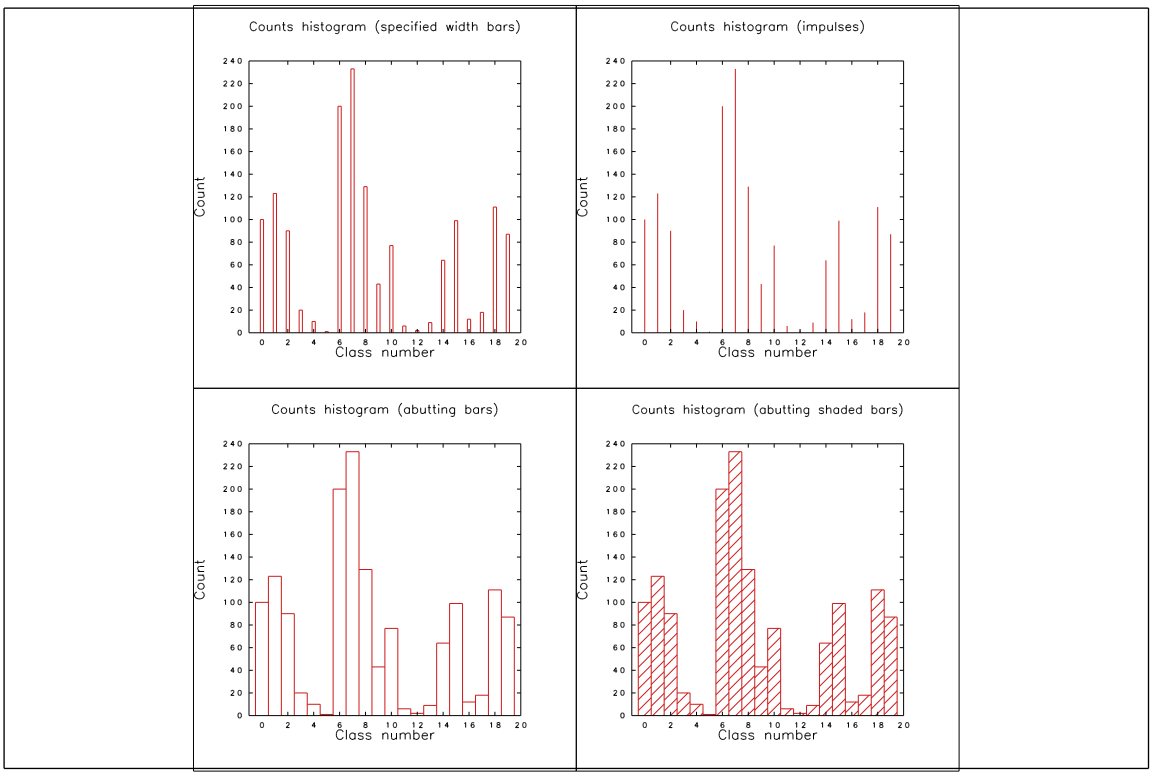


Figure 23: Histograms and sub-figures using `PANE` explicitly

- We try to use integers for the axis values instead of floating point numbers using `INTVALUES BOTH`. This option can be requested for X, Y, both or neither axis. It may not always be honoured, as it will be impossible for many axis ranges. But in this case, it is a reasonable request and it works.

The most obvious new feature, though, is that the four plots showing the different histogram drawing styles appear in a single "figure".

This can be done easily with **DIMFILM**, as graphs are plotted inside any specified "pane" (see above), so all you have to do is use `PANE` appropriately.

Actually, though, that needs some (trivial) manual calculation, but it may also need to account for the device aspect ratio if the output canvas is to be filled. To further simplify creating multiple plots in a single "figure", **GPLOT** has the `SUBFIGGRID` command (which can be abbreviated, fortunately). This lets you specify a rectangular arrangement of "sub-figures" and the current sub-figure to be drawn (before any plotting commands) and calculates an appropriate pane for you, accounting for the device's aspect ratio (if `GRAPHMODE ON` has been used). It also lets you "shrink" the figure, so there is some white space between it and its neighbours.

An example of using this is shown in Figure 24.

The script for that is very similar to the one above, but with these changes:

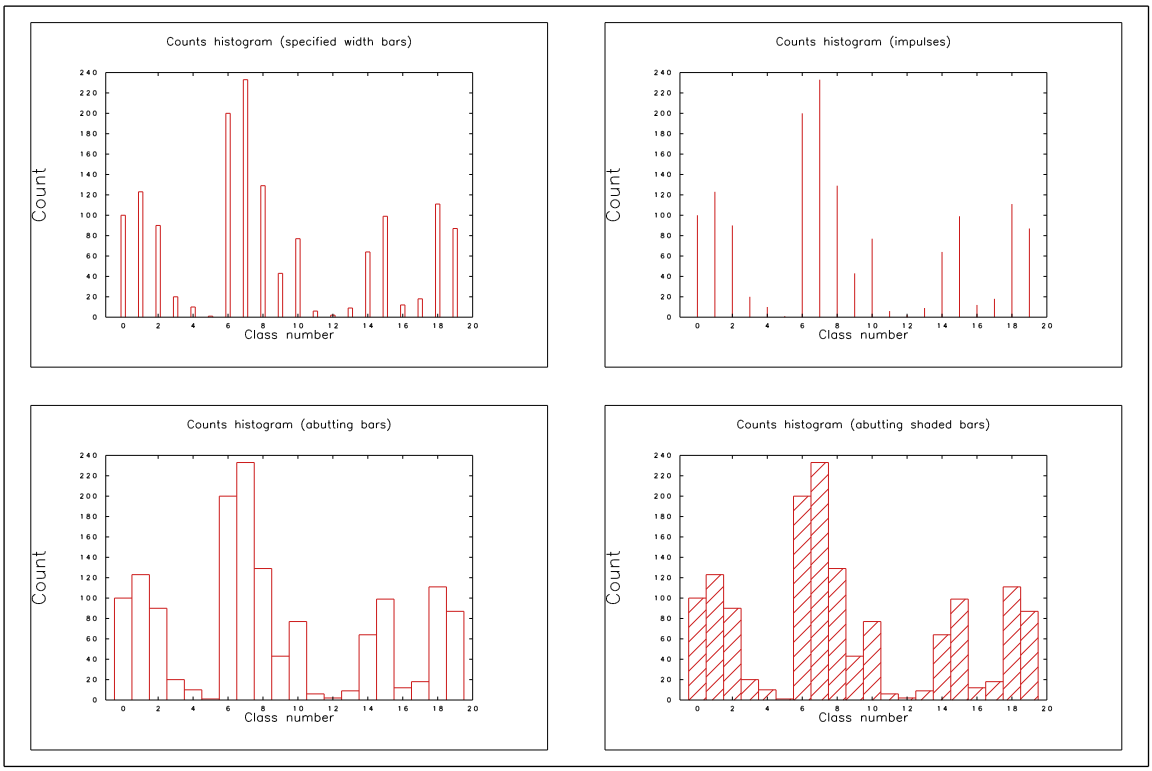


Figure 24: Histograms and sub-figures using SUBFIGGRID

```
# READ A COUNTS ONLY DATA FILE AND MAKE HISTOGRAMS.
#
RESET
GRAPHMODE ON
...
#
# --- ABUT STYLE IN BOTTOM LEFT.
SUBFIG 2 2 1 1 0.95
HISTSTYLE ABUT
...
#
# --- SHADED ABUT STYLE IN BOTTOM RIGHT.
SUBFIG 2 2 2 1 0.95
HISTSTYLE ABUT+SHADE
...
#
# --- SPECIFIED WIDTH BARS IN TOP LEFT.
SUBFIG 2 2 1 2 0.95
HISTSTYLE WIDE 0.25
...
#
# --- THIN LINES IN TOP RIGHT.
SUBFIG 2 2 2 2 0.95
HISTSTYLE LINES
...
#
```

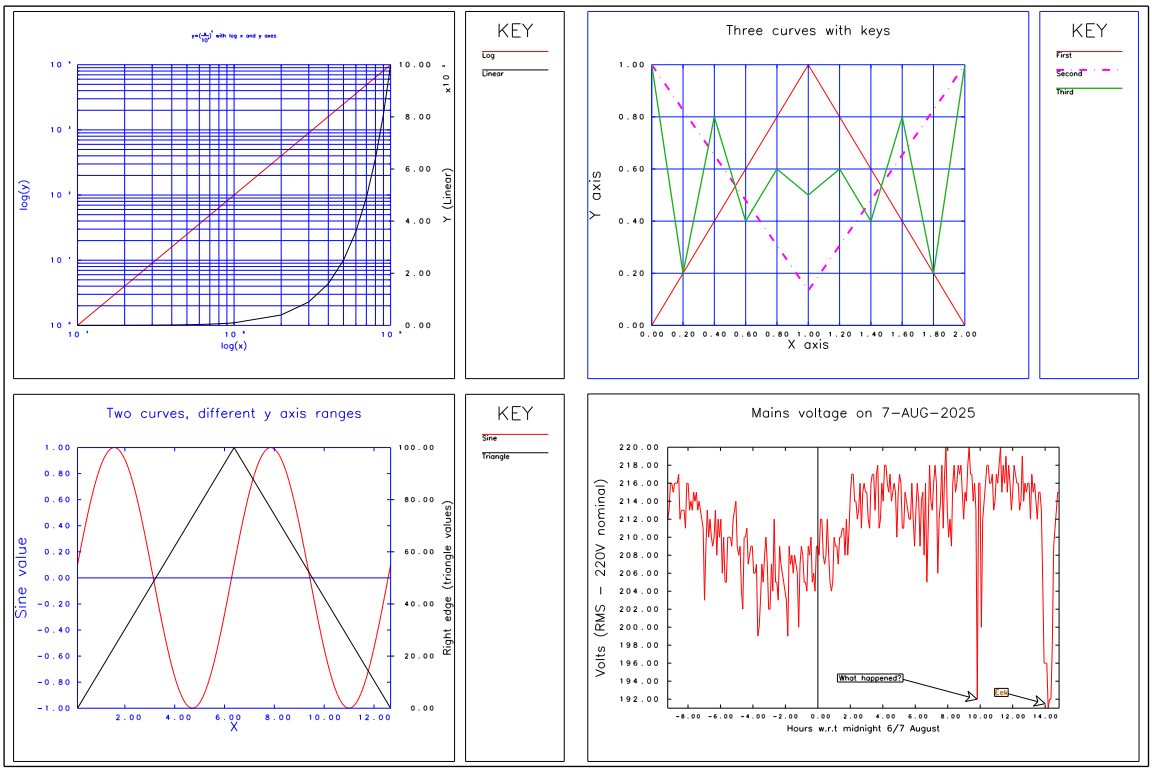


Figure 25: Different plots, some with keys, in one figure

```
OUTLINE DEV
```

9.20 Multiple plots with keys in one figure.

The “sub-figures” feature also works with graphs with keys (some juggling of panes is involved internally). An example of this is Figure 25.

This redraws four of the example plots shown above. Some minor changes to the scripts that drew them were made (any `RESET`, `GRAPHMODE` or other commands that changed the bounds or pane were removed). These modified versions are stored in the script files `OBGF28A`, `OBGF28B`, `OBGF28C` and `OBGF28D`. These are then “called” from a “master” script file `OBGF28M` (obey files can be nested up to 5 levels deep):

```
# FOUR DISPARATE PLOTS, ONE FIGURE.
#
#============================
RESET
GRAPHMODE ON
# --- BOTTOM LEFT OF 4
SUBFIG 2 2 1 1 0.98
#
OBEY OBGF28A
#============================
```

```
RESET
GRAPHMODE ON
# --- TOP RIGHT OF 4
SUBFIG 2 2 2 2 0.98
#
OBEY OBGF28B
#
# ========================
RESET
GRAPHMODE ON
# --- TOP LEFT OF 4
SUBFIG 2 2 1 2 0.98
#
OBEY OBGF28C
# ========================
# --- BOTTOM RIGHT OF 4
RESET
GRAPHMODE ON
SUBFIG 2 2 2 1 0.98
#
OBEY OBGF28D
OUTLINE PANE
#
OUTLINE DEV
```

(Note that files will be stored on "Unix-like" systems with lower case file names but may be referred to in upper case in **GPLOT** – all file names are converted internally to lower case by **GPLOT** on "Unix" before being accessed. These names could appear in lower case – as could all commands – and it would work)

Please example the "called" scripts for more information.

When generating a multi-plot "figure", you must keep the grid size constant in the `SUBFIG` command (e.g. `SUBFIG 2 2`) for every sub-figure. This is probably obvious, but nothing can check for this.

Note the repetition of `RESET` and `GRAPHMODE ON` for each sub-figure. Without this, the various colours and styles set in one sub-figure script would be inherited by the next. These must occur *before* the `SUBFIG` command, so they cannot be left in the script files.

10 Beyond graph plotting

10.1 Using the RPN evaluator to plot simple functions

GPLOT includes a feature for calculating function values rather than reading them from files. This is the Reverse Polish Notation (RPN) evaluator, which provides features very similar to those you would find on classic HP calculators.

One difference is that the RPN evaluator often operates on *arrays* rather than scalar values, in a way that might be familiar to people who have used NumPy or an APL-like language, perhaps.

But with RPN thrown in for extra confusion!

The evaluator has quite a few features beyond those needed for simple function plotting, and its main limitation is perhaps that the “program” length is limited by the 80 character input line limit (beyond which all input is truncated).

A complete description of the available operators can be found in the “cheat sheet” below (which uses a notation familar to FORTH users) and in the PDF format **GPLOT** manual. Here, we just give some examples as an introduction.

Admittedly, the RPN evaluator may not be all that easy to use (the `DUMP` operator is helpful when debugging), but it opens up a lot of possibilities.

When dealing with data from files, **GPLOT** uses between 2 and 4 arrays to store the data to be plotted. Two are used to store X and Y coordinates. If symmetric error bars are to be plotted a third array is used to store that, and asymmetric Y error bars or Y and X error bars use a fourth array.

The RPN evaluator treats these four arrays as the first four levels of a stack. If `READ` is used, their contents will be overwritten by the read data and the array length seen by the evaluator will be the number of points read.

To provide “room” for any but the simplest calculations, another 4 stack levels (arrays) are provided “above” to four used for data from `READ`. (This can be configured with the `NSTACK` command).

It is possible to use the first two X and Y levels of the stack as inputs to the evaluator. More often, though, the contents of the X array are set to cover some range of values by the `ERANGE` command or certain evaluator operators, and the evaluator sets the Y values in the Y array. This is how it can fairly straightforwardly plot mathematical functions.

The evaluator “cheat sheet” below tries to show this stack of arrays diagrammatically.

As a first, very simple, example, let’s compute “y=x^2”. In previous examples the data has been precomputed and read from a file (often a “HERE” file). This script generates the y values for a defined range of x values and plots the result (left hand subfigure, Figure 26).

```
RESET
GRAPHMODE ON
#
CSG ANNOT
COL 0 0 0
CSG TEXT
COL 0 0 0
#
ERANGE 1 1 100 100
EVAL X,2,**
#
XLABEL "X"
YLABEL "Y"
TITLE "X*+2"
GRID BOTH
XYLINE
```

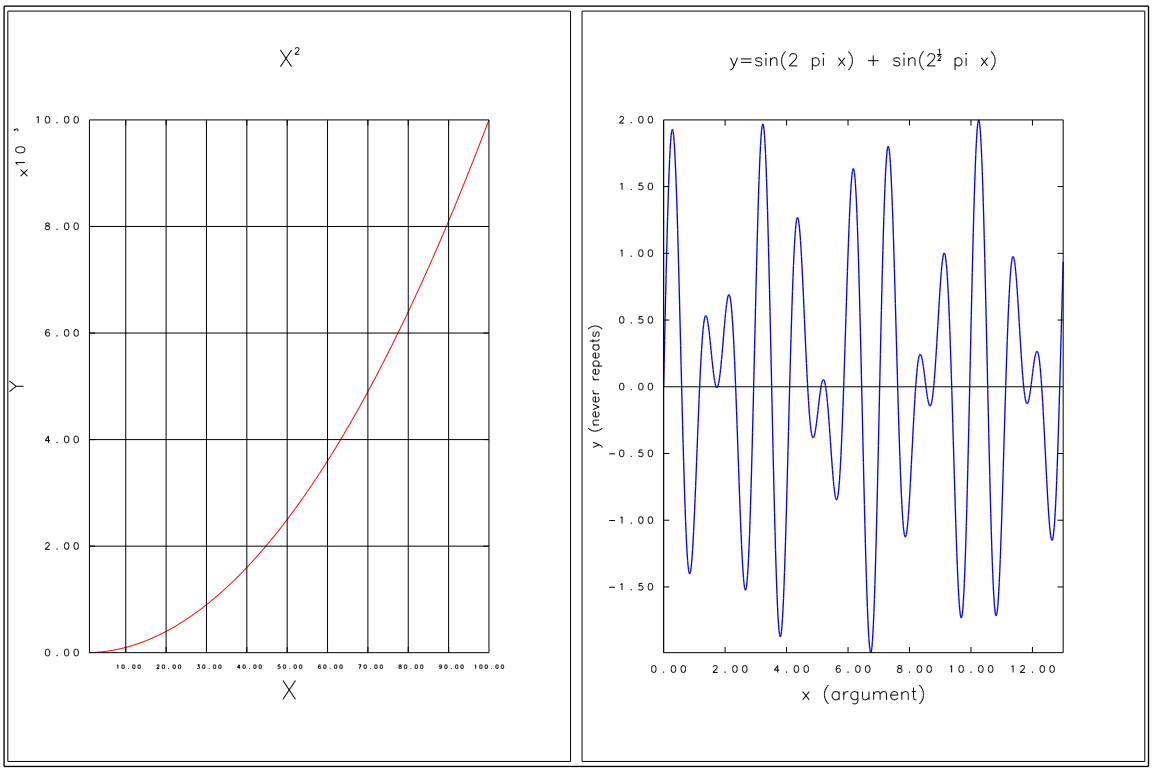


Figure 26: Graphs of functions calculated by the evaluator

```
#
OUTLINE DEV
```

The two commands that generate the data to be plotted are `ERANGE` and `EVAL`. The `ERANGE` command defines a range of x values for which the function is the be evaluated. In the case given here, it is similar to the NumPy "linspace" function. The first argument is the logarithm base, which, if 1, indicates we want to sample a linear range. The next two values are the start and end x values, and the final argument is the number of steps to take between start and end. `ERANGE` puts its output in the X array, which is also stack level 0.

The RPN evaluator does not use the X array as a "normal" part of the stack. It can be read via the `X` operator, which pushes its contents on the stack. The X array can be written to by the `SETX` operator. Otherwise, it is not accessed.

The `EVAL` operator needs one argument (a second is optional) which is an "RPN string" containing operators and operands for the evaluator to, er, evaluate. It may be enclosed in double quotation marks.

Whatever the evaluator leaves in the stack level 1 or Y array can be plotted against the X array contents using `XYLINE` or the other graphing commands.

To make the evaluator more useful, there are 9 "procedure registers" which can contain "RPN

strings". These can be executed by the evalautor using the notation:

```
EVAL @1
EVAL @1,@3
```

which "calls" one or more procedure register contents (sequentially). The contents of a "procedure register" can be set with the PROC command. They can also be set with the LOADPROC command, which looks for a named procedure in the file GPLPROC and loads it into a specified "procedure register" if it is found.

Note that the **GPLOT** evaluator does not actually do a "call and return" when it sees EVAL @1, etc. Instead, it uses string substitution to replace @1 with the string in procedure register 1. It does this repeatedly, if necessary, until there are no remaining @n sequences in the substituted string.

There are also 9 *scalar* "memory" registers, which can be set with the STO command and recalled with RCL. Often, parameters for procedures are put into these registers with STO then accessed in the RPN with the RCL or (as an abbreviation) #.

To make evaluating procedures with parameters more straightforward, the optional second argument to EVAL is a comma separated list of numeric values to STO into consecutive scalar registers (starting with the 1st) before running the RPN string. For example,

```
EVAL @1 "1,2,3,4"
```

would put 1 in scalar register 1, 2 in register 2, and so on, then "call" the procedure in "procedure register" 1.

Finally, there are 9 "string registers" which can be set with arbitrary strings (maximum 80 characters) using the STRING command. These can then be used in an RPN procedure for plotting text.

Returning to plotting simple functions, this script graphs a slightly more complex function:

```
CSG ALL
COL 0 0 0
WIDTH 1
CSG GENERAL
COL 0 0 1
WIDTH 2
#
ERANGE 1 0 13 801
EVAL X,TWPI,*,SIN,X,PI,2,SQRT,*,*,SIN,+
#
XLABEL "*LX (ARGUMENT)"
YLABEL "*LY (NEVER REPEATS)"
TITLE "*LY=SIN(2 PI X) + SIN(2*+*,1$,2$.$+ PI X)"
XYL
```

This is plotted in the righthand sub-figure of Figure 26. This function is mathematically interesting, as it never repeats (i.e. its period is infinite). This will not be the case when it is plotted on a digital computer, though, as all numbers on such machines are rational, but the period will be enormous.

10.2 General drawing - Part 1: Basic functions

DIMFILM provides some basic drawing capabilities apart from its extensive graph plotting functionality. Graph plotting is based on these lower level capabilities, as you would expect.

There is a simple but sound foundation for general drawing with the `BOUNDS` command, which establishes a user defined coordinate system, the `PANE` command, which sets up a rectangle in bounds coordinates outside of which no drawing will occur (a "clip to inside" region) and the `BLANK` command, which sets a rectangle inside of which no drawing will occur (a "clip to outside" region). These commands map directly to **DIMFILM** subroutines.

For convenience, **GPLOT** adds a `CANVAS` command, which just sets the bounds and the pane to the same coordinate ranges.

All general purpose drawing uses the coordinates established by `BOUNDS`. If no explicit `BOUNDS` command is given, this defaults to 0 to 1 by 0 to 1, which will be a square region, centered in the output device drawable rectangle. As noted above, to use the full area of the output device, the aspect ratio of the bounds must match the aspect ratio of the device. This can be done automatically for graph plotting applications using `GRAPHMODE ON`, but you will generally want to choose your own coordinate ranges for general drawing, often bearing the device aspect ratio in mind when doing do.

The low level drawing operations available are `MOVE` to move the "current position" to a specified coordinate, `DRAW` to draw from the current postion to a new position (which becomes the current position) in a straight line, `CIRCLE` which draw a circle of a given radius at a given position, `ARC` which draws a segment of a circle, and `RECT` which draws a rectangle.

Two **GPLOT** commands build slightly on this: `CRECT` draws a rectangle centered on a given location and `PATH` uses the contents of the X and Y arrays to draw a polyline. **GPLOT/DIMFILM** does not support the drawing of any kinds of smooth curves for general drawing purposes (`INTERPOLATE` can interpolate smooth curves between data points for graph plotting, but this can't be used for general drawing).

DIMFILM has very limited geometric transformation support and **GPLOT** does not provide access to this. **GPLOT** does provide a more comprehensive geometric transformation capability through its RPN evaluator, though.

While these graphics facilities are fairly basic, there is also a rather comprehensive set of facilities for drawing text using vector fonts. These are accessed through the `TEXT` and `CTEXT` commands, with various properties set by the `SYMHT`, `SYMANG` and `FONT` commands. The fonts are specific to **DIMFILM** (there is no way to use any "system fonts"!) and are, in this implementation, some of the Hershey fonts. The available fonts can be listed with the `LISTFONT` command.

Special "marker" characters can also be drawn using the `MARKER` command with the optional 2nd argument set to `YES`.

The text output facility allows "markup" to be included in the string to draw, and this extends to sub-scripts, super-scripts, and fractions, which can be nested to two levels. Mathematical notation

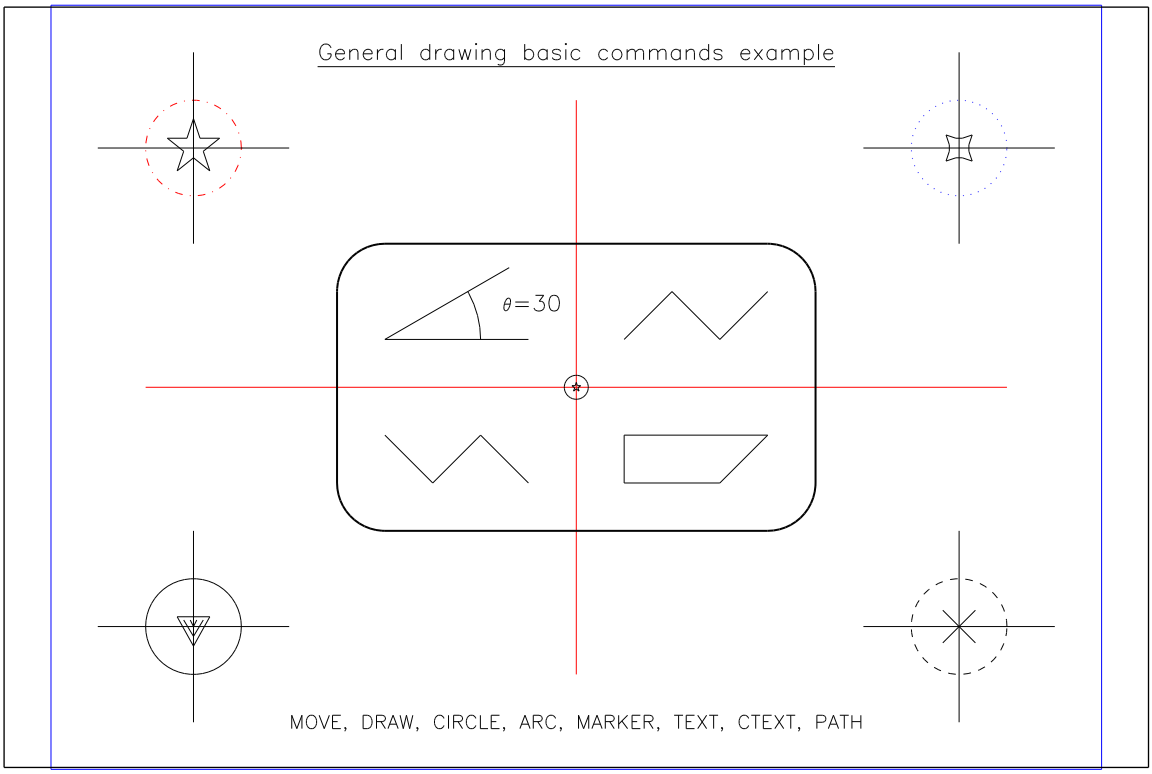


Figure 27: Basic drawing functions

can, to a useful degree, be accommodated by this. Although we have high quality font drawing everywhere now (and for the last few decades), when **DIMFILM** was first released (1973), this was cutting edge. Mathematical notation is still not readily expressible everywhere it might be useful even today, in fact.

An example which shows most of the basic drawing operations in action is Figure 27.

The script for this is a little long, but may be worth presenting in full:

```
RESET
CANVAS 0 22 0 16
#
CSG TEXT
COL 0 0 0
CSG GEN
COL 0 0 0
#
# DRAW CIRCLES
CIRCLE 3 3 1
STYLE DASH
CIRCLE 19 3 1
STYLE DOT
COL 0 0 1
CIRCLE 19 13 1
STYLE DASHDOT
COL 1 0 0
```

```
CIRCLE 3 13 1
STYLE SOLID
COL 0 0 0
#
# DRAW ANGLE
MOVE 7 9
DRAW 10 9
MOVE 7 9
DRAW 9.5981 10.5
ARC 7 9 2 0 30
MOVE 9.45 9.5
SYMHT 0.5
TEXT "*:83=30"
#
# DRAW CENTER CROSS + MARKERS
COL 1 0 0
MOVE 2 8
DRAW 20 8
MOVE 11 2
DRAW 11 14
MOVE 11 8
COL 0 0 0
MARKER 9 Y
MARKER 23 Y
#
# DRAW CIRCLE CROSSES AND MARKERS
MOVE 1 13
DRAW 5 13
MOVE 3 11
DRAW 3 15
SYMHT 3
MOVE 3 13
MARKER 9 Y
#
MOVE 17 13
DRAW 21 13
MOVE 19 11
DRAW 19 15
MOVE 19 13
MARKER 24 Y
#
MOVE 1 3
DRAW 5 3
MOVE 3 1
DRAW 3 5
MOVE 3 3
MARKER 13 Y
#
MOVE 17 3
DRAW 21 3
MOVE 19 1
DRAW 19 5
MOVE 19 3
MARKER 3 Y
#
# DRAW ROUND CORNER RECTANGLE
WIDTH 3
```

```
ARC 7 6 1 180 270
ARC 15 6 1 270 360
ARC 7 10 1 90 180
ARC 15 10 1 0 90
MOVE 6 6
DRAW 6 10
MOVE 16 6
DRAW 16 10
MOVE 7 5
DRAW 15 5
MOVE 15 11
DRAW 7 11
WIDTH 1
#
# ODDITIES
MOVE 7 7
DRAW 8 6
DRAW 9 7
DRAW 10 6
#
MOVE 12 9
DRAW 13 10
DRAW 14 9
DRAW 15 10
#
READ HERE 1 2
12 6
14 6
15 7
12 7
EOF
PATH C
#
SYMHT 0.5
MOVE 11 15
CTEXT 0 "*=G*LENERAL DRAWING BASIC COMMANDS EXAMPLE$="
MOVE 11 1
CTEXT 12 "MOVE, DRAW, CIRCLE, ARC, MARKER, TEXT, CTEXT, PATH"
#
CSG GEN
COL 0 0 1
OUTLINE BOUNDS
OUTLINE DEV
```

Most of this is likely to be self explanatory. The only points that may be unclear are:

- The use of a “special character” from the symbol font (which is loaded with `MATH.SMALL` by default) to draw the “theta” character. This is done by: `TEXT "*:83=30"` where the “theta” is selected by character number (83) using string markup. How do we know which character number to use? By looking at “font tables” which can be generated with one of the supplied script files (`obfont` for “normal” alphabets, `obsyms` for symbols and `obmarks` for markers). This is explained further in a later section (below).
- The `MARKER` command, which will draw a specified marker (which is determined by a number) at the current position *if* a second argument (`YES` - abbreviated, any case) is supplied.

If only one argument is supplied, the `MARKER` command sets the marker number to use for points when plotting graphs.

- Polyline paths can be drawn with the `PATH` command with coordinates defined by a `HERE` file. This is quite a convenient way of setting up a path, actually. The path may bew open or closed depending on the argument to `PATH` which may be `C` for closed or `O` for open.

While you can draw a lot with these basic drawing commands, it can get pretty tedious and long winded. Some of that is inevitable, but some things could be "automated". In this case, the end coordinate for the non-horizontal "angle" line has to be set by `DRAW 9.5981 10.5` where those numbers we found by hand with a calculator! This is the sort of thing a computer ought to be able to do for you!

To help with this, in addition to simple evaluation of `y = f(x)` functions, the "evaluator" was added.

10.3 The RPN Evaluator for drawing 2D parametric functions

Bridging the graph plotting and general drawing worlds is drawing functions of a parameter, `t`, that trace out a path on the 2D plane. These are of the form:

```
x = f(t)
y = f(t)
```

Such functions can generate very pleasing patterns and often appear in "recreational mathematics". Some of the most familiar examples are the curves generated by the "Spirograph" "toy" (see this Wikipedia article), which we will come to shortly.

As a simpler example (from the point of view of **GPLOT** only!) we start with the "Farris mystery curve". The "mystery" is that a five-fold symmetry appears from formula which contain even number and no obvious source of such a symmetry.

While "Spirograph" uses two wheels, with one rotating in slip-free contact with the other, the Farris curves (many variations are possible) can be generated with three wheels in slip-free contact.

Figure 28 shows the "original" mystery curve.

```
# FARRIS MYSTERY CURVE
#
RESET
CANVAS -2 2 -2 2
#
MAXPOINTS 3000
#
EVAL 0,TWPI,2001,XLIN
PROC 2 X,SIN,X,6,*,SIN,2,/,+,X,14,*,COS,3,/,+
PROC 1 X,COS,X,6,*,COS,2,/,+,X,14,*,SIN,3,/,+
EVAL @1,@2,PTHO
#
CSG ALL
```

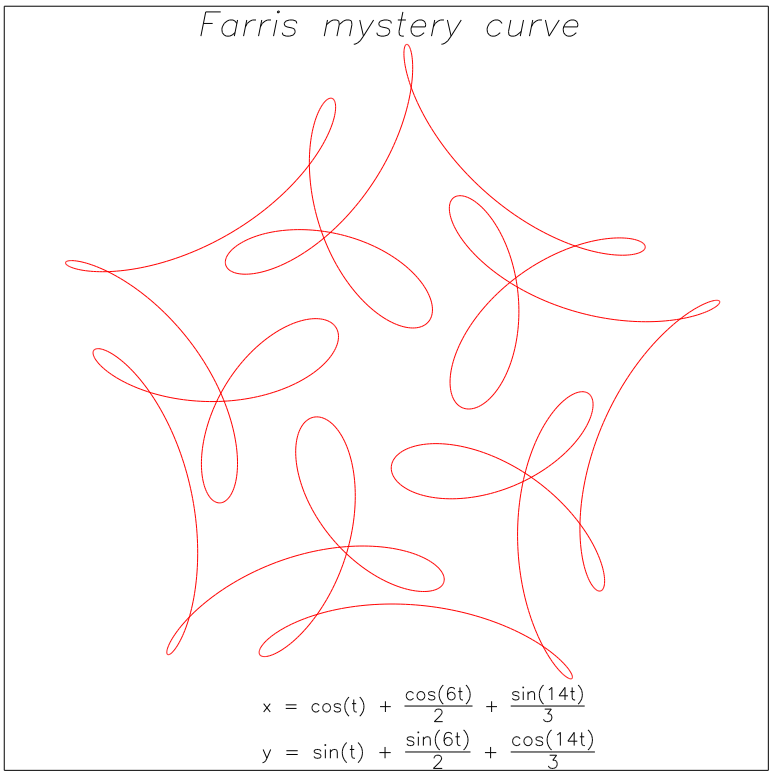


Figure 28: The Farris Mystery Curve

```
COL 0 0 0
MOVE 0  -1.66
CTEXT 1.3 "*LX = COS(T) + *,COS(6T)$,2$. + *,SIN(14T)$,3$."
MOVE 0  -1.9
CTEXT 1.3 "*LY = SIN(T) + *,SIN(6T)$,2$. + *,COS(14T)$,3$."
MOVE 0, 1.9
CTEXT 2 "*IF*LARRIS MYSTERY CURVE"
#
OUTLINE BOUNDS
```

The script that generates this is quite short and simple. The first `EVAL` fills the X array (stack level 0) with 0 to 2 pi in 2001 linear steps. Two procedures are then defined (stored in procedure registers 1 and 2). `PROC 1` calculates X values and `PROC 2` finds Y values. The second `EVAL` evaluates these two procedures in the order that leaves the calculated coordinates on stack for use by the path drawing operator `PTH0`.

It is easy to plot this curve as a graph (i.e. with value labelled axes, etc.) simply by replacing `PTH0` with operators that set the calculated X and Y values into the X and Y arrays (stack levels 0 and 1) used by graph plotting commands (here `XYLINE`). This is shown in Figure 29.

```
# FARRIS MYSTERY CURVE PLOTTED AS A GRAPH.
#
CANVAS -2 2 -2 2
#
MAXPOINTS 3000
```

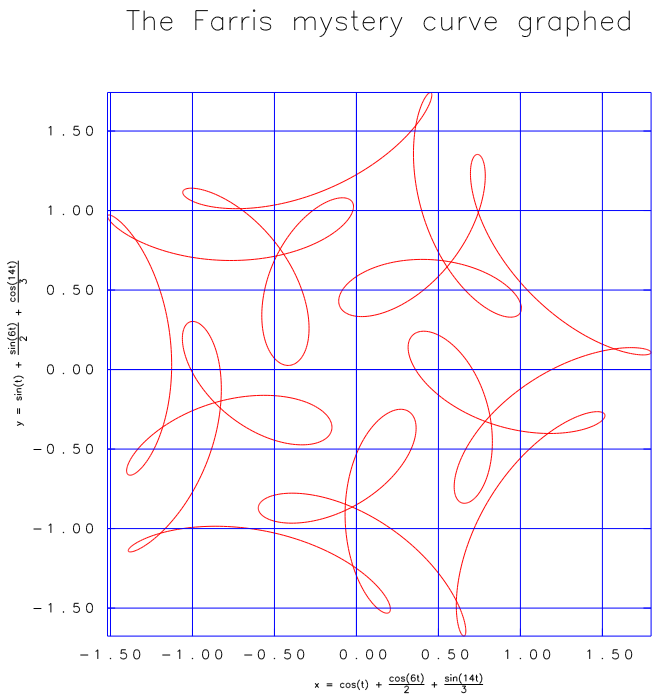


Figure 29: The Farris Mystery Curve graphed

```
CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0 1
CSG GEN
COL 1 0 0
#
EVAL 0,TWPI,2001,XLIN
PROC 2 X,SIN,X,6,*,SIN,2,/,+,X,14,*,COS,3,/,+
PROC 1 X,COS,X,6,*,COS,2,/,+,X,14,*,SIN,3,/,+
EVAL @1,@2,SETX,POP,SETY
GRAPHMODE OFF
TITLE "T*LHE *UF*LARRIS MYSTERY CURVE GRAPHED"
XLABEL "*LX = COS(T) + *,COS(6T)$,2$. + *,SIN(14T)$,3$."
YLABEL "*LY = SIN(T) + *,SIN(6T)$,2$. + *,COS(14T)$,3$."
GRID BOTH
XYLINE
```

The Spirograph is a toy invented by Denys Fisher in 1962-64 and fondly remembered by people of a certain age. It is a simple mechanism which can trace out hypotrochoid and epitrochoid curves based on two wheels (with gear teeth, so they can stay in contact without slipping). One wheel is fixed and the other rotates on the inside or outside of the fixed wheel. This second wheel has holes in which the tip of a pen can be placed to draw curves. If the moving wheel rotates inside the fixed wheel, a hypotrochoid curve is traced out, and if it is outside, an epitrochoid curve is drawn.

Much less accessible mechanisms were developed in the 19th century to draw such curves for use in banknote engraving as they are visually attractive and are (or were) difficult to forge without specialised machinery.

Drawing these curves with **GPLOT** is an interesting exercise. They can be defined with three integer parameters:

- R, the radius of the fixed wheel.
- RL, the radius of the smaller, moving, wheel.
- P, the distance of the "pen hole" from the centre of the moving wheel.

We want to be able to change these easily, so we want to define a script that we can call like this:

```
OB OBSPIRO "65 25 30"
```

where we pass the values of R, RL and P to the script. This is a good opportunity to show how to pass parameters to scripts and to use "nested scripts" to reuse code in the way that subroutines and functions allow in more sophisticated programming environments.

The script OBSPIRO uses these parameters to calculate how many times the moving wheel needs to circle the fixed wheel to complete a closed curve, as well as the degree of symmetry of the curve. Note that the parameters passed to the script are referred to as $1, $2, etc. as they are in Unix scripting languages. Note also that it is possible to use parameters in the RPN strings used by EVAL. This can be very useful.

One "problem" with the parametrised Spirograph simulation is that we don't know how big the pattern will be – i.e. what the range of coordinates will be – so we can't easily set the BOUNDS for the drawing.

The OBSPIRO script deals with this using the BBSTART, BBEND and BBSET commands. Between BBSTART and BBEND, the DRAW command doesn't draw anything (all drawing ultimately comes down to MOVE and DRAW). Instead, the range of X and Y coordinates is accumulated. BBSET can then be used to set these accumulated ranges as the BOUNDS, optionally scaled about the center by a scale factor (here, 1.2).

The tracing out of the Spirograph is done by the script OBSPIRC, which OBSPIRO "calls" twice: once to find the pattern's bounds and once to draw the pattern.

In general, parameters can be passed to obey scripts in two ways:

- Using parameters as described above, where, for example, OB X "1 2 3" results in $1, $2 and $3 in the script X being read as 1, 2 and 3 as a result of *text (string) substitution*.
- Numerical values and strings can be passed through registers. This is necessary if values are calculated by EVAL to be used in a "nested" script.

The OBSPIRO script demonstrates all these features.

```
# SPIROGRAPH PATTERNS - OBSPIRO
#
# FOR MORE INFORMATION, SEE:
#   HTTPS://LINUXGAZETTE.NET/133/LUANA.HTML
#   HTTPS://APERIODICAL.COM/2021/12/THE-MATHEMATICS-OF-SPIROGRAPH/
#
RESET

# THE CONTROL PARAMETERS SHOULD ALL BE INTEGERS.
#
# $1 = R = BIG CIRCLE RADIUS
# $2 = RL = LITTLE CIRCLE RADIUS (MOVES AROUND BIG CIRCLE, IN CONTACT)
# $3 = P = LITTLE CIRCLE PEN RADIUS (FROM CENTRE OF LITTLE CIRCLE)

# THE DEGREE OF SYMMETRY OF THE PATTERN IS MAX(R/GCD(R,RL),RL/GCD(R,RL))

# THE NUMBER OF REVOLUTIONS AROUND THE BIG CIRCLE NEEDED TO JOIN
# UP TO THE START OF THE PATTERN IS:
# NR = RL / GCD(R,RL) (GCD=GREATEST COMMON DIVISOR)

# THE PARAMETER RANGE NEEDED TO DRAW THE THING IS THEN:
# T = [0:2*PI*NR]
#
# AND:
# X(T) = COS(T) * (R - RL) + COS(((R - RL) / RL) * T) * P
# Y(T) = SIN(T) * (R - RL) - SIN(((R - RL) / RL) * T) * P

# CALCULATE NR AND STORE IN REG 9. SAVE GCD(R,RL) IN REG 8.
# NOTE THE DOUBLE QUOTES ARE NEEDED FOR PARAMETER SUBSTITUTION TO WORK
EVAL "$2,$1,$2,GCD,8,=,/,9,="

# FIND THE DEGREE OF SYMMETRY. STORE IN REG 8.
EVAL "$1,8,#,/,$2,8,#,/,MAX,8,="

# FIND THE BOUNDING BOX AND SET BOUNDS.
BBSTART
OB OBSPIRC "$1 $2 $3"
BBEND
BBSET 1.2

# DRAW THE SPIROGRAPH
CSG GENERAL
COL 1 0 0
WIDTH 2
OB OBSPIRC "$1 $2 $3"

# DRAW ANNOTATION (OR NOT)
# CSG GENERAL
# COL 0 0 1
# WIDTH 2
# CIRCLE 0 0 $1
# EVAL "$1,$2,-,0,$2,C"
# EVAL "$1,$2,-,$3,+,0,$3,20,/,C"
# EVAL "$1,0,M,$1,$3,+,$2,-,$3,20,/,-,0,D"

# ADD A PARAMETER DISPLAY
CSG TEXT
```

```
COL 0 0 0
BOUNDS 0 1 0 1
MOVE 0.05 0.05
SYMHT 0.02
TEXT "R=$1, *LR=$2, P=$3"
STRING 9 "S*LYMMETRY="
STRING 8 "(I2)"
EVAL "0.05,0.1,M,9,T,TSC,8,#,8,TVI,TEC"
FONT 2 SCRIPT.SIMPLEX
MOVE 0.8 0.95
SYMHT 0.07
CTEXT 0 "*2S*LPIROGRAPH!"
FONT 2 SERIF.COMPLEX

CSG GEN
COL 0 0 0
OUTLINE BOUNDS
```

The OBSPIRC script does the actual work of calculating pattern coordinates and drawing the result.

```
# SPIROGRAPH PATTERN CORE - OBSPIRC

# $1 = R = BIG CIRCLE RADIUS
# $2 = RL = LITTLE CIRCLE RADIUS (MOVES AROUND BIG CIRCLE, IN CONTACT)
# $3 = P = LITTLE CIRCLE PEN RADIUS (FROM CENTRE OF LITTLE CIRCLE)
# REG 9 = NR = NUMBER OF REVOLUTIONS AROUND BIG CIRCLE
#
# T = [0:2*PI*NR]
#
# X(T) = COS(T) * (R - RL) + COS(((R - RL) / RL) * T) * P
# Y(T) = SIN(T) * (R - RL) - SIN(((R - RL) / RL) * T) * P

STO 1 $1
STO 2 $2
STO 3 $3
MAXPOINTS 3000

ERANGE 1 0 6.28 2001
EVAL 0,TWPI,2001,XLIN,X,9,RCL,*,SETX
EVAL 1,RCL,2,RCL,-,5,STO   ; [5] = R - RL
EVAL 5,RCL,2,RCL,/,6,STO   ; [6] = (R - RL)  / RL
PROC 1 X,COS,5,RCL,*,6,RCL,X,*,COS,3,RCL,*,+ ; X
PROC 2 X,SIN,5,RCL,*,6,RCL,X,*,SIN,3,RCL,*,- ; Y
EVAL @1,@2,PTHO
```

This is quite similar to the “Farris Mystery Curve” script in its approach. Note that it stores the parameters it receives in registers for use in the RPN, although direct substitution could be used. It first generates an angle parameter at 2001 points that “goes around” the required number of times to generate a closed pattern, then it finds the path coordinates based on the angle and draws the result with the open path operator, PTHO.

Figures30, 31 and 32 are some examples of the output of OBSPIRO with different parameter sets.

Still sticking with procedurally generated patterns, but moving away from traditional continuous

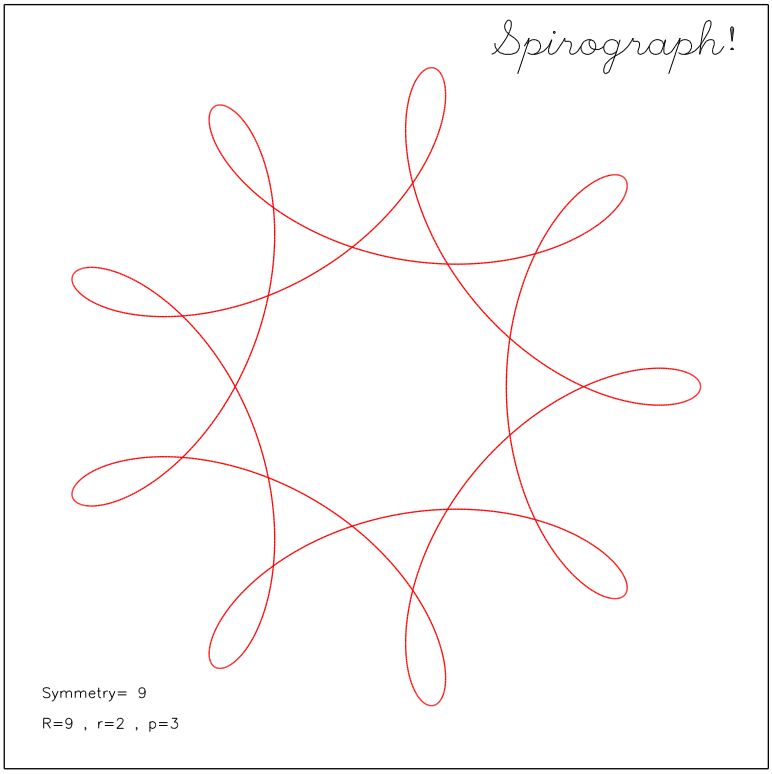


Figure 30: Spirograph Example 1

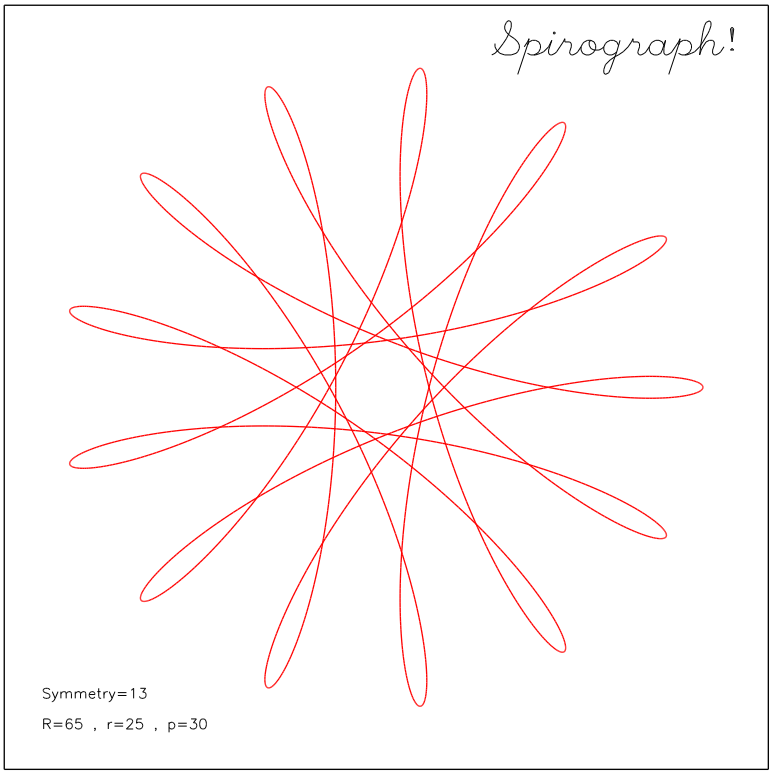


Figure 31: Spirograph Example 2

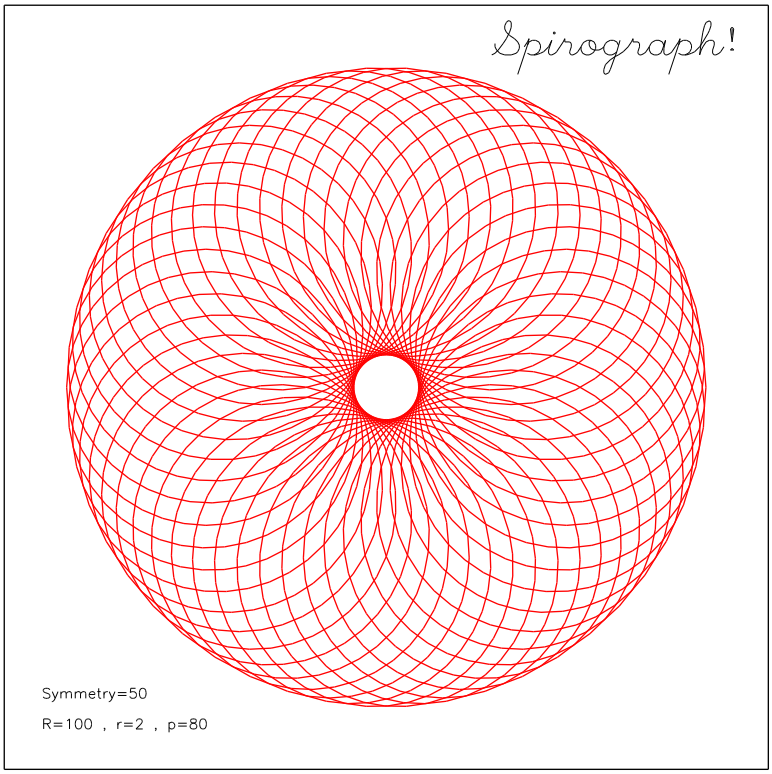


Figure 32: Spirograph Example 3

functions of a variable, there is a traditional form of Japanese Sashiko stitching called Hitomezashi, developed in the Edo period.

These stitch patterns consist of short stitches of a single unit length, made either horizontally or vertically. If two random sequences of length `N` are generated, one for the horizontal direction and another for the vertical, and a stitch is made if the value of the sequence is odd (and not made if it is even) then a very attractive pattern results.

I came across this thanks to this Numberphile, video on YouTube, which gives an excellent explanation of things.

The following obey script draws such patterns:

```
# GENERATE HITOMEZASHI STITCH PATTERNS
#
RESET
CANVAS 0 41 0 41
BLANK 30.5 40.5 37.5 40.5

ERANGE 1 0 40 41
PROC 1 1,RAND,0.5,+,FLR,1,EL,X,+,ODD
EVAL 333,SEED

CSG GEN
COL 0 0 1
WIDTH 3
```

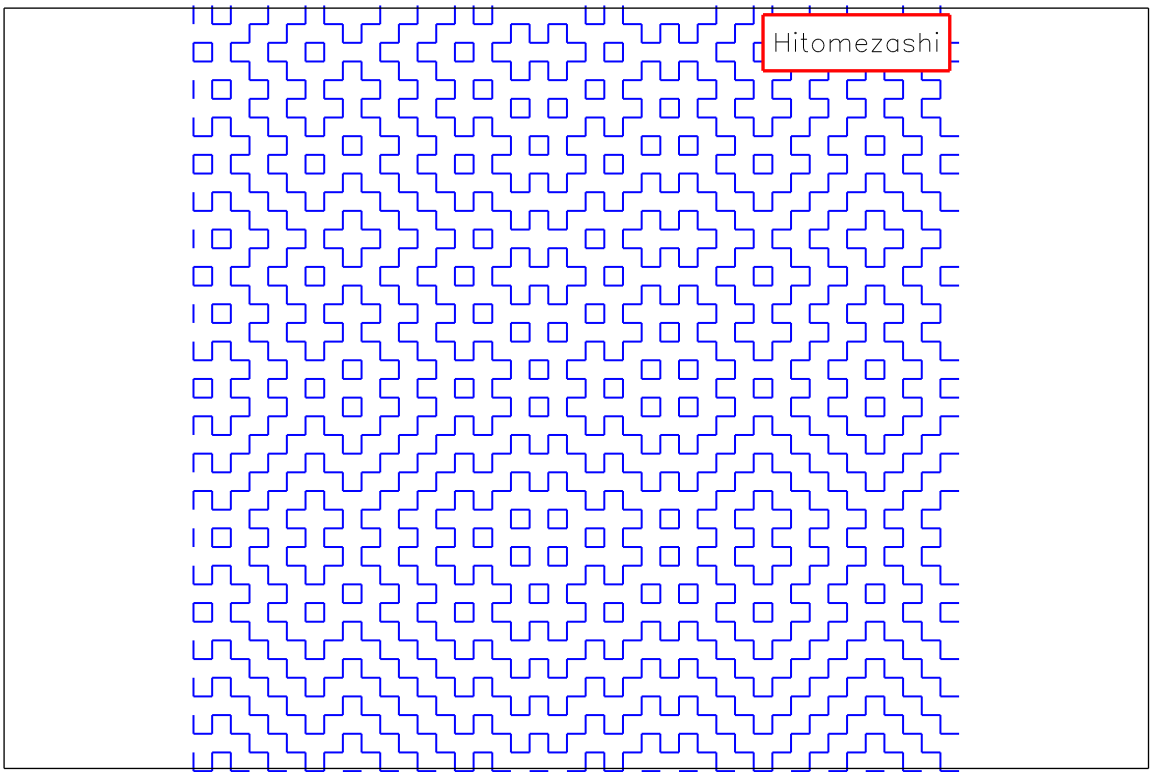


Figure 33: A Hitomezashi stitch pattern

```
ITEVAL 0 40 1 0,I,M,@1,HDSH
ITEVAL 0 40 1 I,0,M,@1,VDSH

CSG ANNOT
COL 1 0 0
WIDTH 5
OUTLINE BLANK

UNBLANK
MOVE 35.5 39
CSG TEXT
COL 0 0 0
WIDTH 1
CTEXT 9 "H*LITOMEZASHI"

OUTLINE DEV
```

This introduces a new version of `EVAL` – iterated evaluation or `ITEVAL`. This repeatedly runs an RPN procedure with a "loop variable" that iterates from a start value to a stop value (inclusive – Fortran DO-loop rules apply) by a step value. The "loop variable" value can be accessed inside the RPN using the `I` operator.

This example also shows how to use `BLANK` and `UNBLANK` to temporarily protect a rectangular region so that it cannot be drawn in.

10.4 General drawing - Part 2: Evaluator assisted

It is possible to use the evaluator to assist with general drawing tasks. In many cases, it is necessary to calculate coordinates from a small set of supplied parameters in order to complete the drawing task. A good example of this was the "labelled angle line pair" shown in the manual general drawing section, where we used a calculator to find the coordinates of the label text literally by hand.

Probably the most convenient way of using the evaluator for such tasks is to add procedures to the *evaluator procedure library* which consists of the file GPLPROC (gplproc on Unix-like systems).

The procedures in this library are, to some extent, a way to extend **GPLOT's** capabilities, or, at least, make some things much more convenient than they otherwise would be.

The contents of GPLPROC include comments that document its procedures. Please read it for full information. To understand its features, here are a couple of examples.

This RELLIPSE procedure draws an ellipse (which **GPLOT**/**DIMFILM** cannot do directly). It has five parameters, the center position (XC,YC), the semi-major and semi-minor axis lengths and a rotation angle (a rotation of the ellipse axes).

```
C A ROTATED ELLIPSE "XC YC A B ANGLE-DEGREES"
RELLIPSE
CL,0,TWPI,360,XLIN,X,&,COS,S,SIN,3,#,4,#,SCL,5,#,D2R,ROT,1,#,2,#,TRN,PTHC
```

To use this procedure to draw a rotated ellipse, two **GPLOT** commands are needed:

```
LOAD 1 RELLIPSE
EVAL @1 "5,1,0.5,0.9,33"
```

The first line loads the required procedure into a procedure register (1 here), then the second executes it, passing it the required arguments. If a given procedure is to be "called" multiple times, it need only be loaded once, of course.

The second example shows a useful feature unique to GPLPROC. A major limitation of the evaluator is that RPN strings are limited to 80 characters. This is not enough for many "real world" applications. When a procedure is loaded from GPLPROC, it can span multiple lines. In such a case, the procedure is loaded into multiple successive procedure registers and ,@n sequences are automatically added to line ends so that the when the procedure is called from an EVAL command, a single long procedure is assembled and executed. A side effect of this scheme is that lines can be no more than 77 characters long if another line follows in the procedure definition.

```
C GEAR "X Y R1 R2 N-TEETH ROT-DEGREES"
GEAR 2
CL,0,5,#,&,1,+,XLIN,3,#,4,#,X,2,/,ODD,SEL,X,TWPI,*,5,#,/,6,#,D2R,+,&
COS,S,SIN,3,G,*,S,3,G,*,1,#,2,#,TRN,PTHO
```

Note that the number of lines in the full procedure (2 here) must be given after the procedure name. These procedures must be loaded into a procedure register number that leaves enough

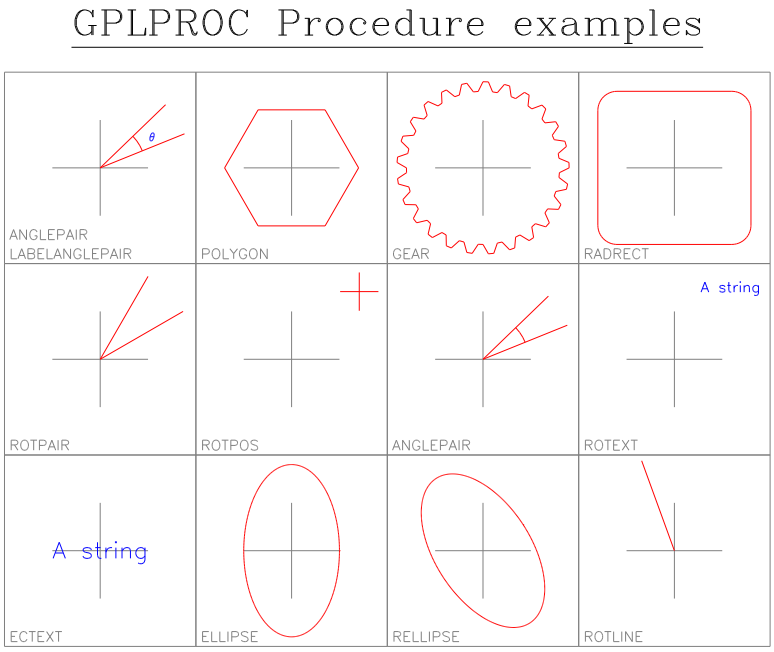


Figure 34: Examples of each procedure in GPLPROC

successive registers to hold the full thing. Here, for example, LOAD 8 GEAR would work, but LOAD 9 GEAR would not.

The script OBPRTST calls every procedure currently defined in GPLPROC as shipped. The output of that is shown in Figure 34.

It is intended that **GPLOT** users add their own procedures to GPLPROC to "extend" **GPLOT** to do things they want to do.

10.5 General drawing - Part 3: Higher level drawing

Although the evaluator can "extend" the capabilities of **GPLOT** to some extent, it has some pretty severe limitations and inconveniences. The main limitations are the maximum size of procedures and the complete lack of branching and looping features. In practice, you cannot "draw anything" with **GPLOT**/**DIMFILM** in the way that you can with PostScript, for example.

There are some drawing operations beyond the (very) basic facilities described above which are sufficiently generally useful to "build in" to **GPLOT**. This hopefully makes it relatively easy to draw things that are commonly useful (easier than using PostScript, for example), while acknowledging that there are still fundamental limitations to what can be drawn.

To this end, **GPLOT** has three "higher level" drawing "primitives":

- Decorated lines
- Boxed text
- Labels with pointing arrows.

These can be used for a broad range of diagrams, but especially block diagrams.

A **decorated line** is a series of connected and directed line segments, the ends of which may have:

- Arrow heads. These will automatically point in the correct direction (hopefully).
- "Half skips". These are (close to) quarter circles, and if a line ends and begins with one of these, it can give the appearence of skipping over another line, as was commonly seen in older circuit diagrams.
- No decorations or "plain".

Such a line is drawn by the `LINE` command which takes a string that defines the line segments and their decorations. Each coordinate or *point description* defining a "vertex" in the "polyline" has the following components:

```
<decoration>x,y[,<annotation_string>]

<decoration> = P (plain) or A (arrow) or S (half skip)

<annotation_string> is a short string (currently limited to 5 characters).
It may contain \textbf{DIMFILM} string markup. It cannot contain spaces.
```

The annotation string, if supplied, will be drawn in a circle at the mid-point of the line segment ended by the point description that specifies it. This is intended to be a *very short* label.

The point descriptions are separated by `">"` characters, which can be read in this context as "to".

Here is an example that defines a two segment decorated line, starting with no decoration and ending in an arrow head with a label, `"Le"`, in the middle of the first of the two line segments:

```
LINE P7,6>P7,21,L*LE>A16,21
```

Additional characteristics of decorated lines are specified by the `ARROWPARM` and `LINEPARM` commands. The latter sets scale factors for drawing skips and annotations, while the former sets the type, size and appearance of arrow heads. Note that the width and colour of decorated lines are set in the usual way (lines, arrows and skips are in `CSG GENERAL`, annotation text in `CSG TEXT`).

Boxed text is intended to be a fairly flexible component for constructing block diagrams, tables and many other drawings. The element is basically text centered in a box of a specified size at a specified position (given either by centre or bottom left).

A number of other features can be added, though.

- The inside of the box can be “filled” with hatched lines.
- A second box, inside the first and surrounding the text, can be drawn. It is a BLANK region, and it can be outlined.
- The text may have multiple lines (up to 5), separated by backslashes.
- The text may be centered or left justified.
- The text can contain **DIMFILM** string markup.
- The position of the boxed text can be automatically “stepped” by specified X and Y “deltas” after eah one is drawn. This makes it much easier to use this for constructing tables and other things.

The main command for this element is BOXTEXT, which draws a given text string at a position given by either a supplied coordinate, or, if the coordinate is omitted, at the position of the last BOXTEXT incremented by previously specified deltas.

A number of secondary commands set current characteristics of BOXTEXT. These are:

- BOXPSIZE which sets the box size and how the position coordinate is to be interpreted.
- BOXPHATCH which sets the hatching parameters (or turns it off).
- BOXPTEXT which sets how text is drawn inside the box. There are two distinct modes: FIX and SCALE. In FIX mode, the WIDTHSCALE parameter sets how many lines will fit in the box height, while in SCALE mode, it sets the fraction of the box width the text should be scaled to fill.
- BOXPBOX which controls whether the outer and inner boxes are outlined.
- BOXPDELTAS which sets the automatic position step after ecah BOXTEXT is drawn.

Labels are the same graphical entities used to label points on a graph, but the coordinates of the thing they point at is in bounds rather than graph coordinates.

Figure 35 is a simple example of how these facilities can be used to draw a small block diagram.

```
RESET
BOUNDS 0 42 0 26

CSG TEXT
COL 0 0 0
CSG ANNOT
COL 0 0 0
WID 2

BOXPSIZE 10 6 NO

BOXT "B*LOX 1" 7 3
BOXT "B*LOX 2" 21 3
```

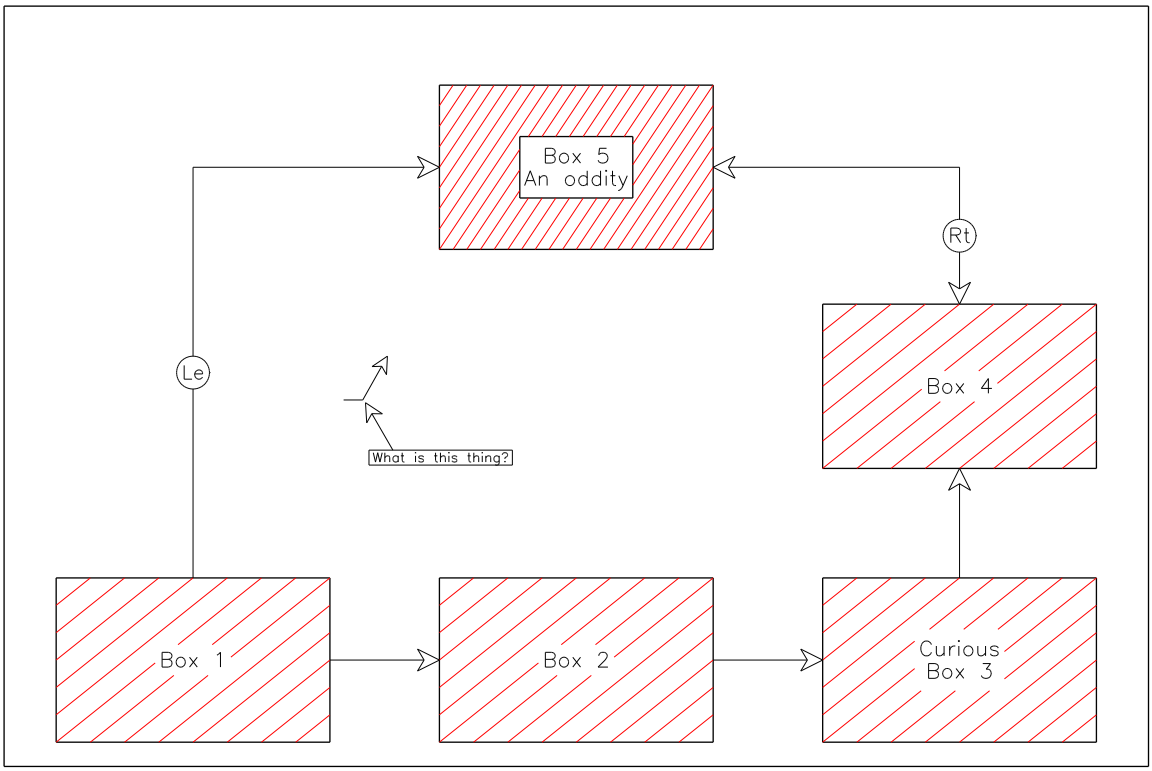


Figure 35: A simple block diagram

```
BOXT "C*LURIOUS\*UB*LOX 3" 35 3

BOXT "B*LOX 4" 35 13

BOXPHATCH 0.05 8 HORIZ
BOXPBOX BOTH
BOXT "B*LOX 5\*UA*LN ODDITY" 21 21

CSG GEN
COL 0 0 0
ARROWPARM BARBED 0.8 2 0.3
LINEPARM 0.5 1.5
LINE P12,3>A16,3
LINE P26,3>A30,3
LINE P35,6>A35,10
LINE A35,16>P35,21,R*LT>A26,21
LINE P7,6>P7,21,L*LE>A16,21

LINE P12.5,12.5>P13.2,12.5>A14.1,14.1
ALABEL 13.2 12.5 2 300 "What is this thing?"

OUTLINE DEVICE
```

It may look painful to create a diagram such as this by entering coordinates, and perhaps it is. However, it may be less painful than it seems at first. A key simplification is to use an integer coordinate system with the box size a useful multiple of the unit size (e.g. so that middle of the

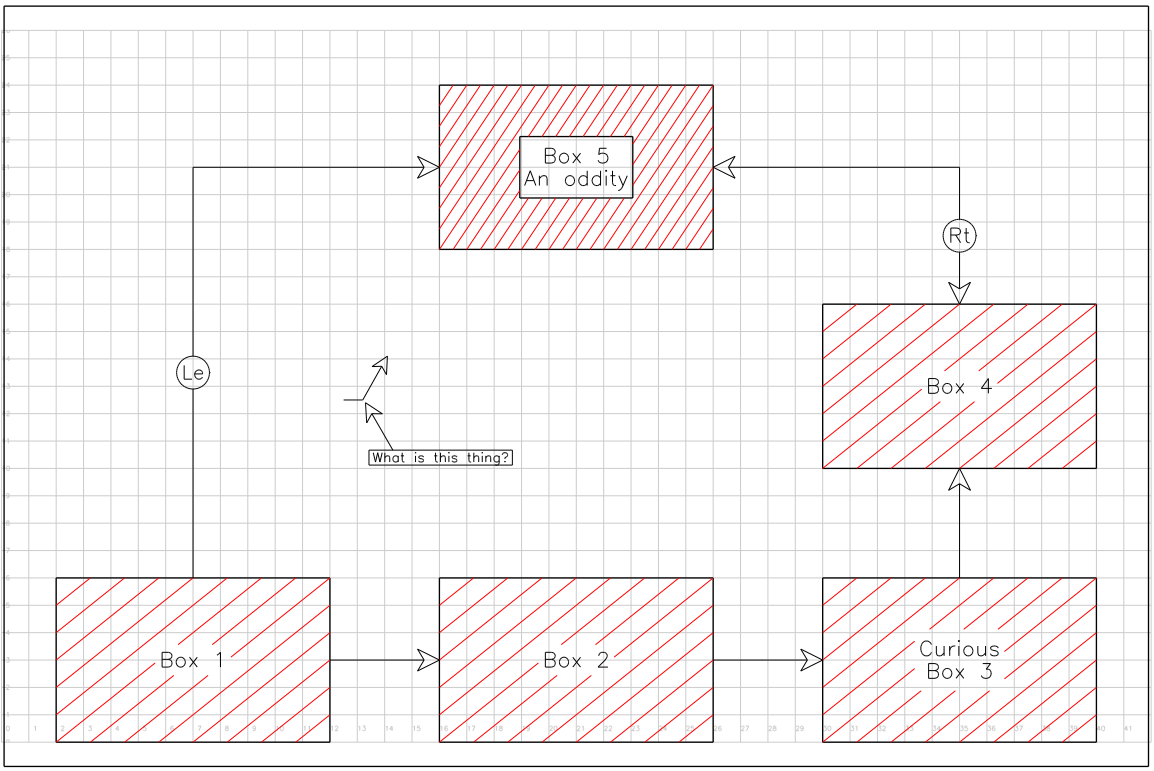


Figure 36: Using a grid to assist with planning a block diagram

box edges is easy to find). It is also helpful to have a grid printed out (!) with the coordinate system shown over a specified range (`BOUNDS`). As an aid to this, the script `OBGRID` can draw such a grid which can be used as the background of the diagram as it is developed (e.g. using the **GTerm** device) or printed to help sketch out the diagram. An example of the use of a grid is shown in Figure 36.

Surely it would be easier to just use, say, Inkscape? Perhaps. I have drawn many block diagrams with Inkscape and in some ways it is obviously easier. But it still takes longer than might be expected to get to a "final" result. It is also very difficult (at least for me) to get lines and boxes to "snap" together precisely (I usually give up on perfecting this, to be honest). The **GPLOT** approach at least results in exact alignment of all elements! In summary, I'm not sure it is all that more time consuming to get to an acceptable result. Which approach you prefer (using a mouse and relying on hand/eye coordination or working out and typing in coordinates) is a matter of personal taste. Obviously, if you want to draw a portrait of your dog, Inkscape (or a similar program) is the only sane choice! For more exactly defined tasks, a "language based" approach can make sense, as proved by the success of vector graphics description languages such as PGF/Tikz.

```
#
# DRAW A GRID WITH NUMBERED AXES. USEFUL FOR BLOCK DIAGRAMS.
# CALL WITH BOUNDS AS PARAMETERS.
#
BOUNDS $1 $2 $3 $4
CSG ALL
```

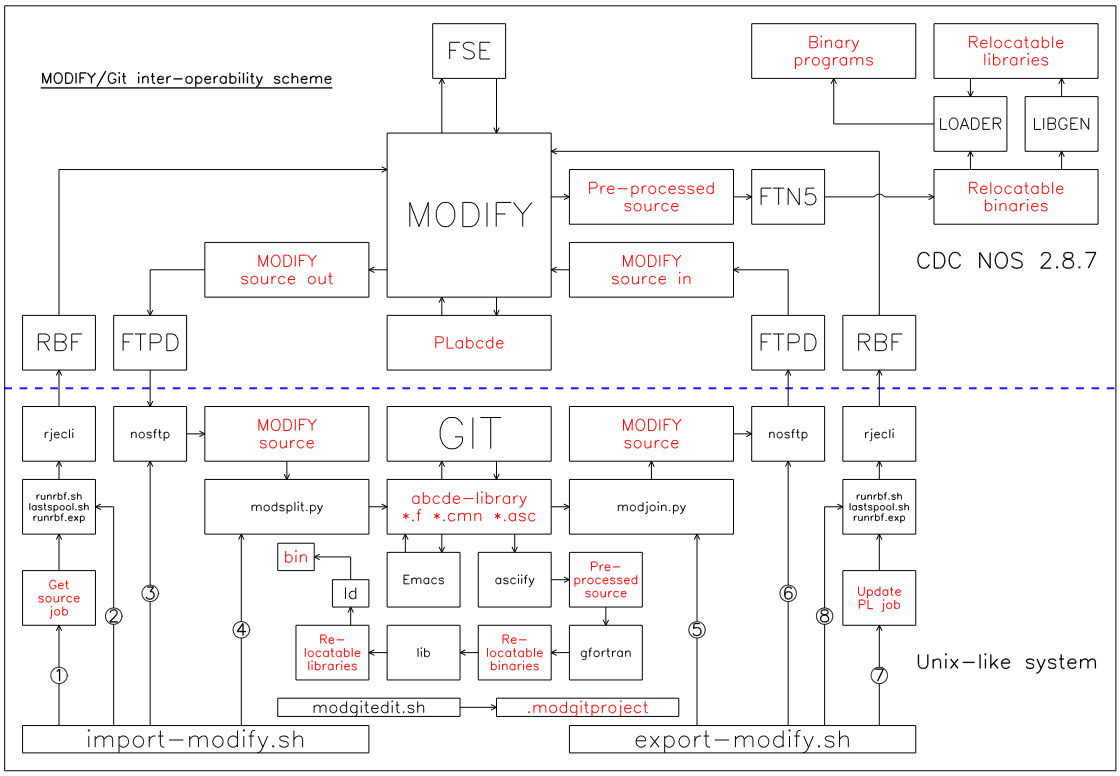


Figure 37: Git-MODIFY Inter-operability scheme components diagram

```
COL 0.8 0.8 0.8
STRING 9 "(I2)"
ITEVAL $3 $4 1 "$1,I,M,$2,I,D"
ITEVAL $1 $2 1 "I,$3,M,I,$4,D"
ITEVAL $3 $4 1 "$4,$3,-,100,/,TH,$1,I,M,I,9,TVI"
ITEVAL $1 $2 1 "$4,$3,-,100,/,TH,I,$3,0.5,+,M,I,9,TVI"
```

For a more sophisticated example of a block diagram drawn with **GPLOT**, look at the script `OBMODIO` which draws the block diagram at the top of the Github `README.md` document showing the component parts of the Git-MODIFY inter-operability scheme. This is shown in Figure 37.

An example of using **GPLOT** (primarily `BOXTEXT`) to produce a drawing other than a block diagram is shown below. This documents the key bindings for the NOS 2 Full Screen Editor with a Macbook laptop keyboard.

Further examples of how the “higher level drawing” features can be used can be seen in the `OBCHT1` and `OBCHT2` scripts which generate the **GPLOT** “cheat sheet” which can be found at the end of this document.

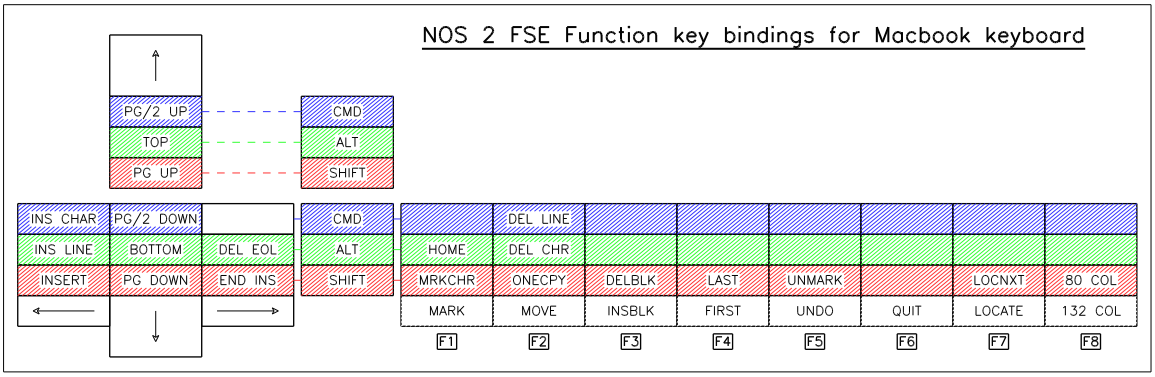


Figure 38: A diagram showing the key layout for FSE on a MacBook

10.6 L-Systems

L-systems or Lindenmeyer systems are a type of formal grammar (as in mathematical logic) developed by Aristid Lindenmeyer, a theoretical biologist, in 1968 to model the processes of plant development. They can be used to generate a variety of attractive images, beyond plants and trees, including self-similar fractals. See this Wikipedia article for more background information.

Lindenmeyer systems generating 2D vector graphics are included in **GPLOT**. The implementation is based on Paul Bourke's description of a 1991 commercial product (probably now long dead) which can be found here. As with many of Paul Bourke's web pages, this is very informative and inspirational.

L-systems work by manipulating strings, the characters of which either represent graphical actions (such as moving forward, drawing a line, or turning through an angle) or are "variables" with no associated graphical meaning.

An L-system is defined by an "axiom string" and a set of "re-writing rules", and, in our case, also a fixed "turning angle". The "axiom string" must be set in string register 1 and up to 8 "re-writing rules" can then appear in string registers 2 to 9 (used consecutively). The initial line angle is set by storing the angle (in degrees) in numeric register 3.

The characters allowed ("vocabulary") are: `F A B C D E M X Y + - [ ]` Seven of these are

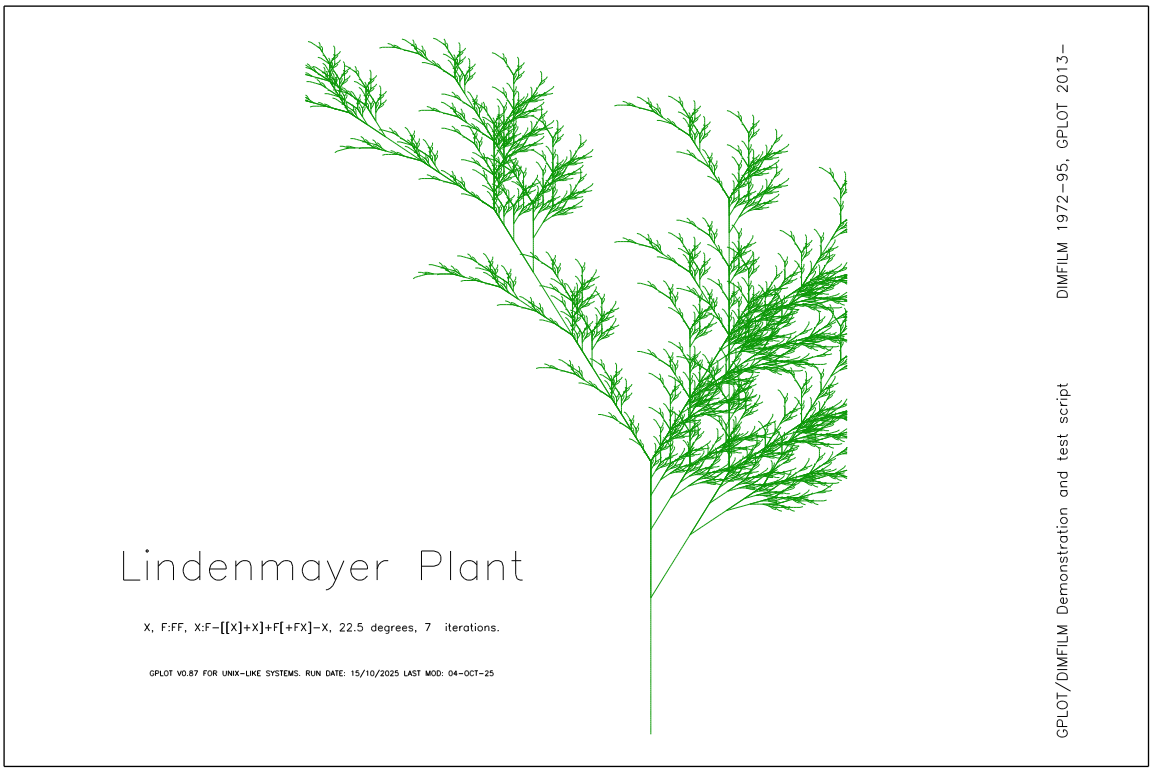


Figure 39: A Lindenmeyer plant also demonstrating other facilities

associated with specific actions. Any alphabetic character can be substituted with a string by a re-writing rule. The seven "action characters" are:

- `F` - draw a unit length line segment at the current drawing angle.
- `D` - as `F`, but some systems need two substitutable drawing characters.
- `M` - move by a unit length at the cureent drawing angle.
- `+` - turn counter-clockwise by the turning angle.
- `-` - turn clockwise by the turning angle.
- `[` - push the current position and drawing angle on a stack.
- `]` - pop the current position and drawing angle off a stack.

This is all very abstract, of course! Paul Bourke's web page gives a simple example which is quite easy to follow, but more complex systems are not obvious (to me, anyway).

Figure 39 is a classic example of a plant drawn with an L-system, drawn by the following script.

```
# PROCEDURE TO DRAW AN ALGORITHMICALLY DEFINED
# PLANT USING A LINDENMAYER SYSTEM.
# USAGE:
```

```
#  OBEY OBPLANT <ITERATIONS>
#   ITERATIONS BETWEEN 5 AND 8 SEEMS REASONABLE.

# INITIALISE GPLOT STATE.
RESET
GRAPHMODE ON

# SET THE COLOUR OF THE PLANT.
CSG GENERAL
COL 0.05 0.6 0.03

# DEFINE THE LINDENMAYER SYSTEM FOR THE PLANT.
STRING 1 X
STRING 2 F:FF
STRING 3 X:F-[[X]+X]+F[+FX]-X
STO 3 90

# DRAW THE PLANT
LSYSTEM 2 $1 22.5

# SETUP TO ADD ANNOTATION
GRAPHMODE OFF
BOUNDS 0 1.33 0 1
CSG TEXT
COL 0 0 0

# DEMONSTRATE CTEXT CENTRED TEXT.
MOVE 0.3325 0.265
CTEXT 0.532 "L*LINDENMAYER *UP*LLANT"
MOVE 0.3325 0.185
CTEXT 0.4655 "X, F:FF, X:F-[[X]+X]+F[+FX]-X, 22.5*L DEGREES, $1 ITERATIONS."

# DEMONSTRATE ROTATED TEXT VIA THE RPN EVALUATOR.
STRING 5 "GPLOT/DIMFILM D*LEMONSTRATION AND TEST SCRIPT"
STRING 6 "  DIMFILM 1972-95, GPLOT 2013-"
EVAL "1.31,0.05,M,90,TA,5,T,1.31,0.6,M,6,T"

# ADD THE GPLOT VERSION INFO, DRAWING CENTRED TEXT
# VIA A STORED PROCEDURE.
VERSION 9
LOADP 1 ECTEXT
EVAL @1 "0.3325,0.125,0.45,9"

OUTLINE DEV
```

Note that, in general, the complexity of the drawing explodes as the number of iterations increases. The **GPLOT** L-system implementation uses scratch files to avoid memory limitations on NOS, so large outputs are possible, but the compute times also explode! The maximum stack depth is set to 20, although this doesn't seem to be much of a limitation in practice.

Here are some other examples.

A complicated, bristly plant (Figure 40) is drawn by the following script.

```
RESET
```

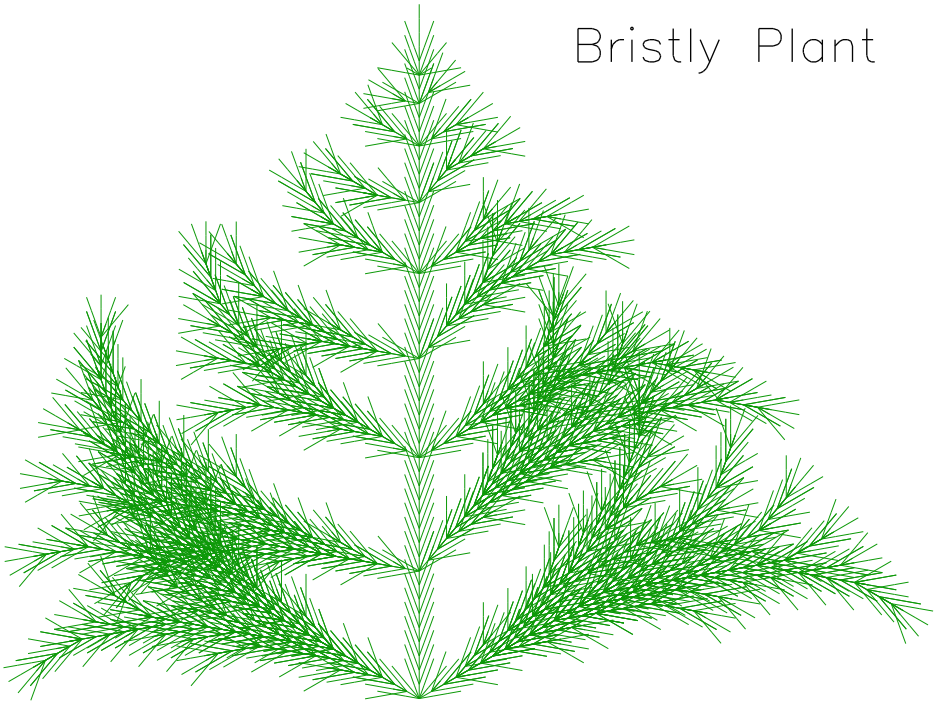


Figure 40: A bristly looking plant

```
# SET THE COLOUR OF THE PLANT.
CSG GENERAL
COL 0.05 0.6 0.03

# DEFINE THE LINDENMAYER SYSTEM FOR THE PLANT.
STRING 1 ABFFF
STRING 2 A:[+++C][---C]YA
STRING 3 C:+X[-C]B
STRING 4 X:-C[+X]B
STRING 5 Y:YB
STRING 6 B:[-FFF][+FFF]F

# INITIAL DRAWING ANGLE.
STO 3 90

# DRAW.
LSYSTEM 5 $1 20
```

Figure 41 shows a Minkowski island, after 4 iterations. L-systems can generate most (all?) 2D fractal curves. Several such curves follow.

```
RESET

STRING 1 F+F+F+F
STRING 2 F:F+F-F-FF+F+F-F
```

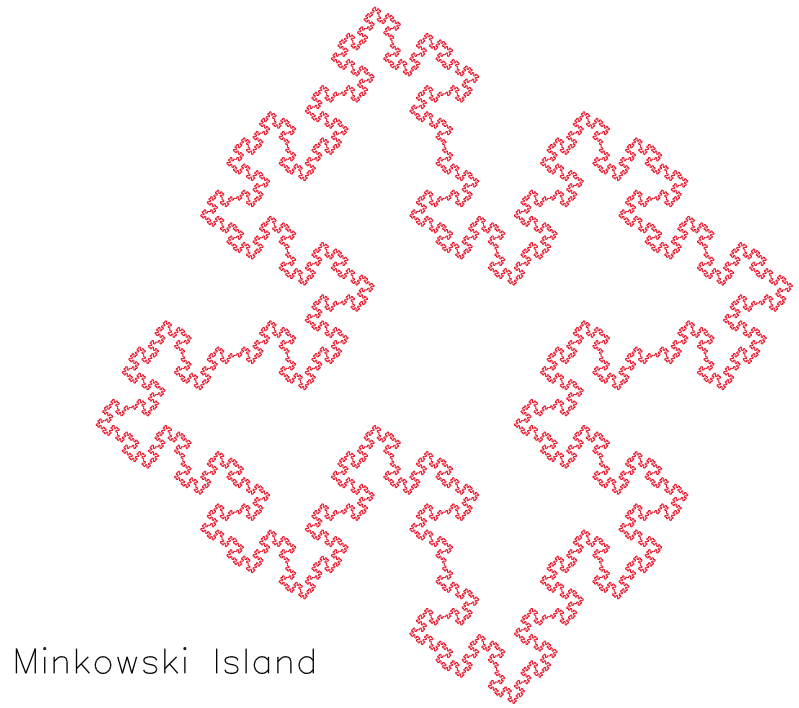


Figure 41: A Minkowski Island (4 iterations)

```
STO 3 0

CSG GENERAL
COL 0.9 0 0.1

LSYSTEM 1 $1 90
```

Figure 42 shows a Sierpinski gasket after 6 iterations, drawn by this script.

```
RESET

STRING 1 F+F+F          ; AXIOM
STRING 2 F:F-F+F+F-F    ; GENERATOR

STO 3 0                 ; INITIAL ANGLE

CSG GENERAL
COL 0.9 0 0.1

# DRAW THE GASKET
LSYSTEM 1 $1 120
```

Figure 43 shows a Koch “snowflake” drawn with rectangular elements is drawn by this script.

```
RESET
```

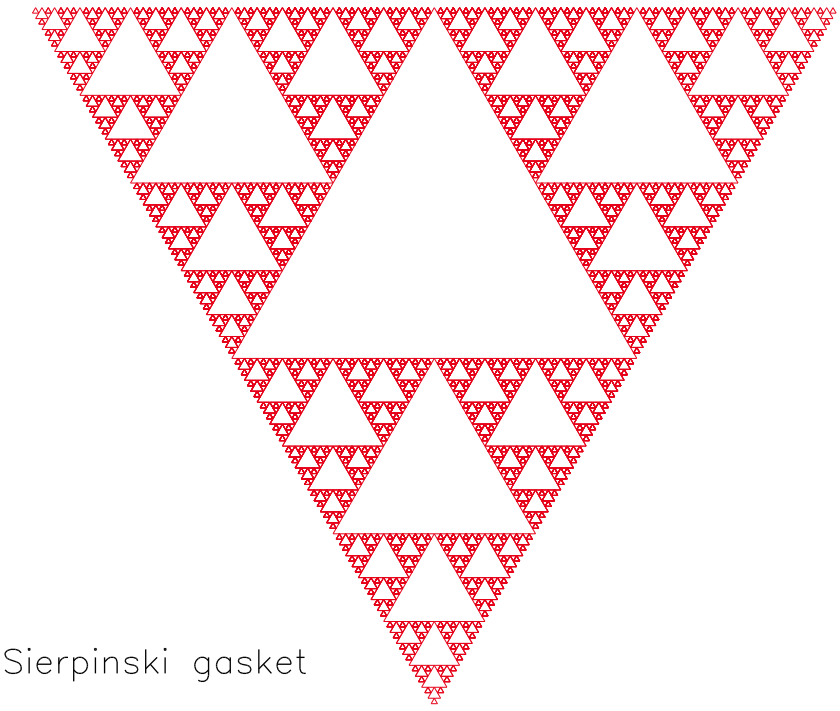


Figure 42: A Sierpinski Gasket (6 iterations)

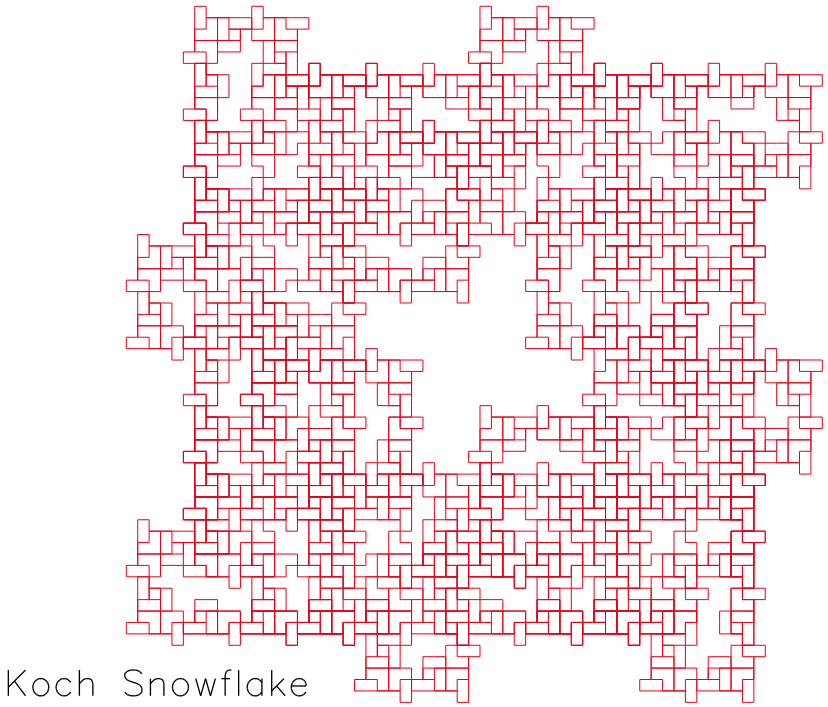


Figure 43: A Koch Snowflake (4 iterations)

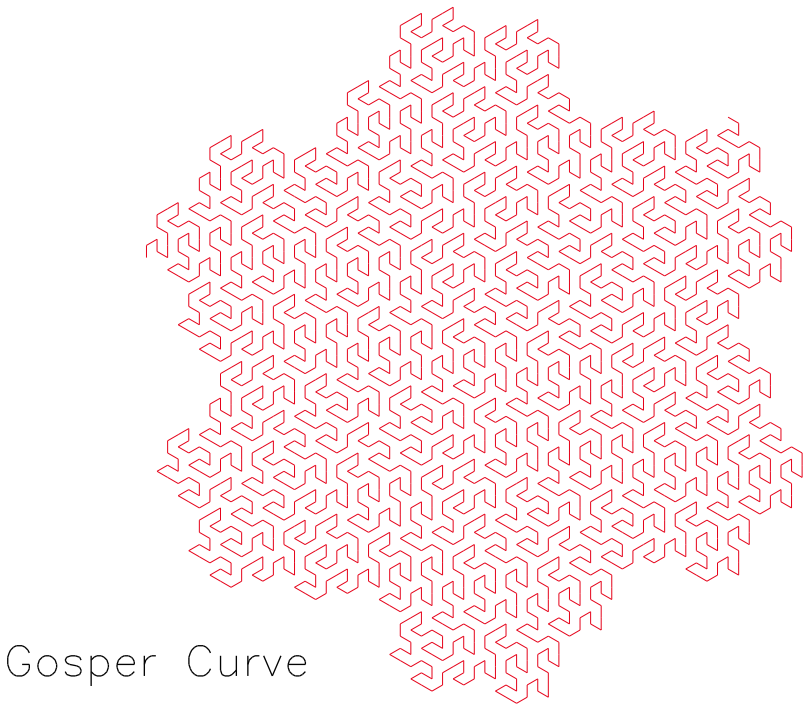


Figure 44: A Gosper Flowsnake curve (4 iterations)

```
STRING 1 F+F+F+F
STRING 2 F:FF+F-F+F+FF

STO 3 0

CSG GENERAL
COL 0.9 0 0.1

LSYSTEM 1 $1 90
```

Finally, Figure 44 shows a Gosper curve or “flowsnake” after 4 iterations, drawn by this script.

```
RESET

STRING 1 F
STRING 2 F:F-D--D+F++FF+D-
STRING 3 D:+F-DD--D-F++F+D

STO 3 90

CSG GENERAL
COL 0.9 0 0.1

LSYSTEM 2 $1 60
```

Part III

Appendices

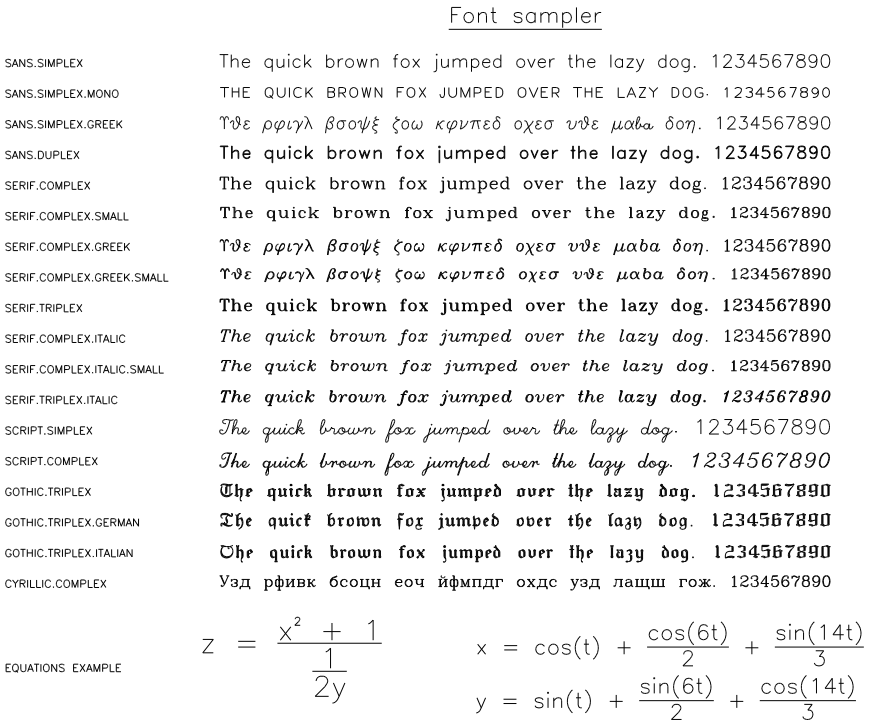


Figure 45: A font sampler

11 Fonts

As previously noted, the fonts that can be used in **GPLOT** are taken from the Hershey vector fonts (originally published in 1967). Figure 45 is a “font sampler” giving a quick overview of what is available.

The pages that follow give full font tables for the alphabetic, symbol and marker fonts. These can be used to find the character numbers needed to insert special characters in text, as well as showing all the fonts in detail.

11.1 Alphabetic fonts

This script will draw an alphabetic font table.

```
PROC 1 I,10,I0IJ,M,I,TC   ; DRAW CHARACTER I AT IJ
PROC 2 I,0.5,-,1,STO,0.5,CHS,M,1,RCL,9.5,D
PROC 3 0.5,CHS,I,0.5,-,1,STO,M,9.5,1,RCL,D
PROC 4 I,10,I0IJ,0.4,-,SWAP,0.4,-,SWAP,M,I,1,TVI
STRING 1 "(I2)"
FONT 1 $1
CANVAS -1 10 -1 10
CSG ALL
COLOUR 0 0 0
WIDTH 1
```

```
SYMHT 0.3
ITEVAL 1 96 1 @1
FONT 1 SANS.SIMPLEX
SYMHT 0.2
COLOUR 0.5 0.5 0.5
ITEVAL 1 96 1 @4
COLOUR 1 0 0
ITEVAL 0 10 1 @2
ITEVAL 0 10 1 @3
MOVE 4.5 9.75
SYMHT 0.4
CTEXT 0 $1
```

The argument to this script is the font name.

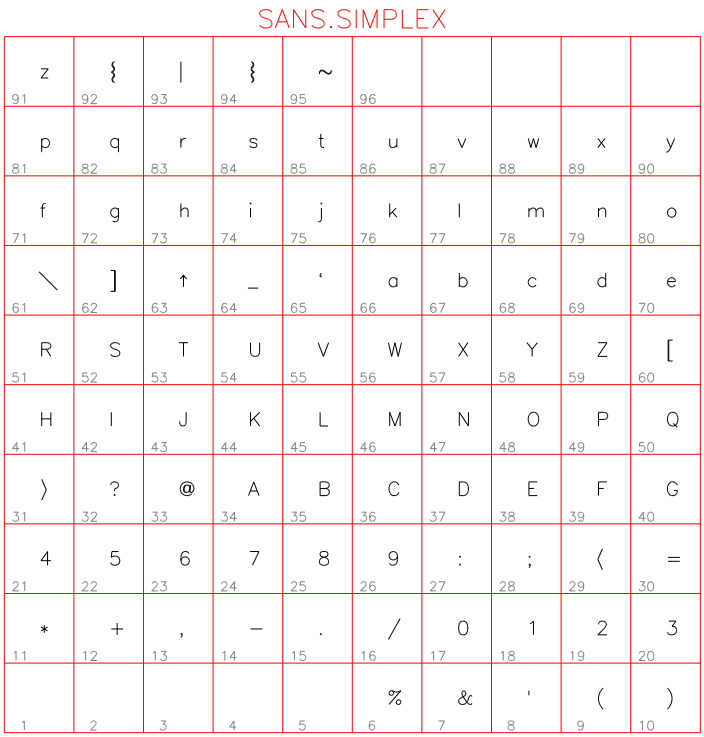


Figure 46: Sans Simplex

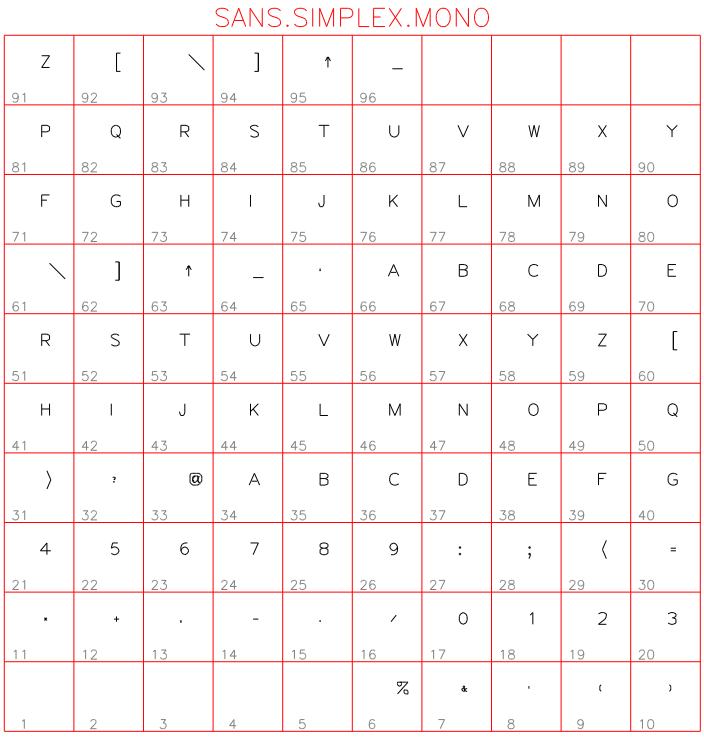


Figure 47: Sans Simplex Mono

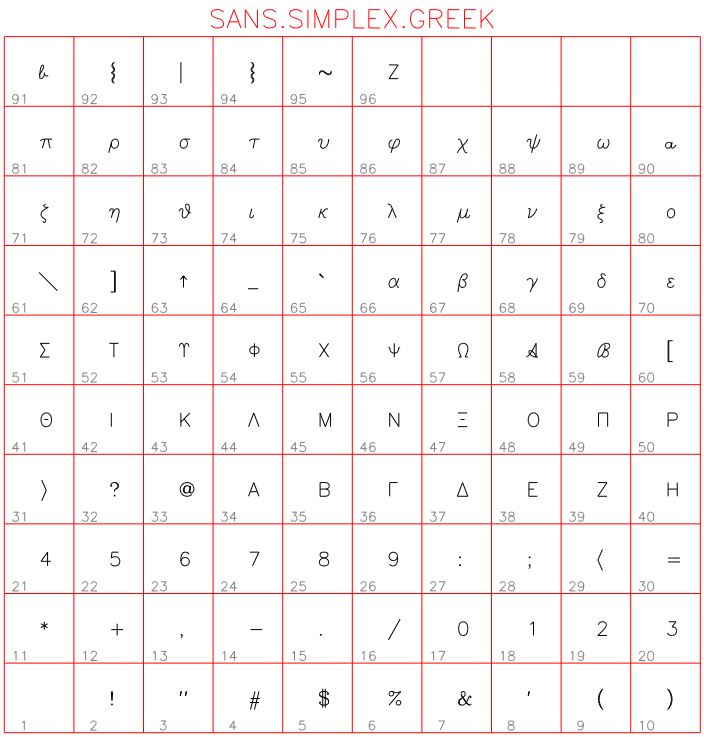


Figure 48: Sans Simplex Greek

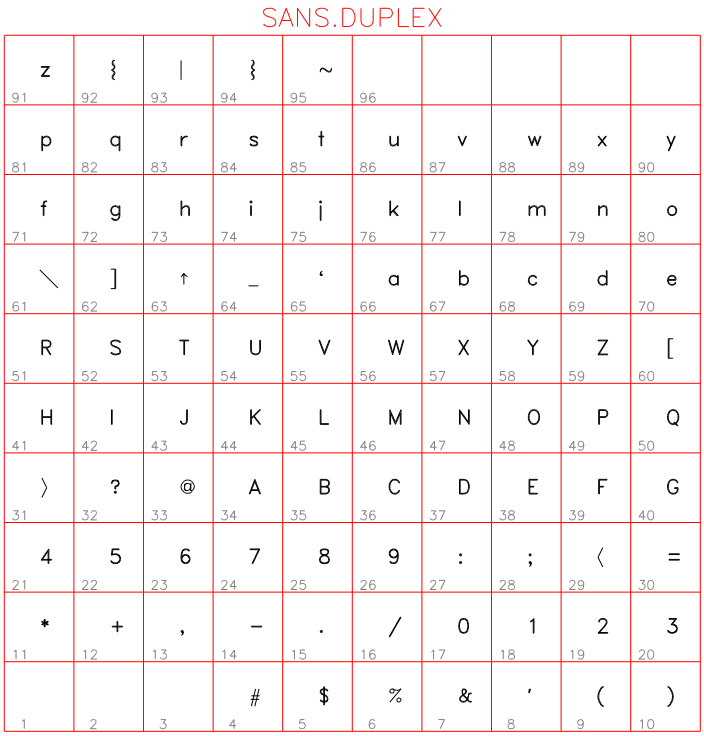


Figure 49: Sans Duplex

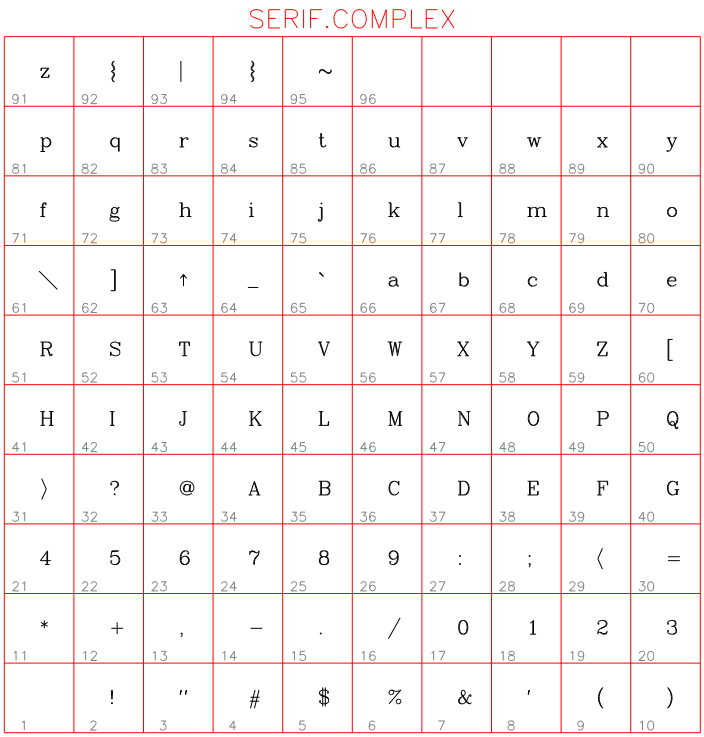


Figure 50: Serif Complex

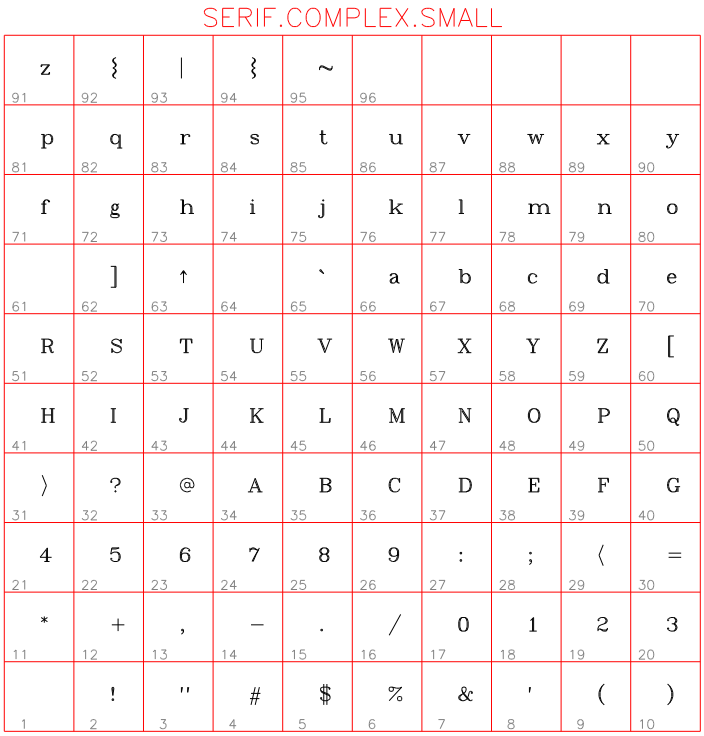


Figure 51: Serif Complex Small

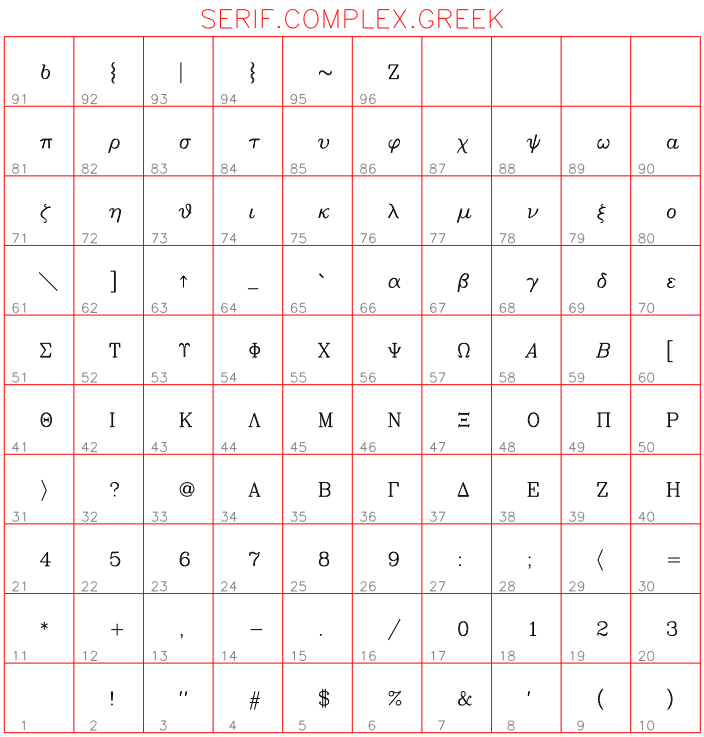


Figure 52: Serif Complex Greek

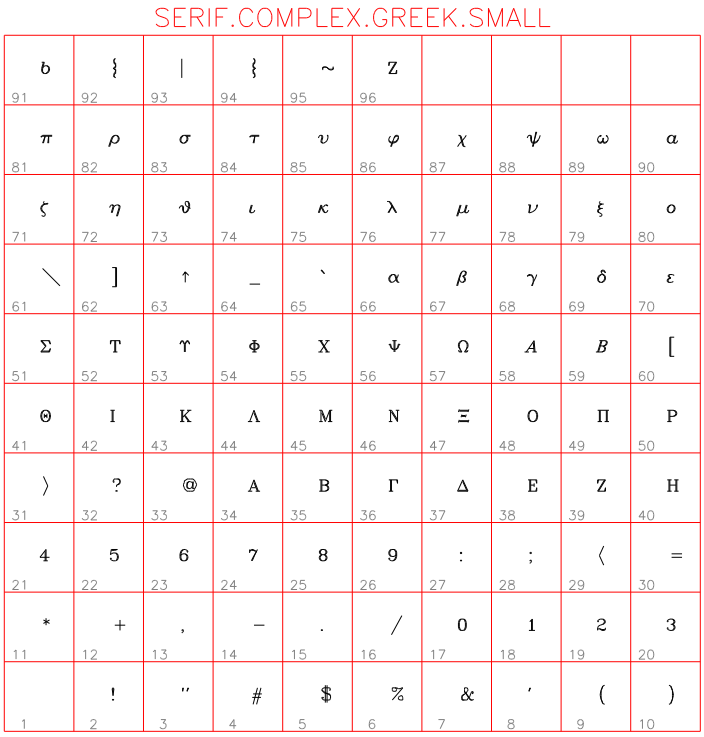


Figure 53: Serif Complex Greek Small

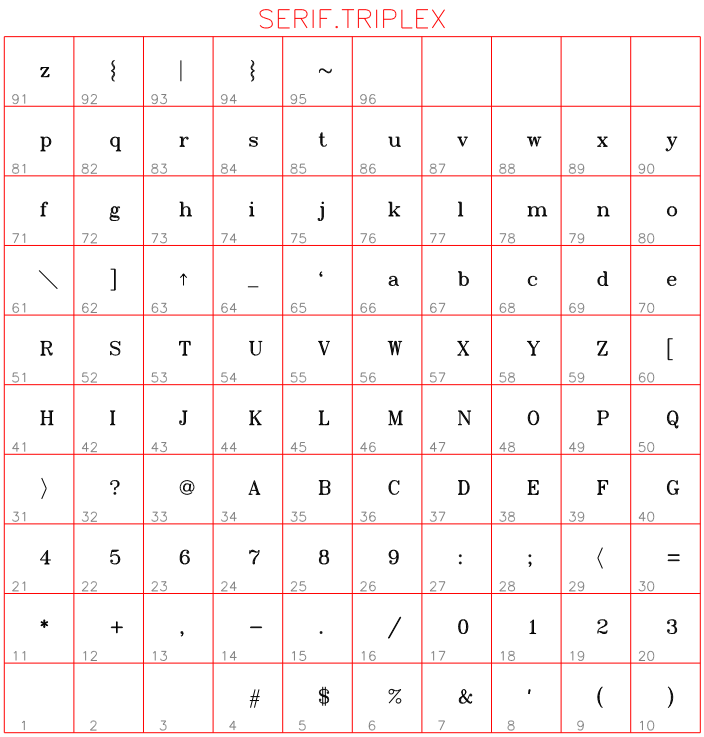


Figure 54: Serif Triplex

Figure 55: Serif Complex Italic

Figure 56: Serif Complex Italic Small

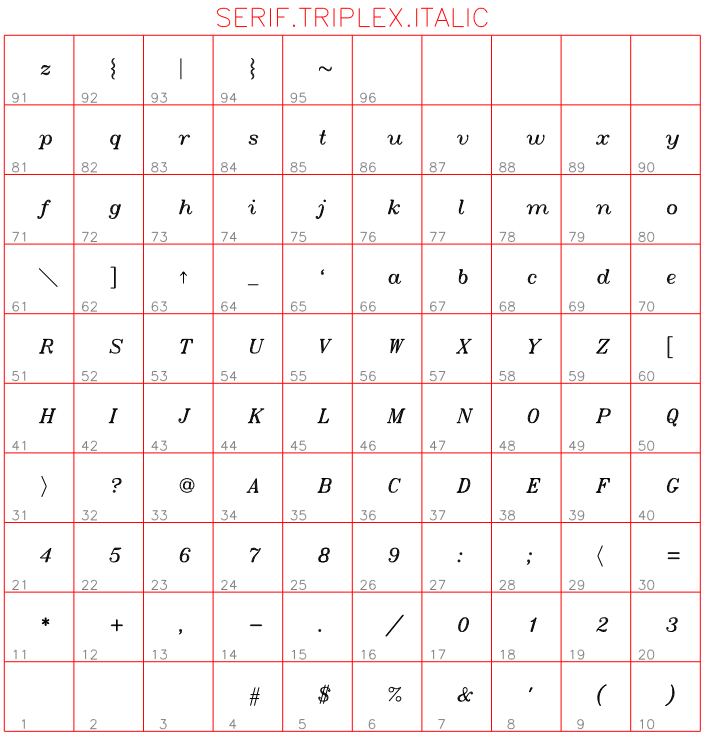


Figure 57: Serif Triplex Italic

Figure 58: Script Simplex

Figure 59: Script Complex

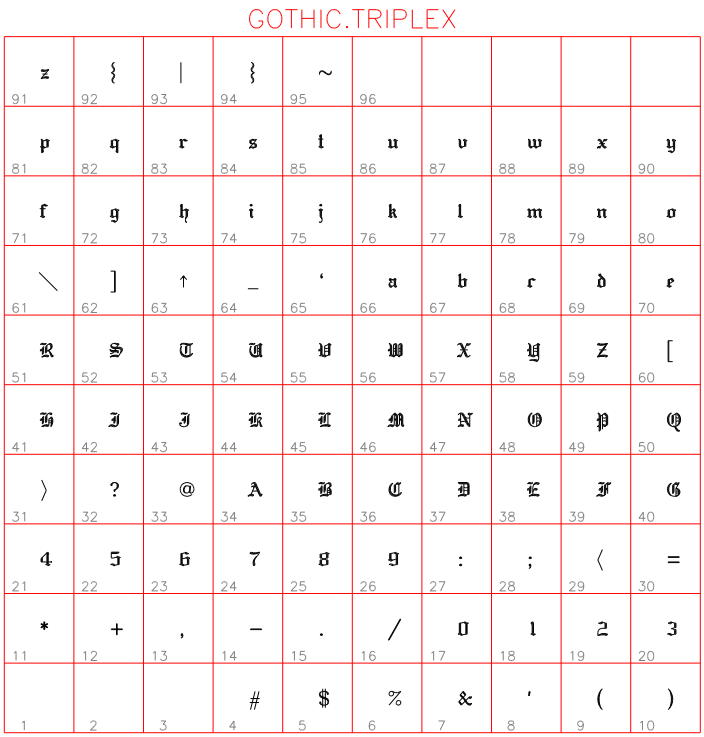


Figure 60: Gothic Triplex

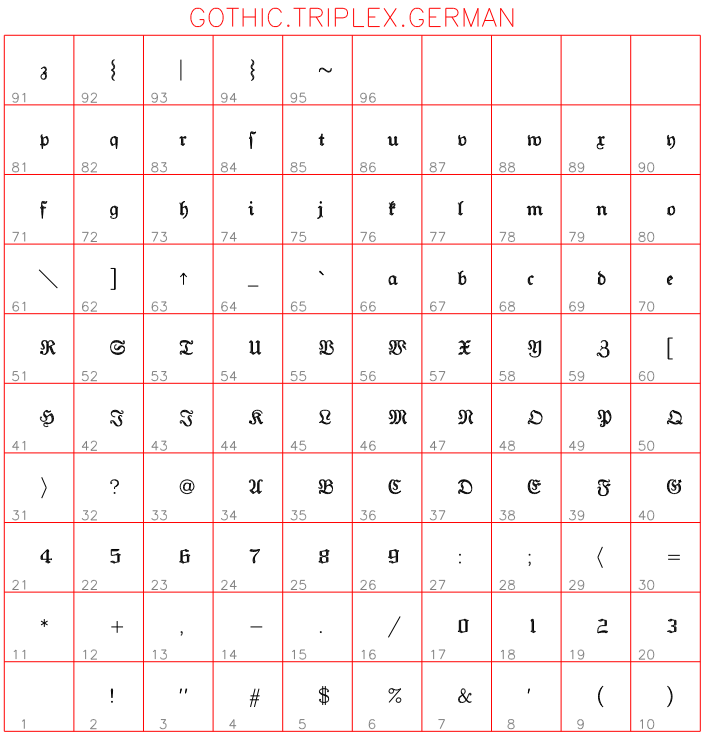


Figure 61: Gothic Triplex German

Figure 62: Gothic Triplex Italian

Figure 63: Cyrillic Complex

11.2 Symbol fonts

This script draws a symbol font table:

```
PROC 1 I,10,IOIJ,M,I,TS    ; DRAW SYMBOL I AT IJ
PROC 2 I,0.5,-,1,STO,0.5,CHS,M,1,RCL,9.5,D
PROC 3 0.5,CHS,I,0.5,-,1,STO,M,9.5,1,RCL,D
PROC 4 I,10,IOIJ,0.4,-,SWAP,0.4,-,SWAP,M,I,1,TVI
STRING 1 "(I2)"
FONT S $1
CANVAS -1 10 -1 10
CSG ALL
COLOUR 0 0 0
WIDTH 1
SYMHT 0.3
ITEVAL 1 96 1 @1
FONT 1 SANS.SIMPLEX
SYMHT 0.2
COLOUR 0.5 0.5 0.5
ITEVAL 1 96 1 @4
COLOUR 1 0 0
ITEVAL 0 10 1 @2
ITEVAL 0 10 1 @3
MOVE 4.5 9.75
SYMHT 0.4
CTEXT 0 $1
```

The argument is the font name.

Figure 64: Math symbols

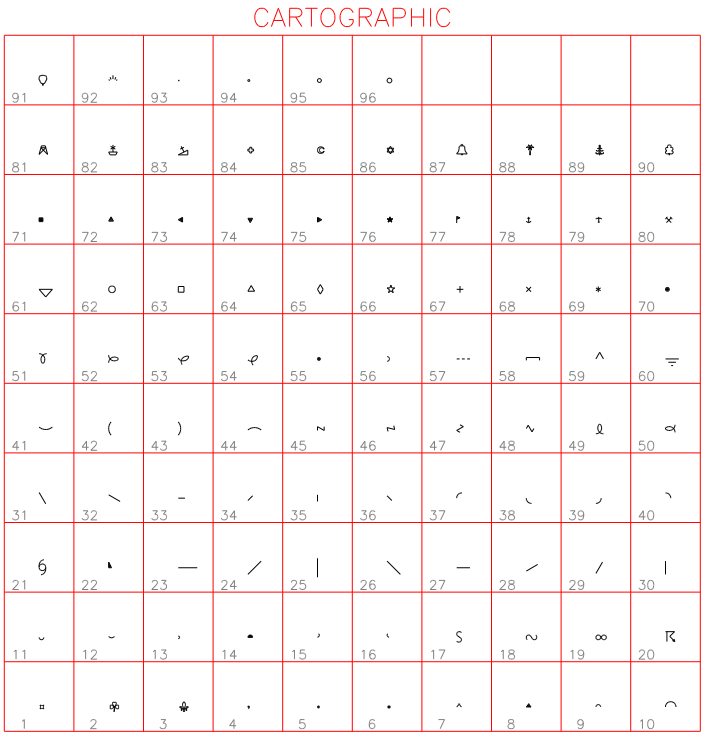


Figure 65: Cartographic symbols

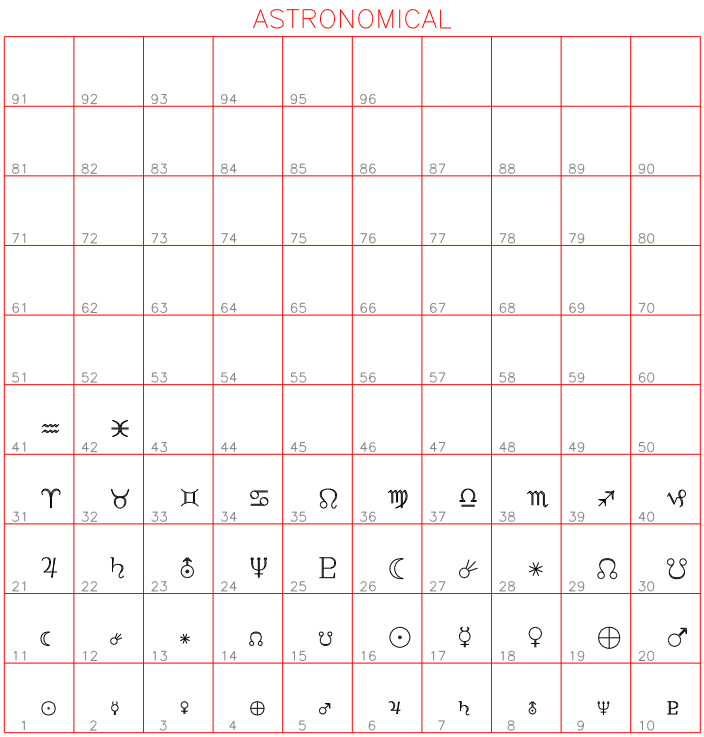


Figure 66: Astronomical symbols

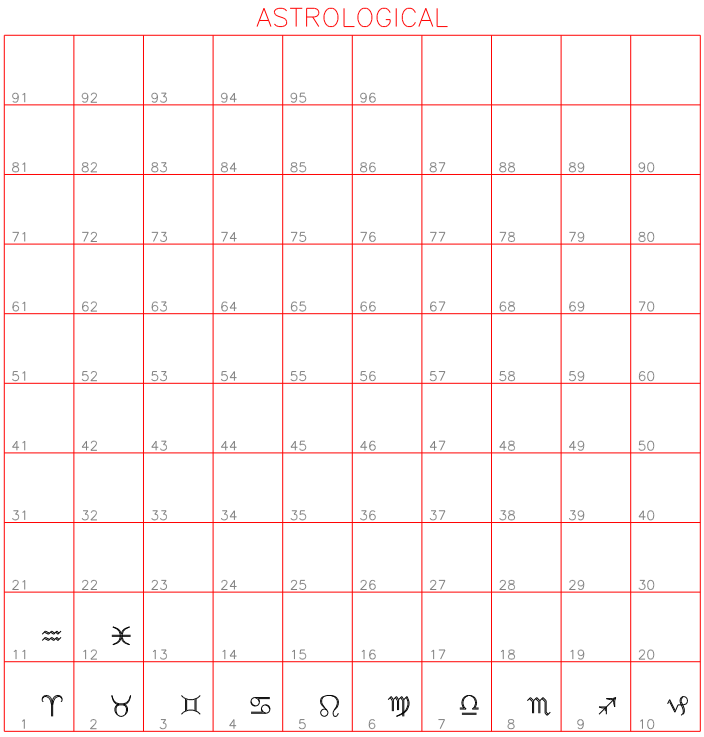


Figure 67: Astrological symbols

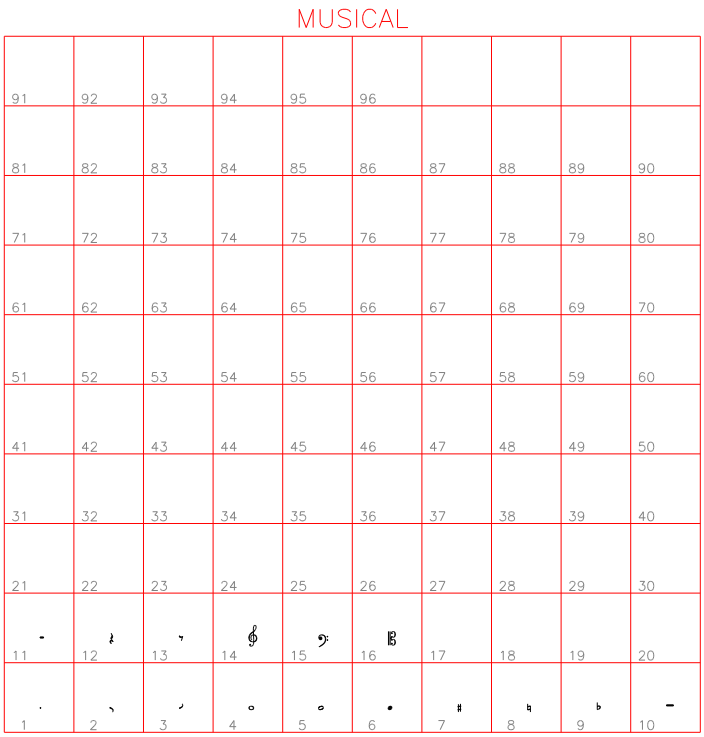


Figure 68: Musical symbols

11.3 Marker font

This script draws a marker font table:

```
PROC 1 I,10,IOIJ,M,I,TM    ; DRAW MARKER I AT IJ
PROC 2 I,0.5,-,1,STO,0.5,CHS,M,1,RCL,9.5,D
PROC 3 0.5,CHS,I,0.5,-,1,STO,M,9.5,1,RCL,D
PROC 4 I,10,IOIJ,0.4,-,SWAP,0.4,-,SWAP,M,I,1,TVI
STRING 1 "(I2)"
FONT M $1
CANVAS -1 10 -1 10
CSG ALL
COLOUR 0 0 0
WIDTH 1
SYMHT 0.3
ITEVAL 1 96 1 @1
FONT 1 SANS.SIMPLEX
SYMHT 0.2
COLOUR 0.5 0.5 0.5
ITEVAL 1 96 1 @4
COLOUR 1 0 0
ITEVAL 0 10 1 @2
ITEVAL 0 10 1 @3
MOVE 4.5 9.75
SYMHT 0.4
CTEXT 0 $1
```

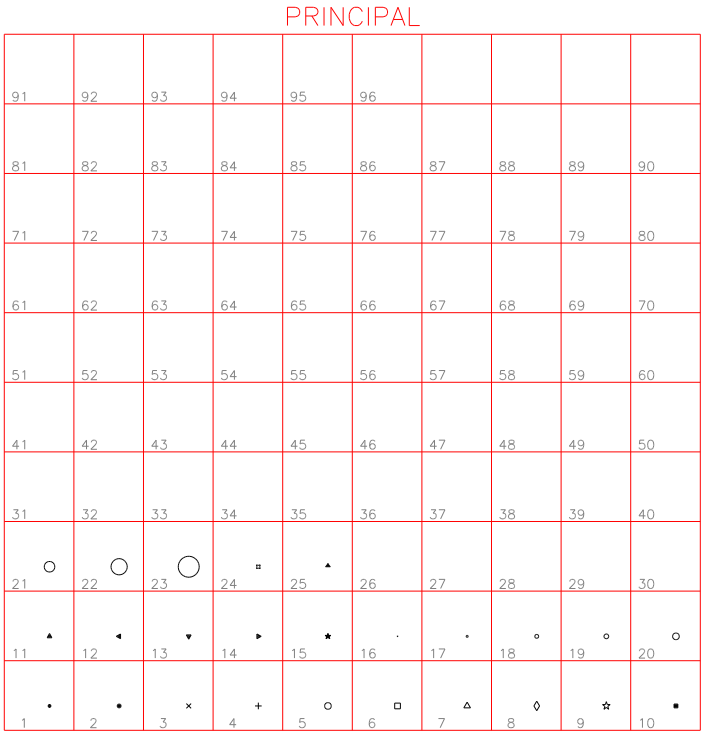


Figure 69: Principal marker font

The argument is the font name.

12 Ancillary Tools and Notes

GPLOT comes with some tools which may be helpful when using it, and perhaps more generally. When using **GTerm**, but especially when using a Tektronix 401x emulated terminal, it is useful to set certain things on “Unix-like” systems to get the best results. Some notes on that can be found here.

Installation instructions for the tools can be found in the Github `README.md` for the **GPLOT** project.

12.1 EPSView

The script `epsview.sh` can be used to display Encapsulated PostScript (EPS) files created by **GPLOT**. This is particularly useful on recent versions on macOS, where the ability to display EPS files simply by clicking on them has been removed by Apple in recent versions, but it also can be useful on Linux.

The most basic usage is:

```
$ epsview.sh epsfile
```

On macOS, this will open a new `Preview` window or tab displaying the contents of the given EPS file after it has been converted to PDF by Ghostscript. The supplied name need not be given the extension `.eps` – this will be added if it isn't there.

On Linux, `xdg-open` is used to open the EPS file (after conversion to PDF). This should find an appropriate application to display the converted PDF file.

The following options are available:

- `-p` — Also output a PNG image file version of the EPS file. This will be named as the input file but with a `.png` extension.
- `-c` — Try to remove (crop) any "margins" from the EPS document.
- `-f` — Apply "fixup" heuristics to EPS files generated on COS. Unfortunately, these become upper case when transferred from COS to NOS. No transfer options have been found that prevent this (so far), so try to correct for this. This is only relevant to **DIMFILM** EPS output from COS.

12.2 SVGView

The Python program `svgview` is an "as minimal as can be" SVG file viewer which can be used to have a quick look at SVG files created by **GPLOT**/**DIMFILM** (and most others, probably). This is based on PySDL2 and is comparatively lightweight (relative to a web browser or Inkscape, anyway!).

To display an SVG file in a pop-up window:

```
svgview afile.svg
```

(the extension need not be `.svg` or present at all). There is only one option: `-w n` which lets the width of the window (and also the resolution at which the SVG is rasterised) be set to `n` pixels. The default is 1280.

A trivial Bash script uses this program to display every `.svg` file in the current working directory in turn. This can help with verifying correct operation of **GPLOT**/**DIMFILM** as explained below.

12.3 NOSFTP

If EPS and SVG graphics files are generated on NOS, it might be necessary to move them to other "mainstream" systems for further use. NOS 2.8.7 has a very good FTP server and this is probably the most straightforward way of moving the files between NOS and "Unix-like" systems.

It is possible to use any FTP client to communicate with the NOS FTP server, and this is fairly straightforward. However, the NOS file system has some very unusual features not seen on "mainstream" systems and extra information must be sent when accessing NOS files. This extra information is easy to forget. NOSFTP is a small Python program that adds the extra information "behind the scenes" so you don't have to remember it.

To run NOSFTP:

```
$ nosftp [options] username hostname
```

Here, `username` is the account name on NOS you want to access and `hostname` is the name of the host running DtCyber (or its IP address). Before logging in, you will be prompted to enter the password for the NOS account and the NOS FTP server will be contacted. After that, you will be prompted to enter commands. The following commands are available:

- `quit` or `exit` or `bye` — Close the FTP server connection and exit NOSFTP.
- `get nosname localname ascii|display|binary` — Get a file from the NOS FTP server. You should use `ascii` if transferring EPS or SVG files so that lower case is correctly preserved.
- `put localname nosname [display]` — Send a file to the NOS FTP server. This command tries to be "intelligent" (which may have been a mistake). If NOSFTP "thinks" the file is binary data, it will save it as a *direct access* file on NOS, otherwise it will save it as an *indirect access* file. For non-binary files, it will send the file to be stored as ASCII unless the optional `display` option is supplied, in which case it will be stored as Display Code (with any lower case converted to upper case). If sending `OBEY` files, you should use the `display` option.
- `bput localname nosname [direct]` — Send a binary file (you determine it is binary rather than NOSFTP guessing it is). By default, this is stored on NOS as an *indirect access* file. To store it as a *direct access* file, add the direct option.
- `mget listname display|ascii extension` — Given a list of files, one per line in a file called `listname`, fetch each file in turn. All the files will be treated in the same way, so they must be of the same "kind". The files must not be binary. The character set for every file can be `display` (upper case only) or `ascii` (mixed case). On the receiving end, all the files will be given the `extension` specified. The names given in the file `listname` must be valid NOS file names. Files with invalid NOS names are skipped, but the rest of the transfers will take place.
- `mput listname display|ascii` — Given a list of files, one per line in a file called `listname`, send each file in turn. The file names in `listname` must be valid NOS file names. No name transformations are made, so the files to be sent must have the same names on the "Unix-like" system with no extensions. The files must not be binary. If lower case is to be preserved, the `ascii` option should be used. For `OBEY` files, use the `display` option.
- `dir` — Show a compact list of all permanent files stored on the NOS account.
- `info nosname` — List full information on a single NOS permanent file.
- `del nosname` — Delete a NOS permanent file.
- `lpwd` — Print the local working directory name.
- `lcd localdir` — Change the local working directory.
- `ls [string]` — List the local working directory, optionally restricting output to file names containing string.

The following command line options are available:

- `-e "NOSFTP command"` — Execute the supplied command, then exit.
- `-p password` — Use the plain text password to login to the NOS FTP server. Terrible for "security" but may be practically useful.
- `-d 0|1|2` — Set the debug level to help diagose problems. Should not be necessary!

12.4 SGFormat

SGFormat is a RUNOFF style document formatter. It was written to try to format the original DIMFILM manual by working from the `.SG` version of the documentation.

The DIMFILM manual was held in three different formats. Two of these were binary: one for an early version of Microsoft Word (.msw extension) and one for some version of Wordstar (.ws2). The files date from the mid-1980's and I have not been able to find any software which can read them. Perhaps the appropriate versions of Word and WordStar still exist somewhere and could be run under something like DOSBox but there is no clear way of using these files today. This is (another) lesson in the fragility of digital data, even when significant trouble is taken to preserve it (e.g. through use of multiple different data formats).

However, the third format was of the "text with mark-up" type and was definitely related to RUNOFF. Again, though, I have not been able to locate any software which can process this specific version of RUNOFF-like mark-up. The file extension (.SG) doesn't seem to be known in relation to any kind of document processing software. Some 40 years ago, though, it must have been.

As a readable text based format, it was the obvious choice from which to try to recreate the DIMFILM manual. A first attempt to do this in 2008 with a C program gave results which were marginally useful but not really acceptable. This showed that a pretty complete implementation of a document formatter was needed to get decent results. Fortunately, that is much easier using Python in 2025 than it was using C 17 years earlier.

Although SGFormat exists solely because the DIMFILM manual needed to be recreated, it could be used to create new documents in the unlikely event anyone wants to do that. The output is plain text and PDF (via PostScript).

Mark-up in this riff on RUNOFF consists of two types:

- Full line commands, beginning with a dot or period: `.` We'll call these "body commands" below.
- Commands embedded in lines of text. These are single letters following a dollar sign: `$`. We'll call these "line commands" below.

Characters which need to be escaped are preceeded by two tildes: `~~`

It should be said that, since there is no information available on the `.SG` format anywhere (that I have found), it isn't certain what the commands should really do! However, what we have chosen

to do for them seems to give a reasonable result for the **DIMFILM** manual. Also, although there are many options that can be set (see below), and these work as intended at their defaults, no real tests have been done on what happens when these are set to other values.

12.4.1 Body commands

The following body commands are implemented. The commands maybe be in lower or upper case.

- `.` — Output a blank line if a single dot is the only thing on an input line.
- `.justify` — Turn on justification for smooth (not ragged) right edge.
- `.part` — Begin a new Part of the document.
- `.chapter` — Start of a new Chapter within a Part.
- `.section` — Start of a new Section within a Chapter.
- `.subsection` — Start of a new Sub-section within a Section.
- `.appendix` — Start of a new Appendix. As per Chapter, but keep counts separate.
- `.foot` — Start of Footer text definition.
- `.endf` — End of Footer text definition.
- `.head` — Start of Header text definition.
- `.endh` — End of Header text definition.
- `.tabset [n,n,...n]` — Define a new set of tab stops or revert to default if no stops given.
- `.display` — Start of a literal section for ASCII art or code that should not be split across pages. The text in such a section is centered horizontally, based on the maximum length of a line in the section. Tab commands should not appear in lines in such sections.
- `.endd` — End a `.display` section.
- `.copy` — Start a literal section, as per `.display`, but do not centre the contents horizonatlly and process tab stops if present.
- `.endc` — End a `.copy` section.
- `.page n` — Set the page number to integer `n`.
- `.newpage` — Start a new page.
- `.blank [n]` — Output n blank lines, or 1 if `n` omitted.
- `.contents` — Insert table of contents in the output here.

The following body commands may appear, but are (intelligently) ignored: `.macro`, `.endm`, `.nl`, `.fill`, `.contig`, `.ee`, `.tt`, `.space`, `.footlength`, `.headlength`, `.set`, `.parspace`, `.table`, `.endtable`.

12.4.2 Line commands

These appear within “normal” (i.e. not body command) lines of input text.

- `(space)` as first character on line — Paragraph break.
- `$T` — Go to next tab stop in a `.copy` section.
- `$T` — If first thing on an input line outise a `.copy` section, indent by body tab size (see below).
- `$I` — Indent by body indent size if not in a `.copy` section (see below).
- `$C` — Centre the following text to the end of the line or $E. Do not use in a `.copy` section.
- `$E` — Right justify the following text to the end of the line.
- `#` — Insert a non-removeable space.
- `~~#` — Insert a hash character.
- `~~_` — Insert an underscore. Unescaped underscores are deleted.
- `~~$` — Insert a dollar sign.
- `~~~` — Insert a tilde. To avoid inadvertantly escaping a following $, add a space between the `~~~` and $.
- `~%date` — Insert a specifed date (see below) or today’s date if not explicitly set.
- `~%page` — Insert the current output page number. Negative page numbers are output as Roman numerals.

Note that `$C` and `$E` can be used on a single line to split it into 3 sections: left justified, centered and right justified. This is useful when defining a footer. For example:

```
.foot

DIMFILM$c~%page $eDRAFT DOCUMENT
Preliminary User Guide $e~%date
.endf
```

This will put `DIMFILM` at the left (left justified), the page number in the centre, and `DRAFT DOCUMENT` right justified (and on a second line, `Preliminary User Guide` left justified and a date right justified) at the foot of every output page.

12.4.3 Control file

A document is described by a JSON format control file. This defines the source files that make up the document and all processing options to be applied to it.

The following keys must be present in a control file.

- `source_files` — The value for this must be a list naming the input files.

- source_dir — The value for this is a string naming the directory where the source files can be found. This is pre-pended to each source file name before trying to open it.
- output_stem — Set the name of the output file, omitting any extension. A plain text file with the extension .txt and a PDF file with extension .pdf will be created based on this.
- output_dir — The value for this is a string naming the directory in which to write output files.

There are many more optional keys.

- page_lines — The value should be a string representing a decimal integer, giving the number of lines on a page, not including header and footer lines. Default is 58.
- line_width — The value should be a string representing a decimal integer, giving the number of characters in a line across the page. Default is 80.
- body_tab_size — Set the tab size for $T outside .copy regions. Default is 6.
- body_indent_size — Set the indent size for $I. Default is 3.
- min_justify_words — Set the minimum number of words on an output line above which right flush justification will be done. If this is done with too few words, very large unsightly gaps appear between the (few) words present. Default is 8.
- ps_page_height_points — Set the PostScript/PDF page height in points. Default is 842 (A4).
- ps_page_width_points — Set the PostScript/PDF page width in points. Default is 595 (A4).
- ps_top_margin_points — Set top margin height for PostScript/PDF in points. Default is 54 (0.75 inch).
- ps_left_margin_points — Set left margin width for PostScript/PDF in points. Default is 54 (0.75 inch).
- ps_text_size_points — Set the character height for PostScript/PDF in points. Default is 10.
- ps_leading — Set the spacing between lines for PostScript/PDF in points. Default is 3.
- ps_font_name — Set the name of the font to be used for PostScript/PDF. Default is Courier. This should be a monospaced font and allowed values will be operating system dependent.
- ps_paper — Set the paper size by name for PostScript/PDF output. Known values are a4, letter and legal. Setting this overrides any previously specified ps_page_height_points and ps_page_width_points values.
- specific_date_string — Set a specific date to be output when ~%date is used. The format is YYYY,MM,DD. If this option is not used, today's date will be output.

- `default_tabset` — The value should be a string containing comma separated integers in ascending order. For example: "`10,15,20,35`". These tab stops will be used throughout the document unless the `.tabset` body command is used.

As a concrete example, this is the control file for the **DIMFILM** manual.

```
{
    "source_files" : [
        "DIMFORM.SG",
        "DIMINTRO.SG",
        "DIMPART1.SG",
        "DIMPART2.SG",
        "DIMPART3.SG",
        "DIMPART4.SG",
        "DIMAPP1.SG",
        "DIMAPP2.SG",
        "DIMAPP3.SG",
        "DIMAPP4.SG"
    ],

    "source_dir" : "../../historic/DIMFILM/DOCS",

    "specific_date_string" : "1984,06,23",

    "output_stem" : "dimfilm-manual",

    "output_dir" : "."
}
```

12.4.4 Running SGFormat

To run this tool (after it has been installed) use:

```
sgformat [-h] [-d] ctrlname
```

where `ctrlname` in the name of the JSON format control file for the document.

The `-d` option turns on debugging mode and should not normally be used.

12.5 Terminal configuration on "Unix-like" systems

By default, "terminals" on "Unix-like" systems are assumed to have certain characteristics defined by the terminal type `xterm-256color`. This is based on the DEC VT-100 escape sequences with many extensions. Users very rarely need to know about their "terminal type" these days, but neither **GTerm** nor Tektronix 401x conform to the `xterm-256color` "standard" (to say the least). **GTerm** largely just ignores extraneous escape sequences it receives, but Tektronix 401x terminals (with some emulators) may be more sensitive. In both cases, things work better if the terminal is explicitly configured.

Although there are many alternative approaches, using Bash rather than zsh and setting the prompt to something minimal is recommended. For example:

```
$ bash
$ export PS1='$ '
```

12.5.1 Tektronix 401x

```
$ export TERM=tek4014
$ tset -e ^H
```

12.5.2 GTerm

```
$ export TERM=dumb
$ tset
```

12.6 A stripped down telnetd for "Unix-like" systems

The **DtCyber** simulator provides terminal access through a built-in telnet server. Most other historic computer system simulators do the same for terminals. However, **GPLOT** can run on "Unix-like" systems and it is often useful to run it with **GTerm** (or, maybe, a Tektronix 401x emulator). In the past, that hasn't been a problem, because "Unix-like" systems came with a telnet server (of course). However, there are now two issues:

- Some systems (e.g. macOS) no longer ship with telnet servers at all, for "security" reasons.
- Telnet servers have been disfavoured for over a decade now, since ssh is a much better choice for most purposes — but not for all. Maintaining them has been low priority, and this has shown in terms of bugs that come and go from release to release. The standard `telnetd` is also quite complex — much more so than it needs to be for most likely applications these days.

To get around these issues, we maintain a fork of **mini-telnetd** which can be built on today's versions of macOS and Linux. This is a very barebones telnet server which nonetheless does everything needed for using **GTerm** on macOS and Linux.

On macOS, **mini-telnetd** can be run as follows (note `sudo` is not required):

```
$ mini-telnetd -p 2323 -l /usr/bin/login
```

On Linux, **mini-telnetd** must be run as `sudo`. However, the login program binary location need not be specified, as the default `/bin/login` is correct there.

```
$ sudo mini-telnetd -p 22323
```

12.7 Building documentation on a “Unix-like” system

The **DIMFILM** and **GPLOT** PDF manuals are stored “pre-built” in the Git repository. If you want to rebuild them, the directories `doc/dimfilm` and `doc/gplot` contain `build.sh` scripts that will build them from source. For the **DIMFILM** manual, the **GPLOT** tools must have been installed and the Python `venv` used for that installation must be active (see `README.md`).

For the **GPLOT** manual, in addition to the **DIMFILM** requirements, a complete TeX/LaTeX environment must have been installed. This is done by installing **TexLive** on Linux or **MacTex** on macOS.

12.8 Verifying GPLOT/DIMFILM is working correctly

The "test suite" for **GPLOT** (and, indirectly, **DIMFILM**) consists of the obey files that create the figures in this document. These aren’t exhaustive tests, of course, but if the full set of Figures can be correctly generated on a system for all four supported devices, it is likely that (almost?) everything is working correctly.

To aid with verification, there are two obey files which generate all the figures in EPS or SVG format. These are `OBALEPS` and `OBALSVG`, which both rely on `OBALTST`.

To run these on NOS, use:

```
ATTACH,GPLOT.
GPLOT,GET=YES,SAVE=YES,OBEY=OBALEPS.
GPLOT,GET=YES,SAVE=YES,OBEY=OBALSVG.
```

You may want to fetch the result files to some system on which they can be viewed after each step, as the `OBALSVG` stage will overwrite the EPS files generated by `OBALEPS` (the output files have the same names – NOS file names have no extensions and are too short to add anything meaningful to distinguish EPS from SVG).

These tests will take some time to run and consume a lot of resources, so it is recommended to use:

```
SETTL,*.
SETJSL,*.
```

to prevent running into time and resource limits.

To fetch the output files from NOS to a "Unix-like" system, you can use NOSFTP with a fetch list. This list is in a file `fetch-list` which can be recreated with `get-images-list.sh` if `OBALTST` is modified (it scans that obey file for output image information). For example, to get all SVG files generated by the above on NOS into a directory called `temp2`:

```
$ mkdir temp2
$ cd temp2
$ nosftp tester nuc1
Password for NOS account:
Contacting NOS FTP server on host: nuc1
230 USER LOGGED IN, PROCEED.
```

```
220 SERVICE READY FOR NEW USER.
Local cwd now: /Volumes/qemu-main/gitprojects/gplot2/temp2
NOS FTP> mget ../obey-files/fetch-list ascii svg
... getting: gr1001 ( 1 )
... retrieving file: gr1001
226 CLOSING DATA CONNECTION.
...
...
... getting: fin001 ( 73 )
... retrieving file: fin001
226 CLOSING DATA CONNECTION.
Got 73 of 73 files OK

NOS FTP> bye

Exiting normally.
221 SERVICE CLOSING CONTROL CONNECTION. LOGGED OUT.

$ ls
cht1001.svg f02t001.svg f05t001.svg f11t001.svg f17t001.svg
g2fm001.svg gh1b001.svg gr12001.svg gr18001.svg gr21001.svg
gr30001.svg iocd001.svg lsys001.svg cht2001.svg f03s001.svg
f06t001.svg f12t001.svg f18t001.svg g2sp001.svg gh2b001.svg
gr13001.svg gr1a001.svg gr22001.svg gr3001.svg  lsbr001.svg
f01m001.svg f03t001.svg f07t001.svg f13t001.svg fin001.svg
g3sp001.svg ghbb001.svg gr14001.svg gr1i001.svg gr27001.svg
gr4001.svg  lsgo001.svg f01s001.svg f04s001.svg f08t001.svg
f14t001.svg fosm001.svg gd01001.svg gr10001.svg gr15001.svg
gr1x001.svg gr28001.svg gr5001.svg  lsks001.svg f01t001.svg
f04t001.svg f09t001.svg f15t001.svg g1fm001.svg gdea001.svg
gr1001.svg  gr16001.svg gr20001.svg gr29001.svg gr7001.svg
lssm001.svg f02s001.svg f05s001.svg f10t001.svg f16t001.svg
g1sp001.svg gemz001.svg gr11001.svg gr17001.svg gr2001.svg
gr2a001.svg gr8001.svg  lssp001.svg
```

There is a script which will run the **SVGView** tool on all SVG files in a directory to make the output easy to examine:

```
svgviewall.sh
```

This is installed when the other tools described above are installed.

To verify correct operation on the two interactive devices, use **GPLOT** interactively, choose the device (`GTERM` or `TEK4K`) then use:

```
OBEY OBALTST
```

and inspect the output visually. The obey script pauses after each output "frame" for the user to type `(return)` (or `Q (return)` to stop the script). On "Unix-like" systems, you may need to use `PREFIX` to specify where obey files are to be found first.

To run the EPS and SVG format verification tests on "Unix-like" systems, use:

```
ugplot obey=obaleps
ugplot obey=obalsvg
```

The output files will have `.svg` or `.eps` extensions in this case, of course.

13 Acknowledgements and history

GPLOT is entirely reliant on **DIMFILM**, written by Dr. John C. Gilbert at The University of London Computer Centre (U.L.C.C.) and maintained primarily by him between 1972 and the mid-1990's. One goal of **GPLOT** is to preserve **DIMFILM**. I can find almost no trace of this excellent software elsewhere.

GPLOT uses character string parsing subroutines written by Dr. Adrian Clark circa 1985. Dr. Clark also wrote a **DIMFILM** device driver for Tektronix storage tube terminals on which the **GPLOT** Tektronix support is based.

The version of **DIMFILM** used here came to me from Dr. Gilbert via Dr. Clark in 2005. Due to a malfunctioning hardware RAID controller, the U.L.C.C. production version was no longer available and the version we have here was assembled by Dr. Gilbert from various sources. Unfortunately, it did not include the original font data. I had a working version of **DIMFILM** using Hershey fonts by 2007 (it was very much a hobby project!). Not many changes were made. The original, unmodified, source is retained in the `historic` sub-tree of the source repository.

As noted above, the original (albeit "interim") **DIMFILM** manual was preserved in several formats, including a very early version of Microsoft Word and some mid-1980's version of WordStar. I have failed to find any software which can use these files. Fortunately, the manual was also kept in a RUNOFF style format (i.e. plain text with markup). I never found a RUNOFF style formatting program that could process these files either (the files have the extension `.sg` which doesn't seem to be known to the Internet for text processing purposes). However, it was possible to write a formatter in Python from scratch to get a PDF version of this essential manual.

The first usable version of **GPLOT** (0.1, late 2013) could not output EPSF files as such, but used a limited character set format that could be trivially translated to EPSF. V0.2 (2022) improved Tektronix 401x support. V0.58 (developed between late 2022 and early 2023) added significant – if idiosyncratic – facilities for function evaluation and other programmability features, as well as SVG output. It also exposed a lot more of **DIMFILM's** capabilities and directly generated "true" EPS files. Most of what can be done with **DIMFILM** using a specially written Fortran program can now be done from **GPLOT**. V0.59 fixed some bugs in parameter substitution. V0.6 (July 2025) tidied up a few things and supported building on "Unix-like" systems as well as NOS. This version also uses the NOS MODIFY – Git inter-operability tools so that the source can be managed equally well on NOS or with Git, with changes made on one system being automatically reflected in the other. This version also automatically generates minor variations of the source code needed for building on COS (**DIMFILM** library only) and Unix, using a single, NOS compatible, code base. Many further changes and additions were made between July and October 2025 – many of them a direct result of trying to use it (e.g. to produce the figures in this manual). The additions include the "higher level drawing" functions.

The current version (0.87) may be genuinely useful (I find it to be).

14 Cheat sheet or Reference card

Very often, a reminder of **GPLOT** commands and operators is all that is needed when using it. Such a cheat sheet (or, more formally, reference card) follows. This is also available as a separate PDF file for easy printing.

GPLOT V0.87 Command Cheat Sheet

SYSTEM

DEVICE	NAME [FILE [DIMENSIONS]] – Select output device [W,H[+X+Y]]
CLEAR	[FILE] – Clear drawing / end output file, change output file name.
FILL	Fill screen with colour (device dependent)
OBEY	NAME [PARAMETER-LIST] – Read commands from file (nestable)
HELP	TOPIC – Display help on topic
STATUS	Display status information
MEMTEST	Test dynamic memory. Generate graph plotting test data
GET	NAME – Get an indirect permanent file (ignored if not on NOS)
LOGFILE	NAME – Open a command log file
MAXPOINTS	NUMBER – Set max data points in internal arrays (default 1000)
NSTACK	N – Set evaluator stack size (minimum 4, default 8)
VERSION	[N] – Version information. Print or put in string register N
WAIT	Wait for user response (enter) on interactive device (eps/svg ignored)
RESET	Reset most state to defaults. Not device or read data.
PREFIX	PATH – Set a prefix to prepend to all filenames. ignored on NOS.
EXIT	Exit GPLOT

DEVICES

GTERM	Python/Qt/OpenGL based colour graphics terminal
TEK4K	Tektronix 4010/4014 terminal
EPSCOL	NAME [W,H+X+Y] – Encapsulated PostScript file (W,H,X,Y INCH)
	... default size 5.5+0.5+0.5 inches
SVG	NAME [W,H] – Scalable Vector Graphics file (W,H PIXEL)
	... default size 800,800 pixels

COLOURS AND STYLES

CSGROUP	ALL/GENERAL/TEXT/ANNOT – Colour/style group to set
COLOUR	R G B – Set rgb colour to use, range 0 to 1
WIDTH	WIDTH – Set line width
STYLE	STYLE SOLID/DASH/DOT/DASHDOT [LEN] – Line style, dash len (all groups)

GRAPH PLOTTING

READ	NAME XCOL YCOL [YECOL [XECOL]] – Read a data file using columns.
	... NAME=HERE, use command input to EOF line
	... XCOL YCOL etc. are column numbers (1 ...) space or comma separated
XYPOINT	Draw xy graph with points
XYLINE	Draw xy graph with lines
XYHISTOGRAM	Draw xy histogram
GRMOVE	X Y – Move to graph coords (X,Y)
GRDRAW	X Y – Draw to graph coords (X,Y)
XYAUTO	Find both axis ranges automatically
XRANGE	XLO YHI – Set X axis range
YRANGE	YLO YHI – Set Y axis range
XYSAME	Keep previous X,Y axis ranges
XLINEAR	Use linear X axis
YLINEAR	Use linear Y axis
XLOG	Use log X axis
YLOG	Use log Y axis
INTVALUES	NONE/X/Y/BOTH – Try to use integers for axis values, if possible.
GRID	NONE/X/Y/BOTH – Grid graph in axis
INTERPOLATE	LINEAR/CUBIC/QUINTIC N – Interpolate with N intermediate points
ASYMYERRORS	ON/OFF – Asymmetric Y error bars (off for X & Y error bars)
HISTSTYLE	ABUT/ABUT+SHADE/LINES/WIDE WIDE+SHADE [WIDTH] – Histogram style
ANNOTATE	ON/OFF – Turn graph annotation on/off
RIGHTANNOT	ON/OFF – Turn right edge annotation on/off (drawn only if annotate off)
TITLE	TEXT [LOWER] – Set title. If lower is yes, below instead of above
XLABEL	TEXT – Set X axis label
YLABEL	TEXT – Set Y axis label
RYLABEL	TEXT – Set right edge Y axis label
GSTYLE	BOXED/AXES/OPEN – Overall graph drawing style.
	... (axes must be in range to draw, see AXCUT)
AXCUT	X0 Y0 – Set point through which axes pass (default: 0,0)
MARKER	NUMBER – Set point marker number
USEKEY	Prepare to create a key (legend) for the graph
ADDKEY	Add a key for the current graph line/points
KEYS	Draw the accumulated keys (after all lines/points)
GRAPHMODE	ON/OFF – Internally set bounds to same aspect ratio as the device.
	... if not ON, the graph appears in a centered, square, subregion
SUBFIGGRID	NX NY IX IY [SHRINK] – Set up pane to draw next graph as a sub-figure.
	... Figures arranged in NX X NY grid. this figure at IX,IY.

L-SYSTEM

LSYSTEM	NRULES NITER ANGLE – Generate and draw an L-system.
	... Axiom in STRING 1, rules in STRINGs 2-9
	... Alphabet: F A B C D E M X Y + - [ ]
F and D	Draw a unit length line segment at the current drawing angle.
M	Move by unit length at the current drawing angle.
+	Turn counter-clockwise by the turning angle.
-	Turn clockwise by the turning angle.
[	Push the current position and angle on to a stack.
]	Pop a position and angle off the stack and make them current.

BASIC DRAWING

BOUNDS	XL XH YL YH – Set plot bounds (user world coordinate system)
PANE	XL XH YL YH – Set pane (clipping) area.
	... also area in which graph will be plotted.
CANVAS	XL XH YL YH – Set bounds and pane (to be the same)
UNPANE	Stop using any pane
BLANK	XL XH YL YH – Set blank area
UNBLANK	Stop using any blank area
OUTLINE	PANE/BLANK/BOUNDS/DEVICE – Outline an area
MOVE	X Y – Move to position
DRAW	X Y – Draw to position
PATH	C/O – Draw a closed / open polyline.
	... coordinates from X,Y arrays (see over).
CIRCLE	X Y R – Draw circle, center X Y, radius R.
ARC	X Y R A1 A2 – Draw a circular arc, center X Y, radius R
	... start angle A1, end angle A2.
RECT	X Y W H – Draw a rectangle, bottom left at X Y, wd W, ht H.
CRECT	X Y W H – Draw a rectangle, center X Y, width W, height H.
FONT	SET FONTNAME – Font/symbols/markers to use.
	... SET: 1 2 3 S M for 3 alphabets, symbol, marker.
LISTFONT	List available fonts
SYMHT	HEIGHT – Text, symbol or marker height in user bounds units
SYMANG	ANGLE – Text drawing angle wrt X axis (ccw degrees)
TEXT	TEXT – Draw text (with format control)
CTEXT	WIDTH TEXT – Text horizontally centered on current XPOS,YPOS,
	... scaled to WIDTH user units (0 for no scaling, use SYMHT)
MARKER	NUMBER YES – Draw marker at current position

HIGHER DRAWING – TEXT BOXES

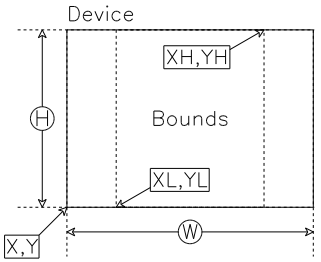
BOXTEXT	TEXT [X Y] – Draw text in a box with centre or bottom left
	... AT X,Y or at the last incremented box position. Box
	... Position increments by specified deltas after each draw.
	... Text is centered vertically and centered or flush left
	... HORIZONTALLY. Text may have up to 5 lines separated by
	... BACKSLASHES.
BOXPSIZE	WIDTH HEIGHT BOTLEFT – Set box size. BOTLEFT YES/NO sets.
	... MEANING OF BOXTEXT X,Y NO = centre, YES = bottom left.
BOXPHATCH	SPACING [ANGLE MODE] – Hatch spacing, angle (in spacings), mode:
	... NONE,VERT/HORIZ/BOTH
BOXPTEXT	WIDTHFRAC MODE FLUSHLEFT – Set how text is drawn. MODE: FIX/SCALE.
	... WIDTHFRAC: Fraction of box width for text in SCALE mode.
BOXPBOX	NONE/OUTER/INNER/BOTH – Draw the box outline, inner box around text.
BOXPDELTAS	DX DY – Automatic box position step after every box draw.

HIGHER DRAWING – DECORATED LINES & LABELS

LINE	LINESPEC – Draw a decorated polyline defined by LINESPEC
	... LINESPEC is a sequence of POINTSPECs separated by) characters.
	... POINTSPEC is (type)X,Y[,A] where x and y are bounds coords.
	... (type) is P for a plain, undecorated, location, A for an
	... arrow with automatically determined direction, or S for a skip
	... (a shape to skip over an existing line). The optional A
	... is (very short) arbitrary text to place in a label in the middle
	... of the line segment the POINTSPEC starts.
	... Example: L I N E A 3 5 , 1 6) P 3 5 , 2 1 , R T) A 2 6 , 2 1
ARROWPARM	TYPE SIZE SHARPNESS BARBEDNESS – Arrow parameters. TYPE:
	... OPEN/CLOSED/BARBED, SIZE in bounds units, SHARPNESS 0.5:5,
	... BARBEDNESS fraction of arrow head in barb.
LINEPARM	SKIPSCALE ANNOTSCALE – Scale of skips and mid-line annotations.
GLABEL	GX GY LENGTH ANGLE TEXT – Draw a label with an arrow and boxed text.
	... GX,GY are graph range coordinates to which the arrow points,
	... LENGTH is the length of the arrow line in bounds units, ANGLE
	... is the angle of the line wrt +X in degrees, TEXT is the
	... text to appear in the label box.
ALABEL	X Y LENGTH ANGLE TEXT – As GLABEL but X,Y in bounds coords.

EVALUATOR (see over for operators)

ERANGE	BASE START STOP NELEM – Set range of values in X/O
	... stack register. BASE 1 linear, else log base.
EVAL	RPN [ARG] – Execute a procedure/evaluate a function using an RPN
	... NOTATION, ARG 1,2, etc. refer to proc registers using @1
ITEVAL	START END STEP RPN – Iterated evaluation over start to end by step.
2DEVAL	RPN – As EVAL, but preserve both X and Y data (see over).
PROC	N RPN – Store RPN code string in proc register N. Use EVAL @N
LOADPROC	N NAME – Load a named procedure from GPLPROC file into proc reg N
STRING	N TEXT – Set contents of string register N
STO	N X – Store real number X in memory register N
RCL	N – Display contents of memory register N
ZEROVAL	V – Set value for divide-by-zero traps. Default 1e-9. 0=no trap.
BBSTART	Accumulate bounding box instead of drawing with M & D operators.
BBEND	Exit bounding box mode and return to drawing with M & D operators.
BBSET	[SCALE] – Set bounds to match bounding box found with BBSTART/BBEND.



TEXT STRING MARKUP CODES

*U	Go to upper case
*L ... $L	Start/end lower case
*B	Backspace
*1 *2 *3	Select a font set
*+ ... $+	Start/end superscript (2 levels)
*- ... $-	Start/end subscript (2 levels)
*N	Reset sub/superscript level to 0
*O ... $O	Sub/super fully below/above last ch
*, NUM $, DEN $.	Construct a fraction
*:nn	Output symbol with code nn
*::nn	Output marker with code nn
*Vnn	Output alphabetic with code nn
*= ... $=	Start/end underlining

COMMAND LINE OPTIONS

GPLOT,KEY=VAL, ...	To run on NOS
UGPLOT KEY=VAL	To run on Unix-like systems.
OBEY=file	Name of obey file to run
PARM=string	Parameters for obey file in dbl qts
GET=yes or NO	Auto GET files on NOS
SAVE=yes or NO	Auto SAVE files on NOS
DEBUG=yes or NO	Show expanded parameters
QUIET=yes or NO	Show expanded command
SLIDE=YES or no	Adjust graph layout for larger text

PLNOTES MARKDOWN INTEGRATION

```
) GPLOT COMMAND
) GPLOT COMMAND
) ... etc.
) ... no DEVICE or CLEAR

(blank line ends GPLOT commands)

Use:
MODEXEC,BUILD,PLNOTES,ARG=ANOTE.
... to create ANOTE.HTML
```

GPLOT V0.87 Evaluator Cheat Sheet

OPERANDS

(DIGITS)	(-- C1)	SET TOS ARRAY TO A LITERAL CONSTANT.
X	(-- A1)	SET TOS TO X RANGE ARRAY.
PI	(-- C1)	SET TOS TO PI.
E	(-- C1)	SET TOS TO E.
I	(-- C1)	SET TOS TO ITERATION NUMBER FROM ITEVAL. 1 IF IN EVAL.
IDX	(-- A1)	SET TOS TO THE ARRAY ELEMENT INDEX.
TWPI	(-- C1)	SET TOS TO 2 PI.
PI/2	(-- C1)	SET TOS TO PI / 2

STACK MANIPULATION

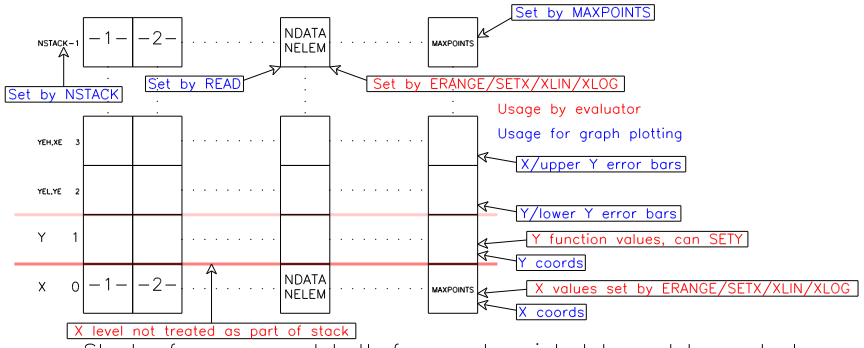
SWAP or S	(A1 A2 -- A2 A1)	SWAP OR EXCHANGE TOP 2 STACK ARRAYS.
DUP or &	(A1 -- A1 A1)	DUPLICATE TOP OF STACK
G	(.. AN .. C1 -- AN)	GET STACK LEVEL AT DEPTH C1 INTO TOP OF STACK
POP	(A1 --)	POP TOP OF STACK
CL	(.. --)	CLEAR STACK
SETX	(A1 -- A1)	OVERWRITE RANGE, GRAPH X VALUES WITH STACK TOP VALUES: X = A1
SETY	(A1 -- A1)	OVERWRITE GRAPH Y VALUES WITH STACK TOP VALUES: Y = A1
XLIN	(C1 C2 C3 --)	X = C1 TO C2 IN C3 LINEAR STEPS
XLOG	(C1 C2 C3 C4 --)	X = C1**C2 TO C1**C3 IN C4 STEPS
EL	(A1 C2 -- A1 C2)	C2 = A1(C2)
I0IJ	(C1 C2 -- C1• C2•)	C1•=MOD(C1,C2), C2•=(C1/C2): 1D TO 2D INDEX CONVERT, 0 BASED.
I1IJ	(C1 C2 -- C1• C2•)	C1•=MOD(C1,C2)+1, C2•=(C1/C2)+1: 1D TO 2D INDEX CONV, 1 BASED.

BASIC ARITHMETIC

+	(A1 A2 -- A1)	ADD ARRAY ELEMENTS: A1 = A1 + A2
-	(A1 A2 -- A1)	SUBTRACT ARRAY ELEMENTS: A1 = A1 - A2
R-	(A1 A2 -- A1)	REVERSE SUBTRACT: A1 = A2 - A1
*	(A1 A2 -- A1)	MULTIPLY ARRAY ELEMENTS: A1 = A1 * A2
**	(A1 A2 -- A1)	EXPONENTIATION: A1 = A1 ** A2
/	(A1 A2 -- A1)	DIVIDE ARRAY ELEMENTS: A1 = A1 / A2
R/	(A1 A2 -- A1)	REVERSE DIVIDE: A1 = A2 / A1
RCP	(A1 -- A1)	RECIPROCAL A1 = 1.0 / A1
CHS	(A1 -- A1)	A1 = -A1
ABS	(A1 -- A1)	A1 = ABS(A1)

MATH FUNCTIONS

SIN	(A1 -- A1)	A1 = SIN(A1)
COS	(A1 -- A1)	A1 = COS(A1)
TAN	(A1 -- A1)	A1 = TAN(A1)
ASIN	(A1 -- A1)	A1 = ARCSIN(A1)
ACOS	(A1 -- A1)	A1 = ARCCOS(A1)
ATAN	(A1 -- A1)	A1 = ARCTAN(A1)
SINH	(A1 -- A1)	A1 = SINH(A1), DOMAIN |X| (742.36
COSH	(A1 -- A1)	A1 = COSH(A1), DOMAIN |X| (742.36
TANH	(A1 -- A1)	A1 = TANH(A1), DOMAIN |X| (742.36
SQRT	(A1 -- A1)	A1 = SQRT(A1)
LOG	(A1 -- A1)	BASE E LOGARITHM: A1 = LN(A1)
LG10	(A1 -- A1)	BASE 10 LOGARITHM: A1 = LOG10(A1)
LOG2	(A1 -- A1)	BASE 2 LOGARITHM: A1 = LOG2(A1)
EXP	(A1 -- A1)	A1 = E ** A1. DOMAIN: -675.81 TO 741.66
RAND	(A1 -- A1)	UNIFORMLY DISTRIBUTED RANDOMS [0:A1): A1 = A1 RANDOM
SEED	(C1 --)	SET RANDOM NUMBER SEED TO A1(1)
GCD	(C1 C2 -- C1)	FIND GREATEST COMMON DIVISOR OF C1, C2



Stack of arrays used both for graph point data and by evaluator

NUMBER RANGE RELATED

MIN	(A1 A2 -- A1)	A1 = MIN(A1,A2)
MAX	(A1 A2 -- A1)	A1 = MAX(A1,A2)
MOD	(A1 A2 -- A1)	A1 = MOD(A1,A2) OR A1 = A1 % A2
SIGN	(A1 -- A1)	A1 = (A1 (0) ? -1 : 1

CONDITIONALS

ODD	(A1 -- A1)	A1 = 1 WHERE INT(A1) IS ODD ELSE 0
GT	(A1 A2 -- A1 A2 A3)	A3 = (A1) A2) ? 1 : 0 ;
LT	(A1 A2 -- A1 A2 A3)	A3 = (A1 (A2) ? 1 : 0 ;
LE	(A1 A2 -- A1 A2 A3)	A3 = (A1 (= A2) ? 1 : 0 ;
GE	(A1 A2 -- A1 A2 A3)	A3 = (A1)= A2) ? 1 : 0 ;
EQ	(A1 A2 -- A1 A2 A3)	A3 = (A1 == A2) ? 1 : 0 ;
NE	(A1 A2 -- A1 A2 A3)	A3 = (A1 = A2) ? 1 : 0 ;
NOT	(A1 -- A1)	A1 = (A1 == 0) ? 1 : 0 ;
SEL	(A1 A2 A3 -- A1 A2 A3)	A3 = (A3 == 0) ? A1 : A2 ;

GRAPHICS

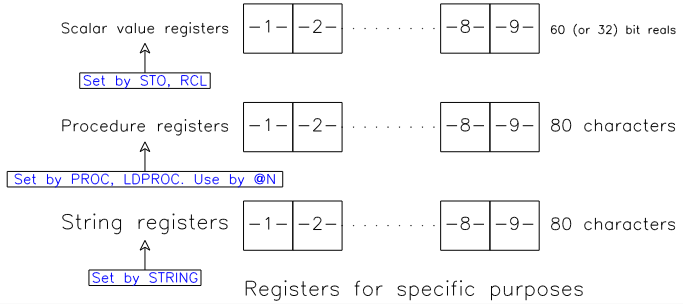
M	(C1 C2 --)	MOVE TO (C1,C2)
D	(C1 C2 --)	DRAW TO (C1,C2)
C	(C1 C2 C3 --)	DRAW CIRCLE, CENTER C1,C2 RADIUS C3.
A	(C1 C2 C3 C4 C5 --)	DRAW ARC, CENTER C1,C2 RADIUS C3, ANGLES C4 TO C5.
BOX	(C1 C2 C3 C4 --)	DRAW RECTANGLE, BOTTOM LEFT C1,C2 WIDTH C3 HEIGHT C4.
HDSH	(A1 -- A1)	DRAW RELATIVE (1,0) IF A1[I]) 0 ELSE MOVE
VDSH	(A1 -- A1)	DRAW RELATIVE (0,1) IF A1[I]) 0 ELSE MOVE
PTHO	(A1 A2 --)	DRAW POLYLINE X=A1,Y=A2, OPEN
PTHC	(A1 A2 --)	DRAW POLYLINE X=A1,Y=A2, CLOSED
T	(C1 --)	DRAW CONTENTS OF STRING REGISTER C1 AS TEXT.
TVF	(C1 C2 -- C1)	DRAW C1 AS TEXT USING FORMAT STRING IN C2.
TVI	(C1 C2 -- C1)	DRAW INT(C1) AS TEXT USING FORMAT STRING IN C2.
TS	(C1 --)	DRAW SYMBOL WITH CODE NUMBER C1
TM	(C1 --)	DRAW MARKER WITH CODE NUMBER C1
TC	(C1 --)	DRAW CHARACTER WITH CODE C1 FROM CURRENT ALPHABET
TH	(C1 --)	SET TEXT/MARKER/SYMBOL HEIGHT
TA	(C1 --)	SET TEXT DRAWING ANGLE IN DEGREES CCW
TSC	(--)	START TEXT CONTINUATION
TEC	(--)	END TEXT CONTINUATION
TLEN	C1 -- C1)	GET THE LENGTH IN BOUNDS AT UNIT HEIGHT OF STRING IN REGISTER C1.
ROT	(A1 A2 C3 -- A1 A2)	ROTATE X=A1,Y=A2 BY C3 RADIANS ABOUT ORIGIN.
TRN	(A1 A2 C3 C4 -- A1 A2)	TRANSLATE X=A1, Y=A2 BY C3,C4
SCL	(A1 A2 C3 C4 -- A1 A2)	SCALE X=A1, Y=A2 BY C3,C4
LAB	(C1 C2 C3 C4 C5 --)	LABEL AT (C1,C2), LEN C3, ANG C4, STRING REG C5

NUMBER TYPE CONVERSIONS

INT	(A1 -- A1)	TRUNCATION A1 = INT(A1)
FRAC	(A1 -- A1)	FRACTIONAL PART A1 = FRAC(A1)
FLR	(A1 -- A1)	A1 = FLOOR(A1)
CEIL	(A1 -- A1)	A1 = CEILING(A1)

SYSTEM

STO or =	(A1 C2 -- A1)	STORE A1(1) (A.K.A. C1) IN REGISTER C2
RCL or (HASH)	(C1 -- C1)	RECALL CONST VALUE FROM REGISTER INT(C1)
P	(--)	PRINT TOS ARRAY FIRST AND LAST ELEMENTS
PE	(A1 C2 C3 -- A1)	PRINT ELEMENTS C2 TO C3 OF A1 IN FREE FORMAT
PC	(C1 -- C1)	PRINT C1 (TOS A[1]) IN FREE FORMAT.
DUMP	(--)	PRINT ALL STACK LEVELS FIRST AND LAST ELEMENTS
R2D	(A1 -- A1)	CONVERT RADIANS TO DEGREES
D2R	(A1 -- A1)	CONVERT DEGREES TO RADIANS



Registers for specific purposes

Figure 70: THE END