Data Analysis & Visualization

Extrema User's Guide

© TRIUMF
4004 Wesbrook Mall
Vancouver, British Columbia V6T 2A3
Phone: 604.222.1047   Fax: 604.222.1074

Table of Contents

Chapter

1

Getting Started

An introduction to Extrema

About This Guide

This guide is an introduction to using **Extrema** for data analysis and visualization. It is not a comprehensive manual detailing all Extrema features, but rather a concise guide to accomplishing the types of tasks most commonly performed by researchers. It intended to be useful to new users, and for experienced users who would like a quick introduction to features or methods they have not used before.

In most cases, this guide instructs by example. Typical data analysis and visualization tasks are described and then performed, both using the GUI and using the scripting language. The tasks are grouped into categories that are treated in subsequent sections of this guide:

The tasks described are generally simple, but practical. Extending them to cases that are more complex is generally straightforward, although you may need to consult the **Extrema Command Reference** for further details.

Documentation that is more comprehensive is available in the following references:

1. **Extrema Command Reference**: a comprehensive guide to the Extrema command language.

2. **Extrema Online Help**: detailed online help is available through the program itself.

In many of the brief procedures described in this guide, the reader will be referred to commands and functions for more information. These commands and functions may be looked up in the above references.

Conventions used in this guide

Examples of messages and prompts written by the program, as well as examples of user typed input are displayed in `typewriter type style.`

Curly brackets, `{}`, enclose parameters that are optional and/or have default values, and indicate that it is not necessary to enter these parameters.

Parentheses, `()`, besides being used in mathematical expressions, also enclose formats.

The backslash, `\`, separates a command from a command qualifier, or a parameter from its qualifier.

Literal quote strings can be delimited by the opening quote, `` ` ``, and the single quote, `'`, or by the single quote at the beginning and the end, or by the double quote, `"`, at the beginning and the end. For example, `` `ABC' ``, `'ABC'`, and `"ABC"` are all valid literal quote strings.

Parentheses, the back slash and quotes **must** be included where indicated.

Extrema is case-insensitive, so input may be provided in upper or lower case, or with a mix of cases. In the examples in this guide, **Extrema** keywords are given in UPPER CASE, while variables and user-defined words are given in lower case, but this is simply for clarity; you do not have to follow this convention. Words that you should replace with your own variable names are given in *italics*.

Installing Extrema

Extrema is distributed as a self-extracting compressed file. You simply need to execute the `extremainstall` program to begin the installation process. After agreeing to the licensing agreement, you then select an installation directory (the default is `C:\Extrema`); and everything else is automatic. The installation program places the **Extrema** icon on the desktop. **Extrema** does not modify the registry, so to uninstall **Extrema** simply delete the files.

Running Extrema

Double-click on the **Extrema** icon to launch the program. By default, the program raises the visualization (graphics) window and the analysis (command input) window. Commands may be typed directly into the analysis window, if you are familiar with the **Extrema** command language. If not, you will probably be more comfortable selecting your actions from the menus and toolbars.

A typical **Extrema** session involves the following steps:

1. load or generate data to work on
2. graph the data, or
3. analyze the data
4. repeat previous two steps until the results are satisfactory
5. customize the presentation of the graph(s)
6. output the graph(s)
7. save data for archival purposes, or for further analysis

Once you become familiar with **Extrema**, or with your particular data analysis and presentation requirements, many of the above steps can be automated. In that case, you can build scripts to automatically perform the routine steps in the above sequence, and possibly the entire sequence itself. **Extrema**'s scripting capabilities include looping and decision-making features, so a fair amount of intelligence can be built into your scripts.

The examples in this guide include instructions for performing operations interactively using the GUI, or using the command language. Command language examples can be used interactively in the command window, or in scripts.

Chapter
2

Data Input

Before you can analyze or visualize your data, you need to get your data into Extrema in a form that can be manipulated.

Data Representation

Data is stored internally in **variables**, which have names that you use to reference the data they contain. Except for a few automatically generated variables, these names are chosen by the user. The first character of a variable name **must** be an alphabetic character, that is, `A` to `Z`, and the maximum number of characters in a name is thirty-two (32). Except for these restrictions, variable names can be any combination of: alphabetic characters (`ABC … XYZ),` digits (`0123456789`), underscore (`_`), and dollar sign (`$`).

- Variable names are case-insensitive, e.g., variable `mydata` is the same as `MyData`.
- Function names are reserved names and cannot be used as variable names.

Variables can contain character data or numeric data. Numeric data are always stored as double-precision real values.

Character (or string) variables can be one of the following types:

- **string scalar**: a simple string of text
- **string array**: an array of text strings

Numeric variables can be one of the following types:

- **scalar**: a number
- **vector**: a one-dimensional array of numbers
- **matrix**: a two-dimensional array of numbers

- **tensor**: a three-dimensional array of numbers (*to be implemented*)

The contents of arrays are indexed sequentially, with a starting index of one (1).

Except for physical memory limitations, there is no limit to the number of variables, or to the length of strings, or to the size of arrays.

Addressing parts of arrays

To refer to an entire array, simply use the variable's name.

To select an individual element from the array, provide the index of the element in square brackets:

```
x[8]          ! 8th element of vector x
y[2,6]        ! value from 2nd row, 6th column of matrix y
```

In all of the above cases, you are referring to a single value, i.e., a scalar. You can also specify a range of indices using the colon (`:`) character:

```
x[8:20]       ! 8th through 20th elements of vector x
y[1:10,1]     ! first 10 rows from the first column of y
```

It is also possible to replace any part of an index with a mathematical expression. For example:

```
x[2^3:10*2]           ! 8th through 20th elements of vector x
y[1:sqrt(100),1]      ! first 10 rows from the first column of y
```

Variables can also be used in indices. For example, suppose you have a vector `z` which holds the values `1, 2,..., 10`. The following are then valid:

```
x[z[2]^3:z[#]*2]      ! 8th through 20th elements of vector x
y[z,1]                ! first 10 rows from the first column of y
```

Such expressions can result in scalars, arrays, vectors, or matrices, depending on the number of dimensions of the result.

SPECIAL
`*` - all indexes
`#` - the last index

The special characters `*` and `#` are also available for use in indices. For example:

```
x[*]             ! all values from vector x
x[#]             ! the last value from vector x
x[#-1]           ! the next to last value from vector x
m[*,*]           ! all rows and all columns of matrix m
m[*,#]           ! all rows and the last column of matrix m
m[*,1:#-1]       ! all rows and all but last column of matrix m
```

Constants

You can type numeric values or constants anywhere a scalar variable or value is expected.

Constant arrays are expressed as a list of values inside square brackets. When typing out vector or matrix values, separate successive indexes with a comma, and successive values within an index with a semicolon.

```
5.03E-8                    ! scalar value
[1;2;4;8]                  ! vector with 4 values
[1;0;0, 0;1;0, 0;0;1]      ! 3 by 3 identity matrix
```

You can also use the `[start:stop:step]` notation to specify regular sequences of values with which to fill the variable:

```
[0:2*pi:0.1]       ! vector from 0 to 2π in steps of 0.1
[10:-10:-2]        ! descending sequence from 10 to –10 in steps of 2
```

Expressions

Extrema allows you to use mathematical expressions anywhere it would expect a variable or value, provided the expression evaluates to the expected type. Simple expressions involving dimensioned variables generally return a value of the same dimension. Thus, if `x` has `10` values, then the expression `sin(x)+1` also has `10` values. Other examples:

```
m[x,#-2:#]         ! the rows denoted in x, and the last 3 columns of m
x*m[n,*]           ! x times the nth row of m
SIN(a+b)           ! the sines of the sums of respective values in a and b
x^2*SIN(x)+1       ! a non-linear function of the values in x
```

There is no limit to the length or complexity of a mathematical expression in **Extrema** .

You can also index the results of an expression, e.g.,

```
(SIN(x)+1)[4:8]         ! selects 4th through 8th values of the expression
```

Reading Data From Files

In most cases, your data will be contained in files. You will need to read these files into **Extrema** variables before you can operate on the data. **Extrema** is quite flexible in allowing you to read files of different formats, although in more complex cases you will need to know the details of the file's data format.

Text files

Text files are human-readable, that is, they contain data written in ASCII format, and they can be viewed or edited with a simple text editor such as Notepad ®. Many spreadsheets can export their data into such a format.

If your text file contains data arranged in rows and columns, with columns delimited by commas or white space, then reading the file is simple.

To read each column into its own vector:

```
READ file1.dat x y                ! read 2 columns into vectors x and y
READ file2.dat a a_err b b_err    ! read data and errors into vectors
```

To read all columns into a single matrix, you must also specify the number of rows in the matrix:

```
READ\matrix file3.dat m nrows   ! read file into matrix m
```

There are also options to read matrices in other ways, for example, by specifying the number of columns in the matrix. See the `READ` command for more information.

Binary files

Binary files are not human readable; if you attempt to view or edit them, you will see a lot of garbage. They contain sequences of numbers or other values in raw machine format. To read this data, you need to know the details of the file format, that is, the sequence and types of data written to the file.

Generating Data

Commonly, you will need to create data spontaneously. In simple cases, you can type in the data directly. Usually, however, you will be working with data sizes that make this approach too tedious. There are numerous methods you can use for bulk data generation.

Sequences

Simple sequences can be generated using the `[start:stop:step]` array notation.

```
pi = ACOS(-1)     ! define scalar pi
X = [0:pi:.01]    ! make a sequence of values from 0 to π in increments of 0.01
```

You can create a regular sequence of values using the `GENERATE` facility. The generated data can be specified using any of the following methods:

- minimum value, maximum value, number of values

- minimum value, maximum value, step size
- minimum value, step size, number of values

You can also request random values instead of a regular step size.

Functions

By applying an expression to an already-existing variable, you can generate a new variable in which every element of the input variable has been modified by the expression. Capture this data in a new variable by simply setting the new variable to equal the expression:

```
y = 10*SIN(x)            ! x is a vector of values
```

If your source data is a monotonically increasing sequence (see above) that serves as the dependent variable, then you will get a fair representation of the function itself over that range. For instance, to produce data representing the function `SIN(x)` over the range 0 to 2π:

```
pi = ACOS(-1)
x = [0:2*pi:0.01]
y = SIN(x)
```

Interpolating Data

You may have a sparse sampling of data that you wish to fill in by interpolation. These techniques are described later, in **Chapter 5: Data Analysis Examples**.

Chapter

3

Drawing Graphs

Extrema can produce a wide variety of graph types, not all of which are described here. This section reviews the types of graphs that are most commonly used, and how to make them. For complete details on the variations that are possible, consult the ***Extrema Command Reference Manual*** *for the appropriate commands*

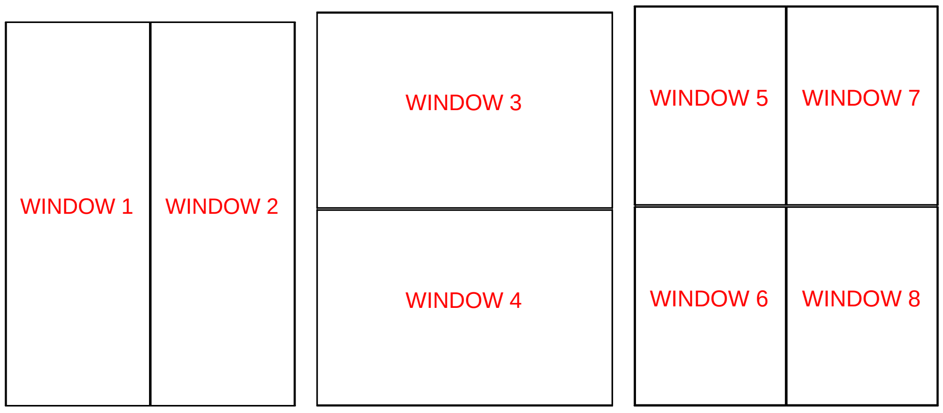
Graphics sub-windows

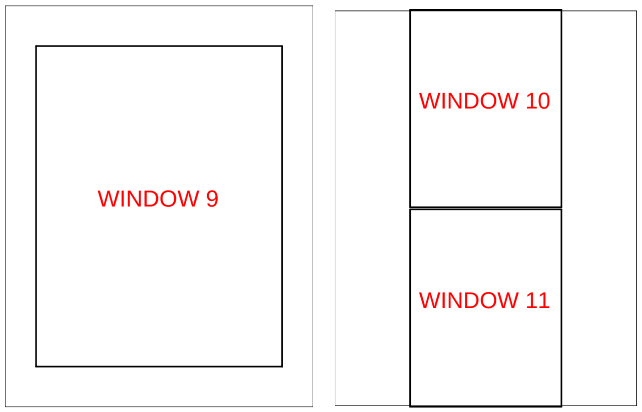
Use the `WINDOW` command to choose and/or define a graphics sub-window. Graphics sub-windows are an easy way to subdivide the graphics output page into rectangular regions, allowing multiple graphs and/or multiple figures and/or multiple text regions. A window is a subset of the page. A window, other than the default zero level window, has a smaller plotting unit range than the full page.

Commensurateness is never lost in a sub-window.

Pre-defined windows

Some of the initial pre-defined windows in `PORTRAIT` orientation are displayed below.

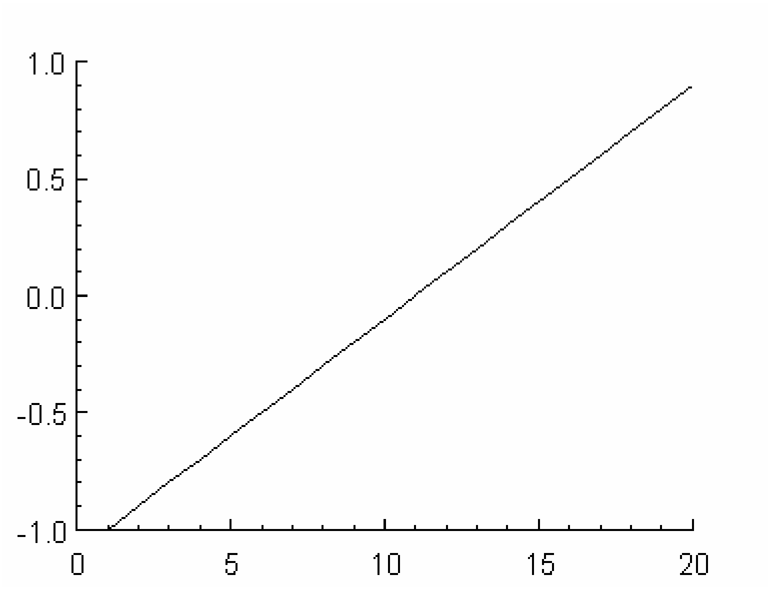




One-dimensional graphs

One-dimensional graphs are created from a single data vector. If you do not provide an independent variable to graph against, **Extrema** will use the vector index as the independent variable.

```
GRAPH [-1:1:0.1]
```



Alternatively, you can bin the values in the vector, to turn it into data pairs (i.e., bins and counts). The resulting vectors can be plotted using any of the two-dimensional graph types below. Binning data is explained in **Chapter 5: Data Analysis Examples**.

Two-dimensional graphs

Two-dimensional graphs represent data pairs. Typically, you will have two vectors of the same size, which should be plotted against each other in some way.

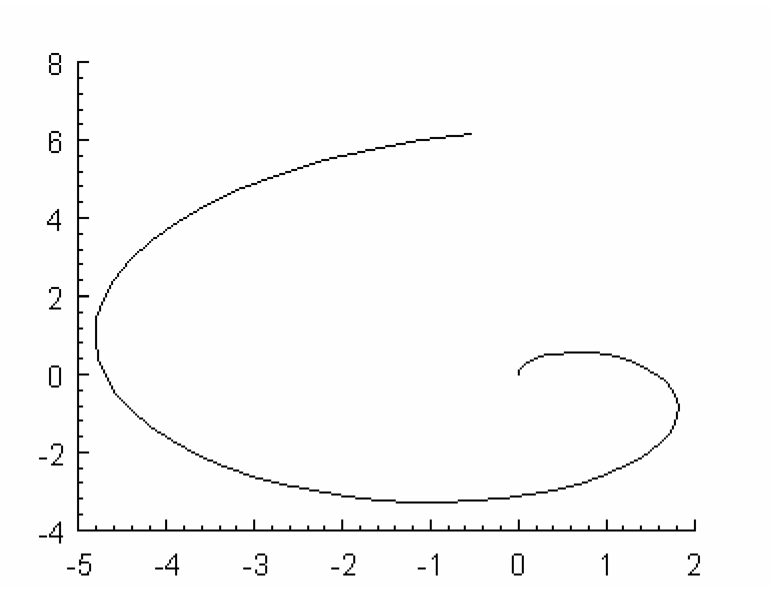
Line graphs

Line graphs connect each subsequent *(x,y)* data point with a line. This presumes that the points are ordered, so that they are connected in sequence.

```
GRAPH x y
```
! draw *y(x)* as a line graph

A parametric line graph is also easy to make. If our parametric independent variable is `T`, then we generate `X` and `Y` vectors by passing `T` through appropriate parametric functions:

```
t = [0:2*pi:.1]
x = t*SIN(t)
y = t*COS(t)
GRAPH x y
```



If your data is already in polar coordinates, you can graph it directly using the `\POLAR` option, e.g.:

```
GRAPH\POLAR radius_vector angle_vector
```

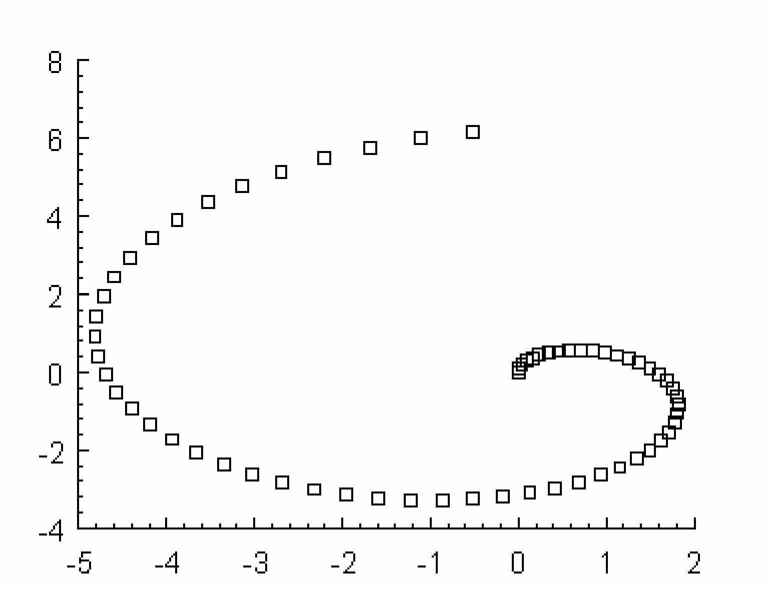
- Angles are presumed to be in degrees.

Note: The plotting symbol (characteristic `PLOTSYMBOL`) must be set to 0 to get a line graph. This is the default, so no special action needs to be taken unless the plotting symbol has been otherwise set (see **Scatterplots**).

Scatterplots

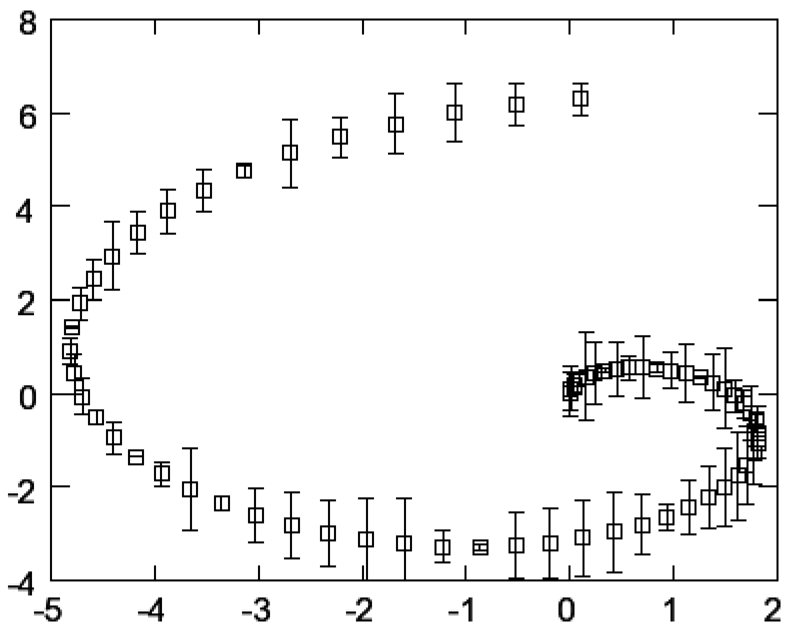
Scatterplots take the corresponding elements of each vector, and plot them as *(x,y)* data points, using whatever plot symbol has been selected. There is no requirement that the vectors be ordered in any particular way. To plot scattered points that are not joined by a line, select a negative symbol type.

```
SET PLOTSYMBOL -1
GRAPH x y
```



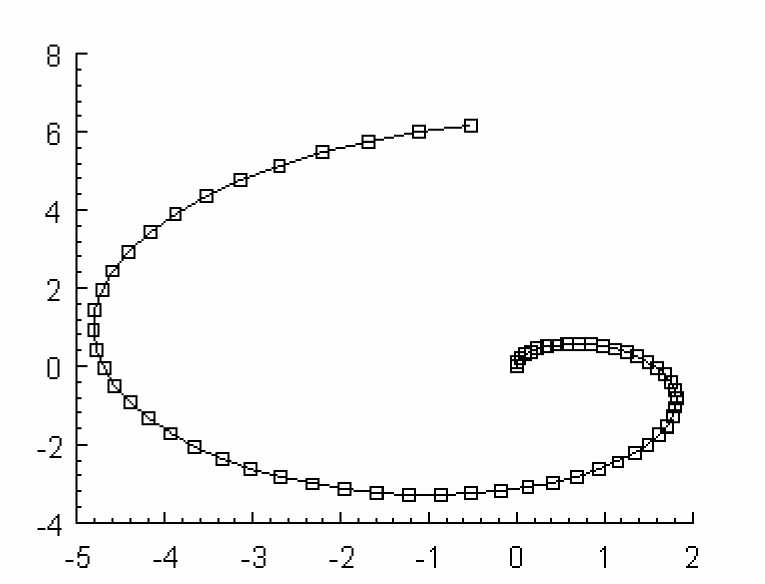
If we have errors in the data also stored in a matching vector, then we can add that information to the plot by specifying the error vector(s):

```
GRAPH x y yerr
```



If your data is ordered, and you would like the data points to be joined with a line, then simply use a positive plotting symbol number:

```
SET PLOTSYMBOL 1
GRAPH x y
```

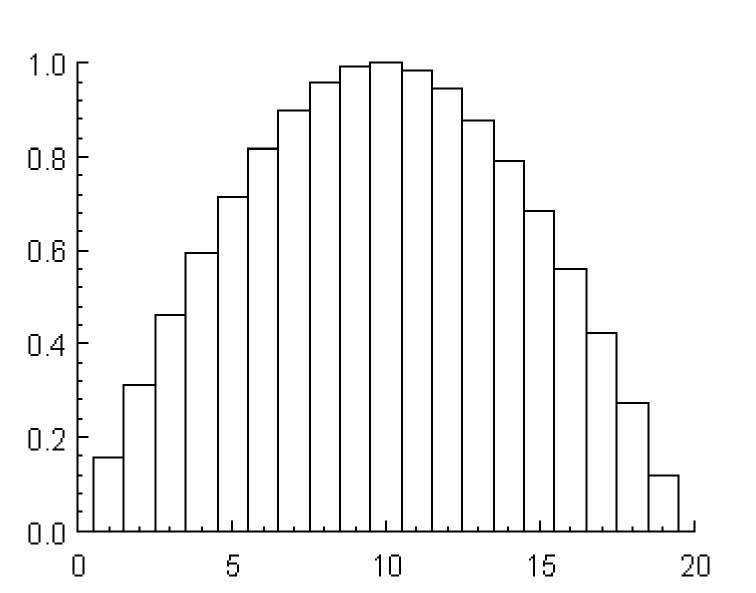


For a detailed listing of plotting symbols, see **Chapter 4: Customizing Graph Presentation**.

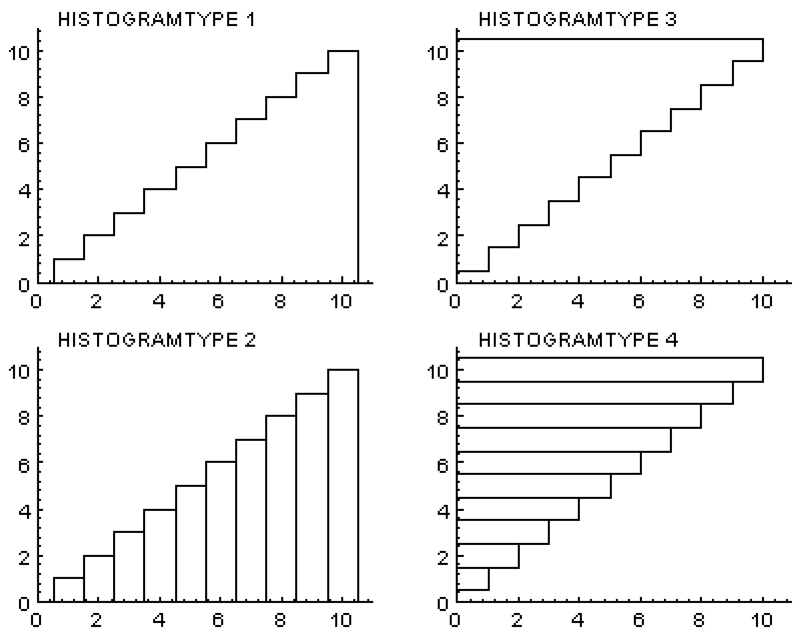
Histograms and bar charts

Histograms (bar charts) with tails going to *y*=0 are drawn by using the `\HISTOGRAM` qualifier with the `GRAPH` command. *x* values are assumed to be bins, and *y* values are assumed to be counts.

```
X = [1:19]
Y = SIN(x/(2*pi))
GRAPH\HISTOGRAM x y
```



The other types of histogram are shown below, each with the appropriate value of `HISTOGRAMTYPE`.



Three-dimensional graphs

Three-dimensional graphs represent data triplets. These are typically interpreted as 3-D functions, *z(x,y)*, and plotted accordingly.

3-D data typically comes in three forms:

- A matrix, whose indices represent the *x* and *y* dimensions, and whose values represent the *z* dimension. Variables of this type are representable as surface functions, and surface plots are usually generated from data of this type.

- Three vectors, which can represent the *x*, *y*, and *z* dimensions in a rectangular coordinate system. Corresponding indexes in these vectors are your data coordinates, so by analogy with 2-D graphs above, the data can be plotted as a scatter plot or a line (by connecting successive points). If it is reasonable to interpret the data as sampling a function *z(x,y)*, then it is also possible to interpolate a regular matrix from it, using the `GRID` operation, and then proceed as in the previous case. Certain types of 3-D plots (e.g., contour plots) can be generated directly from the original data vectors, however.

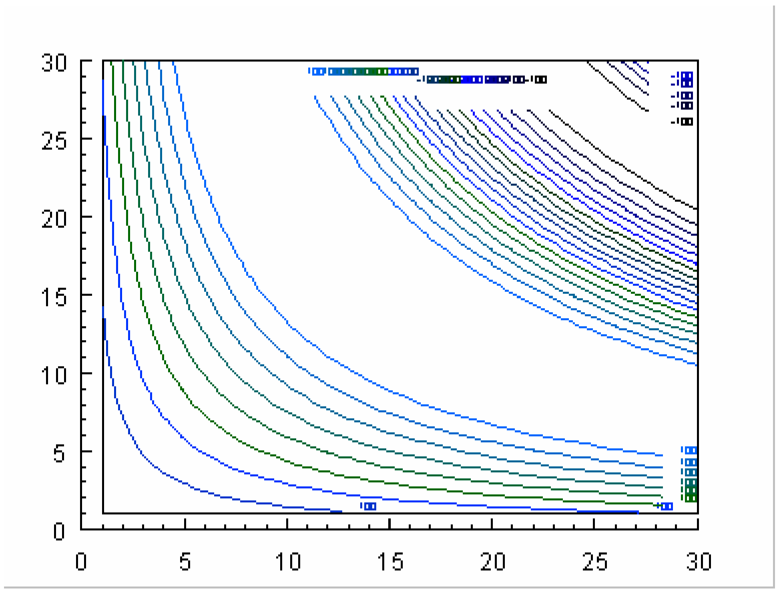
- Two vectors, which contain a series of *(x,y)* data pairs. Normally these would be plotted using a two-dimensional graph type, but they can also be binned to form a 2-D histogram, which is represented as a matrix and plotted as in the first case.

There are a number of different graph types that can be used to represent a surface function, *z(x,y)*. These are summarized below.

Contour Plots

In contour plots, the *z* value is interpreted as an elevation that is indicated using a contour map.

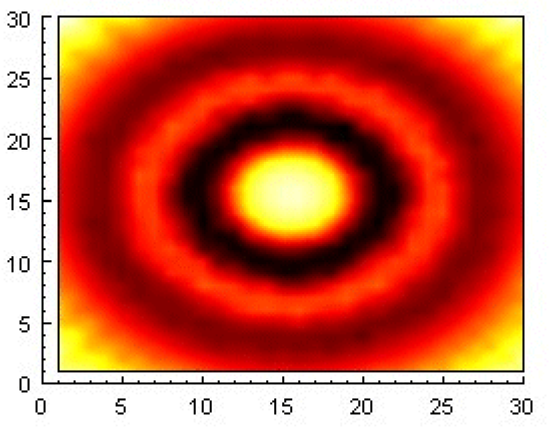
```
CONTOUR m 15        ! m is a matrix
```



Density plots

In density plots, the *z* value is interpreted as an intensity that is indicated using color (default), tone, dithering, or scaled boxes.

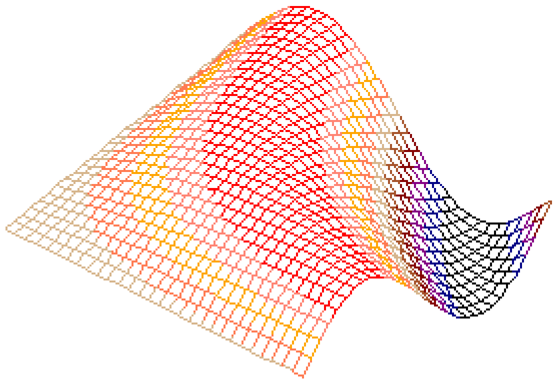
```
DENSITY m           ! m is a matrix
```



Surface plots

In surface plots, the *z* value is interpreted as a 3rd spatial dimension that is drawn in perspective.

```
SURFACE m            ! m is a matrix
```



Four+ dimensional graphs

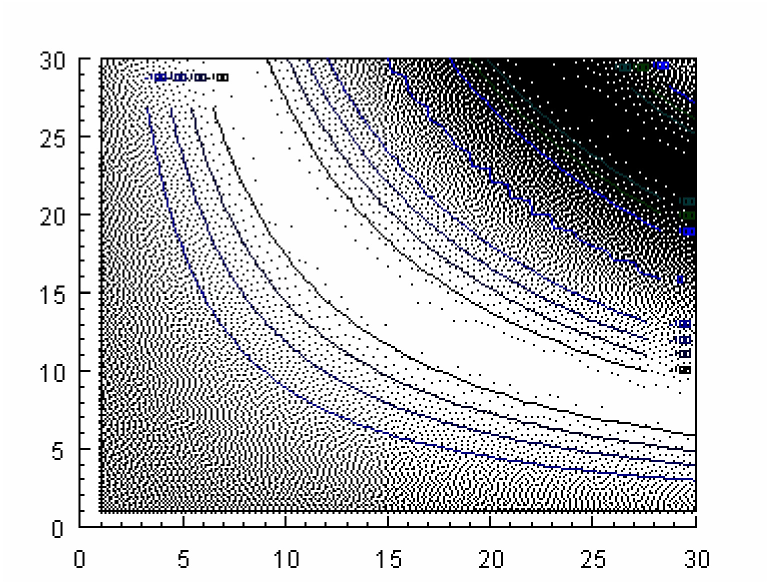
There are limits to how well four- (or more) dimensional data can be represented on a two-dimensional surface, but there are some tricks you can use. Here are a few ideas to help you get started.

Assume we have a tensor of data called `t`, and we wish to graph *t(x,y,z)*. We can slice `t` at different values of `z` and graph these slices using the 3-D graphing techniques above. (Section 3.5, below, explains how to tile multiple graphs on the same drawing.)

You could also build a script to plot each slice in sequence, automatically clearing and plotting the next slice as it went. This would give you a simple animation as you moved up (or down) through the slices.

Graphing two 3-D functions on the same drawing

Say we have two matrixes `f` and `g`, which represent surface functions *f{x,y}* and *g(x,y)* over the same *X* and *Y* ranges. We would like to plot them together on the same drawing for easy comparison. This is most effectively done by plotting `f` using some kind of density plot, and then drawing `g` as a contour plot over top of it. It may help to draw `g` in a different color to ensure it is clearly visible.



Multiple plots on the same drawing

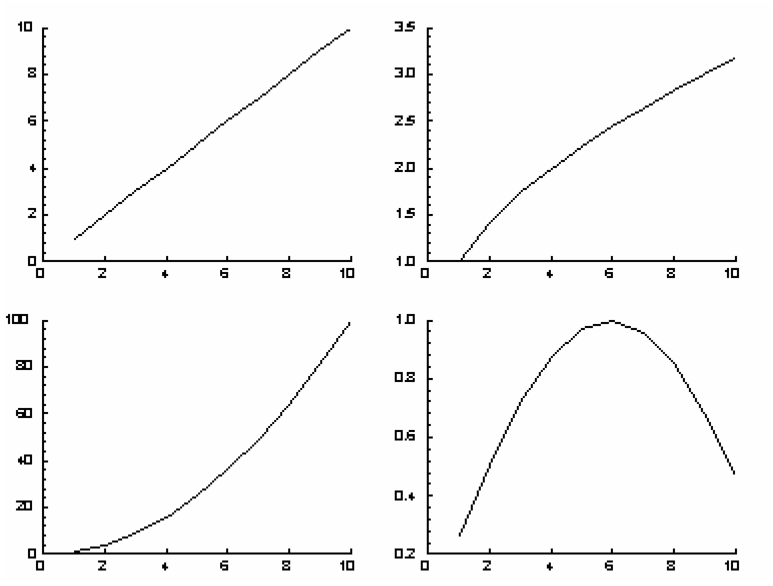
Researchers commonly need to combine graphs into the same drawing, plot multiple data sets on the same graph, draw different graphs with a common axis, and so on. There are many ways **Extrema** can be used to get these effects. A few are mentioned here; see also section 4.5.4.

Tile numerous graphs on the same drawing

Extrema divides the drawing area into **windows**, which can be selected by their number to confine a graph to a particular section of the drawing. Window number 0 is the default.

For example, to tile four graphs on the same drawing, simply select windows 5, 6, 7, and 8 in order, and issue an appropriate `GRAPH` command for each.

```
WINDOW 5
GRAPH x y1
WINDOW 6
GRAPH x y2
WINDOW 7
GRAPH x y3
WINDOW 8
GRAPH x y4
```

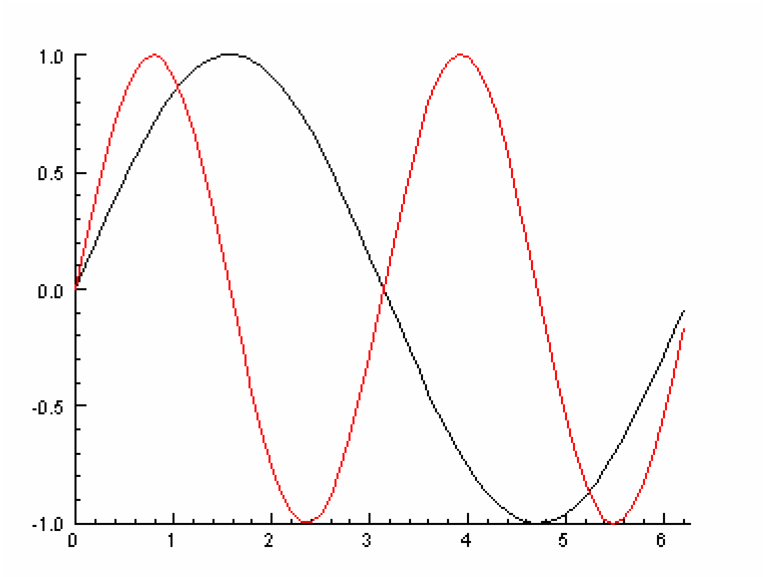


The `WINDOW` command can also be used to define your own custom set of drawing windows.

Draw two sets of data on the same graph at the same scale

Method 1: Manually set the axis scales to values that are appropriate for both graphs (see section 4.5.1). Then draw your first graph. Without clearing the graph, draw the second graph without axes.

```
SCALE 0 2*pi -1 1           ! xmin xmax ymin ymax
GRAPH x y
SET CURVECOLOR RED          ! change color for overlayed curve
GRAPH\OVERLAY x z
```



Method 2: Draw the graph that should be used to autoscale the axes first. Then freeze the axes at those values, before drawing the second graph without axes.

```
GRAPH x y
SCALE                    ! freezes the current scale
GRAPH\OVERLAY x z
```

Method 3: Graph either of the two data curves first, then overlay the second data curve. Then use the `REPLOT` command to redraw both curves on a common scale.

```
GRAPH x y
GRAPH\OVERLAY x z
REPLOT
```

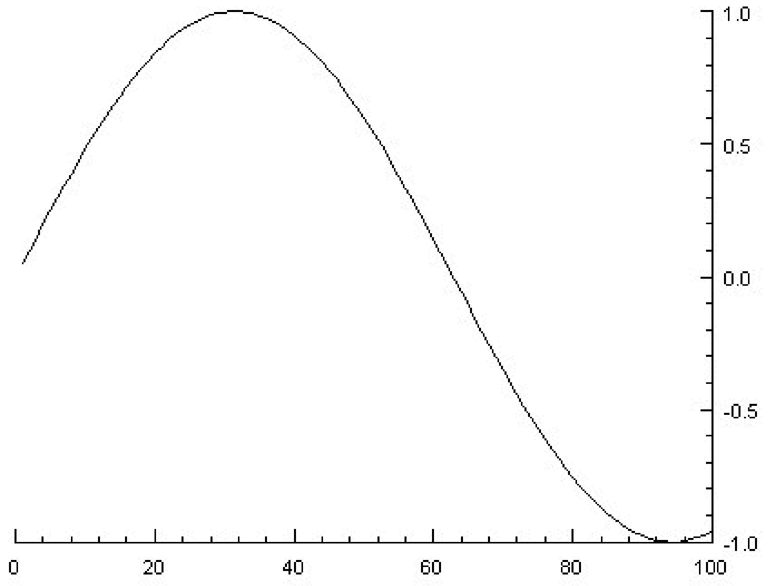
Draw two sets of data on the same graph, but at different Y-scales

Method 1: If you only need a labelled *y*-axis for one data set, the task is easy:

```
GRAPH x y                     ! y-axis is for this graph
GRAPH\OVERLAY x z             ! no y-scale shown for this graph
```

Method 2: By default, the y-axis is drawn at the left hand end of the *x*-axis. The `GRAPH\YONRIGHT` command draws the *y*-axis on the right. For example, the following commands produce the figure below.

```
X = [1:100]
GRAPH\YONRIGHT X SIN(X/20)
```

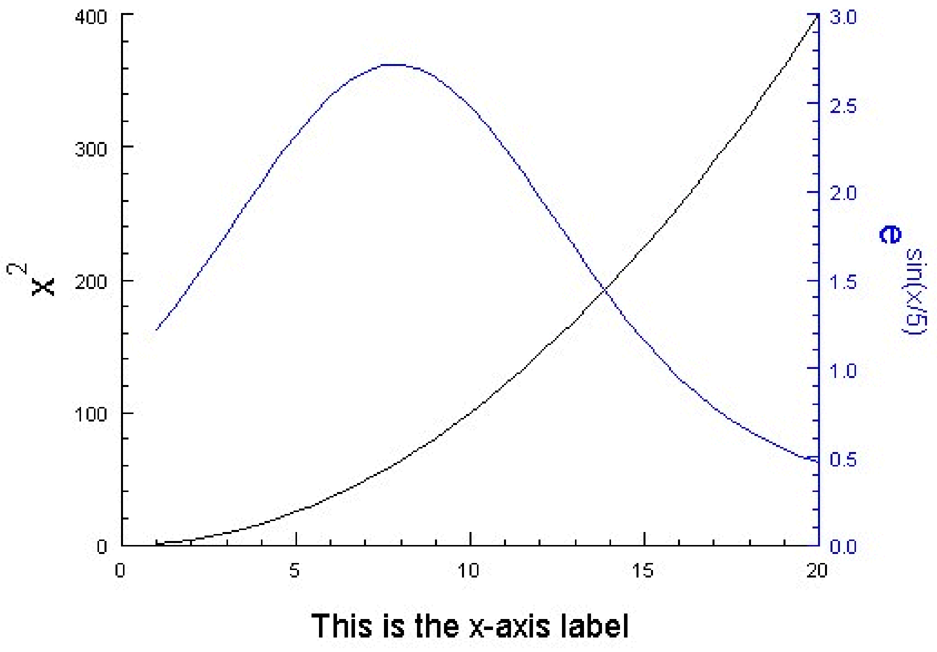


If you need a labelled *y*-axis for both data sets, just graph the first data curve with the *y*-axis on the left (the default), and then graph the second data curve with the *y*-axis on the right. You might want to change the color for the second *y*-axis and curve to distinguish it from the first. For example:

```
x=[1:20:.5]
y1=x^2
y2=EXP(SIN(x/5))
SET
 XLABEL 'This is the x-axis label'
 XLABELON 1
%XLABELHEIGHT 5
 YLABEL 'x<^>2'
 YLABELON 1
%YLABELHEIGHT 5

GRAPH x y1
SET
 YAXISCOLOR blue
 YNUMBERSCOLOR blue
 CURVECOLOR blue
 XAXIS 0
 YLABELCOLOR blue
 YLABEL 'e<^>sin(x/5)'

GRAPH\YONRIGHT x y2
```



The GRAPH\XONTOP command draws the *x*-axis on the top. More information on customizing axes and graph placement is provided in the next chapter.

Chapter

4

Customizing Graph Presentation

Once you can draw your data, you will want to customize the details of the drawing to improve its presentation.

Extrema has a large number of internal parameters used to control the drawing details. By altering these parameters you can vary the appearance of your drawing in a great variety of ways.

The most commonly used parameters can be easily set from the GUI, simply by checking off the desired options from those that are presented. The more obscure parameters may not have any convenient checkboxes, however, and will have to be set manually using a typed command.

Each drawing parameter has a name. To get the value of a parameter, use the function

```
GET characteristic
```

This returns a value that can be viewed interactively, or stored in a variable. To set the value of a parameter, use

```
SET characteristic value
```

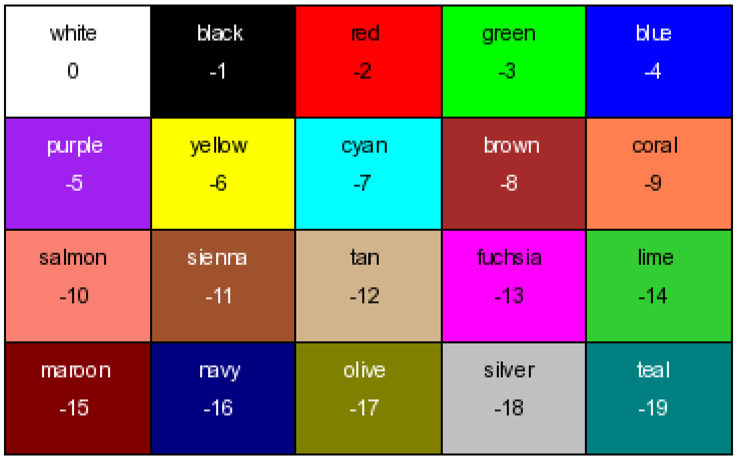
Specific commonly used examples follow. A comprehensive list of parameters is given in the **Extrema Command Reference**.

Many drawing parameters refer to positions on the drawing, which can be expressed in various units, including percentages. To interactively determine which position you would prefer, simply move your mouse over the drawing and the positions will be displayed below in whatever units have been selected.

Colors

Colors are numbered; these numbers are indexes into a color map. A color map can hold up to 256 colors. **Extrema** pre-loads a default color map that should be adequate for most cases, but you can also load your own color map (see the `SET COLORMAP` or the `SET COLORMAPFILE` command). In the GUI, select a color map color simply by clicking on the grid of colors that are presented. In the command language, you will need to know the color number. The default color map is stored in the file `DefaultColorMap.dat`, and the color numbers can be looked up there.

In addition to the color map, **Extrema** also predefines a set of colors that are always available, no matter what color map is currently loaded. These are checked off by name in the GUI, or by selecting a color name or a negative color number in the command language.



white 0	black -1	red -2	green -3	blue -4
purple -5	yellow -6	cyan -7	brown -8	coral -9
salmon -10	sienna -11	tan -12	fuchsia -13	lime -14
maroon -15	navy -16	olive -17	silver -18	teal -19

Default Drawing Color

The characteristic COLOR sets the current drawing color. If set, this color is used for everything placed onto the drawing. By changing this parameter in the course of adding things to the drawing, you can easily control the plotting colors.

```
SET COLOR BLUE               ! draw the curve in blue
GRAPH\OVERLAY x y
SET COLOR BLACK              ! draw the axes in black
GRAPH\AXESONLY x y
```

There are also more specific drawing color parameters used to set the colors of certain graph items, such as labels, axes, and so on. If set, these will override the global default color. If not set, the global color will be used.

```
SET CURVECOLOR BLUE              ! draw the curve in blue
SET XAXISCOLOR BLACK             ! draw the x-axis in black
GRAPH x y
```

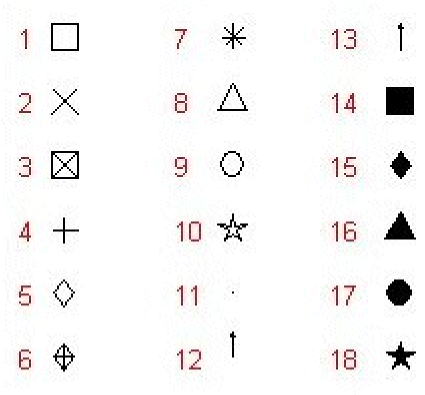
Plotting Symbols

The plotting symbol can be manually selected in the GRAPH window. In the command language, use:

```
SET PLOTSYMBOL n
```

Where n is the symbol number, taken from:

- If n is positive, successive points are connected by lines.
- If n is negative, the corresponding positive value is used, but points will not be connected.
- If n is zero, no plotting symbol is used (but the points are connected; the data is drawn as a simple curve in this case).



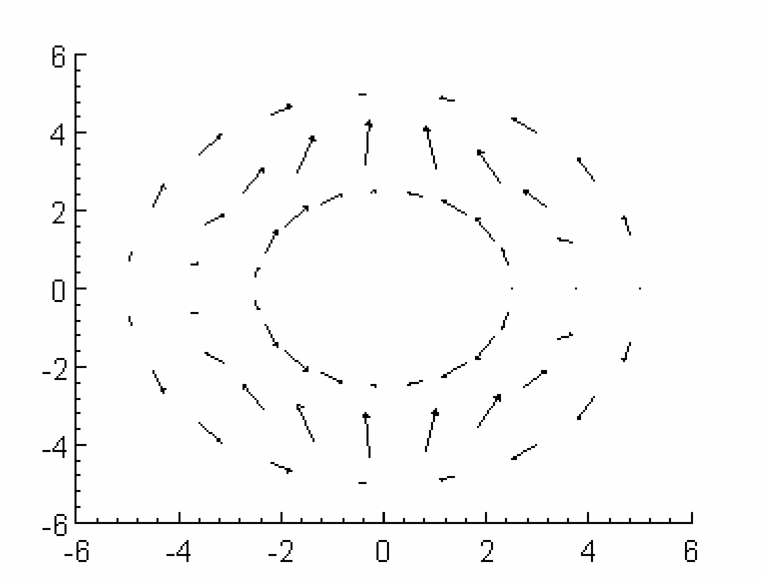
In addition to the plotting symbol, you can also specify the size, color, and angle (in degrees). If scalar values are used for these, the value will apply to every data point. If vector values are used, the vectors should be the length as the data vectors. The corresponding values for each point are used to set the plotting style for that point.

In the GUI, you can simply enter the size, color, and angle vectors (or scalars, or expressions) in the appropriate fields. In the command language, use:

```
SET PLOTSYMBOL symbol
SET %PLOTSYMBOLSIZE size
SET PLOTSYMBOLCOLOR color
SET PLOTSYMBOLANGLE angle
```

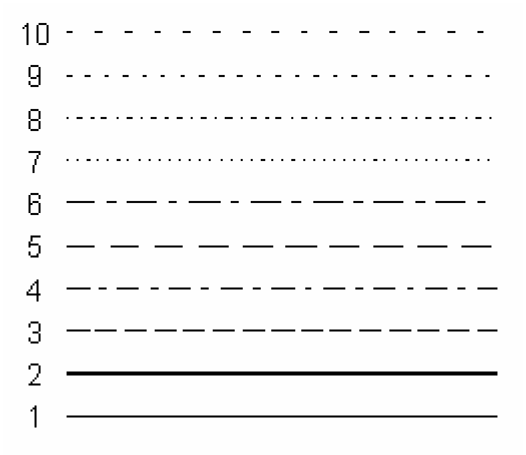
For example, to plot a vector field, we could select an arrow symbol where the arrow is centred at on the data value (#13), and then set the sizes and angles according to two vectors, `magnitude` and `direction`:

```
SET PLOTSYMBOL -13
SET %PLOTSYMBOLSIZE magnitude
SET PLOTSYMBOLCOLOR black
SET PLOTSYMBOLANGLE direction
GRAPH x y                                    ! draw vector field
```



Line type

The line type or style used for drawing lines on your graphs can be selected using the `SET LINETYPE` command:



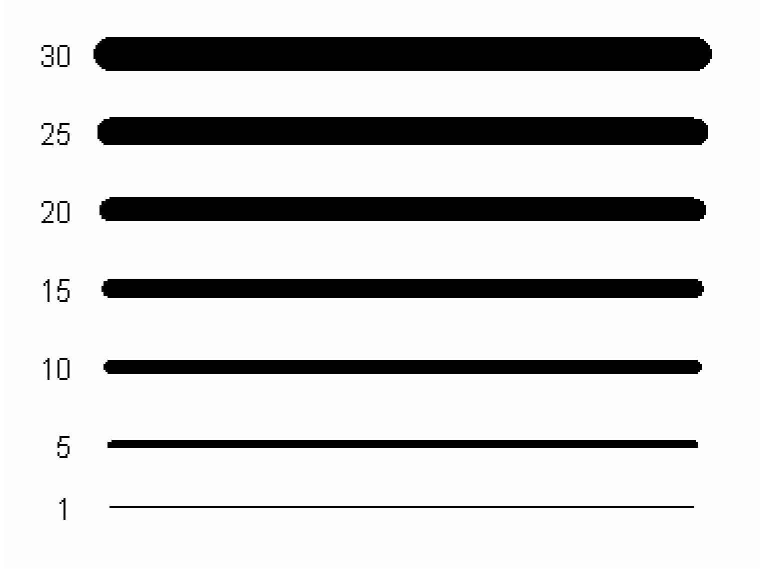
The line style is used only for graphed lines; it is not used for other lines such as graph axes.

Line width

`SET LINEWIDTH n` controls the line width of the axes, the data curve, and the plotting symbols drawn when the `GRAPH` command is entered. The units of

LINEWIDTH are pts, where a pt is 1/72 of an inch. For example, a line width of 36 gives 1/2 inch wide lines. The parameter n must be a scalar.

SET LINEWIDTH is a shorthand way to set CURVELINEWIDTH and PLOTSYMBOLLINEWIDTH.



Text

Graph titles, axis labels, and other drawn text strings have several characteristics that can be altered, in particular the font, size, color, placement, and angle. These can be specified for specific types of text labels (e.g., *x*-axis labels), or for text labels in general.

These parameters are all easily controlled in the GUI by setting the appropriate fields in the TEXT window (or the FONT sub-window).

You can explicitly enter the location on the graph where the label should be placed, or you can manually place it using the mouse. The placement of the label is with respect to a particular point in the box that encloses the text string:

In command mode, the various text drawing parameters are controlled using a number of settings:

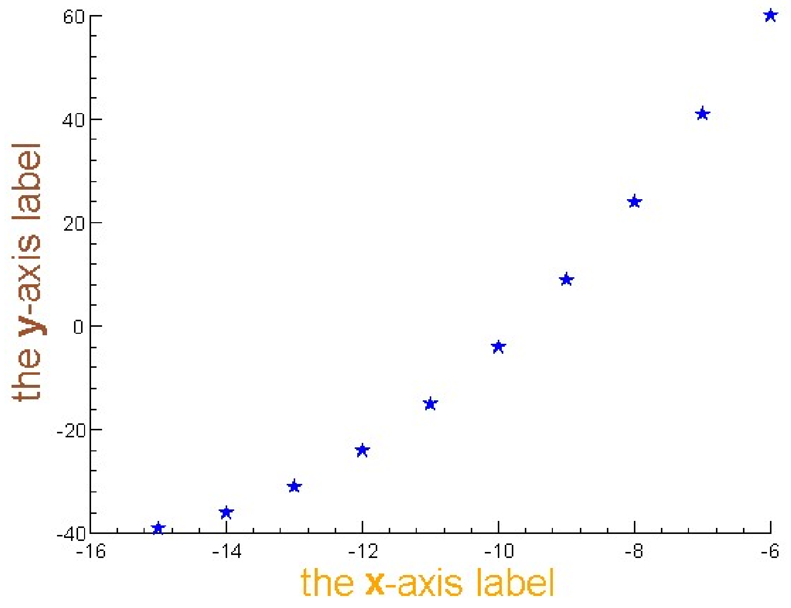
```
SET XTEXTLOCATION x_position
SET YTEXTLOCATION y_position
SET %XTEXTLOCATION xp_position      ! expressed as a percentage
SET %YTEXTLOCATION yp_position      ! expressed as a percentage
SET FONT font_name
```

Axis Labels

Axis labels are a special case of text strings, since they have a standard placement and orientation. The *x*-axis text label is drawn, centred, below the *x*-axis. The *y*-axis text label is drawn, centred, to the left of the *y*-axis. The axis text labels are drawn only when the axes are drawn. The character string may contain format commands.

The `SET XLABEL` command sets the automatic *x*-axis text label. Use the `SET XLABELON` command to toggle off/on drawing the *x*-axis text label. Change the sizes of the text label with `SET XLABELHEIGHT` or `SET %XLABELHEIGHT`. Change the font of the *x*-axis text label with the `SET XLABELFONT` command and change the color of the *x*-axis text label with the `SET XLABELCOLOR` command.

The `SET YLABEL` command sets the automatic *y*-axis text label. Use the `SET YLABELON` command to toggle off/on drawing the *y*-axis text label. Change the sizes of the text label with `SET YLABELHEIGHT` or `SET %YLABELHEIGHT`. Change the font of the *y*-axis text label with the `SET YLABELFONT` command and change the color of the *y*-axis text label with the `SET YLABELCOLOR` command.



Graph Legend

Legends are boxes of descriptive text that describe certain details of the graph. Typically, they are used to label different point types, different line types or colors, contour elevations, fit parameters, and so on.

The LEGENDON characteristic is changed with the SET command and the current value is obtained with the GET command. If LEGENDON ≠ 0, a legend entry is drawn into a legend frame box. A legend entry consists of a short line segment, with optional plotting symbol(s), and a text string. The legend entry is drawn when the GRAPH command is entered. The string portion of the legend entry is expected as the first parameter of the GRAPH command, for example:

```
GRAPH 'legend entry' x y
```

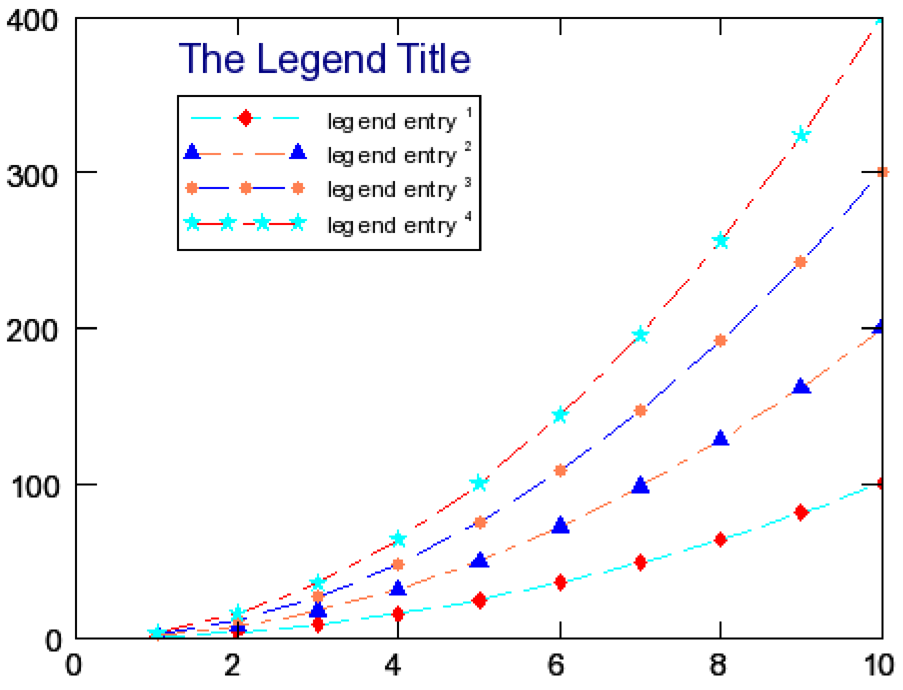
If LEGENDON = 0, a string entered as a first parameter with the GRAPH command is ignored.

Following is an example script using a graph legend and the picture that it produces.

```
x=[1:10]
SET
 LEGENDON 1
 LEGENDTITLECOLOR -16
 LEGENDTITLEFONT 'impact'
 LEGENDTITLE 'The Legend Title'
 LEGENDENTRYLINEON 1
 %LEGENDFRAME 20 60 60 90

symbolSizes = [2;2.5;3;2]
symbols = [15;16;17;18]
colors[1] = 'red'
colors[2] = 'blue'
colors[3] = 'coral'
colors[4] = 'cyan'
DO i = [1:4]
  SET
   PLOTSYMBOL symbols[i]
   PLOTSYMBOLCOLOR colors[i]
   %PLOTSYMBOLSIZE symbolSizes[i]
   LEGENDSYMBOLS i
   CURVECOLOR colors[5-i]
   CURVELINETYPE i+2

  GRAPH 'legend entry<^>'//RCHAR(i) x i*x^2
ENDDO
REPLOT
```



Graph Axes

To graph only the axes for a particular set of data, use:

```
GRAPH\AXESONLY x y
```

To graph a set of data with no axes, use:

```
GRAPH\OVERLAY x y
```

These options are handy if you make multiple drawing passes over the same graph. In the GUI you can simply select the appropriate checkboxes to get the same behaviour.

Scaling

Axes can be manually or automatically scaled.

Auto-scaling is the default, in which the axis will stretch or shrink to accommodate the full range of the plotted data. This is convenient for well-behaved data sets, but maybe not for data with spikes, infinities, or related problems. Autoscaling is also inconvenient when one is overlaying numerous similar graphs, where one requires that the scale be fixed.

Manual axis scaling is done using the SCALES command:

```
SCALES x_min x_max y_min y_max
SCALES x_min x_max n_x_tics y_min y_max n_y_tics
SCALES
```

The first form simply sets axis ranges. The second form also sets the number of large (numbered) tic marks that should be shown for each axis. The last form freezes the axis scales at whatever is their current value.

Tics

The parameters controlling *x*-axis tic marks are:

XTICSON	controls whether or not tic marks, both large and small, are drawn on the *x*-axis.
XTICSBOTHSIDES	controls whether or not tic marks, both large and small, are drawn on both sides of the *x*-axis.
XTICANGLE	controls the angle of the tic marks, both large and small, on the *x*-axis.
XNLINCS	controls the number of large, labelled, tic marks to be displayed on the *x*-axis
XNSINCS	controls the number of small, unlabeled, tic marks to be displayed between the large, labelled, tic marks on the *x*-axis.
XLARGETICLENGTH	controls the length of the large, labelled, tic marks on the *x*-axis.
XSMALLTICLENGTH	controls the length of the optional small tic marks on the *x*-axis. These are the unlabeled tic marks between the large, numbered, tic marks.
XIMAGTICANGLE	controls the angle, in degrees, measured counter clockwise, between the *x*-axis and a line joining the base of each large tic mark on the *x*-axis to the centre of the number labelling that tic mark.
XIMAGTICLENGTH	controls the distance, measured from the base of each large tic mark on the *x*-axis, to the centre of the number labelling that tic mark

The parameters controlling *y*-axis tic marks are:

YTICSON	controls whether or not tic marks, both large and small, are drawn on the *y*-axis.
YTICSBOTHSIDES	controls whether or not tic marks, both large and small, are drawn on both sides of the *y*-axis.
YTICANGLE	controls the angle of the tic marks, both large and small, on the *y*-axis.
YNLINCS	controls the number of large, labelled, tic marks to be displayed on the *y*-axis
YNSINCS	controls the number of small, unlabeled, tic marks to be displayed between the large, labelled, tic marks on the *y* -axis.
YLARGETICLENGTH	controls the length of the large, labelled, tic marks on the *y* -axis.
YSMALLTICLENGTH	controls the length of the optional small tic marks on the *y* -axis. These are the unlabeled tic marks between the large, numbered, tic marks.
YIMAGTICANGLE	controls the angle, in degrees, measured counter clockwise, between the *y* -axis and a line joining the base of each large tic mark on the *y*-axis to the centre of the number labelling that tic mark.
YIMAGTICLENGTH	controls the distance, measured from the base of each large tic mark on the *y*-axis, to the centre of the number labelling that tic mark

Logarithmic axes

To get logarithmic scaling on the *x*-axis, use SET XLOGBASE n, where:

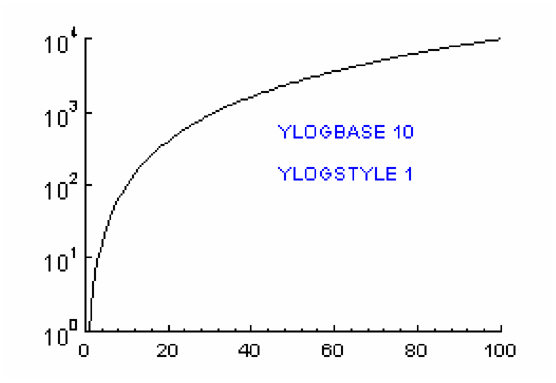
n > 1.0	the *x*-axis will have a logarithmic scale. The base will be the integer part of XLOGBASE, except for the special case: 1.05**e* > XLOGBASE > 0.95**e*, where *e* is the base of the natural logarithms, *e* ≈ 2.718281828, in which case the base will be *e*.
n ≤ 1.0	the *x*-axis will have a linear scale

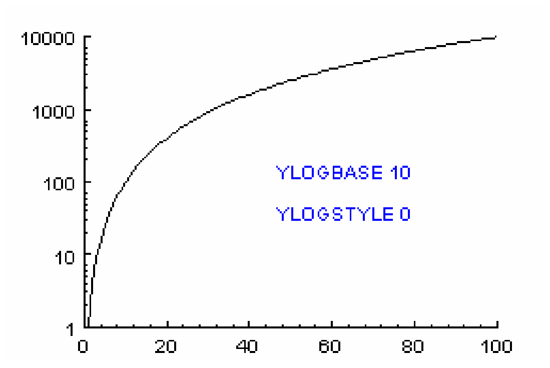
If XLOGSTYLE = 0, and XLOGBASE > 1.0, then the numbers labelling the large tic marks on the *x*-axis are displayed in decimal format. If XLOGSTYLE ≠ 0, and XLOGBASE > 1.0, then the numbers labelling the large tic marks on the *x*-axis are displayed in exponential format.

To get logarithmic scaling on the *y*-axis, use SET YLOGBASE n, where:

n > 1.0	the *y*-axis will have a logarithmic scale. The base will be the integer part of YLOGBASE, except for the special case: 1.05**e* > YLOGBASE > 0.95**e*, where *e* is the base of the natural logarithms, *e* ≈ 2.718281828, in which case the base will be *e*.
n ≤ 1.0	the *y*-axis will have a linear scale

If YLOGSTYLE = 0, and YLOGBASE > 1.0, then the numbers labelling the large tic marks on the *y*-axis are displayed in decimal format. If YLOGSTYLE ≠ 0, and YLOGBASE > 1.0, then the numbers labelling the large tic marks on the *y*-axis are displayed in exponential format.





Axis placement

The placement of the axes can be precisely controlled by manipulating the axis location parameters:

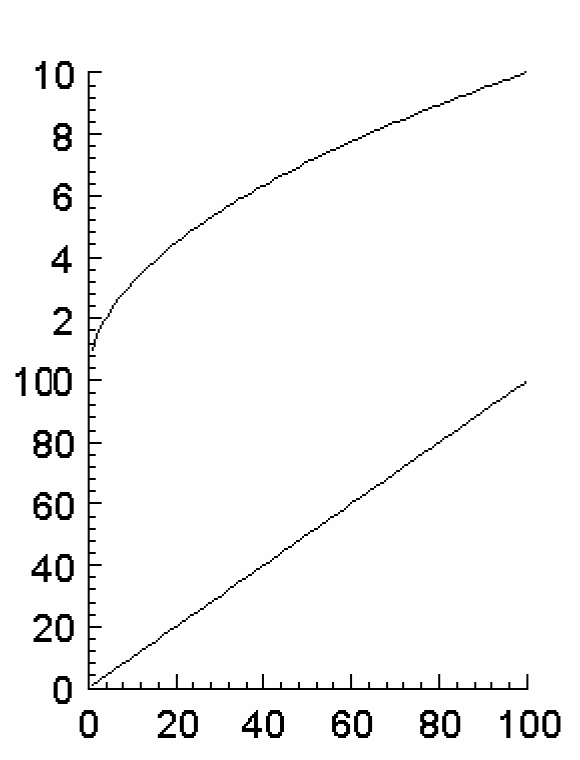
%XLOWERAXIS
%XUPPERAXIS
%YLOWERAXIS

`%YUPPERAXIS`

The percentage versions specify positions as percentages of the current drawing window; otherwise the positions are in the drawing coordinates.

By careful manipulation of these values, you can place one graph at any point on the drawing with respect to another. For instance, to adjoin two graphs along the *x*-axis so that there is an upper graph and a lower graph with a common edge:

1. Set `%YUPPERAXIS` to a reduced value, e.g., 50.
2. Plot the first graph.
3. Set `%YLOWERAXIS` to the value of `%YUPPERAXIS`
4. Set `%YUPPERAXIS` to 85.
5. Turn off drawing of the *x*-axis labels with `SET XAXIS 0`.
6. Plot the second graph.



In practice, there are some other parameters you may need to play with to keep the *y*-axis labelling clean, but the above will suffice in simple cases.

Chapter

5

Data Analysis Examples

Extrema provides numerous tools for data analysis, including data transformation tools, filtering tools, cutting and selection tools.

Elementary data manipulation is done using **Extrema's** built-in expression evaluation capabilities. Any expression involving a variable will return a similar variable, each element of which has been modified by the expression; the return value of the expression can be saved to another variable, or operated on directly.

Examples:

```
y = SIN(x)^2 + COS(x)^2        ! save expression results in variable y
GRAPH x 3*x^2-6x+2             ! graph expression directly
```

Expressions are built up of constants, variables, operators, and functions, which can be combined in any algebraic syntax, as in the examples above.

Operators

In addition to the simple arithmetic operators, `+`, `-`, `*`, `/`, `^` (exponentiation), and `( )` (grouping), there are also special vector and matrix operators:

`><`	- outer product	`<>`	- inner product
`<-`	- matrix transpose	`>-`	- matrix reflect
`/|`	- vector union	`/&`	- vector intersection

and a set of Boolean operators that return true (1) or false (0) values:

`|`	- or	`| |`	- exclusive or
`&`	- and	`\`	- not
`=`	- equal to	`~=`	- not equal to
`>`	- greater than	`<`	- less than
`>=`	- greater than or equal to	`<=`	- less than or equal to

Functions

Extrema has over 200 built-in functions that can perform a wide range of other operations on your data. Examples include:

- conventional mathematical functions, such as the trigonometric functions, logarithms, roots and exponentials, and rounding functions.
- advanced functions, such as Bessel, Clebsch-Gordan, etc.
- calculus functions, such as integral and derivative.
- probability functions
- programmers' functions, such as random number generation, variable tests, looping functions
- array and matrix functions, such as where, eigenvectors and eigenvalues, etc.
- string functions, such as case, date/time, etc.

In all cases, these functions accept data of a certain type, and return data of a certain type; they may be freely used in any expression, so long as the types they return make sense in the expression context.

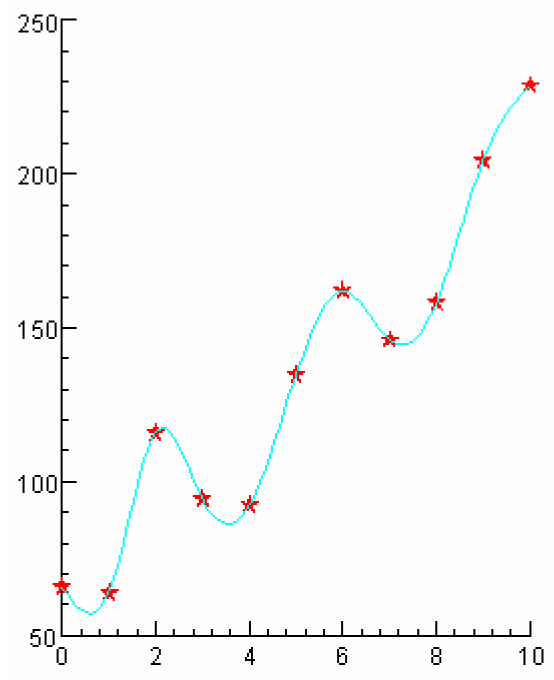
For further information, see **Operators** and **Functions** in the **Extrema Command Reference**.

Fitting

Fitting data, that is, describing a set of data points as some sort of function, is one of the most important forms of data analysis. **Extrema**'s data-fitting capabilities are sophisticated and flexible; complete details are provided in the **Extrema Command Reference**, but some simple examples are given here.

Smoothing

Smoothing is a simple way of fitting a set of data points to a smooth curve. There are several methods of calculating these smooth curves, notably cubic splines under tension (`SMOOTH` and `SPLSMOOTH` functions), and Saviztky-Golay filters (`SAVGOL` function).



Smoothing functions return a smoothed set of data, that is, they accept your data as input, and output a new set of values that fall on a smooth curve of the appropriate type. They can operate on any shape of data without any prior knowledge of the data's shape. (In some cases, there is a requirement that the data be monotonically increasing.) They will not, however, return an actual algebraic function describing the shape of your data. For this you need to do a proper fit (see below).

There are also interpolation functions that will *fill in* missing data using similar smoothing techniques (`INTERP` and `SPLINTERP` functions). Please refer to `SMOOTH`, `SPLSMOOTH`, `SAVGOL`, `INTERP`, and `SPLINTERP` functions.

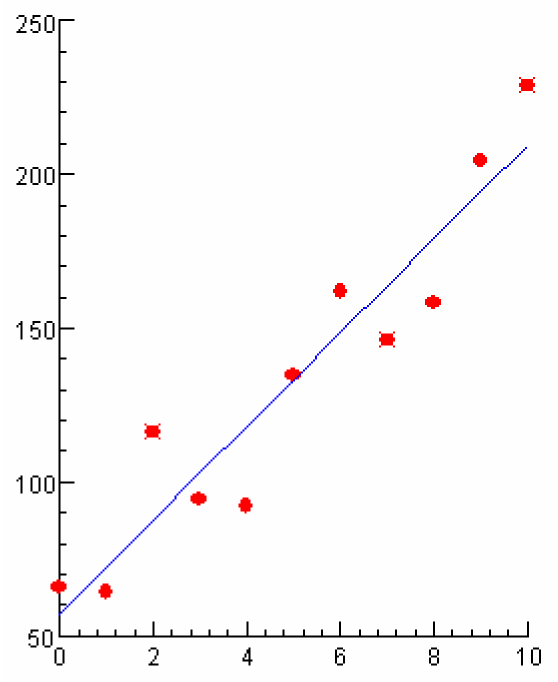
Fitting to a function

To describe your data as a function, you'll need to know in advance what function you will be fitting to. This function will be expressed with a number of `free parameters`, whose precise values are unknown. The purpose of the fit is to determine what values of those free parameters best match the data.

Note: Fitting is an uncertain process by its very nature. There is no guarantee that an appropriate fit will be found in all cases, and there is no guarantee that there is only one such fit that describes the data.

A **free parameter** is like a scalar variable, except that instead of being set by you (or your data analysis operations), it is set by **Extrema** in the course of making the fit. This difference in behaviour means that free parameters are declared differently, so that **Extrema** knows it can vary the parameter, instead of treating it as a fixed constant in the fitting expression.

```
SET PLOTSYMBOLCOLOR RED          !
GRAPH X Y                        ! graph the raw data
SCALAR\FIT A B                   ! declare free parameters
FIT Y=A+B*X                      ! perform the fit
SET PLOTSYMBOL 0                 ! graph the fit function as a line
SET CURVECOLOR BLUE              !
GRAPH X A+B*X                    !
```



Free parameters should be initialized to an appropriate *guess* value, from which the fit will begin. In simple cases, the actual value of the guess is not terribly important; **Extrema** will find the correct value regardless. In more complex cases, the initial guess will affect how the fit progresses, and could affect the final result. In other words, in some cases, different fits can be found depending on where you start, so choosing a reasonable guess to initialize the free parameters can be important. Once the fit is complete, the free parameters will have their fitted values. If **Extrema** failed to find a good fit, the free parameters will have the last values **Extrema** tried to fit with; or, optionally, they can be reset to their initial values upon failure.

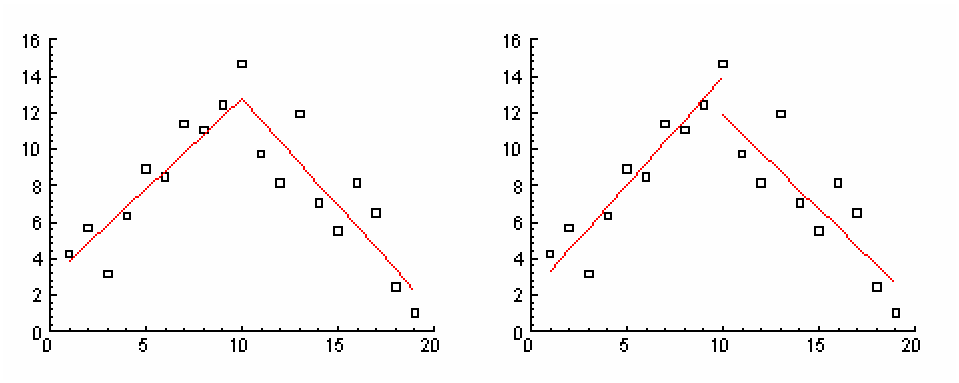
Normally, fitting results in multiple lines of text output describing the fit. The values of the free parameters, and various other values describing the accuracy of the fit, are all contained in this output. **Extrema** can optionally write some of this information into variables, for access by scripts and expressions later in the analysis process.

For more detailed information, please refer to the `FIT` command in the **Extrema Command Reference.**

Fitting different data segments to different functions

In some cases you will want to divide the data into segments or groups, and fit each group separately. For example, suppose you want to fit two line segments to the data such that they join at one end point. Below, on the left, is an example where the two segments are forced to join and, on the right, an example where they are allowed to float.

```
X=[1:19]
Y=[1;2;3;4;5;6;7;8;9;10;9;8;7;6;5;4;3;2;1]+5*ran(x)
WINDOW 5
SET PLOTSYMBOL -1
GRAPH x y
SCALAR\FIT a b c d
X0 = 10
FIT y=(a+b*x)*(x<=x0)+(c+d*x)*(x>=x0)+(a+b*x-c-d*x)*1000*(x=x0)
SET PLOTSYMBOL 0
I1 = WHERE(x<=x0)
I2 = WHERE(x>=x0)
Y1 = a+b*x
Y2 = c+d*x
SET CURVECOLOR red
GRAPH\OVERLAY x[i1] y1[i1]
GRAPH\OVERLAY x[i2] y2[i2]
WINDOW 7
SET PLOTSYMBOL -1
GRAPH x y
FIT y=( a+b*x)*(x<=x0)+(c+d*x)*(x>=x0)
SET PLOTSYMBOL 0
Y1 = a+b*x
Y2 = c+d*x
SET CURVECOLOR red
GRAPH\OVERLAY x[i1] y1[i1]
GRAPH\OVERLAY x[i2] y2[i2]
REPLOT\ALL
```



Binning

Binning data has already been mentioned a few times as a means of converting one-dimensional data into two-dimensional data (`BIN` command), or two-dimensional into three-dimensional (`BIN2D` command).

Simply put, binning counts the data points falling into a certain range. This results in a vector (or vectors, in the 2-D case) describing the ranges (the bins), and a second vector (or matrix) describing the counts.

Simple binning is straightforward. An input vector of values is taken as input, and two output vectors containing the bins and the counts are returned.

```
BIN x xbin xcount                    ! bin the values in x
GRAPH\HISTOGRAM xbin xcount
```

There are many binning options, among them:

- various options for defining the bin boundaries
- the averages of the values in each bin can be returned
- values can be counted conditionally
- counts can be weighted
- lagrange binning

Please refer to the `BIN` command for more information.

Interpolation

There are many cases where one needs to interpolate data, for instance:

- estimating missing data values
- converting an irregular data sample to a monotonically increasing data sample

- representing a set of data points as a smooth function

Interpolation presumes the data can be represented as a smooth function, and that this function passes through all of the data points. Interpolation therefore consists of looking up the *y*-values of this function for any *x* that is not represented in the original data. This is normally done by means of the `INTERP` function, which returns a data vector containing the interpolated values. The `INTERP` function accepts three arguments:

- *x*-vector, a monotonically increasing set of *x* values.
- *y*-vector, the values of *y* at each of the above *x* values.
- *x*-interpolation points, a set of *x*-values at which to interpolate new *y*-values.

The method of interpolation is normally interpolating splines, but an optional fourth argument can be used to select an alternate interpolation method:

▪ `LINEAR`	simple linear interpolation
▪ `LAGRANGE`	general Lagrange interpolation
▪ `FC`	Fritsch and Carlson method of monotone piecewise cubic interpolation

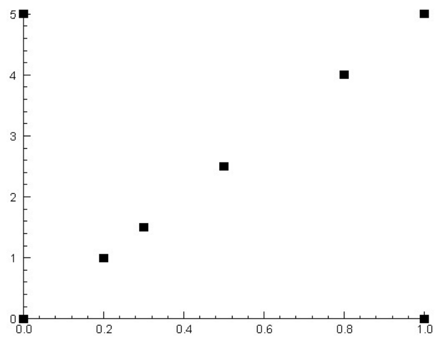
If one's starting data is not monotonically increasing, then one can use the `SPLINTERP(x,y,n)` function instead. It accepts an arbitrary set of *x* and *y* values, and a number of points to interpolate. The output is a 2-column matrix, the first column of which gives the interpolated points (i.e., *x*-values), and the second of which gives the interpolated values (i.e., *y*-values).

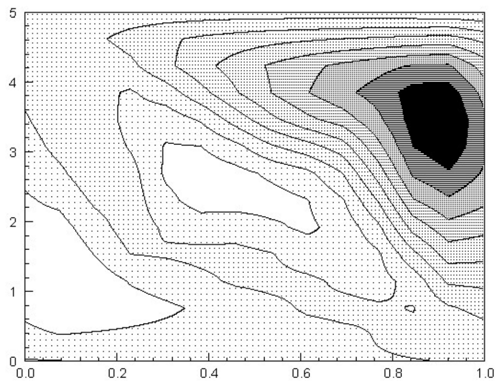
2-D interpolation

Beginning with a scattered set of 3-D data points in three vectors (say, `x`, `y`, and `z`), you can interpolate a regular matrix using the `GRID` command. The three vectors are assumed to represent scattered points, where `z[i]` is the altitude corresponding to the coordinates (`x[i]`,`y[i]`). The set of scattered data points is used to construct a Thiessen triangulation of the plane and a regular matrix, `m`, is interpolated.

For example, the following script produces the pictures below.

```
X=[1;0;1;0;0.2;0.3;0.5;0.8]
Y=[5;5;0;0;1;1.5;2.5;4]
Z=[10;10;10;10;-100;10;-100;500]
GRID\XYOUT X Y Z M XOUT YOUT
SET PLOTSYMBOL -14
GRAPH X Y                              ! produce the graph on the left
SET PLOTSYMBOL 0
DENSITY\DITHER XOUT YOUT M             ! produce the density plot
```





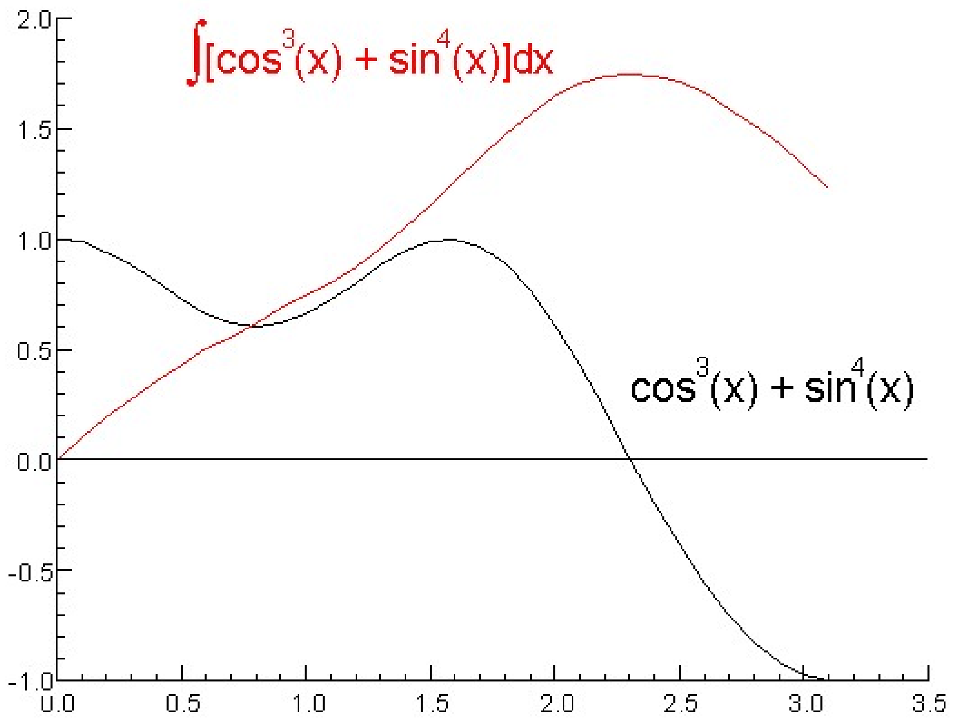
Integration

Integration is the summing of areas and volumes under curves and surfaces. **Extrema** provides you with several tools to accomplish this.

The `INTEGRAL` function is the simplest method; it accepts two vectors representing the *x*-values (monotonically increasing) and *y*-values of the function to be integrated. The return value is the integrated function, i.e., the integral at each *x*-value; there is one additional value appended to the end of this output vector, and that is the integral over the full range of *x*.

For example, to find the area under $\cos^3(x)+\sin^4(x)$ for $0 \le x \le \pi$:

```
pi = ACOS(-1)
x = [0:pi:.1]
yi = INTEGRAL(x,COS(x)^3+SIN(x)^4)
value = yi[#]
```



Other functions

Please refer to the DERIV function (derivative of a function); and the AREA function (area within a polygon), and the VOLUME function (volume under a surface). There are also numerous special integration functions, such as elliptic integral, Fresnel integral, exponential integral, sine integral (SININT) and cosine integral (COSINT).

Two-dimensional integration is typically done using the VOLUME function, which can operate on a variety of data types:

- vectors containing scattered *(x,y)* points
- vectors containing scattered polar coordinate points (angle, radius)
- regular matrix

Data Selection

Filtering, cutting, and other forms of conditional data selection are a big part of many analysis tasks. There are many ways this can be accomplished in **Extrema**.

Many of these techniques involve selecting subsets of vectors, matrices, or tensors, according to some arbitrary condition. A trivial form of data selection simply consists of selecting the desired indexes, for example:

```
good_data = m[#,*]            ! only the last column of the matrix is good
```

If the good data is scattered throughout a vector (say `data`), and you have the indexes of the good values in another vector `good`, then you can select the good data using the notation:

```
good_data = data[good]
```

Determining which indexes are good and which are bad is the tricky part. The `WHERE` function is invaluable for this. It accepts a vector as input, and returns the indexes where the input vector was not equal to zero.

The input vector is usually some kind of Boolean operation on the actual data vector, such that a vector of true/false (1/0) values is actually passed to the `WHERE` function. The return vector of indexes is then used to select the values from the original data vectors.

The power of this function is best illustrated with a few simple examples:

Example 1: select the data points within 1 unit of the origin

We have a scattered set of data points in the vectors `x` and `y`, but we want only the ones that lie within the unit circle, i.e., the points that satisfy `SQRT(x^2+y^2)<=1`.

```
i=WHERE(SQRT(x^2+y^2)<=1)          ! select data in unit circle
                                   ! i is our list of selected indexes
GRAPH x[i] y[i]                    ! graph the selected data
```

Example 2: select only the data points collected within a time window

We have an unordered, scattered set of data points in the vectors `x` and `y`, and the times of each in a vector `t`. Say our time window is defined by `tmin` and `tmax`.

```
i=WHERE(t>=tmin & t<=tmax)         ! select data in time window
GRAPH x[i] y[i]                    ! graph the selected data
```

Example 3: select only the data points whose error is below a threshold

We have a set of data points in the vectors `x` and `y`, with errors denoted by vectors `xerr` and `yerr`. We want to reject any data point with an *x*-error exceeding `xthresh` or *y*-error exceeding `ythresh`.

```
i=WHERE(xerr<=xthresh|yerr<=ythresh)          ! select good data
```

```
GRAPH x[i] y[i] xerr[i] yerr[i]                ! graph the selected data
```

Example 4: eliminate spikes from the data

We have a set of data points in the vectors `x` and `y`, with occasional anomalous (single-point) spikes where the *y*-value goes very high. In the simple case, we can simply filter out any data over a certain *y*-value (say, `ymax`):

```
i=WHERE(y<ymax)
GRAPH x[i] y[i]          ! graph the selected data
```

This won't work if the good data occasionally can rise above `ymax`. In this case you might only want to filter out spikes with a certain minimum height (say, `spike_min`) relative to adjacent good points. Here is a simple way to accomplish that:

```
ydiff[1] = 0        ! get y-differences between each point and the previous point
ydiff[2:LEN(y)] = y[2:#]-y[1:#-1]
i=WHERE(ydiff<spike_min)
GRAPH x[i] y[i]    ! graph the selected data
```

Chapter 6

Output

Printing graphs

Printing your graphs is very easy; simply select Print and proceed as you would for any other Windows printing job.

Exporting graphs for inclusion in other documents

Researchers commonly need to include their graphs in other documents, such as research papers, written reports, or web pages. For these purposes, **Extrema** can export to several industry-standard graphics formats: PostScript (EPS), Portable Network Graphics (PNG), and Joint Photographic Experts Group (JPEG). The HARDCOPY command is used for saving the graphics to a file in one of the supported formats. Encapsulated PostScript is the default format, if no qualifier is entered with the HARDCOPY command.

PostScript & EPS

PostScript (EPS) is the industry standard for printed documents; it provides excellent, publication-quality output that is completely scalable, and is compatible with documents conforming to the Portable Document Format (PDF) or to the TeX and LaTeX systems that are common in scientific publishing.

PNG

Portable Network Graphic (PNG) is a bitmap image format that is supported by most major web browsers, including Explorer and Netscape. As a bitmap format, it is inferior for regular publication purposes, but it is convenient for in-lined image display in web pages. It gives high rates of compression for conventional drawings and plots, and is the recommended graphics format for most drawings.

JPEG

Joint Photographic Experts Group (`JPEG`) images are also stored in a bitmap format, and suffer from all the drawbacks of `PNG` images. They are also optimized for displaying photographic images, and do not generally give good compression for conventional drawings and plots. Some complex drawings that involve smoothly varying gradients of tone or color may benefit from being exported to `JPEG` format, however.

Saving Data

If you have done much data processing, you may want to save your modified data in a file so that you can come back to it in a future **Extrema** session and analyze it further.

The simplest way to write an output file is using the `WRITE` command, which takes your variables and writes them in columns to the designated output file.

```
WRITE mydata.dat x y z        ! write x, y, and z vectors to 3 columns
```

This sort of data file is also easy to read back in to **Extrema** (see section 2). It is fairly portable in general, and could also be imported into most spreadsheet programs, for instance.

There are many options and other uses for the `WRITE` command, including:

- writing text strings, scalars, matrixes, and tensors
- specifying precise formats
- appending to files
- writing individual vectors in multiple columns

Chapter

7

Scripting

Interactive use of **Extrema** is adequate for one-off data analysis and visualization jobs, or for exploratory data analysis, in which you are trying to gain an understanding of your data. There are other cases where completely interactive control over the job are not desirable, for instance:

- favourite configurations, or preferred defaults, that you'd like to load instantly
- routine or repetitive tasks that always involve the same set of steps
- production analysis, in which established data analysis routines are used repeatedly on similar data sets
- long or intensive analysis jobs that don't need baby-sitting

In these cases, you will use scripts to fully or partially automate the process. Examples of **Extrema** scripting commands have been provided throughout this guide; in fact, every **Extrema** operation has both an interactive GUI method, and a corresponding command-driven method to accomplish it. In addition, there are special tools that are only available when in script mode, such as:

- branches
- loops
- subroutines

If you have computer programming experience, you'll recognize these as the essential elements of a programming language. Using these elements, you can make your **Extrema** scripts perform arbitrarily complicated tasks. So long as the analysis procedure is quantifiable in some way, **Extrema** can be configured to make all the necessary decisions, and take the appropriate steps in the handling of your data.

Creating and editing scripts

Extrema scripts are simple text files, with one command per line. You can use any editor or word processor to create and/or edit your script, provided you save it in plain text (`txt`) format.

By default, **Extrema** scripts are assumed to have the extension `.pcm`. Any other extension can be used, however, if it is specified when you run the script.

Every line of the script is an **Extrema** command. By default, the commands are executed in order, unless you have loops or branches (see below).

Comments

Script lines that begin with the exclamation point character, `!`, are **comments**. These lines are ignored by **Extrema**, and are used to add commentary to your scripts to help document what they are doing. (Many of the scripting examples in this guide include comments.) Comments can be placed on their own line, or they can be appended to the end of the line. Everything from the exclamation point character to the end of the line is ignored.

Comment characters are also sometimes used to disable lines in a script.

Running scripts

To run a script, use the syntax `@script`. By default this will execute the script in the file `script.pcm`. If you saved your script in a file with a different extension (say `.txt`), you could say `@script.txt` instead.

By default, scripts run silently (with no output) unless they perform actions that generate some kind of output (e.g., draw graphs, do a fit). In some cases you may prefer that your script be "chatty" so that you can follow its progress. You could insert commands to force some output at particular moments, or you could just include the command `ENABLE ECHO` at or near the top of the script, which causes **Extrema** to echo each command as it is performed. (Use `DISABLE ECHO` to turn this behaviour off.)

Interacting with the user

Not all scripts are meant to run by themselves off in a dark room somewhere. Some scripts will have a real person in front of them, and interaction with this user may be necessary. There are a few special **Extrema** commands for this purpose:

- `DISPLAY`: causes **Extrema** to display a message or other text string
- `INQUIRE`: prints a message (such as a question), and waits for the user to provide an answer. The answer is kept in a variable, and so can be used in subsequent

operations. In the trivial case, this could be a prompt to "`Hit return to continue...`", and the answer is irrelevant. In more sophisticated cases, it could prompt for multiple variable names or values to operate on

Initialization script

You can have **Extrema** automatically run a script when it first starts. This is convenient for setting up physical constants or your favourite graphing parameters and defaults. To make an initialization script, simply create a file named `extrema.init` in the `scripts` subfolder of `C:\extrema` (or wherever you installed **Extrema**).

Subroutines

Scripts can be executed from within other scripts; sometimes the top-level script (invoked by a person) is referred to as the program, while the remaining scripts (invoked by the script) are called the subroutines. Other than that, a subroutine is just like any other script, and is executed the same way.

The `RETURN` command, when encountered, stops execution of the current script, and returns to the next script above. This is a common way of ending a subroutine, although it is not strictly necessary at the very end --- `RETURN` is implied at the end of every script, so it is only needed for abnormal returns in the middle of the script. `RETURN`'ing from a top-level script will return the user to interactive mode.

Subroutine scripts will commonly require input parameters, by means of which the calling script passes information for the subroutine to operate on. Parameters are typically variables that are given after the subroutine name, for example:

`@analyze x y z`   ! pass `x`, `y`, and `z` vectors to the analyze subroutine

Parameters can also include text strings, for instance file names:

`@savedata myfile.dat x y`   ! save the `x` & `y` vectors to `myfile.dat`

Subroutines that require parameters can also be executed interactively, without specifying the parameters. In this case, **Extrema** will interactively prompt you for the missing information.

To use a parameter inside a subroutine, use the notation `?`*n*, where *n* is the number of the parameter to substitute into that spot. For instance, in the previous example, the script `savedata.pcm` might contain the following lines:

```
WRITE\APPEND ?1 ?2 ?3
DISPLAY ?1//' updated with '//LEN(?2)//' points'
```

which would be translated to:

```
WRITE\APPEND myfile.dat x y
DISPLAY 'myfile.dat updated with '//LEN(x)//' points'
```

There is no limit to how many times you call a subroutine, how many different subroutines you call, or how many levels deep subroutines may be nested inside each other.

Branching and looping

Branches are places where the execution of a script can fork. This is usually controlled using the `IF ... THEN` construct. There are two ways this statement can be used:

Method 1:

`IF (`*condition*`) THEN` *command*

Method 2:

`IF (`*condition*`) THEN`
command
command
...*etc.*
`ENDIF`

In both cases, the commands are executed only if the condition evaluates to true (i.e., 1). In method 1, only the single command is executed, whereas in method 2, an entire sequence of commands is executed. If the condition evaluates to false, then the commands are ignored, and execution jumps to the next non-conditional statement.

You can also jump non-conditionally using the `GOTO` statement.
The command `GOTO` *label* command transfers script execution to the given label.

A label is a special statement that does nothing; it is simply a place marker in the script. It consists of just a name (no spaces) followed by a colon. The colon is not used in the `GOTO` statement.

```
start:
...
IF (a>b) THEN RETURN
GOTO start
```

In the above example, the `GOTO` statement restarts the script from the top; this could repeat indefinitely, if we didn't have the conditional `RETURN` statement to exit the script at some point. (This is a simple example of a loop.)

Here is another way to use GOTO statements to continue executing a block of code until some condition is met:

```
begin:  ! loop to execute a block of code until a equals b
...
IF (a != b) THEN GOTO begin
```

If you cannot get **Extrema** to make the key decisions on its own, you could always prompt for a human decision on whether to repeat the loop:

```
begin:      ! loop to execute a block of code until user is happy
...
INQUIRE 'Redo (Y/N)?' answer
IF (EQS(answer),'Y') THEN GOTO begin
```

You could use similar conditional branching statements to control the number of times a section of code gets executed:

```
nloop = 0         ! ugly loop to execute a block of code 10 times
do_loop:
...
nloop = nloop+1
IF (nloop <= 10) THEN GOTO do_loop
```

DO loops

The previous example can be written more concisely as:

```
DO x = [1:10]           ! better loop to execute a block of code 10 times
...
ENDDO
```

The general form of the DO loop is:

```
DO scalar = vector
...
ENDDO
```

The scalar is set to each element of the vector in turn, with the loop code being executed once for each such setting. Thus, the following are all legal DO loops:

```
DO i = [-10:10:2]          ! loop from -10 to +10 in steps of 2
...
ENDDO

DO i = x^2              ! loop over the squares of all values of x
```

```
...
ENDDO

DO x = [1:LEN(a[*,#])]                ! matrix c is the difference between
                                      ! matrixes a and b
  DO y = [1:LEN(a[#,*])]
    c[x,y] = a[x,y] - b[x,y]
  ENDDO
ENDDO
```

This last example is illustrative of the looping mechanism (using nested loops), but otherwise it is a bit artificial. This is how you might handle the operation of subtracting two matrices in a conventional programming language, but loops of this nature are implicit in **Extrema** 's variable handling and expression evaluation. The same operation can be accomplished more efficiently with:

```
c=a-b          ! implicit loop over every element
```