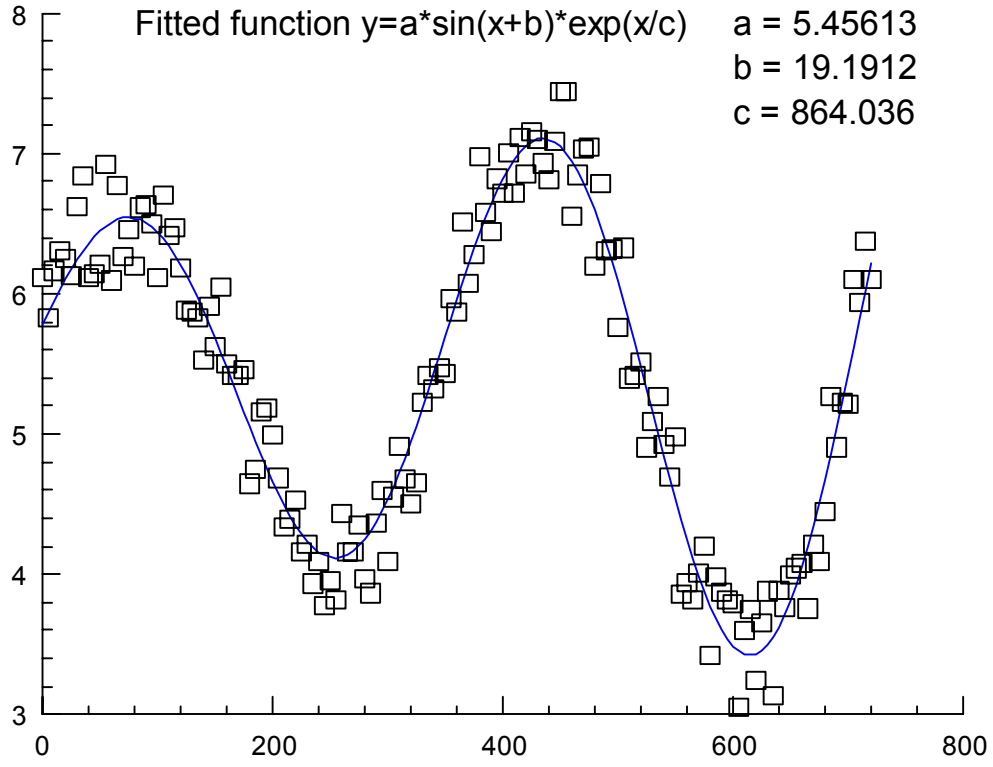
Extrema Tutorial

Data Analysis



Introduction

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.

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 (`$`).

Note	Variable names are case-insensitive, e.g., variable `x` is the same as `X`. 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.

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
```

Note	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
```

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 with value equal to π
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.

Operators

In addition to the simple arithmetic operators:

`+`	-plus	`-`	-minus
`*`	-times	`/`	-divide
`^`	-exponentiation	`()`	-grouping

there are also special vector and matrix operators:

`><`	- outer product	`<>`	- inner product

`<-`	- matrix transpose	`>-`	- matrix reflect
`/|`	- vector union	`/&`	- vector intersection
`//`	-append		

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

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)          ! If x is a vector, then so is y
```

If your source data is a monotonically increasing sequence 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)
```

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.

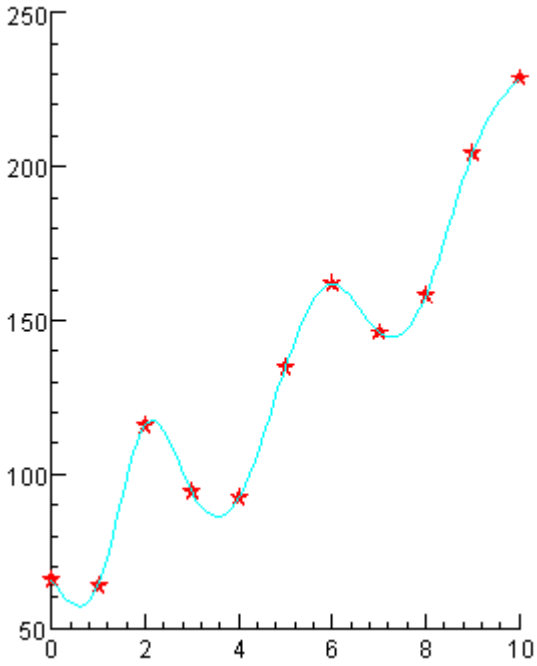
For further information, consult the **Online Help** or the **Extrema Command Reference**.

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.

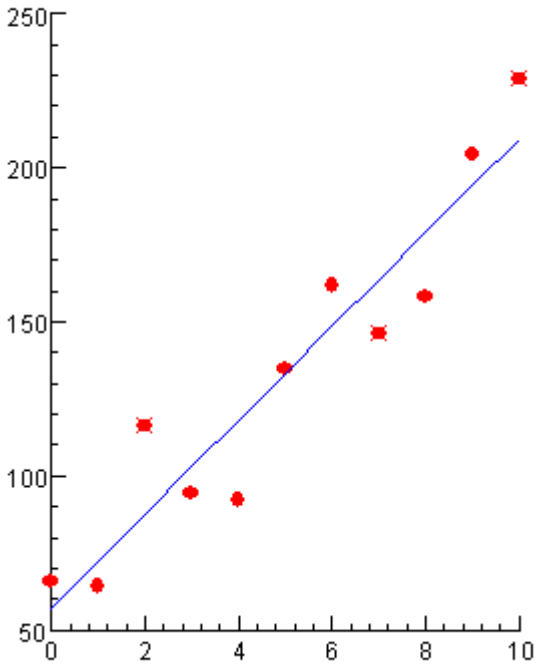
Fitting to a function

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.

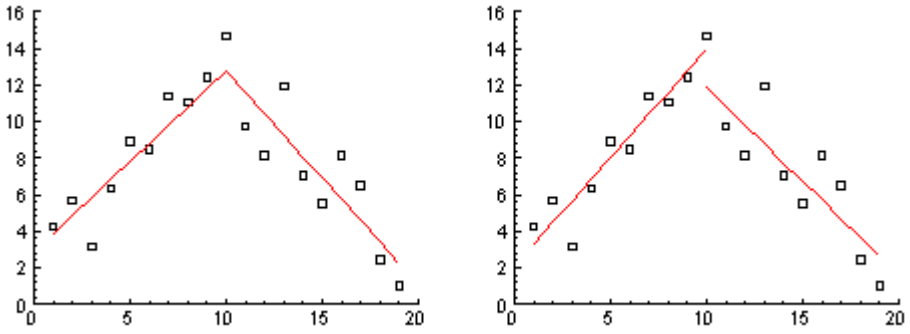
Normally, fitting results in multiple lines of text output describing the fit. The values of the free parameters, and various other values describing the accuracy of the fit, are all contained in this output. **Extrema** can optionally write some of this information into variables, for access by scripts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

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 is 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

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).

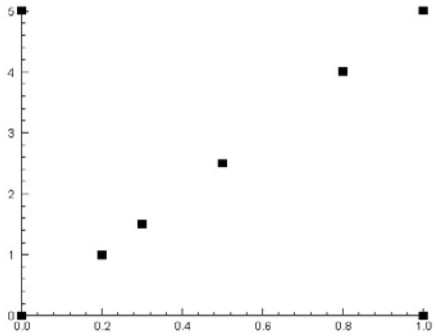
2-D interpolation

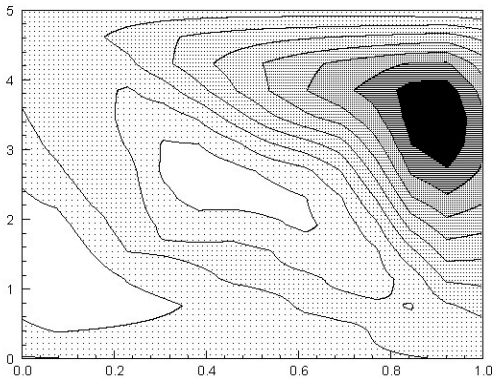
Beginning with a scattered set of 3-D data points in three vectors (say, `x`, `y`, and `z`), you can interpolate a regular matrix using the `GRID` command. The three vectors are assumed to represent scattered points, where `z[i]` is the altitude corresponding to the coordinates `(x[i],y[i])`. The set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





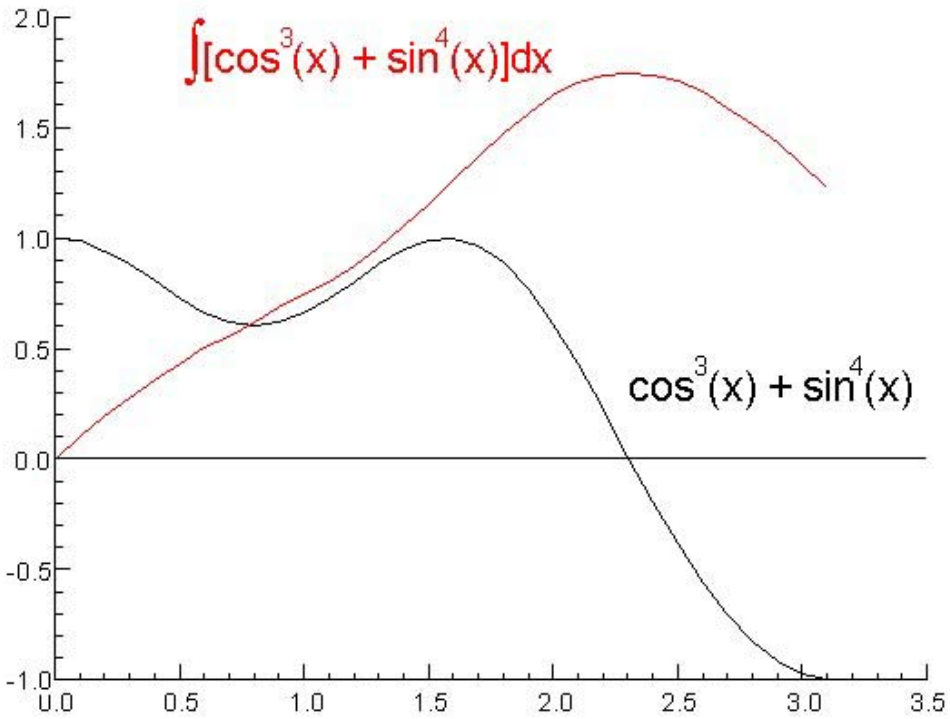
Integration

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
```