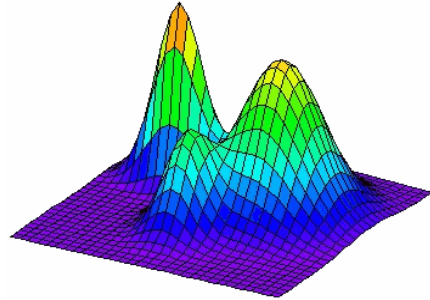




Newcastle Scientific ActiveX Control Math Library Users Manual



1.0	INTRODUCTION	2
2.0	LANGUAGE SPECIFICS	3
2.1	ARRAY CALLING CONVENTIONS	3
2.2	TYPE DEFINITIONS.....	4
2.3	VISUAL STUDIO .NET RECOMMENDATIONS	4
3.0	LEASTSQX	5
3.1	SIMPLE POLYNOMIAL FITTING.....	6
3.2	LINEAR AND NON-LINEAR USER SUPPLIED FUNCTIONS.....	7
3.3	MULTI-DIMENSIONAL FITTING	8
4.0	FFTX	9
4.1	REAL FFT.....	10
4.2	COMPLEX FFT.....	12
4.3	DATA WINDOWING.....	13
5.0	SIGPROCX	14
5.1	IIR (INFINITE IMPULSE RESPONSE) FILTER	14
5.2	FIR (FINITE IMPULSE RESPONSE) FILTER	16
5.3	EXPONENTIAL RUNNING AVERAGE FILTER	19
5.4	THE ALPHA BETA FILTER.....	21
5.5	INTERPOLATION/EXTRAPOLATION FUNCTIONS.....	23
6.0	KALMANFTX	25
7.0	SORTX	31
8.0	ENCRYPTX	33
9.0	ROOTX	34
10.0	C STYLE POINTERS FOR VISUAL BASIC 6.0	35
11.0	DISTRIBUTIONS	36
12.0	GETTING HELP	36



1.0 Introduction

Thank you for your purchase of the Newcastle Scientific ActiveX Control Math Library, which includes controls for least squares fitting, fast Fourier transforms, digital signal processing, Kalman filtering, finding root/minimum/maximum, and numeric sorting. **While these controls were specifically developed for Visual Basic (version 5.0, 6.0) and Visual C++ (version 5.0, 6.0), they can also be used with many other development environments that are fully ActiveX control capable.**

The goal of the MathLibX ActiveX library is to simplify the interface to a number of numerical functions; significantly minimizing the time a programmer will typically require developing code for complex numerical analyses. In most cases, the included functions have a one-minute learning curve.

Included with the controls is a comprehensive sample program, plus Visual Basic source code, demonstrating the majority of the functions and properties. This sample program will be found in the directory `c:\Program Files\MathLibX\sample` (unless you specified a different location during installation). If you are using Visual Basic, then simply load `sample.vpb`. If you are not using Visual Basic, then you will need to use a text editor (e.g., `notepad.exe`) to view `sample.frm` and `module1.bas`.

Please direct questions and comments to info@mathfunctions.com.



2.0 Language Specifics

Newcastle Scientific's MathlibX ActiveX controls are fully compatible with Visual Basic 5.0, Visual Basic 6.0, Visual C++ 5.0, and Visual C++ 6.0. The controls may or may not be compatible with other languages/platforms.

While the short examples in this manual and the majority of the included example project (with source code) are in Visual Basic 6.0, they can easily be converted to C/C++. Please read section 2.1, and 2.2 carefully. In addition to the VB6 examples mentioned above, several VC++ examples are also included.

2.1 Array Calling Conventions

The calling parameters utilized in the MathLibX controls make use of pointers where arrays are passed. VC++ and VB users simply pass the first element of the array. As an example, if the user has an array of data `xx(30)` that needs to be sorted, they call one of the sorting routines passing the `xx` array by passing the first element, as follows:

' Visual Basic Convention

```
Call SortX1.VectorSort(xx(0), 30)
```

// C++ Convention

```
SortX1.VectorSort(&xx[0], 30);
```

This also holds true for multidimensional arrays. If the user has a two dimensional array `xx(100,7)` and they want to sort this array by the 4th column, they would call column sort

' Visual Basic Convention

```
Call SortX1.ColumnSort(xx(0, 0), 100, 7, 4)
```

// C++ Convention

```
SortX1.ColumnSort(&xx[0][0], 100, 7, 4);
```



Important for VC++ users:

Because arrays are stored differently in Visual Basic and VC++, the user must set the property “Is_C_Code” to true if used with VC++ (the default is false so VB users don’t have to worry about this). This only applies to the Kalman filter and the sorting controls since those are the only controls that utilize multidimensional arrays.

To set the “Is_C_Code” to true, simply have, e.g., the following line in your code where it will be called prior to usage of other control functions.

```
SortX1.SetIs_c_code(true); // for the sorting control
```

Array Sizing and Indexing Convention

Unless otherwise stated, all arrays are index from zero to N-1, where N is the dimension size. This is the standard convention with VC++. With VB, when an array is declared, e.g., “redim xx(30) as double”, xx is actually an array of 0 to 30, making the actual length 31. The math library controls do not utilize the last position xx(30) but only xx(0) through xx(29). Please see the examples for clarification.

2.2 Type Definitions

Unless otherwise stated, all integer types utilized in the math libraries are of type “int” for VC++ and type “long” for Visual Basic. This is a four byte integer.

Unless otherwise stated, all floating types utilized in the math libraries are of type “double” for both VC++ and Visual Basic. This is an eight byte float.

2.3 Visual Studio .Net Recommendations

While the MathLibX ActiveX Control Math Library will work with Visual Studio .Net (with a few exceptions), it is highly recommended that the customer uses the MathLibX .Net Component Math Library instead. This Component library was specifically developed for .Net users, and includes VB.net and C#.net example applications.



3.0 LeastSqX

LeastSqX provides powerful linear and non-linear least squares algorithms with an easy to use interface. Write your own function for fitting and pass the function's name to the control, or simply rely on the default general polynomial function.

LeastSqX is a runtime invisible ActiveX control, which performs linear and non-linear least squares fitting through one very simple method call. To use LeastSqX, simply add the control to your Visual Basic project, and make a single call to the FitData method defined by the control. The call consists of passing the number of points, degrees of freedom to fit, array of independent variables (e.g., time), array of dependent variables (e.g., measurements to fit), array of uncertainties of these measurements (may be all set to 0 for non-weighted fitting), function name (or 0 for using a general polynomial fitting function), and initial guess of the coefficients (may be all 0 for linear least squares). Upon return, the coefficients are replaced by the fitted values.

The control automatically determines whether the function you are fitting to is linear or non-linear, and thus automatically determines the optimal fitting method. The user does not have to feed the control partial derivatives of the fitting function, as is often the case for non-linear problems.

While the input parameters include initial estimates of the final fitted coefficients, these may be all set to zero for linear functions.

Note, referring to a function as linear (or non-linear) has nothing to do with whether the function has powers greater than one. For instance " $a + bx + cx^2 + dx^3$ " is a linear function because each coefficient (a, b, c, d) is linear to the function. One way to tell if a function is linear is by taking the partial derivative of the function with respect to each coefficient to verify it is not dependent upon the other coefficients. For instance taking the partial derivative of the function " $a + bx + cx^2 + dx^3$ " with respect to "c" gives you simply x^2 , which is not dependent upon a, b, or d. So the function is linear.

An example of a non-linear function is $a \cdot \exp(-(x-b)^2 / c)$. Taking the partial derivative with respect to a, b, or c gives you a function that is dependent on the other two coefficients. For non-linear functions it is necessary to include estimates of the coefficients. The required closeness of these estimates is of course dependent upon the specific function, so some level of iteration may be required.



3.1 Simple Polynomial Fitting

The included general polynomial fitting function (if only "0" is passed for userfunction address) is simply: $y = a(0) + a(1)*x + a(2)*x^2 + a(3)*x^3 + \dots$ out to desired degree of freedom. See the following code fragment as an example.

```
' Linear Least Squares Example (simple polynomial)

Private Sub Perform_Fit()

' following array sizes must be set to at least the number of data points and
' must be defined as double
Static xd(100) As Double
Static yd(100) As Double
Static sigd(100) As Double

' following array sizes must be set to at least the number of coefficients and must be
' defined as double
Static co(4) As Double
Static std(4) As Double

' other required data
Dim ma As Long, npt As Long
Dim chisq as double

npt = 20 ' number of data points
ma = 3 ' degrees of freedom

' make up some example data
For i = 0 To npt - 1
    xd (i) = i
    yd (i) = 3.2 + 2.1 * xd (i) - 1.7 * xd (i) ^ 2 + 200# * Rnd
    sigd(i) = 0.0
Next i

'for general polynomial, put "0" for user function address.
' upon return the fitted coeff are returned in co(0), co(1), and co(2), and their uncertainites
' are 'returned in std(0), std(1), and std(2). The chisq value is also returned.
Call LeastSqX1.FitData(npt, ma, xd(0), yd(0), sigd(0), 0, co(0), std(0), chisq)

End Sub
```



3.2 Linear and Non-linear User Supplied Functions

Besides the simple polynomial, LeastSqX will fit data to any user-supplied function, whether linear or non-linear. In the case of non-linear, an initial guess of the coefficients must be made. LeastSqX control automatically determines if the supplied user function is linear or non-linear, and then uses the appropriate fitting algorithms. Below is a simple example demonstrating the use of LeastSqX with a non-linear user supplied function.

```
' Non-Linear Least Squares Example (Gaussian curve)
Private Sub Perform_Fit()
' array sizes must be set to at least the number of data points and must be defined as double
Static xd(100) As Double , yd(100) As Double, sigd(100) As Double

' array sizes must be set to at least the number of coefficients and must be defined as double
Static co(10) As Double, std(10) As Double

' other required data
Dim ma As Long, npt As Long, chisq as double

npt = 20 ' number of data points
ma = 4 ' degrees of freedom

' fill in array with some fake data, The truth coeff are 100, 10, 20, 1001
For i = 0 To npt - 1
    xd(i) = i
    yd(i) = 100# * Exp(-(xd(i) - 10) ^ 2) / 20#) + 1001# + (rnd(1)-.5)
    sigd(i) = 1 ' 1 sigma measurement noise of y
Next i

' initial guess of coeff (not required if function is linear)
co(0) = 70#
co(1) = 8
co(2) = 30
co(3) = 0

' call method supplied by control
' Note in this example that the user supplied function is passed to the control. However, to use a
' general polynomial, one simply passes a 0 in the place of "AddressOf userfun". To obtain a log-lin
' fit, take the log of ydat before fitting. To obtain a log-log fit,take the log of both xdat and ydat
' before fitting.
Call LeastSquare1.FitData(npt, ma, xd(0), yd(0), sigd(0), AddressOf userfun, co(0), std(0), chisq)

' now the resulting fit is in the co array, and the uncertainties is in the std array.
Text1.text= "results = " & co(0) & " " & co(1) & " " & co(2) & " " & co(3)

End Sub
```



```
' Below is the user supplied function for use with the above example. This must be in a basic module.
' All user supplied functions must be identical to this example user function shown below,
' except for the actual fit equation
Sub userfun()

Dim i As Long, j as Long
Dim ma As Long
Dim y As Double, x as Double
Static co(10) As Double

' get degrees of freedom
ma = Form1.LeastSquare1.numcoeff

' get value of x to evaluate
x = Form1.LeastSquare1.xx

' get coeff
For i = 0 To ma - 1
    co(i) = Form1.LeastSquare1.getcoefficients(i)
Next i

' evaluate y (this is your function)
y = co(0) * Exp(-(x - co(1)) ^ 2) / co(2)) + co(3)      ' <---FIT EQUATION HERE

' tell control value of y
Form1.LeastSquare1.yy = y

End Sub
```

3.3 Multi-Dimensional Fitting

Also see the demo program for an example of multidimensional least squares. This is where the dependent variable is dependent upon more than one independent variable. For instance a two dimensional Gaussian:

$$Z = co(0) * \text{Exp}(-((X - co(1)) ^ 2 - (Y - co(2)) ^ 2) / 1.5)$$

Note that Z is non-linear and is a function of both X and Y independent variables.



4.0 FFTX

The FFT control allows one to quickly calculate the Fast Fourier Transform and the Inverse Fast Fourier Transform of a set of data. The data can be a real or complex data set. The control also includes the option of Welch windowing the data prior to FFT.

The control is extremely simple to use. To FFT data, simply make a call to the FFT method, passing a vector of data, plus the number of data. Upon return, the data vector has been replaced by its FFT. To take the inverse, make a similar call to the IFFT method. To use Welch windowing prior to FFT, just set the Welch window property to true.

For complex data sets, make a call to the complex FFT method (CFFT) passing vectors of the real and imaginary components of the complex numbers. Use the complex IFFT (ICFFT) to take the complex inverse.

Most users actually find it easier to always use the complex FFT and simply set the imaginary data set to zero when requiring an FFT of real data set.

Please examine the included real and complex FFT/IFFT examples carefully to understand the correct data scaling.



4.1 Real FFT

To perform an FFT on a set of real numbers just send a vector of the data to the control as follows.

```
for i = 0 to number_of_data-1  
    dataY(i*2) = data(i)  
next I
```

```
i = FFTX1.FFT(dataY(0), number_of_data)
```

i=long. A return of zero signifies that the FFT was successful.

number_of_data = long. This is the number of data in dataY to be FFTed.

This must be a 2^N number where N is an long (e.g., 256, 512, 1024, etc.)

dataY(2*number_of_data)=double. This is the vector of data to be FFTed. Note that the data must be put in every other position (see example at the end of this section), and that dataY must be dimensioned twice as large as the dataset.

To perform an IFFT send a vector of data to the control as follows.

```
i = FFTX1.IFFT(dataY(0), number_of_data)
```

i=long. A return of zero signifies that the IFFT was successful.

number_of_data = long (see definition above for the FFT)

dataY(2*number_of_data) =double (see definition above for the FFT)The following code fragment takes the FFT and then the IFFT of a set of data.



The following example code fragment demonstrates how to take the FFT of a dataset, how to properly scale the resulting frequency domain data, and then how to take the IFFT to return back to the original time domain dataset. Please examine this example closely, especially with regard to the proper scaling.

```
' FFT/IFF Example

Private Sub Perform_FFT()

' dimension must be twice as large as the dataset
Redim dataX(2048) as double, dataY(2048) as double
Redim dataXp(2048) as double, dataYp(2048) as double ' (for plotting)

Dim i as long

' some example data
number_of_data=1024
For i = 0 To number_of_data - 1
    dataX(i)= i
    dataY(2*i) = 300.0 * Sin(2# * PI * dataX(i) / 64) 'note every other position
    dataY(2*I+1) = 0.0
Next i

' take FFT of data.
i = FFTX1.FFT(dataY(0), number_of_data)

' scale for plotting
For i = 0 To number_of_data - 1
    dataXp(i) = dataX(i) * 2 * PI / (number_of_data) ' plot x axis as frequency
    dataYp(i) = Sqr(dataY(i * 2) ^ 2 + dataY(i * 2 + 1) ^ 2) * 2 / number_of_data
Next i
plot(dataXp, dataYp) ' plot power vs. frequency

' take IFFT which will equal original dataY vector
i = FFTX1.IFFT(dataY(0), number_of_data)

End Sub
```



4.2 Complex FFT

To FFT a complex set of numbers, put the real components into one vector and the complex components into a second vector. Then make a call to the complex FFT method called "CFFT". Note that this approach also works fine with a straight real vector, simply set the imaginary vector all to zero. In fact with a real data set, CFFT is probably easier to use than FFT, and the results are identical.

```
i = FFTX1.CFFT(datareal(0), dataimaginary(0), number_of_data)
```

i=long. A return of zero signifies that the FFT was successful.

datareal(N)=double. This is the vector of the real value components of the data to be FFTed.

dataimaginary(N)=double. This is the vector of the imaginary value components of the data to be FFTed. **number_of_data** = long. This is the number of data in dataY to be FFTed. This must be a 2^N number where N is an long (e.g., 256, 512, 1024, etc.)

number_of_data = long. This is the number of data in dataY to be FFTed. This must be a 2^N number where N is an long (e.g., 256, 512, 1024, etc.)

To take the inverse FFT of a set of complex numbers put the real components into one vector and the complex components into a second vector then make a call to the complex inverse FFT method called "ICFFT" as follows.

```
i = FFTX1.ICFFT(datareal(0), dataimaginary(0), number_of_data)
```

i=long. A return of zero signifies that the FFT was successful.

datareal(N)=double. This is the vector of the real value components of the data to be FFTed.

dataimaginary(N)=double. This is the vector of the imaginary value components of the data to be FFTed. **number_of_data** = long. This is the number of data in dataY to be FFTed. This must be a 2^N number where N is an long (e.g., 256, 512, 1024, etc.)

number_of_data = long. This is the number of data in dataY to be FFTed. This must be a 2^N number where N is an long (e.g., 256, 512, 1024, etc.)



The following code fragment takes the CFFT and then the ICFFT of a set of complex data.

```
' CFFT/CIFFT Example
Private Sub Perform_CFFT()

    Redim dataX(1024) as double
    Redim datareal(1024) as double, Redim dataimaginary(1024) as double
    Redim datarealp (1024) as double, Redim dataimaginaryp(1024) as double
    Dim i as long

    ' some example data
    number_of_data=1024
    For i = 0 To number_of_data - 1
        dataX(i)=CDBL(i)
        datareal(i) = 300.0 * Sin(2# * PI * dataX(i) * 0.012)
        dataimaginary(i) = 0.0
    Next i

    ' take FFT of data
    i = FFTX1.CFFT(datareal(0), dataimaginary(0), number_of_data)
    For i = 0 To number_of_data
        dataXp(i) = dataX(i) * 2.0 * PI / (number_of_data) ' plot x axis as frequency
        datarealp (i) = datareal(i) * 2 / number_of_data
        dataimaginaryp (i) = dataimaginary(i) * 2 / number_of_data
    Next i

    plot(dataX, datarealp)
    plot(dataX, dataimaginaryp)

    ' take IFFT which will equal original datareal dataimaginary vectors
    i = FFTX1.ICFFT(datareal(0), dataimaginary(0), number_of_data)

End Sub
```

4.3 Data Windowing

To window the data with a Welch style window just set the Window_Welch property value to true before performing the FFT or CFFT.

```
FFTX1.Window_Welch = True
```



5.0 SigProcX

The SigProcX control offers a variety of time domain filtering functions, including an IIR (Infinite Impulse Response) filter, FIR (Finite Impulse Response) filter, Exponential Running Average filter, and an Alpha Beta filter. The control also includes a Cubic Spline and 2nd Order Polynomial interpolation routine. All of these functions are very easy to use, and should benefit anyone analyzing, e.g., time series data for trends, whether the data is of historical or real-time nature.

5.1 IIR (Infinite Impulse Response) Filter

An IIR filter is a digital filter that relies on both the current unfiltered data point as well as the previous filtered data for updating. In essence, an IIR has an infinite (but decaying) memory of past data. The form of an IIR is usually $y = a*(x) + b*(x-) \dots + c*(y-) \dots$

The input parameters to the IIR_Filter include the data sampling rate, the desired cutoff frequency (must be less than 1/2 the sampling rate), the data value to be filtered, and an initialization value which needs to be set to true for the first call.

The returned lowfilter is the filtered data with frequencies below the cutoff, and the highfilter is the filtered data with frequencies above the cutoff.

Important note, the actual results will show some “leakage” through the cutoff filter, which is expected with any digital time domain filter. If it is critical to minimize this leakage in an application, please consider utilizing a frequency domain filter built using the FFT control described in section 4.0 (and shown in the demo Visual Basic source code).



i = SigProcX1.IIR_Filter(Init, dataIn, frequency_cutoff, frequency_sample, lowfilter, highfilter)

i =long. A return of zero signifies that the FFT was successful.

Init = Boolean. Set to true for first call, then set to false. Tells the control to initialize the filter.

dataIn =double. Input unfiltered data value

frequency_cutoff =double. Input cutoff frequency scaled to frequency_sample. In otherwords, the true cutoff frequency in Hz is frequency_cutoff/frequency_sample

frequency_sample =double. Data sampling rate in Hz, e.g., if your data is taken at ½ second intervals then set frequency_sample to 2.

lowfilter =double. This is the filtered output data point with the frequencies above the frequency_cutoff removed (except for the leakage component).

highfilter =double. This is the filtered output data point with the frequencies below the frequency_cutoff removed (except for the leakage component).

Below is a simple code fragment demonstrating a FIR filter.

```
Private Sub Perform_Filter()  
Dim lowfilter As Double, highfilter As Double, Dim i as long, k as long  
Dim frequency_cutoff As Double, frequency_sample As Double  
Static dataY(2000) as double  
number_of_data=2000  
  
' example data  
for i=0 to number_of_data-1  
    dataY(i)=100.0*sin(.1*i) + 30*rnd;  
next i  
  
' Sampling Rate and Cutoff frequency  
frequency_sample = 1.0      ' Data points per second  
frequency_cutoff = 0.2 * frequency_sample  'Cycles per second (Hz)  
  
For i = 0 To number_of_data - 1  
    If i = 0 Then  
        k = SigProcX1.IIR_Filter(True, dataY(i), frequency_cutoff, frequency_sample, _  
            lowfilter, highfilter)  
    else  
        k = SigProcX1.IIR_Filter(False, dataY(i), frequency_cutoff, frequency_sample, _  
            lowfilter, highfilter)  
    End If  
    Print #1, dataY(i), lowfilter, highfilter  
Next I  
  
End sub
```



5.2 FIR (Finite Impulse Response) Filter

The FIR filter relies on only N numbers of data points, and thus does not have the infinite memory that the IIR filter has. The implementation of this FIR allows the user to set exact filter weights to each wavelength in order to create a custom low pass, mid pass, high pass, band pass, or notch filter.

The function internally, at initialization, takes an IFFT of the filter weights and convolves these weights in the time domain with the data to create the FIR filter.

The advantage of this FIR filter is that it has a much sharper response (less leakage) than the IIR filter shown in section 5.1. The disadvantage of this FIR filter is that there is a latency of the output that is 0.5 numweights long. For example, if numweights is set to 32, then the output filtered data corresponds to the input data point 16 points ago. If one made a plot of the unfiltered and filtered data, the filtered data will all be shifted to the left by 16 points.

The parameter “numweights” sets the kernel size of the filter. This must be set to a power of 2, e.g., 2, 4, 8, 16, 32, 64.... The larger the value is set the sharper the response (less leakage), however the slower the function will run. It is recommended that the user does not set this value above 64.

The frequency of each of the numweights filter weights are as follows:

Frequency (i) = $0.5 * i / \text{numweights}$, where $i = 0 \dots \text{numweights} - 1$

These are the primary frequencies of the filter, which means that these frequencies can be filtered out nearly completely (if desired) by setting the weights of the desired primary frequencies to zero. Note that frequencies in the actual raw data that are not a primary frequency will not be completely filtered out. Try increasing the number of weights (numweights) and widening the notch filter such that several primary frequencies are set to zero (band gap filter), until the desired performance is reached.

The input parameters of the FIR filter are as follows.

i = SigProcX1.FIR_Filter(True, dataIn , numweights, filterweights(0), filterout)

i =long. A return of zero signifies that the FFT was successful.

Init = Boolean. Set to true for first call, then set to false. Tells the control to initialize the filter.

dataIn =double. Input unfiltered data value

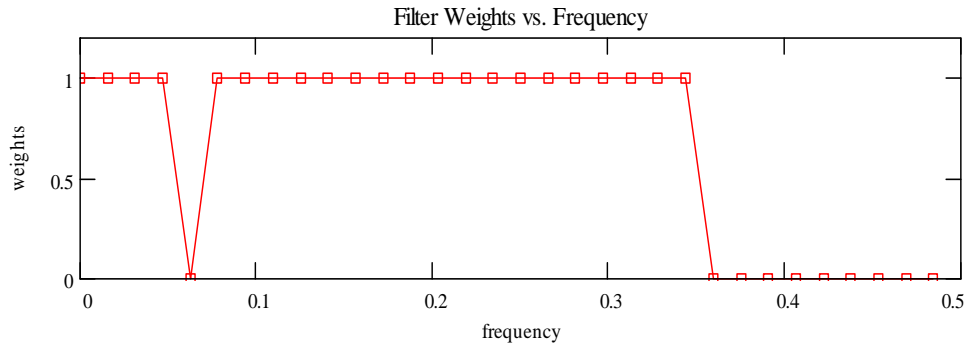
numweights = long. Number of filter components (must be a power of 2, e.g., 2, 4, 8, 16, 32...)

filterweights(0) = double. This “numweights” long vector contains the filter weights.

Filter_data=double. This is the Filter_data output.

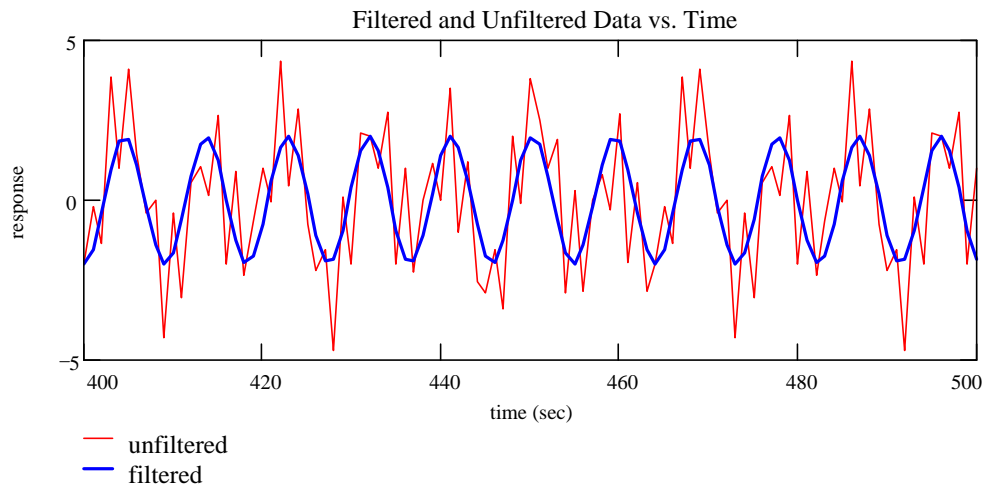


The following plot shows filter weights designed as a combined notch and low pass filter.



The next plot shows the raw unfiltered data (below equation) along with the filtered data. The filtered data nearly removes the first and third sinusoidal components of this equation (first from the notch filter and third from the low pass filter), leaving only $\sin(2\pi \cdot 0.109 \cdot i) \cdot 2$.

$$d_1 := \sin(2 \cdot \pi \cdot 0.063 \cdot i) + \sin(2 \cdot \pi \cdot 0.109 \cdot i) \cdot 2 + \cos(2 \cdot \pi \cdot 0.422 \cdot i) \cdot 2$$





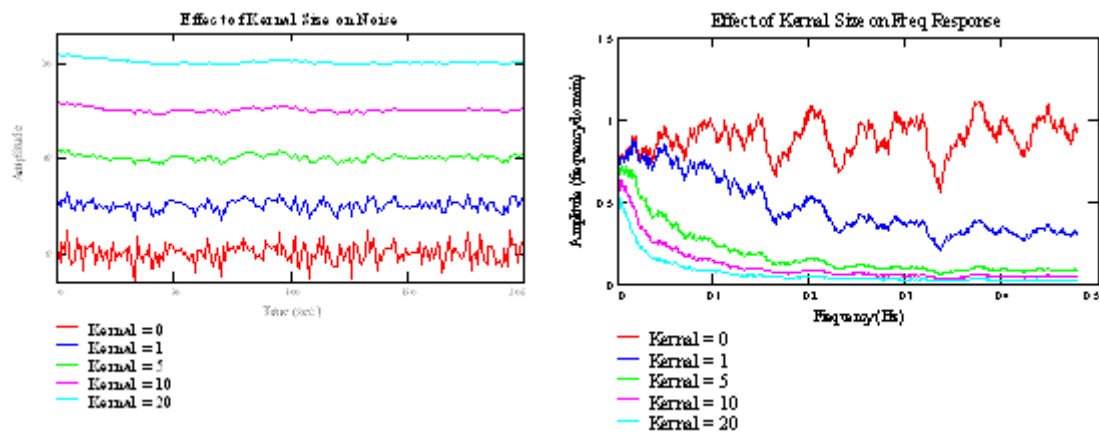
Below is a simple code fragment demonstrating a FIR filter designed to notch at .2 Hz and noise above .3 Hz

```
Private Sub Perform_Filter()  
Dim lowfilter As Double  
Dim i as long, k as long, numweights as long  
Dim frequency_cutoff As Double  
  
Static filterweights(256) As Double  
Static dataY(2000) as double  
number_of_data=2000  
  
numweights = 32  
  
' example data  
for i=0 to number_of_data-1  
    dataY(i)=100.0*sin(.1*2.0*pi*i)+ 50.0*sin(.2*2.0*pi*i)+rnd;  
next i  
  
' Set low pass (filter above .3 hz)  
For i = 0 To numweights  
    If i < CInt(0.3 * numweights) then  
        filterweights(i) = 1#  
    else  
        filterweights(i) = 0#  
    end if  
Next i  
  
' Set notch at .2 Hz  
I = CInt(0.2 * numweights)  
filterweights(i) = 0#  
  
For i = 0 To number_of_data - 1  
    If i = 0 Then  
        k = SigProcX1.FIR_Filter(True, dataY(i), numweights, filterweights(0), lowfilter)  
    Else  
        k = SigProcX1.FIR_Filter(False, dataY(i), numweights, filterweights(0), lowfilter)  
    End If  
    Print #1, dataY(i), lowfilter  
Next I  
  
End sub
```



5.3 Exponential Running Average Filter

Probably the simplest filter to run is the basic low pass exponential running average filter (which actually is a very simple type of IIR filter). Simply set kernel size property to the desired filter level, set the Init Filter property to true and start filtering. The two plots below demonstrate typical filter responses to the kernel settings, in the time (left) and the frequency (right) domains. The curves of the left plot have been vertically offset for clarity.



Since the filtering level (the kernel setting) is very dependent upon the noise level in the data as well as the function form of the underlying data, it is recommended that several trial and error iterations be made.

The relationship between the kernel size and the desired cutoff frequency is

$$\text{Kernel} = 1 / (2.0 * \text{Pi} * \text{Fc}/\text{Fs}),$$
 where Fc = frequency cutoff, and Fs = data sampling frequency.



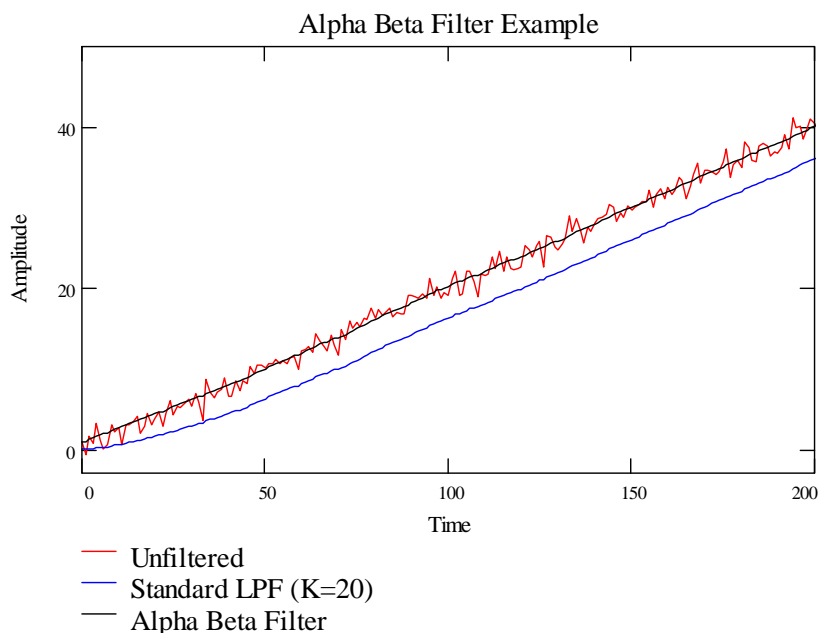
Below is a simple code fragment demonstrating a simple lowpass, midpass, and highpass filter.

```
Private Sub Perform_Filter()  
  
Dim lowfilter As Double, midfilter As Double, highfilter As Double  
dim i as long, k as long, number_of_data as long  
Static dataY(2000) as double  
number_of_data=2000  
  
' example data  
for i=0 to number_of_data-1  
    dataY(i)=100.0*sin(.1*i) + 30*rnd;  
next i  
  
' init filter (kernal =1 / (2.0 * π * Fc) , derive for desired Fc for low and high pass  
SigProcX1.High_pass_kernal = 1  
SigProcX1.Low_pass_kernal = 2  
SigProcX1.Init_Filter = True  
  
'filter data  
For i = 0 To number_of_data - 1  
    k = SigProcX1.Data(dataY(i), lowfilter, midfilter, highfilter)  
    print #1, dataY(i), lowfilter, midfilter, highfilter    ' filtered data  
next I  
  
End Sub
```



5.4 The Alpha Beta Filter

An additional filter included in the SigProcX control is the Alpha Beta filter. This filter is designed for low pass filtering while also minimizing the “data lag” effect. If your dataset has a trend, e.g., upwards, the traditional low pass filter will tend to produce a filtered value below the actual value set. The 50-day running average of the stock market is a prime example of this. The Alpha Beta filter automatically performs a 1st order correction of the trend, and thus will have little or no data lag. For this reason, Alpha Beta filters are often used in applications such as navigation and targeting. The following plot shows an unfiltered data set with an upward trend (red), the same data filtered with a standard low pass filter (kernel = 20) demonstrating the data lag (blue), and the data set filtered with the Alpha Beta filter (black). The Alpha gain is set to $2.0 * \pi * F_c / F_s$ (F_c = frequency cutoff, and F_s = data sampling frequency). This is the exact inverse of the kernel size for the exponential running average filter discussed in the previous section. The Beta Gain level is typically set to $\sim 1/10^{\text{th}}$ to $1/20^{\text{th}}$ the Alpha gain level.





Simple Example of an Alpha Beta type filter.

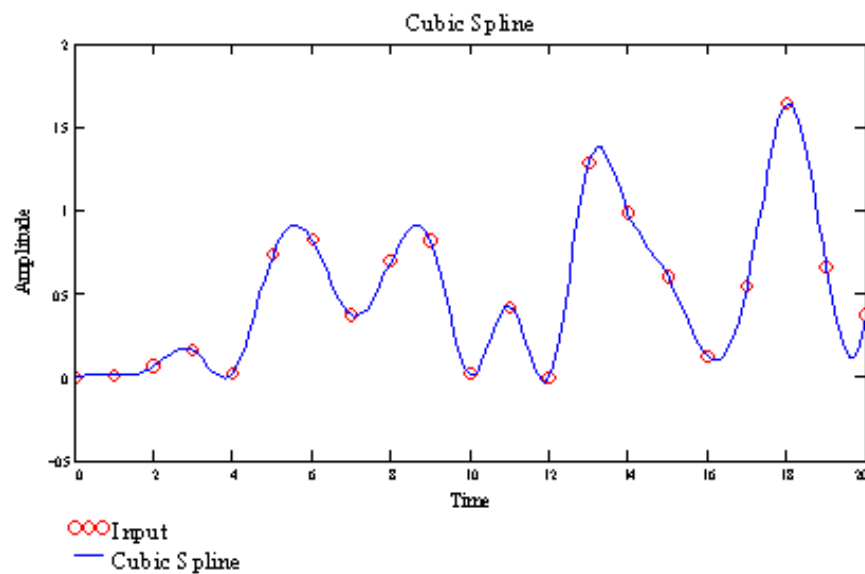
```
Private Sub Perform_Alpha_Beta()  
  
Dim alpha As Double, beta As Double  
dim i as long, k as long, number_of_data as long  
Static dataY(2000) as double  
number_of_data=2000  
  
' example data  
for i=0 to number_of_data-1  
    dataY(i)=100.0*sin(.1*i) + 30*rnd;  
next i  
  
' init filter  
SigProcX1.Alpha_Gain = .1  
SigProcX1.Beta_Gain = .01    ' Recommended Beta gain = ~ 1/10th Alpha gain  
SigProcX1.Init_Filter = True  
'filter data  
For i = 0 To number_of_data - 1  
    k = SigProcX1.alpha_beta(dataY(i), alpha, beta)  
    print #1, dataY(i), alpha , beta    ' filtered data  
next I  
  
End Sub
```



5.5 Interpolation/Extrapolation Functions

The SigProcX control includes two interpolation functions, the “Spline_interpolate” for cubic spline interpolation, and the “interpolate” function for 2nd order polynomial interpolation. While either function can also be utilized for extrapolation, the 2nd order polynomial is probably more reliable for extrapolation of simple trends. Cubic Spline is usually the preferred method for most interpolation applications.

In either of these interpolation functions, pass an array of independent data and an array of dependent data, the desired X, and the function returns the corresponding Y. The Spline_interpolate function also requires a Boolean for initialization. Set this to true whenever a new input dataset is used, and then set to false for subsequent calls with different desired X values (see example below).





Simple Example of a Cubic Spline function call

```
Private Sub Perform_Spline()  
  
ReDim xd(10) As Double, yd(10) As Double  
Dim num As Long  
Dim i As Long  
  
' example data  
num = 10  
For i = 0 To 9  
    xd(i) = i  
    yd(i) = 30# - 0.2 * i + 0.3 * i ^ 2  
Next i  
  
' interpolate the data at 5.4 (note initialization is set to true)  
MsgBox SigProcX1.Spline_interpolate (True, num, xd(0), yd(0), 5.4)  
  
' extrapolate the data at 9.7 (note initialization is set to false)  
MsgBox SigProcX1.Spline_interpolate (False, num, xd(0), yd(0), 9.7)  
  
End Sub
```

Simple Example of a Interpolation/Extrapolation function call with 2nd order Polynomial

```
Private Sub Perform_Interpolation()  
  
ReDim xd(10) As Double, yd(10) As Double  
Dim num As Long  
Dim i As Long  
  
' example data  
num = 10  
For i = 0 To 9  
    xd(i) = i  
    yd(i) = 30# - 0.2 * i + 0.3 * i ^ 2  
Next i  
  
' interpolate the data at 5.4  
MsgBox SigProcX1.Interpolate(num, xd(0), yd(0), 5.4)  
  
' extrapolate the data at 14.3  
MsgBox SigProcX1.Interpolate(num, xd(0), yd(0), 14.3)  
  
End Sub
```



6.0 KalmanFtX

The KalmanFtX control consists of a standard linear Kalman filter, allowing for propagation and updates of the state vector and covariance matrix. Inputs consists of the "Z" vector of measurements, "R" measurement noise matrix, "H" matrix of partials, "X" state vector, " Φ " propagation matrix, "P" current covariance matrix, and "Q" process noise matrix. The output consists of the updated state vector and updated covariance.

Kalman filters are similar to least squares fitting except that measurements can be incorporated into the fitting coefficients one or a few at a time, rather than "batch", or all at once as with least squares. This aspect is especially useful for real-time applications such as those found in navigation, guidance, and control.

This ActiveX control is not designed to serve as a substitute for a proper understanding of Kalman filters. The control will certainly be useful to users with a basic or advanced working knowledge of Kalman filters, as well as a beginner user, provided they have additional literature such as the reference book recommended on page one of this manual.

A call to KalmanFtX looks something like this. "i" is returned as a zero for success, and a -1 if the update failed (e.g., singular P matrix).

```
i = KalmanFtX1.Kalman(Z(0), R(0,0), X(0), P(0,0), H(0,0), Q(0,0), Phi(0, 0), Nx, Nz)
```

Please see the included Kalman Filter example source code in the example included demo program.



The input matrices and vectors are defined here. All are double precision. N_x and N_z are longs. Please see the sample program for a demonstration on the declarations and calling sequence. Upon return both the X state vector and the P covariance matrix have been propagated and updated.

Z is the vector of measurements, of length N_z .

R is the matrix of measurement noise (σ^2) of size N_z, N_z .

X is the vector of states, of length N_x .

P is the covariance matrix, of size N_x, N_x .

H is the matrix of partials, of size N_z, N_x .

Q is the matrix of process noise of size N_x, N_x .

Φ is the propagation or dynamics matrix and is of size N_x, N_x .

N_x is the number of states.

N_z is the number of measurements at each update.

The algorithms utilized by KalmanFtX are listed below.



State Propagation - *Propagates the state vector to the time of the current measurement.*

$$X(-)_k = \Phi_{k-1} \cdot X(+)_k$$

Covariance Propagation - *Propagates the covariance matrix to the time of the current measurement and adds process noise.*

$$P(-)_k = \Phi_{k-1} \cdot P(+)_k \cdot \Phi_{k-1}^T + Q_{k-1}$$

Kalman Filter Gain - *Derives the Gain "weighting" matrix.*

$$K_k = P(-)_k H_k^T [H_k P(-)_k H_k^T + R_k]^{-1}$$

Covariance Update - *Updates the covariance matrix.*

$$P(+)_k = [I - K_k \cdot H_k] P(-)_k$$

State Vector Update - *Updates the state vector with the current measurements, weighted by the gain matrix.*

$$X(+)_k = X(-)_k + K_k [Z_k - H_k X(-)_k]$$



Below is an example of a simple Kalman filter design utilized to update measurements of position and velocity to a three state system consisting of position, velocity, and acceleration. These measurements are obtained every ΔT seconds.

The Φ propagation matrix would be as shown here. The first row shows how the position changes (1 times the position plus ΔT times the velocity + $.5 \cdot \Delta T^2$ times the acceleration). The next row shows how the velocity changes (0 times the position plus 1 times the velocity plus ΔT times the acceleration). The last row shows that the acceleration stays constant (0 times the position plus 0 times the velocity plus 1 times the acceleration). For this example, the measurements are obtained every 10 seconds, so $\Delta T = 10$.

$$\Phi = \begin{pmatrix} 1 & \Delta T & .5 \cdot \Delta T^2 \\ 0 & 1 & \Delta T \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 10 & 50 \\ 0 & 1 & 10 \\ 0 & 0 & 1 \end{pmatrix}$$

Note that just because we are modeling the acceleration as a constant does not mean the acceleration state can not change measurement to measurement. The process noise (Q) introduced to the covariance matrix at each update allows the acceleration state to "drift". One of the most complex aspects of Kalman filters is how the process noise and the propagation matrix work together.

For this example we use a simple process noise of .1 (m/sec/sec) 1 sigma on the acceleration term only. Note that the value is squared in the matrix since the Q consists of variance terms (same for the R and P matrices described below).

$$Q = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & .1^2 \end{pmatrix}$$

The H matrix represents the partial derivative of the measurements with respect to the states, e.g., $H(j,k) = dZ_j/dX_k$, where Z is the measurement vector and X is the state vector. In this case the measurements consist of one position and one velocity value (at each update). So the H matrix would consist of the following values. Recall that H has the dimension of number measurements by number of states. Since one of the measurements (1st row) consists of the position then $dZ_0/dX_0 = 1$ (both Z and X are positions). Also $dZ_0/dX_1 = 0$ and $dZ_0/dX_2 = 0$ since change in position does not effect velocity and acceleration (in this model). Similarly for the velocity measurement (second row).



$$H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}$$

The R matrix consists of the measurement noise. If the measurements are uncorrelated (probably the most common situation) then the matrix will be diagonal. In this example we will use the measurement noise one sigma of 2 meters for position and .5 m/sec for velocity. The R matrix would then be:

$$R = \begin{pmatrix} 2^2 & 0 \\ 0 & .5^2 \end{pmatrix}$$

Lets say that this is the first update, so the input X state vector consists of just a crude estimate of the position, velocity, and acceleration at time "0". In this case the covariance matrix has not yet been propagated or updated. In most situations like this the covariance matrix is initialized with large values on the diagonal, and zeros on the off diagonals. The values utilized are not so critical as long as they are at least as large as your uncertainty and not so large as to cause a singular matrix problem. If you feel your initial guess for the position is +/- 20 m and velocity is +/- 5 m/sec and acceleration is +/- .5 m/sec/sec then an example initial covariance for this problem could be something significantly larger then these. For example:

$$P = \begin{pmatrix} 100^2 & 0 & 0 \\ 0 & 10^2 & 0 \\ 0 & 0 & 1^2 \end{pmatrix}$$

For the purpose of this example take the initial estimate of the state vector (position, velocity, acceleration) as:

$$X = (0 \ 0 \ .2)$$



The measurements are received at 10-second intervals consisting of the following sets:

1st measurement set =

$$Z = (11.3 \quad 1.8)$$

After the call to KalmanFtX1 ($i = \text{KalmanFtX1.Kalman}(Z, R, X, P, H, Q, \Phi, N_x, N_z)$) the covariance vector becomes

$$P = \begin{pmatrix} 3.9986 & 0.0007 & -0.0088 \\ 0.0007 & 0.2494 & 0.0166 \\ -0.0088 & 0.0166 & 0.4556 \end{pmatrix}$$

and the state vector becomes

$$X = (11.299 \quad 1.801 \quad 0.184)$$

Note that this represents the estimate of the position velocity and acceleration of the object 10 seconds after the time where the system was initialized at $X = (0,0,2)$.

Now we can simply recall KalmanFtX1 with the new measurements 10 seconds later using the updated states and covariance from the previous measurement. The Φ propagation matrix stays the same as long as the updates are every 10 seconds. The Q, and R matrix will stay the same (in most implementations). The H matrix will stay the same if this is a linear system. Many systems are non-linear and require recalculations of the H matrix using the most recent state vector estimates. If the update period changes then recalculate the propagation matrix with the new ΔT . Also for non-linear systems the propagation matrix will change with the state vector.



7.0 SortX

The SortX control contains two types of functions, those utilized for sorting a vector (VectorSort), and those utilized for sorting an array of data based on a selected column (ColumnSort).

A property called SortOrder sets whether any future call to the various sorting functions will return the data sorted in ascending or descending order. The default is ascending order.

SortX1.SortOrder = 0 ' for Ascending order (default value)

SortX1.SortOrder = 1 ' for Descending order

The VectorSort functions include VectorSort_Double, VectorSort_Single, VectorSort_Long, VectorSort_Integer, and are used for vectors declared as double, single, long, and integer respectively.

```
Private Sub Perform_Sort()  
  
    ReDim xx(num) As Double  
    Dim num as long  
  
    ' makeup data  
    num=100  
    For i = 0 To num - 1  
        xx(i) = -10 + Rnd(1) * 20#  
    Next i  
  
    ' sort data, after call xx is sorted  
    Call SortX1.VectorSort_Double(xx(0), num)  
  
End Sub
```



The ColumnSort functions include ColumnSort_Double, ColumnSort_Single, ColumnSort_Long, ColumnSort_Integer, and are used for arrays declared as double, single, long, and integer respectively.

```
Private Sub Perform_Column_Sort()

    Dim numrow As Long, numcol As Long, selectedcol As Long

    numrow = 30
    numcol = 4
    selectedcol = 3

    ' important, array must be dimensioned exactly by numrow x numcol
    ReDim xxx(numrow, numcol) As Double

    ' makeup data
    For i = 0 To numrow - 1
        For j = 0 To numcol - 1
            xxx(i, j) = -10 + Rnd(1) * 20#
        Next j
    Next i

    ' sort array row by row based on column "selectedcol", after call xxx is sorted
    Call SortX1.ColumnSort_Double(xxx(0,0), numrow, numcol, selectedcol)

End Sub
```



8.0 EncryptX

The EncryptX control is utilized to encrypt a text field. This control is currently only compatible with Visual Basic (will not work with VC++). The user can encrypt entire ASCII based documents with one simple call. If someone tries to decrypt the encrypted text with the wrong password, the text remains unchanged (remains encrypted).

```
Private Sub Click()  
Dim sdata As string  
Dim password as string  
  
Sdata="Hello, have a nice day" ' some example text  
Password="yourpassword" ' user selected password  
Call EncryptX1.encrypt ("Password ", Sdata, True)  
msgbox Sdata  
Call EncryptX1.encrypt ("Password ", Sdata, false)  
msgbox Sdata  
  
End Sub
```



9.0 RootX

The RootX control is utilized to find a root (zero crossing) of a user supplied function. The control also has a function for finding the minimum or maximum of a function quickly and accurately. These are demonstrated below.

```
Sub findminmaxroot()

Dim ilong As Long
Dim xfinal As Double, yfinal As Double, Dim x1 As Double, x2 As Double, Dim x3 As Double

' initial guess (must bracket root or min or max)
x1 = -1000#
x2 = 1000#

' required accuracy
x3 = 0.0001

' find root (see user supplied function roottest, below)
ilong = RootX1.rootfind(x1, x2, x3, xfinal, yfinal, AddressOf roottest)
MsgBox "Root X Y = " & xfinal & " " & yfinal

' find min (see user supplied function mintest, below)
ilong = RootX1.minmaxfind(x1, x2, x3, xfinal, yfinal, AddressOf mintest)
MsgBox "Minimum X Y = " & xfinal & " " & yfinal

' find max (see user supplied function maxtest, below)
ilong = RootX1.minmaxfind(x1, x2, x3, xfinal, yfinal, AddressOf maxtest)
MsgBox "Maximum X Y = " & xfinal & " " & yfinal

End sub
```

Example User Supplied Functions (must be visible to the above function)

```
Function mintest(ByVal x As Double) As Double
    mintest = (x - 3.45) ^ 2
End Function

Function maxtest(ByVal x As Double) As Double
    ' for maximum, multiply function by -1 (turning max into min).
    maxtest = -1# * ((x - 3.45) ^ 2)
End Function

Function roottest(ByVal x As Double) As Double
    roottest = (x - 3.45)
End Function
```



10.0 C Style Pointers for Visual Basic 6.0

Includes with your MathLibX package is the Newcastle Scientific Pointer Package. While this package is not exactly part of the controls, it may be useful for some programmers, and is included here at no extra cost. Please see pointers.zip.

While it is possible to utilize "C" style pointers using VB, very little is documented about this capability. The usefulness of pointers in Visual Basic include:

- 1) Being able to pass an address of an array or vector of data from C/C++ code to a Visual Basic coded ActiveX control or Dynamic Link Library.
- 2) Being able to pass specific elements (e.g., elements 5-10) of an array or vector without passing the entire array or vector. This is especially useful for numerical analyses of selected parts of very large data arrays.

The Newcastle Scientific Pointer Package includes class modules (with source code) for creating integer, long, single, and double precision pointers, plus straightforward examples of each. Additional instructions can be found in the package (pointers.zip).

**'The following VB6 example prints out 0 through 9 for
'the first call, and 50 through 59 for the second call.**

```
Sub pointertest()
    ReDim x(100) As Double
    for i = 0 To 99
        x(i) = i
    Next i
    Call doublevector(x(0)) ' pass address of first element
    Call doublevector(x(50)) ' pass address of 51th element
End Sub

Sub doublevector(x As Double)
    ' define and set a new pointer from the Newcastle Scientific cpointer_double class
    Dim Data As New cpointer_double
    Call Data.setpointer(x)
    for i = 0 to 9
        debug.print i, Data.ptr(i)
    next i
End Sub
```



11.0 Distributions

To distribute a Visual Basic application that contains the MathLibX controls, you must include the following two files:

c:\windows\system\mathlib.ocx (must be registered)
c:\windows\system\mathdll.dll

To distribute a non Visual Basic application that contains the MathLibX controls, you must distribute the above two files, plus (in some cases) the standard 32 bit Visual Basic run time Libraries (see www.microsoft.com for info and copies of these). If you have any questions or trouble please contact us at info@mathfunctions.com.

12.0 Getting Help

If you are experiencing any kind of trouble with installation, compilation, execution, etc, please contact us at info@mathfunctions.com. In the overwhelming majority of the cases we can respond within 24 hours. Please leave your name and phone number.