LabWindows®/CVI

Advanced Analysis Library Reference Manual

July 1996 Edition

Part Number 320686C-01

Internet Support

GPIB: gpib.support@natinst.com
DAQ: daq.support@natinst.com
VXI: vxi.support@natinst.com
LabVIEW: lv.support@natinst.com
LabWindows: lw.support@natinst.com
Lookout: lookout.support@natinst.com
HiQ: hiq.support@natinst.com
VISA: visa.support@natinst.com
FTP Site: ftp.natinst.com
Web Address: www.natinst.com

Bulletin Board Support

BBS United States: (512) 794-5422 or (800) 327-3077
BBS United Kingdom: 01635 551422
BBS France: 1 48 65 15 59

FaxBack Support

(512) 418-1111

Telephone Support (U.S.)

Tel: (512) 795-8248
Fax: (512) 794-5678

International Offices

Australia 03 9 879 9422, Austria 0662 45 79 90 0, Belgium 02 757 00 20,
Canada (Ontario) 519 622 9310, Canada (Québec) 514 694 8521, Denmark 45 76 26 00,
Finland 90 527 2321, France 1 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186,
Italy 02 413091, Japan 03 5472 2970, Korea 02 596 7456, Mexico 95 800 010 0793,
Netherlands 0348 433466, Norway 32 84 84 00, Singapore 2265886, Spain 91 640 0085,
Sweden 08 730 49 70, Switzerland 056 200 51 51, Taiwan 02 377 1200, U.K. 01635 523545

National Instruments Corporate Headquarters

6504 Bridge Point Parkway Austin, TX 78730-5039 Tel: (512) 794-0100

Warranty

The media on which you receive National Instruments software are warranted not to fail to execute programming instructions, due to defects in materials and workmanship, for a period of 90 days from date of shipment, as evidenced by receipts or other documentation. National Instruments will, at its option, repair or replace software media that do not execute programming instructions if National Instruments receives notice of such defects during the warranty period. National Instruments does not warrant that the operation of the software shall be uninterrupted or error free.

A Return Material Authorization (RMA) number must be obtained from the factory and clearly marked on the outside of the package before any equipment will be accepted for warranty work. National Instruments will pay the shipping costs of returning to the owner parts which are covered by warranty.

National Instruments believes that the information in this manual is accurate. The document has been carefully reviewed for technical accuracy. In the event that technical or typographical errors exist, National Instruments reserves the right to make changes to subsequent editions of this document without prior notice to holders of this edition. The reader should consult National Instruments if errors are suspected. In no event shall National Instruments be liable for any damages arising out of or related to this document or the information contained in it.

EXCEPT AS SPECIFIED HEREIN, NATIONAL INSTRUMENTS MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AND SPECIFICALLY DISCLAIMS ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. CUSTOMER'S RIGHT TO RECOVER DAMAGES CAUSED BY FAULT OR NEGLIGENCE ON THE PART OF NATIONAL INSTRUMENTS SHALL BE LIMITED TO THE AMOUNT THERETOFORE PAID BY THE CUSTOMER. NATIONAL INSTRUMENTS WILL NOT BE LIABLE FOR DAMAGES RESULTING FROM LOSS OF DATA, PROFITS, USE OF PRODUCTS, OR INCIDENTAL OR CONSEQUENTIAL DAMAGES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF. This limitation of the liability of National Instruments will apply regardless of the form of action, whether in contract or tort, including negligence. Any action against National Instruments must be brought within one year after the cause of action accrues. National Instruments shall not be liable for any delay in performance due to causes beyond its reasonable control. The warranty provided herein does not cover damages, defects, malfunctions, or service failures caused by owner's failure to follow the National Instruments installation, operation, or maintenance instructions; owner's modification of the product; owner's abuse, misuse, or negligent acts; and power failure or surges, fire, flood, accident, actions of third parties, or other events outside reasonable control.

Copyright

Trademarks

WARNING REGARDING MEDICAL AND CLINICAL USE OF NATIONAL INSTRUMENTS PRODUCTS

National Instruments products are not designed with components and testing intended to ensure a level of reliability suitable for use in treatment and diagnosis of humans. Applications of National Instruments products involving medical or clinical treatment can create a potential for accidental injury caused by product failure, or by errors on the part of the user or application designer. Any use or application of National Instruments products for or involving medical or clinical treatment must be performed by properly trained and qualified medical personnel, and all traditional medical safeguards, equipment, and procedures that are appropriate in the particular situation to prevent serious injury or death should always continue to be used when National Instruments products are being used. National Instruments products are NOT intended to be a substitute for any form of established process, procedure, or equipment used to monitor or safeguard human health and safety in medical or clinical treatment.

Contents

Figures

Tables

Equations

About This Manual

The *LabWindows/CVI Advanced Analysis Library Reference Manual* describes the functions in the LabWindows/CVI Advanced Analysis Library. To use this manual effectively, you should be familiar with the material presented in the *LabWindows/CVI User Manual*, and with the LabWindows/CVI software. Please refer to the *LabWindows/CVI User Manual* for specific instructions on operating LabWindows/CVI.

Organization of This Manual

The *LabWindows/CVI Advanced Analysis Library Reference Manual* is organized as follows:

- Chapter 1, *Advanced Analysis Library Overview*, contains a brief product overview and general information about the Advanced Analysis Library functions and panels.
- Chapter 2, *Advanced Analysis Library Function Reference*, contains a brief explanation of each of the functions in the LabWindows/CVI Advanced Analysis Library. The LabWindows/CVI Advanced Analysis Library functions are arranged alphabetically.
- Appendix A, *Error Codes*, contains error codes returned by the Advanced Analysis Library functions.
- Appendix B, *Customer Communication*, contains forms you can use to request help from National Instruments or to comment on our products and manuals.
- The *Glossary* contains an alphabetical list and description of terms used in this manual, including acronyms, abbreviations, metric prefixes, mnemonics, and symbols.
- The *Index* contains an alphabetical list of key terms and topics in this manual, including the page where you can find each one.

Conventions Used in This Manual

The following conventions are used in this manual:

bold	Bold text denotes a parameter, menu item, return value, function panel item, or dialog box button or option.
italic	Italic text denotes emphasis, a cross reference, or an introduction to a key concept.
bold italic	Bold italic text denotes a note, caution, or warning.

`monospace` Text in this font denotes text or characters that you should literally enter from the keyboard. Sections of code, programming examples, and syntax examples also appear in this font. This font also is used for the proper names of disk drives, paths, directories, programs, subprograms, subroutines, device names, variables, filenames, and extensions, and for statements and comments taken from program code.

`italic monospace` Italic text in this font denotes that you must supply the appropriate words or values in the place of these items.

< > Angle brackets enclose the name of a key. A hyphen between two or more key names enclosed in angle brackets denotes that you should simultaneously press the named keys, for example, <Ctrl-Alt-Delete>.

» The **»** symbol leads you through nested menu items and dialog box options to a final action. The sequence **File » Page Setup » Options » Substitute Fonts** directs you to pull down the **File** menu, select the **Page Setup** item, select **Options**, and finally select the **Substitute Fonts** option from the last dialog box.

paths Paths in this manual are denoted using backslashes (\) to separate drive names, directories, and files, as in, `drivename\dir1name\dir2name\myfile`.

Acronyms, abbreviations, metric prefixes, mnemonics, and symbols, and terms are listed in the *Glossary*.

Related Documentation

The following documents contain information that you may find helpful as you use advanced analysis functions.

- Baher, H. *Analog & Digital Signal Processing*. New York: John Wiley & Sons. 1990.
- Bates, D.M. and Watts, D.G. *Nonlinear Regression Analysis and its Applications*. New York: John Wiley & Sons. 1988.
- Bracewell, R.N. “Numerical Transforms.” *Science*. Science-248. 11 May 1990.
- Burden, R.L. & Faires, J.D. *Numerical Analysis.* Third Edition. Boston: Prindle, Weber & Schmidt. 1985.
- Chen, C.H. et al. *Signal Processing Handbook*. New York: Marcel Dekker, Inc. 1988.
- DeGroot, M. *Probability and Statistics*, 2nd ed. Reading, Massachusetts: Addison-Wesley Publishing Co. 1986.
- Dowdy, S. and Wearden, S. *Statistics for Research*, 2nd ed. New York: John Wiley & Sons. 1991.

- Dudewicz, E.J. and Mishra, S.N. *Modern Mathematical Statistics.* New York: John Wiley & Sons, 1988.
- Duhamel, P. et al. "On Computing the Inverse DFT." *IEEE Transactions on ASSP*. ASSP-34 (1986): 1 (February).
- Dunn, O. and Clark, V. *Applied Statistics: Analysis of Variance and Regression*, 2nd ed. New York: John Wiley & Sons. 1987.
- Elliot, D.F. *Handbook of Digital Signal Processing Engineering Applications*. San Diego: Academic Press. 1987.
- Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," *Proceedings of the IEEE-66* (1978)-1.
- Maisel, J.E. "Hilbert Transform Works With Fourier Transforms to Dramatically Lower Sampling Rates." *Personal Engineering and Instrumentation News*. PEIN-7 (1990): 2 (February).
- Miller, I. and Freund, J.E. *Probability and Statistics for Engineers*. Englewood Cliffs, N.J.: Prentice-Hall, Inc. 1987.
- Neter, J. et al. *Applied Linear Regression Models*. Richard D. Irwin, Inc. 1983.
- Neuvo, Y., Dong, C.-Y., and Mitra, S.K. "Interpolated Finite Impulse Response Filters," *IEEE Transactions on ASSP*. ASSP-32 (1984): 6 (June).
- O'Neill, M.A. "Faster Than Fast Fourier." BYTE. (1988) (April).
- Oppenheim, A.V. & Schafer, R.W. *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1989.
- Parks, T.W. and Burrus, C.S. *Digital Filter Design*. John Wiley & Sons, Inc.: New York. 1987.
- Pearson, C.E. *Numerical Methods in Engineering and Science.* New York: Van Nostrand Reinhold Co. 1986.
- Press, W.H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press. 1988.
- Rabiner, L.R. & Gold, B. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1975.
- Sorensen, H.V. et al. "On Computing the Split-Radix FFT." *IEEE Transactions on ASSP*. ASSP-34 (1986):1 (February).
- Sorensen, H.V. et al. "Real-Valued Fast Fourier Transform Algorithms." *IEEE Transactions on ASSP*. ASSP-35 (1987): 6 (June).
- Stoer, J. and Bulirsch, R. *Introduction to Numerical Analysis*. New York: Springer-Verlag. 1987.

- Vaidyanathan, P.P. *Multirate Systems and Filter Banks*. Englewood Cliffs, New Jersey: Prentice Hall. 1993.
- Wichman, B. and Hill, D. “Building a Random-Number Generator: A Pascal routine for very-long-cycle random-number sequences.” *BYTE,* March 1987, pp 127-128.

Customer Communication

National Instruments wants to receive your comments on our products and manuals. We are interested in the applications you develop with our products, and we want to help you if you have problems with them. To make it easy for you to contact us, this manual contains comment and technical support forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.

Chapter 1
Advanced Analysis Library Overview

This chapter contains a brief product overview and general information about the Advanced Analysis Library functions and panels.

Product Overview

The LabWindows Advanced Analysis Library adds additional analysis functions to the standard LabWindows/CVI Analysis Library. The Advanced Analysis Library includes functions for signal generation, one-dimensional (1D) and two-dimensional (2D) array manipulation, complex operations, signal processing, statistics, and curve-fitting.

The Advanced Analysis Library Function Panels

The Advanced Analysis Library function panels are grouped in a tree structure according to the types of operations performed. The Advanced Analysis Library Function Tree is shown in Table 1-1.

The first- and second-level bold headings in the tree are the names of function classes and subclasses. Function classes and subclasses are groups of related function panels. The third-level headings in plain text are the names of individual function panels. Each analysis function panel generates one analysis function call. The names of the corresponding analysis function calls are in bold italics to the right of the function panel names.

Table 1-1. The Advanced Analysis Library Function Tree

Analysis	
Signal Generation	
Impulse	***Impulse***
Pulse	***Pulse***
Ramp	***Ramp***
Triangle	***Triangle***
Sine Pattern	***SinePattern***
Uniform Noise	***Uniform***
White Noise	***WhiteNoise***
Gaussian Noise	***GaussNoise***
Arbitrary Wave	***ArbitraryWave***
Chirp	***Chirp***

(continues)

Table 1-1. The Advanced Analysis Library Function Tree (Continued)

Sawtooth Wave	***SawtoothWave***
Sinc Waveform	***Sinc***
Sine Wave	***SineWave***
Square Wave	***SquareWave***
Triangle Wave	***TriangleWave***
Array Operations	
1D Operations	
1D Clear Array	***Clear1D***
1D Set Array	***Set1D***
1D Copy Array	***Copy1D***
1D Array Addition	***Add1D***
1D Array Subtraction	***Sub1D***
1D Array Multiplication	***Mul1D***
1D Array Division	***Div1D***
1D Absolute Value	***Abs1D***
1D Negative Value	***Neg1D***
1D Linear Evaluation	***LinEv1D***
1D Polynomial Evaluation	***PolyEv1D***
1D Scaling	***Scale1D***
1D Quick Scaling	***QScale1D***
1D Maximum & Minimum	***MaxMin1D***
1D Sum of Elements	***Sum1D***
1D Product of Elements	***Prod1D***
1D Array Subset	***Subset1D***
1D Reverse Array Order	***Reverse***
1D Shift Array	***Shift***
1D Clip Array	***Clip***
1D Sort Array	***Sort***
1D Vector Normalization	***Normal1D***
2D Operations	
2D Array Addition	***Add2D***
2D Array Subtraction	***Sub2D***
2D Array Multiplication	***Mul2D***
2D Array Division	***Div2D***
2D Linear Evaluation	***LinEv2D***
2D Polynomial Evaluation	***PolyEv2D***
2D Scaling	***Scale2D***
2D Quick Scaling	***QScale2D***
2D Maximum & Minimum	***MaxMin2D***
2D Sum of Elements	***Sum2D***
2D Matrix Normalization	***Normal2D***

(continues)

Table 1-1. The Advanced Analysis Library Function Tree (Continued)

Complex Operations	
Complex Numbers	
Complex Addition	***CxAdd***
Complex Subtraction	***CxSub***
Complex Multiplication	***CxMul***
Complex Division	***CxDiv***
Complex Reciprocal	***CxRecip***
Complex Square Root	***CxSqrt***
Complex Logarithm	***CxLog***
Complex Natural Log	***CxLn***
Complex Power	***CxPow***
Complex Exponential	***CxExp***
Rectangular to Polar	***ToPolar***
Polar to Rectangular	***ToRect***
1D Complex Operations	
1D Complex Addition	***CxAdd1D***
1D Complex Subtraction	***CxSub1D***
1D Complex Multiplication	***CxMul1D***
1D Complex Division	***CxDiv1D***
1D Complex Linear Evaluation	***CxLinEv1D***
1D Rectangular to Polar	***ToPolar1D***
1D Polar to Rectangular	***ToRect1D***
Signal Processing	
Frequency Domain	
FFT	***FFT***
Inverse FFT	***InvFFT***
Real Valued FFT	***ReFFT***
Real Valued Inverse FFT	***ReInvFFT***
Power Spectrum	***Spectrum***
FHT	***FHT***
Inverse FHT	***InvFHT***
Cross Spectrum	***CrossSpectrum***
Time Domain	
Convolution	***Convolve***
Correlation	***Correlate***
Integration	***Integrate***
Differentiate	***Difference***
Pulse Parameters	***PulseParam***
Decimate	***Decimate***
Deconvolve	***Deconvolve***
Unwrap Phase	***UnWrap1D***

(continues)

Table 1-1. The Advanced Analysis Library Function Tree (Continued)

IIR Digital Filters	
Cascade Filter Functions	
Bessel Cascade Coeff	***Bessel_CascadeCoef***
Butterworth Cascade Coeff	***Bw_CascadeCoef***
Chebyshev Cascade Coeff	***Ch_CascadeCoef***
Inv Chebyshev Cascade Coeff	***InvCh_CascadeCoef***
Elliptic Cascade Coeffs	***Elp_CascadeCoef***
IIR Cascade Filtering	***IIRCascadeFiltering***
Filter Information Utilities	
Allocate Filter Information	***AllocIIRFilterPtr***
Reset Filter Information	***ResetIIRFilter***
Free Filter Information	***FreeIIRFilterPtr***
Cascade to Direct Coefficients	***CascadeToDirectCoef***
One-step Filter Functions	
Lowpass Butterworth	***Bw_LPF***
Highpass Butterworth	***Bw_HPF***
Bandpass Butterworth	***Bw_BPF***
Bandstop Butterworth	***Bw_BSF***
Lowpass Chebyshev	***Ch_LPF***
Highpass Chebyshev	***Ch_HPF***
Bandpass Chebyshev	***Ch_BPF***
Bandstop Chebyshev	***Ch_BSF***
Lowpass Inverse Chebyshev	***InvCh_LPF***
Highpass Inverse Chebyshev	***InvCh_HPF***
Bandpass Inverse Chebyshev	***InvCh_BPF***
Bandstop Inverse Chebyshev	***InvCh_BSF***
Lowpass Elliptic	***Elp_LPF***
Highpass Elliptic	***Elp_HPF***
Bandpass Elliptic	***Elp_BPF***
Bandstop Elliptic	***Elp_BSF***
Old-Style Filter Functions	
Bessell Coefficients	***Bessell_Coef***
Butterworth Coefficients	***Bw_Coef***
Chebyshev Coefficients	***Ch_Coef***
Inverse Chebyshev Coefficients	***InvCh_Coef***
Elliptic Coefficients	***Elp_Coef***
IIR Filtering	***IIRFiltering***
FIR Digital Filters	
Lowpass Window Filters	***Wind_LPF***
Highpass Window Filters	***Wind_HPF***
Bandpass Window Filters	***Wind_BPF***

(continues)

Table 1-1. The Advanced Analysis Library Function Tree (Continued)

Bandstop Window Filters	***Wind_BSF***
Lowpass Kaiser Window	***Ksr_LPF***
Highpass Kaiser Window	***Ksr_HPF***
Bandpass Kaiser Window	***Ksr_BPF***
Bandstop Kaiser Window	***Ksr_BSF***
General Equi-Ripple FIR	***Equi_Ripple***
Lowpass Equi-Ripple FIR	***EquiRpl_LPF***
Highpass Equi-Ripple FIR	***EquiRpl_HPF***
Bandpass Equi-Ripple FIR	***EquiRpl_BPF***
Bandstop Equi-Ripple FIR	***EquiRpl_BSF***
FIR Coefficients	***FIR_Coef***
Windows	
Triangular Window	***TriWin***
Hanning Window	***HanWin***
Hamming Window	***HamWin***
Blackman Window	***BkmanWin***
Kaiser Window	***KsrWin***
Blackman-Harris Window	***BlkHarrisWin***
Tapered Cosine Window	***CosTaperedWin***
Exact Blackman Window	***ExBkmanWin***
Exponential Window	***ExpWin***
Flat Top Window	***FlatTopWin***
Force Window	***ForceWin***
General Cosine Window	***GenCosWin***
Measurement	
AC/DC Estimator	***ACDCEstimator***
Amplitude/Phase Spectrum	***AmpPhaseSpectrum***
Auto Power Spectrum	***AutoPowerSpectrum***
Cross Power Spectrum	***CrossPowerSpectrum***
Impulse Response	***ImpulseResponse***
Network Functions	***NetworkFunctions***
Power Frequency Estimate	***PowerFrequencyEstimate***
Scaled Window	***ScaledWindow***
Spectrum Unit Conversion	***SpectrumUnitConversion***
Transfer Function	***TransferFunction***
Statistics	
Basics	
Mean	***Mean***
Standard Deviation	***StdDev***
Variance	***Variance***
Root Mean Squared Value	***RMS***
Moments about the Mean	***Moment***

(continues)

Table 1-1. The Advanced Analysis Library Function Tree (Continued)

Median	***Median***
Mode	***Mode***
Histogram	***Histogram***
Probability Distributions	
Normal Distribution	***N_Dist***
T-Distribution	***T_Dist***
F-Distribution	***F_Dist***
Chi-Square Distribution	***XX_Dist***
Inv. Normal Distribution	***InvN_Dist***
Inv. T-Distribution	***InvT_Dist***
Inv. F-Distribution	***InvF_Dist***
Inv. Chi-SquareDist.	***InvXX_Dist***
Analysis of Variance	
One-way ANOVA	***ANOVA1Way***
Two-way ANOVA	***ANOVA2Way***
Three-way ANOVA	***ANOVA3Way***
Nonparametric Statistics	
Contingency Table	***Contingency_Table***
Curve Fitting	
Linear Fit	***LinFit***
Exponential Fit	***ExpFit***
Polynomial Fit	***PolyFit***
General Least Squares Fit	***GenLSFit***
Non-Linear Fit	***NonLinear Fit***
OldStyle Function	
Gen Least Squares Fit Coeff	***GenLSFitCoef***
Interpolation	
Polynomial Interpolation	***PolyInterp***
Rational Interpolation	***RatInterp***
Spline Interpolation	***SpInterp***
Spline Interpolant	***Spline***
Vector & Matrix Algebra	
Dot Product	***DotProduct***
Matrix Multiplication	***MatrixMul***
Matrix Inversion	***InvMatrix***
Transpose	***Transpose***
Determinant	***Determinant***
Trace	***Trace***
Solution of Linear Equations	***LinEqs***
LU Decomposition	***LU***
Forward Substitution	***ForwSub***
Backward Substitution	***BackSub***
Get Error String	***GetAnalysisErrorString***

The classes and subclasses in the function tree are described as follows.

- The **Signal Generation** function panels initialize arrays with predefined patterns.
- The **Array Operations** function panels perform arithmetic operations on 1D and 2D arrays.
  - **1D Operations**, a subclass of Array Operations, contains function panels that perform 1D array arithmetic.
  - **2D Operations**, a subclass of Array Operations, contains function panels that perform 2D array arithmetic.
- The **Complex Operations** function panels perform complex arithmetic operations. These function panels can operate on complex scalars or 1D arrays. The real and imaginary parts of complex numbers are processed separately.
  - **Complex Numbers**, a subclass of Complex Operations, contains function panels that perform scalar complex arithmetic.
  - **1D Complex Operations**, a subclass of Complex Operations, contains function panels that perform complex arithmetic on 1D complex arrays.
- The **Signal Processing** function panels perform data analysis in the frequency domain, time domain, or by using digital filters.
  - **Frequency Domain**, a subclass of Signal Processing, contains function panels that perform transformations between the time domain and the frequency domain, and perform analysis in the frequency domain.
  - **Time Domain**, a subclass of Signal Processing, contains function panels that perform direct time series analysis of signals.
  - **IIR Digital Filters**, a subclass of Signal Processing, contains function panels that perform infinite impulse response (IIR) digital filtering on signals by mapping analog specifications into digital specifications. This subclass contains Butterworth, Chebyshev, inverse Chebyshev, and elliptic filters.
  - **FIR Digital Filters**, a subclass of Signal Processing, contains function panels that perform the designs of finite impulse response (FIR) filters. These functions do not actually perform the digital filtering. This subclass contains window and equi-ripple FIR filters.
  - **Windows**, a subclass of Signal Processing, contains function panels that create windows that are frequently used to smooth data and reduce truncation effects in data acquisition applications.

- The **Measurement** function panels perform spectrum analysis using real units such as hertz and seconds.
- The **Statistics** function panels perform basic statistics functions.
  - **Basics**, a subclass of Statistics, contains function panels that use various common methods to describe a set of data.
  - **Probability Distributions**, a subclass of Statistics, contains function panels that operate as cumulative distribution functions from various probability distributions, and other function panels that operate as corresponding inverse functions.
  - **Analysis of Variance**, a subclass of Statistics, contains function panels that perform various analysis of variance in various statistical models.
  - **Nonparametric Statistics**, a subclass of Statistics, contains a function panel that analyzes data without assuming that the data is normally distributed.
- The **Curve Fitting** function panels perform curve fitting using least squares techniques. Linear, exponential, polynomial, and nonlinear fits are available.
- The **Interpolation** function panels take a set of points at which a function is known and guess the value the function takes at some specific intermediate point.
- The **Vector & Matrix Algebra** function panels perform vector and matrix operations. Vectors and matrices are represented by 1D and 2D arrays, respectively.

The online help with each panel contains specific information about operating each function panel.

Hints For Using Advanced Analysis Function Panels

With the analysis function panels, you can interactively manipulate scalars and arrays of data. You will often find it helpful to use the Advanced Analysis Library function panels in conjunction with the User Interface Library functions panels to view the results of analysis routines. When using the Advanced Analysis Library function panels, keep the following things in mind:

- The speed with which analysis functions are performed is greatly affected by the computer on which you are operating LabWindows/CVI. A numeric coprocessor can greatly decrease the execution time of floating-point computations. If you are using an Advanced Analysis Library function panel and nothing seems to happen for an inordinate amount of time, keep the constraints of your hardware in mind.

- LabWindows/CVI can perform many analysis routines for arrays in place; that is, input and output values are stored in the same array. This is important to remember when you are processing large amounts of data. Large double-precision arrays consume a lot of memory. If the results you want do not require that you keep the original array or intermediate arrays of data, perform analysis operations in place where possible.
- The Interactive window maintains a record of generated code. If you forget to keep the code from a function panel, you can cut and paste code between the Interactive and Program windows.

Reporting Analysis Errors

Each analysis function returns an integer error code. If the function is properly executed, the function returns a zero. Otherwise, an appropriate error value is returned.

The return value will correspond to one of the enumeration values of the type `AnalysisLibErrType` declared in the header file `analysis.h`. The analysis functions are declared in the header file with this return type so that the function panel controls for return values will display the symbolic name instead of the integer value of the error code. By declaring a variable with the type `AnalysisLibErrType`, the Variables window displays its value as a symbolic name instead of as an integer.

You can find a list of error codes in Appendix A, *Error Codes*.

About the Fast Fourier Transform (FFT)

The functions in the Frequency Domain subclass are based upon the discrete implementation and optimization of the Fourier Transform integral. The Discrete Fourier Transform (DFT) of a complex sequence X containing n elements is obtained using the following formula:

$$Y_i = \sum_{k=0}^{n-1} X_k * e^{-j2\pi ik/n}, \quad \text{for } i = 0, 1, \ldots, n-1$$

where Y_i is the ith element of the DFT of X and $j = \sqrt{-1}$

The DFT of X also results in a complex sequence Y of n elements. Similarly, the Inverse Discrete Fourier Transform (IDFT) of a complex sequence Y containing n elements is obtained using the following formula.

$$X_i = (1/n)\sum_{k=0}^{n-1} Y_k * e^{j2pik/n}, \quad \text{for } i = 0, 1, \ldots, n-1$$

where X_i is the ith element of the IDFT of Y and $j = \sqrt{-1}$

The discrete implementation of the DFT is a numerically intense process. However, it is possible to implement a fast algorithm when the size of the sequence is a power of two. These algorithms are known as FFTs, and can be found in many introductory texts to digital signal processing (DSP).

The current algorithm implemented in the LabWindows/CVI Advanced Analysis Library is known as the Split-Radix algorithm. This algorithm is highly efficient because it minimizes the number of multiplications, has the form of the Radix-4 algorithm, and the efficiency of the Radix-8 algorithm. The resulting complex FFT sequence has the conventional DSP format as described here.

If there are n number of elements in the complex sequence and $k = n/2$, then the output of the FFT is organized as follows:

Y_0	DC component
Y_1	Positive first harmonic
Y_2	Positive second harmonic
.	.
.	.
.	.
Y_{k-1}	Positive k-1 harmonic
Y_k	Nyquist frequency
Y_{k+1}	Negative k-1 harmonic
.	.
.	.
.	.
Y_{n-2}	Negative second harmonic
Y_{n-1}	Negative first harmonic

The following conventions and restrictions apply to the functions in the Frequency Domain section:

- All arrays must be a power of two: $n = 2^m$, $m = 1, 2, 3, \ldots, 12$.
- Complex sequences are manipulated using two arrays. One array represents the real elements. The other array represents the imaginary elements.

The following notation is used to describe the FFT operations performed in the Frequency Domain class:

- $Y = FFT\ \{X\}$, the sequence *Y* is the FFT of the sequence *X*.
- $Y = FFT^{-1}\ \{X\}$, the sequence *Y* is the inverse FFT of the sequence *X*.

X is usually a complex array but can be treated as a real array.

About Windowing

Almost every application requires you to use finite length signals. This requires that continuous signals be truncated, using a process called windowing.

The simplest window is a rectangular window. Because this window requires no special effort it is commonly referred to as the no window option. Remember, however, that a discrete signal and its spectrum is always affected by a window. Let x_n be a digitized time-domain waveform that has a finite length of *n*. w_n is a window sequence of n points. The windowed output is calculated as follows:

$$y_i = x_i * w_i \tag{1-1}$$

If X, Y, and W are the spectra of x, y, and w, respectively, the time-domain multiplication in equation (1-1) is equivalent to the convolution shown as follows:

$$Y_k = X_k \Theta W_k \tag{1-2}$$

Convolving with the window spectrum always distorts the original signal spectrum in some way. A window spectrum consists of a big main lobe and several side lobes.

The main lobe is the primary cause of lost frequency resolution. When two signal spectrum lines are too close to each other, they may fall in the width of the main lobe, causing the output of the windowed signal spectrum to have only one spectrum line. Use a window with a narrower main lobe to reduce the loss of frequency resolution. It has been shown that a rectangular window has the narrowest main lobe, so that it provides the best frequency resolution.

The side lobes of a window function affect frequency leakage. A signal spectrum line will leak into the adjacent spectrum if the side lobes are large. Once again, the leakage results from the convolution process. Select a window with relatively smaller side lobes to reduce spectral leakage. Unfortunately, a narrower main lobe and smaller side lobes are mutually exclusive. For this reason, selecting a window function is application dependent. An example of a windowed spectrum in the continuous case is shown in Figure 1-1.

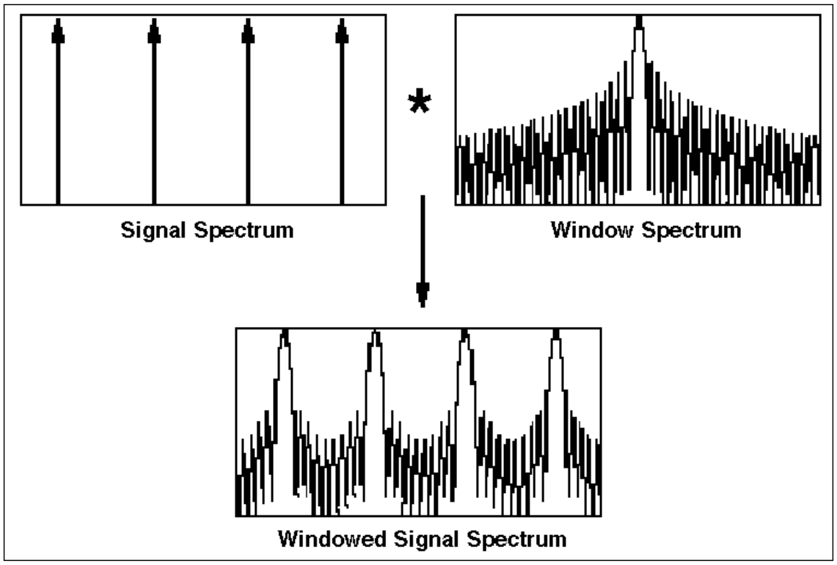

Figure 1-1. A Windowed Spectrum in the Continuous Case

The original signal spectrum is convolved with the window spectrum and the output is a smeared version of the original signal spectrum. In this example, you can still see four distinctive peaks from the original signal, but each peak is smeared and the frequency leakage effect is clear.

Window definitions used in National Instruments analysis libraries are designed in such a way that the window operations in the time domain are exactly equivalent to the operations of the same window in the frequency domain. To meet this requirement, the windows are not symmetrical in the time domain, that is:

$$w_0 \neq w_{N-1} \tag{1-3}$$

where N is the window length. They are usually symmetrical in the frequency domain, however. For example, the Hamming window definition uses the formula:

$$w_i = 0.54 - 0.46\cos(2\pi i/N) \tag{1-4}$$

Other manufacturers may use a slightly different definition, such as:

$$w_i = 0.54 - 0.46\cos(2\pi i/(N-1)) \tag{1-5}$$

The difference is small if N is large.

Equation (1-4) is not symmetrical in the time domain, but it ensures that the time domain windowing is equivalent to the frequency domain windowing. If you want to have a perfectly symmetrical sequence in the time domain, you must write your own windowing function using formula (1-5).

The choice of a window depends on the application. For most applications, the Hamming or Hanning windows deliver good performance.

About Digital Filters

There are two types of digital filters in the LabWindows/CVI Advanced Analysis Library: Finite Impulse Response (FIR) filters and Infinite Impulse Response (IIR) filters. FIR filters have a linear phase response. IIR filters generally have a nonlinear phase response, but offer much better amplitude response.

The choice of a particular type of filter depends upon the application. If you desire a linear phase response, choose one of the FIR filters. If performance and better amplitude response is more important, chose an IIR filter. No matter what type of filter you choose, enter a sampling frequency and other cutoff frequencies when designing your filter. You can design a digital filter using a normalized sampling frequency. The LabWindows/CVI Advanced Analysis Library provides a sampling frequency parameter so that you don't need to normalize other frequencies.

FIR Filters

The FIR filter is a set of filter coefficients that alters the signal spectrum when convolving with the signal. Let c_K be the filter coefficients, x_N be the input signal, and y_N be the output in the following formula:

$$y_i = \sum_{k=0}^{K-1} x_{i-k} * c_k , \qquad i = 0, 1, \ldots, N-1$$

LabWindows/CVI implements the formula using the convolution function `Convolve`. The purpose of an FIR filter is to design the coefficients c_K. Remember that no filtering is actually performed in an FIR filter function. You must subsequently call `Convolve` to perform the filtering. The advantage of doing this is that once you have obtained the filter coefficients, you can use them repeatedly without redesigning the filter.

If you have never used an FIR filter before, start with a window FIR filter. These filters are easy to design, though other techniques may design a better filter with the same number of coefficients.

Choose the window to be used in a window FIR filter with the parameter **WindType**. **WindType** determines the amount of attenuation the window filter can achieve. It also determines the transitional bandwidth of the window filter. The transitional bandwidth is defined as the frequency range from the specified cutoff frequency to the point where the desired attenuation is obtained. A bigger transitional bandwidth usually gives better attenuation. Use a Kaiser window FIR filter for choosing windows that are not available from **WindType**.

If you are experienced in using filters and you want to design an optimal FIR filter, use the LabWindows/CVI Advanced Analysis Library `Equi_Ripple` function. These filters are based on the general Parks-McClellan algorithm, that in turn is based on an alternation theorem in the polynomial approximation. As the name suggests, the frequency response of an `Equi_Ripple` filter has equal ripples within each specified frequency band. The ripples can be different in different bands depending on the weighting factors.

You have to specify more parameters when using `Equi_Ripple` filters. For each frequency band, specify the starting and ending points, the amplitude response and a weighting factor associated with the amplitude response of that band. A weighting factor of 1 is usually sufficient for all bands, but you can select different weighting factors. A bigger weighting factor results in a smaller ripple in the corresponding frequency band; a smaller weighting factor results in a larger ripple.

If you want to design an optimal FIR multiband filter, (lowpass, highpass, bandpass and bandstop), but do not want to specify the weighting factor, use `EquiRpl_LPF`, `EquiRpl_HPF`, `EquiRpl_BPF`, and `EquiRpl_BSF`. These filters call `Equi_Ripple` internally but have simplified input parameters.

Caution: ***The*** `Equi_Ripple` ***filter design does not always converge. In some cases, it will fail and give erroneous results. It is extremely important that you verify the filter design after obtaining the filter coefficients.***

IIR Filters

Mathematically, an IIR digital filter assumes the following form:

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{N_b-1} b_j x_{i-j} - \sum_{k=1}^{N_a-1} a_k y_{i-k}\right) \tag{1-6}$$

where a_k and b_k are the filter coefficients. The current filter output y_i depends upon the current and previous values x_{i-k} and previous output y_{i-k}. If $y_i \neq 0$, its effect on the subsequent points persists indefinitely. This is why these filters are called infinite impulse response filters.

Filters implemented using the structure defined by equation (1-6) directly are known as direct form IIR filters. Direct form implementations are often sensitive to errors introduced by coefficient quantization and by computational, precision limits. Additionally, a filter designed to be stable can become unstable with increasing coefficient length, which is proportional to filter order.

A less sensitive structure can be obtained by breaking up the direct form transfer function into lower order sections, or filter stages. The direct form transfer function of the filter given by equation (1-6) (with $a_0 = 1$) can be written as a ratio of z transforms, as follows:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_{N_b - 1} z^{-(N_b - 1)}}{1 + a_1 z^{-1} + \ldots + a_{N_a - 1} z^{-(N_a - 1)}} \quad (1\text{-}7)$$

By factoring equation (1-7) into second-order sections, the transfer function of the filter becomes a product of second-order filter functions

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_{1k} z^{-1} + a_{2k} z^{-2}}$$

where $N_s = \lfloor N_a/2 \rfloor$ is the largest integer $\leq N_a/2$, and $N_a \geq N_b$. This new filter structure can be described as a cascade of second-order filters.

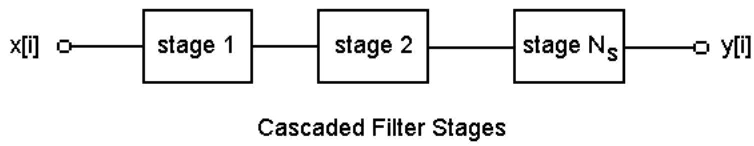

Figure 1-2. Cascaded Filter Stages

Each individual stage is implemented using the direct form II filter structure because it requires a minimum number of arithmetic operations and a minimum number of delay elements (internal filter states). Each stage has one input, one output, and two past internal states ($s_k[i-1]$ and $s_k[i-2]$).

If n is the number of samples in the input sequence, the filtering operation proceeds as in the following equations:

$y_0[i] = x[i]$,

$s_k[i] = y_{k-1}[i-1] - a_{1k}s_k[i-1] - a_{2k}s_k[i-2]$, $k = 1,2,...,N_s$

$y_k[i] = b_{0k}s_k[i] + b_{1k}s_k[i-1] + b_{2k}s_k[i-2]$, $k = 1,2,..., N_s$

$y[i] = y_{Ns}[i]$

for each sample $i = 0, 1, 2,...,n-1$.

For filters with a single cutoff frequency (lowpass and highpass), second-order filter stages can be designed directly. The overall IIR lowpass or highpass filter contains cascaded second-order filters.

For filters with two cutoff frequencies (bandpass and bandstop), fourth-order filter stages are a more natural form. The overall IIR bandpass or bandstop filter is cascaded fourth-order filters. The filtering operation for fourth-order stages proceeds as in the following equations:

$$y_0[i] = x[i],$$

$$s_k[i] = y_{k-1}[i-1] - a_{1k}\, s_k[i-1] - a_{2k}s_k[i-2] - a_{3k}s_k[i-3] - a_{4k}s_k[i-4]$$

$$k = 1, 2, \ldots, N_s$$

$$y_k[i] = b_{0k}s_k[i] + b_{1k}s_k[i-1] + b_{2k}s_k[i-2] + b_{3k}s_k[i-3] + b_{4k}s_k[i-4],$$

$$k = 1, 2, \ldots, N_s$$

$$y[i] = y_{Ns}[i].$$

Notice that in the case of fourth-order filter stages, $N_s = \lfloor (N_a+1)/4 \rfloor$.

The IIR filters provided in the LabWindows/CVI Advanced Analysis Library are derived from analog filters. There are four major types of IIR filters:

- Butterworth filters
- Chebyshev filters
- Inverse Chebyshev filters
- Elliptic filters

Lowpass, highpass, bandpass and bandstop filters are designed for each type of filter. The frequency response of a Butterworth filter is characterized by a smooth response at all frequencies and a monotonic decrease from the specified cut-off frequencies. Butterworth filters are maximally flat in the passband and zero in the stopband. The rolloff between the passband and stopband is slow, so that a lower order Butterworth filter does not provide a good approximation of an ideal filter.

Chebyshev filters have equal ripples in the passband and a monotonically decreasing magnitude response in the stopband. These filters have much sharper rolloffs than Butterworth filters. The inverse Chebyshev filters are similar to Chebyshev filters, except that the ripple occurs in the stopband and the frequency response is flat in the passband.
If ripples are allowable in both the passband and the stopband, use elliptic filters. Elliptic filters have the sharpest rolloffs for the same order compared with Butterworth or Chebyshev filters.

About Measurement Functions

Measurement functions perform DFT-based and FFT-based analysis with signal acquisition for frequency measurement applications as seen in typical frequency measurement instruments such as dynamic signal analyzers.

Several measurement functions perform commonly-used time domain to frequency domain transformations such as amplitude and phase spectrum, signal power spectrum, network transfer function, and so on. Other supportive measurement functions perform scaled time-domain windowing and power and frequency estimation, among other functions.

You can use the measurement functions for the following applications.

- Spectrum analysis applications
  - Amplitude and phase spectrum
  - Power spectrum
  - Scaled time domain window
  - Power and frequency estimate
- Network (frequency response) and dual channel analysis applications
  - Transfer function
  - Impulse response function
  - Network functions (including coherence)
  - Cross power spectrum

The DFT, FFT, and power spectrum are useful for measuring the frequency content of stationary or transient signals. The FFT provides the average frequency content of the signal over the entire time that the signal was acquired. For this reason, you use the FFT mostly for stationary signal analysis (when the signal is not significantly changing in frequency content over the time that the signal is acquired), or when you want only the average energy at each frequency line. A large class of measurement problems falls in this category. For measuring frequency information that changes during the acquisition, you should use joint time-frequency analysis.

The measurement functions are built on top of the signal processing functions and have the following characteristics that model the behavior of traditional benchtop frequency analysis instruments.

- Assumed Real-world time-domain signal input.
- Outputs in magnitude and phase, scaled in units where appropriate, ready for immediate graphing.

- Single-sided spectrums from DC to $\frac{\text{Sampling Frequency}}{2}$.
- Sampling period to frequency interval conversion for graphing with appropriate X-axis units (in Hertz).
- Corrections for the windows being used applied where appropriate.
- Scaled Windows; Each window gives same peak spectrum amplitude result within its amplitude accuracy constraints.
- Viewing of power or amplitude spectrum in various unit formats including decibels and spectral density units $(V^2/Hz, V/\sqrt{Hz})$ and so on.

About Curve Fitting

The algorithm used to find the best curve fit in the Curve Fitting class is the Least Squares method. The purpose of the algorithm is to find the curve coefficients a, which minimize the squared error e(a) in the following formula:

$$e(a) = \sum_i |Y_i - f(X_i, a)|^2$$

where $f(X_i, a)$ is the function representing the desired curve.

You can find the coefficient a by solving the linear system of equations generated by the following formula:

$$\frac{\partial}{\partial a} e(a) = 0$$

Given a set of n sample points (x, y) represented by the sequences X and Y, the curve-fitting functions determine the coefficients that best represent the data. The best fit Z is an array of expected values given the coefficients and the X set of values. Thus you can express Z as a function of X and the following coefficients:

$$Z = f(X, a)$$

When you have established the best fit values, you can obtain the mean squared error (mse) by applying the following formula.

$$mse = \sum_{i=0}^{n-1} (Z_i - Y_i)^2 / n$$

Chapter 2
Advanced Analysis Library Function Reference

This chapter contains a brief explanation of each of the functions in the LabWindows/CVI Advanced Analysis Library. The LabWindows/CVI Advanced Analysis Library functions are arranged alphabetically.

Abs1D

`int` **status** = **Abs1D** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`);

Purpose

Finds the absolute value of the **x** input array. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements
Output	**y**	double-precision array	absolute value of input array

Return Value

status	integer	refer to error codes in Appendix A

ACDCEstimator

`int` **status** = **ACDCEstimator** (`double` **x**`[]`, `int` **n**, `double` ***acEstimate**,
`double` ***dcEstimate**);

Purpose

Computes an estimation of the AC and DC contents of the input signal. **x** is the input signal, usually in volts.

acEstimate (Vrms) is the estimate of the input signal AC content in volts rms, if the input signal is in volts.

dcEstimate (V) is the estimate of the input signal DC content in volts, if the input signal is in volts.

Parameters

Input	**x**	double-precision array	contains the time-domain signal, usually in volts. At least three cycles of the signal must be contained in this array for a valid estimate.
	n	integer	number of elements in the input array.
Output	**acEstimate**	double-precision	contains the estimate of the AC level of the input signal in volts rms if the input signal is volts.
	dcEstimate	double-precision	contains the estimate of the DC level of the input signal in the same units as the input signal.

Return Value

status	integer	refer to error codes in Appendix A

Add1D

`int` **status** = **Add1D** (`double` **x**`[]`, `double` **y**`[]`**,** `int` **n**, `double` **z**`[]`);

Purpose

Adds one-dimensional (1D) arrays. The i^{th} element of the output array is obtained using the following formula.

$$z_i = x_i + y_i$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x**	double-precision array	input array
	y	double-precision array	input array
	n	integer	number of elements to be added
Output	**z**	double-precision array	result array

Return Value

status	integer	refer to error codes in Appendix A

Add2D

`int` **status** = **Add2D** (`void` *__x__, `void` *__y__, `int` **n**, `int` **m**, `void` *__z__);

Purpose

Adds two-dimensional (2D) arrays. The (i^{th}, j^{th}) element of the output array is obtained using the following formula.

$$z_{i,j} = x_{i,j} + y_{i,j}$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x**	double-precision 2D array	input array
	y	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
Output	**z**	double-precision 2D array	result array

Return Value

status	integer	refer to error codes in Appendix A

AllocIIRFilterPtr

`IIRFilterPtr` **filterInformation** = **AllocIIRFilterPtr** (`int` **type**, `int` **order**);

Purpose

Allocates and initializes the **filterInformation** structure, returning a pointer to the filter structure for use with the IIR cascade filter coefficient design calls.

You input the type of the filter (lowpass, highpass, bandpass, or bandstop) and the order. This function will allocate the filter structure as well as the internal coefficient arrays and internal filter state array.

Parameters

Input	**type**	integer	Controls the filter type of IIR filter coefficients.
			`LOWPASS  = 0` (default)
			`HIGHPASS = 1`
			`BANDPASS = 2`
			`BANDSTOP = 3`

	order	integer	Specifies the order of the IIR filter. Default Value: 3

Return Value

filterInformation	`IIRFilterPtr`	Pointer to the filter structure. When an error occurs, **filterInformation** is zero.

Parameter Discussion

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. Call this function to allocate **filterInformation** before calling one of the cascade IIR filter design functions.

The definition of the filter structure is as follows:

```
typedef struct  {
    intnum  type;   /* type of filter (lp,hp,bp,bs) */
    intnum  order;  /* order of filter */
    intnum  reset;  /* 0 - don't reset, 1 - reset */
    intnum  na;     /* number of a coefficients */
    floatnum *a;    /* pointer to a coefficients */
    intnum nb;      /* number of b coefficients */
    floatnum *b;    /* pointer to b coefficients */
    intnum ns;      /* number of internal states */
    floatnum *s;    /* pointer to internal state array */
    } *IIRFilterPtr;
```

AmpPhaseSpectrum

`int` **status = AmpPhaseSpectrum** (`double` **x**`[]`, `int` **n**, `int` **unwrap**, `double` **dt**, `double` **ampSpectrum**`[]`, `double` **phaseSpectrum**`[]`, `double` ***df**);

Purpose

Computes the single-sided, scaled amplitude and phase spectra of a time-domain signal, X. The amplitude spectrum is computed as

$$\left| FFT(X) / n \right|$$

and is converted to single-sided form. The phase spectrum is computed as

phase[FFT(X)]

and is also converted to single-sided form.

Parameters

Input	**x**	double-precision array	Contains the time-domain signal.
	n	integer	The number of elements in the input array. Valid Values: Powers of 2.
	unwrap	integer	Controls the unwrapping of the phase spectrum. Valid values for unwrap: 1: enable phase unwrapping 0: disable phase unwrapping ($-\pi \leq$ phase $\leq +\pi$).
	dt	double-precision	The sample period of the time-domain signal, usually in seconds. **dt** = $1/_{fs}$, where fs is the sampling frequency of the time-domain signal.

(continues)

Parameters (Continued)

Output	**ampSpectrum**	double-precision array	**ampSpectrum** is the single-sided amplitude spectrum magnitude in volts RMS if the input signal is in volts. If the input signal is not in volts, the results are in input signal units RMS. This array must be at least **n**/2 elements long.
	phaseSpectrum	double-precision array	**PhaseSpectrum** is the single-sided phase spectrum in radians. This array must be at least **n**/2 elements long.
	df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. *__df__ = 1/(**n***dt**)

Return Value

status	integer	refer to error codes in Appendix A

ANOVA1Way

`int` **status** = **ANOVA1Way** (`double` **y**`[]`, `int` **level**`[]`, `int` **n**, `int` **k**, `double` ***ssa**, `double` ***msa**, `double` ***f**, `double` ***sig**, `double` ***sse**, `double` ***mse**, `double` ***tss**);

Purpose

Takes an array of experimental observations made at various levels of some factor (with at least one observation per factor) and performs a one-way analysis of variance in the fixed effect model.

The one-way analysis of variance is a test to determine whether the level of the factor has an effect on the experimental outcome.

Parameters

Input	**y**	double-precision array	experimental observations
	level	integer array	the i^{th} element tells in what level of the factor the i^{th} observation falls
	n	integer	the total number of observations
	k	integer	the total number of levels of the factor
Output	**ssa**	double-precision	sum of squares due to the factor
	msa	double-precision	mean square due to the factor
	f	double-precision	calculated F-value
	sig	double-precision	the level of significance at which the null hypothesis must be rejected
	sse	double-precision	sum of squares due to random fluctuation
	mse	double-precision	mean square due to random fluctuation
	tss	double-precision	total sum of squares

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

<u>Factors and Levels</u>

A factor is a way of categorizing data. Data is categorized into levels, beginning with level 0. For example, if you are performing some measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For age, one might have three levels, as given below.

Level	Ages
0:	6 years to 10 years
1:	11 years to 15 years
2:	16 years to 20 years

<u>The Statistical Model</u>

Each experimental outcome is expressed as the sum of three parts while performing the analysis of variance. Let $y_{i,m}$ be the m^{th} observation from the i^{th} level. Each observation is written:

$$y_{i,m} = \mu + \alpha_i + \varepsilon_{i,m}$$

where
- μ is a standard effect
- α_i is the effect of the i^{th} level of the factor
- $\varepsilon_{i,m}$ is a random fluctuation

<u>Assumptions</u>

Assume that the populations of measurements at each level are normally distributed with mean α_i and variance σ_A^2. Assume that the means α_i sum to zero. Finally, assume that for each i and m, $\varepsilon_{i,m}$ is normally distributed with mean 0 and variance σ_A^2.

<u>The Hypothesis</u>

Test the (null) hypothesis that $\alpha_i = 0$ for i = 0, 1, ..., k-1 (where **k** is the total number of levels). In other words, assume from the start that the levels have no effect on the experimental outcome, then look for evidence to the contrary.

<u>The General Method</u>

Break up the total sum of squares **tss**, a measure of the total variation of the data from the overall population mean, into component sums of squares, which may be attributed to different sources.

You now have

$$tss = ssa + sse$$

where **ssa** is a measure of variation that is attributed to the factor, and where **sse** is a measure of variation that is attributed to random fluctuation. Divide by appropriate numbers to obtain the averages **msa** and **mse**. If there is much variation caused by the factor, **msa** will be larger relative to **mse**. The ratio **f** will also be larger relative to **mse**.

If the null hypothesis is true, the ratio **f** is taken from an F distribution with k-1 and n-k degrees of freedom, from which you can calculate probabilities. Given a particular **f**, **sig** is the probability that in sampling from this distribution you get a value larger than **f**.

<u>Testing the Hypothesis</u>

This function generates a number **f** so that, if the hypothesis is true, that number is from an F-distribution with k-1 and n-k degrees of freedom. The function also calculates the probability that a number taken from this F-distribution is larger than **f**. This is the output parameter, **sig**:

$$sig = Prob(x>f)$$

where x is from F(k-1, n-k).

Use the probability **sig** to determine when to reject the hypothesis. To do so, choose a level of significance for the hypothesis. The level of significance is how likely you want it to be that you reject the hypothesis when it is true, and so the level of significance should be small (0.05 is a common choice). Keep in mind that the smaller the level of significance, the more hesitant you are to reject the hypothesis.

The hypothesis is rejected when the output parameter **sig** is less than the chosen level of significance.

<u>Formulas</u>

Let $y_{i,m}$ be the m^{th} observation made at the i^{th} level for m = 0,1, ...,n_i and i = 0,1, ...k.

Let n_i = the number of observations at the i^{th} level.

$$Y_i = \frac{1}{n_i}\sum_{m=0}^{n_i-1} y_{i,m}$$

$$Y_m = \frac{1}{k}\sum_{i=0}^{k-1} y_{i,m}$$

$$Y = \frac{1}{n}\sum_{i=0}^{k-1}\sum_{m=0}^{n_i-1} y_{i,m}$$

$$T = n * Y$$

Then

$$ssa = \sum_{i=0}^{k}\left(\frac{Y_i^2}{n_i}\right) - \frac{Y^2}{n}$$

$$mse = ssa/(k-1)$$

$$sse = \sum_{i=0}^{k-1} \sum_{m=0}^{n_i-1} y^2{}_{i,m} - \sum_{i=0}^{k} \left(\frac{Y_i^2}{n_i} \right)$$

$$mse = sse/(n-k)$$

$$tss = \sum_{i=0}^{k-1} \sum_{m=0}^{n_i-1} {y_{i,m}}^2 - \frac{Y^2}{n}$$

$f = msa/mse$ where **f** is from an F-distribution with (k-1) and (n-k) degrees of freedom.

<u>Example</u>

Suppose that researchers want to know whether the amount of rainfall affects the yield of a crop. The factor (rainfall) is divided into three levels (k=3) as given below.

Level	Rainfall (factor)
0	2 inches
1	3 inches
2	4 inches

The researchers set up 10 plots in various geographical locations chosen so that each plot receives a different amount of rainfall. They record the following information.

Level	Bushels produced from each plot
0	128 122 126 124
1	140 141 143
2	120 118 123

To perform a one-way analysis using the `ANOVA1Way` function, all the numbers of bushels are stored in a double-precision array **y** of size 10. The integer array **level** records the levels in which observations were made. For any particular i, these arrays are set such that y_i is the number of bushels produced by a plot in $level_i$. For example,

$level_i = 0$

y_i = 128, 122, 126, or 124

are valid combinations. Therefore, the input arrays **y** and **level** in this example could be set up for the `ANOVA1Way` function as follows.

y = 128, 122, 126, 124, 140, 141, 143, 120, 118, 123
level = 0, 0, 0, 0, 1, 1, 1, 2, 2, 2

Running the code given in the examples below produces the following result.

sig = 0.0000239

For a level of significance such as 0.05, the `ANOVA1Way` results show that the researchers must reject the hypothesis that the rainfall has no effect on the yield of the crop. In other words, the rainfall does affect the crop yield.

Example

```
double y[10], ssa, msa, f, sig, sse, mse, tss;
int level[10];
int k;
int status;

k = 3;         /* three levels for rainfall */

/* Read in recorded data y(10), level[10] */

status = ANOVA1Way(y, level, 10, k, &ssa, &msa, &f, &sig, &sse, &mse, &tss);
```

ANOVA2Way

`int` **status** = **ANOVA2Way** (`double` **y**`[]`, `int` **levelA**`[]`, `int` **levelB**`[]`, `int` **N**, `int` **L**, `int` **a**, `int` **b**, `void` ***info**, `double` ***sigA**, `double` ***sigB**, `double` ***sigAB**);

Purpose

Takes an array of experimental observations made at various levels of two factors and performs a two-way analysis of variance in any of the following models.

- Fixed effects with no interaction and one observation per cell (**L**=1 per specified levels **a** and **b** of the factors A and B respectively)
- Fixed effects with interaction and **L**>1 observations per cell
- Either of the mixed-effects models (where one factor is taken to have a fixed effect but the other is taken to have a random effect) with interaction and **L**>1 observations per cell
- Random effects with interaction and **L**>1 observations per cell

Any ANOVA looks for evidence that the factors (or interactions among the factors) have a significant effect on experimental outcomes. What varies among models is the method for finding significance.

Parameters

Input	**y**	double-precision array	array of experimental data of N = |a| *|b| *L elements
	levelA	integer array	the i^{th} element tells in what level of factor A the i^{th} observation falls
	levelB	integer array	the i^{th} element tells in what level of factor B the i^{th} observation falls
	N	integer	the total number of observations
	L	integer	the number of observations per cell
	a	integer	the number of levels in factor A. This parameter is negative if A is a random effect
	b	integer	the number of levels in factor B. This parameter is negative if B is a random effect
Output	**info**	double-precision 2D array	a 4 by 4 matrix as follows: ssa dofa msa fa ssb dofb msb fb ssab dofab msab fab sse dofe mse 0.0 where ss designates sums of squares, dof designates degrees of freedom of ss, ms designates mean squares, and f designates F-distributions (depending on the statistical model)
	sigA	double-precision	level of significance at which hypothesis (A) must be rejected
	sigB	double-precision	level of significance at which hypothesis (B) must be rejected
	sigAB	double-precision	level of significance at which hypothesis (AB) must be rejected

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

Factors, Levels and Cells

A factor is a way of categorizing data. Data is categorized into levels, beginning with level 0. For example, if you are performing some measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For factor age, one might have three levels, given below.

0: 6 years to 10 years
1: 11 years to 15 years
2: 16 years to 20 years

Another possible factor is eye color, with the following levels.

0: blue
1: brown
2: green
3: hazel

In this example, an analysis of variance seeks evidence that the ages and eye color of the subjects have an effect on the number of sit-ups performed.

A cell of data consists of all those experimental observations that fall in particular levels of the two factors. In this instance, a cell might consist of those observations made on hazel-eyed individuals between 11 and 15 years old. The number of observations that fall in each cell must be some constant number **L** that does not vary between cells.

Random and Fixed Effects

A factor is taken as a random effect when the factor has a large population of levels about which you want to draw conclusions, but that cannot be sampled at all levels. Levels are sampled at random in the hope of generalizing about all levels.

A factor is taken as a fixed effect when the factor can be sampled from all levels about which you want to draw conclusions.

The input parameters **a** and **b** represent the number of levels in factors A and B, respectively. If factor A is to be random, set **a** to a negative value. If factor B is to be random, set **b** to a negative value. Notice that if there is only one observation per cell, both **a** and **b** must be positive (that is, model 1 is used).

The General Method

Each of the models breaks up the total sum of squares (tss, a measure of the total variation of the data from the overall population mean) into some number of component sums of squares. In model 1,

$$tss = ssa + ssb + sse$$

whereas in models 2 through 4

$$tss = ssa + ssb + ssab + sse.$$

Each component of the sums is a measure of variation attributed to a certain factor or interaction among the factors. The component ssa is a measure of the variation due to factor A, ssb is a measure of the variation due to factor B, ssab is a measure of the variation due to the interaction between factors A and B, and sse is a measure of the variation due to random fluctuation. Notice that with model 1 there is no ssab term. This is what is meant by "no interaction".

If factor A has a strong effect on the experimental observations, msa will be relatively large. Specific ratios of these averages are considered because you know how they are statistically distributed. You can therefore determine how likely it is that factor A is as relatively large as it is.

The Statistical Model

Let $y_{p,q,r}$ be the r^{th} observation at the p^{th} and q^{th} levels of A and B, respectively,

where
$r = 0, 1, ..., L-1.$

Model 1. Express each observation as the sum of four components, so that

$$y_{p,q,r} = \mu + \alpha_p + \beta_q + \varepsilon_{p,q,r}$$

where μ represents a standard effect present in each observation, α_p represents the effect of the p^{th} level of factor A, β_q represents the effect of the q^{th} level of factor B, and $\varepsilon_{p,q,r}$ is a random fluctuation.

Models 2, 3, and 4. Express each observation as the sum of five components, so that

$$y_{p,q,r} = \mu + \alpha_p + \beta_q + (\alpha\beta)_{p,q} + \varepsilon_{p,q,r}$$

where μ represents a standard effect present in each observation, α_p represents the effect of the p^{th} level of factor A, β_q represents the effect of the q^{th} level of factor B, and $\varepsilon_{p,q,r}$ is a random fluctuation. In addition, $(\alpha\beta)_{p,q}$ represents the effect of the interaction between the p^{th} level of factor A and the q^{th} level of factor B.

Assumptions

- Assume that for each p, q and r, $\varepsilon_{p,q,r}$ is normally distributed with mean 0 and variance σ_e^2.
- If a factor such as A is fixed, assume that the populations of measurements at each level are normally distributed with mean α_p and variance σ_A^2. Notice that all the populations at each of the levels are taken to have the same variance. In addition, it is assumed that all the α_p means sum to zero. An analogous assumption is made for B.
- If a factor such as A is random, assume that the effect of the level of A itself, α_p, is a random variable normally distributed with mean 0 and variance σ_A^2. An analogous assumption is made for B.
- If all of the factors, such as A and B, associated with the effect of an interaction $(\alpha\beta)_{p,q}$ are fixed, assume that the populations of measurements at each level are normally distributed with mean $(\alpha\beta)_{p,q}$ and variance σ_{AB}^2. For any fixed p, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all q. Similarly, for any fixed q the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all p.
- If any of the factors, such as A and B, associated with the effect of an interaction $(\alpha\beta)_{p,q}$ are random, then the effect is taken to be a random variable normally distributed with mean 0 and variance σ_{AB}^2. If A is fixed but B is random, assume additionally that for any fixed q, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all p. Similarly, if B is fixed but A is random, assume additionally that for any fixed p, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all q.
- All effects taken to random variables are assumed to be independent.

The Hypotheses

Each of the following hypotheses are different ways of saying that a factor or an interaction among factors has no effect on experimental outcomes. Start by assuming that there are no effects and then seek evidence to contradict these assumptions. The three hypotheses are as follows.

- For (A), α_p= 0 for all levels of p if factor A is fixed; $\sigma_A^2 = 0$ if factor A is random
- For (B), $\beta_q = 0$ for all levels of q if factor B is fixed; $\sigma_B^2 = 0$ if factor B is random
- For (AB), $(\alpha\beta)_{p,q} = 0$ for all levels of p and q if both factors A and B are fixed; $\sigma_{AB}^2 = 0$ if either factor A or factor B is random. (This does not apply to model 1. In model 1, there is no interaction and the associated output parameters are superfluous.)

Testing the Hypotheses

For each hypothesis, the function generates a number so that, if the hypothesis is true, that number will be from a particular F-distribution.

For example, in model 1, fa = msa/mse (associated with hypothesis (A)) is from an F-distribution with a-1 and (a-1)(b-1) degrees of freedom (F(a-1, (a-1)(b-1))), given that hypothesis (A) is true. In models 2, 3, and 4, fa=msa/mse (associated with hypothesis (A)) is from an F-distribution with a-1 and ab(L-1) degrees of freedom (F(a-1, ab(L-1))), given that hypothesis (A) is true. The function calculates the probability that a number taken from a particular F-distribution is larger than the F-value. For example,

sigA = Prob(X > fa)

where X is from F(a-1, (a-1)(b-1)).

Use the probabilities **sigA**, **sigB**, and **sigAB** to determine when to reject the associated hypotheses (A), (B), and (AB). To make this determination, choose a level of significance for each hypothesis. The level of significance is how likely you want it to be that you reject the hypothesis when it is in fact true. Ordinarily, you do not want it to be very likely that you reject the hypothesis when it is true, and so the level of significance should be small (0.05 is a common choice). Keep in mind that the smaller the level of significance, the more hesitant you are to reject the hypothesis.

A particular hypothesis is rejected when the associated output parameter **sigA**, **sigB**, or **sigAB** is less than the level of significance chosen for that hypothesis. If A is a random effect, and the chosen level of significance is 0.05, and **sigA** = 0.03, you must reject the hypothesis that $\sigma_A^2 = 0$, and conclude that factor A does have an effect on the experimental observations.

Formulas

Let $y_{p,q,r}$ be the r^{th} observation at the p^{th} and q^{th} levels of A and B, respectively, where
r = 0, 1, ..., L-1.

Let

$$aa = |a|$$

$$bb = |b|$$

$$T_{p,q} = \sum_{r=0}^{L-1} y_{p,q,r}$$

$$T_p = \sum_{q=0}^{bb-1} T_{p,q}$$

$$T_q = \sum_{p=0}^{aa-1} T_{p,q}$$

T = the total sum of all observations

$$A = \sum_{p=0}^{aa-1} T_p^{\,2} \,/\, (bb * L)$$

$$B = \sum_{q=0}^{bb-1} T_q^{\,2} \,/ (aa * L)$$

$$S = \sum_{p=0}^{aa-1} \sum_{q=0}^{bb-1} T_{p,q^2} / L$$

$$CF = T^2/(aa*bb*L)$$

Then

ssa = A - CF	*msa = ssa/(aa-1) = ssa/dofa*
ssb = B - CF	*msb = ssb/(bb-1) = ssb/dofb*
ssab = S - A - B - CF	*msab = ssab/(a-1)(b-1) = ssab/dofab*
sse = T - S	*mse = sse/(aa*bb*(L-1)) = sse/dofe*

*fa = msa/mse (*if B is fixed*)*
*= msa/msab (*if B is random*)*

*fb = msb/mse (*if A is fixed*)*
*= msb/msab (*if A is random*)*

fab = msab/mse

If f = ms_1/ms_2 and $ms_1 = ss_1/dof_1$ and $ms_2 = ss_2/dof_2$, we assume that f is from an F-distribution with dof_1 and dof_2 degrees of freedom (F(dof_1, dof_2)).

Example

Suppose that researchers want to know how the amount of rainfall and the average temperature affect the yield of a crop. Each factor (rainfall, and temperature) is divided into three levels as follows.

Level	Rainfall (Factor A)
0	2 inches
1	3 inches
2	4 inches

Level	Temperature (Factor B)
0	76-80 degrees
1	81-85 degrees
2	86-90 degrees

A particular plot planted with the crop may be in any one of the 9 different combinations of these levels with the two factors. For example, one combination might be two inches of rain and an average temperature between 76 and 80 degrees, recorded as (0,0). These combinations are called as cells.

The researchers set up 18 plots in various geographical locations chosen so that two plots will fall in each of the 9 cells. To measure the productivity of a particular plot, they record the crop production. Let Rainfall be Factor A and Temperature be Factor B. They record the following information.

(A, B)	**Bushels produced from each plot**
(0, 0)	128 122
(0, 1)	113 108
(0, 2)	116 116
(1, 0)	132 129
(1, 1)	119 121
(1, 2)	126 113
(2, 0)	118 114
(2, 1)	141 133
(2, 2)	121 123

To perform a two-way analysis of variance in the fixed effect model using the `ANOVA2Way` function, all the numbers of bushels are stored in a double-precision array **y** of size 18. The integer arrays **levelA** and **levelB** record the cells in which observations were made. For any particular i, these arrays are set such that y_i is the number of bushels produced by a plot in the ($levelA_i$, $levelB_i$ cell. For example,

$(levelA_i, levelB_i) = (0, 1)$

$y_i = 113$, *or* 108

are valid combinations.

Therefore, the input arrays **y**, **levelA**, and **levelB** in this example could be set up for the `ANOVA2Way` function as follows.

y = 128, 122, 113, 108, 116, 132, 129, 119, 121, 126, 113, 118, 114, 141, 133, 121, 123

levelA = 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2

levelB = 0, 0, 1, 1, 2, 2, 0, 0, 1, 1, 2, 2, 0, 0, 1, 1, 2, 2

Running the code given in the examples below produces the following results.

sigA = 0.026
sigB = 0.203
sigAB = 0.0018

For a level of significance such as 0.05, the `ANOVA2Way` results show that the researchers cannot reject the hypotheses that the combination of rainfall and temperature has any effect on the yield of the crop. In other words, the combination of rainfall and temperature has a significant effect on crop yield.

<u>Example</u>

```
double y[18], sigA, sigB, sigAB, info[4][4];
int levelA[18], levelB[18];
int L, a, b;
int status;

L = 2;                /* two observations per cell */
a = 3;                /* three levels for factor A, Rainfall */
b = 3;                /* three levels for factor B, Temperature */

/* Read in recorded data y[18], levelA[18], levelB[18] */

status = ANOVA2Way(y, levelA, levelB, 18, L, a, b, info, &sigA,
          &sigB, &sigAB);
```

ANOVA3Way

`int` **status =ANOVA3Way** (`double` **y**`[]`, `int` **levelA**`[]`, `int` **levelB**`[]`, `int` **levelC**`[]`, `int` **N**, `int` **L**, `int` **a**, `int` **b**, `int` **c**, `void` ***info**, `double` ***sigA**, `double` ***sigB**, `double` ***sigC**, `double` ***sigAB**, `double` ***sigAC**, `double` ***sigBC**, `double` ***sigABC**);

Purpose

Takes an array of experimental observations made at various levels of three factors and performs a three-way analysis of variance in any of the following models.

- Fixed effects with interaction and **L**>1 observations per cell
- Any of the six mixed-effects models (where one or two factors are taken to have fixed effects but the remaining factors are taken to have random effects) with interaction and **L**>1 observations per cell
- Random effects with interaction and **L**>1 observations per cell

Any ANOVA looks for evidence that the factors (or interactions among the factors) have a significant effect on experimental outcomes. What varies among models is the method for finding significance.

Parameters

Input	**y**	double-precision array	array of experimental data of N = |a| *|b| *|c| *L elements
	levelA	integer array	the i^{th} element tells in what level of factor A the i^{th} observation falls
	levelB	integer array	the i^{th} element tells in what level of factor B the i^{th} observation falls
	levelC	integer array	the i^{th} element tells in what level of factor C the i^{th} observation falls
	N	integer	the total number of observations
	L	integer	the number of observations per cell
	a	integer	the number of levels in factor A. This parameter is negative if A is a random effect
	b	integer	the number of levels in factor B. This parameter is negative if B is a random effect
	c	integer	the number of levels in factor C. This parameter is negative if C is a random effect

(continues)

Parameters (Continued)

Output	**info**	double-precision 2D array	an 8 by 4 matrix as follows: ssa dofa msa fa ssb dofb msb fb ssc dofc msc fc ssab dofab msab fab ssac dofac msac fac ssbc dofbc msbc fbc ssabc dofabc msabc fabc sse dofe mse 0.0 where ss designates sums of squares, dof designates degrees of freedom of ss, ms designates mean squares, and f designates F-distributions (depending on the statistical model)
	sigA	double-precision	level of significance at which hypothesis (A) must be rejected
	sigB	double-precision	level of significance at which hypothesis (B) must be rejected
	sigC	double-precision	level of significance at which hypothesis (C) must be rejected
	sigAB	double-precision	level of significance at which hypothesis (AB) must be rejected
	sigAC	double-precision	level of significance at which hypothesis (AC) must be rejected
	sigBC	double-precision	level of significance at which hypothesis (BC) must be rejected
	sigABC	double-precision	level of significance at which hypothesis (ABC) must be rejected

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

<u>Factors, Levels, and Cells</u>

A factor is a way of categorizing data. Data is categorized into levels, beginning with level 0. For example, if you are performing some measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For age, one might have three levels, as given below.

Levels	Ages
0	6 years to 10 years
1	11 years to 15 years
2	16 years to 20 years

Another possible factor is eye color, with the following levels.

Levels	Eye Colors
0	blue
1	brown
2	green
3	hazel

A third factor might be height with levels in blocks of 10 centimeters. A cell of data consists of all those experimental observations that fall in particular levels of the three factors. In this instance, a cell might consist of those observations made on hazel-eyed individuals between 11 and 15 years old who are between 151 and 160 cm tall. The number of observations that fall in each cell must be some constant number **L** that does not vary between cells.

Random and Fixed Effects

A factor is taken as a random effect when the factor has a large population of levels about which you want to draw conclusions, but that cannot be sampled at all levels. Levels are sampled at random in the hope of generalizing about all levels.

A factor is taken as a fixed effect when the factor can be sampled from all levels about which you want to draw conclusions.

The input parameters **a**, **b**, and **c** represent the number of levels in factors A, B, and C, respectively. If factor A is to be random, set **a** to a negative value. In the same way, set **b** and **c** to negative values if B and C are to be random.

The General Method

Each of the models breaks up the total sum of squares (tss, a measure of the total variation of the data from the overall population mean) into some number of component sums of squares, so that

$$tss = ssa + ssb + ssc + ssab + ssac + ssbc + ssabc + sse$$

Each component in the sum is a measure of variation attributed to a certain factor or interaction among the factors. In this instance, ssa is a measure of the variation due to factor A, ssb is a measure of the variation due to factor B, ssc is a measure of the variation due to factor C, ssab is a measure of the variation due to the interaction between factors A and B, and so on for ssac, ssbc, and ssabc. The variable sse is a measure of the variation due to random fluctuation.

If factor A has a strong effect on the experimental observations, msa will be relatively large. You can look at specific ratios of these averages because you know how they are statistically distributed. You can therefore determine how likely it is that factor A is as relatively large as it is.

The Statistical Model

Let $y_{p,q,r,s}$ be the sth observation at the pth, qth, and rth levels of A, B, and C, respectively, where s = 0, 1, ..., L-1. Express each observation as the sum of eight components, so that

$$y_{p,q,r,s} = \mu + \alpha_p + \beta_q + \gamma_r + (\alpha\beta)_{p,q} + (\alpha\gamma)_{p,r} + (\beta\gamma)_{q,r} + (\alpha\beta\gamma)_{p,q,r} + \varepsilon_{p,q,r,s}$$

where μ represents a standard effect present in each observation; α_p, β_q, and γ_r are the effects of factors A, B, and C respectively; $(\alpha\beta)_{p,q}$, $(\alpha\gamma)_{p,r}$, $(\beta\gamma)_{q,r}$, and $(\alpha\beta\gamma)_{p,q,r}$ are the effects of the corresponding interactions; and $\varepsilon_{p,q,r,s}$ is a random fluctuation.

Assumptions

- Assume that for each p, q r, and s, $\varepsilon_{p,q,r,s}$ is normally distributed with mean 0 and variance σ_e^2.
- If a factor such as A is fixed, assume that the populations of measurements at each level are normally distributed with mean α_p and variance σ_A^2. Notice that all the populations at each of the levels are taken to have the same variance. In addition, it is assumed that all the α_p means sum to zero. Analogous assumptions are made for B and C.
- If a factor such as A is random, assume that the effect of the level of A itself, α_p, is a random variable normally distributed with mean 0 and variance σ_A^2. Analogous assumptions are made for B and C.
- If all of the factors, such as A and B, associated with the effect of an interaction $(\alpha\beta)_{p,q}$ are fixed, assume that the populations of measurements at each level are normally distributed with mean $(\alpha\beta)_{p,q}$ and variance σ_{AB}^2. For any fixed p, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all q. Similarly, for any fixed q, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all p.
- If any of the factors, such as A and B, associated with the effect of an interaction $(\alpha\beta)_{p,q}$ are random, the effect is taken to be a random variable normally distributed with mean 0 and variance σ_{AB}^2. If A is fixed but B is random, assume additionally that for any fixed q, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all p. Similarly, if B is fixed but A is random, assume additionally that for any fixed p, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all q.
- All effects taken to random variables are assumed to be independent.

<u>The Hypotheses</u>

Each of the following hypotheses are different ways of saying that a factor or an interaction among factors has no effect on experimental outcomes. Start by assuming that there are no effects and then seek evidence to contradict these assumptions. The seven hypotheses are as follows.

- For (A), $\alpha_p = 0$ for all levels of p if factor A is fixed; $\sigma_A^2 = 0$ if factor A is random
- For (B), $\beta_q = 0$ for all levels of q if factor B is fixed; $\sigma_B^2 = 0$ if factor B is random
- For (C), $\gamma_r = 0$ for all levels of r if factor C is fixed; $\sigma_C^2 = 0$ if factor C is random
- For (AB), $(\alpha\beta)_{p,q} = 0$ for all levels of p and q if factors A and B are fixed; $\sigma_{AB}^2 = 0$ if either factor A or B is random
- For (AC), $(\alpha\gamma)_{p,r} = 0$ for all levels of p and r if factors A and C are fixed; $\sigma_{AC}^2 = 0$ if either factor A or C is random
- For (BC), $(\beta\gamma)_{q,r} = 0$ for all levels of q and r if factors B and C are fixed; $\sigma_{BC}^2 = 0$ if either factor B or C is random
- For (ABC), $(\alpha\beta\gamma)_{p,q,r} = 0$ for all levels of p, q, and r if factors A, B, and C are fixed; $\sigma_{ABC}^2 = 0$ if any of the factors A, B, or C are random

<u>Testing the Hypotheses</u>

For each hypothesis, the function generates a number so that, if the hypothesis is true, that number will be from a particular F-distribution.

For example, in the fixed-effects model, the number fa = msa/mse (associated with hypothesis (A)) is from an F-distribution with a-1 and abc(L-1) degrees of freedom (F(a-1,abc(L-1))), given that hypothesis (A) is true. The function calculates the probability that a number taken from a particular F-distribution is larger than the F-value. For example,

sigA = Prob(X > fa)

where X is from F(a-1,abc(L-1)).

Use the probabilities **sigA**, **sigB**, **sigC**, **sigAB**, **sigAC**, **sigBC**, and **sigABC** to determine when to reject the associated hypotheses: (A), (B), (C), (AB), (AC), (BC), and (ABC). To do so, choose a level of significance for each hypothesis. The level of significance is how likely you want it to be that you reject the hypothesis when it is in fact true. Ordinarily you do not want it to be very likely that you reject the hypothesis when it is true, and so the level of significance should be small (0.05 is a common choice). Keep in mind that the smaller the level of significance, the more hesitant you are to reject the hypothesis.

A particular hypothesis is rejected when the associated output parameter **sigA**, **sigB**, **sigC**, **sigAB**, **sigAC**, **sigBC**, or **sigABC** is less than the level of significance chosen for that hypothesis. If A is a random effect, and the chosen level of significance is 0.05, and **sigA** = 0.03, you must reject the hypothesis that $\sigma_A^2 = 0$, and conclude that factor A does have an effect on the experimental observations.

With some models there are no appropriate tests for certain hypotheses. In these cases the output parameters directly involved with the testing of those hypotheses will be set to -1.0.

Formulas

Let $y_{p,q,r,s}$ be the s^{th} observation at the p^{th}, q^{th}, and r^{th} levels of A, B, and C, respectively, where s = 0, 1, ..., L-1.

Let

$$aa = |a|$$

$$bb = |b|$$

$$cc = |c|$$

$$T_{p,q,r} = \sum_{s=0}^{L-1} y_{p,q,r,s}$$

$$T_{p,q} = \sum_{r=0}^{cc-1} T_{p,q,r}$$

$$T_{p,r} = \sum_{q=0}^{bb-1} T_{p,q,r}$$

$$T_{q,r} = \sum_{p=0}^{aa-1} T_{p,q,r}$$

$$T_p = \sum_{q=0}^{bb-1} T_{p,q}$$

$$T_q = \sum_{p=0}^{aa-1} T_{p,q}$$

$$T_r = \sum_{p=0}^{aa-1} T_{p,r}$$

T = the total sum of all observations

$$A = \sum_{p=0}^{aa-1} T_p^{\ 2} / (bb * cc * L)$$

$$B = \sum_{q=0}^{bb-1} T_p^{\ 2} / (aa * cc * L)$$

$$C = \sum_{r=0}^{cc-1} T_r^{\ 2} / (aa * bb * L)$$

$$AB = \sum_{p=0}^{aa-1} \sum_{q=0}^{bb-1} T_{p,q}^{\ \ 2} / (cc * L)$$

$$AC = \sum_{p=0}^{aa-1} \sum_{r=0}^{cc-1} T_{p,r}^{\ \ 2} / (bb * L)$$

$$BC = \sum_{q=0}^{bb-1} \sum_{r=0}^{cc-1} T_{q,r}^{\ \ 2} / (aa * L)$$

$$S = \sum_{p=0}^{aa-1} \sum_{q=0}^{bb-1} \sum_{r=0}^{cc-1} T_{p,q,r}^{\ \ 2} / L$$

$$CF = T^2/(aa*bb*cc*L)$$

Then

ssa = A - CF	*msa = ssa/(aa-1) = ssa/dofa*
ssb = B - CF	*msb = ssb/(bb-1) = ssb/dofb*
ssc = C - CF	*msc = ssc/(cc-1) = ssc/dofc*
ssab = AB - A - B + CF	*msab = ssab/(aa-1)(bb-1) = ssab/dofab*
ssac = AC - A - C + CF	*msac = ssac/(aa-1)(cc-1) = ssac/dofac*
ssbc = BC - B - C + CF	*msbc = ssbc/(bb-1)(cc-1) = ssbc/dofbc*

$ssabc = S - AB - AC - BC + A + B + C - CF$ $msabc = ssabc/(aa-1)(bb-1)(cc-1)$
$= ssabc/dofabc$

$mse=sse/(aa*bb*cc)(L-1) = sse/dofe$

fa	= *msa/mse*	(if B and C are fixed)
	= *msa/msab*	(if B is random and C is fixed)
	= *msa/msac*	(if B is fixed and C is random)
fb	= *msb/mse*	(if A and C are fixed)
	= *msb/msab*	(if A is random and C is fixed)
	= *msb/msbc*	(if A is fixed and C is random)
fc	= *msc/mse*	(if A and B are fixed)
	= *msc/msac*	(if A is random and B is fixed)
	= *msc/msbc*	(if A is fixed and B is random)
fab	= *msab/mse*	(if C is fixed)
	= *msab/msabc*	(if C is random)
fac	= *msac/mse*	(if B is fixed)
	= *msac/msabc*	(if B is random)
fbc	= *msbc/mse*	(if A is fixed)
	= *msbc/msabc*	(if A is random)
fabc	= *msabc/mse*	

If $f = ms_1/ms_2$ and $ms_1 = ss_1/dof_1$ and $ms_2 = ss_2/dof_2$, we assume that f is from an F-distribution with dof_1 and dof_2 degrees of freedom ($F(dof_1, dof_2)$).

Example

Suppose that researchers want to know how the number of hours of sunlight, the amount of rainfall, and the average temperature affect the yield of a crop. Each factor (sunlight, rainfall, and temperature) is divided into three levels as follows.

Level	Sunlight (Factor A)
0	5 hours
1	6 hours
2	7 hours

Level	Rainfall (Factor B)
0	2 inches
1	3 inches
2	4 inches

Level	Temperature (Factor C)
0	76-80 degrees
1	81-85 degrees
2	86-90 degrees

A particular plot planted with the crop may be in any one of the 27 different combinations of these levels with the three factors. For example, one combination might be six hours of sunlight with two inches of rainfall and an average temperature between 76 and 80 degrees, recorded as (1,0,0). These combinations are called cells.

The researchers set up 54 plots in various geographical locations chosen so that two plots will fall in each of the 27 cells. To measure the productivity of a particular plot, they record the crop production. Let Sunlight be Factor A, Rainfall be Factor B, and Temperature be Factor C.

They record the following information.

(A, B, C)	**Bushels produced from each plot**
(0, 0, 0)	128 122
(0, 0, 1)	113 108
(0, 0, 2)	116 116
(0, 1, 0)	132 129
(0, 1, 1)	119 121
(0, 1, 2)	126 113
(0, 2, 0)	118 114
(0, 2, 1)	141 133
(0, 2, 2)	121 123
(1, 0, 0)	119 118
(1, 0, 1)	111 115
(1, 0, 2)	143 140
(1, 1, 0)	127 129
(1, 2, 2)	112 113

(A, B, C)	**Bushels produced from each plot**
(1, 1, 1)	128 120
(1, 1, 2)	122 121
(1, 2, 0)	114 115
(1, 2, 1)	116 113
(2, 0, 0)	113 125
(2, 0, 1)	135 131
(2, 0, 2)	145 145
(2, 1, 0)	152 147
(2, 1, 1)	137 141
(2, 1, 2)	171 171
(2, 2, 0)	143 144
(2, 2, 1)	145 147
(2, 2, 2)	121 123

To perform a three-way analysis of variance in the fixed effect model using the LabWindows `ANOVA3Way` function, all the numbers of bushels are stored in a double-precision array **y** of size 54. The integer arrays **levelA**, **levelB**, and **levelC** record the cells in which observations were made. For any particular i, these arrays are set such that y[i] is the number of bushels produced by a plot in the ($levelA_i$, $levelB_i$, $levelC_i$) cell. For example,

($levelA_i$, $levelB_i$, $levelC_i$) = (0, 1, 1)

$y_i = 119 \text{ or } 121$

are valid combinations.

Therefore, the input arrays **y**, **levelA**, **levelB**, and **levelC** in this example could be set up for the `ANOVA3Way` function as follows.

y = 128, 122, 113, 108, 116, 116, 132, 129, ...

levelA = 0, 0, 0, 0, 0, 0, 0, 0, ...

levelB = 0, 0, 0, 0, 0, 0, 1, 1, ...

levelC = 0, 0, 1, 1, 2, 2, 0, 0, ...

Running the code given in the examples below produces the following results.

sigA = $1.11e^{-16}$
sigB = $1.3e^{-8}$
sigC = 0.0072
sigAB = $1.2e^{-8}$
sigAC =$2.0e^{-4}$
sigBC = $4.5e^{-10}$
sigABC = $4.8e^{-10}$

For a level of significance such as 0.05, the `ANOVA3Way` results show that the researchers must reject the hypotheses that sunlight, rainfall and temperature have no effect on the yield of the crop. In other words, all three factors have a significant effect on crop yield.

<u>Example</u>

```
double y[54], sigA, sigB, sigC,sigAB, sigAC, sigBC, sigABC, info[8][4];

int levelA[54], levelB[54], levelC[54];
int L, a, b, c;
int status;

L = 2;         /* two observations per cell */
a = 3;         /* three levels for factor A, Sunlight */
b = 3;         /* three levels for factor B, Rainfall */
c = 3;         /* three levels for factor C, Temperature */

/* Read in recorded data y[54], levelA[54], levelB[54] and levelC[54] */

status = ANOVA3Way(y, levelA, levelB, levelC, 54, L, a, b, c,
         info, &sigA, &sigB, &sigC, &sigAB, &sigAC,
         &sigBC, &sigABC);
```

ArbitraryWave

`int` **status** = **ArbitraryWave** (`int` **n**, `double` **amp**, `double` **f**, `double` *__phase,__
`double` **waveTable**`[]`, `int` **tableSize**, `int` **interp**,
`double` **x**`[]`);

Purpose

Generates an array containing an arbitrary wave, with each cycle described by an interpolated version of the specified **waveTable**. The output array **x** is generated according to the following formula.

$x_i = amp * arb(*phase + f * 360.0 * i)$

where
arb(p) = WT(p modulo 360.0)
f = frequency, cycles/sample

WT(x) is computed according to the following interpolation values.

$$WT(x) = \begin{cases} waveTable_{ix} & \text{for interp } = 0 \\ waveTable_{ix} + dx * (waveTable_{(ix+1)\%tableSize} - waveTable_{ix}) & \text{for interp } = 1 \end{cases}$$

where
ix = (int)x
dx = x - (int)x

and (int) is the integral part of the variable x. This function can be used to simulate a continuous acquisition from an arbitrary wave function generator. The unit of the input ***phase** is in degrees, and ***phase** is set to

(*phase + f * 360.0 * n) modulo 360 before returning.

Parameters

Input	**n**	integer	number of samples to generate.
	amp	double-precision	amplitude of the generated signal.
	f	double-precision	frequency of the generated signal, in normalized units of cycles/sample.
	phase	double-precision	points to the initial **phase**, in degrees, of the generated signal.
	waveTable	double-precision array	contains equally-spaced samples of one cycle of the generated signal.
	tableSize	integer	number of elements contained in the **waveTable** array.
	interp	integer	determines the type of interpolation used in generating the arbitrary wave signal from the **waveTable** samples. 0 = No Interpolation 1 = Linear Interpolation
Output	**phase**	double-precision	upon completion of this function, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to this function to simulate a continuous function generator.
	x	double-precision array	contains the generated arbitrary wave signal.

Return Value

status	integer	refer to error codes in Appendix A

AutoPowerSpectrum

`int` **status** = **AutoPowerSpectrum** (`double` **x**`[]`, `int` **n**, `double` **dt**,
`double` **autoSpectrum**`[]`, `double` *__df__);

Purpose

Computes the single-sided, scaled auto power spectrum of a time-domain signal. The auto power spectrum is defined as

$$FFT(X)\ FFT^*(X) / \mathbf{n}^2$$

where **n** is the number of points in the signal array X and * denotes complex conjugate. The auto power spectrum is converted to a single-sided form.

Parameters

Input	**x**	double-precision array	contains the time-domain signal.
	n	integer	number of elements in the input array. **n** must be a power of 2.
	dt	double-precision	**dt** is the sample period of the time-domain signal, usually in seconds. **dt** = $1/_{fs}$, where fs is the sampling frequency of the time-domain signal.
Output	**autoSpectrum**	double-precision array	**autoSpectrum** is the single-sided amplitude spectrum magnitude in volts RMS if the input signal is in volts. If the input signal is not in volts, the results are in input signal units RMS. This array must be at least **n**/2 elements long.
	df	double-precision	**df** points to the frequency interval, in hertz, if **dt** is in seconds. ***df** = 1/(**n*dt**)

Return Value

status	integer	refer to error codes in Appendix A

BackSub

`int` **status** = **BackSub** (`void` ***a**, `double` **y**`[]`, `int` **n**, `double` **x**`[]`);

Purpose

Solves the linear equations a*x = y by backward substitution. **a** is assumed to be an **n** by **n** lower triangular matrix whose diagonal elements are all ones. **x** is obtained by the following formulas.

$$x_{n-1} = y_{n-1} \,/\, a_{n-1,n-1}$$

$$x_i = (y_i - \sum_{j=i+1}^{n-1} a_{i,j} * x_j) / a_{i,i} \qquad \text{for i = n-2, n-3,…0}$$

The operation can be performed in place; that is, **x** and **y** can be the same array. `BackSub` is used in conjunction with `LU` and `ForwSub` to solve linear equations.

Refer to the `LU` function description for more information.

Parameters

Input	**a**	double-precision 2D array	input matrix
	y	double-precision array	input vector
	n	integer	dimension size of **a**
Output	**x**	double-precision array	solution vector

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/*To solve a linear equation A*x = y   */
double  A[10][10], x[10], y[10];
int p[10];  /* permutation vector */
int sign, n;
n = 10;
LU(A,n,p,&sign);  /* LU decomposition of A */
ForwSub(A,y,n,x,p);  /* forward substitution */
BackSub(A,x,n,x); /* backward substitution */
```

Bessel_CascadeCoef

`int` **status** = **Bessel_CascadeCoef** (`double` **fs**, `double` **fL**, `double` **fH**,
`IIRFilterPtr` **filterInformation**);

Purpose

Generates the set of cascade form filter coefficients to implement an IIR filter as specified by the Bessel filter model.

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function.

To redesign another filter, you should first call `FreeIIRFilterPtr` to free the present filter structure and then call `AllocIIRFilterPtr` with the new type and order parameters before calling this design function.

If the type and order remain the same, and you can call this IIR design function without calling `FreeIIRFilterPtr` and `AllocIIRFilterPtr.` In this case, you should properly reset the filtering operation for that structure by calling `ResetIIRFilter` before the first call to `IIRCascadeFiltering`.

Parameters

Input	**fs**	double-precision	Specifies the sampling frequency in Hz.
	fL	double-precision	Specifies the desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	Specifies the desired upper cutoff frequency of the filter in Hz.
Output	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/* Design a cascade lowpass Bessel IIR filter   */
double      fs, fl, fh, x[256], y[256];
int      type,order,n;
IIRFilterPtr   filterInfo;
n = 256;
fs = 1000.0;
fl = 200.0;
order = 5;
type = 0;      /* lowpass  */
Uniform(n,17,x);
```

```
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   Bessel_CascadeCoef(fs,fl,fh,filterInfo);
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

Bessel_Coef

`int` **status** = **Bessel_Coef** (`int` **type**, `int` **order**, `double` **fs**, `double` **fL**, `double` **fH**,`double` **a**`[]`, `int` **na**, `double` **b**`[]`, `int` **nb**);

Purpose

Generates the set of filter coefficients to implement an IIR filter as specified by the Bessel filter model. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \text{lowpass filter, } \textbf{fH} \text{ is not used.} \\ 1 & \text{highpass filter, } \textbf{fH} \text{ is not used.} \\ 2 & \text{bandpass filter} \\ 3 & \text{bandstop filter} \end{cases}$$

a[**na**] and **b**[**nb**] are the reverse and forward filter coefficients. The actual filtering

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-i}\right)$$

is achieved by using the function `IIRFiltering`.

Parameters

Input	**type**	integer	controls the filter type of the Bessel IIR filter coefficients.
	order	integer	order of the IIR filter.
	fs	double-precision	sampling frequency in Hz.
	fL	double-precision	desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	desired higher cutoff frequency of the filter in Hz.
	na	integer	number of coefficients in the **a** coefficient array.
	nb	integer	number of coefficients in the **b** coefficient array.
Output	**a**	double-precision array	array containing the *reverse* coefficients of the designed IIR filter.
	b	double-precision array	array containing the *forward* coefficients of the designed IIR filter.

Return Value

status	integer	refer to error codes in Appendix A

BlkHarrisWin

`int` **status** = **BlkHarrisWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a 3-term Blackman-Harris window to the input sequence X. If Y represents the output sequence, the elements of Y are obtained using the equation

$$Y_i = X_i\,(0.42323 - 0.49755\cos(2\pi i/n) + 0.07922\cos(4\pi i/n))$$

where **n** is the number of elements in X.

Parameters

Input	**x** **n**	double-precision array integer	contains the input signal. The number of elements in the input array.
Output	**x**	double-precision array	contains the signal after applying the Blackman-Harris window.

Return Value

status	integer	refer to error codes in Appendix A

BkmanWin

`int` **status** = **BkmanWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a Blackman window to the **x** input signal. The Blackman window is defined by the following formula.

$w_i = 0.42 - 0.5\cos(2\pi i/n) + 0.08\cos(4\pi i/n)$ *for* $i = 0, 1, ..., n-1$

The output signal is obtained by the following formula.

$x_i = x_i * w_i$ for $i = 0, 1, ..., n-1$

The window operation is performed in place. The windowed data **x** replaces the input data **x**.

Parameters

Input	**x** **n**	double-precision array integer	input data number of elements in **x**
Output	**x**	double-precision array	windowed data

Return Value

status	integer	refer to error codes in Appendix A

Bw_BPF

`int` **status** = **Bw_BPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl,** `double` **fh**,
`int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandpass Butterworth filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandpass
Butterworth filter.  The pass band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
order = 5;
Uniform (n, 17, x);
status = Bw_BPF (x, n, fs, fl, fh, order, y);
```

Bw_BSF

`int` **status** = **Bw_BSF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**,
`int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandstop Butterworth filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandstop
Butterworth filter.  The stop band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
order = 5;
Uniform (n, 17, x);
status = Bw_BSF (x, n, fs, fl, fh, order, y);
```

Bw_CascadeCoef

`int` **status** = **Bw_CascadeCoef** (`double` **fs**, `double` **fL**, `double` **fH**, `IIRFilterPtr` **filterInformation**);

Purpose

Generates the set of cascade form filter coefficients to implement an IIR filter as specified by the Butterworth filter model.

filterInformationis the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function.

To redesign another filter, you should first call `FreeIIRFilterPtr` to free the present filter structure and then call `AllocIIRFilterPtr` with the new type and order parameters before calling this design function.

If the type and order remain the same, and you can call this IIR design function without calling `FreeIIRFilterPtr` and `AllocIIRFilterPtr`. In this case, you should properly reset the filtering operation for that structure by calling `ResetIIRFilter` before the first call to `IIRCascadeFiltering`.

Parameters

Input	**fs**	double-precision	Specifies the sampling frequency in Hz.
	fL	double-precision	Specifies the desired lower cutoff frequency of the filter in Hz.
	FH	double-precision	Specifies the desired upper cutoff frequency of the filter in Hz
Output	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure whichcontains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/* Design a cascade lowpass Butterworth IIR filter */
double      fs, fl, fh, x[256], y[256];
int      type,order,n;
IIRFilterPtr   filterInfo;
n = 256;
fs = 1000.0;
fl = 200.0;
order = 5;
type = 0;      /* lowpass  */
Uniform(n,17,x);
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   Bw_CascadeCoef(fs,fl,fh,filterInfo);
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

Bw_Coef

`int` **status** = **Bw_Coef** (`int` **type**, `int` **order**, `double` **fs** `double` **fL**, `double` **fH**, `double` **a**`[]`, `int` **na**, `double` **b**`[]`, `int` **nb**);

Purpose

Generates the set of filter coefficients to implement an IIR filter as specified by the Butterworth filter model. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \text{lowpass filter, } \textbf{fH} \text{ is not used.} \\ 1 & \text{highpass filter, } \textbf{fH} \text{ is not used.} \\ 2 & \text{bandpass filter} \\ 3 & \text{bandstop filter} \end{cases}$$

a[**na**] and **b**[**nb**] are the reverse and forward filter coefficients. The actual filtering is achieved by using the function `IIRFiltering`.

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-i}\right)$$

Parameters

Input	**type**	integer	controls the filter type of the Butterworth IIR filter coefficients.
	order	integer	order of the IIR filter.
	fs	double-precision	sampling frequency in Hz.
	fL	double-precision	desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	desired higher cutoff frequency of the filter in Hz.
	na	integer	number of coefficients in the **a** coefficient array.
	nb	integer	number of coefficients in the **b** coefficient array.
Output	**a**	double-precision array	array containing the *reverse* coefficients of the designed IIR filter.
	b	double-precision array	array containing the *forward* coefficients of the designed IIR filter.

Return Value

status	integer	refer to error codes in Appendix A

Bw_HPF

`int` **status** = **Bw_HPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital highpass Butterworth filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order highpass
Butterworth filter. */
double  x[256], y[256], fs, fc;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fc = 200.0;
order = 5;
Uniform (n, 17, x);
status = Bw_HPF (x, n, fs, fc, order, y);
```

Bw_LPF

`int` **status** = **Bw_LPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital lowpass Butterworth filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order lowpass
Butterworth filter. */
double  x[256], y[256], fs, fc;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fc = 200.0;
order = 5;
Uniform (n, 17, x);
status = Bw_LPF (x, n, fs, fc, order, y);
```

CascadeToDirectCoef

`int` **status** = **CascadeToDirectCoef** (`IIRFilterPtr` **filterInformation**, `double` **a**`[]`,`int` **na**, `double` **b**`[]`, `int` **nb**);

Purpose

Converts from the cascade IIR coefficients contained by the **filterInformation** structure to direct form IIR coefficients in arrays a and b. These two arrays must be allocated as for the old-style direct coefficient design functions (`Bw_Coef`,...).

For lowpass and highpass type filters, the direct coefficient arrays must have size (order + 1).

For bandpass and bandstop type filters, the direct coefficient arrays must have size (2*order + 1).

Parameters

Input	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling one of the cascade IIR filter design functions. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.
	na	integer	Specifies the number of coefficients in array a. **na** = order+1 for low or high pass filters **na** = 2*order+1 for bandpass or bandstop filters.
	nb	integer	Specifies the number of coefficients in the B Coefficient Array. **nb** = order+1 for low or high pass filters **nb** = 2*order+1 for bandpass or bandstop filters.
Output	**a**	double-precision array	Array containing the *reverse* coefficients of the direct form IIR filter.
	b	double-precision array	Array containing the *forward* coefficients of the direct form IIR filter.

Return Value

status	integer	Refer to error codes in Appendix A.

Ch_BPF

`int` **status** = **Ch_BPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl,** `double` **fh**,
`double` **ripple**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandpass Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	ripple	double-precision	pass band ripples in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandpass
Chebyshev filter.  The pass band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh, ripple;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
ripple = 0.5;
order = 5;
Uniform (n, 17, x);
status = Ch_BPF (x, n, fs, fl, fh, ripple, order, y);
```

Ch_BSF

`int` **status** = **Ch_BSF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**, `double` **ripple**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandstop Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	ripple	double-precision	pass band ripples in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandstop
Chebyshev filter.  The stop band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh, ripple;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
ripple = 0.5;
order = 5;
Uniform (n, 17, x);
status = Ch_BSF (x, n, fs, fl, fh, ripple, order, y);
```

Ch_CascadeCoef

`int` **status** = **Ch_CascadeCoef** (`double` **fs**, `double` **fL**, `double` **fH**, `double` **ripple**, `IIRFilterPtr` **filterInformation**);

Purpose

Generates the set of cascade form filter coefficients to implement an IIR filter as specified by the Chebyshev filter model.

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function.

To redesign another filter, you should first call `FreeIIRFilterPtr` to free the present filter structure and then call `AllocIIRFilterPtr` with the new type and order parameters before calling this design function.

If the type and order remain the same, and you can call this IIR design function without calling `FreeIIRFilterPtr` and `AllocIIRFilterPtr.` In this case, you should properly reset the filtering operation for that structure by calling `ResetIIRFilter` before the first call to `IIRCascadeFiltering`.

Parameters

Input	**fs**	double-precision	Specifies the sampling frequency in Hz.
	fL	double-precision	Specifies the desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	Specifies the desired upper cutoff frequency of the filter in Hz.
	ripple	double-precision	Specifies the amplitude of the stop band ripple in decibels.
Output	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure whichcontains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/* Design a cascade lowpass Chebyshev IIR filter   */
double      fs, fl, fh, ripple, x[256], y[256];
int      type,order,n;
IIRFilterPtr   filterInfo;
n = 256;
```

```
fs = 1000.0;
fl = 200.0;
ripple = 0.5;
order = 5;
type = 0;      /* lowpass  */
Uniform(n,17,x);
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   Ch_CascadeCoef(fs,fl,fh,ripple,filterInfo);
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

Ch_Coef

`int` **status** = **Ch_Coef** (`int` **type**, `int` **order**, `double` **fs**, `double` **fL**, `double` **fH**, `double` **ripple**, `double` **a**`[]`, `int` **na,** `double` **b**`[]`**,** `int` **nb**);

Purpose

Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev filter model. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \text{lowpass filter, } \textbf{fH} \text{ is not used.} \\ 1 & \text{highpass filter, } \textbf{fH} \text{ is not used.} \\ 2 & \text{bandpass filter} \\ 3 & \text{bandstop filter} \end{cases}$$

a[**na**] and **b**[**nb**] are the reverse and forward filter coefficients. The actual filtering

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i \quad x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-1}\right)$$

is achieved by using the function `IIRFiltering`.

Parameters

Input	**type**	integer	controls the filter type of the Chebyshev IIR filter coefficients.
	order	integer	order of the IIR filter.
	fs	double-precision	sampling frequency in Hz.
	fL	double-precision	desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	desired higher cutoff frequency of the filter in Hz.
	ripple	double-precision	amplitude of the **stopband** ripple in decibels.
	na	integer	number of coefficients in the **a** coefficient array.
	nb	integer	number of coefficients in the **b** coefficient array.
Output	**a**	double-precision array	array containing the *reverse* coefficients of the designed IIR filter.
	b	double-precision array	array containing the *forward* coefficients of the designed IIR filter.

Return Value

status	integer	refer to error codes in Appendix A

Ch_HPF

`int` **status** = **Ch_HPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `double` **ripple**,
`int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital highpass Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	ripple	double-precision	pass band ripples in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order highpass
Chebyshev filter. */
double  x[256], y[256], fs, fc, ripple;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fc = 200.0;
ripple = 0.5;
order = 5;
Uniform (n, 17, x);
status = Ch_HPF (x, n, fs, fc, ripple, order, y);
```

Ch_LPF

`int` **status** = **Ch_LPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `double` **ripple**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital lowpass Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	ripple	double-precision	pass band ripples in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order lowpass
Chebyshev filter. */
double  x[256], y[256], fs, fc, ripple;
int  n, order;
int status;
n = 256;
fs = 1000.0;
fc = 200.0;
ripple = 0.5;
order = 5;
Uniform (n, 17, x);
status = Ch_LPF (x, n, fs, fc, ripple, order, y);
```

Chirp

`int` **status** = **Chirp** (`int` **n**, `double` **amp**, `double` **f1**, `double` **f2**, `double` **x**`[]`);

Purpose

Generates an array containing a chirp pattern. The output array **x** is generated according to the following formula.

$$x_i = amp * sin((\frac{a}{2}\ i + b)\ i)$$

where

a = 2π*(**f2**-**f1**)/n

b = 2π***f1**

f1 = beginning frequency, cycles/sample
f2 = ending frequency, cycles/sample

Parameters

Input	**n**	integer	number of samples to generate.
	amp	double-precision	amplitude of the resulting signal.
	f1	double-precision	beginning frequency of the resulting signal in normalized units of cycles/sample.
	f2	double-precision	ending frequency of the resulting signal in normalized units of cycles/sample.
Output	**x**	double-precision array	contains the generated chirp pattern.

Return Value

status	integer	refer to error codes in Appendix A

Clear1D

`int` **status** = **Clear1D** (`double` **x**`[]`, `int` **n**);

Purpose

Sets the elements of the **x** array to 0.0.

Parameters

Input	**n**	integer	number of elements in **x**
Output	**x**	double-precision array	cleared array

Return Value

status	integer	refer to error codes in Appendix A

Clip

`int` **status** = **Clip** (`double` **x**`[]`, `int` **n**, `double` **upper**, `double` **lower**, `double` **y**`[]`);

Purpose

Clips the input array values. The range of the resulting output array is [**lower** : **upper**]. The i^{th} element of the resulting array is obtained by using the following formula.

$$y_i = \begin{cases} \text{upper} & \text{if} \quad x_i > \text{upper} \\ \text{lower} & \text{if} \quad x_i < \text{lower} \\ x_i & \text{otherwise} \end{cases}$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	upper	double-precision	upper limit
	lower	double-precision	lower limit
Output	**y**	double-precision array	clipped array

Return Value

status	integer	refer to error codes in Appendix A

Contingency_Table

`int` **status** = **Contingency_Table** (`int` **s**, `int` **k**, `int void` ***y,** `double` ***Test_Stat**, `double` ***Sig**);

Purpose

Creates a contingency table in which objects of experimentation are classified and tallied according to two schemes of categorization. This function can be used to perform a test of homogeneity or a test of independence.

Note: ***For both tests, the math is identical. It is not necessary to specify which test is being applied. The only difference is in the hypothesis being tested.***

Parameters

Input	**s**	integer	number of random samples taken in the test of homogeneity, or the number of categories in the first categorization scheme in the test of independence.
	k	integer	number of categories in the test of homogeneity, or the number of categories in the second scheme in the test of independence.
	y	integer 2D array	contingency table, indexed as an **s** by **k** matrix.
Output	**Test_Stat**	double-precision	used to calculate **Sig**. If the hypothesis is true, **Test_Stat** is known to come from a chi-square distribution with (s-1)*(k-1) degrees of freedom.
	Sig	double-precision	level of significance at which the hypothesis must be rejected.

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

A contingency table is a table in which objects of experimentation are classified and tallied according to two schemes of categorization. For example, if the objects of experimentation are individuals, one scheme might be political affiliation: Know-Nothing, Tory, Whig, Mugwump, and so on. Another scheme might be to classify individuals according to how they vote on some issue.

Chi-Square Test of Homogeneity

Take a random sample of some fixed sized from each of the categories in one categorization scheme for the chi-square test of homogeneity. For each of the samples, categorize the objects of experimentation according to the second scheme, and tally them. For example, you might pick 100 Know-Nothings, 100 Whigs, 100 Tories and 100 Mugwumps. Count the number of individuals who vote a certain way for each category. This produces the following contingency table.

	Yes	**No**	**Undecided**
Know-Nothing	36	24	40
Whig	12	53	35
Tory	61	11	28
Mugwump	83	3	14

Notice that the sum of each of the rows equals 100.

Test the hypothesis that the populations from which each sample is taken are identically distributed with respect to the second categorization scheme. For example, you can test the hypothesis that the four samples of politically affiliated individuals are distributed identically with respect to the way they vote. If this hypothesis is true, it means that a Mugwump selected at random is just as likely to vote yes as a Whig selected at random.

Chi-Square Test of Independence

Take only one sample from the total population for the chi-square test of independence. Categorize each object of experimentation and tally them in the two categorization schemes. If you select 500 individuals, for example, you might arrive at the following table.

	Yes	**No**	**Undecided**
Know-Nothing	18	15	18
Whig	55	93	38
Tory	101	83	20
Mugwump	16	31	12

Notice that the sum of each row is different, but that the total number of individuals tallied is 500.

Test the hypothesis that the categorization schemes are independent. For example, if you choose a person at random and he or she turns out to be a Mugwump, then the hypothesis says his or her political affiliation has no impact on how he or she votes on the selected issue.

Testing The Hypothesis

Whichever test is being used, a level of significance must be chosen. This is how likely you want it to be that a true hypothesis is rejected. Ordinarily you do not want it to be very likely, so the level of significance should be small (0.05, or 5%, is a common choice).

The output parameter **Sig** is the level of significance at which the hypothesis is rejected.
Sig = Prob($\chi \geq$ **Test_Stat**), where χ is a random variable from the chi-square distribution with

(s-1)(k-1) degrees of freedom. If **Sig** is less than the level of significance, the hypothesis must be rejected.

Formulas

Let $y_{p,q}$ be the number of occurrences in the $(p,q)^{th}$ cell of the contingency table for p = 0, 1, ..., (s-1) and q = 0, 1, ..., (k-1).

Let

$$y_p = \sum_{q=0}^{k-1} y_{p,q}$$

$$y_q = \sum_{p=0}^{s-1} y_{p,q}$$

$$y = \sum_{p=0}^{s-1} \sum_{q=0}^{k-1} y_{p,q}$$

$$e_{p,q} = (y_p * y_q)/y$$

$$Test\_Stat = \sum_{p=0}^{s-1} \sum_{q=0}^{k-1} \frac{[y_{p,q} - e_{p,q}]^2}{e_{p,q}}$$

Example

```
/* Generate random contingency table.  Because rows will not have identical
sums, use the chi-square test of independence. */
int s=10, k=10, y[10][10], i, j, status;
double Test_Stat, Sig, temp[1];
for(i=0; i<s; i++)
   for(j=0; j<k; j++)
   {
      WhiteNoise (1, 5, ,17, temp);
      temp[0] += 6.0;
      y[i][j] = (int) temp[0];
   }
status = Contingency_Table (s, k, y, &Test_Stat, &Sig);
```

Convolve

`int` **status = Convolve** (`double` **x**`[]`, int **n**, `double` **y**`[]`, `int` **m**, `double` **cxy**`[]`);

Purpose

Finds the convolution of the **x** and **y** input arrays. The convolution is obtained by the following formula.

$$cxy_i = \sum_{k=a}^{b} x_k * y_{i-k}$$

where	$a = 0, b = i$	for $0 \le i < m$
	$a = i - m + 1, b = i$	for $m \le i < n$
	$a = i - m + 1, b = n - 1$	for $n \le i \le n + m - 1$

Note: ***This formula description assumes that m ≤ n. For m > n, exchange (x, y) and (m, n) in the above equations.***

Parameters

Input	**x**	double-precision array	**x** input array
	n	integer	number of elements in **x**
	y	double-precision array	**y** input array
	m	integer	number of elements in **y**
Output	**cxy**	double-precision array	convolution array

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

The size of the output array must be at least (**n** + **m** - 1) elements long. This algorithm executes more efficiently if the sizes of the input arrays are a power of two.

Example

```
/* Generate two arrays with random numbers and find their
   convolution.   */
double  x[256], y[256], cxy[512];
int  n, m;
```

```
n = 256;
m = 256;
Uniform (n, 17, x);
Uniform (m, 17, y);
Convolve (x, n, y, m, cxy);
```

Copy1D

`int` **status** = **Copy1D** (`double` **x**, `int` **n**, `double` **y**`[]`);

Purpose

Copies the elements of the **x** array. This function is useful to duplicate arrays for in-place operations.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
Output	**y**	double-precision array	duplicated array

Return Value

status	integer	refer to error codes in Appendix A

Correlate

`int` **status** = **Correlate** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`, `int` **m**, `double` **rxy**`[]`);

Purpose

Finds the correlation of the input arrays. The correlation is obtained by the following formula.

$$Rxy_i = \sum_{k=0}^{m-1} x_{k+n-1-i} * y_k$$

$y_j = 0$ when $j < 0$ or $j \geq m$

and $x_i = 0$ when $j < 0$ or $j \geq n$

Parameters

Input	**x**	double-precision array	**y** input array
	n	integer	number of elements in **x**
	y	double-precision array	**y** input array
	m	integer	number of elements in **y**
Output	**rxy**	double-precision array	correlation array

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

The size of the output array must be at least (**n** + **m** -1) elements long.

Example

```
/* Generate two arrays with random numbers and find their correlation. */
double  x[256], y[256], cxy[512];
int  n, m;
n = 256;
m = 256;
Uniform (n, 17, x);
Uniform (m, 17, y);
Correlate (x, n, y, m, cxy);
```

CosTaperedWin

`int` **status** = **CosTaperedWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a cosine tapered window to the input sequence X. If Y represents the output sequence, the elements of Y are obtained from the equation:

$$y_i = \begin{cases} 0.5\ x_i\ (1-\cos(2 p\ i/n)) & i = 0, 1, \ldots, m-1 \\ x_i & i = m, m+1, \ldots, n-m-1 \\ 0.5\ x_i\ (1-\cos(2 p\ i/n)) & i = n-m, n-m+1, \ldots, n-1 \end{cases}$$

where m = round (**n**/10)

Parameters

Input	**x** **n**	double-precision array integer	contains the input signal. number of elements in the input array.
Output	**x**	double-precision array	contains the signal after applying the Tapered Cosine window.

Return Value

status	integer	refer to error codes in Appendix A

CrossPowerSpectrum

`int` **status** = **CrossPowerSpectrum** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **dt**, `double` **magSxy**`[]`, `double` **phaseSxy**`[]`, `double` ***df**);

Purpose

Computes the single-sided, scaled cross power spectrum of two time-domain signals. The cross power spectrum is defined as:

$$Sxy = FFT(Y)\,FFT^*(X) / (\mathbf{n}^2)$$

where **n** is the number of points in arrays X and Y. **magSxy** and **phaseSxy** are single-sided magnitude and phase spectra of Sxy.

Parameters

Input	**x** **y** **n** **dt**	double-precision array double-precision array integer double-precision	time-domain signal X. time-domain signal Y. The number of elements in the input array. Valid Values: Powers of 2. **dt** is the sample period of the time-domain signal, usually in seconds. **dt** = $1/_{fs}$, where fs is the sampling frequency of the time-domain signal.

(continues)

Parameters (Continued)

Output	**magSxy**	double-precision array	**magSxy** is the single-sided magnitude cross power spectrum between signals X and Y in volts RMS squared if the input signals are in volts. If the input signals are not in volts, the results are in input signal units RMS squared. This array must be at least **n**/2 elements long.
	phaseSxy	double-precision array	**phaseSxy** is the single-sided phase cross spectrum in radians showing the difference between the phases of signal Y and signal X. This array must be at least **n**/2 elements long.
	df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. *__df__ = 1/(**n***dt**)

Return Value

status	integer	refer to error codes in Appendix A

CrossSpectrum

`int` **status** = **CrossSpectrum** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **realSxy**`[]`,
`double` **imagSxy**`[]`);

Purpose

Computes the double-sided cross power spectrum, Sxy, of the input sequences X and Y according to the following formula.

$$\boldsymbol{CrossSpectrum} = \frac{FFT^{*}(X)FFT(Y)}{n^{2}}$$

where **n** is the number of samples in both input sequences, and FFT*[X] is the complex conjugate of FFT[X]. **n** must be a power of 2. The input sequences are copied to internal buffers before the FFTs are computed. The output arrays are the real and imaginary parts of the cross spectrum **CrossSpectrum**.

Parameters

Input	**x**	double-precision array	time-domain signal X.
	y	double-precision array	time-domain signal Y.
	n	integer	number of elements in the input arrays. The number must be a power of 2.
Output	**realSxy**	double-precision array	real part of the double-sided cross power spectrum between signals X and Y. The size of this array must be **n**.
	imagSxy	double-precision array	imaginary part of the double-sided cross power spectrum between signals X and Y. The size of this array must be **n**.

Return Value

status	integer	refer to error codes in Appendix A

CxAdd

`int` **status** = **CxAdd** (`double` **xr**, `double` **xi**, `double` **yr**, `double` **yi**, `double` ***zr**, `double` ***zi**);

Purpose

Adds two complex numbers. The resulting complex number is obtained using the following formulas.

$zr = xr + yr$

$zi = xi + yi$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
	yr	double-precision	real part of y
	yi	double-precision	imaginary part of y
Output	**zr**	double-precision pointer	real part of z
	zi	double-precision pointer	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxAdd1D

`int` **status** = **CxAdd1D** (`double` **xr**`[]`, `double` **xi**`[]`, `double` **yr**`[]`, `double` **yi**`[]`, `int` **n**,`double` **zr**`[]`, `double` **zi**`[]`);

Purpose

Adds two 1D complex arrays. The ith element of the resulting complex array is obtained using the following formulas.

$zr_i = xr_i + yr_i$

$zi_i = xi_i + yi_i$

The operations can be performed in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**xr**	double-precision array	real part of x
	xi	double-precision array	imaginary part of x
	yr	double-precision array	real part of y
	yi	double-precision array	imaginary part of y
	n	integer	number of elements
Output	**zr**	double-precision array	real part of z
	zi	double-precision array	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxDiv

`int` **status** = **CxDiv** (`double` **xr**, `double` **xi**, `double` **yr**, `double` **yi**, `double` *__zr__, `double` *__zi__);

Purpose

Divides two complex numbers. The resulting number is obtained using the following formulas.

$$zr = (xr*yr + xi*yi) / (yr^2 + yi^2)$$

$$zi = (xi*yr - xr*yi) / (yr^2 + yi^2)$$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
	yr	double-precision	real part of y
	yi	double-precision	imaginary part of y
Output	**zr**	double-precision	real part of z
	zi	double-precision	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxDiv1D

`int` **status = CxDiv1D** (`double` **xr**`[]`, `double` **xi**`[]`, `double` **yr**`[]`, `double` **yi**`[]`,
`int` **n**,`double` **zr**`[]`, `double` **zi**`[]`);

Purpose

Divides two 1D complex arrays. The i^{th} element of the resulting complex array is obtained using the following formula.

$$zr_i = (xr_i * yr_i + xi_i * yi_i) / (yr_i^2 + yi_i^2)$$

$$zi_i = (xi_i * yr_i - xr_i * yi_i) / (yr_i^2 + yi_i^2)$$

zr can be in place with xr; zi can be in place with xi.

Parameters

Input	**xr**	double-precision array	real part of x
	xi	double-precision array	imaginary part of x
	yr	double-precision array	real part of y
	yi	double-precision array	imaginary part of y
	n	integer	number of elements
Output	**zr**	double-precision array	real part of z
	zi	double-precision array	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxExp

`int` **status** = **CxExp** (`double` **xr**, `double` **xi**, `double` ***yr**, `double` ***yi**);

Purpose

Computes the exponential of a complex number. The resulting complex number is obtained using the following formula.

$$(yr, yi) = e^{(xr, xi)}$$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxLinEv1D

`int` **status =CxLinEv1D** (`double` **xr**`[]`, `double` **xi**`[]`, `int` **n**, `double` **ar**, `double` **ai**, `double` **br**, `double` **bi**, `double` **yr**`[]`, `double` **yi**`[]`);

Purpose

Performs a complex linear evaluation of a 1D complex array. The i^{th} element of the resulting complex array is obtained using the following formulas.

$$yr_i = ar*xr_i - ai*xi_i + br$$

$$yi_i = ar*xi_i + ai*xr_i + bi$$

The operations can be performed in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**xr**	double-precision array	real part of x
	xi	double-precision array	imaginary part of x
	n	integer	number of elements
	ar	double-precision	real part of a
	ai	double-precision	imaginary part of a
	br	double-precision	real part of b
	bi	double-precision	imaginary part of b
Output	**yr**	double-precision array	real part of y
	yi	double-precision array	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxLn

`int` **status** = **CxLn** (`double` **xr**, `double` **xi**, `double` ***yr**, `double` ***yi**);

Purpose

Computes the natural logarithm of a complex number. The resulting complex number is obtained using the following formula.

$$(yr, yi) = Log_e(xr, xi)$$

where e = 2.718….

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxLog

`int` **status** = **CxLog** (`double` **xr**, `double` **xi**, `double` ***yr**, `double` ***yi**);

Purpose

Computes the logarithm (base 10) of a complex number. The resulting complex number is obtained using the following formula.

$$(yr, yi) = Log_{10}(xr, xi)$$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxMul

`int` **status** = **CxMul** (`double` **xr**, `double` **xi**, `double` **yr**, `double` **yi**, `double` ***zr**, `double` ***zi**);

Purpose

Multiplies two complex numbers. The resulting complex number is obtained using the following formulas.

$$zr = xr*yr - xi*yi$$

$$zi = xr*yi + xi*yr$$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
	yr	double-precision	real part of y
	yi	double-precision	imaginary part of y
Output	**zr**	double-precision	real part of z
	zi	double-precision	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxMul1D

`int` **status** = **CxMul1D** (`double` **xr**`[]`, `double` **xi**`[]`, `double` **yr**`[]`, `double` **yi**`[]`,
`int` **n**, `double` **zr**`[]`, `double` **zi**`[]`);

Purpose

Multiplies two 1D complex arrays. The i^{th} element of the resulting complex array is obtained using the following formulas.

$zr_i = xr_i * yr_i - xi_i * yi_i$

$zi_i = xr_i * yi_i + xi_i * yr_i$

The operations can be performed in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**xr**	double-precision array	real part of x
	xi	double-precision array	imaginary part of x
	yr	double-precision array	real part of y
	yi	double-precision array	imaginary part of y
	n	integer	number of elements
Output	**zr**	double-precision array	real part of z
	zi	double-precision array	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxPow

`int` **status** = **CxPow** (`double` **xr**, `double` **xi**, `double` **a**, `double` ***yr**, `double` ***yi**);

Purpose

Computes the power of a complex number. The resulting complex number is obtained using the following formula.

$(yr, yi) = (xr, xi)^a$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
	a	double-precision	exponent
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxRecip

`int` **status = CxRecip** (`double` **xr**, `double` **xi**, `double` ***yr**, `double` ***yi**);

Purpose

Finds the reciprocal of a complex number. The resulting complex number is obtained using the following formulas.

$$yr = xr / (xr^2 + xi^2)$$

$$yi = -xi / (xr^2 + xi^2)$$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxSqrt

`int` **status** = **CxSqrt** (`double` **xr**, `double` **xi**, `double` ***yr**, `double` ***yi**);

Purpose

Computes the square root of a complex number. The resulting complex number is obtained using the following formula.

$(yr, yi) = (xr, xi)^{1/2}$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
Output	**yr**	double-precision	real part of y
	yi	double-precision	imaginary part of y

Return Value

status	integer	refer to error codes in Appendix A

CxSub

`int` **status** = **CxSub** (`double` **xr**, `double` **xi**, `double` **yr**, `double` **yi**, `double` ***zr**, `double` ***zi**);

Purpose

Subtracts two complex numbers. The resulting complex number is obtained using the following formulas.

$zr = xr - yr$

$zi = xi - yi$

Parameters

Input	**xr**	double-precision	real part of x
	xi	double-precision	imaginary part of x
	yr	double-precision	real part of y
	yi	double-precision	imaginary part of y
Output	**zr**	double-precision	real part of z
	zi	double-precision	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

CxSub1D

`int` **status** = **CxSub1D** (`double` **xr**`[]`, `double` **xi**`[]`, `double` **yr**`[]`, `double` **yi**`[]`, `int` **n**, `double` **zr**`[]`, `double` **zi**`[]`);

Purpose

Subtracts two 1D complex arrays. The i^{th} element of the resulting complex array is obtained using the formulas.

$$zr_i = xr_i - yr_i$$

$$zi_i = xi_i - yi_i$$

The operations can be performed in place; that is, the input and output complex arrays can be the same.

Parameters

Input	**xr**	double-precision array	real part of x
	xi	double-precision array	imaginary part of x
	yr	double-precision array	real part of y
	yi	double-precision array	imaginary part of y
	n	integer	number of elements
Output	**zr**	double-precision array	real part of z
	zi	double-precision array	imaginary part of z

Return Value

status	integer	refer to error codes in Appendix A

Decimate

`int` **status** = **Decimate** (`double` **x**`[]`, `int` **n**, `int` **dFact**, `int` **ave**, `double` **y**`[]`);

Purpose

Decimates the input sequence X by the decimating factor. If Y represents the decimated output sequence, the elements of the sequence Y are obtained using the following equation.

$$y_i = \begin{cases} x_{i*dFact} & ave = 0 \\ \dfrac{1}{dFact} \displaystyle\sum_{k=0}^{dFact-1} x_{i*dFact+k} & ave = 1 \end{cases}$$

where

i = 0, 1, 2 ... size-1

size = (int) ***(n/dFact****) and is the size of the output sequence*

Parameters

Input	**x**	double-precision array	contains the input array to be decimated.
	n	integer	number of elements in the input array.
	dFact	integer	amount by which to decimate **x** to form **y**.
	ave	integer	specifies whether averaging is used in decimating **x**.
Output	**y**	double-precision array	contains the output array, which is **x** decimated by the **dFact**. The size of this array must be (int) **n**/**dFact**.

Return Value

status	integer	refer to error codes in Appendix A

Deconvolve

`int` **status = Deconvolve** (`double` **y**`[]`, `int` **ny**, `double` **x**`[]`, `int` **nx**, `double` **h**`[]`);

Purpose

Computes the deconvolution of **y** with **x**. **y** is assumed to be the result of the convolution of **x** with some system response. The deconvolution operation is realized using Fourier transform pairs. The output sequence **h** is obtained using the following equation.

$$\boldsymbol{h} = InvFFT\{\ Y(f) / X(f)\ \}$$

where

X(f) is the Fourier transform of **x**
Y(f) is the Fourier transform of **y**
InvFFT() is the inverse Fourier transform

Parameters

Input	**y**	double-precision array	input array to be deconvolved with **x**.
	ny	integer	number of elements in **y**.
	x	double-precision array	input array with which **y** is deconvolved.
	nx	integer	The number of elements in **x**. **nx** ≤ **ny**.
Output	**h**	double-precision array	output array which is **y** deconvolved with **x** This array must be (**ny** - **nx** + 1) elements long.

Return Value

status	integer	refer to error codes in Appendix A

Determinant

`int` **status** = **Determinant** (`void` ***x**, `int` **n**, `double` ***det**);

Purpose

Finds the determinant of an **n** by **n** 2D input matrix.

Parameters

Input	**x**	double-precision 2D array	input matrix
	n	integer	dimension size of input matrix
Output	**det**	double-precision	determinant

Note: ***The input matrix must be an* n *by* n *square matrix.***

Return Value

status	integer	refer to error codes in Appendix A

Difference

`int` **status** = **Difference** (`double` **x**`[]`, `int` **n**, `double` **dt,** `double` **xInit**,
`double` **xFinal**,`double` **y**`[]`);

Purpose

Finds the discrete difference of the input array. The i^{th} element of the resulting array is obtained using the following formula.

$$y_i = [x_{i+1} - x_{i-1}] / (2 * dt)$$

where $\mathbf{x}_{-1}$ = **xInit** and $\mathbf{x}_n$ = **xFinal**.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	dt	double-precision	sampling interval
	xInit	double-precision	initial condition
	xFinal	double-precision	final condition
Output	**y**	double-precision array	differentiated array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/*Generate an array with random numbers and differentiate it.*/
double  x[200], y[200];
double  dt, xInit, xFinal;
int  n;
n = 200;
dt = 0.001;
xInit = -0.5;
xFinal = -0.25;
Uniform (n, 17, x);
Integrate (x, n, dt, xInit, xFinal, y);
```

Div1D

`int` **status** = **Div1D** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **z**`[]`);

Purpose

Divides two 1D arrays. The i[th] element of the output array is obtained using the following formula.

$$z_i = x_i / y_i$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x** **y** **n**	double-precision array double-precision array integer	**x** input array **y** input array number of elements to be divided
Output	**z**	double-precision array	result array

Return Value

status	integer	refer to error codes in Appendix A

Div2D

`int` **status** = **Div2D** (`void` *__x__, `void` *__y__, `int` **n**, `int` **m**, `void` *__z__);

Purpose

Divides two 2D arrays. The (i^{th}, j^{th}) element of the output array is obtained using the following formula.

$$z_{i,j} = x_{i,j} / y_{i,j}$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x** **y** **n** **m**	double-precision 2D array double-precision 2D array integer integer	**x** input array **y** input array number of elements in first dimension number of elements in second dimension
Output	**z**	double-precision 2D array	result array

Return Value

status	integer	refer to error codes in Appendix A

DotProduct

`int` **status** = **DotProduct** (`double` **`x[]`**, `double` **y**, `int` **n**, `double` *__dotProd__);

Purpose

Computes the dot product of the **x** and **y** input arrays. The dot product is obtained using the following formula.

$$dotProd = x \bullet y = \sum_{i=0}^{n-1} x_i * y_i$$

Parameters

Input	**x**	double-precision array	**x** input vector
	y	double-precision array	**y** input vector
	n	integer	number of elements
Output	**dotProd**	double-precision	dot product

Return Value

status	integer	refer to error codes in Appendix A

Elp_BPF

`int` **status** = **Elp_BPF** (`double` **`x[]`**, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**, `double` **ripple**, `double` **atten**, `int` **order**, `double` **`y[]`**);

Purpose

Filters the input array using a digital bandpass elliptic filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	ripple	double-precision	pass band ripples in dB
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandpass
elliptic filter.  The pass band is from 200.0 to 300.0 */
double  x[256], y[256], fs, fl, fh, ripple, atten;
int  n, order;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
ripple = 0.5;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
Elp_BPF (x, n, fs, fl, fh, ripple, atten, order, y);
```

Elp_BSF

`int` **status** = **Elp_BSF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**, `double` **ripple**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandstop elliptic filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	ripple	double-precision	pass band ripples in dB
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandstop
elliptic filter.  The stop band is from 200.0 to 300.0 */
double  x[256], y[256], fs, fl, fh, ripple, atten;
int  n, order;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
ripple = 0.5;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
Elp_BSF (x, n, fs, fl, fh, ripple, atten, order, y);
```

Elp_CascadeCoef

`int` **status** = **Elp_CascadeCoef** (`double` **fs**, `double` **fL**, `double` **fH**, `double` **ripple**, `double` **atten**, `IIRFilterPtr` **filterInformation**);

Purpose

Generates the set of cascade form filter coefficients to implement an IIR filter as specified by the Elliptic (or Cauer) filter model.

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function.

To redesign another filter, you should first call `FreeIIRFilterPtr` to free the present filter structure and then call `AllocIIRFilterPtr` with the new type and order parameters before calling this design function.

If the type and order remain the same, and you can call this IIR design function without calling `FreeIIRFilterPtr` and `AllocIIRFilterPtr`. In this case, you should properly reset the filtering operation for that structure by calling `ResetIIRFilter` before the first call to `IIRCascadeFiltering`.

Parameters

Input	**fs**	double-precision	Specifies the sampling frequency in Hz.
	fL	double-precision	Specifies the desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	Specifies the desired upper cutoff frequency of the filter in Hz
	ripple	double-precision	Specifies the amplitude of the **stopband** ripple in decibels.
	atten	double-precision	Specifies the **stopband** attenuation, in decibels, of the IIR filter to be designed.
Output	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/* Design a cascade lowpass Elliptic IIR filter */
double      fs, fl, fh, ripple, atten, x[256], y[256];
int      type,order,n;
```

```
IIRFilterPtr   filterInfo;
n = 256;
fs = 1000.0;
fl = 200.0;
ripple = 0.5;
atten = 40.0;
order = 5;
type = 0;      /* lowpass  */
Uniform(n,17,x);
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   Elp_CascadeCoef(fs,fl,fh,ripple,atten,filterInfo);
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

Elp_Coef

`int` **status** = **Elp_Coef** (`int` **type**, `int` **order**, `double` **fs**, `double` **fL**, `double` **fH**, `double` **ripple**, `double` **atten**, `double` **a**`[]`, `int` **na**, `double` **b**`[]`, `int` **nb**);

Purpose

Generates the set of filter coefficients to implement an IIR filter as specified by the Elliptic (or Cauer) filter model. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \textit{lowpass filter, } \textbf{fH} \textit{ is not used.} \\ 1 & \textit{highpass filter, } \textbf{fH} \textit{ is not used.} \\ 2 & \textit{bandpass filter} \\ 3 & \textit{bandstop filter} \end{cases}$$

a[**na**] and **b**[**nb**] are the reverse and forward filter coefficients. The actual filtering

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-i}\right)$$

is achieved by using the function `IIRFiltering`.

Parameters

Input	**type**	integer	controls the filter type of the Elliptic IIR filter coefficients.
	order	integer	order of the IIR filter.
	fs	double-precision	sampling frequency in Hz.
	fL	double-precision	desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	desired higher cutoff frequency of the filter in Hz.
	ripple	double-precision	amplitude of the stop band ripple in decibels.
	atten	double-precision	stop band attenuation, in decibels, of the IIR filter to be designed.
	na	integer	number of coefficients in the **a** coefficient array.
	nb	integer	number of coefficients in the **b** coefficient array.
Output	**a**	double-precision array	array containing the *reverse* coefficients of the designed IIR filter.
	b	double-precision array	array containing the *forward* coefficients of the designed IIR filter.

Return Value

status	integer	refer to error codes in Appendix A

Elp_HPF

`int` **status** = **Elp_HPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**,
`double` **ripple**,`double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital highpass elliptic filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	ripple	double-precision	pass band ripples in dB
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order highpass
elliptic filter. */
double  x[256], y[256], fs, fc, ripple, atten;
int  n, order;
n = 256;
fs = 1000.0;
fc = 200.0;
ripple = 0.5;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
Elp_HPF (x, n, fs, fc, ripple, atten, order, y);
```

Elp_LPF

`int` **status** = **Elp_LPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `double` **ripple**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital lowpass elliptic filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	ripple	double-precision	pass band ripples in dB
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order lowpass elliptic
filter. */
double  x[256], y[256], fs, fc, ripple, atten;
int  n, order;
n = 256;
fs = 1000.0;
fc = 200.0;
ripple = 0.5;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
Elp_LPF (x, n, fs, fc, ripple, atten, order, y);
```

Equi_Ripple

`int` **status** = **Equi_Ripple** (`int` **bands**, `double` **A**`[]`, `double` **wts**`[]`, `double` **fs**,
`double` **cutoffs**`[]`, `int` **type**, `int` **n**, `double` **coef**`[]`,
`double` ***delta**);

Purpose

Designs a multiband FIR linear phase filter, a differentiator, or a Hilbert Transform using the Parks-McClellan algorithm. The frequency response in each band has equal ripples that can be adjusted by a weighting factor. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**bands**	integer	number of bands of the filter
	A	double-precision array	desired frequency response magnitude of each band
	wts	double-precision array	weighting factor for each band
	fs	double-precision	sampling frequency
	cutoffs	double-precision array	end frequencies of each band
	type	integer	filter type
	n	integer	filter length
Output	**coef**	double-precision array	filter coefficients
	delta	double-precision	normalized ripple size

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

Generally, when **type** = 1 and **bands** ≥ 2, **Equi_Ripple** designs a multiband filter. When **type** = 2, **bands** = 1, and **n** is even, **Equi_Ripple** designs a differentiator. When **type** = 3, **bands** = 1, and **n** is even, **Equi_Ripple** designs a Hilbert Transform. For more information, please refer to *Digital Filter Design* by Parks and Burrus, or "A computer program for designing optimum FIR linear phase digital filters," by McClellan, *et al*, *IEEE Transactions on Audio and Electroacoustics*, vol. AU-21, no. 6, pp. 506-525, Nov. 1973.

Using This Function

Although **Equi_Ripple** is the most flexible way to design an FIR linear phase filter, it has more complex parameters and requires some DSP knowledge. You may find it more convenient to use `EquiRpl_LPF`, `EquiRpl_HPF`, `EquiRpl_BPF`, and `EquiRpl_BSF`. These functions, which provide lowpass, highpass, bandpass and bandstop FIR filters with equal weighting factors in all bands, are special cases of `Equi_Ripple` with simplified parameters.

For more information about windowing, see the section *About Windowing* in Chapter 1, *Advanced Analysis Library Overview*.

Example 1

```
/* Design a 24-point lowpass filter and filter the incoming signal. */
double  x[256], coef[24], y[280], fs, delta;
double  A[2];           /* array of frequency responses */
double  wts[2];         /* array of weighting factors */
double cutoffs[4];      /* frequency points */
```

```
int  n, m;
int bands;              /* number of bands */
int type;               /* filter type */

bands = 2;              /* one pass band and one stop band */
fs = 1000.0;            /* sampling frequency */
A[0] = 1.0;             /* 1 for the pass band */
A[1] = 0.0;             /* 0 for the stop band */
wts[0] = 1.0;           /* weighting factor for the pass band */
wts[1] = 1.0;           /* weighting factor for the stop band */
cutoffs[0] = 0.0;
cutoffs[1] = 300.0;     /* the first stop band [0, 300.0] */
cutoffs[2] = 400.0;
cutoffs[3] = 500.0;     /* the pass band [400, 500] */
type = 1;               /* multiple band filter */
n = 24;                 /* filter length */
m = 256;
Equi_Ripple (bands, A, wts, fs, cutoffs, type, n, coef, &delta);
Convolve (coef, n, x, m, y);/*convolve the filter with the signal */
```

Example 2

```
/* Design a 31-point bandpass filter and filter the incoming signal. */
double  x[256], coef[55], y[287], fs, delta;
double  A[3];           /* array of frequency responses */
double  wts[3];         /* array of weighting factors */
double cutoffs[6];      /* frequency points */
int  n, m;
int bands;              /* number of bands */
int type;               /* filter type */

bands = 3;              /* one pass band and two stop bands */
fs = 1000.0;            /* sampling frequency */
A[0] = 0.0;             /* 0 for the first stop band */
A[1] = 1.0;             /* 1 for the stop band */
A[2] = 0.0;             /* 0 for second stop band */
wts[0] = 10.0;             /* weighting factor for the first stop band */
wts[1] = 1.0;           /* weighting factor for the pass band */
wts[2] = 4.0;           /* weighting factor for the second stop band */
cutoffs[0] = 0.0;
cutoffs[1] = 200.0;     /* the first stop band [0, 200.0] */
cutoffs[2] = 250.0;
cutoffs[3] = 350.0;     /* the pass band [250, 350] */
cutoffs[4] = 400.0;
cutoffs[5] = 500.0;     /* the second stop band */
type = 1;               /* multiple band filter */
n = 31;                 /* filter length */
m = 256;
Equi_Ripple (bands, A, wts, fs, cutoffs, type, n, coef, &delta);
Convolve (coef, n, x, m, y);/*convolve the filter with the signal */
```

Example 3

```
/* Design a 30-point differentiator. */
double  coef[30], fs, delta;
double  A[1];           /* array of frequency responses */
double  wts[1];         /* array of weighting factors */
double cutoffs[2];      /* frequency points */
int  n;
int bands;              /* number of bands */
int type;               /* filter type */

bands = 1;              /* one pass band and one stop band */
fs = 1000.0;            /* sampling frequency */
A[0] = 1.0;             /* 1 for the band */
wts[0] = 1.0;           /* weighting factor for the band */
cutoffs[0] = 0.0;
cutoffs[1] = 500.0;     /* the entire frequency range */
type = 2;               /* differentiator */
n = 30;                 /* filter length */
Equi_Ripple (bands, A, wts, fs, cutoffs, type, n, coef, &delta);
```

Example 4

```
/* Design a 20-point Hilbert transform. */
double  coef[20], fs, delta;
double  A[1];           /* array of frequency responses */
double  wts[1];         /* array of weighting factors */
double cutoffs[2];      /* frequency points */
int  n;
int bands;              /* number of bands */
int type;               /* filter type */

bands = 1;              /* one pass band and one stop band */
fs = 1000.0;            /* sampling frequency */
A[0] = 1.0;             /* 1 for the band */
wts[0] = 1.0;           /* weighting factor for the band */
cutoffs[0] = 100.0;
cutoffs[1] = 500.0;
type = 3;               /* Hilbert transform */
n = 20;                 /* filter length */
Equi_Ripple (bands, A, wts, fs, cutoffs, type, n, coef, &delta);
```

EquiRpl_BPF

`int` **status** = **EquiRpl_BPF** (`double` **fs**, `double` **f1**, `double` **f2**, `double` **f3**, `double` **f4**, `int` **n**, `double` **coef**`[]`, `double` ***delta**);

Purpose

Designs a bandpass FIR linear phase filter using the Parks-McClellan algorithm. The function is a special case of the general Parks-McClellan algorithm. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	f1	double-precision	cutoff frequency 1
	f2	double-precision	cutoff frequency 2
	f3	double-precision	cutoff frequency 3
	f4	double-precision	cutoff frequency 4
	n	integer	filter length
Output	**coef**	double-precision array	filter coefficients
	delta	double-precision	normalized ripple size

Parameter Discussion

There are two stop bands and one pass band. The first stop band is [0, **f1**] and the second stop band is [**f4**, **fs**/2]. The pass band is [**f2**, **f3**]. **f1**, **f2**, **f3**, and **f4** must be in ascending order. Refer to the `Equi_Ripple` function description for more information.

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Design a 51-point bandpass filter and filter the incoming signal. */
double  x[256], coef[25], y[301], fs, f1, f2, f3, f4, delta;
int  n, m;
fs = 1000.0;          /* sampling frequency */
f1 = 200.0;           /* the first stop band [0, 200] */
f2 = 250.0;
f3 = 350.0;           /* the pass band [250, 350] */
f4 = 400.0;           /* the second stop band [400, 500] */
n = 51;               /* filter length */
m = 256;
EquiRpl_BPF (fs, f1, f2, f3, f4, n, coef, &delta);
Convolve (coef, n, x, m, y);/*convolve the filter with the signal */
```

EquiRpl_BSF

`int` **status** = **EquiRpl_BSF** (`double` **fs**, `double` **f1**, `double` **f2**, `double` **f3**, `double` **f4**, `int` **n**, `double` **coef**`[]`, `double` ***delta**);

Purpose

Designs a bandstop FIR linear phase filter using the Parks-McClellan algorithm. The function is a special case of the general Parks-McClellan algorithm. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	f1	double-precision	cutoff frequency 1
	f2	double-precision	cutoff frequency 2
	f3	double-precision	cutoff frequency 3
	f4	double-precision	cutoff frequency 4
	n	integer	filter length
Output	**coef**	double-precision array	filter coefficients
	delta	double-precision	normalized ripple size

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

There are two pass bands and one stop band. The first pass band is [0, **f1**] and the second pass band is [**f4**, **fs**/2]. The stop band is [**f2**, **f3**]. **f1**, **f2**, **f3**, and **f4** must be in ascending order. Refer to the `Equi_Ripple` function description for more information.

Example

```
/* Design a 51-point bandstop filter and filter the incoming signal. */
double  x[256], coef[25], y[301], fs, f1, f2, f3, f4, delta;
int  n, m;
fs = 1000.0;            /* sampling frequency */
f1 = 200.0;             /* the first pass band [0, 200] */
f2 = 250.0;
f3 = 350.0;             /* the stop band [250, 350] */
f4 = 400.0;             /* the second pass band [400, 500] */
n = 51;                 /* filter length */
m = 256;
EquiRpl_BSF (fs, f1, f2, f3, f4, n, coef, &delta);
Convolve (coef, n, x, m, y);  /* convolve the filter with the signal */
```

EquiRpl_HPF

`int` **status = EquiRpl_HPF** (`double` **fs**, `double` **f1**, `double` **f2**, `int` **n**, `double` **coef** `[]`, `double` ***delta**);

Purpose

Designs a highpass FIR linear phase filter using the Parks-McClellan algorithm. The function is a special case of the general Parks-McClellan algorithm. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	f1	double-precision	cutoff frequency 1
	f2	double-precision	cutoff frequency 2
	n	integer	filter length
Output	**coef**	double-precision array	filter coefficients
	delta	double-precision	normalized ripple size

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

There is one stop band and one pass band. The stop band is [0, **f1**] and the pass band is [**f2**, **fs**/2]. Refer to the `Equi_Ripple` function description for more information.

Example

```
/* Design a 25-point highpass filter and filter the incoming signal. */
double  x[256], coef[25], y[281], fs, f1, f2, delta;
int  n, m;
fs = 1000.0;            /* sampling frequency */
f1 = 300.0;             /* the stop band [0, 300] */
f2 = 400.0;             /* the pass band [400, 500] */
n = 25;                 /* filter length */
m = 256;
EquiRpl_HPF (fs, f1, f2, n, coef, &delta);
Convolve (coef, n, x, m, y);  /* convolve the filter with the signal */
```

EquiRpl_LPF

`int` **status** = **EquiRpl_LPF** (`double` **fs**, `double` **f1**, `double` **f2**, `int` **n**, `double` **coef**`[]`, `double` ***delta**);

Purpose

Designs a lowpass FIR linear phase filter using the Parks-McClellan algorithm. The function is a special case of the general Parks-McClellan algorithm. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	f1	double-precision	cutoff frequency 1
	f2	double-precision	cutoff frequency 2
	n	integer	filter length
Output	**coef**	double-precision array	filter coefficients
	delta	double-precision	normalized ripple size

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

There is one pass band and one stop band. The pass band is [0, **f1**] and the stop band is [**f2**, **fs**/2]. Refer to the `Equi_Ripple` function description for more information.

Example

```
/* Design a 25-point lowpass filter and filter the incoming signal. */
double  x[256], coef[25], y[281], fs, f1, f2, delta;
int  n, m;
fs = 1000.0;            /* sampling frequency */
f1 = 300.0;             /* the pass band [0, 300] */
f2 = 400.0;             /* the stop band [400, 500] */
n = 25;                 /* filter length */
m = 256;
EquiRpl_LPF (fs, f1, f2, n, coef, &delta);
Convolve (coef, n, x, m, y);/* convolve the filter with the signal*/
```

ExBkmanWin

`int` **status = ExBkmanWin** (`double` **x[]**, `int` **n**);

Purpose

Applies an exact Blackman window to the input sequence X. If Y represents the output sequence, the elements of Y are obtained using the following equation.

$$Y_i = X_i\,(a_0 - a_1 * \cos(2\pi i/n) + a_2 * \cos(4\pi i/n)),\ i=0, \ldots, n-1$$

where

$a_0 = 7938.0/18608.0$
$a_1 = 9240.0/18608.0$
$a_2 = 1430.0/18608.0$

Parameters

Input	**x**	double-precision array	contains the input signal.
	n	integer	number of elements in the input array.
Output	**x**	double-precision array	contains the signal after applying the exact Blackman window.

Return Value

status	integer	refer to error codes in Appendix A

ExpFit

`int` **status = ExpFit** (`double` **x[]**, `double` **y[]**, `int` **n**, `double` **z[]**, `double` ***a**, `double` ***b**, `double` ***mse**);

Purpose

Finds the coefficient values that best represent the exponential fit of the data points (**x**, **y**) using the least squares method. The i^{th} element of the output array is obtained by using the following formula..

$$z_i = a * e^{b * x_i}$$

The mean squared error (**mse**) is obtained using the following formula.

$$mse = \sum_{i=0}^{n-1} |z_i - y_i|^2 / n$$

where **n** is the number of sample points.

Parameters

Input	**x**	double-precision array	**x** values
	y	double-precision array	**y** values
	n	integer	number of sample points
Output	**z**	double-precision array	best exponential fit
	a	double-precision	amplitude
	b	double-precision	exponential constant
	mse	double-precision	mean squared error

Note: ***The* y *values must be all positive or all negative to perform an exponential fit.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an exponential pattern and find the best exponential fit. */
double  x[200], y[200], z[200];
double  first, last, a, b, amp, decay, mse;
int  n;

n = 200;
first = 0.0;
last = 1.99E2;
Ramp (n, first, last, x);     /*  x[i] = i */

a = 3.5;
b = -2.75;
for (i=0; i<n; i++)
   y[i] = a * exp(b*x[i]);
/* Find the best exponential fit in z.*/
ExpFit (x, y, n, z, &amp, &decay, &mse);
```

ExpWin

`int` **status** = **ExpWin** (`double` **x**`[]`, `int` **n**, `double` **final**);

Purpose

Applies an exponential window to the input sequence X. If Y represents the output sequence, the elements of Y are obtained with the following formula.:

$$Y_i = X_i e^{ai}$$

where
a = ln(f)/(n-1)
f is the final value
n is the number of elements in X.

Parameters

Input	**x**	double-precision array	on input, **x**[n] contains the input signal.
	n	integer	number of elements in the input array.
	final	double-precision	final value of the exponential window function.
Output	**x**	double-precision array	on output, **x**[n] contains the signal after applying the exponential window.

Return Value

status	integer	refer to error codes in Appendix A

F_Dist

`int` **status** = **F_Dist** (`double` **f**, `int` **n**, `int` **m**, `double` ***p**);

Purpose

Calculates the one-sided probability **p**:

$$p = prob(F \leq f)$$

where F is a random variable from the F-distribution with **n** and **m** degrees of freedom.

Parameters

Input	**f**	double-precision	$-\infty < \mathbf{f} < \infty$
	n	integer	degrees of freedom
	m	integer	degrees of freedom
Output	**p**	double-precision	probability ($0 \le p < 1$)

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double x, p;
int n, m;
x = -123.456;
n = 6;
m = 7;
F_Dist (x, n, m, &p);
/* Now p = 0 because F distributed variables are non-negative. */
```

FFT

`int` **status** = **FFT** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**);

Purpose

Computes the Fast Fourier Transform of the complex data. Let X = x + jy be the complex array, then:

$$Y = FFT\{X\}$$

The operation is done in place and the input arrays **x** and **y** are overwritten. See the *About the Fast Fourier Transform (FFT)* section in Chapter 1.

Parameters

Input	**x**	double-precision array	real part of complex array
	y	double-precision array	imaginary part of complex array
	n	integer	number of elements
Output	**x**	double-precision array	real part of FFT
	y	double-precision array	imaginary part of FFT

Note: ***The number of elements (*n*) must be a power of two.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate two arrays with random numbers and compute its
   Fast Fourier Transform. */
double  x[256], y[256];
int  n;
n = 256;
Uniform (n, 17, x);
Uniform (n, 17, y);
FFT (x, y, n);
```

FHT

`int` **status** = **FHT** (`double` **x**`[]`, `int` **n**);

Purpose

Computes the Fast Hartley Transform using the following formula.

$$X_k = \sum_{i=0}^{n-1} x_i cas(2\pi ik / n)$$

where X_k is the k^{th} point of the FHT, and cas (k) = cos (k) + sin (k).

The operation is done in place and the **x** input array is overwritten.

Parameters

Input	**x** **n**	double-precision array integer	array to be transformed number of elements
Output	**x**	double-precision array	Hartley Transform

Note: ***The number of elements (*****n*****) must be a power of two.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers and compute its */
/* Fast Hartley Transform. */
double  x[256];
int  n;
n = 256;
Uniform (n, 17, x);
FHT (x, n);
```

FIR_Coef

`int` **status** = **FIR_Coef** (`int` **type**, `double` **fs**, `double` **fL**, `double` **fH**, `int` **taps**, `double` **coef**`[]`);

Purpose

Generates a set of FIR filter coefficients based on the window design method. This function returns the coefficients as the truncated impulse response of an ideal frequency response of the selected filter type. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \text{lowpass filter, } \textbf{fH} \text{ is not used.} \\ 1 & \text{highpass filter, } \textbf{fH} \text{ is not used.} \\ 2 & \text{bandpass filter} \\ 3 & \text{bandstop filter} \end{cases}$$

The actual filtering

$$y_n = \sum_{i=0}^{taps-1} Coef_i \cdot x_{n-1}$$

is achieved by using the convolution function `Convolve`.

Parameters

Input	**type**	integer	controls the filter type of the FIR filter coefficients to be designed.
	fs	double-precision	sampling frequency in hertz.
	fL	double-precision	desired lower cutoff frequency in hertz.
	fH	double-precision	desired upper cutoff frequency in hertz.
	taps	integer	desired length of the FIR filter.
Output	**coef**	double-precision array	computed output window FIR filter coefficients.

Return Value

status	integer	refer to error codes in Appendix A

FlatTopWin

`int` **status** = **FlatTopWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a flat top window to the input sequence x. If y represents the output sequence, the elements of y are obtained using the following equation.

$$y_i = x_i\,(0.2810639 - 0.5208972\cos(2\pi i/n) + 0.1980399\cos(4\pi i/n))$$

where **n** is the number of elements in x.

Parameters

Input	**x**	double-precision array	on input, **x**[n] contains the input signal.
	n	integer	number of elements in the input array.
Output	**x**	double-precision array	on output, **x**[n] contains the signal after applying the flat top window.

Return Value

status	integer	refer to error codes in Appendix A

ForceWin

`int` **status = ForceWin** (`double` **x**`[]`, `int` **n**, `double` **duty**);

Purpose

Applies a force window to the input sequence x:

$$x_i = \begin{cases} x_i & 0 \le i \le int[(duty/100)*n] \\ 0 & elsewhere \end{cases}$$

Parameters

Input	**x**	double-precision array	on input, **x**[n] contains the input signal.
	n	integer	number of elements in the input array.
	duty	double-precision	duty cycle, in percent, of the force window.
Output	**x**	double-precision array	on output, **x**[n] contains the signal after applying the force window.

Return Value

status	integer	refer to error codes in Appendix A

ForwSub

`int` **status = ForwSub** (`void` ***a**, `double` **y**`[]`, `int` **n**, `double` **x**`[]`**,** `int` **p**`[]`);

Purpose

Solves the linear equations a*x = y by forward substitution. **a** is assumed to be an **n** by **n** lower triangular matrix whose diagonal elements are all ones. **x** is obtained by the following formulas.

$$x_0 = y_0$$

$$x_i = y_i \sum_{j=0}^{i-1} a_{i,j} * x_j \qquad for\ i = 1, 2, \ldots n\text{-}1$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**a**	double-precision 2D array	input matrix
	y	double-precision array	input vector
	n	integer	dimension size of **a**
	p	integer array	permutation vector
Output	**x**	double-precision array	solution vector

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

`ForwSub` is used in conjunction with `LU` and `BackSub` to solve linear equations. The parameter **p** is obtained from `LU`. If you are not using the `LU` function, set **p**[**i**] = **i**.

Refer to the `LU` function description for more information.

Example

```
/* To solve a linear equation A*x = y */
double  A[10][10], x[10], y[10];
int p[10];                 /* permutation vector */
int sign, n;

n = 10;
LU (A, n, p, &sign);       /* LU decomposition of A */
ForwSub (A, y, n, x, p);   /* forward substitution */
BackSub (A, x, n, x);      /* backward substitution */
```

FreeIIRFilterPtr

`int` **status** = **FreeIIRFilterPtr** (`IIRFilterPtr` **filterInformation**);

Purpose

Frees the IIR cascade filter structure and all internal arrays.

Parameters

Input	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

GaussNoise

`int` **status** = **GaussNoise** (`int` **n**, `double` **sDev**, `int` **seed**, `double` **noise**`[]`);

Purpose

Generates an array of random Gaussian numbers distributed with expected zero mean value, and specified standard deviation.

Parameters

Input	**n**	integer	number of samples
	sDev	double-precision	desired standard deviation
	seed	integer	initial seed value
Output	**noise**	double-precision array	Gaussian noise pattern

Return Value

status	integer	refer to error codes in Appendix A

Using This Function

The expected standard deviation of the returned pattern is the one specified by the user. The expected mean value is zero; that is, the noise array values are expected to be centered about zero. When seed ≥ 0, a new random sequence is generated using the seed value. When seed < 0, the previously generated random sequence continues.

Example

```
/* The following code generates an array of random Gaussian distributed
numbers. */
double  x[20], sDev;
int  n;
n = 20;
sDev = 5.0;
GaussNoise (n, sDev, 17, x);
```

GenCosWin

`int` **status** = **GenCosWin** (`double` **x**`[]`, `int` **n**, `double` **a**`[]`, `int` **na**);

Purpose

Applies a general cosine window to the input sequence x. If y represents the output sequence, the elements of y are obtained using the following formula.

$$y_i = x_i \sum_{k=0}^{na-1} (-1)^k a_k \cos(2\pi ki/n)$$

where

- **a** is the array of coefficients
- **na** is the number of coefficients
- **n** is the number of elements in **x**

Parameters

Input	**x**	double-precision array	on input, **x**[n] contains the input signal.
	n	integer	number of elements in the input array.
	a	double-precision array	general cosine coefficient array.
	na	integer	number of elements in the **a**.
Output	**x**	double-precision array	on output, **x**[n] contains the signal after applying the general Cosine Window.

Return Value

status	integer	refer to error codes in Appendix A

GenLSFit

`int` **status** = **GenLSFit** (`void *`**H**, `int` **n**, `int` **k**, `double` **y**`[]`, `double` **stdDev**`[]`,
`int` **algorithm**, `double` **z**`[]`, `double` **b**`[]`, `double` **covar**`[]`,
`double *`**mse**);

Purpose

Finds the Best Fit k-dimensional plane and the set of linear coefficients using the least chi-squares method for observation data sets,

$$\{x_{i0}, x_{i1}, \ldots, x_{ik-1}, x_i\}$$

where i = 0, 1,..., n - 1, and
n = the number of your observation data sets.

Parameters

Input	**H**	2D double-precision array	An n-by-k matrix, which contains the observation data $\{x_{i0}, x_{i1}, \ldots, x_{ik-1}\}$ i = 0,1,...,*n*-1, where n is the number of rows in **H**, *k* is the number of columns in **H**.
	n	integer	Number of rows of **H** as well as the number of elements in **y**.
	k	integer	Number of columns of **H** as well as the number of elements in **b**.
	y	1D double-precision array	Number of elements in y should be equal to the number of rows in **H**.
	stdDev	1D double-precision array	Standard deviation σ_i for data point (x_i, y_i). If they are equal or if you do not know, pass an empty array, and the function will ignore this parameter. The size of this array should be equal to n.
	algorithm	integer	Algorithm to be used in solving the multiple linear regression model. The algorithm has six selections: 0: SVD 1: Givens 2: Givens2 3: Householder 4: LU decomposition 5: Cholesky algorithm

(continues)

Parameters (Continued)

Output	**z**	1D double-precision array	Fitted data computed by using the coefficients **b**.
	b	1D double-precision array	Set of coefficients that minimize χ^2, which is defined in equation (2-2).
	covar	2D double-precision array	Matrix of covariances C with *k*-by-*k* elements. c_{ik} is the covariance between b_i and b_k and c_{ii} is the variance of b_i. If you pass an empty array for **covar**, the function will not compute this matrix.
	mse	double-precision	Mean squared error.

Using This Function

You can use `GenLSFit` to solve multiple linear regression problems. You can also use it to solve for the linear coefficients in a multiple-function equation.

The general least squares linear fit problem can be described as follows. Given a set of observation data, find a set of coefficients that fit the linear "model."

$$y_i = b_0 x_{i0} + \ldots + b_{k-1} x_{ik-1}$$

$$= \sum_{j=0}^{k-1} b_j x_{ij} \qquad i = 0,1,\ldots,\ n - 1 \qquad (2\text{-}1)$$

where **b** is the set of coefficients,
n is the number of elements in **y** and the number of rows of **H**, and
k is the number of elements in **b**.
x_{ij} is your observation data, which is contained in **H**.

$$H = \begin{bmatrix} x_{00} & x_{01} & \cdots & x_{0k-1} \\ x_{10} & x_{11} & & x_{1k-1} \\ \vdots & & & \\ x_{n-10} & x_{n-12} & \cdots & x_{n-1k-1} \end{bmatrix}$$

Equation (2-1) can also be written as Y = HB.
This is a multiple linear regression model, which uses several variables

$x_{i0}, x_{i1}, ..., x_{ik-1}$,

to predict one variable y_i. In contrast, the `LinFit`, `ExpFit`, and `PolyFit` functions are all based on a single predictor variable, which uses one variable to predict another variable.
In most cases, we have more observation data than coefficients. The equations in (2-1) may not produce the solution. The fit problem becomes to find the coefficients B that minimizes the difference between the observed data, y_i and the predicted value,

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij} \ .$$

This function uses the least chi-squares plane method to obtain the coefficients in (2-1), that is, finding the solution, B, which minimizes the following quantity.

$$\chi^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - \sum_{i=0}^{k-1} b_j x_{ij}}{\sigma_i}\right)^2 = \left|H_0 B - Y_0\right|^2 \qquad (2\text{-}2)$$

where $h_{0ij} = \frac{x_{ij}}{\sigma_i}, y_{0i} = \frac{y_i}{\sigma_i}, i = 0,1,\ldots,n-1; j = 0,1,\ldots,k-1.$

In equation (2-2) , σ_i is the standard deviation, **StdDev**. If the measurement errors are independent and normally distributed with constant standard deviation $\sigma_i = \sigma$, the preceding equation is also the least squares estimation.

There are different ways to minimize χ^2. One way to minimize χ^2 is to set the partial derivatives of χ^2 to zero with respect to $b_0, b_1,..., b_{k-1}$.

$$\left(\begin{array}{l} \frac{\partial\chi^2}{\partial b_0} = 0 \\ \frac{\partial\chi^2}{\partial b_1} = 0 \\ \quad . \\ \quad . \\ \quad . \\ \frac{\partial\chi^2}{\partial b_{k-1}} = 0 \end{array}\right.$$

The preceding equations can be derived to

$$H_0^T H_0 B = H_0^T Y \ . \qquad (2\text{-}3)$$

H_0^T is the transposition of H_0.

Equation (2-3) and the one preceding it are also called normal equations of the least squares problems. You can solve them using LU or Cholesky factorization algorithms, but the solution from the normal equations is susceptible to round-off error.

An alternative, and preferred way to minimize χ^2 is to find the least squares solution of equations

$$H_0 B = Y_{0.}$$

You can use QR or SVD factorization to find the solution, B. For QR factorization, you can choose Householder, Givens, and Givens2 (also called fast Givens).

Different algorithms can give you different precision, and in some cases, if one algorithm cannot solve the equation, perhaps another algorithm can. You can try different algorithms to find the one best suited to your data.

The covariance matrix **covar** is computed as follows.

$$\text{covar} = \left(H_0^T H_0\right)^{-1}$$

The best fitted curve **z** is given by the following formula.

$$z_i = \sum_{j=0}^{k-1} b_j x_{ij}$$

The **mse** is obtained using the following formula.

$$mse = \frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2$$

The polynomial fit that has a single predictor variable can be thought of as a special case of multiple regression. If the observation data sets are $\{x_i, y_i\}$ where i = 0, 1, ..., n-1, the model for polynomial fit is as follows.

$$y_i = \sum_{j=0}^{k-i} b_j x_i^j = b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_{k-1} x_i^{k-1} \qquad (2\text{-}4)$$

where $i = 0, 1, 2, \ldots, n - 1$.

Comparing equations (2-1) and (2-4) shows that $x_{ij} = x_i^j$. In other words,

$$x_{i0} = x_i^0, x_{i1} = x_i, x_{i2} = x_i^2, \ldots x_{ik-1} = x_i^{k-1}.$$

In this case, you can build H as follows:

$$H = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{k-1} \\ 1 & x_1 & x_1^2 & & x_1^{k-1} \\ \vdots & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & & x_{n-1}^{k-1} \end{bmatrix}$$

Instead of using $x_{ij} = x_j^i$, you can also choose another function formula to fit the data sets $\{x_i, y_i\}$. In general, you can select $x_{ij} = f_j(x_i)$. Here, $f_j(x_i)$ is the function model that you choose to fit your observation data. In polynomial fit, $f_j(x_i) = x_i^j$.
In general, you can build *H* as follows:

$$H = \begin{bmatrix} f_0(x_0) & f_1(x_0) & f_2(x_0) & \cdots & f_{k-1}(x_0) \\ f_0(x_1) & f_1(x_1) & f_2(x_1) & \cdots & f_{k-1}(x_1) \\ \vdots & & & & \\ f_0(x_{n-1}) & f_1(x_{n-1}) & f_2(x_{n-1}) & \cdots & f_{k-1}(x_{n-1}) \end{bmatrix}$$

Your fit model is:

$$y_i = b_0 f_0(x) + b_1 f_1(x) + \ldots + b_{k-1} f_{k-1}(x).$$

The following two examples show how to use this function. The first example uses the `GenLSFit` function to perform multiple regression analysis based entirely on tabulated observation data. The second solves for the linear coefficients in a multiple-function equation.

Example: Predicting Cost

Suppose you want to estimate the total cost (in dollars) of a production of baked scones; using the quantity produced, X1, and the price of one pound of flour, X2. To keep things simple, the following five data points form this sample data table.

Cost (dollars) **Y**	**Quantity** **X1**	**Flour Price** **X2**
$150	295	$3.00
$75	100	$3.20
$120	200	$3.10
$300	700	$2.80
$50	60	$2.50

You want to estimate the coefficients to the following equation.

$$Y = b_0 + b_1\text{X1} + b_2\text{X2}$$

The only parameters that you need to build are **H** (observation matrix) and y arrays. Each column of **H** is the observed data for each independent variable: the first column is one because the coefficient b_0 is not associated with any independent variable.
H should be filled in as:

$$H = \begin{bmatrix} 1 & 295 & 3 \\ 1 & 100 & 3.20 \\ 1 & 200 & 3.10 \\ 1 & 700 & 280 \\ 1 & 60 & 250 \end{bmatrix}$$

The following code is based on this example.

```
/*The example of predicting cost using GenLSFit */
int k, n, algorithm, status;
double H[5][3], y[5], z[5], b[3], X1[5],X2[5], mse;
double *stdDev=0, *covar=0;   /* define empty arrays, the function will ignore
these parameters. */
n = 5;
k = 3;
/* Read in data for X1,X2 and y  */
   .
   .
   .
/* Construct matrix H   */
for(i=0;i<n;i++) {
   H[i][0] = 1;   /* fill in the first column of H. */
   H[i][1] = X1[i];  /* fill in the second column of H. */
   H[i][2] = X2[i];  /* fill in the third column of H. */
}
algorithm = 0;    /* use SVD algorithm */
status = GenLSFit(H,n,k,y,stdDev,algorithm,z,b,covar,&mse);
```

Example: Linear Combinations

Suppose that you have collected samples from a transducer (Y Values) and you want to solve for the coefficients of the model.

$$y = b_0 + b_1 \sin(\omega x) + b_2 \cos(\omega x) + b_3 x^3$$

To build *H*, you set each column to the independent functions evaluated at each x value. Assuming there are 100 x values, *H* would be the following array.

$$H = \begin{bmatrix} 1 & \sin(\omega x_0) & \cos(\omega x_0) & x_0 \\ 1 & \sin(\omega x_1) & \cos(\omega x_1) & x_1^2 \\ 1 & \sin(\omega x_2) & \cos(\omega x_2) & x_2^2 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & \sin(\omega x_{99}) & \cos(\omega x_{99}) & x_{99}^2 \end{bmatrix}$$

The following code is based on this example.

```
/*The example of linear combinations using GenLSFit   */
int i, k, n, algorithm, status;
double H[100][4], y[100], z[100], b[4],x[100],mse, w;
double *stdDev=0, *covar=0;   /* define empty arrays, the function will ignore
these parameters. */
n = 100;
k = 4;
w = 0.2;
/* Read in data for x and y  */
   .
   .
   .
/* Construct matrix H   */
for(i=0;i<n;i++) {
   H[i][0] = 1;      /* fill in the first column of H. */
   H[i][1] = sin(w*x[i]);  /* fill in the second column of H. */
   H[i][2] = cos(w*x[i]);  /* fill in the third column of H. */
   H[i][3] = pow(x[i],3);  /* fill in the fourth column of H. */
}
algorithm = 0;    /* use SVD algorithm */
status = GenLSFit(H,n,k,y,stdDev,algorithm,z,b,covar,&mse);
```

GenLSFitCoef

`int` **status = GenLSFitCoef** (`void` ***H**, `int` **n**, `int` **k**, `double` **y**`[]`, `double` **b**`[]`,
`int` **algorithm**);

Purpose

Finds the k-dimension linear curve values and the set of k-dimension linear fit coefficients, which describe the k-dimension linear curve that best represents the input data set using the least-squares solution. The general form of the k-dimension linear fit is as follows.

Let i = 0, 1, ..., n be your i^{th} observation and $x_{ij}, \ldots, x_{ik-1}$ be k-1 observed x points and y_i be observed y point, the **H** matrix is composed by

$$H_{nxk} = \begin{bmatrix} 1 & x_{01} & x_{02} & \ldots & x_{0k-1} \\ 1 & x_{11} & x_{12} & \ldots & x_{1k-1} \\ \cdot \\ \cdot \\ \cdot \\ 1 & x_{n-1,1} & x_{n-1,2} & \ldots & x_{n-1,k-1} \end{bmatrix}$$

The general LS linear fit coefficient $\mathbf{b}_k$ is obtained by minimizing the quantity

$$Q = \sum_{i=0}^{n-1} (y_i - z_i)^2 = \sum_{i=0}^{n-1} (y_i - b_0 - \sum_{j=1}^{k-1} b_j x_{ij})^2$$

The algorithm has the following valid selection.

0: using the singular value decomposition (defaults)
1: using the Givens decomposition
2: using the square root free Givens decomposition
3: using the Household transformation
4: using the LU decomposition
5: using the Cholesky decomposition

Each algorithm may offer different precision depending on the input data. Given the coefficient vector **b**[k] and **H**, the fitted data z_i can be computed by a simple matrix multiplication

$$Z = \boldsymbol{H} \bullet \boldsymbol{b}$$

and the mean squared error can be computed by

$$mse = \frac{1}{n} \sum_{i=0}^{n-1} (z_i - y_i)^2$$

Parameters

Input	**H**	double-precision 2D array	input matrix which represents the formula you use to fit the data set {X,Y}. **H**[i][j] are the function values of X[i].
	n	integer	number of rows used in **H**, as well as the number of elements in **y**.
	k	integer	number of columns used in **H**, as well as the number of elements in **b**.
	y	double-precision array	array containing the y coordinates of the (x,y) data sets to be fitted.
	algorithm	integer	algorithm to be used in solving the multiple linear regression model.
Output	**b**	double-precision array	contains the set of linear coefficients that best fit the multiple linear regression model in a least squares sense. The size of this array must be at least **k**.

Return Value

status	integer	refer to error codes in Appendix A

GetAnalysisErrorString

`char` ***message** = **GetAnalysisErrorString** (`int` **errorNum**)

Purpose

Converts the error number returned by an Analysis Library function into a meaningful error message.

Parameters

Input	**errorNum**	integer	status returned by Analysis function.

Return Value

message	string	explanation of Error

HamWin

`int` **status = HamWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a Hamming window to the **x** input signal. The Hamming window is defined by the formula.

$$w_i = 0.54 - 0.46 * \cos(2\pi\, i/n) \quad \text{for } i = 0, 1, ..., n-1$$

The output signal is obtained by the following formula.

$$x_i = x_i * w_i \qquad \text{for } i = 0, 1, ..., n-1$$

The window operation is performed in place. The windowed data **x** replaces the input data **x**.

Parameters

Input	**x** **n**	double-precision array integer	input data number of elements in **x**
Output	**x**	double-precision array	windowed data

Return Value

status	integer	refer to error codes in Appendix A

HanWin

`int` **status = HanWin** (`double` **x** `[]`, `int` **n**);

Purpose

Applies a Hanning window to the **x** input signal. The Hanning window is defined by the following formula.

$$w_i = 0.5 - 0.5 * \cos(2\pi i/n) \quad \text{for } i = 0, 1, ..., n-1$$

The output signal is obtained by the following formula.

$$x_i = x_i * w_i \qquad \text{for } i = 0, 1, ..., n-1$$

The window operation is performed in place. The windowed data **x** replaces the input data **x**.

Parameters

Input	**x** **n**	double-precision array integer	input data number of elements in **x**
Output	**x**	double-precision array	windowed data

Return Value

status	integer	refer to error codes in Appendix A

Histogram

`int` **status** = **Histogram** (`double` **x**`[]`, `int` **n**, `double` **xBase**, `double` **xTop**, `int` **hist**`[]`, `double` **axis**`[]`**,** `int` **intervals**);

Purpose

Computes the histogram of the **x** input array. The histogram is obtained by counting the number of times that the elements in the input array fall in the i^{th} interval. Let

$$\Delta x = (xTop - xBase) / intervals$$

$$y_i(x) = \begin{cases} 1 & \text{if } i\Delta x \le x - xBase < (i+1)\Delta x \\ 0 & \text{otherwise} \end{cases}$$

The i^{th} element of the histogram is:

$$hist_i = \sum_{j=0}^{n-1} y_i(x_j)$$

The values of the histogram axis are the mid-point values of the intervals.

$$axis_i = i\Delta x + \Delta x/2 + xBase$$

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	xBase	double-precision	lower range
	xTop	double-precision	upper range
	intervals	integer	number of intervals
Output	**hist**	integer array	histogram of **x**
	axis	double-precision array	histogram axis array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/*Generate a Gaussian distributed random array and find its histogram.*/
double  x[2000], axis[50], max, min;
int  hist[50], n, intervals, imax, imin;
n = 2000;
intervals = 50;
GaussNoise (n, 1.0E0, 17, x);
MaxMin (x, n, &max, &imax, &min, &imin);
Histogram (x, n, min, max, hist, axis, intervals);
```

IIRCascadeFiltering

`int` **status** = **IIRCascadeFiltering** (`const double` **x**`[]`, `int` **n**, `IRFilterPtr` **filterInformation**, `double` **y**`[]`);

Purpose

Filters the input sequence using the cascade IIR filter specified by the **filterInformation** structure. Each of the IIR cascaded stages is 2nd order for lowpass and highpass filters, and 4th order for bandpass and bandstop filters.

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` and then call one of the cascade IIR design functions (`Bw_CascadeCoef`, `Ch_CascadeCoef`, `Elp_CascadeCoef`, `InvCh_CascadeCoef`, `Bessel_CascadeCoef`) before calling this function.

The internal filter state information for the filtering operation is kept in the **filterInformation** structure, so this function can be called in a loop, continually filtering new input array data, producing new output filtered data.

If you have finished filtering one set of input data and wish to filter a completely new data set, you should call `ResetIIRFilter` before calling this function with the new data. `ResetIIRFilter` will cause the internal filter state information to be cleared before the next filtering operation.

Parameters

Input	**x**	const double-precision	Array containing the raw data to be filtered.
	n	integer	Specifies the number of points in both the input **x** and output **y**.
	filterInformation	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filtering function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.
Output	**y**	double-precision array	Array contains the output of the IIR Filtering operation. The size of this array must be at least **n**.

Return Value

status	integer	Refer to error codes in Appendix A.

IIRFiltering

`int` **status = IIRFiltering** (`double` **x**`[]`, `int` **nx**, `double` **a**`[]`, `double` **y1**`[]`, `int` **na**,
`double` **b**`[]`, `double` **x1**`[]`, `int` **nb**, `double` **y**`[]`);

Purpose

Filters the input sequence using the IIR filter specified by reverse coefficients **a**[**na**]and forward coefficients **b**[**nb**] by

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-i}\right)$$

The reverse and forward coefficients are obtained by respective IIR Coefficient functions such as `Bw_Coef( )`.

Parameters

Input	**x**	double-precision array	raw data to be filtered.
	nx	integer	number of points in both the input X array.
	a	double-precision array	array containing the *reverse* coefficients for the IIR filtering operation.
	y1	double-precision array	**y1** [**na**-1] contains the initial conditions, or states. The size of this array must be at least **na**-1.
	na	integer	number of coefficients in both the **a** Coefficients array and the **y1** conditions array.
	b	double-precision array	array containing the *forward* coefficients for the IIR filtering operation.
	x1	double-precision array	**x1** [**nb**-1] contains the initial conditions, or states. The size of this array must be at least **nb**-1.
	nb	integer	number of coefficients in both the **b** Coefficients array and the **x** conditions array.

(continues)

Parameters (Continued)

Output	**y1**	double-precision array	on output, **y1**[**na**-1] contains the final conditions for the next iterations.
	x1	double-precision array	on output, **x1** [**nb**-1] contains the final conditions for the next iterations.
	y	double-precision array	y array contains the output of the IIR filtering operation. The size of this array must be at least **nx**.

Return Value

status	integer	refer to error codes in Appendix A

Impulse

`int` **status** = **Impulse** (`int` **n**, `double` **amp**, `int` **index**, `double` **x[]**);

Purpose

Generates an array of numbers that has the pattern of an impulse waveform. The i^{th} element of the output array is obtained using the following formula.

$$x_i = \begin{cases} \textbf{amp} & \text{if } i = \textbf{index} \\ 0 & \text{otherwise} \end{cases}$$

Parameters

Input	**n**	integer	number of elements in **x**
	amp	double-precision	amplitude
	index	integer	impulse index
Output	**x**	double-precision array	impulse array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates the impulse pattern
x = { 0.0, 0.0, 1.5, 0.0, 0.0 }. */
double  x[5], amp;
int  n, i;
n = 5;
i = 2;
amp = 1.5;
Impulse (n, amp, i, x);
```

ImpulseResponse

`int` **status** = **ImpulseResponse** (`double` **stimulus**`[]`, `double` **response**`[]`, `int` **n**, `double` **impulse**`[]`);

Purpose

Computes the impulse response of a network based on time-domain signals stimulus and response. The impulse response is in the time domain. The impulse response is the inverse Fourier transform of the transfer function.

impulse *= Inverse Real FFT [Sxy(f) / Sxx(f)]*

where Sxy(f) is the two-sided cross power spectrum of the **stimulus** (x) with the **response** (y), and Sxx(f) is the two-sided auto power spectrum of the stimulus.

Parameters

Input	**stimulus**	double-precision array	contains the time-domain signal, usually the network stimulus.
	response	double-precision array	contains the time-domain signal, usually the network response.
	n	integer	number of elements in the input array. Valid Values: Powers of 2.
Output	**impulse**	double-precision array	impulse contains the impulse response of the network based on time-domain signals stimulus and response. The size of this array must be at least **n**.

Return Value

status	integer	refer to error codes in Appendix A

Integrate

`int` **status** = **Integrate** (`double` **x**`[]`, `int` **n**, `double` **dt**, `double` **xInit,** `double` **xFinal**, `double` **y**`[]`);

Purpose

Computes the discrete integral of the input array. The i^{th} element of the resulting array is obtained using the following formula.

$$y_i = \sum_{j=0}^{i} \left[x_{j-1} + 4x_j + x_{j+1} \right] * dt/6$$

where $\mathbf{x}_{-1}$ = **xInit** and $\mathbf{x}_n$ = **xFinal**.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	dt	double-precision	sampling interval
	xInit	double-precision	initial condition
	xFinal	double-precision	final condition
Output	**y**	double-precision array	integrated array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers and integrate it. */
double  x[200], y[200];
double  dt, xInit, xFinal;
int  n;
n = 200;
dt = 0.001;
xInit = -0.5;
xFinal = -0.25;
Uniform (n, 17, x);
Integrate (x, n, dt, xInit, xFinal, y);
```

InvCh_BPF

`int` **status =InvCh_BPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandpass inverse Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandpass inverse
Chebyshev filter.  The pass band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh, atten;
int  n, order;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
InvCh_BPF (x, n, fs, fl, fh, atten, order, y);
```

InvCh_BSF

`int` **status** = **InvCh_BSF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fl**, `double` **fh**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital bandstop inverse Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order bandstop inverse
Chebyshev filter.  The stop band is from 200.0 to 300.0. */
double  x[256], y[256], fs, fl, fh, atten;
int  n, order;
n = 256;
fs = 1000.0;
fl = 200.0;
fh = 300.0;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
InvCh_BSF (x, n, fs, fl, fh, atten, order, y);
```

InvCh_CascadeCoef

`int` **status** = **InvCh_CascadeCoef** (`double` **fs**, `double` **fL**, `double` **fH**, `double` **atten**, `IIRFilterPtr` **filterInformation**);

Purpose

Generates the set of cascade form filter coefficients to implement an IIR filter as specified by the inverse Chebyshev filter model.

filterInformation is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function.

To redesign another filter, you should first call `FreeIIRFilterPtr` to free the present filter structure and then call `AllocIIRFilterPtr` with the new type and order parameters before calling this design function.

If the type and order remain the same, and you can call this IIR design function without calling `FreeIIRFilterPtr` and `AllocIIRFilterPtr`. In this case, you should properly reset the filtering operation for that structure by calling `ResetIIRFilter` before the first call to `IIRCascadeFiltering`.

Parameters

Input	**fs**	double-precision	Specifies the sampling frequency in Hz.
	fL	double-precision	Specifies the desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	Specifies the desired upper cutoff frequency of the filter in Hz
	atten	double-precision	Specifies the stop band attenuation, in decibels, of the IIR filter to be designed.
Output	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before calling this cascade IIR filter design function. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/* Design a cascade lowpass inverse Chebyshev IIR filter */
double      fs, fl, fh, atten, x[256], y[256];
int      type,order,n;
IIRFilterPtr   filterInfo;
n = 256;
fs = 1000.0;
fl = 200.0;
atten = 60.0;
order = 5;
type = 0;      /* lowpass  */
Uniform(n,17,x);
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   InvCh_CascadeCoef(fs,fl,fh,atten,filterInfo);
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

InvCh_Coef

`int` **status** = **InvCh_Coef** (`int` **type**, `int` **order**, `double` **fs**, `double` **fL**, `double` **fH**,`double` **atten**, `double` **a**`[]`, `int` **na**, `double` **b**`[]`, `int` **nb**);

Purpose

Generates the set of filter coefficients to implement an IIR filter as specified by the inverse Chebyshev filter model. The **type** parameter has the following valid values.

$$\textbf{type} = \begin{cases} 0 & \text{lowpass filter, } \textbf{fH} \text{ is not used.} \\ 1 & \text{highpass filter, } \textbf{fH} \text{ is not used.} \\ 2 & \text{bandpass filter} \\ 3 & \text{bandstop filter} \end{cases}$$

a[**na**] and **b**[**nb**] are the reverse and forward filter coefficients. The actual filtering

$$y_n = \frac{1}{a_0}\left(\sum_{i=0}^{nb-1} b_i\, x_{n-i} - \sum_{i=1}^{na-1} a_i y_{n-i}\right)$$

is achieved by using the function `IIRFiltering`.

Parameters

Input	**type**	integer	controls the filter type of the inverse Chebyshev IIR filter coefficients.
	order	integer	order of the IIR filter.
	fs	double-precision	sampling frequency in Hz.
	fL	double-precision	desired lower cutoff frequency of the filter in Hz.
	fH	double-precision	desired lower cutoff frequency of the filter in Hz.
	atten	double-precision	stop band attenuation, in decibels, of the IIR filter to be designed.
	na	integer	number of coefficients in the **a** coefficient array.
	nb	integer	number of coefficients in the **b** coefficient array.
Output	**a**	double-precision array	array containing the *reverse* coefficients of the designed IIR filter.
	b	double-precision array	array containing the *forward* coefficients of the designed IIR filter.

Return Value

status	integer	refer to error codes in Appendix A

InvCh_HPF

`int` **status** = **InvCh_HPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital highpass inverse Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order highpass inverse
Chebyshev filter. */
double  x[256], y[256], fs, fc, atten;
int  n, order;
n = 256;
fs = 1000.0;
fc = 200.0;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
InvCh_HPF (x, n, fs, fc, atten, order, y);
```

InvCh_LPF

`int` **status** = **InvCh_LPF** (`double` **x**`[]`, `int` **n**, `double` **fs**, `double` **fc**, `double` **atten**, `int` **order**, `double` **y**`[]`);

Purpose

Filters the input array using a digital lowpass inverse Chebyshev filter. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	fs	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	atten	double-precision	stop band attenuation in dB
	order	integer	filter order
Output	**y**	double-precision array	filtered data

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random signal and filter it using a fifth order lowpass inverse
Chebyshev filter. */
double  x[256], y[256], fs, fc, atten;
int  n, order;
n = 256;
fs = 1000.0;
fc = 200.0;
atten = 40.0;
order = 5;
Uniform (n, 17, x);
InvCh_LPF (x, n, fs, fc, atten, order, y);
```

InvF_Dist

`int` **status** = **InvF_Dist** (`double` **p**, `int` **n**, `int` **m**, `double` ***f**);

Purpose

Calculates **f**, given a probability $0 \le p < 1$, such that

$$prob(F < f) = p$$

where F is a random variable from an F-distribution with **n** and **m** degrees of freedom.

Parameters

Input	**p** **n** **m**	double-precision integer integer	probability (0 ≤ **p** < 1) degrees of freedom degrees of freedom
Output	**f**	double-precision	the unique number **f** such that prob(F < f) = p, where F is a random variable from an F-distribution with **n** and **m** degrees of freedom.

Note: ***When* p = *0,* f = *0.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double p, f;
int n,m;
p = 0.635;
n = 2;
m = 4;
InvF_Dist (p, n, m, &f);
```

InvFFT

`int` **status** = **InvFFT** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**);

Purpose

Computes the inverse Fast Fourier Transform of the complex data. Let X = x + jy be the complex array, then:

$$Y = FFT^{-1} \{X\}$$

The operation is done in place and the input arrays **x** and **y** are overwritten.

Parameters

Input	**x**	double-precision array	real part of complex array
	y	double-precision array	imaginary part of complex array
	n	integer	number of elements
Output	**x**	double-precision array	real part of IFFT
	y	double-precision array	imaginary part of IFFT

Note: ***The number of elements* (n) *must be a power of two*.**

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate two arrays with random numbers and compute its inverse Fast
Fourier Transform. */
double  x[256], y[256];
int  n;
n = 256;
Uniform (n, 17, x);
Uniform (n, 17, y);
InvFFT (x, y, n);
```

InvFHT

`int` **status** = **InvFHT** (`double` **x**`[]`, `int` **n**);

Purpose

Computes the inverse Fast Hartley Transform using the following formula:

$$x_i = \tfrac{1}{n}\sum_{k=0}^{n-1} X_k \, cas(2\pi ik / n)$$

where x_i is the i^{th} point of the inverse FHT, and cas (x) = cos (x) + sin (x).

The operation is done in place and the **x** input array is overwritten.

Parameters

Input	**x** **n**	double-precision array integer	array to be transformed number of elements
Output	**x**	double-precision array	inverse Fast Hartley Transform

Note: ***The number of elements (*****n*****) must be a power of two.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers and compute its inverse Fast Hartley
Transform. */
double  x[256];
int  n;
n = 256;
Uniform (n, 17, x);
InvFHT (x, n);
```

InvMatrix

`int` **status** = **InvMatrix** (`void` ***x**, `int` **n**, `void` ***y**);

Purpose

Finds the inverse matrix of an input matrix. The operation can be performed in place; that is, **x** and **y** can be the same matrices.

Parameters

Input	**x** **n**	double-precision 2D array integer	input matrix dimension size of matrix
Output	**y**	double-precision 2D array	inverse matrix

Note: ***The input matrix must be an n-by-n square matrix***.

Return Value

status	integer	refer to error codes in Appendix A

InvN_Dist

`int` **status** = **InvN_Dist** (`double` **p**, `double *`**x**);

Purpose

Calculates **x**, given a probability 0 < **p** < 1, such that:

prob(X < x) = p

where X is a random variable from a standard normal distribution.

Parameters

Input	**p**	double-precision	probability (0 < **p** < 1)
Output	**x**	double-precision	the unique number **x** such that prob(X < x) = p, where X is a random variable from a standard normal distribution

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double p, x;
p = 0.5;
InvN_Dist (p, &x);
```

InvT_Dist

`int` **status = InvT_Dist** (`double` **p**, `int` **n**, `double` ***t**);

Purpose

Calculates **t**, given a probability 0 < **p** < 1, such that:

$$prob(\ T < t\ ) = p$$

where T is a random variable from a T-distribution with **n** degrees of freedom.

Parameters

Input	**p** **n**	double-precision integer	probability (0 < **p** < 1) degrees of freedom
Output	**t**	double-precision	the unique number **t** such that prob(T < t) = p, where T is a random variable from a T-distribution with **n** degrees of freedom

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double p, t;
int n;
p = 0.635;
n = 2;
InvT_Dist (p, n, &t);
```

InvXX_Dist

`int` **status = InvXX_Dist** (`double` **p**, `int` **n**, `double` ***x**);

Purpose

Calculates **x**, given a probability 0 ≤ **p** < 1, such that:

$$prob(\ \chi < x\ ) = p$$

where χ is a random variable from a chi-square distribution with **n** degrees of freedom.

Parameters

Input	**p** **n**	double-precision integer	probability ($0 \leq$ **p** < 1) degrees of freedom
Output	**x**	double-precision	the unique number **x** such that prob(χ < x) = p, where χ is a random variable from a chi-square distribution with **n** degrees of freedom.

Note: ***When* p = *0*, x = *0.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double p, x;
int n;
p = 0.635;
n = 2;
InvXX_Dist (p, n, &x);
```

Ksr_BPF

`int` **status** = **Ksr_BPF** (`double` **fs**, `double` **fl**, `double` **fh**, `int` **n**,
`double` **coef**`[]`,`double` **beta**);

Purpose

Designs a digital bandpass FIR linear phase filter using a Kaiser window. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	n	integer	number of filter coefficients
	beta	double-precision	shape parameter
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Some **beta** values and their equivalent windows are listed in the following table:

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Digital Signal Processing* by Oppenheim and Schafer for more information.

Example

```
/* Design a 55-point bandpass FIR linear phase filter using a Kaiser window
with beta = 4.5.  Filter the incoming signal with the designed filter. */
double  x[256], coef[55], y[310], fs, fl, fh, beta;
int  n, m;
fs = 1000.0;         /* sampling frequency */
fl = 200.0;          /* desired lower cutoff frequency */
fh = 300.0;          /* desired higher cutoff frequency */
                     /* pass band is from 200.0 to 300.0 */
n = 55;              /* filter length */
beta = 3;
m = 256;
Ksr_BPF (fs, fl, fh, n, coef, beta);
Convolve (coef, n, x, m, y);/* convolve the filter with the signal */
```

Ksr_BSF

`int` **status = Ksr_BSF** (`double` **fs**, `double` **fl**, `double` **fh**, `int` **n**, `double` **coef**`[]`,
`double` **beta**);

Purpose

Designs a digital bandstop FIR linear phase filter using a Kaiser window. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	n	integer	number of filter coefficients
	beta	double-precision	shape parameter
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Some **beta** values and their equivalent windows are listed in the following table:

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Digital Signal Processing* by Oppenheim and Schafer for more information.

Example

```
/* Design a 55-point bandstop FIR linear phase filter using a Kaiser window
with beta = 4.5.  Filter the incoming signal with the designed filter. */
double  x[256], coef[55], y[310], fs, fl, fh, beta;
int  n, m;
fs = 1000.0;          /* sampling frequency */
fl = 200.0;           /* desired lower cutoff frequency */
fh = 300.0;           /* desired higher cutoff frequency */
                      /* stop band is from 200.0 to 300.0 */
n = 55;               /* filter length */
beta = 3;
m = 256;
Ksr_BSF (fs, fl, fh, n, coef, beta);
Convolve (coef, n, x, m, y);/* convolve the filter with the signal */
```

Ksr_HPF

`int` **status** = **Ksr_HPF** (`double` **fs**, `double` **fc**, `int` **n**, `double` **coef**`[]`, `double` **beta**);

Purpose

Designs a digital highpass FIR linear phase filter using a Kaiser window. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	n	integer	number of filter coefficients
	beta	double-precision	shape parameter
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Some **beta** values and their equivalent windows are listed in the following table:

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Digital Signal Processing* by Oppenheim and Schafer for more information.

Example

```
/* Design a 55-point highpass FIR linear phase filter using a Kaiser window
with beta = 4.5.  Filter the incoming signal with the designed filter. */
double  x[256], coef[55], y[310], fs, fc, beta;
int  n, m;
fs = 1000.0;                  /* sampling frequency */
fc = 200.0;                   /* desired cutoff frequency */
n = 55;                       /* filter length */
beta = 4.5;
m = 256;
Ksr_HPF (fs, fc, n, coef, beta);
Convolve (coef, n, x, m, y);  /* convolve the filter with the signal */
```

Ksr_LPF

`int` **status** = **Ksr_LPF** (`double` **fs**, `double` **fc**, `int` **n**, `double` **coef**`[]`, `double` **beta**);

Purpose

Designs a digital lowpass FIR linear phase filter using a Kaiser window. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	n	integer	number of filter coefficients
	beta	double-precision	shape parameter
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Some **beta** values and their equivalent windows are listed in the following table:

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Digital Signal Processing* by Oppenheim and Schafer for more information.

Example

```
/* Design a 55-point lowpass FIR linear phase filter using a Kaiser window
with beta = 4.5.  Filter the incoming signal with the designed filter. */
double  x[256], coef[55], y[310], fs, fc, beta;
int  n, m;
fs = 1000.0;          /* sampling frequency */
fc = 200.0;        /* desired cutoff frequency */
n = 55;               /* filter length */
beta = 4.5;
m = 256;
Ksr_LPF (fs, fc, n, coef, beta);
Convolve (coef, n, x, m, y);  /* convolve the filter with*/
                     /*the signal */
```

KsrWin

`int` **status** = **KsrWin** (`double` **x**`[]`, `int` **n**, `double` **beta**);

Purpose

Applies a Kaiser window to the **x** input signal. The Kaiser window is defined by the formula:

$$w_i = Io\ (beta * (1.0 - a^2)^{1/2})\ /\ Io\ (beta) \quad for\ i = 0, 1, ..., n-1$$

where $a = |1 - 2i/ n|$; and Io represents the zeroth-order modified Bessel function of the first kind.

The output signal is obtained by the formula:

$$x_i = x_i * w_i \qquad \text{for } i = 0, 1, ..., n-1$$

The window operation is performed in place. The windowed data **x** replaces the input data **x**.

Parameters

Input	**x**	double-precision array	input data
	n	integer	number of elements in **x**
	beta	double-precision	shape parameter
Output	**x**	double-precision array	windowed data

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Some **beta** values and their equivalent windows are listed in the following table:

beta	**Window**
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Digital Signal Processing* by Oppenheim and Schafer for more information.

LinEqs

`int` **status** = **LinEqs** (`void` ***A**, `double` **y**`[]`, `int` **n**, `double` **x**`[]`);

Purpose

Solves the linear system of equations:

$$Ax = y$$

Parameters

Input	**A**	double-precision 2D array	input matrix
	y	double-precision 1D array	known vector
	n	integer	dimension size of system
Output	**x**	double-precision 1D array	solution of vector

Note: ***The* A *input matrix must be an n-by-n square matrix.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Find the solution to the linear system of equations. */
double  A[10][10], y[10], x[10];
int  n;
n = 10;
   :
LinEqs (A, y, n, x);
```

LinEv1D

`int` **status** = **LinEv1D** (`double` **x**`[]`, `int` **n**, `double` **a**, `double` **b**, `double` **y**`[]`);

Purpose

Performs a linear evaluation of a 1D array. The i^{th} element of the output array is obtained using the formula:

$$y_i = a * x_i + b$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
	a	double-precision	multiplicative constant
	b	double-precision	additive constant
Output	**y**	double-precision array	linearly evaluated array

Return Value

status	integer	refer to error codes in Appendix A

LinEv2D

`int` **status** = **LinEv2D** (`void` ***x**, `int` **n**, `int` **m**, `double` **a**, `double` **b**, `void` ***y**);

Purpose

Performs a linear evaluation of a 2D array. The (i^{th}, j^{th}) element of the output array is obtained using the formula:

$$y_{i,j} = a * x_{i,j} + b$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
	a	double-precision	multiplicative constant
	b	double-precision	additive constant
Output	**y**	double-precision 2D array	linearly evaluated array

Return Value

status	integer	refer to error codes in Appendix A

LinFit

`int` **status** = **LinFit** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **z**`[]`, `double` ***slope**, `double` ***intercept**, `double` ***mse**);

Purpose

Finds the **slope** and **intercept** values that best represent the linear fit of the data points (**x**, **y**) using the least squares method. The i^{th} element of the output array is obtained by using the following formula:

$$z_i = slope * x_i + intercept$$

The mean squared error (**mse**) is obtained using the following formula:

$$mse = \sum_{i=0}^{n-1} |z_i - y_i|^2 / n$$

where **n** is the number of sample points.

Parameters

Input	**x**	double-precision array	**x** values
	y	double-precision array	**y** values
	n	integer	number of sample points
Output	**z**	double-precision array	best fit array
	slope	double-precision	slope of line
	intercept	double-precision	y-intercept
	mse	double-precision	mean squared error

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a ramp pattern and find the best linear fit. */
double  x[200], y[200], z[200];
double  start, end, a, b, slope, intercept, mse;
int  n;

n = 200;
start = 0.0;
end = 1.99E2;
Ramp (n, start, end, x);      /* x[i] = i */

a = 3.5;
b = -2.75;
LinEv1D (x, n, a, b, y);      /* y[i] = a*x[i] + b */

/* Find the best linear fit in z. */
LinFit (x, y, n, z, &slope, &intercept, &mse);
```

LU

`int` **status** = **LU** (`void` ***a**, `int` **n**, `int` **p**`[]`, `int` ***sign**);

Purpose

Performs an LU matrix decomposition.

$$a = L * U$$

where L is an **n** by **n** lower triangular matrix whose main diagonal elements are all ones, and U is an upper triangular matrix.

Parameters

Input	**a**	double-precision 2D array	input matrix
	n	integer	dimension size
Output	**a**	double-precision 2D array	LU decomposition
	p	integer array	permutation vector
	sign	integer	row exchange indicator

Note: ***The input matrix is overwritten by the*** **`LU`** ***output matrices.***

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

After the function executes, the input matrix **a** is replaced with two triangular matrices. L occupies the lower triangular part of **a** and U occupies the upper triangular part of **a**. The permutation vector **p** records possible row exchange information in the LU decomposition. **sign** = 0 indicates that there is no such exchange or that there is an even number of such exchanges. **sign** = 1 indicates that there is an odd number of such exchanges. **p** and **sign** are useful when solving the linear equations or computing the determinant. LU is most useful when used in conjunction with `BackSub` and `ForwSub` to solve a set of linear equations with the same matrix **a**.

For more information, refer to *Numerical Recipes* by Press, *et al.*, Cambridge University Press.

MatrixMul

`int` **status** = **MatrixMul** (`void` *__x__, `void` *__y__, `int` **n**, `int` **k**, `int` **m**, `void` *__z__);

Purpose

Multiplies two 2D input matrices. The (i[th], j[th]) element of the output matrix is obtained using the formula:

$$z_{i,j} = \sum_{p=0}^{k-1} x_{i,p} * y_{p,j}$$

Parameters

Input	**x**	double-precision 2D array	**x** input matrix
	y	double-precision 2D array	**y** input matrix
	n	integer	first dimension of **x**
	k	integer	second dimension of **x**; first dimension of **y**
	m	integer	second dimension of **y**
Output	**z**	double-precision 2D array	output matrix

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

Be careful to use the correct array sizes. The following array sizes must be met:

- **x** must be (**n** by **k**).
- **y** must be (**k** by **m**).
- **z** must be (**n** by **m**).

Example

```
/* Multiply two matrices. Note: A x B _ B x A,  in general.*/
double  x[10][20], y[20][15], z[10][15];
int n, k, m;
n = 10;
k = 20;
m = 15;
MatrixMul (x, y, n, k, m, z);
```

MaxMin1D

`int` **status** = **MaxMin1D** (`double` **x**`[]`, `int` **n**, `double` ***max**, `int` ***imax**, `double` ***min**, `int` ***imin**);

Purpose

Finds the maximum and minimum values in the input array, as well as the respective indices of the first occurrence of the maximum and minimum values.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
Output	**max**	double-precision	maximum value
	imax	integer	index of **max** in **x** array
	min	double-precision	minimum value
	imin	integer	index of **min** in **x** array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random and find the maximum and minimum values. */
double  x[20], y[20];
double  max, min;
int  n, imax, imin;
n = 20;
Uniform (n, 17, x);
MaxMin1D (x, n, &max, &imax, &min, &imin);
```

MaxMin2D

`int` **status** = **MaxMin2D** (`void` ***x**, `int` **n**, `int` **m**, `double` ***max**, `int` ***imax**, `int` ***jmax**, `double` ***min**, `int` ***imin**, `int` ***jmin**);

Purpose

Finds the maximum and the minimum values in the 2D input array, as well as the respective indices of the first occurrence of the maximum and minimum values. The **x** array is scanned by rows.

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension of **x**
	m	integer	number of elements in second dimension of **x**
Output	**max**	double-precision	maximum value
	imax	integer	index of **max** in **x** array (first dimension)
	jmax	integer	index of **max** in **x** array (second dimension)
	min	double-precision	minimum value
	imin	integer	index of **min** in **x** array (first dimension)
	jmin	integer	index of **min** in **x** array (second dimension)

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* This example finds the maximum and minimum values as well as their location
within the array. */
double x[5][10], max, min;
int n, m, imax, jmax, imin, jmin;
n = 5;
m = 10;
MaxMin2D (x, n, m, &max, &imax, &jmax, &min, &imin, &jmin);
```

Mean

`int` **status** = **Mean** (`double` **x**`[]`, `int` **n**, `double` ***meanval**);

Purpose

Computes the mean (average) value of the input array. The following formula is used to find the mean.

$$\text{meanval} = \sum_{i=0}^{n-1} x_i \;/\; n$$

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
Output	**meanval**	double-precision	mean value

Return Value

status	integer	refer to error codes in Appendix A

Median

`int` **status** = **Median** (`double` **x**`[]`, `int` **n**, `double` ***medianval**);

Purpose

Finds the median value of the **x** input array. To find the median value, the input array is first sorted in ascending order. Let S be the sorted array, then:

$$\text{medianval} = \begin{cases} S\left(\frac{n}{2}\right) & \text{if n is odd} \\ 0.5*(S\left(\frac{n}{2}-1\right)+S\left(\frac{n}{2}\right)) & \text{if n is even} \end{cases}$$

Note: ***The* x *input array is not changed.***

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements in **x**
Output	**medianval**	double-precision	median value

Return Value

status	integer	refer to error codes in Appendix A

Mode

`int` **status** = **Mode** (`double` **x**`[]`, `int` **n**, `double` **xBase**, `double` **xTop**, `int` **intervals**, `double` ***modeval**);

Purpose

Finds the mode of the **x** input array. The mode is defined as the value that most often occurs in a given set of samples. This function determines the mode in terms of the histogram of the input array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	xBase	double-precision	lower range
	xTop	double-precision	upper range
	intervals	integer	number of intervals
Output	**modeval**	double-precision	mode value

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a Gaussian distributed random array and find its mode. */
double  x[2000], max, min, modeval;
int  n, intervals, imax, imin;
n = 2000;
intervals = 50;
GaussNoise (n, 1.0E0, 17, x);
MaxMin1D (x, n, &max, &imax, &min, &imin);
Mode (x, n, min, max, intervals, &modeval);
```

Moment

`int` **status** = **Moment** (`double` **x**`[]`, `int` **n**, `int` **order**, `double` ***momentval**);

Purpose

Computes the moment about the mean of the input array with the specified order. The formulas used to find the moment are as follows.

$$momentval = \sum_{i=0}^{n-1} \frac{[x_i - ave]^{order}}{n}$$

$$ave = \sum_{i=0}^{n-1} x_i \;/\; n$$

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	order	integer	moment order
Output	**momentval**	double-precision	moment about the mean

Note: order *must be greater than zero*.

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers and determine its skewness (third
order moment) and its kurtosis (fourth order moment). */
double  x[200], skew, kurtosis;
int  n, order;
n = 200;
Uniform (n, 17, x);
order = 3;
Moment (x, n, order, &skew);
order = 4;
Moment (x, n, order, &kurtosis);
```

Mul1D

`int` **status** = **Mul1D** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **z**`[]`);

Purpose

Multiplies two 1D arrays. The i[th] element of the output array is obtained using the following formula.

$$z_i = x_i * y_i$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x**	double-precision array	**x** input array
	y	double-precision array	**y** input array
	n	integer	number of elements to be multiplied
Output	**z**	double-precision array	result array

Return Value

status	integer	refer to error codes in Appendix A

Mul2D

`int` **status** = **Mul2D** (`void` *__x__, `void` *__y__, `int` **n**, `int` **m**, `void` *__z__);

Purpose

Multiplies two 2D arrays. The (i^{th}, j^{th}) element of the output array is obtained using the following formula.

$$z_{i,j} = x_{i,j} * y_{i,j}$$

The operation can be performed in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input	**x**	double-precision 2D array	**x** input array
	y	double-precision 2D array	**y** input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
Output	**z**	double-precision 2D array	result array

Return Value

status	integer	refer to error codes in Appendix A

N_Dist

`int` **status** = **N_Dist** (`double` **x**, `double` ***p**);

Purpose

Computes the one-sided probability **p**:

$p = prob(X \le x)$

where X is a random variable from a standard normal distribution.

Parameters

Input	**x**	double-precision	$-\infty < \mathbf{x} < \infty$
Output	**p**	double-precision	probability (0 < **p** < 1)

Return Value

status	integer	refer to error codes in Appendix A

Note: ***For computing the two-sided probability $p_2 = prob(-x \le X \le x)$, the following formula can be used.***
$p_2 = 1.0 - 2 * prob(X \le -x)$

Example

```
double x, p;
x = -123.456;
N_Dist (x, &p);
```

Neg1D

`int` **status** = **Neg1D** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`);

Purpose

Negates the elements of the input array. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements
Output	**y**	double-precision array	negated values of the **x** input array

Return Value

status	integer	refer to error codes in Appendix A

NetworkFunctions

```
int status = NetworkFunctions (void *stimulus, void *response, int n, int numFrames,
                               double dt, double magSxy[], double phaseSxy[],
                               double magHf[], double phaseHf[], double
                               coherence[], double impulse[], double *df);
```

Purpose

Computes the single-sided coherence function along with the averaged single-sided cross power spectrum, averaged single-sided frequency response (transfer function), and impulse response, from a 2D array of stimulus signals and a 2D array of response signals.

The network functions are computed as follows.

avg cross power = avg of Sxy(f)
avg transfer function = avg Sxy(f)/avg Sxx(f)
avg impulse response = Inverse Real FFT(avg two-sided transfer function)
coherence = $|\text{averaged Sxy(f)}|^2$ /[avg Sxx(f) x avg Syy(f)]

where

Sxy(f) is the two-sided cross power spectrum of x and y
Sxx(f) is the two-sided auto power spectrum of x
Syy(f) is the two-sided auto power spectrum of y
x is the stimulus signal
y is the response signal

stimulus is a 2D array containing a time-domain signal, usually the network stimulus. **response** is a 2D array containing a time-domain signal, usually the network response.

Each row in the stimulus array represents one frame of the network stimulus and is associated with one row of the response array, which represents one frame of the network response.

Parameters

Input	**stimulus**	double-precision 2D array	contains the time-domain signal, usually the network stimulus. The number of rows should be equal to **numFrames**, and the number of columns should be equal to the **n**. The size of this array must be at least: **numFrames*****n**.
	response	double-precision 2D array	contains the time-domain signal, usually the network stimulus. The number of rows should be equal to **numFrames**, and the number of columns should be equal to the **n**. The size of this array must be at least: **numFrames*****n**.
	n	integer	number of elements in one frame of the input stimulus and response arrays.
	numFrames	integer	number of frames (rows) contained in the input stimulus and response arrays.
	dt	double-precision	sample period of the time-domain signal, usually in seconds. **dt** = $1/_{fs}$, where fs is the sampling frequency of the time-domain signal.
Output	**magSxy**	double-precision array	averaged single-sided cross power spectrum between the stimulus and response, in volts rms^2 if the input signals are in volts. If the input signals are not in volts, the results are in input signal units RMS squared. This array must be at least **n**/2 elements long.
	phaseSxy	double-precision array	averaged single-sided phase spectrum in radians showing the difference between the phases of the response signal and the stimulus signal. This array must be at least **n**/2 elements long.
	magHf	double-precision array	magnitude of the averaged single-sided transfer function between the stimulus and response signals. This array must be at least **n**/2 elements long.
	phaseHf	double-precision array	phase, in radians of the averaged single-sided transfer function between the stimulus and response signals.

(continues)

Parameters (Continued)

	coherence	double-precision array	averaged single-sided coherence function spectrum. The coherence function shows the frequency content of the response due to the stimulus and measures the validity of the network frequency response measurement. This array must be at least **n**/2 elements long.
	impulse	double-precision array	contains the impulse response of the network based on time-domain signals stimulus and response. Impulse is computed from the averaged frequency response of the stimulus and response signals. The size of this array must be at least **n**.
	df	double-precision	points to the frequency interval, in hertz, if **dt** is in seconds. ***df** = 1/(**n*****dt**)

Return Value

status	integer	refer to error codes in Appendix A

NonLinearFit

`int` **status** = **NonLinearFit** (`double` **x**`[]`, `double` **y**`[]`, `double` **z**`[]`, `int` **n**,
`ModelFun` ***modelFunction**, `double` **a**`[]`, `int` **ncoef**,
`double` ***MSE**);

Purpose

This function uses the Levenberg-Marquardt algorithm to determine the least squares set of coefficients that best fit the set of input data points(X,Y) as expressed by a nonlinear function y=f(x,a) where a is the set of coefficients. This function also gives the best fit curve y=f(x,a).

The user needs to pass a pointer to the nonlinear function f(**x**,**a**) along with a set of initial guess coefficients **a**[**ncoef**]. `NonLinearFit` does not always give the correct answer. The correct output sometimes depends on the initial choice of **a**[**ncoef**]. It is very important to verify the final result.

The output mse (mean squared error) is computed using the following formula.

$$mse = \frac{1}{n}\sum_{i=0}^{n-1} \left[y_i - f(x_i, a)\right]^2$$

Parameters

Input	**x**	double-precision array	The array of x coordinates of the (x,y) data sets to be fitted.
	y	double-precision array	The array of y coordinates of the (x,y) data sets to be fitted.
	n	integer	The number of elements in both the x and y arrays.
	modelFunction	ModelFun pointer	A pointer to the model function, f(x[i],a), used in the nonlinear fitting algorithm. The model function must be defined as follows: `double ModelFunct (double x, double a[], int ncoef);` where **a**[ncoef] are the function coefficients.
	a	double-precision array	On input, **a**[ncoef] gives a set of initial guess coefficients.
	ncoef	integer	Number of coefficients.
Output	**z**	double-precision array	Best fit array, y = f(x,a).
	a	double-precision array	Best fit coefficients.
	MSE	double-precision	Mean squared error between **y** and **z.**

Return Value

status	integer	refer to error codes in Appendix A

Normal1D

`int` **status** = **Normal1D** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`, `double` ***ave**, `double` ***sDev**);

Purpose

Normalizes a 1D input vector. The output vector is of the following form.

$y_i = (x_i - ave) / sDev$

where **ave** and **sDev** are the mean and the standard deviation of the input vector. Refer to the `StdDev` function for the formulas used to find the mean and the standard deviation.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input vector
	n	integer	number of elements
Output	**y**	double-precision array	normalized vector
	ave	double-precision	mean value of **x**
	sDev	double-precision	standard deviation of **x**

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a vector (1D array) with random samples and normalize it. */
double  x[200], y[200], ave, sDev;
int  n;
n = 200;
Uniform (n, 17, x);
Normal1D (x, n, y, &ave, &sDev);
```

Normal2D

`int` **status** = **Normal2D** (`void` *__x__, `int` **n**, `int` **m**, `void` *__y__, `double` *__ave__, `double` *__sDev__);

Purpose

Normalizes a 2D input matrix. The output matrix is of the following form.

$$y_{i,j} = \left(x_{i,j} - ave\right) / \, sDev$$

where **ave** and **sDev** are the mean and the standard deviation of the input matrix. Refer to the `StdDev` function for the formulas used to find the mean and the standard deviation.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision 2D array	input matrix
	n	integer	size of first dimension
	m	integer	size of second dimension
Output	**y**	double-precision 2D array	normalized matrix
	ave	double-precision	mean value of **x**
	sDev	double-precision	standard deviation of **x**

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Normalize a matrix (2D array). */
double  x[10][20], y[10][20], ave, sDev;
int  n, m;
n = 10;
m = 20;
  :
Normal2D (x, n, m, y, &ave, &sDev);
```

PolyEv1D

`int` **status** = **PolyEv1D** (`double` **x**`[]`, `int` **n**, `double` **coef**`[]`, `int` **k**, `double` **y**`[]`);

Purpose

Performs a polynomial evaluation on the input array. The i^{th} element of the output array is obtained using the following formula.

$$y_i = \sum_{j=0}^{k-1} coef_j * x_i^{\,j}$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
	coef	double-precision array	coefficients array
	k	integer	number of coefficients
Output	**y**	double-precision array	polynomially evaluated array

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The order of the polynomial is equal to the number of elements in the coefficients array minus one; that is, if there are **k** elements in the **coef** array, then order = **k** - 1.

Example

```
/* Generate an array with random numbers, let the coefficients be { 1, 2, 3,
4, 5 } generated by the Ramp function and find the polynomial evaluation of
the array. */
double  x[20], y[20], a[5];
double  first, last;
int  n, k;
n = 20;
k = 5;
first = 1.0;
last = 5.0;
Uniform (n, 17, x);
Ramp (k, first, last, a);
PolyEv1D (x, n, a, k, y);
```

PolyEv2D

`int` **status** = **PolyEv2D** (`void` ***x**, `int` **n**, `int` **m**, `double` **coef**`[]`, `int` **k**, `void` ***y**);

Purpose

Performs a polynomial evaluation on a 2D input array. The (i^{th}, j^{th}) element of the output array is obtained using the following formula.

$$y_{i,j} = \sum_{p=0}^{k-1} coef_p * x_{i,j}{}^p$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
	coef	double-precision array	coefficients array
	k	integer	number of coefficients
Output	**y**	double-precision 2D array	polynomially evaluated array

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The order of the polynomial is equal to the number of elements in the coefficients array minus one; that is, if there are **k** elements in the **coef** array, then order = **k** - 1.

Example

```
/* Perform a polynomial evaluation of a 2D array, let the coefficients be { 1,
2, 3, 4, 5 } generated by the Ramp function and find the polynomial evaluation
of the array. */
double  x[5][10], y[5][10], a[5];
double  first, last;
int  n, m, k;
n = 5;
k = 5;
m = 10;
first = 1.0;
last = 5.0;
Ramp (k, first, last, a);
PolyEv2D (x, n, m, a, k, y);
```

PolyFit

`int` **status** = **PolyFit** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `int` **order**, `double` **z**`[]`, `double` **coef**`[]`, `double` ***mse**);

Purpose

Finds the coefficients that best represent the polynomial fit of the data points (**x**, **y**) using the least squares method. The i[th] element of the output array is obtained by using the following formula.

$$z_i = \sum_{n=0}^{order} coef_n x_i^{\,n}$$

The mean squared error (**mse**) is obtained using the following formula.

$$mse = \sum_{i=0}^{n-1} \left| z_i - y_i \right|^2 / n$$

where **order** is the polynomial order and **n** is the number of sample points.

Parameters

Input	**x**	double-precision array	x values
	y	double-precision array	y values
	n	integer	number of sample points
	order	integer	polynomial order
Output	**z**	double-precision array	best fit
	coef	double-precision array	polynomial coefficients
	mse	double-precision	mean squared error

Note: ***The size of the coefficients array must be* order *+1.***

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a 10th order polynomial pattern with random coefficients and find
the polynomial fit. */
double  x[200], y[200], z[200], a[11], coef[11];
```

```
double  first, last, mse;
int  n, k, order;

n = 200;
first = 0.0;
last = 1.99E2;
Ramp (n, first, last, x)      /* x[i] = i */

k = 11;
Uniform (k, 17, a);
PolyEv1D (x, n, a, k, y);     /* polynomial pattern */

                  /* Find the best polynomial fit */
order = 10;
PolyFit (x, y, n, order, z, coef, &mse);
```

PolyInterp

`int` **status** = **PolyInterp** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **x_val**, `double` ***Interp_Val**, `double` ***Error**);

Purpose

Returns the value of the unique polynomial P of degree n-1 passing through the **n** points $(x_i,f(x_i))$ at **x_val**, along with an estimate of the error in the interpolation, given a set of **n** points $(x_i,f(x_i))$ in the plane where f is some function, and given a value **x_val** at which f is to be interpolated or extrapolated.

Parameters

Input	**x**	1D double-precision array	Values at which the function to be interpolated is known.
	y	1D double-precision array	Function values f(x) at the known **x** values.
	n	integer	Number of points in **x** and in **y**.
	x_val	double-precision	Value at which f is to be interpolated or extrapolated.
Output	**Interp_Val**	double-precision	Interpolated or extrapolated value at **x_val**.
	Error	double-precision	Estimate of the error in the interpolation.

Using This Function

All input arrays should be the same size. If the value of **x_val** is in the range of **x**, this function performs interpolation. Otherwise, it performs extrapolation. If **x_val** is too far from the range of **x**, **Error** might be large, and it would not produce a satisfactory extrapolation.

Example

```
/* Pick points randomly, pick an x in the range of X-values, run a polynomial
through the points, and interpolate at x_val. */
double X[10], Y[10], Interp_Val, Error, x_val, high, low;
int n, i;
n = 10;
WhiteNoise (n, 5.0, 17, X);
WhiteNoise (n, 5.0, 17, Y);
high = X[0];
low = X[0];
for(i=0; i<n; i++) {
   if (X[i] > high) high = X[i];
   if (X[i] < low) low = X[i];
}
x_val = (high + low)/2.0;
PolyInterp (x, y, n, x_val, &Interp_Val, &Error);
```

PowerFrequencyEstimate

`int` **status** = **PowerFrequencyEstimate** (`double` **autoSpectrum**`[]`, `int` **n,**
`double` **searchFreq**,
`WindowStruct` **windowConstants**, `double` **df**,
`int` **span**, `double` ***freqPeak,** `double` ***powerPeak)**;

Purpose

Computes the estimated power and frequency around a peak in the power spectrum of a time-domain signal. With this function, you can achieve good frequency estimates for measured peaks that lie between frequency lines on the spectrum. This function also makes corrections for the window function you use.

The estimated frequency peak is computed with the following formula.

$$freqPeak = \frac{\sum_{j=i-span/2}^{i+span/2} autoSpectrum_j\, j * df}{\sum_{j=i-span/2}^{i+span/2} autoSpectrum_j}$$

The estimated power peak is computed as follows.

$$powerPeak = \sum_{j=i-span/2}^{i+span/2} (autoSpectrum_j) / enbw$$

where

i = index of the searchfreq,
df is the frequency interval, usually in hertz, as output by the `AutoPowerSpectrum` function
enbw is windowConstants.enbw as output by the `ScaledWindow` function.

Parameters

Input	**autoSpectrum**	double-precision array	The single-sided power spectrum as output by the `AutoPowerSpectrum` function.
	n	integer	The number of elements in the input AutoSpectrum array.
	searchFreq	double-precision	The frequency (usually in hertz) of the frequency around which you want to estimate the frequency and power. If **searchFreq** is less than zero, or, is not a valid frequency, this function will automatically search for the maximum peak in the **autoSpectrum** array and estimate the frequency and power around the maximum peak.
	windowConstants	WindowConst	A structure containing the following useful constants for the selected window: *enbw* is the equivalent noise bandwidth of the selected window. You can use this value to compute the power in a given frequency span. *coherentgain* is the peak gain of the window, relative to the peak gain of the Rectangular window. This value is used to normalize peak signal gains to that of the Rectangular window. This structure is output by the `ScaledWindow` function.

(continues)

Parameters (Continued)

	df	double-precision	The frequency interval, in hertz, as output by the following functions: `AmpPhaseSpectrum`, `AutoPowerSpectrum`, `CrossPowerSpectrum`, `NetworkFunctions`, `TransferFunction`.
	span	integer	The number of frequency lines (bins) around the peak to be included in the peak frequency and power estimation. The power in **span**/2 frequency lines before the peak frequency line, the peak frequency line itself, and **span**/2 frequency lines after the peak are included in the estimation.
Output	**freqPeak**	double-precision	Points to the estimated frequency of the estimated peak power in autospectrum.
	powerPeak	double-precision	Points to the estimated peak power in autospectrum.

Return Value

status	integer	refer to error codes in Appendix A

Prod1D

`int` **status** = **Prod1D** (`double` **x**`[]`, `int` **n**, `double` ***prod**);

Purpose

Finds the product of the **n** elements of the input array. The product of the elements is obtained using the following formula.

$$prod = \prod_{i=0}^{n-1} x_i$$

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements
Output	**prod**	double-precision	product of elements

Return Value

status	integer	refer to error codes in Appendix A

Pulse

`int` **status** = **Pulse** (`int` **n**, `double` **amp**`[]`, `int` **delay**, `int` **width**, `double` **pulsePattern**`[]`);

Purpose

Generates an array of numbers representing the pattern of a pulse waveform. The i[th] element of the output array is obtained using the formula.

$$pulsePattern_i = \begin{cases} \mathbf{amp} & \text{if } \mathbf{delay} \leq i < (\mathbf{delay} + \mathbf{width}) \\ 0 & \text{otherwise} \end{cases}$$

for i = 0, 1, 2, ..., n-1

Parameters

Input	**n** **amp** **delay** **width**	integer double-precision integer integer	number of samples pulse amplitude pulse delay pulse width
Output	**pulsePattern**	double-precision array	pulse pattern array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates the following pulse pattern
   pulsePattern = { 0.0, 0.0, 0.0, 2.0, 2.0, 2.0, 2.0, 2.0,
   0.0, 0.0 }. */
double  pulsePattern[10], amp;
int  n, delay, width;
n = 10;
delay = 3;
width = 5;
amp = 2.0;
Pulse (n, amp, delay, width, pulsePattern);
```

PulseParam

`int` **status** = **PulseParam** (`double` **pulsePattern**`[]`, `int` **n**, `double` ***amp**, `double` ***amp90**, `double` ***amp50**, `double` ***amp10**, `double` ***top**, `double` ***base**, `double` ***topOvershoot**, `double` ***baseOvershoot**, `int` ***delay**, `int` ***width**, `int` * **riseTime**, `int` ***fallTime**, `double` ***slewRate**);

Purpose

Analyzes the input array values for a pulse pattern and determines the pulse parameters that best describe the pulse pattern. It is assumed that the input array has a *bimodal distribution*, a distribution containing two distinct peak values.

Parameters

Input	**pulsePattern**	double-precision array	input array
	n	integer	number of elements
Output	**amp**	double-precision	amplitude
	amp90	double-precision	90% amplitude
	amp50	double-precision	50% amplitude
	amp10	double-precision	10% amplitude
	top	double-precision	top value
	base	double-precision	base value
	topOvershoot	double-precision	top overshoot
	baseOvershoot	double-precision	base overshoot
	delay	integer	pulse delay
	width	integer	width delay
	riseTime	integer	rise time
	fallTime	integer	fall time
	slewRate	double-precision	slew rate

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The returned parameters are as follows.

top = upper mode
base = lower mode
amp = **top** - **base**
amp90 = 90% amplitude
amp50 = 50% amplitude
amp10 = 10% amplitude
topOvershoot = maximum value - **top**
baseOvershoot = **base** - minimum value
delay = rising edge index (50% amplitude)
width = falling edge index (50% amplitude) - **delay**
riseTime = 90% amplitude index - 10% amplitude index on rising edge
fallTime = 10% amplitude index - 90% amplitude index on falling edge
slewRate = (90% amplitude - 10% amplitude)/**riseTime**

The parameters **delay**, **width**, **riseTime**, and **fallTime** are integers because the input is a discrete representation of a signal.

Example

```
/* Generate a noisy pulse pattern and determine its pulse parameters. */
double  x[200], y[200], amp, amp90, amp50, amp10, top, base;
double  topOvershoot, baseOvershoot, slewRate, noiseLevel;
int  n, delay, width, riseTime, fallTime;

n = 200;
amp = 5.0;
delay = 50;
width = 100;
noiseLevel = 0.5;
Pulse (n, amp, delay, width, x);    /* Generate a pulse */
WhiteNoise (n, noiseLevel, 17, y);     /* Generate noise signal */
Add1D (x, y, n, x);       /* Noisy Pulse */
PulseParam (x, n, &amp, &amp90, &amp50, &amp10, &top, &base, &topOvershoot,
           &baseOvershoot, &delay, &width, &riseTime, &fallTime, &slewRate);
```

QScale1D

`int` **status** = **QScale1D** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`, `double` ***scale**);

Purpose

Scales the input array by its maximum absolute value. The i^{th} element of the scaled array can be obtained using the following formula.

$$y_i = x_i \,/\, scale$$

where **scale** is the maximum absolute value in the input array. The constant **scale** is determined by the function.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
Output	**y**	double-precision array	scaled array
	scale	double-precision	scaling constant

Return Value

status	integer	refer to error codes in Appendix A

QScale2D

`int` **status** = **QScale2D** (`void` ***x**, `int` **n**, `int` **m**, `void` ***y**, `double` ***scale**);

Purpose

Scales a 2D input array by its maximum absolute value. The (i^{th}, j^{th}) element of the scaled array can be obtained using the following formula.

$$y_{i,j} = x_{i,j} \,/\, scale$$

where **scale** is the maximum absolute value of the input array. The constant **scale** is determined by the function.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
Output	**y**	double-precision 2D array	scaled array
	scale	double-precision	scaling constant

Return Value

status	integer	refer to error codes in Appendix A

Ramp

`int` **status** = **Ramp** (`int` **n**, `double` **first**, `double` **last**, `double` **rampvals**`[]`);

Purpose

Generates an output array representing a ramp pattern. The i^{th} element of the output array is obtained using the formula.

$$rampvals_i = first + i\Delta x$$

where Δx = (**last** - **first**) / (**n**-1).

Parameters

Input	**n**	integer	number of samples
	first	double-precision	initial ramp value
	last	double-precision	final ramp value
Output	**rampvals**	double-precision array	ramp array

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The value of **last** does not have to be greater than the value of **first**. If the condition **last** < **first** is met, then a negatively sloped ramp pattern is generated.

Example

```
/* The following code generates the pattern–rampvals = { -5.0, -4.0, -3.0, -
2.0, -1.0, 0.0, 1.0, 2.0, 3.0, 4.0, 5.0 }. */
double  rampvals[11], first, last;
int  n;
n = 11;
first = -5.0;
last = 5.0;
Ramp (n, first, last, rampvals);
```

RatInterp

`int` **status** = **RatInterp** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **x_val**, `double` ***Interp_Val**, `double` ***Error**);

Purpose

Returns the value of a particular rational function P(x)/Q(x) passing through the n points (x_i, $f(x_i)$) at **x_val**,

given a set of n points (x_i, $f(x_i)$) in the plane where f is some function, and
a value **x_val** at which f is to be interpolated; and
where P and Q are polynomials, and
n is the number of elements in **x**.

The function P(x)/Q(x) is the unique rational function that passes through the given points and satisfies the following conditions.

If **n** is odd,

$$\deg(P) = \deg(Q) = (n-1)/2,$$

if **n** is even,

$$\deg(Q) = n/2$$

$$\deg(P) = n/2 - 1$$

where deg( ) is the order of the polynomial function.

Parameters

Input	**x**	1D double-precision array	Values at which the function to be interpolated is known.
	y	1D double-precision array	Function values at the known **x** values.
	n	integer	Number of points in **x** and in **y**.
	x_val	double-precision	Value at which *f* is to be interpolated or extrapolated.
Output	**Interp_Val**	double-precision	Interpolated value at **x_val**.
	Error	double-precision	Estimate of the error in the interpolation.

Using This Function

All input arrays should be the same size. If the value of **x_val** is in the range of **x**, this function performs interpolation. Otherwise, it performs extrapolation. If **x_val** is too far from the range of **x**, **Error** might be large, and it would not produce a satisfactory extrapolation.

Example

```
/* Pick points randomly, pick an x in the range of x-values, run a rational
function through the points and interpolate at x_val. */
double x[10], y[10], Interp_Val, Error, x_val, high, low;
int n, i;
n = 10;
WhiteNoise (n, 5.0, 17, x);
WhiteNoise (n, 5.0, 17, y);
high = x[0];
low = x[0];
for(i=0; i<n; i++) {
   if (x[i] > high) high = x[i];
   if (x[i] < low) low = x[i];
}
x_val = (high + low)/2.0;
RatInterp (x, y, n, x_val, &Interp_Val, &Error);
```

ReFFT

`int` **status** = **ReFFT** (`double` **x**`[]`,double **y**`[]`, int **n**);

Purpose

Computes the Fast Fourier Transform of a real input array.

Parameters

Input	**x**	double-precision array	array to be transformed
	n	integer	number of elements
Output	**x**	double-precision array	real part of Fourier Transform
	y	double-precision array	imaginary part of Fourier Transform

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The number of elements (**n**) must be a power of two. The operation is done in place and the input array **x** is overwritten. The output array **y** must be at least the same size as the input array **x** because performing an FFT on a real array results in a complex sequence.

Example

```
/* Generate an array with random numbers and compute its Fast Fourier
Transform. */
double  x[256], y[256];
int  n;
n = 256;
Uniform (n, x);
ReFFT (x, y, n);
```

ReInvFFT

`int` **status** = **ReInvFFT** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**);

Purpose

Computes the inverse Fast Fourier Transform of a complex sequence that results in a real output array.

Parameters

Input	**x**	double-precision array	real part to be transformed
	y	double-precision array	imaginary part to be transformed
	n	integer	number of elements
Output	**x**	double-precision array	real inverse Fourier Transform

Parameter Discussion

The number of elements (**n**) must be a power of 2. The operation is done in place, and the input array **x** is overwritten. The **y** array is unchanged.

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers. */
/* Compute it’s real inverse Fast Fourier Transform. */
double x[256], y[256];
int  n;
n = 256;
Uniform (n, 17, x);
Uniform (n, 17, y);
ReInvFFT (x, y, n);
```

ResetIIRFilter

`int` **status** = **ResetIIRFilter** (`IIRFilterPtr` **filterInformation**);

Purpose

Sets the reset flag in the filterInfo filter structure, so that the internal filter state information is reset to zero before the next cascade IIR filtering operation.

Parameters

Input	**filterInformation**	IIRFilterPtr	**filterInformation** is the pointer to the filter structure which contains the filter coefficients and the internal filter information. Please refer to the function `AllocIIRFilterPtr` for further information about the filter structure.

Return Value

status	integer	Refer to error codes in Appendix A.

Example

```
/*How to use function ResetIIRFilter   */
double      fs, fl, fh, x[256], y[256];
int      type,order,n;
IIRFilterPtr   filterInfo;
n = 256;
fs = 1000.0;
fl = 200.0;
order = 5;
type = 0;      /* lowpass  */
filterInfo = AllocIIRFilterPtr(type,order);
if(filterInfo!=0) {
   Bw_CascadeCoef(fs,fl,fh, filterInfo);
   Uniform(n,17,x);
   IIRCascadeFiltering(x,n,filterInfo,y);
   Uniform(n,20,x);
   ResetIIRFilter(filterInfo);      /* reset the filter for a new data set. */
   IIRCascadeFiltering(x,n,filterInfo,y);
   FreeIIRFilterPtr(filterInfo);
}
```

Reverse

int **status** = **Reverse** (double **x[]**, int **n**, double **y[]**);

Purpose

Reverses the order of the elements of the input array using the following formula.

$$y_i = x_{n-i-1} \qquad \text{for } i = 0, 1, \ldots, n-1$$

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements
Output	**y**	double-precision array	reversed array

Return Value

status	integer	refer to error codes in Appendix A

RMS

int **status** = **RMS** (double **x[]**, int **n**, double ***rmsval**);

Purpose

Computes the root mean squared (rms) value of the input array. The formula used to find the rms value is as follows.

$$rmsval = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^{\,2}}$$

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements in **x**
Output	**rmsval**	double-precision	root mean squared value

Return Value

status	integer	refer to error codes in Appendix A

SawtoothWave

`int` **status** = **SawtoothWave** (`int` **n**, `double` **amp**, `double` **f**, `double` ***phase**, `double` **x**`[]`);

Purpose

Generates an array containing a sawtooth wave. The output array **x** is generated according to the following formula.

$$x_i = amp * sawtooth(*phase + f*360*i)$$

where

$$sawtooth\ (p) = \begin{cases} \dfrac{p\ modulo\ 360}{180.0} & 0 \le p\ modulo\ 360 < 180 \\ \dfrac{p\ modulo\ 360}{180.0} - 2.0 & 180 \le p\ modulo\ 360 < 360 \end{cases}$$

This function can be used to simulate a continuous acquisition from an sawtooth wave function generator. The unit of the input ***phase** is in degrees, and ***phase** is set to (***phase** + **f***360***n**) modulo 360 before returning.

Parameters

Input	**n**	integer	number of samples to generate.
	amp	double-precision	amplitude of the resulting signal.
	f	double-precision	frequency of the resulting signal in normalized units of cycles/sample.
	phase	double-precision pointer	points to the initial **phase**, in degrees, of the generated signal.
Output	**phase**	double-precision	upon completion of this function, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to this function to simulate a continuous function generator.
	x	double-precision array	contains the generated sawtooth wave signal.

Return Value

status	integer	refer to error codes in Appendix A

Scale1D

`int` **status** = **Scale1D** (`double` **x**`[]`, `int` **n**, `double` **y**`[]`, `double` ***offset**,
`double` ***scale**);

Purpose

Scales the input array. The scaled output array is in the range [-1 : 1]. The i^{th} element of the scaled array can be obtained using the following formulas.

$$y_i = (x_i - offset) / scale$$

$$scale = (max - min) / 2$$

$$offset = min + scale$$

where max and min are the maximum and minimum values in the input array, respectively. The function determines the values of the constants **scale** and **offset**. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
Output	**y**	double-precision array	scaled array
	offset	double-precision	offsetting constant
	scale	double-precision	scaling constant

Return Value

status	integer	refer to error codes in Appendix A

Scale2D

`int` **status** = **Scale2D** (`void` ***x**, `int` **n**, `int` **m**, `void` ***y**, `double` ***offset**, `double` ***scale**);

Purpose

Scales the input array. The scaled output array is in the range [-1 : 1]. The i^{th}, j^{th} element of the scaled array can be obtained using the following formulas.

$$y_{i,j} = (x_{i,j} - offset) / scale$$

$$scale = (max - min) / 2$$

$$offset = min + scale$$

where max and min are the maximum and minimum values in the input array, respectively. The function determines the values of the constants **scale** and **offset**.

The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
Output	**y**	double-precision 2D array	scaled array
	offset	double-precision	offsetting constant
	scale	double-precision	scaling constant

Return Value

status	integer	refer to error codes in Appendix A

ScaledWindow

`int` **status** = **ScaledWindow** (`double` **x**`[]`, `int` **n,** `int` **windowType**,
`WindowConst` ***windowConstants**);

Purpose

Applies a scaled window to the time-domain signal and outputs window constants for further analysis.

The windowed time-domain signal is scaled so that when the power or amplitude spectrum of the windowed waveform is computed, all windows provide the same level within the accuracy constraints of the window. This function also returns important window constants for the selected window. These constants are useful when you use functions that perform computations on the power spectrum, such as the PowerFrequencyEstimate.

windowType has the following values.

0: Uniform
1: Hanning
2: Hamming
3: Blackman-Harris
4: Exact Blackman
5: Blackman
6: Flattop
7: Four Term Blackman-Harris
8: Seven Term Blackman-Harris

x is the time-domain signal multiplied by the scaled window.

windowConstants is a structure containing the following important constants for the selected window. windowStruct is defined by the following C typedef statement.

```
typedef struct {
          double enbw;
          double coherentgain;
          } WindowConst;
```

enbw is the equivalent noise bandwidth of the selected window. You can use this value to compute the power in a given frequency span.

coherentgain is the peak gain of the window, relative to the peak gain of the Rectangular window. You can use this value to normalize peak signal gains to that of the Rectangular window.

Parameters

Input	**x**	double-precision 1D array	Input array containing time-domain signal to be windowed.
	n	integer	The number of elements in the input array.
	windowType	integer	The type of the window function to apply to the input signal.
Output	**x**	double-precision array	The windowed version of **x**.
	windowConstants	WindowConst pointer	A structure containing the following useful constants for the selected window: *enbw* is the equivalent noise bandwidth of the selected window. You can use this value to compute the power in a given frequency span. *coherentgain* is the peak gain of the window, relative to the peak gain of the Uniform window. This value is used to normalize peak signal gains to that of the Uniform window.

Return Value

status	integer	refer to error codes in Appendix A

Set1D

`int` **status** = **Set1D** (`double` **x**`[]`, `int` **n**, `double` **a**);

Purpose

Sets the elements of the **x** array to a constant value.

Parameters

Input	**n**	integer	number of elements in **x**
	a	double-precision	constant value
Output	**x**	double-precision array	result array (set to the value of **a**)

Return Value

status	integer	refer to error codes in Appendix A

Shift

`int` **status** = **Shift** (`double` **x**`[]`, `int` **n**, `int` **shifts**, `double` **y**`[]`);

Purpose

Shifts the elements of the input array using the following formula.

$$y_i = x_{i-shifts}$$

The number of **shifts** specified can be in the positive (right) or negative (left) direction.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	shifts	integer	number of shifts
Output	**y**	double-precision array	shifted array

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

This is not a circular shift. Shifted values are not retained, and the trailing portion of the shift is replaced with zero. The operation cannot be done in place; that is, the input and output arrays cannot be the same.

Example

```
/* Generate an array with random numbers and shift it by 20 samples. */
double  x[200], y[200];
int  n;
int  shifts;
n = 200;
```

```
shifts = 20;
Uniform (n, 17, x);
Shift (x, n, shifts, y);
```

Sinc

`int` **status** = **Sinc** (`int` **n**, `double` **amp**, `double` **delay**, `double` **dt**, `double` **x**`[]`);

Purpose

Generates an array containing a sinc pattern. The output array **x** is generated according to the following formula.

$$x_i = amp * Sinc(i * dt - delay)$$

where

$$Sinc(x) = \frac{\sin(\pi x)}{\pi x}$$

Parameters

Input	**n**	integer	The number of samples to generate.
	amp	double-precision	The amplitude of the resulting signal.
	delay	double-precision	Shifts the peak value of the sinc pattern to the index.
	dt	double-precision	The sampling interval. It is inversely proportional to the width of the main lobe of the generated sinc pattern.
Output	**x**	double-precision array	Contains the generated sinc pattern.

Return Value

status	integer	refer to error codes in Appendix A

SinePattern

`int` **status** = **SinePattern** (`int` **n**, `double` **amp**, `double` **phase**, `double` **cycles**, `double` **sine**`[]`);

Purpose

Generates an output array with a sinusoidal pattern. The i[th] element of the double-precision output array is obtained using the following formula.

$$sine_i = amp * \sin(2\pi * i * cycles / n + \pi * phase / 180)$$

The **phase** value is assumed to be in degrees and not in radians.

Parameters

Input	**n**	integer	number of samples
	amp	double-precision	amplitude
	phase	double-precision	phase (in degrees)
	cycles	double-precision	number of cycles
Output	**sine**	double-precision array	sinusoidal pattern

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates a cosinusoidal pattern. */
double  x[8], amp, phase, cycles;
int  n;
n = 8;
amp = 1.0;
phase = 90.0;
cycles = 1.5;
SinePattern (n, amp, phase, cycles, x);
```

SineWave

`int` **status = SineWave** (`int` **n**, `double` **amp**, `double` **f**, `double` ***phase**, `double` **x[]**);

Purpose

Generates an array containing a sine wave. The output array **x** is generated according to the following formula.

$\mathbf{x}_i = \mathbf{amp} * \sin(ph_i)$

where

$$ph_i = \frac{\pi}{180}(*\textbf{phase} + \textbf{f}*360.0*i)$$

where

f *= frequency, cycles/sample*

This function can be used to simulate a continuous acquisition from a sine wave function generator. The unit of the input **phase** is in degrees, and **phase** is set to (***phase** + **f***360***n**) modulo 360 before returning.

Parameters

Input	**n**	integer	The number of samples to generate.
	amp	double-precision	The amplitude of the resulting signal.
	f	double-precision	The frequency of the resulting signal in normalized units of cycles/sample.
	phase	double-precision pointer	Points to the initial **phase**, in degrees, of the generated signal.
Output	**phase**	double-precision	Upon completion of this function, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to this function to simulate a continuous function generator.
	x	double-precision array	Contains the generated sine wave signal.

Return Value

status	integer	refer to error codes in Appendix A

Sort

`int` **status** = **Sort** (`double` **x**`[]`, `int` **n**, `int` **direction**, `double` **y**`[]`);

Purpose

Sorts the **x** input array in ascending or descending order. The operation can be performed in place; that is, **x** and **y** can be the same array.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements to be sorted
	direction	integer	zero: ascending nonzero: descending
Output	**y**	double-precision array	sorted array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random array of numbers and sort them in ascending order. */
double  x[200], y[200];
int  n;
int  dir;
n = 200;
dir = 0;
Uniform (n, 17, x);
Sort (x, n, dir, y);
```

Spectrum

`int` **status = Spectrum** (`double` **x**`[]`, `int` **n**);

Purpose

Computes the power spectrum of the input real data. The operation is done in place and the input array **x** is overwritten. The following formula is used to obtain the power spectrum.

$$Power\ Spectrum = |\ FFT\ \{X\}\ |^2 / n^2$$

The number of elements (**n**) must be a power of two.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
Output	**x**	double-precision array	power spectrum

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate an array with random numbers and compute its power spectrum. */
double  x[256];
int  n;
n = 256;
Uniform (n, 17, x);
Spectrum (x, n);
```

SpectrumUnitConversion

`int` **status = SpectrumUnitConversion** (`double` **spectrum**`[]`, `int` **n,** `int` **type**,
`int` **scalingMode**, `int` **displayUnits**, `double` **df**,
`WindowConst` **windowConstants**,
`double` **convertedSpectrum**`[]`, `char` **unitString**`[]`);

Purpose

Converts the input **spectrum** (power, amplitude, or gain) to alternate formats including Log (dB or dBm) and spectral density.

spectrum is the input array containing a spectrum of the type specified by the **type** selector.

type = 0 Power ($Vrms^2$)–computed by `AutoSpectrum()`
type = 1 Amplitude (Vrms)–computed by `AmpPhaseSpectrum()`
type = 2 Gain (amplitude ratio)–computed by `TransferFunction()`

The **unitString** is a character array that specifies the base unit of the time domain waveform from which the input **spectrum** is computed. The signal unit is often set to "V" (volts). The size of **unitString** must be at least 12+ size of (input **unitString**).

The **scalingMode** control has three selections for the output unit type.

scalingMode = 0 Linear
scalingMode = 1 dB
scalingMode = 2 dBm

displayUnit has the following selections for the display unit (assuming V for the base unit).

0: Vrms (volts rms)
1: Vpk (volts peak)
2: $Vrms^2$ (volts squared rms)
3: Vpk^2 (volts squared peak)
4: $Vrms/\sqrt{Hz}$ (volts rms per root Hz)
5: $Vpk/\sqrt{Hz}$ (volts peak per root Hz)
6: $Vrms^2/Hz$ (volts squared rms per Hz)
7: Vpk^2/Hz (volts squared peak per Hz)

The last four selections are amplitude spectral density (4,5) and power spectral density (6,7). The structure **windowConstants** contains constants for the selected window (from the `ScaledWindow` function). You need this input only when you use the spectral density output formats (the last four display unit selections).

Parameters

Input	**spectrum**	double-precision array	The input array containing a spectrum of the type specified by the **spectrum** selector. It should be a power, amplitude, or gain spectrum.
	n	integer	The number of elements in the input spectrum.
	type	integer	The type of the input spectrum. Valid values of Type are: 0: Power ($Vrms^2$) 1: Amplitude (Vrms) 2: Gain (amplitude ratio)
	scalingMode	integer	The type of the scaling of the output spectrum. Valid values of Scaling Mode are: 0: Linear 1: dB 2: dBm
	displayUnits	integer	The unit of the output spectrum, (assuming "V" for the input Unit String) Valid values of **displayUnits** are: 0: *Vrms* (volts rms) 1: *Vpk* (volts peak) 2: $Vrms^2$ (volts square rms) 3: Vpk^2 (volts squared peak) 4: $Vrms/\sqrt{Hz}$ (volts rms per square root of Hz) 5: $Vpk/\sqrt{Hz}$ (volts peak per square root of Hz) 6: $Vrms^2/Hz$ (volts squared rms per Hz) 7: Vpk^2/Hz (volts squared peak per Hz)

(continues)

Parameters (Continued)

	df	double-precision	The frequency interval, in hertz, as output by the following functions: `AmpPhaseSpectrum`, `AutoPowerSpectrum`, `CrossPowerSpectrum`, `NetworkFunctions`, `TransferFunction`
	windowConstants	WindowConst pointer	A structure containing the following useful constants for the selected window: *enbw* is the equivalent noise bandwidth of the selected window. You can use this value to compute the power in a given frequency span. *coherentgain* is the peak gain of the window, relative to the peak gain of the Rectangular window. This value is used normalize peak signal gains to that of the Rectangular window. This structure is output by the `ScaledWindow` function.
	unitString	string	A string that contains, on input, the base unit of the analyzed signal (V for a voltage signal).
Output	**convertedSpectrum**	double-precision array	The input spectrum (power, amplitude, or gain) converted to alternate formats including Log (dB or dBm) and spectral density. The size of this array must be at n.
	unitString	string	Contains, upon completion of this function, the unit of the output Converted Spectrum. The size of this string must be at least the (size of the input **unitString**) + 12.

Return Value

status	integer	refer to error codes in Appendix A

SpInterp

`int` **status = SpInterp** (`double` **x**`[]`, `double` **y**`[]`, `double` **y2**`[]`, `int` **n**, `double` **x_val**, `double` ***Interp_Val**);

Purpose

Performs a cubic spline interpolation of the function f at a value **x_val** (where **x_val** is in the range of x_i's), given a tabulated function of the form $y_i = f(x_i)$ for $i = 0, 1, ..., n-1$, with $x_i < x_{i+1}$, and given the second derivatives that specify the interpolant at the **n** nodes of **x** (these are supplied by the `Spline` procedure). If **x_val** falls in the interval $[x_i, x_{i+1}]$, then the interpolated value is as follows.

$$\text{Interp\_Val} = Ay_i + By_{i+1} + Cy''_i + Dy''_{i+1}$$

where

$$A = \frac{x_{i+1} - x\_val}{x_{i+1} - x_i}$$

$$B = 1 - A$$

$$C = (A^3 - A)(x_{i+1} - x_i)^2/6$$

$$D = (B^3 - B)(x_{i+1} - x_i)^2/6$$

Parameters

Input	**x**	double-precision array	the *x* values at which *f* is known. These values must be in ascending order.
	y	double-precision array	the function values $y_i = f(x_i)$
	y2	double-precision array	the array of second derivatives which specify the interpolant
	n	integer	the number of elements in **x**, **y**, and **y2**
	x_val	double-precision	the *x* value at which *f* is to be interpolated
Output	**Interp_Val**	double-precision	the interpolated value

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Choose ascending X-values.  Pick corresponding Y-values randomly.  Set
boundary conditions and specify the cubic spline interpolant that is run
through the points.  Pick an x in range of X's and interpolate.  Pick another
x and interpolate again. */
double X[100], Y[100], Y2[100], B1, B2, x_val;
int n, i;
n = 100;
for(i=0; i<n; i++)
   X[i] = i * 0.1;
WhiteNoise (n, 5.0, 17, Y);
b1=0.0;
b2=0.0;
Spline (X, Y, n, b1, b2, Y2);
x_val = 0.331;
SpInterp (X, Y, Y2, n, x_val, &Interp_Val);

x_val = 0.7698;
SpInterp (X, Y, Y2, n, x_val, &Interp_Val);
```

Spline

`int` **status** = **Spline** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **b1**, `double` **b2**, `double` **y2**`[]`);

Purpose

Calculates the second derivatives used by the cubic spline interpolant (the continuously differentiable curve to be run though the **n** points (x_i, y_i)), given a tabulated function of the form $y_i = f(x_i)$ for $i = 0, 1, ..., n-1$, with $x_i < x_i + 1$, and given the boundary conditions **b1** and **b2** such that the interpolant's second derivative matches the specified values at x_0 and x_{n-1}.

This array can be used with the `SpInterp` function to calculate an interpolation value.

Parameters

Input	**x**	double-precision array	the x values at which f is known; these values must be in ascending order
	y	double-precision array	the function values $y_i = f(x_i)$
	n	integer	the number of elements in **x**, **y**, and **y2**
	b1	double-precision	the first boundary condition (x''_0)
	b2	double-precision	the second boundary condition(x''_{n-1})
Output	**y2**	double-precision array	the array of second derivatives that specify the interpolant

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Choose ascending X-values.  Pick corresponding Y-values randomly.  Set
boundary conditions and specify the cubic spline interpolant that is run
through the points. */
double X[100], Y[100], Y2[100], b1, b2;
int n, i;
n = 100;
for(i=0; i<n; i++)
   X[i] = i * 0.1;
WhiteNoise (n, 5.0, 17, Y);
b1=0.0;
b2=0.0;
Spline (X, Y, n, b1, b2, Y2);
```

SquareWave

`int` **status** = **SquareWave** (`int` **n**, `double` **amp**, `double` **f**, `double` ***phase,**
`double` **dutyCycle**, `double` **x**`[]`);

Purpose

Generates an array containing a square wave. The output array **x** is generated according to the following formula.

$$x_i = amp * square(*phase + f + 360.0 * i)$$

where
f = frequency, cycles/sample

$$\text{square}(p) = \begin{cases} 1.0 & 0 \le p \text{ modulo } 360 < \frac{\text{duty}}{100} * 360 \\ -1.0 & \frac{\text{duty}}{100} * 360 \le p \text{ modulo } 360 < 360 \end{cases}$$

This function can be used to simulate a continuous acquisition from a square wave function generator. The unit of the input ***phase** is in degrees, and ***phase** is set to (***phase** + **f** *360***n**) modulo 360 before returning.

Parameters

Input	**n**	integer	The number of samples to generate.
	amp	double-precision	The amplitude of the resulting signal.
	f	double-precision	The frequency of the resulting signal in normalized units of cycles/sample.
	dutyCycle	double-precision	Contains the duty cycle, in percent, of the generated square wave signal.
	phase	double-precision	Points to the initial **phase**, in degrees, of the generated signal.
Output	**phase**	double-precision	Upon completion of this function, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to this function to simulate a continuous function generator.
	x	double-precision array	Contains the generated square wave signal.

Return Value

status	integer	refer to error codes in Appendix A

StdDev

`int` **status** = **StdDev** (`double` **x**`[]`, `int` **n**, `double` *__meanval__, `double` *__sDev__);

Purpose

Computes the standard deviation and the mean (average) values of the input array. The formulas used to find the mean and the standard deviation are as follows.

$$meanval = \sum_{i=0}^{n-1} x_i / n$$

$$sDev = \sqrt{\sum_{i=0}^{n-1} [x_i - meanval]^2 / n}$$

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
Output	**meanval**	double-precision	mean value
	sDev	double-precision	standard deviation

Return Value

status	integer	refer to error codes in Appendix A

Sub1D

`int` **status** = **Sub1D** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **z**`[]`);

Purpose

Subtracts two 1D arrays. The i^{th} element of the output array can be obtained using the following formula.

$$z_i = x_i - y_i$$

The operation can be performed in place; that is, **z** can be either **x** or **y**.

Parameters

Input	**x** **y** **n**	double-precision array double-precision array integer	**x** input array **y** input array number of elements to be subtracted
Output	**z**	double-precision array	result array

Return Value

status	integer	refer to error codes in Appendix A

Sub2D

`int` **status** = **Sub2D** (`void` ***x**, `void` ***y**, `int` **n**, `int` **m**, `void` ***z**);

Purpose

Subtracts two 2D arrays. The (i^{th}, j^{th}) element of the output array is obtained using the formula.

$$z_{i,j} = x_{i,j} - y_{i,j}$$

The operation can be performed in place; that is, **z** can be either **x** or **y**.

Parameters

Input	**x** **y** **n** **m**	double-precision 2D array double-precision 2D array integer integer	**x** input array **y** input array number of elements in first dimension number of elements in second dimension
Output	**z**	double-precision 2D array	result array

Return Value

status	integer	refer to error codes in Appendix A

Subset1D

`int` **status** = **Subset1D** (`double` **x**`[]`, `int` **n**, `int` **index**, `int` **length**, `double` **y**`[]`);

Purpose

Extracts a subset of the **x** input array containing the number of elements specified by the **length** and starting at the **index** element.

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements in **x**
	index	integer	initial **index** for the subset
	length	integer	number of elements copied to the subset
Output	**y**	double-precision array	subset array

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following example generates y ={0.0, 1.0, 2.0, 3.0 }*/
double  x[11], y[4], first, last;
int  n, index, length;
n = 11;
index = 5;
length = 4;
first = -5.0;
last = 5.0;
Ramp (n, first, last, x);
Subset1D (x, n, index, length, y);
```

Sum1D

`int` **status** = **Sum1D** (`double` **x**`[]`, `int` **n**, `double` ***sum**);

Purpose

Finds the **sum** of the elements of the input array. The formula used to obtain the **sum** of the elements is as follows.

$$sum = \sum_{i=0}^{n-1} x_i$$

Parameters

Input	**x**	double-precision array	input array
	n	integer	number of elements
Output	**sum**	double-precision	sum of elements

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* Generate a random array and sum the elements. */
double  x[20], sum;
int  n;
n = 20;
Uniform (n, 17, x);
Sum1D (x, n, &sum);
```

Sum2D

`int` **status** = **Sum2D** (`void` ***x**, `int` **n**, `int` **m**, `double` ***sum**);

Purpose

Finds the **sum** of the elements in the input 2D array. The **sum** is obtained using the following formula.

$$sum = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_{i,j}$$

Parameters

Input	**x**	double-precision 2D array	input array
	n	integer	number of elements in first dimension
	m	integer	number of elements in second dimension
Output	**sum**	double-precision	sum of the elements

Return Value

status	integer	refer to error codes in Appendix A

T_Dist

`int` **status** = **T_Dist** (`double` **t**, `int` **n**, `double` ***p**);

Purpose

Computes the one-sided probability **p**:

$$p = prob(T \leq t)$$

where T is a random variable from the T-distribution with **n** degrees of freedom.

Parameters

Input	**t**	double-precision	$-\infty < t < \infty$
	n	integer	degrees of freedom
Output	**p**	double-precision	probability ($0 \leq p < 1$)

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double t, p;
int n;
t = -123.456;
n = 6;
T_Dist (t, n, &p);
```

ToPolar

`int` **status** = **ToPolar** (`double` **x**, `double` **y**, `double` ***mag**, `double` ***phase**);

Purpose

Converts the rectangular coordinates (**x**, **y**) to polar coordinates (**mag**, **phase**). The formulas used to obtain the polar coordinates are as follows.

$$mag = \sqrt{x^2 + y^2}$$

$$phase = arctan\,(y / x)$$

The **phase** value is in the range of [-π to π].

Parameters

Input	**x**	double-precision	**x** coordinate
	y	double-precision	**y** coordinate
Output	**mag**	double-precision	magnitude
	phase	double-precision	phase (in radians)

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/*Convert the rectangular coordinates to polar coordinates. */
double  x, y, mag, phase;
x = 1.5;
y = -2.5;
ToPolar (x, y, &mag, &phase);
```

ToPolar1D

`int` **status** = **ToPolar1D** (`double` **x**`[]`, `double` **y**`[]`, `int` **n**, `double` **mag**`[]`, `double` **phase**`[]`);

Purpose

Converts the set of rectangular coordinate points (**x**, **y**) to a set of polar coordinate points (**mag**, **phase**). The i^{th} element of the polar coordinate set is obtained using the following formulas.

$$mag_i = \sqrt{x_i^{\,2} + y_i^{\,2}}$$

$$phase_i = \arctan(y_i \;/\; x_i)$$

The **phase** value is in the range of [-π to π].

The operations can be performed in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input	**x**	double-precision array	**x** coordinate
	y	double-precision array	**y** coordinate
	n	integer	number of elements
Output	**mag**	double-precision array	magnitude
	phase	double-precision array	phase (in radians)

Return Value

status	integer	refer to error codes in Appendix A

ToRect

`int` **status** = **ToRect** (`double` **mag**, `double` **phase**, `double` ***x**, `double` ***y**);

Purpose

Converts the polar coordinates (**mag**, **phase**) to rectangular coordinates (**x**, **y**). The formulas used to obtain the rectangular coordinates are as follows.

$$x = mag * cos(phase)$$

$$y = mag * sin(phase)$$

Parameters

Input	**mag**	double-precision	magnitude
	phase	double-precision	phase (in radians)
Output	**x**	double-precision	**x** coordinate
	y	double-precision	**y** coordinate

Return Value

status	integer	refer to error codes in Appendix A

ToRect1D

`int` **status** = **ToRect1D** (`double` **mag**`[]`, `double` **phase**`[]`, `int` **n**, `double` **x**`[]`, `double` **y**`[]`);

Purpose

Converts the set of polar coordinate points (**mag**, **phase**) to a set of rectangular coordinate points (**x**, **y**). The i^{th} element of the rectangular set is obtained using the following formulas.

$$x_i = mag_i * cos(phase_i)$$

$$y_i = mag_i * sin(phase_i)$$

The operations can be performed in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input	**mag** **phase** **n**	double-precision array double-precision array integer	magnitude phase (in radians) number of elements
Output	**x** **y**	double-precision array double-precision array	**x** coordinate **y** coordinate

Return Value

status	integer	refer to error codes in Appendix A

Trace

`int` **status** = **Trace** (`void` ***x**, `int` **n**, `double` ***traceval**);

Purpose

Finds the trace of the 2D input matrix **x**. The trace is the sum of the matrix elements along the main diagonal. The trace is obtained using the following formula.

$$trace = \sum_{i=0}^{n-1} x_{i,i}$$

The input matrix must be an **n** by **n** square matrix.

Parameters

Input	**x** **n**	double-precision 2D array integer	input matrix size of matrix
Output	**traceval**	double-precision	trace

Return Value

status	integer	refer to error codes in Appendix A

TransferFunction

`int` **status = TransferFunction** (`double` **stimulus**`[]`, `double` **response**`[]`, `int` **n,** `double` **dt**, `double` **magHf**`[]`**,** `double` **phaseHf**`[]`**,** `double` *__df__);

Purpose

Computes the single-sided transfer function (also known as the frequency response) from the time-domain stimulus signal and the time-domain response signal of a network under test.

The transfer function is computed as follows.

FFT(**response**) / FFT(**stimulus**)

and then this result is transformed to single-sided magnitude and phase.

Parameters

Input	**stimulus**	double-precision array	Contains the time-domain signal, usually the network stimulus.
	response	double-precision array	Contains the time-domain signal, usually the network response.
	n	integer	The number of elements in the input **stimulus** and **response** arrays. Valid Values: Powers of 2.
	dt	double-precision	The sample period of the time-domain signals, usually in seconds. **dt** = $1/_{fs}$, where fs is the sampling frequency of the time-domain signals.
Output	**magHf**	double-precision array	The magnitude of the averaged single-sided transfer function between the **stimulus** and **response** signals. This array must be at least n/2 elements long.
	phaseHf	double-precision array	The phase, in radians of the averaged single-sided transfer function between the **stimulus** and **response** signals. This array must be at least n/2 elements long.
	df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. *__df__ = 1/(**n***__dt__).

Return Value

status	integer	refer to error codes in Appendix A

Transpose

`int` **status** = **Transpose** (`void` ***x**, `int` **n**, `int` **m**, `void` ***y**);

Purpose

Finds the transpose of a 2D input matrix. The (i^{th}, j^{th}) element of the resulting matrix is given by the following formula.

$$y_{i,j} = x_{j,i}$$

Parameters

Input	**x**	double-precision 2D array	input matrix
	n	integer	size of first dimension
	m	integer	size of second dimension
Output	**y**	double-precision 2D array	transpose matrix

Note: ***If the input matrix is dimensioned (*n *by* m*), then the output matrix must be dimensioned (*m *by* n*).***

Return Value

status	integer	refer to error codes in Appendix A

Triangle

`int` **status** = **Triangle** (`int` **n**, `double` **amp**, `double` **tri**`[]`);

Purpose

Generates an output array that has a triangular pattern. The i^{th} element of the double-precision output array is obtained using the following formulas.

$$tri_i = amp\ (1 - |\ 2i - n\ |\ /\ n\ ) \qquad \text{if } \mathbf{n} \text{ is even}$$

$$tri_i = amp\ (1 - |\ 2i - n + 1\ |\ /\ (n - 1)\ ) \qquad \text{if } \mathbf{n} \text{ is odd}$$

Parameters

Input	**n** **amp**	integer double-precision	number of samples amplitude
Output	**tri**	double-precision array	triangular pattern

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates the pattern  tri = { 0.0, 1.0, 2.0, 3.0, 4.0,
3.0, 2.0, 1.0 }. */
double  tri[8], amp;
int  n;
n = 8;
amp = 4.0;
Triangle (n, amp, tri);
```

TriangleWave

`int` **status** = **TriangleWave** (`int` **n**, `double` **amp**, `double` **f**, `double` ***phase**, `double` **x**`[]`);

Purpose

Generates an array containing a triangle wave. The output array **x** is generated according to the following formula.

$$x_i = \mathbf{amp} * tri(*\mathbf{phase} + \mathbf{f} * 360.0 * i)$$

where

$$\mathbf{f} = frequency,\ cycles/sample$$

$$tri(p) = \begin{cases} 2*((p \text{ modulo } 360)/180) & 0 \le p \text{ modulo } 360 < 90 \\ 2*(1 - (p \text{ modulo } 360)/180) & 90 \le p \text{ modulo } 360 < 270 \\ 2*((p \text{ modulo } 360)/180 - 2) & 270 \le p \text{ modulo } 360 < 360 \end{cases}$$

This function can be used to simulate a continuous acquisition from a triangle wave function generator. The unit of the input ***phase** is in degrees, and ***phase** is set to (***phase** + **f***360***n**) modulo 360 before returning.

Parameters

Input	**n**	integer	The number of samples to generate.
	amp	double-precision	The amplitude of the resulting signal.
	f	double-precision	The frequency of the resulting signal in normalized units of cycles/sample.
	phase	double-precision pointer	Points to the initial **phase**, in degrees, of the generated signal.
Output	**phase**	double-precision	Upon completion of this function, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to this function to simulate a continuous function generator.
	x	double-precision array	Contains the generated triangle wave signal.

Return Value

status	integer	refer to error codes in Appendix A

TriWin

`int` **status** = **TriWin** (`double` **x**`[]`, `int` **n**);

Purpose

Applies a triangular window to the **x** input signal. The triangular window is defined by:

$w_i = 1 - |2*i-n| /n$ for i = 0, 1, ..., n-1

The output signal is obtained by:

$x_i = x_i * w_i$ for i = 0, 1, ..., n-1

The window operation is performed in place. The windowed data **x** replaces the input data **x**.

Parameters

Input	**x** **n**	double-precision array integer	input data number of elements in **x**
Output	**x**	double-precision array	windowed data

Return Value

status	integer	refer to error codes in Appendix A

Uniform

`int` **status** = **Uniform** (`int` **n**, `int` **seed**, `double` **x**`[]`);

Purpose

Generates an array of random numbers that are uniformly distributed between zero and one.

Parameters

Input	**n** **seed**	integer integer	number of samples seed value
Output	**x**	double-precision array	random pattern between 0 and 1

Parameter Discussion

When seed ≥ 0, a new random sequence is generated using the seed value. When seed < 0, the previously generated random sequence continues.

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates an array of random numbers between 0 and 1. */
double  x[20];
int  n;
n = 20;
Uniform (n, 17, x);
```

UnWrap1D

`int` **status = UnWrap1D** (`double` **phase**`[]`, `int` **n**);

Purpose

Unwraps the discontinuous phase values that are in the range from -π to π to create continuous values. The input array, **phase**, is overwritten.

Parameters

Input	**phase**	double-precision array	array of discontinuous phase values
	n	integer	number of elements
Output	**phase**	double-precision array	array of continuous phase values

Return Value

status	integer	refer to error codes in Appendix A

Variance

`int` **status = Variance** (`double` **x**`[]`, `int` **n**, `double` ***meanval**, `double` ***var**);

Purpose

Computes the variance and the mean (average) values of the input array. The following formulas are used to find the mean and the variance.

$$meanval = \sum_{i=0}^{n-1} x_i / n$$

$$var = \sum_{i=0}^{n-1} \left[x_i - meanval\right]^2 / n$$

Parameters

Input	**x** **n**	double-precision array integer	input array number of elements in **x**
Output	**meanval** **var**	double-precision double-precision	mean value variance

Return Value

status	integer	refer to error codes in Appendix A

WhiteNoise

`int` **status** = **WhiteNoise** (`int` **n**, `double` **amp**, `int` **seed**, `double` ***noise**`[]`);

Purpose

Generates an array of random numbers that are uniformly distributed between **-amp** and **amp**.

Parameters

Input	**n** **amp** **seed**	integer double-precision integer	number of samples amplitude seed value
Output	**noise**	double-precision array	noise pattern

Parameter Discussion

When seed ≥ 0, a new random sequence is generated using the seed value. When seed < 0, the previously generated random sequence continues.

Return Value

status	integer	refer to error codes in Appendix A

Example

```
/* The following code generates an array of random numbers between -5 and 5.
*/
double  x[20], amp;
int  n;
n = 20;
amp = 5.0;
WhiteNoise (n, amp, 17, x);
```

Wind_BPF

`int` **status** = **Wind_BPF** (`double` **fs**, `double` **fl**, `double` **fh**, `int` **n**, `double` **coef**`[]`,
`int` **windType**);

Purpose

Designs a digital bandpass FIR linear phase filter using a windowing technique. Five windows are available. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	n	integer	number of filter coefficients
	windType	integer	window type
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The parameter **windType** selects one of the five windows as shown in the following table.

windType	Window	Attenuation (dB)	Transition Bandwidth (fs/n)
1	Rectangular	21	0.9
2	Triangular	25	1.18
3	Hanning	44	2.5
4	Hamming	53	3.13
5	Blackman	74	4.6

Using This Function

The attenuation value determines the approximate peak value of the sidelobes. Transition bandwidth determines a frequency range over which the filter response changes from the pass band to the stop band or from the stop band to the pass band. For more information, refer to *Digital Signal Processing* by Oppenheim and Schafer.

Example

```
/* Design a 55-point bandpass FIR linear phase filter that can achieve at
least a 44 dB attenuation and filter the incoming signal with the designed
filter. */
double  x[256], coef[55], y[310], fs, fl, fh;
int  n, m, windType;
fs = 1000.0;            /* sampling frequency */
fl = 200.0;             /* desired lower cutoff frequency */
fh = 300.0;             /* desired higher cutoff frequency */
                        /* pass band is from 200.0 to 300.0 */
n = 55;                 /* filter length */
windType = 3;           /* using Hanning window */
m = 256;
Wind_BPF (fs, fl, fh, n, coef, windType);
Convolve (coef, n, x, m, y);  /* convolve the filter with the signal */
```

Wind_BSF

`int` **status** = **Wind_BSF** (`double` **fs**, `double` **fl**, `double` **fh**, `int` **n**, `double` **coef**`[]`,
`int` **windType**);

Purpose

Designs a digital bandstop FIR linear phase filter using a windowing technique. Five windows are available. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fl	double-precision	lower cutoff frequency
	fh	double-precision	higher cutoff frequency
	n	integer	number of filter coefficients
	windType	integer	window type
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The parameter **windType** selects one of the five windows as shown in the following table.

windType	Window	Attenuation (dB)	Transition Bandwidth (fs/n)
1	Rectangular	21	0.9
2	Triangular	25	1.18
3	Hanning	44	2.5
4	Hamming	53	3.13
5	Blackman	74	4.6

Using This Function

The attenuation value determines the approximate peak value of the sidelobes. Transition bandwidth determines a frequency range over which the filter response changes from the pass band to the stop band or from the stop band to the pass band. For more information, refer to *Digital Signal Processing* by Oppenheim and Schafer.

Example

```
/* Design a 55-point bandstop FIR linear phase filter that can achieve at
least a 44 dB attenuation and filter the incoming signal with the designed
filter. */
double  x[256], coef[55], y[310], fs, fl, fh;
int  n, m, windType;
fs = 1000.0;            /* sampling frequency */
fl = 200.0;             /* desired lower cutoff frequency */
fh = 300.0;             /* desired higher cutoff frequency */
                        /* stop band is from 200.0 to 300.0 */
n = 55;                 /* filter length */
windType = 3;           /* using Hanning window */
```

```
m = 256;
Wind_BSF (fs, fl, fh, n, coef, windType);
Convolve (coef, n, x, m, y);  /* convolve the filter with*/
                        /* the signal */
```

Wind_HPF

`int` **status** = **Wind_HPF** (`double` **fs**, `double` **fc**, `int` **n**, `double` **coef**`[]`, `int` **windType**);

Purpose

Designs a digital highpass FIR linear phase filter using a windowing technique. Five windows are available. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	n	integer	number of filter coefficients
	windType	integer	window type
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The parameter **windType** selects one of the five windows as shown in the following table.

windType	Window	Attenuation (dB)	Transition Bandwidth (fs/n)
1	Rectangular	21	0.9
2	Triangular	25	1.18
3	Hanning	44	2.5
4	Hamming	53	3.13
5	Blackman	74	4.6

Using This Function

The attenuation value determines the approximate peak value of the sidelobes. Transition bandwidth determines a frequency range over which the filter response changes from the pass band to the stop band or from the stop band to the pass band. For more information, refer to *Digital Signal Processing* by Oppenheim and Schafer.

Example

```
/* Design a 55-point highpass FIR linear phase filter that can achieve at
least a 44 dB attenuation and filter the incoming signal with the designed
filter. */
double  x[256], coef[55], y[310], fs, fc;
int  n, m, windType;
fs = 1000.0;                  /* sampling frequency */
fc = 200.0;                   /* desired cutoff frequency */
n = 55;                       /* filter length */
windType = 3;                 /* using Hanning window */
m = 256;
Wind_HPF (fs, fc, n, coef, windType);
Convolve (coef, n, x, m, y);  /* convolve the filter with */
                              /* the signal */
```

Wind_LPF

`int` **status** = **Wind_LPF** (`double` **fs**, `double` **fc**, `int` **n**, `double` **coef**`[]`, `int` **windType**);

Purpose

Designs a digital lowpass FIR linear phase filter using a windowing technique. Five windows are available. This function generates only the filter coefficients. No filtering of data is actually performed.

Parameters

Input	**fs**	double-precision	sampling frequency
	fc	double-precision	cutoff frequency
	n	integer	number of filter coefficients
	windType	integer	window type
Output	**coef**	double-precision array	filter coefficients

Return Value

status	integer	refer to error codes in Appendix A

Parameter Discussion

The parameter **windType** selects one of the five windows as shown in the following table.

windType	Window	Attenuation (dB)	Transition Bandwidth (fs/n)
1	Rectangular	21	0.9
2	Triangular	25	1.18
3	Hanning	44	2.5
4	Hamming	53	3.13
5	Blackman	74	4.6

Using This Function

The attenuation value determines the approximate peak value of the sidelobes. Transition bandwidth determines a frequency range over which the filter response changes from the pass band to the stop band or from the stop band to the pass band. For more information, refer to *Digital Signal Processing* by Oppenheim and Schafer.

Example

```
/* Design a 55-point lowpass FIR linear phase filter that can achieve at least
a 44 dB attenuation and filter the incoming signal with the designed filter.
*/
double  x[256], coef[55], y[310], fs, fc;
int  n, m, windType;
fs = 1000.0;            /* sampling frequency */
fc = 200.0;             /* desired cutoff frequency */
n = 55;                 /* filter length */
windType = 3;           /* using Hanning window */
m = 256;
Wind_LPF (fs, fc, n, coef, windType);
Convolve (coef, n, x, m, y);  /* convolve the filter with */
                        /* the signal */
```

XX_Dist

`int` **status** = **XX_Dist** (`double` **x**, `int` **n**, `double` ***p**);

Purpose

Approximates the one-sided probability **p**:

$p = prob(X \leq x)$

where X is a random variable from the χ^2-distribution with **n** degrees of freedom.

Parameters

Input	**x** **n**	double-precision integer	$-\infty < x < \infty$ degrees of freedom
Output	**p**	double-precision	probability ($0 \leq p < 1$)

Return Value

status	integer	refer to error codes in Appendix A

Example

```
double x, p;
int n;
x = -123.456;
n = 6;
XX_Dist (x, n, &p);
/* Now p = 0 because χ2 distributed variables are non-negative.*/
```

Appendix A
Error Codes

This appendix contains error codes returned by the Advanced Analysis Library functions. If an error condition occurs during a call to any of the functions in the LabWindows Analysis Library, the return value **status** will contain the returned error code. This code is a value that specifies the type of error that occurred. Table A-2 lists the error codes in numeric order. For your convenience, Table A-1 lists the error codes alphabetically by symbolic name.

Table A-1. Advanced Analysis Library Error Codes, Sorted Alphabetically

Symbolic Name	Code	Error Message
ArraySizeAnlysErr	-20008	The specified conditions on the input arrays have not been met.
AttenGTRippleAnlysErr	-20028	The attenuation must be greater than the ripple amplitude.
AttenGTZeroAnlysErr	-20025	The attenuation must be greater than zero.
BalanceAnlysErr	-20047	The data is unbalanced.
BandSpecAnlysErr	-20023	Invalid band specification.
BaseGETopAnlysErr	-20101	Base must be less than Top.
BetaFuncAnlysErr	-20057	The parameter to the beta function must meet the condition: $0 < p < 1$.
CategoryAnlysErr	-20055	Invalid number of categories or samples.
ColumnAnlysErr	-20051	The first column in the X matrix must be all ones.
CyclesAnlysErr	-20012	The number of cycles must meet the condition: $0 < \text{cycles} \le \text{samples}$.
DataAnlysErr	-20045	The total number of data points must be equal to product of (levels/each factor) * (observations/cell).
DecFactAnlysErr	-20022	The decimating factor must meet the condition: $0 < \text{decimating factor} \le \text{samples}$.
DelayWidthAnlysErr	-20014	The delay and width must meet the condition: $0 \le (\text{delay} + \text{width}) < \text{samples}$.
DimensionAnlysErr	-20058	Invalid number of dimensions or dependent variables.
DistinctAnlysErr	-20049	The x-values must be distinct.

(continues)

Table A-1. Advanced Analysis Library Error Codes (Continued)

Symbolic Name	Code	Error Message
DivByZeroAnlysErr	-20060	Divide by zero.
DtGTZeroAnlysErr	-20016	dt or dx must be greater than zero.
EqRplDesignAnlysErr	-20031	The filter cannot be designed with the specified input parameters.
EqSamplesAnlysErr	-20002	Input sequences must be the same size.
EvenSizeAnlysErr	-20033	The number of coefficients must be odd for this filter.
FactorAnlysErr	-20043	The level of factor is outside the allowable range.
FreedomAnlysErr	-20052	Invalid degrees of freedom.
IndexLengthAnlysErr	-20018	The index and lenght must meet the condition: $0 \leq (index + length) < samples$.
IndexLTSamplesAnlysErr	-20017	The index must meet the condition: $0 \leq index < samples$.
InvSelectionAnlysErr	-20061	Invalid selection.
IIRFilterInfoAnlysErr	-20066	The information in the IIR filter structure is invalid.
LevelsAnlysErr	-20042	The number of levels is outside the allowable range.
MaxIterAnlysErr	-20062	Maximum iteration exceeded.
MixedSignAnlysErr	-20036	The second array must be all positive or negative and nonzero.
ModelAnlysErr	-20048	The Random Effect model was requested when the Fixed Effect model is required.
NoAnlysErr	0	No error; the call was successful.
NyquistAnlysErr	-20020	The cut-off frequency, fc, must meet the condition: $0 \leq fc \leq fs/2$.
ObservationsAnlysErr	-20044	There must be at least one observation.
OddSizeAnlysErr	-20034	The number of coefficients must be even for this filter.
OrderGEZeroAnlysErr	-20103	Order must be greater than or equal to zero.
OrderGTZeroAnlysErr	-20021	The order must be greater than zero..
OutOfMemAnlysErr	-20001	There is not enough memory left to perform the specified routine.

(continues)

Table A-1. Advanced Analysis Library Error Codes (Continued)

Symbolic Name	Code	Error Message
PoleAnlysErr	-20050	The interpolating function has a pole at the requested value.
PolyAnlysErr	-20063	Invalid polynomial.
PowerOfTwoAnlysErr	-20009	The size of the input array must be a valid power of two: $size = 2^m$.
ProbabilityAnlysErr	-20053	The probability must meet the condition: $0 < p < 1$.
RippleGTZeroAnlysErr	-20024	The ripple must be greater than zero.
SamplesGEThreeAnlysErr	-20007	The number of samples must be greater than or equal to three.
SamplesGETwoAnlysErr	-20006	The number of samples must be greater than or equal to two.
SamplesGEZeroAnlysErr	-20004	The number of samples must be greater than or equal to zero.
SamplesGTZeroAnlysErr	-20003	The number of samples must be greater than zero.
ShiftRangeAnlysErr	-20102	The shifts must meet the condition: |shifts| < samples.
SingularMatrixAnlysErr	-20041	The input matrix is singular. The system of equations cannot be solved.
SizeGTOrderAnlysErr	-20037	The array size must be greater than the order.
SquareMatrixAnlysErr	-20040	The input matrix must be a square matrix.
TableAnlysErr	-20056	The contingency table has a negative number.
UpperGELowerAnlysErr	-20019	The upper value must be greater than or equal to the lower value.
ZeroVectorAnlysErr	-20065	The elements of the vector cannot be all zero.

Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically

Symbolic Name	Code	Error Message
NoAnlysErr	0	No error; the call was successful.
OutOfMemAnlysErr	-20001	There is not enough memory left to perform the specified routine.
EqSamplesAnlysErr	-20002	Input sequences must be the same size.
SamplesGTZeroAnlysErr	-20003	The number of samples must be greater than zero.
SamplesGEZeroAnlysErr	-20004	The number of samples must be greater than or equal to zero.
SamplesGETwoAnlysErr	-20006	The number of samples must be greater than or equal to two.
SamplesGEThreeAnlysErr	-20007	The number of samples must be greater than or equal to three.
ArraySizeAnlysErr	-20008	The specified conditions on the input arrays have not been met.
PowerOfTwoAnlysErr	-20009	The size of the input array must be a valid power of two: $\text{size} = 2^m$.
CyclesAnlysErr	-20012	The number of cycles must meet the condition: 0 < cycles ≤ samples.
DelayWidthAnlysErr	-20014	The delay and width must meet the condition: 0 ≤ (delay + width) < samples.
DtGTZeroAnlysErr	-20016	dt or dx must be greater than zero.
IndexLTSamplesAnlysErr	-20017	The index must meet the condition: 0 ≤ index < samples.
IndexLengthAnlysErr	-20018	The index and lenght must meet the condition: 0 ≤ (index + length) < samples.
UpperGELowerAnlysErr	-20019	The upper value must be greater than or equal to the lower value.
NyquistAnlysErr	-20020	The cut-off frequency, fc, must meet the condition: 0 ≤ fc ≤ fs/2.
OrderGTZeroAnlysErr	-20021	The order must be greater than zero.
DecFactAnlysErr	-20022	The decimating factor must meet the condition: 0 < decimating factor ≤ samples.
BandSpecAnlysErr	-20023	Invalid band specification.
RippleGTZeroAnlysErr	-20024	The ripple must be greater than zero.

(continues)

Table A-2. Advanced Analysis Library Error Codes (Continued)

Symbolic Name	Code	Error Message
AttenGTZeroAnlysErr	-20025	The attenuation must be greater than zero.
AttenGTRippleAnlysErr	-20028	The attenuation must be greater than the ripple amplitude.
EqRplDesignAnlysErr	-20031	The filter cannot be designed with the specified input parameters.
EvenSizeAnlysErr	-20033	The number of coefficients must be odd for this filter.
OddSizeAnlysErr	-20034	The number of coefficients must be even for this filter.
MixedSignAnlysErr	-20036	The second array must be all positive or negative and nonzero.
SizeGTOrderAnlysErr	-20037	The array size must be greater than the order.
SquareMatrixAnlysErr	-20040	The input matrix must be a square matrix.
SingularMatrixAnlysErr	-20041	The input matrix is singular. The system of equations cannot be solved.
LevelsAnlysErr	-20042	The number of levels is outside the allowable range.
FactorAnlysErr	-20043	The level of factor is outside the allowable range.
ObservationsAnlysErr	-20044	There must be at least one observation.
DataAnlysErr	-20045	The total number of data points must be equal to product of (levels/each factor) * (observations/cell).
BalanceAnlysErr	-20047	The data is unbalanced.
ModelAnlysErr	-20048	The Random Effect model was requested when the Fixed Effect model is required.
DistinctAnlysErr	-20049	The x-values must be distinct.
PoleAnlysErr	-20050	The interpolating function has a pole at the requested value.
ColumnAnlysErr	-20051	The first column in the X matrix must be all ones.
FreedomAnlysErr	-20052	Invalid degrees of freedom.
ProbabilityAnlysErr	-20053	The probability must meet the condition: $0 < p < 1$.
CategoryAnlysErr	-20055	Invalid number of categories or samples.
TableAnlysErr	-20056	The contingency table has a negative number.

(continues)

Table A-2. Advanced Analysis Library Error Codes (Continued)

Symbolic Name	Code	Error Message
BetaFuncAnlysErr	-20057	The parameter to the beta function must meet the condition: 0 < p < 1.
DimensionAnlysErr	-20058	Invalid number of dimensions or dependent variables.
DivByZeroAnlysErr	-20060	Divide by zero.
InvSelectionAnlysErr	-20061	Invalid selection.
MaxIterAnlysErr	-20062	Maximum iteration exceeded.
PolyAnlysErr	-20063	Invalid polynomial.
ZeroVectorAnlysErr	-20065	The elements of the vector cannot be all zero.
IIRFilterInfoAnlysErr	-20066	The information in the IIR filter structure is invalid.
BaseGETopAnlysErr	-20101	Base must be less than Top.
ShiftRangeAnlysErr	-20102	The shifts must meet the condition: |shifts| < samples.
OrderGEZeroAnlysErr	-20103	Order must be greater than or equal to zero.

Appendix B
Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve technical problems you might have as well as a form you can use to comment on the product documentation. Filling out a copy of the *Technical Support Form* before contacting National Instruments helps us help you better and faster.

National Instruments provides comprehensive technical assistance around the world. In the U.S. and Canada, applications engineers are available Monday through Friday from 8:00 a.m. to 6:00 p.m. (central time). In other countries, contact the nearest branch office. You may fax questions to us at any time.

Electronic Services

Bulletin Board Support

National Instruments has BBS and FTP sites dedicated for 24-hour support with a collection of files and documents to answer most common customer questions. From these sites, you can also download the latest instrument drivers, updates, and example programs. For recorded instructions on how to use the bulletin board and FTP services and for BBS automated information, call (512) 795-6990. You can access these services at:

- United States: (512) 794-5422 or (800) 327-3077
  Up to 14,400 baud, 8 data bits, 1 stop bit, no parity
- United Kingdom: 01635 551422
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity
- France: 1 48 65 15 59
  Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

FaxBack Support

FaxBack is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access FaxBack from a touch-tone telephone at the following number: (512) 418-1111.

FTP Support

To access our FTP site, log on to our Internet host, `ftp.natinst.com`, as anonymous and use your Internet address, such as `joesmith@anywhere.com`, as your password. The support files and documents are located in the /`support` directories.

E-Mail Support (currently U.S. only)

You can submit technical support questions to the appropriate applications engineering team through e-mail at the Internet addresses listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

GPIB:	`gpib.support@natinst.com`
DAQ:	`daq.support@natinst.com`
VXI:	`vxi.support@natinst.com`
LabVIEW:	`lv.support@natinst.com`
LabWindows:	`lw.support@natinst.com`
HiQ:	`hiq.support@natinst.com`
Lookout:	`lookout.support@natinst.com`
VISA:	`visa.support@natinst.com`

Fax and Telephone Support

National Instruments has branch offices all over the world. Use the list below to find the technical support number for your country. If there is no National Instruments office in your country, contact the source from which you purchased your software to obtain support.

	Telephone	**Fax**
Australia	03 9 879 9422	03 9 879 9179
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Canada (Ontario)	519 622 9310	
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	90 527 2321	90 502 2930
France	1 48 14 24 24	1 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	95 800 010 0793	5 520 3282
Netherlands	0348 433466	0348 430673
Norway	32 84 84 00	32 84 86 00
Singapore	2265886	2265887
Spain	91 640 0085	91 640 0533
Sweden	08 730 49 70	08 730 43 70
Switzerland	056 200 51 51	056 200 51 55
Taiwan	02 377 1200	02 737 4644
U.K.	01635 523545	01635 523154

Technical Support Form

Photocopy this form and update it each time you make changes to your software or hardware, and use the completed copy of this form as a reference for your current configuration. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

If you are using any National Instruments hardware or software products related to this problem, include the configuration forms from their user manuals. Include additional pages if necessary.

Name ____

Company ____

Address ____

____

Fax (_____)_________________________ Phone (_____)_____________________________

Computer brand ______________ Model ________________ Processor ___________________

Operating system: Windows 3.1, Windows for Workgroups 3.11, Windows NT 3.1, Windows NT 3.5, Windows 95, other (include version number)____

Clock Speed __________MHz RAM ___________MB Display adapter _________________

Mouse ____yes ____no Other adapters installed _____________________________________

Hard disk capacity ______MB Brand ____

Instruments used ____

National Instruments hardware product model __________________ Revision ____________________

Configuration ____

National Instruments software product ________________________ Version ____________________

Configuration ____

The problem is ____

____

____

____

List any error messages ____

____

____

____

The following steps will reproduce the problem ____

____

____

Hardware and Software Configuration Form

Record the settings and revisions of your hardware and software on the line to the right of each item. Complete a new copy of this form each time you revise your software or hardware configuration, and use this form as a reference for your current configuration. When you complete this form accurately before contacting National Instruments for technical support, our applications engineers can answer your questions more efficiently.

National Instruments Products

Data Acquisition Hardware Revision ______________________________

Interrupt Level of Hardware ______________________________

DMA Channels of Hardware ______________________________

Base I/O Address of Hardware ______________________________

NI-DAQ, LabVIEW, or
LabWindows Version ______________________________

Other Products

Computer Make and Model ______________________________

Microprocessor ______________________________

Clock Frequency ______________________________

Type of Video Board Installed ______________________________

Operating System ______________________________

Operating System Version ______________________________

Operating System Mode ______________________________

Programming Language ______________________________

Programming Language Version ______________________________

Other Boards in System ______________________________

Base I/O Address of Other Boards ______________________________

DMA Channels of Other Boards ______________________________

Interrupt Level of Other Boards ______________________________

Documentation Comment Form

National Instruments encourages you to comment on the documentation supplied with our products. This information helps us provide quality products to meet your needs.

Title: **LabWindows®/CVI Advanced Analysis Library Reference Manual**

Edition Date: **July 1996**

Part Number: **320686C-01**

Please comment on the completeness, clarity, and organization of the manual.

____

If you find errors in the manual, please record the page numbers and describe the errors.

____

Thank you for your help.

Name ____

Title ____

Company ____

Address ____

Fax (_____)________________________ Phone (_____)________________________

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, TX 78730-5039

Fax to: Technical Publications
National Instruments Corporation
(512) 794-5678

Glossary

Prefix	Meaning	Value
p-	pico-	10^{-12}
n-	nano-	10^{-9}
µ-	micro-	10^{-6}
m-	milli-	10^{-3}
k-	kilo-	10^{3}
M-	mega-	10^{6}

Numbers

1D	one-dimensional
2D	two-dimensional

A

active window	The window affected by user input at a given moment. The title of an active window is highlighted.
ANOVA	analysis of variance
Array Display	A mechanism for viewing and editing numeric arrays.
auto-exclusion	A mechanism that prevents pre-existing lines from executing in the Interactive Execution Window.

B

bps	bits per second
breakpoint	An interruption in the execution of a program.

C

caption bar	An area directly beneath the command bar at the top of a window that displays the name of the file you are working on.
cm	centimeters
command bar	An area along the top of a window that contains the names of the LabWindows/CVI command menus.
common control	A function panel control that specifies the first parameter in both primary and secondary functions associated with a function panel. A common control appears on a function panel in the same color or intensity as a primary control.
control	An input and output device that appears on a function panel for specifying function parameters and displaying function results.

D

DFT	Discrete Fourier Transform
dialog box	A prompt mechanism in which you specify additional information needed to complete a command.
DSP	digital signal processing

E

excluded code	Code that is ignored during compilation and execution. Excluded lines of code are displayed in a different color than included lines of code.

F

FFT	Fast Fourier Transform
FHT	Fast Hartley Transform
FIR	finite impulse response

function panel	A user interface to the LabWindows/CVI libraries in which you can interactively execute library functions and generate code for inclusion in a program.
function panel window	A window that contains one or more function panels.
function tree	The hierarchical structure in which the functions in a library or an instrument driver are grouped. The function tree simplifies access to a library or instrument driver by presenting functions organized according to the operation they perform, as opposed to a single linear listing of all available functions.

G

Generated Code box	A small window located at the bottom of the function panel screen that displays the code produced by the manipulation of function panel controls.
global control	A function panel control that displays the contents of global variables in a library function. Global controls allow you to monitor global variables in a function that are not specifically returned as results by the function. These are read-only controls that cannot be altered by the user, and do not contribute a parameter to the generated code.

H

hex	hexadecimal
Hz	hertz

I

IDFT	inverse Discrete Fourier Transform
IFFT	inverse Fast Fourier Transform
IFHT	inverse Fast Hartley Transform
IIR	infinite impulse response
in.	inches

input control	A function panel control that accepts a value typed in from the keyboard. An input control can have a default value associated with it. This value appears in the control when the panel is first displayed.
input focus	A mechanism for emphasis displayed on the screen as a highlight on an item, signifying that the item is active. User input affects the item in the dialog box that has the input focus.
Interactive Execution window	A LabWindows/CVI window in which sections of code may be executed without creating an entire program.

K

ksamples	1,000 samples

L

`.LFP` file	A file containing information about the function tree and function panels for a LabWindows/CVI permanent library.
list box	A dialog box item that displays a list of possible choices.

M

MB	megabytes of memory
menu	An area accessible from the command bar that displays a subset of the possible command choices.
mse	mean squared error

O

output control	A function panel control that displays a value determined by the function you execute.

P

Project window	A window that keeps track of the components that make up your current project. The Project window maintains a list of files such as source files, uir files, header files, or object modules, and also contains status information about each file in your project.
prompt command	A command that requires additional information before it can be executed; a prompt command appears on a pull-down menu suffixed with three ellipses (…).

R

return value control	A function panel control that displays a value returned from a function as a return value rather than as a formal parameter.
rms	root mean squared

S

scroll bars	Areas along the bottom and right sides of a window that show your relative position in the file. Scroll bars can be used with a mouse to move about in the window.
scrollable text box	A dialog box item that displays text in a scrollable display.
s	seconds
select	To choose the item that the next executed action will affect by moving the input focus (highlight) to a particular item or area.
shortcut key commands	A combination of keystrokes that provide a means of executing a command without accessing a menu in the command bar.
Source window	A LabWindows/CVI work area in which complete programs are edited and executed. This window is designated by the file extension .c.
Standard Input/ Output window	A LabWindows/CVI work area in which output to and input from the user take place.
standard libraries	The LabWindows/CVI Analysis, Formatting and I/O, GPIB/GPIB-488.2, RS-232, TCP/IP, DDE libraries and the ANSI C Library.
String Display	A mechanism for viewing and editing string variables and arrays.

V

V	volts
Variable Display	A display that shows the values of the variables that are currently defined in LabWindows/CVI.

Index

Numbers

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X