LabWindows/CVI

LabWindows/CVI
Advanced Analysis Library
Reference Manual

NATIONAL INSTRUMENTS™
The Software is the Instrument™

February 1998 Edition
Part Number 320686D-01

Internet Support
E-mail: `support@natinst.com`
FTP Site: `ftp.natinst.com`
Web Address: `http://www.natinst.com`

Bulletin Board Support
BBS United States: 512 794 5422
BBS United Kingdom: 01635 551422
BBS France: 01 48 65 15 59

Fax-on-Demand Support
512 418 1111

Telephone Support (USA)
Tel: 512 795 8248
Fax: 512 794 5678

International Offices
Australia 03 9879 5166, Austria 0662 45 79 90 0, Belgium 02 757 00 20, Brazil 011 288 3336,
Canada (Ontario) 905 785 0085, Canada (Québec) 514 694 8521, Denmark 45 76 26 00, Finland 09 725 725 11,
France 01 48 14 24 24, Germany 089 741 31 30, Hong Kong 2645 3186, Israel 03 6120092, Italy 02 413091,
Japan 03 5472 2970, Korea 02 596 7456, Mexico 5 520 2635, 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,
United Kingdom 01635 523545

National Instruments Corporate Headquarters
6504 Bridge Point Parkway Austin, Texas 78730-5039 USA Tel: 512 794 0100

© Copyright 1994, 1998 National Instruments Corporation. All rights reserved.

Important Information

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

Under the copyright laws, this publication may not be reproduced or transmitted in any form, electronic or mechanical, including photocopying, recording, storing in an information retrieval system, or translating, in whole or in part, without the prior written consent of National Instruments Corporation.

Trademarks

CVI™, `natinst.com`™, National Instruments™, the National Instruments logo, and The Software is the Instrument™ are trademarks of National Instruments Corporation.

Product and company names listed are trademarks or trade names of their respective companies.

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

About This Manual

Organization of This Manual xiii
Conventions Used in This Manual xiii
Related Documentation xiv
Customer Communication xvi

Chapter 1
Advanced Analysis Library Overview

Product Overview 1-1
Advanced Analysis Library Function Panels 1-1
Class and Subclass Descriptions 1-8
Hints for Using Advanced Analysis Function Panels 1-10
Reporting Analysis Errors 1-11
About the Fast Fourier Transform (FFT) 1-11
About Windowing 1-13
About Digital Filters 1-15
FIR Filters 1-16
IIR Filters 1-17
About Measurement Functions 1-19
About Curve Fitting Functions 1-21
About Vector & Matrix Algebra Functions 1-21

Chapter 2
Advanced Analysis Library Function Reference

Abs1D 2-2
ACDCEstimator 2-3
Add1D 2-4
Add2D 2-5
AllocIIRFilterPtr 2-6
AmpPhaseSpectrum 2-8
ANOVA1Way 2-10
ANOVA2Way 2-16
ANOVA3Way 2-27
ArbitraryWave 2-42
AutoPowerSpectrum 2-44
BackSub 2-46
Bessel_CascadeCoef 2-48
Bessel_Coef 2-50

BkmanWin 2-52
BlkHarrisWin 2-53
Bw_BPF 2-54
Bw_BSF 2-56
Bw_CascadeCoef 2-58
Bw_Coef 2-60
Bw_HPF 2-62
Bw_LPF 2-64
CascadeToDirectCoef 2-66
Ch_BPF 2-68
Ch_BSF 2-70
Ch_CascadeCoef 2-72
Ch_Coef 2-74
Ch_HPF 2-76
Ch_LPF 2-78
CheckPosDef 2-80
Chirp 2-81
Cholesky 2-82
Clear1D 2-84
Clip 2-85
ConditionNumber 2-86
Contingency_Table 2-88
Convolve 2-92
Copy1D 2-94
Correlate 2-95
CosTaperedWin 2-97
CrossPowerSpectrum 2-98
CrossSpectrum 2-100
CxAdd 2-102
CxAdd1D 2-103
CxCheckPosDef 2-104
CxCholesky 2-105
CxConditionNumber 2-107
CxDeterminant 2-109
CxDiv 2-111
CxDiv1D 2-112
CxDotProduct 2-113
CxEigenValueVector 2-114
CxExp 2-116
CxGenInvMatrix 2-117
CxGenLinEqs 2-119
CxLinEv1D 2-121
CxLn 2-123

CxLog 2-124
CxLU 2-125
CxMatrixMul 2-127
CxMatrixNorm 2-129
CxMatrixRank 2-131
CxMul 2-133
CxMul1D 2-134
CxOuterProduct 2-135
CxPolyRoots 2-137
CxPow 2-139
CxPseudoInverse 2-140
CxQR 2-142
CxRecip 2-144
CxSpecialMatrix 2-145
CxSqrt 2-148
CxSub 2-149
CxSub1D 2-150
CxSVD 2-151
CxSVDS 2-153
CxTrace 2-154
CxTranspose 2-155
Decimate 2-156
Deconvolve 2-157
Determinant 2-158
Difference 2-159
Div1D 2-161
Div2D 2-162
DotProduct 2-163
Elp_BPF 2-164
Elp_BSF 2-166
Elp_CascadeCoef 2-168
Elp_Coef 2-170
Elp_HPF 2-172
Elp_LPF 2-174
Equi_Ripple 2-176
EquiRpl_BPF 2-180
EquiRpl_BSF 2-182
EquiRpl_HPF 2-184
EquiRpl_LPF 2-186
ExBkmanWin 2-188
ExpFit 2-189
ExpWin 2-191
F_Dist 2-192

FFT 2-193
FHT 2-195
FIR_Coef 2-197
FlatTopWin 2-199
ForceWin 2-200
ForwSub 2-201
FreeAnalysisMem 2-203
FreeIIRFilterPtr 2-204
GaussNoise 2-205
GenCosWin 2-206
GenDeterminant 2-207
GenEigenValueVector 2-209
GenInvMatrix 2-211
GenLinEqs 2-213
GenLSFit 2-215
GenLSFitCoef 2-224
GetAnalysisErrorString 2-227
HamWin 2-228
HanWin 2-229
HarmonicAnalyzer 2-230
Histogram 2-232
IIRCascadeFiltering 2-234
IIRFiltering 2-236
Impulse 2-238
ImpulseResponse 2-239
Integrate 2-241
InvCh_BPF 2-243
InvCh_BSF 2-245
InvCh_CascadeCoef 2-247
InvCh_Coef 2-249
InvCh_HPF 2-251
InvCh_LPF 2-253
InvF_Dist 2-255
InvFFT 2-257
InvFHT 2-259
InvMatrix 2-261
InvN_Dist 2-262
InvT_Dist 2-263
InvXX_Dist 2-264
Ksr_BPF 2-265
Ksr_BSF 2-267
Ksr_HPF 2-269
Ksr_LPF 2-271

KsrWin 2-273
LinEqs 2-275
LinEv1D 2-276
LinEv2D 2-277
LinFit 2-278
LU 2-280
MatrixMul 2-282
MatrixNorm 2-284
MatrixRank 2-286
MaxMin1D 2-288
MaxMin2D 2-289
Mean 2-291
Median 2-292
Mode 2-293
Moment 2-294
Mul1D 2-296
Mul2D 2-297
N_Dist 2-298
Neg1D 2-299
NetworkFunctions 2-300
NonLinearFit 2-303
NonLinearFitWithMaxIters 2-305
Normal1D 2-307
Normal2D 2-309
NumericIntegration 2-311
OuterProduct 2-314
PeakDetector 2-315
PolyEv1D 2-318
PolyEv2D 2-320
PolyFit 2-322
PolyInterp 2-324
PowerFrequencyEstimate 2-326
Prod1D 2-329
PseudoInverse 2-330
Pulse 2-332
PulseParam 2-334
QR 2-337
QScale1D 2-339
QScale2D 2-340
Ramp 2-341
RatInterp 2-343
ReFFT 2-345
ReInvFFT 2-346

ResetIIRFilter 2-347
Reverse 2-349
RMS 2-350
SawtoothWave 2-351
Scale1D 2-353
Scale2D 2-355
ScaledWindow 2-357
Set1D 2-359
Shift 2-360
Sinc 2-362
SinePattern 2-363
SineWave 2-365
Sort 2-367
SpecialMatrix 2-368
Spectrum 2-371
SpectrumUnitConversion 2-372
SpInterp 2-376
Spline 2-378
SquareWave 2-380
StdDev 2-382
Sub1D 2-383
Sub2D 2-384
Subset1D 2-385
Sum1D 2-386
Sum2D 2-387
SVD 2-388
SVDS 2-390
SymEigenValueVector 2-391
T_Dist 2-393
ToPolar 2-394
ToPolar1D 2-395
ToRect 2-396
ToRect1D 2-397
Trace 2-398
TransferFunction 2-399
Transpose 2-401
Triangle 2-402
TriangleWave 2-403
TriWin 2-405
Uniform 2-406
UnWrap1D 2-407
Variance 2-408
WhiteNoise 2-409

Wind_BPF 2-410
Wind_BSF 2-412
Wind_HPF 2-414
Wind_LPF 2-416
XX_Dist 2-418

Appendix A
Error Codes

Appendix B
Customer Communication

Glossary

Index

Figures

Figure 1-1. Windowed Spectrum in the Continuous Case 1-14
Figure 1-2. Cascaded Filter Stages 1-18

Tables

Table A-1. Advanced Analysis Library Error Codes, Sorted Alphabetically A-1
Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically A-4

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 in alphabetical order.
- Appendix A, *Error Codes*, contains error codes the Advanced Analysis Library functions return.
- 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 abbreviations, acronyms, 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:

This icon to the left of bold italicized text denotes a note, which alerts you to important information.

This icon to the left of bold italicized text denotes a caution, which advises you of precautions to take to avoid injury, data loss, or a system crash.

bold	Bold text denotes the names of menus, menu items, parameters, dialog box buttons, and 1D and 2D arrays. 1D arrays appear in lowercase, and 2D arrays appear in uppercase.
bold italic	Bold italic text denotes a note or caution.
italic	Italic text denotes variables, emphasis, a cross reference, an introduction to a key concept, or a single number or one element of an array or a matrix. Parameter names in formulas appear in italic text. This font also denotes text from which you supply the appropriate word or value.
`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. This font is also used for the proper names of disk drives, paths, directories, programs, functions, filenames and extensions, and for statements and comments taken from programs.
`monospace italic`	Italic text in this font denotes that you must enter the appropriate words or values in the place of these items.

Related Documentation

The following documents contain information you might 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., and Faires, J.D. *Numerical Analysis*, 3rd ed. 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, MA: 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.
- Golub, G.H., and VanLoan, C.F. *Matrix Computations*, 2nd ed. Baltimore: The Johns Hopkins University Press, 1989.
- 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).
- McClellan, J.H. “A Computer Program for Designing Optimum FIR Linear Phase Digital Filters,” *IEEE Transactions on Audio and Electroacoustics.* AU-21 (1973): (December).
- Miller, I., and Freund, J.E. *Probability and Statistics for Engineers*. Englewood Cliffs, NJ: 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., and Schafer, R.W. *Discrete-Time Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1989.
- Parks, T.W., and Burrus, C.S. *Digital Filter Design*. New York: John Wiley & Sons, 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., and Gold, B. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 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, NJ: Prentice-Hall, Inc., 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 if you have problems with them. To make it easy for you to contact us, this manual contains comment and configuration forms for you to complete. These forms are in Appendix B, *Customer Communication*, at the end of this manual.

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, curve-fitting, and matrix operations.

Advanced Analysis Library Function Panels

The Advanced Analysis Library function panels are grouped in the following tree structure according to the types of operations they perform.

The first- and second-level 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 are the names of individual function panels. Each analysis function panel generates one analysis function call.

The following shows the structure of the Advanced Analysis Library function tree.

- Signal Generation
- Array Operations
  - 1D Operations
  - 2D Operations
- Complex Operations
  - Complex Numbers
  - 1D Complex Operations
- Signal Processing
  - Frequency Domain
  - Time Domain

Signal Processing (continued)
- IIR Digital Filters
  - Cascade Filter Functions
    - Filter Information Utilities
  - One-Step Filter Functions
  - Old-Style Filter Functions
- FIR Digital Filters
- Windows

Measurement

Statistics
- Basics
- Probability Distributions
- Analysis of Variance
- Nonparametric Statistics

Curve Fitting
- OldStyle Function

Interpolation

Vector & Matrix Algebra
- Real Matrices
- Complex Matrices

Additional Numerical Methods

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree

Class/Panel Name	Function Name
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`
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`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Array Operations (continued)	
1D Operations (continued)	
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`
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`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Complex Operations (continued)	
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`
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`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Signal Processing (continued)	
IIR Digital Filters (continued)	
One-Step Filter Functions (continued)	
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`
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`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Signal Processing (continued)	
Windows (continued)	
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`
Total Harmonic Distortion	`HarmonicAnalyzer`
Statistics	
Basics	
Mean	`Mean`
Standard Deviation	`StdDev`
Variance	`Variance`
Root Mean Squared Value	`RMS`
Moments about the Mean	`Moment`
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-Square Dist.	`InvXX_Dist`
Analysis of Variance	
One-way ANOVA	`ANOVA1Way`
Two-way ANOVA	`ANOVA2Way`
Three-way ANOVA	`ANOVA3Way`
Nonparametric Statistics	
Contingency Table	`Contingency_Table`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Curve Fitting	
Linear Fit	`LinFit`
Exponential Fit	`ExpFit`
Polynomial Fit	`PolyFit`
General Least Squares Fit	`GenLSFit`
Non-Linear Fit	`NonLinearFit`
Non-Linear Fit with Maximum Iterations	`NonLinearFitWithMaxIters`
OldStyle Function	
Gen Least Squares Fit Coeff	`GenLSFitCoef`
Interpolation	
Polynomial Interpolation	`PolyInterp`
Rational Interpolation	`RatInterp`
Spline Interpolation	`SpInterp`
Spline Interpolant	`Spline`
Vector & Matrix Algebra	
Real Matrices	
Create Special Matrix	`SpecialMatrix`
Dot Product	`DotProduct`
Transpose	`Transpose`
Determinant	`Determinant`
Determinant (General)	`GenDeterminant`
Trace	`Trace`
Invert Matrix	`InvMatrix`
Invert Matrix (General)	`GenInvMatrix`
Solution of Linear Equations	`LinEqs`
Solution of Linear Eqs (General)	`GenLinEqs`
Multiply Matrices	`MatrixMul`
Outer Product	`OuterProduct`
Rank	`MatrixRank`
Norm	`MatrixNorm`
Condition Number	`ConditionNumber`
Eigenvalues & Eigenvectors (Symmetric)	`SymEigenValueVector`
Eigenvalues & Eigenvectors (General)	`GenEigenValueVector`
Singular Values of a Matrix	`SVDS`
SVD Factorization	`SVD`
QR Factorization	`QR`
Cholesky Factorization	`Cholesky`
PseudoInverse Matrix	`PseudoInverse`
Test Positive Definiteness	`CheckPosDef`
LU Decomposition	`LU`
Forward Substitution	`ForwSub`
Backward Substitution	`BackSub`
Complex Matrices	
Create Special Complex Matrix	`CxSpecialMatrix`
Complex Dot Product	`CxDotProduct`

Table 1-1. Functions in the Advanced Analysis Library Overview Function Tree (Continued)

Class/Panel Name	Function Name
Vector & Matrix Algebra (continued)	
Complex Matrices (continued)	
Complex Transpose	`CxTranspose`
Complex Determinant	`CxDeterminant`
Complex PseudoInverse Matrix	`CxPseudoInverse`
Complex Trace	`CxTrace`
Complex Invert Matrix	`CxGenInvMatrix`
Solution of Complex Linear Eqs	`CxGenLinEqs`
Complex Multiply Matrices	`CxMatrixMul`
Complex Outer Product	`CxOuterProduct`
Complex Rank	`CxMatrixRank`
Complex Norm	`CxMatrixNorm`
Complex Condition Number	`CxConditionNumber`
Complex Eigenvalues & Eigenvectors	`CxEigenValueVector`
Complex Singular Values	`CxSVDS`
Complex SVD Factorization	`CxSVD`
Complex QR Factorization	`CxQR`
Complex Cholesky Factorization	`CxCholesky`
Complex Test Positive Definite	`CxCheckPosDef`
Complex LU Factorization	`CxLU`
Additional Numerical Methods	
Complex Polynomial Roots	`CxPolyRoots`
Numeric Integration	`NumericIntegration`
Peak Detector	`PeakDetector`
Free Analysis Memory	`FreeAnalysisMem`
Get Error String	`GetAnalysisErrorString`

Class and Subclass Descriptions

- 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 functions process the real and imaginary parts of complex numbers 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 that 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, and total harmonic distortion analysis.
- 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 contains 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 a 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.
  - Real Matrices, a subclass of Vector & Matrix Algebra, contains functions that operate on real-valued matrices and vectors.
  - Complex Matrices, a subclass of Vector & Matrix Algebra, contains functions that operate on matrices and vectors that contain complex numbers. The complex numbers are represented in the form of a structure that the ComplexNum typedef defines.
- The Additional Numerical Methods function panels perform operations such as numeric integration and peak detection that are widely used in signal processing applications.

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 manipulate scalars and arrays of data interactively. You might 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, remember the following:

- The computer on which you run LabWindows/CVI affects the processing speed of the analysis functions. A numeric coprocessor, especially, increases the speed of floating-point computations. If you are using an Analysis Library function panel and nothing seems to happen for an unusually long time, remember the constraints of your hardware.
- Many analysis routines for arrays run in place. That is, the functions can store the input and output data in the same array. This point is important to keep in mind when you process 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 executes properly, the function returns a zero; otherwise, the function returns an appropriate error value.

The return value corresponds to one of the enumeration values of the type that `AnalysisLibErrType` declares 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 display the symbolic name instead of the integer value of the error code. Declaring a variable to be the type `AnalysisLibErrType` allows the Variables window to display 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 on the discrete implementation and optimization of the Fourier Transform integral. The functions obtain the Discrete Fourier Transform (DFT) of a complex sequence X that contains n elements using the following formula:

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

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 functions obtain the Inverse Discrete Fourier Transform (IDFT) of a complex sequence Y that contains n elements using the following formula:

$$X_i = \frac{1}{n}\sum_{k=0}^{n-1} Y_k e^{\frac{j2\pi ik}{n}} \quad \text{for } i = 0, 1, \ldots, n-1$$

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 about 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 and 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 in the following paragraphs.

If there are *n* number of elements in the complex sequence and $k = \frac{n}{2}$, 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 subclass:

- All arrays must be a power of two: $n = 2^m$, for $m = 1, 2, 3, \ldots, 12$.
- The functions manipulate complex sequences using two arrays. One array contains the real elements. The other array contains the imaginary elements.

This manual uses the following notation to describe the FFT operations the functions in the Frequency Domain subclass perform:

- $Y = \mathrm{FFT}(X)$, the sequence *Y* is the FFT of the sequence *X*.
- $Y = \mathrm{FFT}^{-1}(X)$, 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 window always affects a discrete signal and its spectrum. 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:

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

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 frequency domain convolution shown as follows:

$$Y_k = X_k \Theta W_k$$

Convolving with the window spectrum always distorts the original signal spectrum in some way. A window spectrum consists of a mainlobe and several *sidelobes*.

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

The sidelobes of a window function affect frequency leakage. A signal spectrum line leaks into the adjacent spectrum if the sidelobes are large. Once again, the leakage results from the convolution process. Select a window with relatively smaller sidelobes to reduce spectral leakage. Unfortunately, a narrower mainlobe and smaller sidelobes are mutually exclusive.

For this reason, selecting a window function is application dependent. Figure 1-1 shows an example of a windowed spectrum in the continuous case.

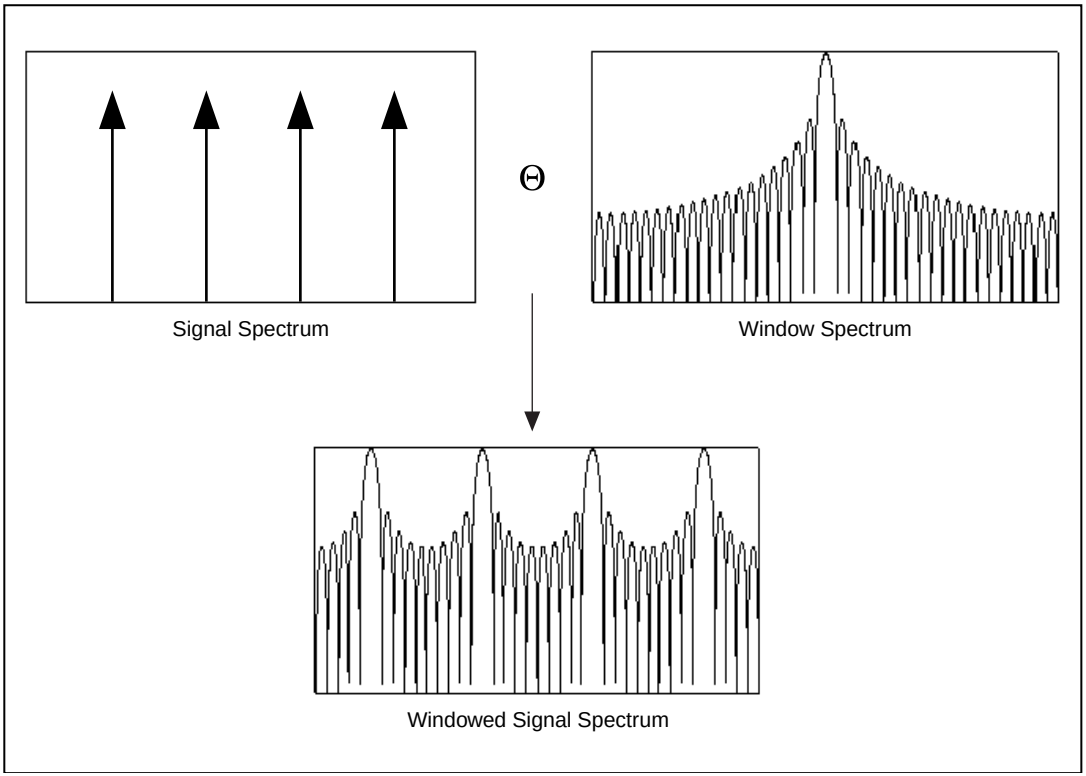

Figure 1-1. Windowed Spectrum in the Continuous Case

The original signal spectrum in Figure 1-1 is convolved with the window spectrum, and the output is a smeared version of the original signal spectrum. In Figure 1-1, 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 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:

$$w_0 \neq w_{N-1} \qquad \text{where } N \text{ is the window length}$$

However, the windows are usually symmetrical in the frequency domain. For example, the Hamming window definition uses the formula:

$$w_i = 0.54 - 0.46 \cos\left(\frac{2\pi i}{N}\right) \tag{1-2}$$

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

$$w_i = 0.54 - 0.46 \cos\left(\frac{2\pi i}{N-1}\right) \tag{1-3}$$

The difference is small if N is large.

Equation (1-2) 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 Equation (1-3).

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 on the application. If you want a linear phase response, choose one of the FIR filters. If performance and better amplitude response are more important, chose an IIR filter. No matter what type of filter you choose, enter a sampling frequency and other cutoff frequencies when you design 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 do not 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, for $k = 0, 1, 2, \ldots,$ be the filter coefficients, x_N the input signal, and y_N the output in the following formula:

$$y_i = \sum_{k=0}^{K-1} x_{i-k} c_k \quad \text{when } i = 0, 1, \ldots, N-1$$

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 an FIR filter function does not actually perform filtering. You must subsequently call `Convolve` to perform the filtering. The advantage of this process is that after you obtain 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 might design a better filter with the same number of coefficients.

Use the **windType** parameter to choose the window type to use in a window FIR filter. **windType** determines the amount of attenuation the window filter can achieve. It also determines the transitional bandwidth of the window filter, which is 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 to choose 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 you use `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 one 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, such as 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 you obtain the filter coefficients.***

IIR Filters

Mathematically, an IIR digital filter assumes the following form:

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{N_b-1} b_j x_{i-j} - \sum_{k=1}^{N_a-1} a_k y_{i-k}\right) \quad \text{where } a_k \text{ and } b_k \text{ are the filter coefficients} \qquad (1\text{-}4)$$

The current filter output y_i depends on 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. For these reasons, these filters are called infinite impulse response filters.

Filters implemented directly using the structure Equation (1-4) defines are known as direct-form IIR filters. Direct-form implementations are often sensitive to errors introduced by coefficient quantization and by computational precision limits. Also, 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 (1-4) (with $a_0 = 1$) can be written as a ratio of z transforms, as follows:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \ldots + b_{N_b-1} z^{-(N_b-1)}}{1 + a_1 z^{-1} + \ldots + a_{N_a-1} z^{-(N_a-1)}} \qquad (1\text{-}5)$$

By factoring Equation (1-5) into second-order sections, the transfer function of the filter becomes a product of second-order filter functions:

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

where $N_s = \left\lfloor \frac{N_a}{2} \right\rfloor$ is the largest integer $\leq \frac{N_a}{2}$, and $N_a \geq N_b$

This new filter structure can be described as a cascade of second-order filters, as shown in Figure 1-2.

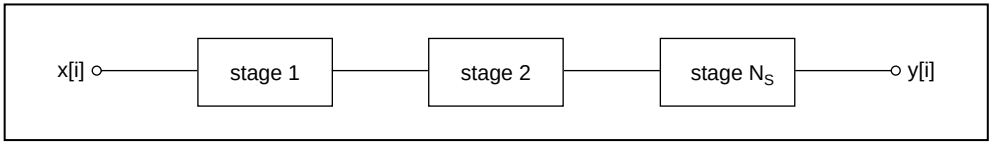

Figure 1-2. Cascaded Filter Stages

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, or 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:

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

$$s_k[i] = y_{k-1}[i-1] - a_{1_k} s_k[i-1] - a_{2_k} s_k[i-2] \quad \text{for } k = 1, 2, \ldots, N_s$$

$$y_k[i] = b_{0_k} s_k[i] + b_{1_k} s_k[i-1] + b_{2_k} s_k[i-2] \quad \text{for } k = 1, 2, \ldots, N_s$$

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

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

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

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

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

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

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

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

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

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

Notice that in the case of fourth-order filter stages, $N_s = \left\lfloor \frac{N_a + 1}{4} \right\rfloor$.

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 exist 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 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, and total harmonic distortion analysis.

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, or 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 functions 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, 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 $\frac{fs}{2}$, where *fs* is the sampling frequency.
- 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 Functions

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:

$$e(a) = \sum_i \left| Y_i - \mathrm{f}(X_i, a) \right|^2$$ where f(X_i, *a*) is the function that represents the desired curve

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

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

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:

$$Z = \mathrm{f}(X, a)$$

When you establish the best fit values, you can obtain the mean squared error (*mse*) by applying the following formula:

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

About Vector & Matrix Algebra Functions

The functions in the Vector & Matrix Algebra class perform operations such as multiplication, transposition, and outer product calculation on 2D arrays or matrices. You can use these functions to calculate matrix properties, such as determinant, rank, norm, and condition number.

Many applications require you to solve a linear system of equations and/or to determine the eigenvalues and eigenvectors of a matrix. This class contains functions you can use for this purpose and functions that you can use to calculate different types of factorizations, such as Cholesky factorization, QR factorization, and Singular Value Decomposition. You can use the functions in this class to calculate special types of matrices, such as Toeplitz matrix, Vandermonde matrix, and Companion matrix.

2

Advanced Analysis Library Function Reference

This chapter contains a brief explanation of each of the functions in the LabWindows/CVI Advanced Analysis Library in alphabetical order.

Abs1D

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

Purpose

Finds the absolute value of the **x** input array. `Abs1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Absolute value of input array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ACDCEstimator

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

Purpose

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

acEstimate is the estimate of the input signal AC content in volts, root-mean-square, if the input signal is in volts.

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

Parameters

Input

Name	Type	Description
x	double-precision array	Contains the time-domain signal, usually in volts. This array must contain at least three cycles of the signal for a valid estimate.
n	integer	Number of elements in the input array.

Output

Name	Type	Description
acEstimate	double-precision	Contains the estimate of the AC level of the input signal in volts, root-mean-square, 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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Add1D

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

Purpose

Adds 1D arrays. `Add1D` obtains the ith element of the output array using the following formula:

$$z_i = x_i + y_i$$

`Add1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
y	double-precision array	Input array.
n	integer	Number of elements to add.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Add2D

```
int status = Add2D (void *x, void *y, int n, int m, void *z);
```

Purpose

Adds 2D arrays. `Add2D` obtains the $(i, j)^{th}$ element of the output array using the following formula:

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

`Add2D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

AllocIIRFilterPtr

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

Purpose

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

You input the type of the filter, such as lowpass, highpass, bandpass, or bandstop, and the order. `AllocIIRFilterPtr` allocates the filter structure as well as the internal coefficient arrays and internal filter state array.

Parameters

Input

Name	Type	Description
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. The default value is `3`.

Return Value

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

Parameter Discussion

filterInformation is the pointer to the filter structure that contains the filter coefficients and the internal filter information. Call this function to allocate **filterInformation** before you call 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

Calculates the single-sided, scaled amplitude and phase spectra of a time-domain signal, *X*. `AmpPhaseSpectrum` calculates the amplitude spectrum as

$$\left|\frac{\mathrm{FFT}(X)}{n}\right|$$

and converts it to single-sided form. `AmpPhaseSpectrum` calculates the phase spectrum as

$$\mathrm{phase}(\mathrm{FFT}(X))$$

and converts it to single-sided form.

Parameters

Input

Name	Type	Description
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.
unwrap	integer	Controls the unwrapping of the phase spectrum. Valid values: `1` = enable phase unwrapping `0` = disable phase unwrapping $(-\pi \le phase \le +\pi)$
dt	double-precision	Sampling period of the time-domain signal, usually in seconds. **dt** = $1/fs$, where *fs* is the sampling frequency of the time-domain signal.

Output

Name	Type	Description
ampSpectrum	double-precision array	Single-sided amplitude spectrum magnitude in volts, root-mean-square, if the input signal is in volts. If the input signal is not in volts, the results are in input signal units, root-mean-square. This array must be at least **n**/2 elements long.
phaseSpectrum	double-precision array	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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 you make at various levels of some factor, with at least one observation per factor, and performs a one-way analysis of variance (ANOVA) 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

Name	Type	Description
y	double-precision array	Experimental observations.
level	integer array	The ith element tells in what level of the factor the ith observation falls.
n	integer	Total number of observations.
k	integer	Total number of levels of the factor.

Output

Name	Type	Description
ssa	double-precision	Sum of squares as a result of the factor.
msa	double-precision	Mean square as a result of the factor.
f	double-precision	Calculated F-value.
sig	double-precision	Level of significance at which you must reject the null hypothesis.
sse	double-precision	Sum of squares as a result of random fluctuation.
mse	double-precision	Mean square as a result of random fluctuation.
tss	double-precision	Total sum of squares.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

Factors and Levels

A factor is a way of categorizing data. You can categorize data into levels, beginning with level 0. For example, if you perform a measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For age, you might have three levels, as shown in Table 2-1.

Table 2-1. Age Levels

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

General Method

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 might be attributed to different sources.

You now have:

$$tss = ssa + sse$$

where *ssa* is a measure of variation that is attributed to the factor
sse is a measure of variation that is attributed to random fluctuation

Divide by appropriate numbers to obtain the averages **msa** and **mse**. If the factor causes much variation, **msa** will be larger relative to **mse**. The ratio **f** also will 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 sampling from this distribution results in a value larger than **f**.

Statistical Model

`ANOVA1Way` expresses each experimental outcome as the sum of three parts while it performs the analysis of variance. Let $y_{i,m}$ be the mth observation from the ith level. Each observation is written as:

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

where μ is a standard effect

α_i is the effect of the ith level of the factor

$\varepsilon_{i,m}$ is a random fluctuation

Assumptions

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.

Hypothesis

Test the null hypothesis that $\alpha_i = 0$ for $i = 0, 1, \ldots, k-1$, 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.

Testing the Hypothesis

`ANOVA1Way` generates a number **f** so that if the hypothesis is true, that number is from an F-distribution with $\mathbf{k}-1$ and $\mathbf{n}-\mathbf{k}$ degrees of freedom. `ANOVA1Way` also calculates the probability that a number taken from this F-distribution is larger than **f**. This is the output parameter **sig**:

$$sig = \text{prob}(x > f) \quad \text{where } x \text{ is from } \mathrm{F}(k-1, n-k)$$

Use the probability **sig** to determine when to reject the hypothesis by choosing a level of significance for the hypothesis. The level of significance determines how likely you are to reject the hypothesis when it is in fact true. Thus, the level of significance should be small, for example, 0.05. Remember that the smaller the level of significance, the less likely you are to reject the hypothesis.

Reject the hypothesis when the output parameter **sig** is less than the level of significance you choose.

Formulas

Let $y_{i,m}$ be the mth observation at the ith level for $m = 0, 1, \ldots, n_i$ and $i = 0, 1, \ldots, k$.

Let n_i = the number of observations at the ith level.

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

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

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

$$T = n \times Y$$

Then:

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

$$mse = \frac{ssa}{k-1}$$

$$sse = \sum_{i=0}^{k-1}\sum_{m=0}^{n_i-1} y_{i,m}^2 - \sum_{i=0}^{k}\frac{Y_i^2}{n_i}$$

$$mse = \frac{sse}{n-k}$$

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

$f = \dfrac{msa}{mse}$ where f is from an F-distribution with $k-1$ and $n-k$ degrees of freedom

Example

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 shown in Table 2-2.

Table 2-2. Rainfall Levels

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. Table 2-3 shows their results.

Table 2-3. Plot Production

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 `ANOVA1Way`, you store all the numbers of bushels 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*, you must set these arrays such that y_i is the number of bushels a plot produces in the ith **level**. For example:

$$level_i = 0$$

$$y_i = 128, 122, 126, \text{ or } 124$$

are valid combinations. Therefore, you can set up the input arrays **y** and **level** in this example for `ANOVA1Way` 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 in the following example produces:

$$sig = 0.0000239$$

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 crop yield. 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:

- Model 1: 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.
- Model 2: Fixed effects with interaction and **L** > 1 observations per cell.
- Model 3: 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.
- Model 4: 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. The method for finding significance varies among models.

Parameters

Input

Name	Type	Description
y	double-precision array	Array of experimental data of $\mathbf{N} = \lvert\mathbf{a}\rvert \times \lvert\mathbf{b}\rvert \times \mathbf{L}$ elements.
levelA	integer array	The ith element tells in what level of factor A the ith observation falls.
levelB	integer array	The ith element tells in what level of factor B the ith observation falls.
N	integer	Total number of observations.
L	integer	Number of observations per cell.
a	integer	Number of levels in factor A; negative if A is a random effect.
b	integer	Number of levels in factor B; negative if B is a random effect.

Output

Name	Type	Description
info	double-precision 2D array	A 4-by-4 matrix as follows: $\begin{bmatrix} ssa & dofa & msa & fa \\ ssb & dofb & msb & fb \\ ssab & dofab & msab & fab \\ sse & dofe & mse & 0.0 \end{bmatrix}$ 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 you must reject hypothesis A.
sigB	double-precision	Level of significance at which you must reject hypothesis B.
sigAB	double-precision	Level of significance at which you must reject hypothesis AB.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

Factors, Levels, and Cells

A factor is a way of categorizing data. You can categorize data into levels, beginning with level 0. For example, if you perform a measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For age, you might have three levels, as shown in Table 2-4.

Table 2-4. Age Levels

Level	Age
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 levels shown in Table 2-5.

Table 2-5. Eye Color Levels

Level	Eye Color
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 they perform.

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 years and 15 years old. The number of observations that fall in each cell must be a 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 you want to draw conclusions about, but that you cannot sample 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 you can sample the factor from all levels you want to draw conclusions about.

The input parameters **a** and **b** represent the number of levels in factors A and B, respectively. If factor A is random, set **a** to a negative value. If factor B is random, set **b** to a negative value. If only one observation per cell exists, both **a** and **b** must be positive. Use model 1 as previously described.

General Method

Each of the previous models breaks up the total sum of squares (*tss*), which is a measure of the total variation of the data from the overall population mean, into a 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 as a result of factor A; *ssb* is a measure of the variation as a result of factor B; *ssab* is a measure of the variation as a result of the interaction between factors A and B; and *sse* is a measure of the variation as a result of random fluctuation. Notice that there is no *ssab* term with model 1. Thus, no interaction exists.

If factor A has a strong effect on the experimental observations, **msa** is 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.

Statistical Model

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

In 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 pth level of factor A
β_q represents the effect of the qth level of factor B
$\varepsilon_{p,q,r}$ is a random fluctuation

In models 2 through 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 pth level of factor A

β_q represents the effect of the qth level of factor B

$\varepsilon_{p,q,r}$ is a random fluctuation

$(\alpha\beta)_{p,q}$ represents the effect of the interaction between the pth level of factor A and the qth 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. All the populations at each of the levels have the same variance. In addition, assume that all the α_p means sum to zero. Make an analogous assumption 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. Make an analogous assumption for B.
- If all 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, assume that the effect is a random variable normally distributed with mean 0 and variance σ_{AB}^2. If A is fixed but B is random, assume also 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 also that for any fixed *p*, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all *q*.
- Assume that all effects taken to random variables are independent.

Hypotheses

Each of the following hypotheses are different ways of stating 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 hypothesis A, $a_p = 0$ for all levels of p if factor A is fixed; $\sigma_A^2 = 0$ if factor A is random.
- For hypothesis B, $\beta_q = 0$ for all levels of q if factor B is fixed; $\sigma_B^2 = 0$ if factor B is random.
- For hypothesis 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 factor B is random. This does not apply to model 1, where no interaction exists and the associated output parameters are superfluous.

Testing the Hypotheses

For each hypothesis, `ANOVA2Way` generates a number so that if the hypothesis is true, that number is from a particular F-distribution.

For example, in model 1, $fa = msa/mse$, associated with hypothesis A, is from an F-distribution with $\mathbf{a} - 1$ and $(\mathbf{a} - 1) \times (\mathbf{b} - 1)$ degrees of freedom, given that hypothesis A is true. In models 2 through 4, $fa = msa/mse$, associated with hypothesis A, is from an F-distribution with $\mathbf{a} - 1$ and $ab(\mathbf{L} - 1)$ degrees of freedom, given that hypothesis A is true. `ANOVA2Way` calculates the probability that a number taken from a particular F-distribution is larger than the F-value. For example:

$$sigA = \text{prob}(X > fa) \quad \text{where } X \text{ is from } \mathrm{F}(a - 1, (a - 1)(b - 1))$$

Use the probabilities **sigA**, **sigB**, and **sigAB** to determine when to reject the associated hypotheses A, B, and AB by choosing a level of significance for each hypothesis. The level of significance determines how likely you are to reject the hypothesis when it is in fact true. Thus, the level of significance should be small, for example, 0.05. Remember that the smaller the level of significance, the less likely you are to reject the hypothesis.

Reject a particular hypothesis when the associated output parameter **sigA**, **sigB**, or **sigAB** is less than the level of significance you chose for that hypothesis. If A is a random effect, 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 has an effect on the experimental observations.

Formulas

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

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:

$$A = \sum_{p=0}^{aa-1} \frac{T_p^2}{bb \times L}$$

$$B = \sum_{q=0}^{bb-1} \frac{T_q^2}{aa \times L}$$

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

$$CF = \frac{T^2}{aa \times bb \times L}$$

Then:

$$ssa = A - CF$$

$$msa = \frac{ssa}{aa-1} = \frac{ssa}{dofa}$$

$$ssb = B - CF$$

$$msb = \frac{ssb}{bb-1} = \frac{ssb}{dofb}$$

$$ssab = S - A - B - CF$$

$$msab = \frac{ssab}{(a-1)(b-1)} = \frac{ssab}{dofab}$$

$$sse = T - S$$

$$mse = \frac{sse}{aa \times bb \times (L-1)} = \frac{sse}{dofe}$$

$$fa = \begin{cases} \dfrac{msa}{mse} & \text{if } B \text{ is fixed} \\ \dfrac{msa}{msab} & \text{if } B \text{ is random} \end{cases}$$

$$fb = \begin{cases} \dfrac{msb}{mse} & \text{if } A \text{ is fixed} \\ \dfrac{msb}{msab} & \text{if } A \text{ is random} \end{cases}$$

$$fab = \frac{msab}{mse}$$

If:

$$f = \frac{ms_1}{ms_2}$$

$$ms_1 = \frac{ss_1}{dof_1}$$

$$ms_2 = \frac{ss_2}{dof_2}$$

assume that *f* is from an F-distribution with dof_1 and dof_2 degrees of freedom.

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 shown in Table 2-6 and Table 2-7.

Table 2-6. Rainfall Levels

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

Table 2-7. Temperature Levels

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

A particular plot planted with the crop might appear in any one of the nine different combinations of these levels with the two factors. For example, one combination might be 2 inches of rain and an average temperature between 76 degrees and 80 degrees, recorded as (0,0). Call these combinations cells.

The researchers set up 18 plots in various geographical locations chosen so that two plots fall in each of the nine cells. To measure the productivity of a particular plot, they record the crop production. Let rainfall be factor A and temperature be factor B. Table 2-8 shows their results.

Table 2-8. Plot Production

(A, B)	Bushels Produced from Each Plot
(0, 0)	128 122
(0, 1)	113 108
(0, 2)	116 116
(1, 0)	132 129

Table 2-8. Plot Production (Continued)

(A, B)	Bushels Produced from Each Plot
(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 `ANOVA2Way`, you store all the numbers of bushels 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*, you set these arrays such that y_i is the number of bushels a plot produces in the ($levelA_i$, $levelB_i$) cell. For example:

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

$$y_i = 113 \text{ or } 108$$

are valid combinations. Therefore, you can set up the input arrays **y**, **levelA**, and **levelB** in this example for `ANOVA2Way` 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 in the following example produces:

$$sigA = 0.026$$

$$sigB = 0.203$$

$$sigAB = 0.0018$$

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 crop yield. In other words, the combination of rainfall and temperature has a significant effect on crop yield.

Example

```
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:

- Model 1: Fixed effects with interaction and **L** > 1 observations per cell.
- Model 2: 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.
- Model 3: 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. The method for finding significance varies among models.

Parameters

Input

Name	Type	Description
y	double-precision array	Array of experimental data of $\mathbf{N} = \lvert\mathbf{a}\rvert \times \lvert\mathbf{b}\rvert \times \lvert\mathbf{c}\rvert \times \mathbf{L}$ elements.
levelA	integer array	The ith element tells in what level of factor A the ith observation falls.
levelB	integer array	The ith element tells in what level of factor B the ith observation falls.
levelC	integer array	The ith element tells in what level of factor C the ith observation falls.
N	integer	Total number of observations.
L	integer	Number of observations per cell.
a	integer	Number of levels in factor A; negative if A is a random effect.

Name	Type	Description
b	integer	Number of levels in factor B; negative if B is a random effect.
c	integer	Number of levels in factor C; negative if C is a random effect.

Output

Name	Type	Description
info	double-precision 2D array	An 8-by-4 matrix as follows: $\begin{bmatrix} ssa & dofa & msa & fa \\ ssb & dofb & msb & fb \\ ssc & dofc & msc & fc \\ ssab & dofab & msab & fab \\ ssac & dofac & msac & fac \\ ssabc & dofabc & msabc & fabc \\ sse & dofe & mse & 0.0 \end{bmatrix}$ 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 you must reject hypothesis A.
sigB	double-precision	Level of significance at which you must reject hypothesis B.
sigC	double-precision	Level of significance at which you must reject hypothesis C.
sigAB	double-precision	Level of significance at which you must reject hypothesis AB.
sigAC	double-precision	Level of significance at which you must reject hypothesis AC.

Name	Type	Description
sigBC	double-precision	Level of significance at which you must reject hypothesis BC.
sigABC	double-precision	Level of significance at which you must reject hypothesis ABC.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

Factors, Levels, and Cells

A factor is a way of categorizing data. You can categorize data into levels, beginning with level 0. For example, if you perform a measurement on individuals, such as counting the number of sit-ups they can perform, one such categorization method is age. For age, you might have three levels, as shown in Table 2-9.

Table 2-9. Age Levels

Level	Age
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 levels shown in Table 2-10.

Table 2-10. Eye Color Levels

Level	Eye Color
0	blue
1	brown
2	green
3	hazel

A third factor might be height with levels in blocks of 10 cm. 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 years old and 15 years old who are between 151 cm and 160 cm tall. The number of observations that fall in each cell must be a 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 you want to draw conclusions about, but that you cannot sample 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 you want to draw conclusions about.

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

General Method

Each of the previous models breaks up the total sum of squares (*tss*), which is a measure of the total variation of the data from the overall population mean, into a 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 as a result of factor A; *ssb* is a measure of the variation as a result of factor B; *ssc* is a measure of the variation as a result of factor C; *ssab* is a measure of the variation as a result of the interaction between factors A and B; and so on for *ssac*, *ssbc*, and *ssabc*. The variable *sse* is a measure of the variation as a result of random fluctuation.

If factor A has a strong effect on the experimental observations, **msa** is 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.

Statistical Model

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

Express each observation as the sum of eight components so that:

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

where μ represents a standard effect present in each observation

α_p, β_q, and Υ_r are the effects of factors A, B, and C respectively

$(\alpha\beta)_{p,q}$, $(\alpha\Upsilon)_{p,r}$, $(\beta\Upsilon)_{q,r}$, and $(\alpha\beta\Upsilon)_{p,q,r}$ are the effects of the corresponding interactions

$\varepsilon_{p,q,r,s}$ 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. All the populations at each of the levels have the same variance. In addition, assume that all the α_p means sum to zero. Make analogous assumptions 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. Make analogous assumptions for B and C.
- If all 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, assume that the effect is a random variable normally distributed with mean 0 and variance σ_{AB}^2. If A is fixed but B is random, assume also 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 also that for any fixed p, the $(\alpha\beta)_{p,q}$ means sum to zero when summing over all q.
- Assume that all effects taken to random variables are independent.

Hypotheses

Each of the following hypotheses are different ways of stating 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 hypothesis A, $\alpha_p = 0$ for all levels of *p* if factor A is fixed; $\sigma_A^2 = 0$ if factor A is random.
- For hypothesis B, $\beta_q = 0$ for all levels of *q* if factor B is fixed; $\sigma_B^2 = 0$ if factor B is random.
- For hypothesis C, $\Upsilon_r = 0$ for all levels of *r* if factor C is fixed; $\sigma_C^2 = 0$ if factor C is random.
- For hypothesis 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 hypothesis AC, $(\alpha\Upsilon)_{p,r} = 0$ 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 hypothesis BC, $(\beta\Upsilon)_{q,r} = 0$ 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 hypothesis ABC, $(\alpha\beta\Upsilon)_{p,q,r} = 0$ 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.

Testing the Hypotheses

For each hypothesis, `ANOVA3Way` generates a number so that if the hypothesis is true, that number is 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 $\mathbf{a} - 1$ and $abc(\mathbf{L} - 1)$ degrees of freedom, given that hypothesis A is true. `ANOVA3Way` calculates the probability that a number taken from a particular F-distribution is larger than the F-value. For example:

$$sigA = \text{prob}(X > fa) \quad \text{where } X \text{ is from } \mathrm{F}(a - 1, abc(L - 1))$$

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 by choosing a level of significance for each hypothesis. The level of significance determines how likely you are to reject the hypothesis when it is in fact true. Thus, the level of significance should be small, for example, 0.05. Remember that the smaller the level of significance, the less likely you are to reject the hypothesis.

Reject a particular hypothesis when the associated output parameter **sigA**, **sigB**, **sigC**, **sigAB**, **sigAC**, **sigBC**, or **sigABC** is less than the level of significance you choose for that hypothesis. If A is a random effect, 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 has an effect on the experimental observations.

With some models, no appropriate tests exist for certain hypotheses. In these cases, `ANOVA3Way` sets the output parameters directly involved with the testing of those hypotheses to –1.0.

Formulas

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

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:

$$A = \sum_{p=0}^{aa-1} \frac{T^2}{bb \times cc \times L}$$

$$B = \sum_{q=0}^{bb-1} \frac{T_p^2}{aa \times cc \times L}$$

$$C = \sum_{r=0}^{cc-1} \frac{T_r^2}{aa \times bb \times L}$$

$$AB = \sum_{p=0}^{aa-1} \sum_{q=0}^{bb-1} \frac{T_{p,q}^2}{cc \times L}$$

$$AC = \sum_{p=0}^{aa-1} \sum_{r=0}^{cc-1} \frac{T_{p,r}^2}{bb \times L}$$

$$BC = \sum_{q=0}^{bb-1} \sum_{r=0}^{cc-1} \frac{T_{q,r}^2}{aa \times L}$$

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

$$CF = \frac{T^2}{aa \times bb \times cc \times L}$$

Then:

$$ssa = A - CF$$

$$msa = \frac{ssa}{aa-1} = \frac{ssa}{dofa}$$

$$ssb = B - CF$$

$$msb = \frac{ssb}{bb-1} = \frac{ssb}{dofb}$$

$$ssc = C - CF$$

$$msc = \frac{ssc}{cc-1} = \frac{ssc}{dofc}$$

$$ssab = AB - A - B + CF$$

$$msab = \frac{ssab}{(aa-1)(bb-1)} = \frac{ssab}{dofab}$$

$$ssac = AC - A - C + CF$$

$$msac = \frac{ssac}{(aa-1)(cc-1)} = \frac{ssac}{dofac}$$

$$ssbc = BC - B - C + CF$$

$$msbc = \frac{ssbc}{(bb-1)(cc-1)} = \frac{ssbc}{dofbc}$$

$$ssabc = S - AB - AC - BC + A + B + C - CF$$

$$msabc = \frac{ssabc}{(aa-1)(bb-1)(cc-1)} = \frac{ssabc}{dofabc}$$

$$mse = \frac{sse}{(aa \times bb \times cc)(L-1)} = \frac{sse}{dofe}$$

$$fa = \begin{cases} \dfrac{msa}{mse} & \text{if } B \text{ and } C \text{ are fixed} \\ \dfrac{msa}{msab} & \text{if } B \text{ is random and } C \text{ is fixed} \\ \dfrac{msa}{msac} & \text{if } B \text{ is fixed and } C \text{ is random} \end{cases}$$

$$fb = \begin{cases} \dfrac{msb}{mse} & \text{if } A \text{ and } C \text{ are fixed} \\ \dfrac{msb}{msab} & \text{if } A \text{ is random and } C \text{ is fixed} \\ \dfrac{msb}{msbc} & \text{if } A \text{ is fixed and } C \text{ is random} \end{cases}$$

$$fc = \begin{cases} \dfrac{msc}{mse} & \text{if } A \text{ and } B \text{ are fixed} \\ \dfrac{msc}{msac} & \text{if } A \text{ is random and } B \text{ is fixed} \\ \dfrac{msc}{msbc} & \text{if } A \text{ is fixed and } B \text{ is random} \end{cases}$$

$$fab = \begin{cases} \dfrac{msab}{mse} & \text{if } C \text{ is fixed} \\ \dfrac{msab}{msabc} & \text{if } C \text{ is random} \end{cases}$$

$$fac = \begin{cases} \dfrac{msac}{mse} & \text{if } B \text{ is fixed} \\ \dfrac{msac}{msabc} & \text{if } B \text{ is random} \end{cases}$$

$$fbc = \begin{cases} \dfrac{msbc}{mse} & \text{if } A \text{ is fixed} \\ \dfrac{msbc}{msabc} & \text{if } A \text{ is random} \end{cases}$$

$$fabc = \frac{msabc}{mse}$$

If:

$$f = \frac{ms_1}{ms_2}$$

$$ms_1 = \frac{ss_1}{dof_1}$$

$$ms_2 = \frac{ss_2}{dof_2}$$

assume that *f* is from an F-distribution with dof_1 and dof_2 degrees of freedom.

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 shown in Tables 2-11, 2-12, and 2-13.

Table 2-11. Sunlight Levels

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

Table 2-12. Rainfall Levels

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

Table 2-13. Temperature Levels

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

A particular plot planted with the crop might appear in any one of the 27 different combinations of these levels with the three factors. For example, one combination might be 6 hours of sunlight with 2 inches of rainfall and an average temperature between 76 degrees and 80 degrees, recorded as (1,0,0). Call these combinations cells.

The researchers set up 54 plots in various geographical locations chosen so that two plots 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.

Table 2-14 shows their results.

Table 2-14. Plot Production

(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

Table 2-14. Plot Production (Continued)

(A, B, C)	Bushels Produced from Each Plot
(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
(1, 1, 1)	128 120
(1, 1, 2)	122 121
(1, 2, 0)	114 115
(1, 2, 1)	116 113
(2, 0, 0)	135 131
(2, 0, 1)	145 145
(2, 0, 2)	152 147
(2, 1, 0)	137 141
(2, 1, 1)	171 171
(2, 1, 2)	135 131
(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 `ANOVA3Way`, you store all the numbers of bushels 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, you set these arrays such that y_i is the number of bushels a plot produces 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, you can set up the input arrays **y**, **levelA**, **levelB**, and **levelC** in this example for `ANOVA3Way` as follows:

$$y = 128, 122, 113, 108, 116, 116, 132, 129, \ldots$$

$$levelA = 0, 0, 0, 0, 0, 0, 0, 0, \ldots$$

$$levelB = 0, 0, 0, 0, 0, 0, 1, 1, \ldots$$

$$levelC = 0, 0, 1, 1, 2, 2, 0, 0, \ldots$$

Running the code in the following example produces:

$$sigA = 1.11e^{-16}$$

$$sigB = 1.3e^{-8}$$

$$sigC = 0.0072$$

$$sigAB = 1.2e^{-8}$$

$$sigAC = 2.0e^{-4}$$

$$sigBC = 4.5e^{-10}$$

$$sigABC = 4.8e^{-10}$$

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 crop yield. In other words, all three factors have a significant effect on crop yield.

Example

```
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 that contains an arbitrary wave, with each cycle described by an interpolated version of the **waveTable** you specify. `ArbitraryWave` generates the output array **x** according to the following formula:

$$x_i = amp \times \mathrm{arb}(phase + f \times 360.0 \times i)$$

where $\mathrm{arb}(p) = \mathrm{WT}(p \text{ modulo } 360.0)$
f is frequency in cycles per sample

`ArbitraryWave` calculates WT(x) according to the following interpolation values:

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

where $ix = (int)x$
$dx = x - (int)x$
(int) is the integral part of the variable x

You can use `ArbitraryWave` to simulate a continuous acquisition from an arbitrary wave function generator. The unit of the input **phase** is in degrees, and `ArbitraryWave` sets **phase** to (**phase** + **f** × 360.0 × **n**) modulo 360.0 before it returns.

Parameters

Input

Name	Type	Description
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.

Name	Type	Description
waveTable	double-precision array	Contains equally spaced samples of one cycle of the generated signal.
tableSize	integer	Number of elements the **waveTable** array contains.
interp	integer	Determines the type of interpolation to use to generate the arbitrary wave signal from the **waveTable** samples. `0` = no interpolation `1` = linear interpolation

Output

Name	Type	Description
phase	double-precision	Upon completion of `ArbitraryWave`, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to `ArbitraryWave` to simulate a continuous function generator.
x	double-precision array	Contains the generated arbitrary wave signal.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

AutoPowerSpectrum

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

Purpose

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

$$\frac{\mathrm{FFT}(X)\ \mathrm{FFT}^*(X)}{n^2}$$

where n is the number of points in the signal array X

* denotes a complex conjugate

`AutoPowerSpectrum` converts the auto power spectrum to a single-sided form.

Parameters

Input

Name	Type	Description
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	Sampling period of the time-domain signal, usually in seconds. **dt** = $1/fs$, where fs is the sampling frequency of the time-domain signal.

Output

Name	Type	Description
autoSpectrum	double-precision array	Single-sided amplitude spectrum magnitude in volts, root-mean-square, if the input signal is in volts. If the input signal is not in volts, the results are in input signal units, root-mean-square. This array must be at least $\mathbf{n}/2$ elements long.
df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. $\mathbf{df} = 1/(\mathbf{n} \times \mathbf{dt})$

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

BackSub

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

Purpose

Solves the linear equations $\mathbf{a} \times \mathbf{x} = \mathbf{y}$ by backward substitution. `BackSub` assumes **a** is an **n**-by-**n** lower triangular matrix in which the diagonal elements all equal one. `BackSub` obtains **x** using the following formulas:

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

$$x_i = \frac{y_i - \sum_{j=i+1}^{n-1} a_{i,j} x_j}{a_{i,j}}$$

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

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

Parameters

Input

Name	Type	Description
a	double-precision 2D array	Input matrix.
y	double-precision array	Input vector.
n	integer	Dimension size of **a**.

Output

Name	Type	Description
x	double-precision array	Solution vector.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that 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 you call `Bessel_CascadeCoef`.

If the type and order remain the same, you can call `Bessel_CascadeCoef` 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

Name	Type	Description
fs	double-precision	Specifies the sampling frequency in hertz.
fl	double-precision	Specifies the desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Specifies the desired upper cutoff frequency of the filter in hertz.

Output

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. **type** has the valid values as shown in Table 2-15.

Table 2-15. Valid type Values

Value	Description
0	lowpass filter; **fh** is not used
1	highpass filter; **fh** is not used
2	bandpass filter
3	bandstop filter

a and **b** are the reverse and forward filter coefficients. Use `IIRFiltering` to achieve the actual filtering:

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

Parameters

Input

Name	Type	Description
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 hertz.
fl	double-precision	Desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Desired higher cutoff frequency of the filter in hertz.

Name	Type	Description
na	integer	Number of coefficients in the **a** coefficient array.
nb	integer	Number of coefficients in the **b** coefficient array.

Output

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the designed IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the designed IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

BkmanWin

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

Purpose

Applies a Blackman window to the **x** input signal. The following formula defines the Blackman window:

$$w_i = 0.42 - 0.5\cos\left(\frac{2\pi i}{n}\right) + 0.08\cos\left(\frac{4\pi i}{n}\right) \quad \text{for } i = 0, 1, \ldots, n-1$$

`BkmanWin` obtains the output signal using the following formula:

$$x_i = x_i \times w_i \quad \text{for } i = 0, 1, \ldots, n-1$$

The function performs the window operation in place; that is, the windowed data **x** replaces the input data **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x.**

Output

Name	Type	Description
x	double-precision array	Windowed data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, `BlkHarrisWin` obtains the elements of **y** using the formula:

$$y_i = x_i\left(0.42323 - 0.49755\cos\left(\frac{2\pi i}{n}\right) + 0.07922\cos\left(\frac{4\pi i}{n}\right)\right)$$

where n is the number of elements in x

Parameters

Input

Name	Type	Description
x	double-precision array	Contains the input signal.
n	integer	Number of elements in the input array.

Output

Name	Type	Description
x	double-precision array	Contains the signal after `BlkHarrisWin` applies the Blackman-Harris window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Bw_BPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Bw_BSF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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.

filterInformation is the pointer to the filter structure that 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 you call `Bw_CascadeCoef`.

If the type and order remain the same, you can call `Bw_CascadeCoef` 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

Name	Type	Description
fs	double-precision	Specifies the sampling frequency in hertz.
fl	double-precision	Specifies the desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Specifies the desired upper cutoff frequency of the filter in hertz.

Output

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. **type** has the valid values as shown in Table 2-16.

Table 2-16. Valid type Values

Value	Description
0	lowpass filter; **fh** is not used
1	highpass filter; **fh** is not used
2	bandpass filter
3	bandstop filter

a and **b** are the reverse and forward filter coefficients. Use `IIRFiltering` to achieve actual filtering:

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

Parameters

Input

Name	Type	Description
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 hertz.
fl	double-precision	Desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Desired higher cutoff frequency of the filter in hertz.
na	integer	Number of coefficients in the **a** coefficient array.
nb	integer	Number of coefficients in the **b** coefficient array.

Output

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the designed IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the designed IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Bw_HPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Bw_LPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 the **filterInformation** structure contains to direct-form IIR coefficients in arrays **a** and **b**. These two arrays must be allocated in the same way as the old-style direct coefficient design functions, for example, `Bw_Coef`.

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 you call `CascadeToDirectCoef`.

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

For bandpass and bandstop type filters, the direct coefficient arrays must equal ($2 \times order + 1$).

Parameters

Input

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.
na	integer	Specifies the number of coefficients in the **a** coefficient array. **na** = $order + 1$ for low- or highpass filters **na** = $2 \times order + 1$ for bandpass or bandstop filters
nb	integer	Specifies the number of coefficients in the **b** coefficient array. **nb** = $order + 1$ for low- or highpass filters **nb** = $2 \times order + 1$ for bandpass or bandstop filters

Output

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the direct-form IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the direct-form IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Ch_BPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Ch_BSF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that 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 you call `Ch_CascadeCoef`.

If the type and order remain the same, you can call `Ch_CascadeCoef` 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

Name	Type	Description
fs	double-precision	Specifies the sampling frequency in hertz.
fl	double-precision	Specifies the desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Specifies the desired upper cutoff frequency of the filter in hertz.
ripple	double-precision	Specifies the amplitude of the stopband ripple in decibels.

Output

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. **type** has the valid values as shown in Table 2-17.

Table 2-17. Valid type Values

Value	Description
0	lowpass filter; **fh** is not used
1	highpass filter; **fh** is not used
2	bandpass filter
3	bandstop filter

a and **b** are the reverse and forward filter coefficients. Use `IIRFiltering` to achieve the actual filtering:

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

Parameters

Input

Name	Type	Description
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 hertz.
fl	double-precision	Desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Desired higher cutoff frequency of the filter in hertz.

Name	Type	Description
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

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the designed IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the designed IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Ch_HPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Ch_LPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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);
```

CheckPosDef

```
int status = CheckPosDef (void *A, int n, int *positiveDefinite);
```

Purpose

Checks if the real, square input matrix is positive definite. A real, square matrix is positive definite if and only if it is symmetric and the quadratic form

$$x^T A x > 0$$

is true for all nonzero vectors *x*. For more information on positive definite matrices, refer to *Matrix Computations* by G.H. Golub and C.F. VanLoan.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input square matrix.
n	integer	Number of elements in one dimension of the matrix.

Output

Name	Type	Description
positiveDefinite	integer	`1` if the input matrix is positive definite; `0` otherwise.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Chirp

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

Purpose

Generates an array that contains a chirp pattern. `Chirp` generates the output array **x** according to the following formula:

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

where $a = \dfrac{2\pi \times (f2 - f1)}{n}$

$b = 2\pi \times f1$

and where $f1$ is the beginning frequency in cycles per sample

$f2$ is the ending frequency in cycles per sample

Parameters

Input

Name	Type	Description
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

Name	Type	Description
x	double-precision array	Contains the generated chirp pattern.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Cholesky

```
int status = Cholesky (void *A, int n, void *R);
```

Purpose

Calculates the Cholesky factorization of a real, symmetric positive definite input matrix. If the input matrix is not positive definite, `Cholesky` returns an error.

The following formula defines the Cholesky factorization of an **n**-by-**n** symmetric positive definite matrix **A**:

$$A = R^T R$$

where R is an upper triangular matrix of dimensions **n**-by-**n**
R^T is the transpose of R

Cholesky factorization is similar to LU factorization for symmetric positive definite matrices. If the matrix in your application is positive definite, use Cholesky factorization rather than LU factorization for the following reasons:

- The algorithm is well defined.
- Numerical stability does not require pivoting.
- Cholesky factorization requires about half the programming time and less memory than LU factorization.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input square, positive definite matrix.
n	integer	Number of elements in one dimension of the matrix.

Output

Name	Type	Description
R	double-precision 2D array	Result matrix of the Cholesky decomposition.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Clear1D

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

Purpose

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

Parameters

Input

Name	Type	Description
n	integer	Number of elements in **x**.

Output

Name	Type	Description
x	double-precision array	Cleared array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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**]. `Clip` obtains the ith element of the resulting array using the following formula:

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

`Clip` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x**.
upper	double-precision	Upper limit.
lower	double-precision	Lower limit.

Output

Name	Type	Description
y	double-precision array	Clipped array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ConditionNumber

```
int status = ConditionNumber (void *A, int n, int m, int normType, double *c);
```

Purpose

Calculates the condition number of a real input matrix **A**. The **normType** parameter indicates what type of norm to use to calculate the condition number. The input matrix **A** does not need to be square when **normType** is 2-norm.

The following formula defines the condition number **c** of a matrix **A**:

$$c = \|A\|_p \times \|A^{-1}\|_p \qquad \text{where } \|A\|_p \text{ is the p-norm of the matrix } A$$

The **normType** value defines the p-norm behavior. For a 2-norm **normType**, **c** is the ratio of the largest singular value of **A** to the smallest singular value of **A**.

The condition number of a matrix indicates how near to singular the matrix is. A matrix with a large condition number is nearly singular, and a matrix with a condition number close to one is far from singular. The condition number of a matrix is always greater than or equal to one, and it can help assess the accuracy of a solution to a linear system of equations and matrix inversion.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input matrix. If **normType** is 2-norm, the matrix can be square or rectangular; otherwise, it must be square.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
normType	integer	Type of p-norm function to use to calculate the condition number.

Output

Name	Type	Description
c	double-precision	Condition number of the matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **normType** parameter indicates what type of norm to use to calculate the condition number. Table 2-18 shows valid norm type values.

Table 2-18. Valid Norm Type Values

Norm Type	Value	Meaning
2-norm	0	Largest singular value of **A**.
1-norm	1	Largest column sum of **A**.
Frobenius-norm	2	Square root of the sum of the diagonal elements of $\mathbf{A}^{\mathbf{T}}\mathbf{A}$, where $\mathbf{A}^{\mathbf{T}}$ is the complex conjugate transpose of **A**.
Infinite-norm	3	Largest row sum of **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 to classify and tally objects of experimentation according to two schemes of categorization. Use `Contingency_Table` 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 you apply. The only difference is in the hypothesis you test.***

Parameters

Input

Name	Type	Description
s	integer	Number of random samples 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 categorization scheme in the test of independence.
y	integer 2D array	Contingency table, indexed as an **s**-by-**k** matrix.

Output

Name	Type	Description
Test_Stat	double-precision	Use to calculate **Sig**. If the hypothesis is true, **Test_Stat** is known to come from a chi-square distribution with $(\mathbf{s} - 1) \times (\mathbf{k} - 1)$ degrees of freedom.
Sig	double-precision	Level of significance at which you must reject the hypothesis.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

A contingency table is a table in which you can classify and tally objects of experimentation 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 an issue.

Chi-Square Test of Homogeneity

Take a random sample of a fixed size 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 contingency table shown in Table 2-19.

Table 2-19. Contingency Table

Category	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 you take each sample 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 you select at random is just as likely to vote yes as a Whig you select 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 receive the results shown in Table 2-20.

Table 2-20. Contingency Table

Category	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 is a Mugwump, the hypothesis states that his or her political affiliation has no effect on how he or she votes on the issue you select.

Testing the Hypothesis

Whichever test you use, you must choose a level of significance. This determines how likely you are to reject a true hypothesis. Thus, the level of significance should be small, for example, 0.05.

The output parameter **Sig** is the level of significance at which you reject the hypothesis:

$$Sig = \text{prob}(\chi \geq Test_Stat)$$

where χ is a random variable from the chi-square distribution with $(s-1)\times(k-1)$ degrees of freedom

If **Sig** is less than the level of significance, you must reject the hypothesis.

Formulas

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

Let:

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

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

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

$$e_{p,q} = \frac{y_p \times y_q}{y}$$

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

Example

```
/* Generate a 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. `Convolve` obtains the convolution using the following formula:

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

where $y = 0,\ b = i$ 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*** $\mathbf{m} \le \mathbf{n}$***. For*** $\mathbf{m} > \mathbf{n}$***, exchange*** (**x**, **y**) ***and*** (**m**, **n**) ***in the previous equations.***

Parameters

Input

Name	Type	Description
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

Name	Type	Description
cxy	double-precision array	Convolution array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

The size of the output array must be at least $(\mathbf{n} + \mathbf{m} - 1)$ 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. Use `Copy1D` to duplicate arrays for in place operations.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
y	double-precision array	Duplicated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Correlate

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

Purpose

Finds the correlation of the input arrays. `Correlate` obtains the correlation using the following formula:

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

$$y_j = 0 \quad \text{when } j < 0 \text{ or } j \geq m$$

$$x_i = 0 \quad \text{when } j < 0 \text{ or } j \geq n$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
rxy	double-precision array	Correlation array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

The size of the output array must be at least $(\mathbf{n} + \mathbf{m} - 1)$ 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, `CosTaperedWin` obtains the elements of **y** from the formula:

$$y_i = \begin{cases} 0.5\ x_i \left(1 - \cos\left(\frac{2\pi i}{n}\right)\right) & i = 0, 1, ..., m - 1 \\ x_i & i = m, m + 1, n - m - 1 \\ 0.5\ x_i \left(1 - \cos\left(\frac{2\pi i}{n}\right)\right) & i = n - m, n - m + 1, n - 1 \end{cases}$$

where $m = round\left(\frac{n}{10}\right)$

Parameters

Input

Name	Type	Description
x	double-precision array	Contains the input signal.
n	integer	Number of elements in the input array.

Output

Name	Type	Description
x	double-precision array	Contains the signal after `CosTaperedWin` applies the cosine-tapered window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CrossPowerSpectrum

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

Purpose

Calculates the single-sided, scaled cross power spectrum of two time-domain signals. The following formula defines the cross power spectrum:

$$Sxy = \frac{\mathrm{FFT}(y)\mathrm{FFT}(x)}{n^2} \quad \text{where } n \text{ is the number of points in arrays } x \text{ and } y$$

magSxy and **phaseSxy** are single-sided magnitude and phase spectra of *Sxy*.

Parameters

Input

Name	Type	Description
x	double-precision array	Time-domain signal **x**.
y	double-precision array	Time-domain signal **y**.
n	integer	Number of elements in the input array. **n** must be a power of 2.
dt	double-precision	Sampling period of the time-domain signal, usually in seconds. **dt** $= 1/fs$, where *fs* is the sampling frequency of the time-domain signal.

Output

Name	Type	Description
magSxy	double-precision array	Single-sided magnitude cross power spectrum between signals **x** and **y** in volts rms square if the input signals are in volts. If the input signals are not in volts, the results are in input signal units rms square. This array must be at least $\mathbf{n}/2$ elements long.
phaseSxy	double-precision array	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 $\mathbf{n}/2$ elements long.
df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. $\mathbf{df} = 1/(\mathbf{n} \times \mathbf{dt})$

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CrossSpectrum

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

Purpose

Calculates the double-sided cross power spectrum, *Sxy*, of the input sequences **x** and **y** according to the following formula:

$$Sxy = \frac{\mathrm{FFT}^*(x)\mathrm{FFT}(y)}{n^2}$$

where n is the number of samples in both input sequences
$\mathrm{FFT}^*(x)$ is the complex conjugate of $\mathrm{FFT}(x)$

n must be a power of two. `CrossSpectrum` copies the input sequences to internal buffers before it calculates the FFTs. The output arrays are the real and imaginary parts of the cross spectrum.

Parameters

Input

Name	Type	Description
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. **n** must be a power of 2.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxAdd

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

Purpose

Adds two complex numbers, *x* and *y*. `CxAdd` obtains the resulting complex number, *z*, using the following formulas:

$$zr = xr + yr$$

$$zi = xi + yi$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision pointer	Real part of *z*.
zi	double-precision pointer	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxAdd1D

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

Purpose

Adds two 1D complex arrays, **x** and **y**. `CxAdd1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = xr_i + yr_i$$

$$zi_i = xi_i + yi_i$$

`CxAdd1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxCheckPosDef

```
int status = CxCheckPosDef (void *A, int n, int *positiveDefinite);
```

Purpose

Checks if the complex, square input matrix **A** is positive definite. A complex, square matrix is positive definite if and only if it is symmetric and the quadratic form

$$x^H Ax > 0$$

is true for all nonzero vectors *x*. For more information on positive definite matrices, refer to *Matrix Computations* by G.H. Golub and C.F. VanLoan.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix.
n	integer	Number of elements in one dimension of the matrix.

Output

Name	Type	Description
positiveDefinite	integer	`1` if the input matrix is positive definite; `0` otherwise.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxCholesky

```
int status = CxCholesky (void *A, int n, void *R);
```

Purpose

Calculates the Cholesky factorization of a complex, symmetric positive definite input matrix. If the input matrix is not positive definite, `CxCholesky` returns an error.

The following formula defines the Cholesky factorization of an **n**-by-**n** symmetric positive definite matrix **A**:

$$A = R^H R$$

where R is an upper triangular matrix of dimensions **n**-by-**n**
R^H is the complex conjugate transpose of R

Cholesky factorization is similar to LU factorization for symmetric positive definite matrices. If the matrix in your application is positive definite, use Cholesky factorization rather than LU factorization for the following reasons:

- The algorithm is well defined.
- Numerical stability does not require pivoting.
- Cholesky factorization requires about half the programming time and less memory than LU factorization.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix.
n	integer	Number of elements in one dimension of the matrix.

Output

Name	Type	Description
R	ComplexNum 2D array	Result matrix of the Cholesky decomposition.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxConditionNumber

```
int status = CxConditionNumber (void *A, int n, int m, int normType,
                                double *c);
```

Purpose

Calculates the condition number of a complex input matrix **A**. The **normType** parameter indicates what type of norm to use to calculate the condition number. The input matrix **A** does not need to be square when **normType** is 2-norm.

The following formula defines the condition number **c** of a matrix **A**:

$$c = \|A\|_p \times \left\|A^{-1}\right\|_p \quad \text{where } \|A\|_p \text{ is the p-norm of the matrix } A$$

The **normType** value defines the type of norm. For a 2-norm **normType**, **c** is the ratio of the largest singular value of **A** to the smallest singular value of **A**.

The condition number of a matrix indicates how near to singular the matrix is. A matrix with a large condition number is nearly singular, and a matrix with a condition number close to one is far from singular. The condition number of a matrix is always greater than or equal to one, and it can help assess the accuracy of a solution to a linear system of equations and matrix inversion.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix. If **normType** is 2-norm, the matrix can be square or rectangular; otherwise, it must be square.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
normType	integer	Type of p-norm function to use to calculate the condition number.

Output

Name	Type	Description
c	double-precision	Complex condition number of the matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

The **normType** parameter indicates what type of norm to use to calculate the condition number. Table 2-21 shows valid norm type values.

Table 2-21. Valid Norm Type Values

Norm Type	Value	Meaning
2-norm	0	Largest singular value of **A**.
1-norm	1	Largest column sum of **A**.
Frobenius-norm	2	Square root of the sum of the diagonal elements of $\mathbf{A}^{\mathbf{H}}\mathbf{A}$, where $\mathbf{A}^{\mathbf{H}}$ is the complex conjugate transpose of **A**.
Infinite-norm	3	Largest row sum of **A**.

CxDeterminant

```
int status = CxDeterminant (void *A, int n, int matrixType, ComplexNum *det);
```

Purpose

Calculates the complex determinant of a square, complex input matrix **A**. The input matrix can be upper or lower triangular, general, or positive definite. For an upper or lower triangular matrix, the determinant equals the product of the diagonal elements of the matrix. For a positive definite matrix, `CxDeterminant` first calculates the Cholesky factorization of the input matrix and then calculates the determinant as the square of the determinant of **R**.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix.
n	integer	Number of elements in one dimension of the matrix.
matrixType	integer	Type of the input matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
det	ComplexNum	Complex determinant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Table 2-22 shows valid matrix type values.

Table 2-22. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

CxDiv

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

Purpose

Divides two complex numbers, *x* and *y*. `CxDiv` obtains the resulting complex number, *z*, using the following formulas:

$$zr = \frac{(xr \times yr + xi \times yi)}{yr^2 + yi^2}$$

$$zi = \frac{(xi \times yr - xr \times yi)}{yr^2 + yi^2}$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxDiv1D

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

Purpose

Divides two 1D complex arrays, **x** and **y**. `CxDiv1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = \frac{(xr_i \times yr_i + xi_i \times yi_i)}{yr_i^2 + yi_i^2}$$

$$zi_i = \frac{(xi_i \times yr_i - xr_i \times yi_i)}{yr_i^2 + yi_i^2}$$

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

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxDotProduct

```
int status = CxDotProduct (ComplexNum x[], ComplexNum y[], int n,
                           ComplexNum *dotProduct);
```

Purpose

Calculates the dot product of the complex input arrays **x** and **y**. Use the following formula to obtain the dot product *d*:

$$d = \sum_{i=0}^{n-1} x_i \times y_i$$

Parameters

Input

Name	Type	Description
x	ComplexNum array	First complex input vector.
y	ComplexNum array	Second complex input vector.
n	integer	Number of elements in each vector.

Output

Name	Type	Description
dotProduct	ComplexNum	Complex dot product.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxEigenValueVector

```
int status = CxEigenValueVector (void *A, int n, int matrixType,
                                 int outputChoice, ComplexNum eigenValues[],
                                 void *eigenVectors);
```

Purpose

Calculates the eigenvalues λ and the corresponding eigenvectors **x** of a complex, square input matrix **A**. The following formula defines the eigenvalues and the corresponding eigenvectors:

$$Ax = \lambda x$$

The **matrixType** parameter indicates the type of the input matrix. The input matrix can be a general or a Hermitian matrix. The **outputChoice** parameter determines what `CxEigenValueVector` calculates. Depending on your application, you can choose to calculate just the eigenvalues or to calculate both the eigenvalues and the eigenvectors. The **eigenValues** output parameter is a 1D array of **n** complex numbers. The **eigenVectors** output parameter is an **n**-by-**n**, complex matrix (2D array) that contains the eigenvectors of the input matrix. Each ith column of this matrix is the eigenvector that corresponds to the ith component of the **eigenValues**. Each eigenvector is normalized so that its largest component equals one.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix.
n	integer	Number of elements in one dimension of the matrix.
matrixType	integer	Pass `0` for general matrix; `1` for Hermitian matrix. Choose the matrix type correctly because it significantly affects the speed of computation.
outputChoice	integer	Pass `0` for eigenvalues only; `1` for both eigenvalues and eigenvectors.

Output

Name	Type	Description
eigenValues	ComplexNum array	Result eigenvalues of the input matrix.
eigenVectors	ComplexNum 2D array	Result eigenvectors of the input matrix. You can pass `NULL` if **outputChoice** is `0`.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxExp

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

Purpose

Calculates the exponential of a complex number, *x*. `CxExp` obtains the resulting complex number, *y*, using the following formula:

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

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxGenInvMatrix

```
int status = CxGenInvMatrix (void *A, int n, int matrixType, void *B);
```

Purpose

Calculates the inverse of a complex, square input matrix **A**. If **B** denotes the inverse of the matrix **A**:

$$AB = I \quad \text{where } I \text{ is the identity matrix}$$

The input matrix can be an upper or lower triangular matrix, a general, square matrix, or a positive definite matrix. You can save significant computation time if you properly specify the input matrix type.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex, square matrix.
n	integer	Number of elements in one dimension of the matrix.
matrixType	integer	Type of the input matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
B	ComplexNum 2D array	Result complex inverse matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Table 2-23 shows valid matrix type values.

Table 2-23. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

CxGenLinEqs

```
int status = CxGenLinEqs (void *A, int n, int m, ComplexNum y[],
                          int matrixType, ComplexNum x[]);
```

Purpose

Solves for the unknown vector **x** in the linear system of equations:

$$Ax = y \tag{2-1}$$

where A is the complex input matrix
y is the known vector on the right side

The input matrix can be square or rectangular. The number of elements in **y** must equal the number of rows in the matrix **A**.

`CxGenLinEqs` calculates the solution using the Singular Value Decomposition technique.

In the case of non-singular, square matrices, in which no row or column is a linear combination of any other row or column, `CxGenLinEqs` solves for the unique solution **x**.

Two possibilities exist in the case of rectangular matrices. If the number of rows is greater than the number of columns, the system has more equations than unknowns and is an overdetermined system. Because the solution that satisfies Equation (2-1) might not exist, this procedure finds the least square solution **x**, which minimizes $\|A\|_2$. If the number of rows is less than the number of columns, the system has more unknowns than equations and is an underdetermined system. It might have infinite solutions that satisfy Equation (2-1). This procedure calculates the minimum 2-norm solution.

If the input matrix is rank deficient, `CxGenLinEqs` returns a warning.

The **matrixType** parameter specifies the type of the input matrix. The input matrix can be an upper or lower triangular matrix, a general matrix, or a positive definite matrix.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix. The matrix can be square or rectangular.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Name	Type	Description
y	ComplexNum array	Complex array that contains the set of known vector coefficients.
matrixType	integer	Type of the input matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
x	ComplexNum array	Solution to the linear system of equations.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes. If the input matrix is rank-deficient, `CxGenLinEqs` returns the warning code `20001`.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Table 2-24 shows valid matrix type values.

Table 2-24. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

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, **x** and **y**. `CxLinEv1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$yr_i = ar \times xr_i - ai \times xi_i + br$$

$$yi_i = ar \times xi_i + ai \times xr_i + bi$$

`CxLinEv1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
yr	double-precision array	Real part of **y**.
yi	double-precision array	Imaginary part of **y**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxLn

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

Purpose

Calculates the natural logarithm of a complex number, *x*. `CxLn` obtains the resulting complex number, *y*, using the following formula:

$$(yr, yi) = \log_e(xr, xi) \quad \text{where } e = 2.178\ldots$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxLog

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

Purpose

Calculates the logarithm (base 10) of a complex number, *x*. `CxLog` obtains the resulting complex number, *y*, using the following formula:

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

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxLU

```
int status = CxLU (void *A, int n, int p[], int *sign);
```

Purpose

Performs an LU decomposition on the complex, square matrix **A**:

$$PA = LU$$

where L is an **n**-by-**n** lower triangular matrix with all diagonal elements equal to one

U is an upper triangular matrix

P is an identity matrix with some rows exchanged based on the information in the permutation vector **p**

On output, the **U** matrix occupies the upper triangular part of the input matrix, including the diagonal elements, and the **L** matrix occupies the lower part.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Square matrix to factorize.
n	integer	Number of elements in one dimension of the matrix.

Output

Name	Type	Description
A	ComplexNum 2D array	LU factorized matrix.
p	integer array	Permutation vector.
sign	integer	Row exchange indicator.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

After `CxLU` executes, **p** contains possible row exchange information. **sign** helps calculate the determinant. **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.

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxMatrixMul

```
int status = CxMatrixMul (void *X, void *Y, int n, int k, int m, void *Z);
```

Purpose

Multiplies two matrices. Use the following formula to obtain the output matrix:

$$z_{i,j} = \sum_{p=0}^{k-1} x_{i,p} \times y_{p,j} \quad \text{for } i = 0, 1, 2, \ldots, n-1;\ j = 0, 1, 2, \ldots, m-1$$

Parameters

Input

Name	Type	Description
X	ComplexNum 2D array	First matrix to multiply.
Y	ComplexNum 2D array	Second matrix to multiply.
n	integer	Number of rows in **X**.
k	integer	Number of columns in **X**, and number of rows in **Y**.
m	integer	Number of columns in **Y**.

Output

Name	Type	Description
Z	ComplexNum 2D array	Result of the matrix multiplication.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Confirm that the matrix sizes are valid for matrix multiplication. You must meet the following size constraints:

- Input matrix **X** must be **n** by **k**.
- Input matrix **Y** must be **k** by **m**.
- Output matrix **Z** must be **n** by **m**.

CxMatrixNorm

```
int status = CxMatrixNorm (void *A, int n, int m, int normType, double *norm);
```

Purpose

Calculates the norm of the complex input matrix **A**. The input matrix can be square or rectangular. The norm of a matrix is a scalar that gives some measure of the size of the elements in the matrix. It is similar to the concept of magnitude or absolute value for scalar numbers.

There are different ways to calculate the norm of a matrix. The **normType** parameter indicates which type of norm to use to calculate the norm.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
normType	integer	Type of norm to calculate. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
norm	double-precision	Calculated norm of the input matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

The **normType** parameter indicates what type of norm to use to calculate the condition number. Table 2-25 shows valid norm type values.

Table 2-25. Valid Norm Type Values

Norm Type	Value	Meaning
2-norm	0	Largest singular value of **A**.
1-norm	1	Largest column sum of **A**.
Frobenius-norm	2	Square root of the sum of the diagonal elements of $\mathbf{A}^{\mathbf{T}}\mathbf{A}$, where $\mathbf{A}^{\mathbf{T}}$ is the complex conjugate transpose of **A**.
Infinite-norm	3	Largest row sum of **A**.

CxMatrixRank

```
int status = CxMatrixRank (void *A, int n, int m, double tolerance,
                           int *rank);
```

Purpose

Calculates the rank of the complex input matrix **A**. The input matrix can be square or rectangular.

The maximum number of linearly independent rows or columns of the matrix defines the rank of a matrix. The rank is always less than or equal to the minimum of the number of rows and columns of the matrix. If the rank equals this minimum value, the matrix is a full-rank matrix. Otherwise, it is a rank-deficient matrix.

The rank of a matrix can be calculated in a number of ways. `CxMatrixRank` first calculates the singular values of the input matrix and then calculates the rank as the number of singular values of the input matrix that are larger than the input tolerance.

You must specify the input **tolerance** as a positive number close to machine precision. If the matrix in your application is a full-rank matrix, any small value of **tolerance** gives the same **rank**. If the matrix in your application is a rank-deficient matrix, different values of **tolerance** can result in different values of **rank**.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
tolerance	double-precision	Tolerance value. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
rank	integer	Rank of the input matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Matrix rank is the number of singular values in the input matrix that are larger than the **tolerance**. Set **tolerance** close to *eps*, which is the smallest possible double-precision, floating-point number.

CxMul

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

Purpose

Multiplies two complex numbers, *x* and *y*. `CxMul` obtains the resulting complex number, *z*, using the following formulas:

$$zr = xr \times yr - xi \times yi$$

$$zi = xr \times yi + xi \times yr$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxMul1D

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

Purpose

Multiplies two 1D complex arrays, **x** and **y**. `CxMul1D` obtains the ith element of the resulting complex array, **z**, using the following formulas:

$$zr_i = xr_i \times yr_i - xi_i \times yi_i$$

$$zi_i = xr_i \times yi_i + xi_i \times yr_i$$

`CxMul1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxOuterProduct

```
int status = CxOuterProduct (ComplexNum x[], int nx, ComplexNum y[], int ny,
                             void *outerProduct);
```

Purpose

Calculates the outer product of the complex input vectors **x** and **y**.

Let x_i represent the elements of the **nx**-element vector **x** for $i = 0, 1, 2, \ldots, nx - 1$.

Let y_j represent the elements of the **ny**-element vector **y** for $j = 0, 1, 2, \ldots, ny - 1$.

The outer product of these two vectors is a matrix **O** of dimensions **n**-by-**m**, where the $(i, j)^{th}$ element of **O** is given by:

$$o_{i,j} = x_i \times y_j$$

Parameters

Input

Name	Type	Description
x	ComplexNum array	Input complex vector **x**.
nx	integer	Number of elements in **x**.
y	ComplexNum array	Input complex vector **y**.
ny	integer	Number of elements in **y**.

Output

Name	Type	Description
outerProduct	ComplexNum 2D array	Calculated outer product matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxPolyRoots

```
int status = CxPolyRoots (double x[], int n, ComplexNum y[]);
```

Purpose

Calculates the roots of a polynomial. The polynomial coefficients must be real numbers. The roots can be complex or real. The number of roots of the polynomial equals one less than the number of coefficients of the polynomial. Consider the following example:

$$P(x) = x^2 + 7x + 10$$

For this example, the elements of the input array **x** are [1, 7, 10]. The parameter **n** represents the number of coefficients, which is three. The output parameter **y** contains the roots of this polynomial, which are [–5, –2].

Parameters

Input

Name	Type	Description
x	double-precision array	Array of polynomial coefficients. The highest order coefficient is the first element in the array.
n	integer	Number of coefficients in **x**.

Output

Name	Type	Description
y	ComplexNum array	Array of polynomial roots. Contains **n** – 1 elements. The roots can be complex. Real roots have a zero imaginary part.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxPow

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

Purpose

Calculates the power of a complex number, *x*. `CxPow` obtains the resulting complex number, *y*, using the following formula:

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

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.
a	double-precision	Exponent.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxPseudoInverse

```
int status = CxPseudoInverse (void *A, int n, int m, double tolerance,
                              void *B);
```

Purpose

Calculates the generalized inverse of the complex input matrix **A**. The input matrix can be square or rectangular. The dimensions of the input matrix **A** are **n**-by-**m**. The dimensions of the output matrix (inverse) **B** are **m**-by-**n**.

Note ***In the case of rectangular matrices with* n < m *(number of rows less than number of columns), take the complex conjugate transpose of the input matrix before you pass it to* `CxPseudoInverse`*. The actual pseudoinverse is then the complex conjugate transpose of the matrix calculated by* `CxPseudoInverse`*.***

`CxPseudoInverse` uses the Singular Value Decomposition (SVD) technique. Define the pseudoinverse of a scalar s to be $1/s$ if s does not equal zero, and zero otherwise. Similarly, define the pseudoinverse of a diagonal matrix by transposing the matrix and then taking the scalar pseudoinverse of each entry. If $A^{\dagger}$ denotes the pseudoinverse of a matrix **A** whose singular value decomposition is given by:

$$A = USV^T$$

then:

$$A^{\dagger} = US^{\dagger}V^T$$

where $S^{\dagger}$ is the pseudoinverse of the diagonal matrix S that contains the singular values of A

The pseudoinverse exists for square and rectangular matrices. If the input matrix is square and nonsingular, the pseudoinverse is the same as the general matrix inverse.

Note ***Do not use* `CxPseudoInverse` *to calculate the inverse of a square matrix because it takes more time. Use* `CxGenInvMatrix` *instead.***

The **tolerance** parameter must be a small positive number close to machine precision. `CxPseudoInverse` sets all singular values of the input matrix smaller than **tolerance** equal to zero.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
tolerance	double-precision	Tolerance value. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
B	ComplexNum 2D array	Calculated pseudoinverse matrix. It is **m**-by-**n**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

The value of **tolerance** determines the level of accuracy in your final solution. Set **tolerance** close to eps, which is the smallest possible double-precision, floating-point number.

CxQR

```
int status = CxQR (void *A, int n, int m, void *Q, void *R);
```

Purpose

Calculates the QR factorization of the complex input matrix **A**. The input matrix can be square or rectangular.

The following formula defines the QR factorization of a **n**-by-**m** matrix **A** such that:

$$A = QR$$

where Q is an orthogonal matrix of dimensions **n**-by-**n**
R is an upper triangular matrix of dimensions **n**-by-**m**

In general, QR factorization can be calculated in many different ways. In `CxQR`, QR factorization uses the Householder algorithm. You can use QR factorization to solve linear systems with more equations than unknowns.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
Q	ComplexNum 2D array	Calculated orthogonal matrix of the QR factorization.
R	ComplexNum 2D array	Calculated upper triangular matrix of the QR factorization.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxRecip

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

Purpose

Finds the reciprocal of a complex number, *x*. `CxRecip` obtains the resulting complex number, *y*, using the following formulas:

$$yr = \frac{xr}{xr^2 + xi^2}$$

$$yi = \frac{-xi}{xr^2 + xi^2}$$

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxSpecialMatrix

```
int status = CxSpecialMatrix (int matrixType, int m, ComplexNum x[], int nx,
                              ComplexNum y[], int ny, void *Z);
```

Purpose

Generates a special type of complex matrix depending on the value of **matrixType**. There are five possible matrix types: Identity, Diagonal, Toeplitz, Vandermonde, and Companion. Table 2-26 shows each matrix type and its behavior.

Table 2-26. Matrix Types and Behaviors

Matrix Type	Behavior
Identity	`CxSpecialMatrix` generates an **m**-by-**m** identity matrix.
Diagonal	`CxSpecialMatrix` generates an **nx**-by-**nx** diagonal matrix with diagonal elements that are the elements of **x**.
Toeplitz	`CxSpecialMatrix` generates an **nx**-by-**ny** Toeplitz matrix, which has **x** as its first column and **y** as its first row. If the first element of **x** and **y** are different, `CxSpecialMatrix` uses the first element of **x**.
Vandermonde	`CxSpecialMatrix` generates an **nx**-by-**nx** Vandermonde matrix in which the kth column, for $k = 0, 1, 2, \ldots, \mathbf{nx} - 1$, equals the $\mathbf{nx} - k - 1^{\text{th}}$ power of the elements of **x**. The elements of a Vandermonde matrix are defined as follows: $b_{i,j} = x_i^{nx-j-1}$ where $i, j = 0, 1, \ldots, nx - 1$
Companion	`CxSpecialMatrix` generates an $(\mathbf{nx} - 1)$-by-$(\mathbf{nx} - 1)$ companion matrix. Assuming that the vector **x** consists of polynomial coefficients where the first element of **x** is the coefficient of the highest order and the last element of **x** is the constant term in the polynomial, `CxSpecialMatrix` constructs the corresponding companion matrix as follows: The first row of the matrix is $b_{0,j-1} = \frac{-x_j}{x_0}$ for $j = 1, 2, \ldots, \mathbf{nx} - 1$ and the remaining rows of the generated matrix form an identity matrix. The eigenvalues of a companion matrix are the roots of the corresponding polynomial.

Parameters

Input

Name	Type	Description
matrixType	integer	Type of matrix to generate. Refer to the following *Parameter Discussion* section.
m	integer	Number of rows and columns to generate when **matrixType** is Identity matrix.
x	ComplexNum array	Complex vector used to generate a Diagonal matrix, Toeplitz matrix, Vandermonde matrix, or Companion matrix.
nx	integer	Number of elements in vector **x**.
y	ComplexNum array	Second vector to use to generate the Toeplitz matrix.
ny	integer	Number of elements in vector **y**.

Output

Name	Type	Description
Z	integer	Generated matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

Table 2-27 shows valid matrix type values.

Table 2-27. Valid Matrix Type Values

Matrix Type	Value
Identity matrix	0
Diagonal matrix	1
Toeplitz matrix	2
Vandermonde matrix	3
Companion matrix	4

CxSqrt

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

Purpose

Calculates the square root of a complex number, *x*. `CxSqrt` obtains the resulting complex number, *y*, using the following formula:

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

Parameters

Input

Name	Type	Description
xr	double-precision	Real part of *x*.
xi	double-precision	Imaginary part of *x*.

Output

Name	Type	Description
yr	double-precision	Real part of *y*.
yi	double-precision	Imaginary part of *y*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxSub

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

Purpose

Subtracts two complex numbers, *x* and *y*. The resulting complex number, *z*, is obtained using the following formulas:

$$zr = xr - yr$$

$$zi = xi - yi$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision	Real part of *z*.
zi	double-precision	Imaginary part of *z*.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxSub1D

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

Purpose

Subtracts two 1D complex arrays, **x** and **y**. `CxSub1D` obtains the ith element of the resulting complex array, **z**, using the formulas:

$$zr_i = xr_i - yr_i$$

$$zi_i = xi_i - yi_i$$

`CxSub1D` can perform the operations in place; that is, the input and output complex arrays can be the same.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
zr	double-precision array	Real part of **z**.
zi	double-precision array	Imaginary part of **z**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

CxSVD

```
int status = CxSVD (void *A, int n, int m, void *U, ComplexNum s[], void *V);
```

Purpose

Calculates the Singular Value Decomposition (SVD) factorization of the complex input matrix **A**. The input matrix can be square or rectangular.

The following formula defines the SVD factorization of an **n**-by-**m** matrix **A**:

$$A = USV^H$$

where U is an orthogonal matrix of dimensions **n**-by-**m**
V is an orthogonal matrix of dimensions **m**-by-**m**
S is a diagonal matrix of dimensions **m**-by-**m**

V^H represents the complex conjugate transpose of **V**. The diagonal elements of S are called the singular values of **A** and are arranged in descending order. `CxSVD` stores the diagonal elements of S in the output array **s**.

The Singular Value Decomposition is an eigenvalue-like decomposition for rectangular matrices. You can use it to calculate the condition number of a matrix or to solve linear, least square problems. SVD is useful for ill-conditioned or rank-deficient problems because it can detect small singular values.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
U	ComplexNum 2D array	The **n**-by-**m** orthogonal matrix SVD factorization generates.
s	ComplexNum array	Array that contains the singular values of **A**, in descending order.
V	ComplexNum 2D array	The **m**-by-**m** orthogonal matrix SVD factorization generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxSVDS

```
int status = CxSVDS (void *A, int n, int m, ComplexNum s[]);
```

Purpose

CxSVDS is similar to CxSVD, but it calculates only the singular values that result from the Singular Value Decomposition factorization of the complex input matrix. The input matrix can be square or rectangular.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
s	ComplexNum array	Array that contains the singular values of **A**, in descending order.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxTrace

```
int status = CxTrace (void *A, int n, int m, ComplexNum *trace);
```

Purpose

Calculates the trace of a complex matrix. The trace of a matrix is the sum of all its diagonal elements.

`CxTrace` uses the following formula to obtain trace *t*:

$$t = \sum_{i=0}^{k-1} a_{i,i} \quad \text{where } k = \min(n, m)$$

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
trace	ComplexNum	Sum of the diagonal elements of **A**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

CxTranspose

```
int status = CxTranspose (void *A, int n, int m, void *B);
```

Purpose

Calculates the complex conjugate transpose of a 2D, complex input matrix. The following formula defines the $(i, j)^{th}$ element of the resulting matrix:

$$b_{i,j} = a_{j,i}^{*} \quad \text{where * denotes a complex conjugate}$$

If $z = x + j \times y$ is a complex number, then $x - j \times y$ is the complex conjugate of z.

Parameters

Input

Name	Type	Description
A	ComplexNum 2D array	Input complex matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
B	ComplexNum	Calculated complex conjugate transpose matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

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, `Decimate` obtains the elements of the sequence **y** using the following formula:

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

where $i = 0, 1, 2, \ldots, size - 1$

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

Parameters

Input

Name	Type	Description
x	double-precision array	Contains the input array to decimate.
n	integer	Number of elements in the input array.
dFact	integer	Amount by which to decimate **x** to form **y**.
ave	integer	Specifies whether to use averaging in decimating **x**.

Output

Name	Type	Description
y	double-precision array	Contains the output array, which is **x** decimated by the **dFact**. The size of this array must be $int(\mathbf{n}/\mathbf{dFact})$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Deconvolve

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

Purpose

Calculates the deconvolution of **y** with **x**. `Deconvolve` assumes **y** to be the result of the convolution of **x** with some system response. The function realizes the deconvolution operation using Fourier transform pairs. `Deconvolve` obtains the output sequence **h** using the following formula:

$$h = \mathrm{InvFFT}\left(\frac{Y(f)}{X(f)}\right)$$

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

Name	Type	Description
y	double-precision array	Input array to deconvolve with **x**.
ny	integer	Number of elements in **y**.
x	double-precision array	Input array with which to deconvolve **y**.
nx	integer	Number of elements in **x**. **nx** ≤ **ny**

Output

Name	Type	Description
h	double-precision array	Output array that is **y** deconvolved with **x**. This array must be (**ny** – **nx** + 1) elements long.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Determinant

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

Purpose

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

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Dimension size of input matrix.

Output

Name	Type	Description
det	double-precision	Determinant.

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

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Difference` obtains the ith element of the resulting array using the following formula:

$$y_i = \frac{x_{i+1} - x_{i-1}}{2dt} \quad \text{where } x_{-1} = xInt \text{ and } x_n = xFinal$$

`Difference` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Differentiated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, **x** and **y**. `Div1D` obtains the ith element of the output array, **z**, using the following formula:

$$z_i = \frac{x_i}{y_i}$$

`Div1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to divide.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Div2D

```
int status = Div2D (void *x, void *y, int n, int m, void *z);
```

Purpose

Divides two 2D arrays. `Div2D` obtains the $(i, j)^{th}$ element of the output array using the following formula:

$$z_{i,j} = \frac{x_{i,j}}{y_{i,j}}$$

`Div2D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

DotProduct

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

Purpose

Calculates the dot product of the **x** and **y** input arrays. `DotProduct` obtains the dot product using the following formula:

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

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input vector.
y	double-precision array	**y** input vector.
n	integer	Number of elements.

Output

Name	Type	Description
dotProd	double-precision	Dot product.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Elp_BPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
atten	double-precision	Stop band attenuation in decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Elp_BSF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
atten	double-precision	Stop band attenuation in decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that 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 you call `Elp_CascadeCoef`.

If the type and order remain the same, you can call `Elp_CascadeCoef` 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

Name	Type	Description
fs	double-precision	Specifies the sampling frequency in hertz.
fl	double-precision	Specifies the desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Specifies the desired upper cutoff frequency of the filter in hertz
ripple	double-precision	Specifies the amplitude of the stop band ripple in decibels.
atten	double-precision	Specifies the stop band attenuation, in decibels, of the IIR filter to design.

Output

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. **type** has the valid values as shown in Table 2-28.

Table 2-28. Valid type Values

Value	Description
0	lowpass filter; **fh** is not used
1	highpass filter; **fh** is not used
2	bandpass filter
3	bandstop filter

a and **b** are the reverse and forward filter coefficients. Use `IIRFiltering` to achieve the actual filtering:

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

Parameters

Input

Name	Type	Description
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 hertz.
fl	double-precision	Desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Desired higher cutoff frequency of the filter in hertz.

Name	Type	Description
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

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the designed IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the designed IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Elp_HPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
atten	double-precision	Stop band attenuation in decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Elp_LPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
atten	double-precision	Stop band attenuation in decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 you can adjust by a weighting factor. `Equi_Ripple` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.
delta	double-precision	Normalized ripple size.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, 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*.

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 digital signal processing (DSP) knowledge. You might 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, refer to the *About Windowing* section 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. `EquiRpl_BPF` is a special case of the general Parks-McClellan algorithm. `EquiRpl_BPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.
delta	double-precision	Normalized ripple size.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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.

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. `EquiRpl_BSF` is a special case of the general Parks-McClellan algorithm. `EquiRpl_BSF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.
delta	double-precision	Normalized ripple size.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `EquiRpl_HPF` is a special case of the general Parks-McClellan algorithm. `EquiRpl_HPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
f1	double-precision	Cutoff frequency 1.
f2	double-precision	Cutoff frequency 2.
n	integer	Filter length.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.
delta	double-precision	Normalized ripple size.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `EquiRpl_LPF` is a special case of the general Parks-McClellan algorithm. `EquiRpl_LPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
f1	double-precision	Cutoff frequency 1.
f2	double-precision	Cutoff frequency 2.
n	integer	Filter length.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.
delta	double-precision	Normalized ripple size.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, `ExBkmanWin` obtains the elements of **y** using the following formula:

$$y_i = x_i\left(a_0 - a_1 \times \cos\left(\frac{2\pi i}{n}\right) + a_2 \times \cos\left(\frac{4\pi i}{n}\right)\right) \quad \text{for } i = 0, \ldots, n-1$$

where $a_0 = \dfrac{7938.0}{18608.0}$

$a_1 = \dfrac{9240.0}{18608.0}$

$a_2 = \dfrac{1430.0}{18608.0}$

Parameters

Input

Name	Type	Description
x	double-precision array	Contains the input signal.
n	integer	Number of elements in the input array.

Output

Name	Type	Description
x	double-precision array	Contains the signal after `ExBkmanWin` applies the exact Blackman window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `ExpFit` obtains the ith element of the output array using the following formula:

$$z_i = ae^{bx_i}$$

`ExpFit` obtains the mean squared error (**mse**) using the following formula:

$$mse = \frac{\sum_{i=0}^{n-1} |z_i - y_i|^2}{n}$$ where n is the number of sample points

Parameters

Input

Name	Type	Description
x	double-precision array	**x** values.
y	double-precision array	**y** values.
n	integer	Number of sample points.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, `ExpWin` obtains the elements of **y** using the following formula:

$$y_i = x_i e^{ai}$$

where $a = \frac{\ln(f)}{n-1}$

f is the final value

n is the number of elements in x

Parameters

Input

Name	Type	Description
x	double-precision array	On input, **x** contains the input signal.
n	integer	Number of elements in the input array.
final	double-precision	Final value of the exponential window function.

Output

Name	Type	Description
x	double-precision array	On output, **x** contains the signal after `ExpWin` applies the exponential window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

F_Dist

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

Purpose

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

$$p = \text{prob}(F \leq f)$$

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

Parameters

Input

Name	Type	Description
f	double-precision	$-\infty < \mathbf{f} < \infty$.
n	integer	Degrees of freedom.
m	integer	Degrees of freedom.

Output

Name	Type	Description
p	double-precision	Probability $(0 \leq \mathbf{p} < 1)$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates the Fast Fourier Transform of the complex data. Let $X = x + jy$ be the complex array:

$$Y = \mathrm{FFT}(X)$$

FFT can perform the operation in place and overwrite the input arrays **x** and **y**. Refer to the *About the Fast Fourier Transform (FFT)* section in Chapter 1, *Advanced Analysis Library Overview*.

Parameters

Input

Name	Type	Description
x	double-precision array	Real part of complex array.
y	double-precision array	Imaginary part of complex array.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Real part of FFT.
y	double-precision array	Imaginary part of FFT.

Note **n** ***must be a power of two.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Generate two arrays with random numbers and calculate the 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

Calculates the Fast Hartley Transform using the following formula:

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

where X_k is the kth point of the FHT

$\mathrm{cas}(k) = \cos(k) + \sin(k)$

`FHT` can perform the operation in place and overwrite the **x** input array.

Parameters

Input

Name	Type	Description
x	double-precision array	Array to transform.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Hartley Transform.

☞ **Note** **n** ***must be a power of two.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Generate an array with random numbers and calculate 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. `FIR_Coef` returns the coefficients as the truncated impulse response of an ideal frequency response of the selected filter type. **type** has the valid values shown in Table 2-29.

Table 2-29. Valid type Values

Value	Description
`0`	lowpass filter; **fh** is not used
`1`	highpass filter; **fh** is not used
`2`	bandpass filter
`3`	bandstop filter

Use `Convolve` to achieve the actual filtering:

$$y_n = \sum_{i=0}^{taps-1} coef_i \times x_{n-1}$$

Parameters

Input

Name	Type	Description
type	integer	Controls the filter type of the FIR filter coefficients to design.
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

Name	Type	Description
coef	double-precision array	Calculated output window FIR filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, `FlatTopWin` obtains the elements of **y** using the following formula:

$$y_i = x_i\left(0.2810639 - 0.5208972\cos\left(\frac{2\pi i}{n}\right) + 0.1980399\cos\left(\frac{4\pi i}{n}\right)\right)$$

where n is the number of elements in x

Parameters

Input

Name	Type	Description
x	double-precision array	On input, **x** contains the input signal.
n	integer	Number of elements in the input array.

Output

Name	Type	Description
x	double-precision array	On output, **x** contains the signal after `FlatTopWin` applies the flat top window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ForceWin

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

Purpose

Applies a force window to the input sequence **x**:

$$x_i = \begin{cases} x_i & 0 \le i \le int\left(\left(\frac{duty}{100}\right) \times n\right) \\ 0 & \text{elsewhere} \end{cases}$$

Parameters

Input

Name	Type	Description
x	double-precision array	On input, **x** 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

Name	Type	Description
x	double-precision array	On output, **x** contains the signal after `ForceWin` applies the force window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ForwSub

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

Purpose

Solves the linear equations $a \times x = y$ by forward substitution. `ForwSub` assumes **a** to be an **n**-by-**n** lower triangular matrix with all diagonal elements equal to one. `ForwSub` obtains **x** using the following formulas:

$$x_0 = y_0$$

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

`ForwSub` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
x	double-precision array	Solution vector.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

Use ForwSub in conjunction with LU and BackSub to solve linear equations. ForwSub obtains the parameter **p** from LU. If you are not using LU, 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 */
```

FreeAnalysisMem

```
void FreeAnalysisMem (void *pointer);
```

Purpose

Frees the memory that `PeakDetector` allocated internally for the output arguments.

Parameter

Input

Name	Type	Description
pointer	void pointer	Pointer to memory to free.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following code example shows how to use `FreeAnalysisMem` in conjunction with `PeakDetector`.

```
main()
{
    double *x = NULL;
    double *amplitudes = NULL;
    double *locations = NULL;
    double *secondDerivatives = NULL;
    int err = 0;
    int xSize;

    /* Insert code here to determine xSize. */

    x = (double *)malloc (xSize * sizeof(double));

    err = PeakDetector(x, xSize, 0.01, 3, 0, 1, 0, &count, &locations,
    &amplitudes, &secondDerivatives);

    /* Memory is allocated internally by the PeakDetector function for
    the locations, amplitudes and second_derivatives outputs. Use the
    FreeAnalysisMem function to free this memory. */

    FreeAnalysisMem(locations);
    FreeAnalysisMem(amplitudes);
    FreeAnalysisMem(secondDerivatives);
}
```

FreeIIRFilterPtr

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

Purpose

Frees the IIR cascade filter structure and all internal arrays.

Parameter

Input

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 the standard deviation you specify.

Parameters

Input

Name	Type	Description
n	integer	Number of samples.
sDev	double-precision	Standard deviation you specify.
seed	integer	Initial seed value.

Output

Name	Type	Description
noise	double-precision array	Gaussian noise pattern.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

You specify the expected standard deviation of the pattern `GaussNoise` returns. The expected mean value is zero; that is, the noise array values are expected to be centered about zero. When **seed** ≥ 0, `GaussNoise` generates a new random sequence 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, `GenCosWin` obtains the elements of **y** using the following formula:

$$y_i = x_i \sum_{k=0}^{na-1} (-1)^k a_k \cos\left(\frac{2\pi k i}{n}\right)$$

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

Parameters

Input

Name	Type	Description
x	double-precision array	On input, **x** 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 **a**.

Output

Name	Type	Description
x	double-precision array	On output, **x** contains the signal after `GenCosWin` applies the general cosine window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

GenDeterminant

```
int status = GenDeterminant (void *A, int n, int matrixType, double *det);
```

Purpose

Calculates the determinant of the real, square input matrix **A**. In contrast to `Determinant`, `GenDeterminant` allows you to specify the type of matrix type with the **matrixType** parameter. The input matrix can be upper or lower triangular, general, or positive definite.

For upper or lower triangular matrices, the determinant equals the product of the diagonal elements of the matrix. For a positive definite matrix, `GenDeterminant` first calculates the Cholesky factorization of the input matrix and then calculates the determinant as the square of the determinant of the upper triangular matrix **R**. Refer to the `Cholesky` function description for more information.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input square matrix.
n	integer	Number of elements in one dimension of the matrix.
matrixType	integer	Type of the matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
det	double	Determinant of the input matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Table 2-30 shows valid matrix type values.

Table 2-30. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

GenEigenValueVector

```
int status = GenEigenValueVector (void *A, int n, int outputChoice,
                                  ComplexNum eigenValues[], void *eigenVectors);
```

Purpose

Calculates the eigenvalues λ and the corresponding eigenvectors **x** of a real, square input matrix **A**. The following formula defines the eigenvalues and the corresponding eigenvectors:

$$Ax = \lambda x$$

Although the input matrix is real, the eigenvalues and the eigenvectors can be complex if the matrix is not symmetric.

The **outputChoice** parameter determines what to calculate. Depending on your application, you can choose to calculate just the eigenvalues or to calculate both the eigenvalues and the eigenvectors.

The **eigenValues** output parameter is a 1D, complex array of **n** elements. The **eigenVectors** output parameter is an **n**-by-**n**, complex matrix (2D array). Each ith column of this matrix is the eigenvector that corresponds to the ith component of the **eigenValues**. Each eigenvector is normalized so that its largest component equals one.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input square matrix.
n	integer	Number of elements in one dimension of the matrix.
outputChoice	integer	Pass `0` for eigenvalues only; `1` for both eigenvalues and eigenvectors.

Output

Name	Type	Description
eigenValues	ComplexNum array	Resulting eigenvalues of the input matrix.
eigenVectors	ComplexNum 2D array	Resulting eigenvectors of the input matrix. You can pass `NULL` if **outputChoice** is `0`.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The following C typedef statement defines the ComplexNum structure:

```
typedef struct {
   double real;
   double imaginary;
   } ComplexNum;
```

GenInvMatrix

```
int status = GenInvMatrix (void *A, int n, int matrixType, void *B);
```

Purpose

Calculates the inverse of the real, square input matrix **A**. If **B** denotes the inverse of the matrix **A**:

$$AB = I \quad \text{where } I \text{ is the identity matrix}$$

In contrast to `InvMatrix`, `GenInvMatrix` allows you to specify the type of the input matrix with the **matrixType** parameter. The input matrix can be an upper or lower triangular matrix, a general, square matrix, or a positive definite matrix. You can save significant computation time if you properly specify the type of the matrix.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input square matrix.
n	integer	Number of elements in one dimension of the matrix.
matrixType	integer	Type of the matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
B	double-precision 2D array	Calculated inverse matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Table 2-31 shows valid matrix type values.

Table 2-31. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

GenLinEqs

```
int status = GenLinEqs (void *A, int n, int m, double y[], int matrixType,
                        double x[]);
```

Purpose

Solves for the unknown vector **x** in the linear system of equations:

$$Ax = y \tag{2-2}$$

where A is the real input matrix
y is the known vector on the right side

The input matrix can be square or rectangular. The number of elements in **y** must equal the number of rows in the matrix **A**.

`GenLinEqs` calculates the solution using the Singular Value Decomposition technique.

In the case of non-singular, square matrices, in which no row or column is a linear combination of any other row or column, `GenLinEqs` solves for the unique solution **x**.

Two possibilities exist in the case of rectangular matrices. If the number of rows is greater than the number of columns, the system has more equations than unknowns and is an overdetermined system. Because the solution that satisfies the Equation (2-2) might not exist, this procedure finds the least square solution **x**, which minimizes $\|A\|_2$. If the number of rows is less than the number of columns, the system has more unknowns than equations and is an underdetermined system. It might have infinite solutions that satisfy Equation (2-2). This procedure calculates the minimum 2-norm solution.

If the input matrix is rank-deficient, `GenLinEqs` returns a warning.

The **matrixType** parameter specifies the type of the input matrix. The input matrix can be an upper or lower triangular matrix, a general matrix, or a positive definite matrix.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input matrix. The matrix can be square or rectangular.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Name	Type	Description
y	double-precision array	Complex array that contains the set of known vector coefficients.
matrixType	integer	Type of the input matrix. Choose the matrix type correctly because it significantly affects the speed of computation.

Output

Name	Type	Description
x	double-precision array	Solution to the linear system of equations.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes. If the input matrix is rank-deficient, `GenLinEqs` returns the warning code `20001`.

Parameter Discussion

Table 2-32 shows valid matrix type values.

Table 2-32. Valid Matrix Type Values

Matrix Type	Value
General matrix	0
Positive definite	1
Upper triangular	2
Lower triangular	3

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:

$$(x_{i,0}, x_{i,1}, \ldots, x_{i,k-1}, x_i)$$

where $i = 0, 1, \ldots, n-1$

n = the number of your observation data sets

Parameters

Input

Name	Type	Description
H	double-precision 2D array	An **n**-by-**k** matrix that contains the observation data $(x_{i,0}, x_{i,1}, \ldots, x_{i,k-1})$ for $i = 0, 1, \ldots,$ **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	double-precision array	Number of elements in **y** should equal the number of rows in **H**.

Name	Type	Description
stdDev	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 `GenLSFit` ignores this parameter. The size of this array should equal **n**.
algorithm	integer	Algorithm to use to solve the multiple linear regression model. The algorithm has six selections: `0` = SVD `1` = Givens `2` = Givens2 `3` = Householder `4` = LU decomposition `5` = Cholesky algorithm

Output

Name	Type	Description
z	double-precision array	Fitted data `GenLSFit` calculates by using the coefficients **b**.
b	double-precision array	Set of coefficients that minimize χ^2, which Equation (2-3) defines.
covar	double-precision 2D array	Matrix of covariances with **k**-by-**k** elements. $c_{j,k}$ is the covariance between b_j and b_k, and $c_{j,j}$ is the variance of b_j. If you pass an empty array for **covar**, `GenLSFit` does not calculate this matrix.
mse	double-precision	Mean squared error.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

You can use `GenLSFit` to solve multiple linear regression problems and 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:

$$y_i = b_0 x_{i,0} + \ldots + b_{k-1} x_{i,k-1}$$

$$= \sum_{j=0}^{k-1} b_j x_{i,j} \quad \text{for } i = 0, 1, \ldots, n-1 \tag{2-3}$$

where b is the set of coefficients
n is the number of elements in y and the number of rows of H
k is the number of elements in b

$x_{i,j}$ is your observation data, which H contains

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

You can write Equation (2-2) as $Y = HB$.

The previous discussion leads to a multiple linear regression model, which uses several variables:

$$x_{i,0}, x_{i,1}, \ldots, x_{i,k-1}$$

to predict one variable y_i. In contrast, `LinFit`, `ExpFit`, and `PolyFit` 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 formulas in Equation (2-3) might not produce the solution. The fit problem becomes to find the coefficients B that minimize the difference between the observed data, y_i, and the predicted value:

$$z_i = \sum_{j=0}^{k-1} b_j x_{i,j}$$

`GenLSFit` uses the least chi-squares plane method to obtain the coefficients in Equation (2-3), that is, finding the solution, B, which minimizes the following quantity:

$$\chi^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - \sum_{i=0}^{k-1} b_j x_{i,j}}{\sigma_i}\right)^2 = \left|H_0 B - Y_0\right|^2 \qquad (2\text{-}4)$$

where $h_{0_{i,j}} = \frac{x_{i,j}}{\sigma_i}, y_{0_i} = \frac{y_i}{\sigma_i}$ for $i = 0, 1, \ldots, n-1$ for $j = 0, 1, \ldots, k-1$

In Equation (2-4), σ_i is the standard deviation, **stdDev**. If the measurement errors are independent and normally distributed with constant standard deviation $\sigma_i = \sigma$, Equation (2-4) 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 $b_0, b_1, \ldots, b_{k-1}$:

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

The previous equations can be written as:

$$H_0^T H_0 B = H_0^T Y \tag{2-5}$$

H_0^T is the transposition of H_0.

Equations (2-5) and (2-4) 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.

The preferred way to minimize χ^2 is to find the least squares solution of the equations:

$$H_0 B = Y_0$$

You can use QR or Singular Value Decomposition factorization to find the solution, *B*. For QR factorization, you can choose Householder, Givens, or Givens2, also called fast Givens.

Different algorithms can give you different precision. 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.

`GenLSFit` calculates the covariance matrix **covar** as follows:

$$covar = (H_0^T H_0)^{-1}$$

The best fitted curve **z** is given by the following formula:

$$z_i = \sum_{j=0}^{k-1} b_j x_{i,j}$$

`GenLSFit` obtains the **mse** using the following formula:

$$mse = \frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2$$

You can think of the polynomial fit that has a single predictor variable as a special case of multiple regression. If the observation data sets are (x_i, y_i) where $i = 0, 1, \ldots, n-1$, the model for polynomial fit is as follows:

$$y_i = \sum_{j=0}^{k-i} b_j x_i^j = b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_{k-1} x_i^{k-1} \quad \text{where } i = 0, 1, 2, \ldots, n-1 \tag{2-6}$$

Comparing Equations (2-3) and (2-6) shows that $x_{ij} = x_i^j$. In other words:

$$x_{i0} = x_i^0, x_{i,1} = x_i, x_{i,2} = x_i^2, \ldots, x_{i,k-1} = x_i^{k-1}$$

In this case, you can build **H** as follows:

$$H = \begin{bmatrix} 1 & x_0 & x_0^2 & \ldots & x_0^{k-1} \\ 1 & x_1 & x_1^2 & & x_1^{k-1} \\ \vdots & & & & \\ 1 & x_{n-1} & x_{n-1} & & x_{n-1}^{k-1} \end{bmatrix}$$

Instead of using $x_{i,j} = x_i^j$, you can choose another function formula to fit the data sets (x_i, y_i). In general, you can select $x_{i,j} = f_j(x_i)$. 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:

$$H = \begin{bmatrix} f_0(x_0) & f_1(x_0) & f_2(x_0) & \ldots & f_{k-1}(x_0) \\ f_0(x_1) & f_1(x_1) & f_2(x_1) & & f_{k-1}(x_1) \\ \vdots & & & & \\ f_0(x_{n-1}) & f_1(x_{n-1}) & f_2(x_{n-1}) & & f_{k-1}(x_{n-1}) \end{bmatrix}$$

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 `GenLSFit`. The first example uses the 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, X_1, and the price of one pound of flour, X_2. To keep things simple, the following five data points form the sample data table shown in Table 2-33.

Table 2-33. Sample Data Table

Cost (dollars) Y	Quantity X_1	Flour Price X_2
$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 formula:

$$Y = b_0 + b_1 X_1 + b_2 X_2$$

The only parameters you must 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.

Fill in **H** as follows:

$$H = \begin{bmatrix} 1 & 295 & 3 \\ 1 & 100 & 3.20 \\ 1 & 200 & 3.10 \\ 1 & 700 & 280 \\ 1 & 60 & 250 \end{bmatrix}$$

The following code is based on this example.

```
/* 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 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**, set each column to the independent functions evaluated at each *x* value. Assuming there are 100 *x* values, **H** would be the following array:

$$H = \begin{bmatrix} 1 & \sin(\omega x_0) & \cos(\omega x_0) & x_0 \\ 1 & \sin(\omega x_1) & \cos(\omega x_1) & x_1^2 \\ 1 & \sin(\omega x_2) & \cos(\omega x_2) & x_2^2 \\ \vdots & & & \\ 1 & \sin(\omega x_{99}) & \cos(\omega x_{99}) & x_{99}^2 \end{bmatrix}$$

The following code is based on this example.

```
/* 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 set of linear fit coefficients, which describe the linear curve that best represents the input data `GenLSFitCoef` uses to obtain the least squares solution technique. The general form of the **k**-dimension linear fit is as follows:

Let $i = 0, 1, \ldots, n$ be your i^{th} observation

$x_{i,j}, \ldots, x_{i,k-1}$ be $k-1$ observed x points

y_i be observed y points

Compose the **H** matrix as follows:

$$H = \begin{bmatrix} 1 & x_{0,1} & x_{0,2} & \cdots & x_{0,k-1} \\ 1 & x_{1,1} & x_{1,2} & & x_{1,k-1} \\ \vdots & & & & \\ 1 & x_{n-1,1} & x_{n-1,2} & & x_{n-1,k-1} \end{bmatrix}$$

`GenLSFitCoef` obtains the general LS linear fit coefficient b_k by minimizing the quantity:

$$Q = \sum_{i=0}^{n-1} (y_i - z_i)^2 = \sum_{i=0}^{n-1} \left(y_i - b_0 - \sum_{j=1}^{k-1} b_j x_{i,j} \right)^2$$

Parameters

Input

Name	Type	Description
H	double-precision 2D array	Input matrix that 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 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**.

Name	Type	Description
y	double-precision array	Array that contains the **y**-coordinates of the (*x*, *y*) data sets to fit.
algorithm	integer	Algorithm to use to solve the multiple linear regression model.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The algorithm has the valid selection as shown in Table 2-34.

Table 2-34. Valid Algorithm Selections

Selection	Description
`0`	Singular value decomposition (default)
`1`	Givens decomposition
`2`	Square root free Givens decomposition
`3`	Household transformation
`4`	LU decomposition
`5`	Cholesky decomposition

Each algorithm might offer different precision depending on the input data. Given the coefficient vector **b** and **H**, `GenLSFitCoef` can calculate the fitted data z_i by a simple matrix multiplication:

$$Z = H \times b$$

and can calculate the mean squared error by:

$$mse = \frac{1}{n}\sum_{i=0}^{n-1}(z_i - y_i)^2$$

GetAnalysisErrorString

```
char *message = GetAnalysisErrorString (int errorNum)
```

Purpose

Converts the error number an Analysis Library function returns into a meaningful error message.

Parameter

Input

Name	Type	Description
errorNum	integer	Status an Analysis Library function returns.

Return Value

Name	Type	Description
message	string	Explanation of error.

HamWin

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

Purpose

Applies a Hamming window to the **x** input signal. The following formula defines the Hamming window:

$$w_i = 0.54 - 0.46 \times \cos\left(\frac{2\pi i}{n}\right) \quad \text{for } i = 0, 1, \ldots, n-1$$

`HamWin` obtains the output signal using the following formula:

$$x_i = x_i \times w_i \quad \text{for } i = 0, 1, \ldots, n-1$$

`HamWin` performs the window operation in place; that is, the windowed data **x** replaces the input data **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
x	double-precision array	Windowed data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

HanWin

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

Purpose

Applies a Hanning window to the **x** input signal. The following formula defines the Hanning window:

$$w_i = 0.5 - 0.5\cos\left(\frac{2\pi i}{n}\right) \quad \text{for } i = 0, 1, \ldots, n-1$$

`HanWin` obtains the output signal using the following formula:

$$x_i = x_i \times w_i \quad \text{for } i = 0, 1, \ldots, n-1$$

`HanWin` performs the window operation in place; that is, the windowed data **x** replaces the input data **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
x	double-precision array	Windowed data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

HarmonicAnalyzer

```
int status = HarmonicAnalyzer (const double autoPowerSpectrum[],
                               int autoPowerSpectrumSize, int frameSize,
                                int numberOfHarmonics, int windowType,
                               double samplingRate, int fundamental_Frequency,
                               double harmonicAmplitude[],
                               double harmonicFrequency[], int *percent_THD,
                               int *percentTHDNoise);
```

Purpose

Finds the amplitude and frequency of the fundamental and harmonic components present in **autoPowerSpectrum**. `HarmonicAnalyzer` also calculates the percent of total harmonic distortion and the total harmonic distortion plus noise.

If the sampling rate is 1,000 Hz and the fundamental frequency is 250 Hz, the number of harmonics is limited by **samplingRate**$/(2 \times$ **fundamental_Frequency**$) = 2$. If you set **numberOfHarmonics** equal to 4, `HarmonicAnalyzer` sets the third and the fourth element of the **harmonicAmplitude** and **harmonicFrequency** array equal to 0.0.

Typically, you should pass the time-domain signal to `ScaledWindow` and then to `AutoPowerSpectrum`. You then pass the output of `AutoPowerSpectrum` to `HarmonicAnalyzer`.

Parameters

Input

Name	Type	Description
autoPowerSpectrum	double-precision array	Single-sided auto power spectrum of the windowed signal.
autoPowerSpectrumSize	integer	Number of elements in **autoPowerSpectrum**.
frameSize	integer	Number of samples in the time-domain signal array.
numberOfHarmonics	integer	Number of harmonic components.
windowType	integer	Window type the function applies to the time-domain signal.
samplingRate	double	Input sampling rate in hertz.
fundamental_Frequency	integer	Estimate of the fundamental frequency.

Output

Name	Type	Description
harmonicAmplitude	double-precision array	Amplitudes of the fundamental components and its harmonics.
harmonicFrequency	double-precision array	Frequencies of the fundamental component and its harmonics.
percent_THD	integer	Percent total harmonic distortion present in autoPowerSpectrum.
percentTHDNoise	integer	Percent total harmonic distortion plus noise present in autoPowerSpectrum.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Histogram

```
int status = Histogram (double inputArray[], int numberOfElements,
                        double base, double top, int histogramArray[],
                        double axisArray[], int intervals);
```

Purpose

Calculates the histogram of the **inputArray**. If the input sequence is

$$X = \{0, 1, 3, 3, 4, 4, 4, 5, 5, 8\}$$

the Histogram: $h(X)$ of X for eight **intervals** is

$$h(x) = \{h_0, h_1, h_2, h_3, h_4, h_5, h_6, h_7\} = \{1, 1, 0, 2, 3, 2, 0, 1\}$$

Notice that the histogram of the input sequence X is a function of X.

The function obtains Histogram: $h(X)$ as follows: `Histogram` scans the input sequence X to determine the range of values in it. Then the function establishes the interval width, Δx, according to the specified number of **intervals**,

$$\Delta x = \frac{max - min}{m}$$

where max is the maximum value found in the input sequence X
min is the minimum value found in the input sequence X
m is the specified number of **intervals**

Let χ represent the output sequence X because the histogram is a function of X. The function evaluates elements of χ using

$$\chi_i = min + 0.5 \times \Delta x + i \times \Delta x \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

`Histogram` defines the ith interval Δ_i to be the range of values from $\chi_i - 0.5 \times \Delta x$ up to but not including $\chi_i + 0.5 \times \Delta x$

$$\Delta_i = [\chi_i - 0.5 \times \Delta x : \chi_i + 0.5 \times \Delta x) \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

and defines the function $y_i(x)$ to be

$$y_i(x) = \begin{cases} 1 & \text{if } x \in \text{ union of } \Delta_i \\ 0 & \text{elsewhere} \end{cases}$$

`Histogram` has unity value if the value of x falls within the specified interval. Otherwise it is zero. Notice that the interval Δ_i is centered about χ_i, and its width is Δ_x.

The last interval, Δ_{m-1}, is defined as $[\chi_{m-i} - 0.5 \times \Delta x : \chi_{m-i} + 0.5 \times \Delta x]$. In other words, if a value equals *max*, it is counted as belonging to the last interval.

Finally, `Histogram` evaluates the histogram sequence h using

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \quad \text{for } i = 0, 1, 2, \ldots, m-1$$

where h_i represents the elements of the output sequence Histogram: $h(X)$
n is the number of elements in the input sequence X

`Histogram` obtains the histogram by counting the number of times the elements in the input array fall in the ith interval.

Parameters

Input

Name	Type	Description
inputArray	double-precision array	Input array.
numberOfElements	integer	Number of elements in **inputArray**.
base	double-precision	Lower range.
top	double-precision	Upper range.
intervals	integer	Number of intervals.

Output

Name	Type	Description
histogramArray	integer array	Histogram of **inputArray**.
axisArray	double-precision array	Histogram axis array; contains the midpoint values of the intervals.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 second order for lowpass and highpass filters, and fourth order for bandpass and bandstop filters.

filterInformation is the pointer to the filter structure that 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`, or `Bessel_CascadeCoef` before you call `IIRCascadeFiltering`.

The **filterInformation** structure contains the internal filter state information for the filtering operation so you can call `IIRCascadeFiltering` in a loop to continually filter new input array data and produce new output filtered data.

If you finish filtering one set of input data and want to filter a completely new data set, call `ResetIIRFilter` before you call `IIRCascadeFiltering` with the new data. `ResetIIRFilter` causes the internal filter state information to clear before the next filtering operation.

Parameters

Input

Name	Type	Description
x	const double-precision	Array that contains the raw data to filter.
n	integer	Specifies the number of points in both the input **x** and output **y**.
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Output

Name	Type	Description
y	double-precision array	Array that contains the output of the IIR filtering operation. The size of this array must be at least **n**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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** and forward coefficients **b** 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

Name	Type	Description
x	double-precision array	Raw data to filter.
nx	integer	Number of points in both the **x** coefficients array and the **x1** conditions array.
a	double-precision array	Array that contains the *reverse* coefficients for the IIR filtering operation.
y1	double-precision array	**y1** 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 that contains the *forward* coefficients for the IIR filtering operation.
x1	double-precision array	**x1** 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.

Output

Name	Type	Description
y1	double-precision array	on output, **y1** contains the final conditions for the next iterations.
x1	double-precision array	on output, **x1** contains the final conditions for the next iterations.
y	double-precision array	**y** array that contains the output of the IIR filtering operation. The size of this array must be at least **nx**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Impulse` obtains the ith element of the output array using the following formula:

$$x_i = \begin{cases} amp & \text{if } i = index \\ 0 & \text{otherwise} \end{cases}$$

Parameters

Input

Name	Type	Description
n	integer	Number of elements in **x**.
amp	double-precision	Amplitude.
index	integer	Impulse index.

Output

Name	Type	Description
x	double-precision array	Impulse array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates 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:

$$impulse = \mathrm{ReInvFFT}\left(\frac{Sxy(f)}{Sxx(f)}\right)$$

where $Sxy(f)$ is the two-sided cross power spectrum of the **stimulus** (x) with the **response** (y)

$Sxx(f)$ is the two-sided auto power spectrum of the stimulus

Parameters

Input

Name	Type	Description
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. **n** must be a power of 2.

Output

Name	Type	Description
impulse	double-precision array	Impulse that 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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Integrate

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

Purpose

Calculates the discrete integral of the input array. `Integrate` obtains the ith element of the resulting array using the following formula:

$$y_i = \sum_{j=0}^{i} (x_{j-1} + 4x_j + x_{j+1}) \times \frac{dt}{6} \quad \text{where } x_{-1} = xInit \text{ and } x_n = xFinal$$

`Integrate` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision array	Integrated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `InvCh_BPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `InvCh_BSF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that contains the filter coefficients and the internal filter information. You must allocate this structure by calling `AllocIIRFilterPtr` before you call 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 you call `InvCh_CascadeCoef`.

If the type and order remain the same, 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

Name	Type	Description
fs	double-precision	Specifies the sampling frequency in hertz.
fl	double-precision	Specifies the desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Specifies the desired upper cutoff frequency of the filter in hertz
atten	double-precision	Specifies the stop band attenuation, in decibels, of the IIR filter to design.

Output

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. **type** has the valid values shown in Table 2-35.

Table 2-35. Valid type Values

Value	Description
0	lowpass filter; **fh** is not used
1	highpass filter; **fh** is not used
2	bandpass filter
3	bandstop filter

a and **b** are the reverse and forward filter coefficients. Use `IIRFiltering` to achieve the actual filtering:

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

Parameters

Input

Name	Type	Description
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 hertz.
fl	double-precision	Desired lower cutoff frequency of the filter in hertz.
fh	double-precision	Desired lower cutoff frequency of the filter in hertz.

Name	Type	Description
atten	double-precision	Stop band attenuation, in decibels, of the IIR filter to design.
na	integer	Number of coefficients in the **a** coefficient array.
nb	integer	Number of coefficients in the **b** coefficient array.

Output

Name	Type	Description
a	double-precision array	Array that contains the *reverse* coefficients of the designed IIR filter.
b	double-precision array	Array that contains the *forward* coefficients of the designed IIR filter.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `InvCh_HPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `InvCh_LPF` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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 decibels.
order	integer	Filter order.

Output

Name	Type	Description
y	double-precision array	Filtered data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 $(0 \le \mathbf{p} < 1)$, such that:

$$\text{prob}(F < f) = p$$

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

Parameters

Input

Name	Type	Description
p	double-precision	Probability $(0 \le \mathbf{p} < 1)$.
n	integer	Degrees of freedom.
m	integer	Degrees of freedom.

Output

Name	Type	Description
f	double-precision	The unique number **f** such that $\text{prob}(F < f) = \mathbf{p}$, where F is a random variable from an F-distribution with **n** and **m** degrees of freedom.

☞ **Note** ***When*** $\mathbf{p} = 0$, $\mathbf{f} = 0$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates the inverse Fast Fourier Transform of the complex data. Let $X = x + jy$ be the complex array:

$$Y = \mathrm{FFT}^{-1}(X)$$

`InvFFT` performs the operation in place and overwrites the input arrays **x** and **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	Real part of complex array.
y	double-precision array	Imaginary part of complex array.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Real part of inverse FFT.
y	double-precision array	Imaginary part of inverse FFT.

Note **n** ***must be a power of two.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Generate two arrays with random numbers and calculate the 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

Calculates the inverse Fast Hartley Transform using the following formula:

$$x_i = \frac{1}{n}\sum_{k=0}^{n-1} X_k \operatorname{cas}\left(\frac{2\pi ik}{n}\right)$$

where x_i is the ith point of the inverse FHT

$\operatorname{cas}(x) = \cos(x) + \sin(x)$

`InvFHT` performs the operation in place and overwrites the **x** input array.

Parameters

Input

Name	Type	Description
x	double-precision array	Array to transform.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Inverse Fast Hartley Transform.

Note **n** ***must be a power of two.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Generate an array with random numbers and calculate 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. `InvMatrix` can perform the operation in place; that is, **x** and **y** can be the same matrices.

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Dimension size of matrix.

Output

Name	Type	Description
y	double-precision 2D array	Inverse matrix.

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

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

InvN_Dist

```
int status = InvN_Dist (double p, double *x);
```

Purpose

Calculates **x**, given a probability $(0 < \mathbf{p} < 1)$, such that:

$\text{prob}(X < x) = p$ where X is a random variable from a standard normal distribution

Parameters

Input

Name	Type	Description
p	double-precision	Probability $(0 < \mathbf{p} < 1)$.

Output

Name	Type	Description
x	double-precision	The unique number **x** such that $\text{prob}(X < \mathbf{x}) = \mathbf{p}$, where X is a random variable from a standard normal distribution.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 $(0 < \mathbf{p} < 1)$, such that:

$$\text{prob}(T < t) = p$$

where T is a random variable from a T-distribution with **n** degrees of freedom

Parameters

Input

Name	Type	Description
p	double-precision	Probability $(0 < \mathbf{p} < 1)$.
n	integer	Degrees of freedom.

Output

Name	Type	Description
t	double-precision	The unique number **t** such that $\text{prob}(T < \mathbf{t}) = \mathbf{p}$, where T is a random variable from a T-distribution with **n** degrees of freedom.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 $(0 \le \mathbf{p} < 1)$, such that:

$$\text{prob}(\chi < x) = p$$

where χ is a random variable from a chi-square distribution with **n** degrees of freedom

Parameters

Input

Name	Type	Description
p	double-precision	Probability $(0 \le \mathbf{p} < 1)$.
n	integer	Degrees of freedom.

Output

Name	Type	Description
x	double-precision	The unique number **x** such that $\text{prob}(\chi < \mathbf{x}) = \mathbf{p}$, where χ is a random variable from a chi-square distribution with **n** degrees of freedom.

Note ***When*** $\mathbf{p} = 0$, $\mathbf{x} = 0$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Ksr_BPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Table 2-36 lists some **beta** values and their equivalent windows.

Table 2-36. beta Values and Equivalent Windows

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Discrete-Time 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. `Ksr_BSF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Table 2-37 lists some **beta** values and their equivalent windows.

Table 2-37. beta Values and Equivalent Windows

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Discrete-Time 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. `Ksr_HPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
fc	double-precision	Cutoff frequency.
n	integer	Number of filter coefficients.
beta	double-precision	Shape parameter.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Table 2-38 lists some **beta** values and their equivalent windows.

Table 2-38. beta Values and Equivalent Windows

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Discrete-Time 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. `Ksr_LPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
fc	double-precision	Cutoff frequency.
n	integer	Number of filter coefficients.
beta	double-precision	Shape parameter.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Table 2-39 lists some **beta** values and their equivalent windows.

Table 2-39. beta Values and Equivalent Windows

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Discrete-Time 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 following formula defines the Kaiser window:

$$w_i = \frac{\mathrm{Io}(beta \times (1.0 - a^2)^{1/2})}{\mathrm{Io}(beta)} \quad \text{for } i = 0, 1, \ldots, n-1$$

where $a = \left|1 - \frac{2i}{n}\right|$

Io represents the *zero*th-order modified Bessel function of the first kind

`KsrWin` obtains the output signal using the formula:

$$x_i = x_i \times w_i \quad \text{for } i = 0, 1, \ldots, n-1$$

`KsrWin` performs the window operation in place; that is, the windowed data **x** replaces the input data **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x**.
beta	double-precision	Shape parameter.

Output

Name	Type	Description
x	double-precision array	Windowed data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **beta** parameter controls the shape of a Kaiser window. A larger **beta** value results in a narrower Kaiser window. Table 2-40 lists some **beta** values and their equivalent windows.

Table 2-40. beta Values and Equivalent Windows

beta	Window
0.00	Rectangular
1.33	Triangle
3.86	Hanning
4.86	Hamming
7.04	Blackman

Refer to *Discrete-Time 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

Name	Type	Description
A	double-precision 2D array	Input matrix.
y	double-precision array	Known vector.
n	integer	Dimension size of system.

Output

Name	Type	Description
x	double-precision array	Solution of vector.

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

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, **x**. `LinEv1D` obtains the ith element of the output array, **y**, using the formula:

$$y_i = a \times x_i + b$$

`LinEv1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.
a	double-precision	Multiplicative constant.
b	double-precision	Additive constant.

Output

Name	Type	Description
y	double-precision array	Linearly evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, **x**. `LinEv2D` obtains the $(i, j)^{th}$ element of the output array, **y**, using the formula:

$$y_{i,j} = a \times x_{i,j} + b$$

`LinEv2D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
y	double-precision 2D array	Linearly evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `LinFit` obtains the ith element of the output array, **z**, using the following formula:

$$z_i = slope \times x_i + intercept$$

`LinFit` obtains the mean squared error (**mse**) using the following formula:

$$mse = \frac{\sum_{i=0}^{n-1} |z_i - y_i|^2}{n}$$ where n is the number of sample points

Parameters

Input

Name	Type	Description
x	double-precision array	**x** values.
y	double-precision array	**y** values.
n	integer	Number of sample points.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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:

$$A = LU$$

where L is an **n**-by-**n** lower triangular matrix with main diagonal elements all equal to one
U is an upper triangular matrix

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input matrix.
n	integer	Dimension size.

Output

Name	Type	Description
A	double-precision 2D array	LU decomposition.
p	integer array	Permutation vector.
sign	integer	Row exchange indicator.

Note **L *and* U *output matrices overwrite the input matrix.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

After `LU` executes, `LU` replaces the input matrix **A** 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. Use `LU` in conjunction with `BackSub` and `ForwSub` to solve a set of linear equations with the same matrix **A**.

Refer to *Numerical Recipes in C: The Art of Scientific Computing* by Press, et al., for more information.

MatrixMul

```
int status = MatrixMul (void *X, void *Y, int n, int k, int m, void *Z);
```

Purpose

Multiplies two 2D input matrices, **X** and **Y**. `MatrixMul` obtains the $(i, j)^{th}$ element of the output matrix, **Z**, using the formula:

$$Z_{i,j} = \sum_{p=0}^{k-1} x_{i,p} \times y_{p,j}$$

Parameters

Input

Name	Type	Description
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

Name	Type	Description
Z	double-precision 2D array	Output matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Confirm that the array sizes are correct. You must meet the following array sizes:

- **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);
```

MatrixNorm

```
int status = MatrixNorm (void *A, int n, int m, int normType, double *norm);
```

Purpose

Calculates the norm of a real input matrix **A**. The input matrix can be square or rectangular. The norm of a matrix is a scalar that gives some measure of the size of the elements in the matrix. It is similar to the concept of magnitude or absolute value for scalar numbers.

There are different ways to calculate the norm of a matrix. The **normType** parameter indicates which type of norm to use to calculate the norm.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
normType	integer	Type of norm to calculate. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
norm	double-precision	Calculated norm of the input matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **normType** parameter indicates what type of norm to use to calculate the condition number. Table 2-41 shows valid norm type values.

Table 2-41. Valid Norm Type Values

Norm Type	Value	Meaning
2-norm	0	Largest singular value of **A**.
1-norm	1	Largest column sum of **A**.
Frobenius-norm	2	Square root of the sum of the diagonal elements of $\mathbf{A}^{\mathbf{T}}\mathbf{A}$, where $\mathbf{A}^{\mathbf{T}}$ is the complex conjugate transpose of **A**.
Infinite-norm	3	Largest row sum of **A**.

MatrixRank

```
int status = MatrixRank (void *A, int n, int m, double tolerance, int *rank);
```

Purpose

Calculates the rank of the real input matrix **A**. The input matrix can be square or rectangular.

The maximum number of linearly independent rows or columns of the matrix defines the rank of a matrix. The rank is always less than or equal to the minimum of the number of rows and columns of the matrix. If the rank equals this minimum value, the matrix is a full-rank matrix. Otherwise, it is a rank-deficient matrix.

The rank of a matrix can be calculated in a number of ways. `MatrixRank` first calculates the singular values of the input matrix and then calculates the rank as the number of singular values of the input matrix that are larger than the input tolerance.

You must specify the input **tolerance** as a positive number close to machine precision. If the matrix in your application is a full-rank matrix, any small value of **tolerance** gives the same **rank**. If the matrix in your application is a rank-deficient matrix, different values of **tolerance** can result in different values of **rank**.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
tolerance	double-precision	Tolerance value. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
rank	integer	Rank of the input matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Matrix rank is the number of singular values in the input matrix that are larger than the **tolerance**. Set **tolerance** close to eps, which is the smallest possible double-precision, floating-point number.

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 and the respective indices of the first occurrence of the maximum and minimum values.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 and the respective indices of the first occurrence of the maximum and minimum values. `MaxMin2D` scans the **X** array by rows.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates the mean, or average, value of the input array. `Mean` calculates the mean using the following formula:

$$meanval = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
meanval	double-precision	Mean value.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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, `Median` first sorts the input array in ascending order. Let S be the sorted array:

$$medianval = \begin{cases} S\left(\frac{n}{2}\right) & \text{if } n \text{ is odd} \\ 0.5 \times \left(S\left(\frac{n}{2} - 1\right) + S\left(\frac{n}{2}\right)\right) & \text{if } n \text{ is even} \end{cases}$$

Note ***The* x *input array does not change.***

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
medianval	double-precision	Median value.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Mode` determines the mode in terms of the histogram of the input array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
modeval	double-precision	Mode value.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates the moment about the mean of the input array with the specified order. `Moment` uses the following formulas to find the moment:

$$momentval = \sum_{i=0}^{n-1} \frac{(x_i - ave)^{order}}{n} \quad \text{where } ave = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.
order	integer	Moment order.

Output

Name	Type	Description
momentval	double-precision	Moment about the mean.

Note **order** ***must be greater than zero.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Mul1D` obtains the ith element of the output array using the following formula:

$$z_i = x_i \times y_i$$

`Mul1D` can perform the operation in place; that is, **z** can be the same array as either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to multiply.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Mul2D

```
int status = Mul2D (void *X, void *Y, int n, int m, void *Z);
```

Purpose

Multiplies two 2D arrays, **X** and **Y**. `Mul2D` obtains the $(i, j)^{th}$ element of the output array, **Z**, using the following formula:

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

`Mul2D` can perform the operation in place; that is, **Z** can be the same array as either **X** or **Y**.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
Z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

N_Dist

```
int status = N_Dist (double x, double *p);
```

Purpose

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

$p = \text{prob}(X \le x)$ where X is a random variable from a standard normal distribution

Parameters

Input

Name	Type	Description
x	double-precision	$-\infty < \mathbf{x} < \infty$.

Output

Name	Type	Description
p	double-precision	Probability $(0 < \mathbf{p} < 1)$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Note ***For computing the two-sided probability*** $p_2 = \text{prob}(-x \le X \le x)$***, you can use the formula*** $p_2 = 1.0 - 2 \times \text{prob}(X \le -x)$**.**

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. `Neg1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Negated values of the **x** input array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates the single-sided coherence function along with the averaged single-sided cross power spectrum, averaged single-sided frequency response, or transfer function, and impulse response from a 2D array of stimulus signals and a 2D array of response signals.

`NetworkFunctions` calculates the network functions as follows:

$$\text{avg cross power} = \text{average}(Sxy(f))$$

$$\text{avg transfer function} = \frac{\text{average}(Sxy(f))}{\text{average}(Sxx(f))}$$

$$\text{average impulse response} = \text{ReInvFFT}(\text{avg two-sided transfer function})$$

$$\text{coherence} = \frac{|\text{average } Sxy(f)|^2}{\text{average } Sxx(f) \times \text{average } Syy(f)}$$

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 that contains a time-domain signal, usually the network stimulus.
RESPONSE is a 2D array that contains 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

Name	Type	Description
STIMULUS	double-precision 2D array	Contains the time-domain signal, usually the network stimulus. The number of rows should equal **numFrames**, and the number of columns should equal **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 equal **numFrames**, and the number of columns should equal **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, or rows, the input stimulus and response arrays contain.
dt	double-precision	Sampling period of the time-domain signal, usually in seconds. **dt** = $1/fs$, where *fs* is the sampling frequency of the time-domain signal.

Output

Name	Type	Description
MAGSXY	double-precision array	Averaged single-sided cross power spectrum between the stimulus and response, in volts rms square if the input signals are in volts. If the input signals are not in volts, the results are in input signal units rms square. 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.

Name	Type	Description
MAGHF	double-precision array	Magnitude of the averaged single-sided transfer function between the stimulus and response signals. This array must be at least $\mathbf{n}/2$ elements long.
PHASEHF	double-precision array	Phase, in radians of the averaged single-sided transfer function between the stimulus and response signals.
COHERENCE	double-precision array	Averaged single-sided coherence function spectrum. The coherence function shows the frequency content of the response as a result of the stimulus and measures the validity of the network frequency response measurement. This array must be at least $\mathbf{n}/2$ elements long.
IMPULSE	double-precision array	Contains the impulse response of the network based on time-domain signals stimulus and response. `NetworkFunctions` calculates the impulse 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. $\mathbf{df} = 1/(\mathbf{n} \times \mathbf{dt})$

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

NonLinearFit

```
int status = NonLinearFit (double x[], double y[], double z[], int n,
                           ModelFun *modelFunction, double a[], int ncoef,
                           double *mse);
```

Purpose

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 $\mathbf{y} = \mathrm{f}(\mathbf{x}, \mathbf{a})$ where **a** is the set of coefficients. `NonLinearFit` also gives the best fit curve $\mathbf{y} = \mathrm{f}(\mathbf{x}, \mathbf{a})$.

You must pass a pointer to the nonlinear function f(**x**, **a**) along with a set of initial guess coefficients **a**. `NonLinearFit` does not always give the correct answer. The correct output sometimes depends on the initial choice of **a**. It is very important to verify the final result.

`NonLinearFit` calculates the output **mse** (mean squared error) using the following formula:

$$mse = \frac{\sum_{i=0}^{n-1} (y_i - \mathrm{f}(x, a))^2}{n}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Array of x-coordinates of the (x, y) data sets to fit.
y	double-precision array	Array of y-coordinates of the (x, y) data sets to fit.
n	integer	Number of elements in both the **x** and **y** arrays.
modelFunction	ModelFun	Pointer to the model function, f($\mathbf{x_i}$, **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** contains the function coefficients.

Name	Type	Description
a	double-precision array	On input, **a** gives a set of initial guess coefficients.
ncoef	integer	Number of coefficients (size of **a**).

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

NonLinearFitWithMaxIters

```
int status = NonLinearFitWithMaxIters (double x[], double y[], double z[],
                                       int n, int maximumIterations,
                                       ModelFun *modelFunction, double a[], int ncoef,
                                       double *mse);
```

Purpose

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 $\mathbf{y} = \mathrm{f}(\mathbf{x}, \mathbf{a})$ where **a** is the set of coefficients. `NonLinearFitWithMaxIters` also gives the best fit curve $\mathbf{y} = \mathrm{f}(\mathbf{x}, \mathbf{a})$.

You must pass a pointer to the nonlinear function f(**x**, **a**) along with a set of initial guess coefficients **a**. `NonLinearFit` does not always give the correct answer. The correct output sometimes depends on the initial choice of **a**. It is very important to verify the final result.

`NonLinearFitWithMaxIters` calculates the output **mse** (mean squared error) using the following formula:

$$mse = \frac{\sum_{i=0}^{n-1} (y_i - \mathrm{f}(x, a))^2}{n}$$

Note ***If*** `NonLinearFitWithMaxIters` ***reaches the maximum number of iterations without reaching a solution, it returns an error. The outputs*** **z**, **a**, ***and*** **mse** ***contain the best filtered array, best fit coefficients, and the mean square error at the end of maximum iterations.***

Parameters

Input

Name	Type	Description
x	double-precision array	Array of x-coordinates of the (x, y) data sets to fit.
y	double-precision array	Array of y-coordinates of the (x, y) data sets to fit.
n	integer	Number of elements in both the **x** and **y** arrays.

Name	Type	Description
maximumIterations	integer	Maximum number of iterations allowed.
modelFunction	ModelFun pointer	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** contains the function coefficients.
a	double-precision array	On input, **a** gives a set of initial guess coefficients.
ncoef	integer	Number of coefficients (size of **a**).

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Normal1D

```
int status = Normal1D (double x[], int n, double y[], double *ave,
                        double *sDev);
```

Purpose

Normalizes a 1D input vector. The output vector has the following form:

$$y_i = \frac{x_i - ave}{sDev}$$

where *ave* and *sDev* are the mean and the standard deviation of the input vector

Refer to the `StdDev` function description for the formulas `Normal1D` uses to find the mean and the standard deviation.

`Normal1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input vector.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Normalized vector.
ave	double-precision	Mean value of **x**.
sDev	double-precision	Standard deviation of **x**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 has the following form:

$$y_{i,j} = \frac{x_{i,j} - ave}{sDev}$$

where *ave* and *sDev* are the mean and the standard deviation of the input matrix

Refer to the `StdDev` function description for the formulas `Normal2D` uses to find the mean and the standard deviation.

`Normal2D` can perform the operation in place; that is, **X** and **Y** can be the same array.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	Input matrix.
n	integer	Size of first dimension.
m	integer	Size of second dimension.

Output

Name	Type	Description
Y	double-precision 2D array	Normalized matrix.
ave	double-precision	Mean value of **X**.
sDev	double-precision	Standard deviation of **X**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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);
```

NumericIntegration

```
int status = NumericIntegration (double x[], int n, double dt, int method,
                                 double *ir);
```

Purpose

Performs numeric integration on the data the input array **x** contains using one of the following four numeric integration methods: Trapezoidal Rule, Simpson's Rule, Simpson's 3/8 Rule, or Bode Rule. You normally obtain the data to integrate by sampling some function f(*t*) at multiples of **dt.** Your samples are f(0), f(*dt*), f(2*dt*), and so on. **dt** is the sampling step size.

Applying Multiple Methods when Number of Points Is Insufficient

If you do not provide a sufficient number of points for the integration method you choose, `NumericIntegration` applies the method you choose to all points it can. For the points that remain, `NumericIntegration` uses the next possible lower-order method.

For example, if you choose Bode Rule as the integration method, Table 2-42 shows how `NumericIntegration` evaluates the integral for different numbers of data points. If you provide 224 points and choose the Bode Rule method, `NumericIntegration` arrives at the result by performing 55 Bode Rule method partial evaluations and one Simpson's 3/8 Rule method evaluation.

Table 2-42. Bode Rule Example

Number of Points	Partial Evaluations Performed
224	55 Bode, 1 Simpson's 3/8
225	56 Bode
226	56 Bode, Trapezoidal
227	56 Bode, 1 Simpson's
228	57 Bode, 1 Simpson's 3/8

Formulas for Integration Methods

For $i = 0, 1, 2, \ldots$, int $((\mathbf{n} - 1)/k)$, where **n** is the number of data points, k is an integer dependent on the method, and **x** is the input array, Table 2-43 shows the basic formulas for each of the four integration methods.

Table 2-43. Formulas for Integration Methods

Integration Method	Formula
Trapezoidal Rule	$(dt/2) \times (x_i + x_{i+1})$ for $k = 1$
Simpson's Rule	$(dt/3) \times (x_{2i} + 4x_{2i+1} + x_{2i+2})$ for $k = 2$
Simpson's 3/8 Rule	$(dt/8) \times (3x_{3i} + 9x_{3i+1} + 9x_{3i+2} + 3x_{3i+3})$ for $k = 3$
Bode Rule	$(dt/45) \times (14x_{4i} + 64x_{4i+1} + 24x_{4i+2} + 64x_{4i+3} + 14x_{4i+4})$ for $k = 4$

Each method depends on the sampling interval, **dt**, and calculates the integral by using successive applications of the basic formula to perform partial evaluations. The number of points each partial evaluation uses represents the order of the method. The result is the sum of these successive partial evaluations.

Parameters

Input

Name	Type	Description
x	double-precision array	Array that contains data to integrate.
n	integer	Number of elements in **x**.
dt	integer	Interval size, which represents the sampling step size to use to obtain the data.
method	integer	Integration method.

Output

Name	Type	Description
ir	double	Result of the numeric integration.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Table 2-44 shows valid method parameter values.

Table 2-44. Valid Integration Method Values

Integration Method	Value
Trapezoidal Rule	0
Simpson's Rule	1
Simpson's 3/8 Rule	2
Bode Rule	3

OuterProduct

```
int status = OuterProduct (double x[], int nx, double y[], int ny,
                           void *outerProduct);
```

Purpose

Calculates the outer product of the real input vectors **x** and **y**.

Let x_i represent the elements of the **nx**-element vector **x** for $i = 0, 1, 2, \ldots, \mathbf{nx} - 1$.

Let y_j represent the elements of the **ny**-element vector **y** for $j = 0, 1, 2, \ldots, \mathbf{ny} - 1$.

The outer product of these two vectors is a matrix **O** of dimensions **n**-by-**m**, where the $(i, j)^{th}$ element of **O** is given by:

$$o_{i,j} = x_i \times y_j$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input real vector **x**.
nx	integer	Number of elements in **x**.
y	double-precision array	Input real vector **y**.
ny	integer	Number of elements in **y**.

Output

Name	Type	Description
outerProduct	double-precision 2D array	Calculated outer product matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

PeakDetector

```
int status = PeakDetector (double x[], int n, double threshold,
                           int width, int polarity, int initialize,
                           int endOfData, int *count, double **locations,
                           double **amplitudes, double **secondDerivatives);
```

Purpose

Finds the location, amplitude, and second derivatives of peaks or valleys in the input array **x**.

The input data might be a single array or consecutive blocks of data, which are useful when the application involves large data arrays or real-time processing. The **initialize** and **endOfData** parameters help you work with consecutive blocks of data. For example, if you have three blocks of data, you can perform peak detection on them according to the following pseudocode:

```
 for i = 1 to 3
   Acquire data
   if (i == 1)
      Initialize = True
   else
      Initialize = False
   if (i == 3)
      EndOfData = True
   else
      EndOfData = False
   Set width, threshold, choice
   Call PeakDectector function
   Copy the calculated locations, amplitudes and second derivatives
   to different variables so they won't be overwritten during the next
   iteration of the loop.
continue
```

`PeakDetector` is based on an algorithm that fits a quadratic polynomial to sequential groups of data points. The **width** value specifies the number of data points to use. The best choice for the value of **width** is `3`. Larger widths can reduce the apparent amplitude of peaks and shift the apparent locations.

For each peak or valley, `PeakDetector` tests the quadratic fit against the **threshold** level. `PeakDetector` ignores peaks with heights lower than the **threshold** or valleys with troughs higher than the **threshold**.

You must use the **initialize** and **endOfData** parameters to notify `PeakDetector` when you pass the first and last blocks as the **x** parameter so that `PeakDetector` can initialize and release data internal to the peak detection algorithm.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in input array **x**.
threshold	double	Threshold value to use to reject peaks and valleys that are too small.
width	integer	Span, which specifies the number of consecutive data points to use in the quadratic least squares fit.
polarity	integer	Pass `0` to detect peaks; pass `1` to detect valleys.
initialize	integer	Pass a nonzero value if the current input array is the *first* data block (or the *only* data block) to process; otherwise, pass `0`.
endOfData	integer	Pass a nonzero value if the current input array is the *last* data block (or the *only* data block) to process; otherwise, pass `0`.

Output

Name	Type	Description
count	integer	Contains the number of peaks or valleys found in the current block of data. This is the size of the three output arrays: **locations**, **amplitudes**, and **secondDerivatives**.
locations	double-precision pointer	Dynamically allocated array that contains the locations of the peaks or valleys `PeakDetector` finds in the current block of data.

Name	Type	Description
amplitudes	double-precision pointer	Dynamically allocated array that contains the amplitudes of the peaks or valleys `PeakDetector` finds in the current block of data.
secondDerivatives	double-precision pointer	Dynamically allocated array that contains the second derivatives of the peaks or valleys `PeakDetector` finds in the current block of data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The **threshold** parameter eliminates the effect of noise in the input data. `PeakDetector` ignores any peak with a fitted amplitude that is less than **threshold** and any valley with a fitted amplitude that is greater than **threshold**.

The **width** parameter value should not exceed approximately half of the half-width of the peaks or valleys. It can be much smaller for noise-free data.

The elements of the generated **locations** array represent indices from the beginning of processing, the most recent call to `PeakDetector` with a nonzero **initialize** value.

When you no longer need the **locations**, **amplitudes**, or **secondDerivatives** arrays, free them using `FreeAnalysisMem`.

PolyEv1D

```
int status = PolyEv1D (double x[], int n, double coef[], int k, double y[]);
```

Purpose

Performs a polynomial evaluation on the input array. `PolyEv1D` obtains the ith element of the output array using the following formula:

$$y_i = \sum_{j=0}^{k-1} coef_j \times x_i^j$$

`PolyEv1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.
coef	double-precision array	Coefficients array.
k	integer	Number of coefficients.

Output

Name	Type	Description
y	double-precision array	Polynomially evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The order of the polynomial equals the number of elements in the coefficients array minus one; that is, if there are **k** elements in the **coef** array, then $order = \mathbf{k} - 1$.

Example

```
/* Generate an array with random numbers, let the Ramp function
generate the coefficients { 1, 2, 3, 4, 5} 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. `PolyEv2D` obtains the $(i, j)^{th}$ element of the output array using the following formula:

$$y_{i,j} = \sum_{p=0}^{k-1} coef_p \times x_{i,j}^p$$

`PolyEv2D` can perform the operation in place; that is, **X** and **Y** can be the same array.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
Y	double-precision 2D array	Polynomially evaluated array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The order of the polynomial equals the number of elements in the coefficients array minus one; that is, if there are **k** elements in the **coef** array, then $order = \mathbf{k} - 1$.

Example

```
/* Perform a polynomial evaluation of a 2D array, let the Ramp function
generate the coefficients {1, 2, 3, 4, 5} 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. `PolyFit` obtains the ith element of the output array using the following formula:

$$z_i = \sum_{n=0}^{order} coef_n x_i^n$$

`PolyFit` obtains the mean squared error (**mse**) using the following formula:

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

where *order* is the polynomial order, and *n* is the number of sample points

If the elements in **x** are large and **order** is also large, you might see unstable results. One solution is to scale the input data elements to the range [–1: 1]. To do this, perform the following steps:

1. Find the number, for example, *k*, in **x** that has the largest magnitude, or absolute value.
2. Divide all elements in the array by the absolute value of *k*.
3. Apply `Polyfit` and rescale the results in the output array by multiplying all elements in the output array by the absolute value of *k*.

Parameters

Input

Name	Type	Description
x	double-precision array	*x* values.
y	double-precision array	*y* values.
n	integer	Number of sample points.
order	integer	Polynomial order.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Calculates 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

Name	Type	Description
x	double-precision array	Values at which the function to be interpolated is known.
y	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

Name	Type	Description
Interp_Val	double-precision	Interpolated or extrapolated value at **x_val**.
Error	double-precision	Estimate of the error in the interpolation.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

All input arrays should be the same size. If the value of **x_val** is in the range of **x**, `PolyInterp` performs interpolation; otherwise, it performs extrapolation. If **x_val** is too far from the range of **x**, **Error** might be large, and `PolyInterp` 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

Calculates the estimated power and frequency around a peak in the power spectrum of a time-domain signal. With `PowerFrequencyEstimate`, you can achieve good frequency estimates for measured peaks that lie between frequency lines on the spectrum. `PowerFrequencyEstimate` also makes corrections for the window function you use.

`PowerFrequencyEstimate` calculates the estimated frequency peak using the following formula:

$$freqPeak = \frac{\sum_{j=\frac{i-span}{2}}^{\frac{i+span}{2}} autoSpectrum_j j \times df}{\sum_{j=\frac{i-span}{2}}^{\frac{i+span}{2}} autoSpectrum_j}$$

`PowerFrequencyEstimate` calculates the estimated power peak as follows:

$$powerPeak = \frac{\sum_{j=\frac{i-span}{2}}^{\frac{i+span}{2}} autoSpectrum_j}{enbw}$$

where i = index of the *searchFreq*

df is the frequency interval, usually in hertz, as output by the `AutoPowerSpectrum` function

enbw is the equivalent noise bandwidth member of the structure *windowConstants* as output by the `ScaledWindow` function

Parameters

Input

Name	Type	Description
autoSpectrum	double-precision array	Single-sided power spectrum as output by `AutoPowerSpectrum`.
n	integer	Number of elements in the input **autoSpectrum** array.
searchFreq	double-precision	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, `PowerFrequencyEstimate` automatically searches for the maximum peak in the **autoSpectrum** array and estimates the frequency and power around the maximum peak.
windowConstants	WindowStruct	Structure that contains the following useful constants for the selected window: **enbw** is the equivalent noise bandwidth of the selected window. You can use this value to calculate the power in a given frequency span. **coherentgain** is the peak gain of the window, relative to the peak gain of the Rectangular window. `PowerFrequencyEstimate` uses this value to normalize peak signal gains to that of the Rectangular window. `ScaledWindow` creates the **windowConstants** structure.

Name	Type	Description
df	double-precision	Frequency interval, in hertz, as output by `AmpPhaseSpectrum`, `AutoPowerSpectrum`, `CrossPowerSpectrum`, `NetworkFunctions`, or `TransferFunction`.
span	integer	Number of frequency lines, or bins, around the peak to include in the peak frequency and power estimation. The estimation includes 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.

Output

Name	Type	Description
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

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Prod1D

```
int status = Prod1D (double x[], int n, double *prod);
```

Purpose

Finds the product of the **n** elements of the input array. `Prod1D` obtains the product of the elements using the following formula:

$$prod = \prod_{i=0}^{n-1} x_i$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
prod	double-precision	Product of elements.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

PseudoInverse

```
int status = PseudoInverse (void *A, int n, int m, double tolerance, void *B);
```

Purpose

Calculates the generalized inverse of the real input matrix **A**. The input matrix can be square or rectangular. The dimensions of the input matrix **A** are **n**-by-**m**. The dimensions of the output matrix (inverse) **B** are **m**-by-**n**.

Note ***In the case of rectangular matrices with*** $\mathbf{n} < \mathbf{m}$ ***(number of rows less than number of columns), take the transpose of the input matrix before you pass it to*** `PseudoInverse`***. The actual pseudoinverse is then the transpose of the result matrix*** `PseudoInverse` ***calculates.***

`PseudoInverse` uses the Singular Value Decomposition (SVD) technique. Define the pseudoinverse of a scalar s to be $1/s$ if s does not equal zero, and zero otherwise. Similarly, define the pseudoinverse of a diagonal matrix by transposing the matrix and then taking the scalar pseudoinverse of each entry. If $A^{\dagger}$ denotes the pseudoinverse of a matrix **A** whose singular value decomposition is given by:

$$A = USV^T$$

then:

$$A^{\dagger} = US^{\dagger}V^T$$

where $S^{\dagger}$ is the pseudoinverse of the diagonal matrix S that contains the singular values of A

The pseudoinverse exists for both square and rectangular matrices. If the input matrix is square and nonsingular, the pseudoinverse is the same as the general matrix inverse.

Note ***Do not use*** `PseudoInverse` ***to calculate the inverse of a square matrix because it takes more time. Use*** `GenInvMatrix` ***instead.***

The **tolerance** parameter must be a small positive number close to machine precision. `PseudoInverse` sets all singular values of the input matrix smaller than **tolerance** equal to zero.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input real matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
tolerance	double-precision	Tolerance value. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
B	double-precision 2D array	Calculated pseudoinverse matrix. It is **m**-by-**n**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The value of **tolerance** determines the level of accuracy in your final solution. Set **tolerance** close to eps, which is the smallest possible double-precision, floating-point number.

Pulse

```
int status = Pulse (int n, double amp[], int delay, int width,
                           double pulsePattern[]);
```

Purpose

Generates an array of numbers that represents the pattern of a pulse waveform. `Pulse` obtains the ith element of the output array using the formula:

$$pulsePattern_i = \begin{cases} amp & \text{if } delay \le i < (delay + width) \\ 0 & \text{otherwise} \end{cases}$$

for $i = 0, 1, 2, \ldots, n-1$

Parameters

Input

Name	Type	Description
n	integer	Number of samples.
amp	double-precision	Pulse amplitude.
delay	integer	Pulse delay.
width	integer	Pulse width.

Output

Name	Type	Description
pulsePattern	double-precision array	Pulse pattern array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `PulseParam` assumes that the input array has a *bimodal distribution*, a distribution that contains two distinct peak values.

Parameters

Input

Name	Type	Description
pulsePattern	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
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.

Name	Type	Description
fallTime	integer	Fall time.
slewRate	double-precision	Slew rate.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 of 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);
```

QR

```
int status = QR (void *A, int n, int m, int algorithm, void *Q, void *R);
```

Purpose

Calculates the QR factorization of the real input matrix **A**. The input matrix can be square or rectangular.

The following formula defines the QR factorization of a **n**-by-**m** matrix **A**:

$$A = QR$$

where Q is an orthogonal matrix of dimensions **n**-by-**n**
R is an upper triangular matrix of dimensions **n**-by-**m**

QR can calculate factorization in many ways. QR provides three methods for the factorization: Householder, Givens, and Fast Givens. You can use QR factorization to solve linear systems with more equations than unknowns.

☞ **Note** ***In the case of rectangular matrices with* n > m *(number of rows greater than number of columns), you cannot use the Fast Givens algorithm.***

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input real matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.
algorithm	integer	Algorithm to use. Refer to the following *Parameter Discussion* section.

Output

Name	Type	Description
Q	double-precision 2D array	Calculated orthogonal matrix of the QR factorization.
R	double-precision 2D array	Calculated upper triangular matrix of the QR factorization.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Table 2-45 shows valid algorithm values for the factorization methods.

Table 2-45. Valid Algorithm Values

Algorithm	Value
Householder	0
Givens	1
Fast Givens	2

QScale1D

```
int status = QScale1D (double x[], int n, double y[], double *scale);
```

Purpose

Scales the input array by its maximum absolute value. `QScale1D` can obtain the ith element of the scaled array using the following formula:

$$y_i = \frac{x_i}{scale}$$ where *scale* is the maximum absolute value in the input array

`QScale1D` determines the constant **scale**.

`QScale1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Scaled array.
scale	double-precision	Scaling constant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `QScale2D` can obtain the $(i, j)^{th}$ element of the scaled array using the following formula:

$$y_{i,j} = \frac{x_{i,j}}{scale}$$ where *scale* is the maximum absolute value of the input array

`QScale2D` determines the constant **scale**.

`QScale2D` can perform the operation in place; that is, **X** and **Y** can be the same array.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
Y	double-precision 2D array	Scaled array.
scale	double-precision	Scaling constant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Ramp

```
int status = Ramp (int n, double first, double last, double rampvals[]);
```

Purpose

Generates an output array that represents a ramp pattern. `Ramp` obtains the ith element of the output array using the formula:

$$rampvals_i = first + i\Delta x \quad \text{where } \Delta x = \frac{last - first}{n - 1}$$

Parameters

Input

Name	Type	Description
n	integer	Number of samples.
first	double-precision	Initial ramp value.
last	double-precision	Final ramp value.

Output

Name	Type	Description
rampvals	double-precision array	Ramp array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

The value of **last** does not have to be greater than the value of **first**. If the condition **last** < **first** is met, `Ramp` generates a negatively sloped ramp pattern.

Example

```
/* The following code generates the pattern  {-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. *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:

$$\left.\deg(P) = \deg(Q) = \frac{n-1}{2}\right\} \quad \text{if } n \text{ is odd}$$

$$\left.\begin{aligned}\deg(Q) &= \frac{n}{2}\\ \deg(P) &= \frac{n}{2} - 1\end{aligned}\right\} \quad \text{if } n \text{ is even} \qquad \text{where deg( ) is the order of the polynomial function}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Values at which the function to be interpolated is known.
y	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 the rational function is to be interpolated or extrapolated.

Output

Name	Type	Description
Interp_Val	double-precision	Interpolated value at **x_val**.
Error	double-precision	Estimate of the error in the interpolation.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Using This Function

All input arrays should be the same size. If the value of **x_val** is in the range of **x**, `RatInterp` performs interpolation; otherwise, it performs extrapolation. If **x_val** is too far from the range of **x**, **Error** might be large, and `RatInterp` 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

Calculates the Fourier Transform of a real input array.

Parameters

Input

Name	Type	Description
x	double-precision array	Array to transform.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Real part of Fast Fourier Transform.
y	double-precision array	Imaginary part of Fast Fourier Transform.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

n must be a power of two. `ReFFT` performs the operation in place and overwrites the input array **x**. 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 calculate the 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

Calculates the inverse Fast Fourier Transform of a complex sequence that results in a real output array.

Parameters

Input

Name	Type	Description
x	double-precision array	Real part to transform.
y	double-precision array	Imaginary part to transform.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Real inverse Fast Fourier Transform.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

n must be a power of two. `ReInvFFT` performs the operation in place and overwrites the input array **x**. The **y** array remains unchanged.

Example

```
/* Generate an array with random numbers and calculate its 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

Name	Type	Description
filterInformation	IIRFilterPtr	Pointer to the filter structure that contains the filter coefficients and the internal filter information. Refer to the `AllocIIRFilterPtr` function description for more information about the filter structure.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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:

$$y_i = x_{n-i-1} \quad \text{for } i = 0, 1, \ldots, n-1$$

`Reverse` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Reversed array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

RMS

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

Purpose

Calculates the root-mean-square (rms) value of the input array. `RMS` uses the following formula to find the rms value:

$$rms = \sqrt{\frac{\sum_{i=0}^{n-1} x_i^2}{n}}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
rmsval	double-precision	Root-mean-square value.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

SawtoothWave

```
int status = SawtoothWave (int n, double amp, double f, double *phase,
                           double x[]);
```

Purpose

Generates an array that contains a sawtooth wave. `SawtoothWave` generates the output array **x** according to the following formula:

$$x_i = amp \times \text{sawtooth}(phase + f \times 360.0 \times i)$$

$$\text{where sawtooth } (p) = \begin{cases} \dfrac{p \text{ modulo } 360.0}{180.0} & 0 \le p \text{ modulo } 360.0 < 180.0 \\ \dfrac{p \text{ modulo } 360.0}{180.0} - 2 & 180.0 \le p \text{ modulo } 360.0 < 360.0 \end{cases}$$

You can use `SawtoothWave` to simulate a continuous acquisition from a sawtooth wave function generator. The unit of the input **phase** is in degrees, and `SawtoothWave` sets **phase** to (**phase** + **f** × 360.0 × **n**) modulo 360.0 before it returns.

Parameters

Input

Name	Type	Description
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 to the initial **phase**, in degrees, of the generated signal.

Output

Name	Type	Description
phase	double-precision	Upon completion of `SawtoothWave`, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to `SawtoothWave` to simulate a continuous function generator.
x	double-precision array	Contains the generated sawtooth wave signal.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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]. `Scale1D` can obtain the ith element of the scaled array using the following formulas:

$$y_i = \frac{x_i - offset}{scale}$$

$$scale = \frac{max - min}{2}$$

$$offset = min + scale$$

where *max* and *min* are the maximum and minimum values in the input array, respectively

`Scale1D` determines the values of the constants **scale** and **offset**. `Scale1D` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
y	double-precision array	Scaled array.
offset	double-precision	Offsetting constant.
scale	double-precision	Scaling constant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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]. `Scale2D` can obtain the $(i, j)^{th}$ element of the scaled array using the following formulas:

$$y_{i,j} = \frac{x_{i,j} - offset}{scale}$$

$$scale = \frac{max - min}{2}$$

$$offset = min + scale$$

where *max* and *min* are the maximum and minimum values in the input array, respectively

`Scale2D` determines the values of the constants **scale** and **offset**.

`Scale2D` can perform the operation in place; that is, **X** and **Y** can be the same array.

Parameters

Input

Name	Type	Description
X	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
Y	double-precision 2D array	Scaled array.
offset	double-precision	Offsetting constant.
scale	double-precision	Scaling constant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ScaledWindow

```
int status = ScaledWindow (double x[], int n, int windowType,
                           WindowStruct *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 `ScaledWindow` calculates the power or amplitude spectrum of the windowed waveform, all windows provide the same level within the accuracy constraints of the window. `ScaledWindow` also returns important window constants for the window you select. These constants are useful when you use functions that perform computations on the power spectrum, such as `PowerFrequencyEstimate`.

windowType has the values shown in Table 2-46.

Table 2-46. windowType Values

Value	Description
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 that contains the following important constants for the selected window. WindowStruct is defined by the following C typedef statement.

```
typedef struct {
   double enbw;
   double coherentgain;
   } WindowStruct;
```

enbw is the equivalent noise bandwidth of the window you select. You can use this value to calculate 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

Name	Type	Description
x	double-precision array	Input array that contains time-domain signal to window.
n	integer	Number of elements in the input array.
windowType	integer	Type of the window function to apply to the input signal.

Output

Name	Type	Description
x	double-precision array	Windowed version of **x**.
windowConstants	WindowStruct	Pointer to a structure that contains the following useful constants for the selected window: **enbw** is the equivalent noise bandwidth of the selected window. You can use this value to calculate the power in a given frequency span. **coherentgain** is the peak gain of the window, relative to the peak gain of the Uniform window. `ScaledWindow` uses this value to normalize peak signal gains to that of the Uniform window.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Set1D

```
int status = Set1D (double x[], int n, double a);
```

Purpose

Sets the elements of the **x** array to a constant value.

Parameters

Input

Name	Type	Description
n	integer	Number of elements in **x**.
a	double-precision	Constant value.

Output

Name	Type	Description
x	double-precision array	Result array; set to the value of **a**.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Shift

```
int status = Shift (double x[], int n, int shifts, double y[]);
```

Purpose

Shifts the elements of the input array using the following formula:

$$y_i = x_{i-shifts}$$

You can specify the number of **shifts** to be in the positive (right) or negative (left) direction.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.
shifts	integer	Number of shifts.

Output

Name	Type	Description
y	double-precision array	Shifted array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

This is not a circular shift. `Shift` does not retain the shifted values and replaces the trailing portion of the shift with zero. `Shift` cannot perform the operation 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 that contains a sinc pattern. `Sinc` generates the output array **x** according to the following formula:

$$x_i = amp \times \mathrm{sinc}(i \times dt - delay) \quad \text{where } \mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$$

Parameters

Input

Name	Type	Description
n	integer	Number of samples to generate.
amp	double-precision	Amplitude of the resulting signal.
delay	double-precision	Shifts the peak value of the sinc pattern to the index.
dt	double-precision	Sampling interval; inversely proportional to the width of the main lobe of the sinc pattern `Sinc` generates.

Output

Name	Type	Description
x	double-precision array	Contains the sinc pattern `Sinc` generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

SinePattern

```
int status = SinePattern (int n, double amp, double phase, double cycles,
                          double sine[]);
```

Purpose

Generates an output array with a sinusoidal pattern. `SinePattern` obtains the ith element of the double-precision output array using the following formula:

$$sine_i = amp \times \sin\left(\frac{2\pi i \times cycles}{n} + \frac{\pi \times phase}{180.0}\right)$$

`SinePattern` assumes the **phase** value is in degrees and not in radians.

Parameters

Input

Name	Type	Description
n	integer	Number of samples.
amp	double-precision	Amplitude.
phase	double-precision	Phase, in degrees.
cycles	double-precision	Number of cycles.

Output

Name	Type	Description
sine	double-precision array	Sinusoidal pattern.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that contains a sine wave. `SineWave` generates the ith element of the output array **x** according to the following formula:

$$x_i = amp \times \sin(ph_i) \quad \text{where } ph_i = \frac{\pi}{180.0}(phase + f \times 360.0 \times i)$$

where the normalized frequency is the ratio of actual frequency to the sampling frequency

For example, if the actual frequency desired is 100 Hz and the sampling frequency is 500 Hz, the normalized frequency is 0.2. You can use `SineWave` to simulate a continuous acquisition from a sine wave function generator. The unit of the input **phase** is in degrees, and `SineWave` sets **phase** to (**phase** + **f** × 360.0 × **n**) modulo 360.0 before it returns.

Parameters

Input

Name	Type	Description
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	Pointer to the initial **phase**, in degrees, of the sine wave signal `SineWave` generates.

Output

Name	Type	Description
phase	double-precision	Upon completion of `SineWave`, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to `SineWave` to simulate a continuous function generator.
x	double-precision array	Contains the sine wave signal `SineWave` generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Sort

```
int status = Sort (double x[], int n, int direction, double y[]);
```

Purpose

Sorts the **x** input array in ascending or descending order. `Sort` can perform the operation in place; that is, **x** and **y** can be the same array.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements to sort.
direction	integer	`0` = ascending nonzero = descending

Output

Name	Type	Description
y	double-precision array	Sorted array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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);
```

SpecialMatrix

```
int status = SpecialMatrix (int matrixType, int m, double x[], int nx,
                            double y[], int ny, void *Z);
```

Purpose

Generates a special type of real matrix depending on the value of **matrixType**. There are five possible matrix types: Identity, Diagonal, Toeplitz, Vandermonde, and Companion. Table 2-47 shows each matrix type and its behavior.

Table 2-47. Matrix Type and Behaviors

Matrix Type	Behavior
Identity	`SpecialMatrix` generates an **m**-by-**m** identity matrix.
Diagonal	`SpecialMatrix` generates an **nx**-by-**nx** diagonal matrix with diagonal elements that are the elements of **x**.
Toeplitz	`SpecialMatrix` generates an **nx**-by-**ny** Toeplitz matrix, which has **x** as its first column and **y** as its first row. If the first element of **x** and **y** are different, `SpecialMatrix` uses the first element of **x**.
Vandermonde	`SpecialMatrix` generates an **nx**-by-**nx** Vandermonde matrix in which the kth column, for $k = 0, 1, 2, \ldots, \mathbf{nx} - 1$, equals the $(\mathbf{nx} - k - 1)^{th}$ power of the elements of **x**. The elements of a Vandermonde matrix are as follows: $b_{i,j} = x_i^{\mathbf{nx}-j-1}$ where $i, j = 0, 1, \ldots, nx - 1$
Companion	`SpecialMatrix` generates an $\mathbf{nx} - 1$-by-$\mathbf{nx} - 1$ companion matrix. Assuming that the vector **x** consists of polynomial coefficients where the first element of **x** is the coefficient of the highest order and the last element of **x** is the constant term in the polynomial, `SpecialMatrix` constructs the corresponding companion matrix as follows: The first row of the matrix is $b_{0,j-1} = \frac{-x_j}{x_0}$ for $j = 1, 2, \ldots, nx - 1$ and the remaining rows of the generated matrix form an identity matrix. The eigenvalues of a companion matrix contain the roots of the corresponding polynomial.

Parameters

Input

Name	Type	Description
matrixType	integer	Type of matrix to generate. Refer to the following *Parameter Discussion* section.
m	integer	Number of rows and columns to generate when **matrixType** is Identity matrix.
x	double-precision array	Complex vector used to generate a Diagonal matrix, Toeplitz matrix, Vandermonde matrix, or Companion matrix.
nx	integer	Number of elements in vector **x**.
y	double-precision array	Second vector to use to generate the Toeplitz matrix.
ny	integer	Number of elements in vector **y**.

Output

Name	Type	Description
Z	integer	Generated matrix.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

Table 2-48 shows valid matrix type values.

Table 2-48. Valid Matrix Type Values

Matrix Type	Value
Identity matrix	0
Diagonal matrix	1
Toeplitz matrix	2
Vandermonde matrix	3
Companion matrix	4

Spectrum

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

Purpose

Calculates the power spectrum of the input real data. `Spectrum` performs the operation in place and overwrites the input array **x**. `Spectrum` uses the following formula to obtain the power spectrum *ps*:

$$ps = \frac{|\mathrm{FFT}(x)|^2}{n^2}$$

n must be a power of two.

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	Power spectrum.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Generate an array with random numbers and calculate 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,
                                     WindowStruct windowConstants,
                                     double convertedSpectrum[], char unitString[]);
```

Purpose

Converts the input **spectrum**, which is the power, amplitude, or gain, to alternate formats, including log, decibels or *dBm*, and spectral density.

spectrum is the input array that contains a spectrum of the type the **type** selector specifies. **type** has the values shown in Table 2-49.

Table 2-49. Valid type Values

Value	Description
0	Power (volts rms square); `AutoSpectrum` calculates
1	Amplitude (volts, root-mean-square); `AmpPhaseSpectrum` calculates
2	Gain (amplitude ratio); `TransferFunction` calculates

unitString is a character array that specifies the base unit of the time domain waveform from which `SpectrumUnitConversion` calculates the input **spectrum**. The signal unit is often set to `"V"` (volts). The size of **unitString** must be at least (12 + size of **unitString**).

scalingMode has three selections for the output unit type, as shown in Table 2-50.

Table 2-50. Valid scalingMode Values

Value	Description
0	Linear
1	Decibels
2	dBm

displayUnit has the selections for the display unit, assuming volts for the base unit, as shown in Table 2-51.

Table 2-51. Valid displayUnit Values

Value	Description
0	V_{rms} (volts, root-mean-square)
1	V_{pk} (volts peak)
2	V_{rms}^2 (volts rms square)
3	V_{pk}^2 (volts peak square)
4	$V_{rms}/\sqrt{Hz}$ (volts, root-mean-square, per root hertz)
5	$V_{pk}/\sqrt{Hz}$ (volts peak per root hertz)
6	V_{rms}^2/Hz (volts rms square per hertz)
7	V_{pk}^2/Hz (volts peak square per hertz)

The last four selections are amplitude spectral density (4, 5) and power spectral density (6, 7). The structure **windowConstants** contains constants for the window you select in `ScaledWindow`. You need this input only when you use the spectral density output formats, or the last four display unit selections.

Parameters

Input

Name	Type	Description
spectrum	double-precision array	Input array that contains a spectrum of the type the **spectrum** selector specifies. It should be a power, amplitude, or gain spectrum.
n	integer	Number of elements in the input spectrum.
type	integer	Type of the input spectrum.
scalingMode	integer	Type of the scaling of the output spectrum.
displayUnits	integer	Unit of the output spectrum, assuming `"V"` for the input **unitString**.

Name	Type	Description
df	double-precision	The frequency interval, in hertz, as output by `AmpPhaseSpectrum`, `AutoPowerSpectrum`, `CrossPowerSpectrum`, `NetworkFunctions`, or `TransferFunction`.
windowConstants	WindowStruct	Structure that contains the following useful constants for the selected window: **enbw** is the equivalent noise bandwidth of the selected window. You can use this value to calculate the power in a given frequency span. **coherentgain** is the peak gain of the window, relative to the peak gain of the Rectangular window. `SpectrumUnitConversion` uses this value normalize peak signal gains to that of the Rectangular window. `ScaledWindow` outputs this structure.
unitString	string	String that contains, on input, the base unit of the analyzed signal; `"V"` for a voltage signal.

Output

Name	Type	Description
convertedSpectrum	double-precision array	Input spectrum, power, amplitude, or gain, to convert to alternate formats, including log, decibels or dBm, and spectral density. The size of this array must be at least **n**.
unitString	string	Contains, upon completion of `SpectrumUnitConversion`, the unit of the output **convertedSpectrum**. The size of this string must be at least (12 + size of **unitString**).

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 same range as x_i, given a tabulated function of the form $y_i = f(x_i)$ for $i = 0, 1, \ldots, n-1$, with $x < x_i + 1$, and given the second derivatives that specify the interpolant at the **n** nodes of **x**. The `Spline` procedure supplies the second derivatives. If **x_val** falls in the interval $[x_i, x_i + 1]$, the interpolated value is as follows:

$$Interp_Val = Ay_i + By_{i+1} + Cy''_i + Dy''_{i+1}$$

where $A = \dfrac{x_{i+1} - x_val}{x_{i+1} - x_i}$

$$B = 1 - A$$

$$C = \frac{(A^3 - A)(x_{i+1} - x_i)^2}{6}$$

$$D = \frac{(B^3 - B)(x_{i+1} - x_i)^2}{6}$$

y'' denotes the second derivative of **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	x values at which *f* is known; these values must be in ascending order.
y	double-precision array	Function values $y_i = f(x_i)$.
y2	double-precision array	Array of second derivatives that specify the interpolant.
n	integer	Number of elements in **x**, **y**, and **y2**.
x_val	double-precision	x value at which *f* is to be interpolated.

Output

Name	Type	Description
Interp_Val	double-precision	Interpolated value.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
/* Choose ascending X-values. Pick corresponding Y-values randomly.
Set boundary conditions and specify the cubic spline interpolant to
run through the points. Pick an x in the same range as X 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, given a tabulated function of the form $y_i = f(x_i)$ for $i = 0, 1, \ldots, n-1$, 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}.

You can use this array with `SpInterp` to calculate an interpolation value.

Parameters

Input

Name	Type	Description
x	double-precision array	x values at which *f* is known; these values must be in ascending order.
y	double-precision array	Function values $y_i = f(x_i)$.
n	integer	Number of elements in **x**, **y**, and **y2**.
b1	double-precision	First boundary condition x''_0.
b2	double-precision	Second boundary condition (x''_{n-1}).

Output

Name	Type	Description
y2	double-precision array	Array of second derivatives that specify the interpolant.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

These second derivatives represent the continuously differentiable curve to run though the **n** points (x_i, y_i).

Example

```
/* Choose ascending X-values. Pick corresponding Y-values randomly.
Set boundary conditions and specify the cubic spline interpolant to
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 that contains a square wave. `SquareWave` generates the output array **x** according to the following formula:

$$x_i = amp \times \text{square}(phase + f \times 360.0 \times i) \quad \text{where } f \text{ is normalized frequency}$$

$$\text{square}(p) = \begin{cases} 1.0 & 0 \le p \text{ modulo } 360.0 < \dfrac{duty}{100.0} \times 360.0 \\ -1.0 & \dfrac{duty}{100.0} \times 360.0 \le p \text{ modulo } 360.0 < 360.0 \end{cases}$$

You can use `SquareWave` to simulate a continuous acquisition from a square wave function generator. The unit of the input **phase** is in degrees, and `SquareWave` sets **phase** to (**phase** + **f** × 360.0 × **n**) modulo 360.0 before it returns.

Parameters

Input

Name	Type	Description
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.
dutyCycle	double-precision	Contains the duty cycle, in percent, of the square wave signal `SquareWave` generates.
phase	double-precision	Points to the initial **phase**, in degrees, of the square wave signal `SquareWave` generates.

Output

Name	Type	Description
phase	double-precision	Upon completion of `SquareWave`, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to `SquareWave` to simulate a continuous function generator.
x	double-precision array	Contains the square wave signal `SquareWave` generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

StdDev

```
int status = StdDev (double x[], int n, double *meanval, double *sDev);
```

Purpose

Calculates the standard deviation and the mean, or average, values of the input array. `StdDev` uses the following formulas to find the mean and the standard deviation:

$$meanval = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

$$sDev = \sqrt{\frac{\sum_{i=0}^{n-1} (x_i - meanval)^2}{n}}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
meanval	double-precision	Mean value.
sDev	double-precision	Standard deviation.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Sub1D

```
int status = Sub1D (double x[], double y[], int n, double z[]);
```

Purpose

Subtracts two 1D arrays. `Sub1D` can obtain the ith element of the output array using the following formula:

$$z_i = x_i - y_i$$

`Sub1D` can perform the operation in place; that is, **z** can be either **x** or **y**.

Parameters

Input

Name	Type	Description
x	double-precision array	**x** input array.
y	double-precision array	**y** input array.
n	integer	Number of elements to subtract.

Output

Name	Type	Description
z	double-precision array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Sub2D

```
int status = Sub2D (void *X, void *Y, int n, int m, void *Z);
```

Purpose

Subtracts two 2D arrays. `Sub2D` obtains the $(i, j)^{th}$ element of the output array using the formula:

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

`Sub2D` can perform the operation in place; that is, **Z** can be either **X** or **Y**.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
Z	double-precision 2D array	Result array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Subset1D

```
int status = Subset1D (double x[], int n, int index, int length, double y[]);
```

Purpose

Extracts a subset of the input array. The output array contains the number of elements you specify by the **length**. `Subset1D` starts copying from **x** to **y** at the **index** element of **x**.

Parameters

Input

Name	Type	Description
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 to copy to the subset.

Output

Name	Type	Description
y	double-precision array	Subset array.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Sum1D` obtains the **sum** of the elements using the following formula:

$$sum = \sum_{i=0}^{n-1} x_i$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements.

Output

Name	Type	Description
sum	double-precision	Sum of elements.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Sum2D` obtains the **sum** using the following formula:

$$sum = \sum_{i=0}^{n-1} \sum_{j=0}^{m-1} x_{i,j}$$

Parameters

Input

Name	Type	Description
X	double-precision 2D array	Input array.
n	integer	Number of elements in first dimension.
m	integer	Number of elements in second dimension.

Output

Name	Type	Description
sum	double-precision	Sum of the elements.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

SVD

```
int status = SVD (void *A, int n, int m, void *U, double s[], void *V);
```

Purpose

Calculates the Singular Value Decomposition (SVD) factorization of the real input matrix **A**. The input matrix can be square or rectangular.

The following formula defines the SVD factorization of an **n**-by-**m** matrix **A**:

$$A = USV^T$$

where U is an orthogonal matrix of dimensions **n**-by-**m**
V is an orthogonal matrix of dimensions **m**-by-**m**
S is a diagonal matrix of dimensions **m**-by-**m**

The diagonal elements of **S** are called the singular values of **A** and are arranged in descending order. SVD stores them in the output array **s**. The columns of the output matrices **U** and **V** are the corresponding singular vectors.

The Singular Value Decomposition is an eigenvalue-like decomposition for rectangular matrices. You can use it to calculate the condition number of a matrix or to solve linear, least square problems. SVD is useful for ill-conditioned or rank-deficient problems because it can detect small singular values.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input real matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
U	double-precision 2D array	The **n**-by-**m** orthogonal matrix SVD factorization generates.
s	double-precision array	Array that contains the singular values of **A**, in descending order.
V	double-precision 2D array	The **m**-by-**m** orthogonal matrix SVD factorization generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

SVDS

```
int status = SVDS(void *A, int n, int m, double s[]);
```

Purpose

SVDS is similar to SVD, but it calculates only the singular values that result from the Singular Value Decomposition factorization of the real input matrix. The input matrix can be square or rectangular.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input real matrix.
n	integer	Number of rows in **A**.
m	integer	Number of columns in **A**.

Output

Name	Type	Description
s	double-precision array	Array that contains the singular values of **A**, in descending order.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

SymEigenValueVector

```
int status = SymEigenValueVector (void *A, int n, int outputChoice,
                                  double eigenValues[], void *eigenVectors);
```

Purpose

Calculates the eigenvalues λ and the corresponding eigenvectors **x** of a real, symmetric square input matrix **A**. The following formula defines the eigenvalues and the corresponding eigenvectors:

$$Ax = \lambda x$$

The eigenvalues and the eigenvectors are all real-valued.

The **outputChoice** parameter determines what to calculate. Depending on your application, you can choose to calculate just the eigenvalues or to calculate both the eigenvalues and the eigenvectors.

The **eigenValues** output parameter is a 1D, real array of **n** elements. The **eigenVectors** output parameter is an **n**-by-**n** real matrix (2D array). Each ith column of this matrix is the eigenvector that corresponds to the ith component of the **eigenValues**. Each eigenvector is normalized so that its largest component equals one.

Parameters

Input

Name	Type	Description
A	double-precision 2D array	Input symmetric square matrix.
n	integer	Number of elements in one dimension of the matrix.
outputChoice	integer	Pass `0` for eigenvalues only; `1` for both eigenvalues and eigenvectors.

Output

Name	Type	Description
eigenValues	double-precision array	Resulting eigenvalues of the input matrix.
eigenVectors	double-precision 2D array	Resulting eigenvectors of the input matrix. You can pass `NULL` if **outputChoice** is `0`.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

T_Dist

```
int status = T_Dist (double t, int n, double *p);
```

Purpose

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

$$p = \text{prob}(T \leq t)$$

where T is a random variable from the T-distribution with **n** degrees of freedom

Parameters

Input

Name	Type	Description
t	double-precision	$-\infty < t < \infty$.
n	integer	Degrees of freedom.

Output

Name	Type	Description
p	double-precision	Probability $(0 \leq \mathbf{p} < 1)$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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**). `ToPolar` obtains the polar coordinates using the following formulas:

$$mag = \sqrt{x^2 + y^2}$$

$$phase = \arctan\left(\frac{y}{x}\right)$$

The **phase** value is in the range $[-\pi : \pi]$.

Parameters

Input

Name	Type	Description
x	double-precision	**x**-coordinate.
y	double-precision	**y**-coordinate.

Output

Name	Type	Description
mag	double-precision	Magnitude.
phase	double-precision	Phase, in radians.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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**). `ToPolar1D` obtains the ith element of the polar coordinate set using the following formulas:

$$mag_i = \sqrt{x_i^2 + y_i^2}$$

$$phase_i = \arctan\left(\frac{y_i}{x_i}\right)$$

The **phase** value is in the range $[-\pi : \pi]$.

`ToPolar1D` can perform the operations in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input

Name	Type	Description
x	double-precision array	**x**-coordinate.
y	double-precision array	**y**-coordinate.
n	integer	Number of elements.

Output

Name	Type	Description
mag	double-precision array	Magnitude.
phase	double-precision array	Phase, in radians.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

ToRect

```
int status = ToRect (double mag, double phase, double *x, double *y);
```

Purpose

Converts the polar coordinates (**mag**, **phase**) to rectangular coordinates (**x**, **y**). `ToRect` obtains the rectangular coordinates using the following formulas:

$$x = mag \times \cos(phase)$$

$$y = mag \times \sin(phase)$$

Parameters

Input

Name	Type	Description
mag	double-precision	Magnitude.
phase	double-precision	Phase, in radians.

Output

Name	Type	Description
x	double-precision	**x**-coordinate.
y	double-precision	**y**-coordinate.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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**). `ToRect1D` obtains the ith element of the rectangular set using the following formulas:

$$x_i = mag_i \times \cos(phase_i)$$

$$y_i = mag_i \times \sin(phase_i)$$

`ToRect1D` can perform the operations in place; that is, **x** and **mag**, and **y** and **phase**, can be the same arrays, respectively.

Parameters

Input

Name	Type	Description
mag	double-precision array	Magnitude.
phase	double-precision array	Phase, in radians.
n	integer	Number of elements.

Output

Name	Type	Description
x	double-precision array	**x**-coordinate.
y	double-precision array	**y**-coordinate.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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. `Trace` obtains the trace 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

Name	Type	Description
X	double-precision 2D array	Input matrix.
n	integer	Size of matrix.

Output

Name	Type	Description
traceval	double-precision	Trace.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

TransferFunction

```
int status = TransferFunction (double stimulus[], double response[], int n,
                               double dt, double magHf[], double phaseHf[],
                               double *df);
```

Purpose

Calculates 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.

`TransferFunction` calculates the transfer function **Hf** as follows:

$$Hf = \frac{\mathrm{FFT}(response)}{\mathrm{FFT}(stimulus)}$$

and transforms this result to single-sided magnitude and phase.

Parameters

Input

Name	Type	Description
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 **stimulus** and **response** arrays. **n** must be a power of 2.
dt	double-precision	Sampling period of the time-domain signals, usually in seconds. **dt** $= 1/fs$, where *fs* is the sampling frequency of the time-domain signals.

Output

Name	Type	Description
magHf	double-precision array	Magnitude of the averaged single-sided transfer function between the **stimulus** and **response** signals. This array must be at least $\mathbf{n}/2$ elements long.
phaseHf	double-precision array	Phase, in radians, of the averaged single-sided transfer function between the **stimulus** and **response** signals. This array must be at least $\mathbf{n}/2$ elements long.
df	double-precision	Points to the frequency interval, in hertz, if **dt** is in seconds. $\mathbf{df} = 1/(\mathbf{n} \times \mathbf{dt})$

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Transpose

```
int status = Transpose (void *x, int n, int m, void *y);
```

Purpose

Finds the transpose of a 2D input matrix. `Transpose` obtains the $(i, j)^{th}$ element of the resulting matrix using the following formula:

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

Parameters

Input

Name	Type	Description
x	double-precision 2D array	Input matrix.
n	integer	Size of first dimension.
m	integer	Size of second dimension.

Output

Name	Type	Description
y	double-precision 2D array	Transpose matrix.

Note ***If the input matrix has* n*-by-*m *dimensions, the output matrix must have* m*-by-*n *dimensions.***

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Triangle

```
int status = Triangle (int n, double amp, double tri[]);
```

Purpose

Generates an output array that has a triangular pattern. `Triangle` obtains the ith element of the double-precision output array using the following formulas:

$$tri_i = \begin{cases} amp\left(\dfrac{1-|2i-n|}{n}\right) & \text{if } n \text{ is even} \\ amp\left(\dfrac{1-|2i-n+1|}{n-1}\right) & \text{if } n \text{ is odd} \end{cases}$$

Parameters

Input

Name	Type	Description
n	integer	Number of samples.
amp	double-precision	Amplitude.

Output

Name	Type	Description
tri	double-precision array	Triangular pattern.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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 that contains a triangle wave. `TriangleWave` generates the output array **x** according to the following formula:

$$x_i = amp \times \mathrm{tri}(phase + f \times 360.0 \times i) \quad \text{where } f = \text{ frequency, cycles/sample}$$

$$\mathrm{tri}(p) = \begin{cases} 2 \times \dfrac{p \text{ modulo } 360.0}{180.0} & 0 \le p \text{ modulo } 360.0 < 90.0 \\ 2 \times \dfrac{1 - (p \text{ modulo } 360.0)}{180.0} & 90.0 \le p \text{ modulo } 360.0 < 270.0 \\ 2 \times \dfrac{p \text{ modulo } 360.0}{180.0 - 2.0} & 270.0 \le p \text{ modulo } 360.0 < 360.0 \end{cases}$$

You can use `TriangleWave` to simulate a continuous acquisition from a triangle wave function generator. The unit of the input **phase** is in degrees, and `TriangleWave` sets **phase** to (**phase** + **f** × 360.0 × **n**) modulo 360.0 before it returns.

Parameters

Input

Name	Type	Description
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	Points to the initial **phase**, in degrees, of the triangle wave signal `TriangleWave` generates.

Output

Name	Type	Description
phase	double-precision	Upon completion of `TriangleWave`, **phase** points to the **phase** of the next portion of the signal. Use this parameter in the next call to `TriangleWave` to simulate a continuous function generator.
x	double-precision array	Contains the triangle wave signal `TriangleWave` generates.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

TriWin

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

Purpose

Applies a triangular window to the **x** input signal. The following formula defines the triangular window:

$$w_i = \frac{1 - |2 \times i - n|}{n} \quad \text{for } i = 0, 1, \ldots, n-1$$

`TriWin` obtains the output signal using the following formula:

$$x_i = x_i \times w_i \quad \text{for } i = 0, 1, \ldots, n-1$$

`TriWin` performs the window operation in place; that is, the windowed data **x** replaces the input data **x**.

Parameters

Input

Name	Type	Description
x	double-precision array	Input data.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
x	double-precision array	Windowed data.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Name	Type	Description
n	integer	Number of samples.
seed	integer	Seed value.

Output

Name	Type	Description
x	double-precision array	Random pattern between 0 and 1.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

When **seed** ≥ 0, `Uniform` generates a new random sequence using the seed value.
When **seed** < 0, the previously generated random sequence continues.

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 $[-\pi : \pi]$ to create continuous values. `UnWrap1D` overwrites the input array **phase**.

Parameters

Input

Name	Type	Description
phase	double-precision array	Array of discontinuous phase values.
n	integer	Number of elements.

Output

Name	Type	Description
phase	double-precision array	Array of continuous phase values.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Variance

```
int status = Variance (double x[], int n, double *meanval, double *var);
```

Purpose

Calculates the variance and the mean, or average, values of the input array. `Variance` uses the following formulas to find the mean and the variance:

$$meanval = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

$$var = \frac{\sum_{i=0}^{n-1} (x_i - meanval)^2}{n}$$

Parameters

Input

Name	Type	Description
x	double-precision array	Input array.
n	integer	Number of elements in **x**.

Output

Name	Type	Description
meanval	double-precision	Mean value.
var	double-precision	Variance.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

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

Name	Type	Description
n	integer	Number of samples.
amp	double-precision	Amplitude.
seed	integer	Seed value.

Output

Name	Type	Description
noise	double-precision array	Noise pattern.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

When **seed** ≥ 0, `WhiteNoise` generates a new random sequence using the seed value. When **seed** < 0, the previously generated random sequence continues.

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. `Wind_BPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

windType selects one of the five windows as shown in Table 2-52.

Table 2-52. Valid windType Values

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 *Discrete-Time 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. `Wind_BSF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
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

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

windType selects one of the five windows as shown in Table 2-53.

Table 2-53. Valid windType Values

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 *Discrete-Time 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. `Wind_HPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
fc	double-precision	Cutoff frequency.
n	integer	Number of filter coefficients.
windType	integer	Window type.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

windType selects one of the five windows as shown in Table 2-54.

Table 2-54. Valid windType Values

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 *Discrete-Time 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. `Wind_LPF` generates only the filter coefficients; it does not actually perform data filtering.

Parameters

Input

Name	Type	Description
fs	double-precision	Sampling frequency.
fc	double-precision	Cutoff frequency.
n	integer	Number of filter coefficients.
windType	integer	Window type.

Output

Name	Type	Description
coef	double-precision array	Filter coefficients.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Parameter Discussion

windType selects one of the five windows as shown in Table 2-55.

Table 2-55. Valid windType Values

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 *Discrete-Time 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**:

$$p = \text{prob}(X \leq x)$$

where X is a random variable from the χ^2-distribution with **n** degrees of freedom

Parameters

Input

Name	Type	Description
x	double-precision	$-\infty < x < \infty$.
n	integer	Degrees of freedom.

Output

Name	Type	Description
p	double-precision	Probability $(0 \leq \mathbf{p} < 1)$.

Return Value

Name	Type	Description
status	integer	Refer to Appendix A for error codes.

Example

```
double x, p;
int n;
x = -123.456;
n = 6;
XX_Dist (x, n, &p);
/* Now p = 0 because chi-square distributed variables are
non-negative. */
```

A

Error Codes

This appendix contains error codes the Advanced Analysis Library functions return. If an error condition occurs during a call to any of the functions in the LabWindows Analysis Library, the return value **status** contains 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`	Specified conditions on the input arrays have not been met.
`AttenGTRippleAnlysErr`	`-20028`	Attenuation must be greater than the ripple amplitude.
`AttenGTZeroAnlysErr`	`-20025`	Attenuation must be greater than zero.
`BalanceAnlysErr`	`-20047`	Data is unbalanced.
`BandSpecAnlysErr`	`-20023`	Invalid band specification.
`BaseGETopAnlysErr`	`-20101`	Base must be less than top.
`BetaFuncAnlysErr`	`-20057`	Parameter to the beta function must meet the condition $0 < p < 1$.
`CategoryAnlysErr`	`-20055`	Invalid number of categories or samples.
`ColumnAnlysErr`	`-20051`	First column in the X matrix must be all ones.
`CyclesAnlysErr`	`-20012`	Number of cycles must meet the condition $0 < \text{cycles} \leq \text{samples}$.
`DataAnlysErr`	`-20045`	Total number of data points must be equal to product of $(\text{levels}/\text{each factor}) \times (\text{observations}/\text{cell})$.
`DecFactAnlysErr`	`-20022`	Decimating factor must meet the condition $0 < \text{decimating factor} \leq \text{samples}$.

Table A-1. Advanced Analysis Library Error Codes, Sorted Alphabetically (Continued)

Symbolic Name	Code	Error Message
DelayWidthAnlysErr	-20014	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	x-values must be distinct.
DivByZeroAnlysErr	-20060	Divide by zero.
DtGTZeroAnlysErr	-20016	dt or dx must be greater than zero.
EqRplDesignAnlysErr	-20031	Filter cannot be designed with the specified input parameters.
EqSamplesAnlysErr	-20002	Input sequences must be the same size.
EvenSizeAnlysErr	-20033	Number of coefficients must be odd for this filter.
FactorAnlysErr	-20043	Level of factor is outside the allowable range.
FreedomAnlysErr	-20052	Invalid degrees of freedom.
IndexLengthAnlysErr	-20018	Index and length must meet the condition $0 \le (\text{index} + \text{length}) < \text{samples}$.
IndexLTSamplesAnlysErr	-20017	Index must meet the condition $0 \le \text{index} < \text{samples}$.
InvSelectionAnlysErr	-20061	Invalid selection.
IIRFilterInfoAnlysErr	-20066	Information in the IIR filter structure is invalid.
LevelsAnlysErr	-20042	Number of levels is outside the allowable range.
MaxIterAnlysErr	-20062	Maximum iteration exceeded.
MixedSignAnlysErr	-20036	Second array must be all positive or negative and nonzero.
ModelAnlysErr	-20048	Random Effect model was requested when the Fixed Effect model is required.
NoAnlysErr	0	No error; the call was successful.

Table A-1. Advanced Analysis Library Error Codes, Sorted Alphabetically (Continued)

Symbolic Name	Code	Error Message
NyquistAnlysErr	-20020	Cut-off frequency, *fc*, must meet the condition $0 \le fc \le (fs/2)$.
ObservationsAnlysErr	-20044	There must be at least one observation.
OddSizeAnlysErr	-20034	Number of coefficients must be even for this filter.
OrderGEZeroAnlysErr	-20103	Order must be greater than or equal to zero.
OrderGTZeroAnlysErr	-20021	Order must be greater than zero.
OutOfMemAnlysErr	-20001	There is not enough memory left to perform the specified routine.
PoleAnlysErr	-20050	Interpolating function has a pole at the requested value.
PolyAnlysErr	-20063	Invalid polynomial.
PowerOfTwoAnlysErr	-20009	Size of the input array must be a valid power of two: $\text{size} = 2^m$.
ProbabilityAnlysErr	-20053	Probability must meet the condition $0 < p < 1$.
RippleGTZeroAnlysErr	-20024	Ripple must be greater than zero.
SamplesGEThreeAnlysErr	-20007	Number of samples must be greater than or equal to three.
SamplesGETwoAnlysErr	-20006	Number of samples must be greater than or equal to two.
SamplesGEZeroAnlysErr	-20004	Number of samples must be greater than or equal to zero.
SamplesGTZeroAnlysErr	-20003	Number of samples must be greater than zero.
ShiftRangeAnlysErr	-20102	Shifts must meet the condition $\lvert\text{shifts}\rvert < \text{samples}$.
SingularMatrixAnlysErr	-20041	Input matrix is singular. The system of equations cannot be solved.
SizeGTOrderAnlysErr	-20037	Array size must be greater than the order.
SquareMatrixAnlysErr	-20040	Input matrix must be a square matrix.

Table A-1. Advanced Analysis Library Error Codes, Sorted Alphabetically (Continued)

Symbolic Name	Code	Error Message
TableAnlysErr	-20056	Contingency table has a negative number.
UpperGELowerAnlysErr	-20019	Upper value must be greater than or equal to the lower value.
ZeroVectorAnlysErr	-20065	Elements of the vector cannot be all zero.

Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically

Code	Symbolic Name	Error Message
0	NoAnlysErr	No error; the call was successful.
-20001	OutOfMemAnlysErr	There is not enough memory left to perform the specified routine.
-20002	EqSamplesAnlysErr	Input sequences must be the same size.
-20003	SamplesGTZeroAnlysErr	Number of samples must be greater than zero.
-20004	SamplesGEZeroAnlysErr	Number of samples must be greater than or equal to zero.
-20006	SamplesGETwoAnlysErr	Number of samples must be greater than or equal to two.
-20007	SamplesGEThreeAnlysErr	Number of samples must be greater than or equal to three.
-20008	ArraySizeAnlysErr	Specified conditions on the input arrays have not been met.
-20009	PowerOfTwoAnlysErr	Size of the input array must be a valid power of two: size $= 2^m$.
-20012	CyclesAnlysErr	Number of cycles must meet the condition $0 < \text{cycles} \leq \text{samples}$.
-20014	DelayWidthAnlysErr	Delay and width must meet the condition $0 \leq (\text{delay} + \text{width}) < \text{samples}$.
-20016	DtGTZeroAnlysErr	dt or dx must be greater than zero.
-20017	IndexLTSamplesAnlysErr	Index must meet the condition: $0 \leq \text{index} < \text{samples}$.

Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically (Continued)

Code	Symbolic Name	Error Message
-20018	IndexLengthAnlysErr	Index and length must meet the condition $0 \le (\text{index} + \text{length}) < \text{samples}$.
-20019	UpperGELowerAnlysErr	Upper value must be greater than or equal to the lower value.
-20020	NyquistAnlysErr	Cut-off frequency, *fc*, must meet the condition: $0 \le fc \le (fs/2)$.
-20021	OrderGTZeroAnlysErr	Order must be greater than zero.
-20022	DecFactAnlysErr	Decimating factor must meet the condition $0 < \text{decimating factor} \le \text{samples}$.
-20023	BandSpecAnlysErr	Invalid band specification.
-20024	RippleGTZeroAnlysErr	Ripple must be greater than zero.
-20025	AttenGTZeroAnlysErr	Attenuation must be greater than zero.
-20028	AttenGTRippleAnlysErr	Attenuation must be greater than the ripple amplitude.
-20031	EqRplDesignAnlysErr	Filter cannot be designed with the specified input parameters.
-20033	EvenSizeAnlysErr	Number of coefficients must be odd for this filter.
-20034	OddSizeAnlysErr	Number of coefficients must be even for this filter.
-20036	MixedSignAnlysErr	Second array must be all positive or negative and nonzero.
-20037	SizeGTOrderAnlysErr	Array size must be greater than the order.
-20040	SquareMatrixAnlysErr	Input matrix must be a square matrix.
-20041	SingularMatrixAnlysErr	Input matrix is singular. The system of equations cannot be solved.
-20042	LevelsAnlysErr	Number of levels is outside the allowable range.
-20043	FactorAnlysErr	Level of factor is outside the allowable range.
-20044	ObservationsAnlysErr	There must be at least one observation.

Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically (Continued)

Code	Symbolic Name	Error Message
-20045	DataAnlysErr	Total number of data points must be equal to product of (levels/each factor) × (observations/cell).
-20047	BalanceAnlysErr	Data is unbalanced.
-20048	ModelAnlysErr	Random Effect model was requested when the Fixed Effect model is required.
-20049	DistinctAnlysErr	x-values must be distinct.
-20050	PoleAnlysErr	Interpolating function has a pole at the requested value.
-20051	ColumnAnlysErr	First column in the X matrix must be all ones.
-20052	FreedomAnlysErr	Invalid degrees of freedom.
-20053	ProbabilityAnlysErr	Probability must meet the condition $0 < p < 1$.
-20055	CategoryAnlysErr	Invalid number of categories or samples.
-20056	TableAnlysErr	Contingency table has a negative number.
-20057	BetaFuncAnlysErr	Parameter to the beta function must meet the condition $0 < p < 1$.
-20058	DimensionAnlysErr	Invalid number of dimensions or dependent variables.
-20060	DivByZeroAnlysErr	Divide by zero.
-20061	InvSelectionAnlysErr	Invalid selection.
-20062	MaxIterAnlysErr	Maximum iteration exceeded.
-20063	PolyAnlysErr	Invalid polynomial.
-20065	ZeroVectorAnlysErr	Elements of the vector cannot be all zero.
-20066	IIRFilterInfoAnlysErr	Information in the IIR filter structure is invalid.
-20101	BaseGETopAnlysErr	Base must be less than top.

Table A-2. Advanced Analysis Library Error Codes, Sorted Numerically (Continued)

Code	Symbolic Name	Error Message
-20102	ShiftRangeAnlysErr	Shifts must meet the condition $\lvert shifts \rvert < samples$.
-20103	OrderGEZeroAnlysErr	Order must be greater than or equal to zero.

B

Customer Communication

For your convenience, this appendix contains forms to help you gather the information necessary to help us solve your technical problems and a form you can use to comment on the product documentation. When you contact us, we need the information on the Technical Support Form and the configuration form, if your manual contains one, about your system configuration to answer your questions as quickly as possible.

National Instruments has technical assistance through electronic, fax, and telephone systems to quickly provide the information you need. Our electronic services include a bulletin board service, an FTP site, a fax-on-demand system, and e-mail support. If you have a hardware or software problem, first try the electronic support systems. If the information available on these systems does not answer your questions, we offer fax and telephone support through our technical support centers, which are staffed by applications engineers.

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
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: 01 48 65 15 59
Up to 9,600 baud, 8 data bits, 1 stop bit, no parity

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.

Fax-on-Demand Support

Fax-on-Demand is a 24-hour information retrieval system containing a library of documents on a wide range of technical information. You can access Fax-on-Demand from a touch-tone telephone at 512 418 1111.

E-Mail Support (Currently USA Only)

You can submit technical support questions to the applications engineering team through e-mail at the Internet address listed below. Remember to include your name, address, and phone number so we can contact you with solutions and suggestions.

`support@natinst.com`

Telephone and Fax 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.

Country	Telephone	Fax
Australia	03 9879 5166	03 9879 6277
Austria	0662 45 79 90 0	0662 45 79 90 19
Belgium	02 757 00 20	02 757 03 11
Brazil	011 288 3336	011 288 8528
Canada (Ontario)	905 785 0085	905 785 0086
Canada (Quebec)	514 694 8521	514 694 4399
Denmark	45 76 26 00	45 76 26 02
Finland	09 725 725 11	09 725 725 55
France	01 48 14 24 24	01 48 14 24 14
Germany	089 741 31 30	089 714 60 35
Hong Kong	2645 3186	2686 8505
Israel	03 6120092	03 6120095
Italy	02 413091	02 41309215
Japan	03 5472 2970	03 5472 2977
Korea	02 596 7456	02 596 7455
Mexico	5 520 2635	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
United Kingdom	01635 523545	01635 523154
United States	512 795 8248	512 794 5678

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 (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 reproduce the problem: _____

_____

_____

_____

_____

LabWindows/CVI 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. Completing this form accurately before contacting National Instruments for technical support helps our applications engineers answer your questions more efficiently.

National Instruments Products

Hardware revision _____

Interrupt level of hardware _____

DMA channels of hardware _____

Base I/O address of hardware _____

Programming choice _____

National Instruments software _____

Other boards in system _____

Base I/O address of other boards _____

DMA channels of other boards _____

Interrupt level of other boards _____

Other Products

Computer make and model _____

Microprocessor _____

Clock frequency or speed _____

Type of video board installed _____

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: February 1998

Part Number: 320686D-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 _____

_____

E-Mail Address _____

Phone (___) _____ Fax (___) _____

Mail to: Technical Publications
National Instruments Corporation
6504 Bridge Point Parkway
Austin, Texas 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/Symbols

1D one-dimensional.

2D two-dimensional.

A

ANOVA analysis of variance.

C

cm centimeters.

D

DFT Discrete Fourier Transform.

DSP digital signal processing.

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 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.

H

Hz	hertz.

I

IDFT	inverse Discrete Fourier Transform.
IIR	infinite impulse response.

M

mse	mean squared error.

R

rms	root-mean-square.

S

sidelobes — The lobes that have lower peak amplitude than the mainlobe and that lie on either side of the mainlobe in the frequency spectrum. The mainlobe is the lobe that has the highest peak amplitude and that is usually centered around zero frequency in the frequency spectrum.

V

V — volts.

Index

Numbers

1D array functions. *See* one-dimensional array operation functions.
1D complex operation functions. *See* one-dimensional complex operation functions.
2D array functions. *See* two-dimensional array operation functions.

A

Abs1D function, 2-1
ACDCEstimator function, 2-3
Add1D function, 2-4
Add2D function, 2-5
Advanced Analysis Library functions
- class and subclass descriptions, 1-8 to 1-10
- error codes
  - alphabetical list, A-1 to A-4
  - numeric list, A-4 to A-7
- function panels
  - array operation functions, 1-2 to 1-3
  - complex operation functions, 1-3 to 1-4
  - curve fitting functions, 1-7
  - function tree (table), 1-1 to 1-8
  - hints for using, 1-10
  - interpolation functions, 1-7
  - measurement functions, 1-6
  - signal generation functions, 1-2
  - signal processing functions, 1-4 to 1-6
  - statistics functions, 1-6
  - vector and matrix algebra functions, 1-7 to 1-8
- function reference
  - Abs1D, 2-1
  - ACDCEstimator, 2-3
  - Add1D, 2-4
  - Add2D, 2-5
  - AllocIIRFilterPtr, 2-6 to 2-7
  - AmpPhaseSpectrum, 2-8 to 2-9
  - ANOVA1Way, 2-10 to 2-15
  - ANOVA2Way, 2-16 to 2-26
  - ANOVA3Way, 2-27 to 2-41
  - ArbitraryWave, 2-42 to 2-43
  - AutoPowerSpectrum, 2-44 to 2-45
  - BackSub, 2-46 to 2-47
  - Bessel_CascadeCoef, 2-48 to 2-49
  - Bessel_Coef, 2-50 to 2-51
  - BkmanWin, 2-52
  - BlkHarrisWin, 2-53
  - Bw_BPF, 2-54 to 2-55
  - Bw_BSF, 2-56 to 2-57
  - Bw_CascadeCoef, 2-58 to 2-59
  - Bw_Coef, 2-60 to 2-61
  - Bw_HPF, 2-62 to 2-63
  - Bw_LPF, 2-64 to 2-65
  - CascadeToDirectCoef, 2-66 to 2-67
  - Ch_BPF, 2-68 to 2-69
  - Ch_BSF, 2-70 to 2-71
  - Ch_CascadeCoef, 2-72 to 2-73
  - Ch_Coef, 2-74 to 2-75
  - Ch_HPF, 2-76 to 2-77
  - Ch_LPF, 2-78 to 2-79
  - CheckPosDef, 2-80
  - Chirp, 2-81
  - Cholesky, 2-82 to 2-83
  - Clear1D, 2-84
  - Clip, 2-85
  - ConditionNumber, 2-86 to 2-87
  - Contingency_Table, 2-88 to 2-91
  - Convolve, 2-92 to 2-93
  - Copy1D, 2-94
  - Correlate, 2-95 to 2-96
  - CosTaperedWin, 2-97
  - CrossPowerSpectrum, 2-98 to 2-99

CrossSpectrum, 2-100 to 2-101
CxAdd, 2-102
CxAdd1D, 2-103
CxCheckPosDef, 2-104
CxCholesky, 2-105 to 2-106
CxConditionNumber, 2-107 to 2-108
CxDeterminant, 2-109 to 2-110
CxDiv, 2-111
CxDiv1D, 2-112
CxDotProduct, 2-113
CxEigenValueVector, 2-114 to 2-115
CxExp, 2-116
CxGenInvMatrix, 2-117 to 2-118
CxGenLinEqs, 2-119 to 2-120
CxLinEv1D, 2-121 to 2-122
CxLn, 2-123
CxLog, 2-124
CxLU, 2-125 to 2-126
CxMatrixMul, 2-127 to 2-128
CxMatrixNorm, 2-129 to 2-130
CxMatrixRank, 2-131 to 2-132
CxMul, 2-133
CxMul1D, 2-134
CxOuterProduct, 2-135 to 2-136
CxPolyRoots, 2-137 to 2-138
CxPow, 2-139
CxPseudoInverse, 2-140 to 2-141
CxQR, 2-142 to 2-143
CxRecip, 2-144
CxSpecialMatrix, 2-145 to 2-147
CxSqrt, 2-148
CxSub, 2-149
CxSub1D, 2-150
CxSVD, 2-151 to 2-152
CxSVDS, 2-153
CxTrace, 2-154
CxTranspose, 2-155
Decimate, 2-156
Deconvolve, 2-157
Determinant, 2-158
Difference, 2-159 to 2-160
Div1D, 2-161
Div2D, 2-162
DotProduct, 2-163
Elp_BPF, 2-164 to 2-165
Elp_BSF, 2-166 to 2-167
Elp_CascadeCoef, 2-168 to 2-169
Elp_Coef, 2-170 to 2-171
Elp_HPF, 2-172 to 2-173
Elp_LPF, 2-174 to 2-175
Equi_Ripple, 2-176 to 2-179
EquiRpl_BPF, 2-180 to 2-181
EquiRpl_BSF, 2-182 to 2-183
EquiRpl_HPF, 2-184 to 2-185
EquiRpl_LPF, 2-186 to 2-187
ExBkmanWin, 2-188
ExpFit, 2-189 to 2-190
ExpWin, 2-191
F_Dist, 2-192
FFT, 2-193 to 2-194
FHT, 2-195 to 2-196
FIR_Coef, 2-197 to 2-198
FlatTopWin, 2-199
ForceWin, 2-200
ForwSub, 2-201 to 2-202
FreeAnalysisMem, 2-203
FreeIIRFilterPtr, 2-204
GaussNoise, 2-205
GenCosWin, 2-206
GenDeterminant, 2-207 to 2-208
GenEigenValueVector,
2-209 to 2-210
GenInvMatrix, 2-211 to 2-212
GenLinEqs, 2-213 to 2-214
GenLSFit, 2-215 to 2-223
GenLSFitCoef, 2-224 to 2-226
GetAnalysisErrorString, 2-227
HamWin, 2-228
HanWin, 2-229
HarmonicAnalyzer, 2-230 to 2-231
Histogram, 2-232 to 2-233

IIRCascadeFiltering, 2-234 to 2-235
IIRFiltering, 2-236 to 2-237
Impulse, 2-238
ImpulseResponse, 2-239 to 2-240
Integrate, 2-241 to 2-242
InvCh_BPF, 2-243 to 2-244
InvCh_BSF, 2-245 to 2-246
InvCh_CascadeCoef, 2-247 to 2-248
InvCh_Coef, 2-249 to 2-250
InvCh_HFP, 2-251 to 2-252
InvCh_LPF, 2-253 to 2-254
InvF_Dist, 2-255 to 2-256
InvFFT, 2-257 to 2-258
InvFHT, 2-259 to 2-260
InvMatrix, 2-261
InvN_Dist, 2-262
InvT_Dist, 2-263
InvXX_Dist, 2-264
Ksr_BPF, 2-265 to 2-266
Ksr_BSF, 2-267 to 2-268
Ksr_HPF, 2-269 to 2-270
Ksr_LPF, 2-271 to 2-272
KsrWin, 2-273 to 2-274
LinEqs, 2-275
LinEv1D, 2-276
LinEv2D, 2-277
LinFit, 2-278 to 2-279
LU, 2-280 to 2-281
MatrixMul, 2-282 to 2-283
MatrixNorm, 2-284 to 2-285
MatrixRank, 2-286 to 2-287
MaxMin1D, 2-288
MaxMin2D, 2-289 to 2-290
Mean, 2-291
Median, 2-292
Mode, 2-293
Moment, 2-294 to 2-295
Mul1D, 2-296
Mul2D, 2-297
N-Dist, 2-298
Neg1D, 2-299
NetworkFunctions, 2-300 to 2-302
NonLinearFit, 2-303 to 2-304
NonLinearFitWith MaxIters, 2-305
Normal1D, 2-307 to 2-308
Normal2D, 2-309 to 2-310
NumericIntegration, 2-311 to 2-313
OuterProduct, 2-314
PeakDetector, 2-315 to 2-317
PolyEv1D, 2-318 to 2-319
PolyEv2D, 2-320 to 2-321
PolyFit, 2-322 to 2-323
PolyInterp, 2-324 to 2-325
PowerFrequencyEstimate, 2-326 to 2-328
Prod1D, 2-329
PseudoInverse, 2-330 to 2-331
Pulse, 2-332 to 2-333
PulseParam, 2-234 to 2-336
QR, 2-337 to 2-338
QScale1D, 2-339
QScale2D, 2-340
Ramp, 2-341 to 2-342
RatInterp, 2-343 to 2-344
ReFFT, 2-345
ReInvFFT, 2-346
ResetIIRFilter, 2-347 to 2-348
Reverse, 2-349
RMS, 2-350
SawtoothWave, 2-351 to 2-352
Scale1D, 2-353 to 2-354
Scale2D, 2-355 to 2-356
ScaledWindow, 2-357 to 2-358
Set1D, 2-359
Shift, 2-360 to 2-361
Sinc, 2-362
SinePattern, 2-363 to 2-364
SineWave, 2-365 to 2-366
Sort, 2-367
SpecialMatrix, 2-368 to 2-370
Spectrum, 2-371

SpectrumUnitConversion, 2-372 to 2-375
SpInterp, 2-376 to 2-377
Spline, 2-378 to 2-379
SquareWave, 2-380 to 2-381
StdDev, 2-382
Sub1D, 2-383
Sub2D, 2-384
Subset1D, 2-385
Sum1D, 2-386
Sum2D, 2-387
SVD, 2-388 to 2-389
SVDS, 2-390
SymEigenValueVector, 2-391 to 2-392
T_Dist, 2-393
ToPolar, 2-394
ToPolar1D, 2-395
ToRect, 2-396
ToRect1D, 2-397
Trace, 2-398
TransferFunction, 2-399
Transpose, 2-401
Triangle, 2-402
TriangleWave, 2-403 to 2-404
TriWin, 2-405
Uniform, 2-406
UnWrap1D, 2-407
Variance, 2-408
WhiteNoise, 2-409
Wind_BPF, 2-410 to 2-411
Wind_BSF, 2-412 to 2-413
Wind_HPF, 2-414 to 2-415
Wind_LPF, 2-416 to 2-417
XX_Dist, 2-418
AllocIIRFilterPtr function, 2-6 to 2-7
AmpPhaseSpectrum function, 2-8 to 2-9
analysis of variance functions
ANOVA1Way, 2-10 to 2-15
assumptions, 2-12
examples, 2-14 to 2-15
factors and levels, 2-11
formulas, 2-13
general method of using, 2-11
hypothesis, 2-12
parameters, 2-10
purpose, 2-10
return value, 2-11
statistical method, 2-12
ANOVA2Way, 2-16 to 2-26
assumptions, 2-20
examples, 2-24 to 2-26
factors, levels, and cells, 2-18
formulas, 2-21 to 2-23
general method of using, 2-19
hypotheses, 2-20 to 2-21
parameters, 2-16 to 2-17
purpose, 2-16
random and fixed effects, 2-18 to 2-19
return value, 2-17
statistical model, 2-19 to 2-20
ANOVA3Way, 2-27 to 2-41
assumptions, 2-31
examples, 2-37 to 2-41
factors, levels, and cells, 2-29 to 2-30
formulas, 2-33 to 2-37
general method of using, 2-30
hypotheses, 2-32 to 2-33
parameters, 2-27 to 2-29
purpose, 2-27
random and fixed effects, 2-30
return value, 2-29
statistical model, 2-31
definition, 1-9
function tree, 1-6
ArbitraryWave function, 2-42 to 2-43
array analysis, performing in place, 1-10
array operation functions
Abs1D, 2-1
Add1D, 2-4
Add2D, 2-5

Clear1D, 2-84
Copy1D, 2-94
definition, 1-8
Div1D, 2-161
Div2D, 2-162
function tree, 1-2 to 1-3
LinEv1D, 2-276
LinEv2D, 2-277
MaxMin1D, 2-288
MaxMin2D, 2-289 to 2-290
Mul1D, 2-296
Mul2D, 2-297
Neg1D, 2-299
PolyEv1D, 2-318 to 2-319
PolyEv2D, 2-320 to 2-321
Prod1D, 2-329
QScale1D, 2-339
QScale2D, 2-340
Scale1D, 2-353 to 2-354
Scale2D, 2-355 to 2-356
Set1D, 2-359
Sub1D, 2-383
Sub2D, 2-384
Subset1D, 2-385
Sum1D, 2-386
Sum2D, 2-387
UnWrap1D, 2-407
AutoPowerSpectrum function, 2-44 to 2-45

B

BackSub function, 2-46 to 2-47
basic statistics functions
definition, 1-9
function tree, 1-6
Histogram, 2-232 to 2-233
Mean, 2-291
Median, 2-292
Mode, 2-293
Moment, 2-294 to 2-295
RMS, 2-350
StdDev, 2-382
Variance, 2-408
Bessel_CascadeCoef function, 2-48 to 2-49
Bessel_Coef function, 2-50 to 2-51
BkmanWin function, 2-52
BlkHarrisWin function, 2-53
Bw_BPF function, 2-54 to 2-55
Bw_BSF function, 2-56 to 2-57
Bw_CascadeCoef function, 2-58 to 2-59
Bw_Coef function, 2-60 to 2-61
Bw_HPF function, 2-62 to 2-63
Bw_LPF function, 2-64 to 2-65

C

CascadeToDirectCoef function, 2-66 to 2-67
Ch_BPF function, 2-68 to 2-69
Ch_BSF function, 2-70 to 2-71
Ch_CascadeCoef function, 2-72 to 2-73
Ch_Coef function, 2-74 to 2-75
Ch_HPF function, 2-76 to 2-77
Ch_LPF function, 2-78 to 2-79
CheckPosDef function, 2-80
Chirp function, 2-81
chi-square tests, 2-89 to 2-90
Cholesky function, 2-82 to 2-83
Clear1D function, 2-84
Clip function, 2-85
complex matrix functions. *See* vector and matrix algebra functions.
complex operation functions
CxAdd, 2-102
CxAdd1D, 2-103
CxDiv, 2-111
CxDiv1D, 2-112
CxExp, 2-116
CxLinEv1D, 2-121 to 2-122
CxLn, 2-123
CxLog, 2-124
CxMul, 2-133
CxMul1D, 2-134

CxPow, 2-139
CxRecip, 2-144
CxSqrt, 2-148
CxSub, 2-149
CxSub1D, 2-150
definition, 1-9
function tree, 1-3 to 1-4
ToPolar, 2-394
ToPolar1D, 2-395
ToRect, 2-396
ToRect1D, 2-397
ConditionNumber function, 2-86 to 2-87
Contingency_Table function, 2-88 to 2-91
chi-square test of homogeneity, 2-89
chi-square test of independence, 2-90
example, 2-91
formulas, 2-91
hypothesis testing, 2-90
parameters, 2-88
purpose, 2-88
return value, 2-89
Convolve function, 2-92 to 2-93
Copy1D function, 2-94
Correlate function, 2-95 to 2-96
CosTaperedWin function, 2-97
CrossPowerSpectrum function, 2-98 to 2-99
CrossSpectrum function, 2-100 to 2-101
curve fitting, 1-21
curve fitting functions
definition, 1-10
ExpFit, 2-189 to 2-190
function tree, 1-7
GenLSFit, 2-215 to 2-223
GenLSFitCoef, 2-224 to 2-226
LinFit, 2-278 to 2-279
NonLinearFit, 2-303 to 2-304
PolyFit, 2-322 to 2-323
customer communication, *xvi*, B-1 to B-2
CxAdd function, 2-102
CxAdd1D function, 2-103
CxCheckPosDef function, 2-104
CxCholesky function, 2-105 to 2-106
CxConditionNumber function, 2-107 to 2-108
CxDeterminant function, 2-109 to 2-110
CxDiv function, 2-111
CxDiv1D function, 2-112
CxDotProduct function, 2-113
CxEigenValueVector function, 2-114 to 2-115
CxExp function, 2-116
CxGenInvMatrix function, 2-117 to 2-118
CxGenLinEqs function, 2-119 to 2-120
CxLinEv1D function, 2-121 to 2-122
CxLn function, 2-123
CxLog function, 2-124
CxLU function, 2-125 to 2-126
CxMatrixMul function, 2-127 to 2-128
CxMatrixNorm function, 2-129 to 2-130
CxMatrixRank function, 2-131 to 2-132
CxMul function, 2-133
CxMul1D function, 2-134
CxOuterProduct function, 2-135 to 2-136
CxPolyRoots function, 2-137 to 2-138
CxPow function, 2-139
CxPseudoInverse function, 2-140 to 2-141
CxQR function, 2-142 to 2-143
CxRecip function, 2-144
CxSpecialMatrix function, 2-145 to 2-147
CxSqrt function, 2-148
CxSub function, 2-149
CxSub1D function, 2-150
CxSVD function, 2-151 to 2-152
CxSVDS function, 2-153
CxTrace function, 2-154
CxTranspose function, 2-155

D

Decimate function, 2-156
Deconvolve function, 2-157
Determinant function, 2-158
Difference function, 2-159 to 2-160

digital filters. *See* FIR filters; IIR filters.
Discrete Fourier Transform (DFT), 1-11
Div1D function, 2-161
Div2D function, 2-162
documentation
    conventions used in manual, *xiii-xiv*
    organization of manual, *xiii*
    related documentation, *xiv-xvi*
DotProduct function, 2-163

E

electronic support services, B-1 to B-2
Elp_BPF function, 2-164 to 2-165
Elp_BSF function, 2-166 to 2-167
Elp_CascadeCoef function, 2-168 to 2-169
Elp_Coef function, 2-170 to 2-171
Elp_HPF function, 2-172 to 2-173
Elp_LPF function, 2-174 to 2-175
e-mail support, B-2
Equi_Ripple function
    description, 2-176 to 2-177
    designing FIR filters, 1-16
    examples, 2-177 to 2-179
    problems with convergence (caution), 1-17
EquiRpl_BPF function, 2-180 to 2-181
EquiRpl_BSF function, 2-182 to 2-183
EquiRpl_HPF function, 2-184 to 2-185
EquiRpl_LPF function, 2-186 to 2-187
error codes
    alphabetical list, A-1 to A-4
    numeric list, A-4 to A-7
errors
    converting error number with GetAnalysisErrorString function, 2-227
    reporting analysis errors, 1-11
ExBkmanWin function, 2-188
ExpFit function, 2-189 to 2-190
ExpWin function, 2-191

F

Fast Fourier Transform (FFT), 1-11 to 1-12. *See also* frequency domain functions.
fax and telephone support numbers, B-2
Fax-on-Demand support, B-2
F_Dist function, 2-192
FFT function, 2-193 to 2-194
FHT function, 2-195 to 2-196
finite impulse response functions. *See* FIR digital filter functions; FIR filters.
FIR digital filter functions
    definition, 1-9
    FIR_Coef, 2-197 to 2-198
    function tree, 1-5
    Ksr_BPF, 2-265 to 2-266
    Ksr_BSF, 2-267 to 2-268
    Ksr_HPF, 2-269 to 2-270
    Ksr_LPF, 2-271 to 2-272
    Wind_BPF, 2-410 to 2-411
    Wind_BSF, 2-412 to 2-413
    Wind_HPF, 2-414 to 2-415
    Wind_LPF, 2-416 to 2-417
FIR filters
    compared with IIR filters, 1-15
    definition, 1-15
    designing, 1-16 to 1-17
FIR_Coef function, 2-197 to 2-198
FlatTopWin function, 2-199
ForceWin function, 2-200
ForwSub function, 2-201 to 2-202
Fourier Transform integral, 1-11
FreeAnalysisMem function, 2-203
FreeIIRFilterPtr function, 2-204
frequency domain functions
    conventions and restrictions related to Fast Fourier Transform, 1-12
    CrossSpectrum, 2-100 to 2-101
    definition, 1-9
    FFT, 2-193 to 2-194
    FHT, 2-195 to 2-196

function tree, 1-4
InvFFT, 2-257 to 2-258
InvFHT, 2-259 to 2-260
notation for describing Fast Fourier Transform operations, 1-12
ReFFT, 2-345
ReInvFFT, 2-346
Spectrum, 2-371
FTP support, B-1
function panels. *See under* Advanced Analysis Library functions.

G

GaussNoise function, 2-205
GenCosWin function, 2-206
GenDeterminant function, 2-207 to 2-208
GenEigenValueVector function, 2-209 to 2-210
generated code stored in Interactive window, 1-10
GenInvMatrix function, 2-211 to 2-212
GenLinEqs function, 2-213 to 2-214
GenLSFit function, 2-215 to 2-223
example, 2-221 to 2-223
parameters, 2-215 to 2-216
purpose, 2-215
return value, 2-216
using the function, 2-217 to 2-220
GenLSFitCoef function, 2-224 to 2-226
GetAnalysisErrorString function, 2-227

H

HamWin function, 2-228
HanWin function, 2-229
HarmonicAnalyzer function, 2-230 to 2-231
Histogram function, 2-232 to 2-233

I

IEW. *See* Interactive Execution window.
IIR digital filter functions
AllocIIRFilterPtr, 2-6 to 2-7
Bessel_CascadeCoef, 2-48 to 2-49
Bessel_Coef, 2-50 to 2-51
Bw_BPF, 2-54 to 2-55
Bw_BSF, 2-56 to 2-57
Bw_CascadeCoef, 2-58 to 2-59
Bw_Coef, 2-60 to 2-61
Bw_HPF, 2-62 to 2-63
Bw_LPF, 2-64 to 2-65
CascadeToDirectCoef, 2-66 to 2-67
Ch_BPF, 2-68 to 2-69
Ch_BSF, 2-70 to 2-71
Ch_CascadeCoef, 2-72 to 2-73
Ch_Coef, 2-74 to 2-75
Ch_HPF, 2-76 to 2-77
Ch_LPF, 2-78 to 2-79
definition, 1-9
Elp_BPF, 2-164 to 2-165
Elp_BSF, 2-166 to 2-167
Elp_CascadeCoef, 2-168 to 2-169
Elp_Coef, 2-170 to 2-171
Elp_HPF, 2-172 to 2-173
Elp_LPF, 2-174 to 2-175
Equi_Ripple, 2-176 to 2-179
EquiRpl_BPF, 2-180 to 2-181
EquiRpl_BSF, 2-182 to 2-183
EquiRpl_HPF, 2-184 to 2-185
EquiRpl_LPF, 2-186 to 2-187
FreeIIRFilterPtr, 2-204
function tree, 1-4 to 1-5
IIRCascadeFiltering, 2-234 to 2-235
IIRFiltering, 2-236 to 2-237
InvCh_BPF, 2-243 to 2-244
InvCh_BSF, 2-245 to 2-246
InvCh_CascadeCoef, 2-247 to 2-248
InvCh_Coef, 2-249 to 2-250
InvCh_HFP, 2-251 to 2-252

InvCh_LPF, 2-253 to 2-254
ResetIIRFilter, 2-347 to 2-348
IIR filters, 1-17 to 1-19
cascaded filter stages, 1-18
compared with FIR filters, 1-15
direct form, 1-17
fourth order, 1-18 to 1-19
mathematical form, 1-17
second order, 1-18
types, 1-19
IIRCascadeFiltering function, 2-234 to 2-235
IIRFiltering function, 2-236 to 2-237
Impulse function, 2-238
ImpulseResponse function, 2-239 to 2-240
infinite impulse response functions. *See* IIR digital filter functions; IIR filters.
Integrate function, 2-241 to 2-242
Interactive Execution window, 1-10
interpolation functions
definition, 1-10
function tree, 1-7
PolyInterp, 2-324 to 2-325
RatInterp, 2-343 to 2-344
SpInterp, 2-376 to 2-377
Spline, 2-378 to 2-379
InvCh_BPF function, 2-243 to 2-244
InvCh_BSF function, 2-245 to 2-246
InvCh_CascadeCoef function, 2-247 to 2-248
InvCh_Coef function, 2-249 to 2-250
InvCh_HFP function, 2-251 to 2-252
InvCh_LPF function, 2-253 to 2-254
InvF_Dist function, 2-255 to 2-256
InvFFT function, 2-257 to 2-258
InvFHT function, 2-259 to 2-260
InvMatrix function, 2-261
InvN_Dist function, 2-262
InvT_Dist function, 2-263
InvXX_Dist function, 2-264

K

Ksr_BPF function, 2-265 to 2-266
Ksr_BSF function, 2-267 to 2-268
Ksr_HPF function, 2-269 to 2-270
Ksr_LPF function, 2-271 to 2-272
KsrWin function, 2-273 to 2-274

L

LinEqs function, 2-275
LinEv1D function, 2-276
LinEv2D function, 2-277
LinFit function, 2-278 to 2-279
LU function, 2-280 to 2-281

M

manual. *See* documentation.
matrix algebra functions. *See* vector and matrix algebra functions.
MatrixMul function, 2-282 to 2-283
MatrixNorm function, 2-284 to 2-285
MatrixRank function, 2-286 to 2-287
MaxMin1D function, 2-288
MaxMin2D function, 2-289 to 2-290
Mean function, 2-291
measurement functions
ACDCEstimator, 2-3
AmpPhaseSpectrum, 2-8 to 2-9
AutoPowerSpectrum, 2-44 to 2-45
characteristics, 1-20
CrossPowerSpectrum, 2-98 to 2-99
definition, 1-9
function tree, 1-6
HarmonicAnalyzer, 2-230 to 2-231
ImpulseResponse, 2-239 to 2-240
NetworkFunctions, 2-300 to 2-302
PowerFrequencyEstimate, 2-326 to 2-328
purpose and use, 1-19 to 1-20
ScaledWindow, 2-357 to 2-358

SpectrumUnitConversion, 2-372 to 2-375
TransferFunction, 2-399
Median function, 2-292
Mode function, 2-293
Moment function, 2-294 to 2-295
Mul1D function, 2-296
Mul2D function, 2-297

N

N-Dist function, 2-298
Neg1D function, 2-299
NetworkFunctions function, 2-300 to 2-302
NonLinearFit function, 2-303 to 2-304
NonLinearFitWith MaxIters function, 2-305
nonparametric statistics function
Contingency_Table, 2-88 to 2-91
definition, 1-9
function tree, 1-6
Normal1D function, 2-307 to 2-308
Normal2D function, 2-309 to 2-310
NumericIntegration function, 2-311 to 2-313

O

one-dimensional array operation functions
Abs1D, 2-1
Add1D, 2-4
definition, 1-8
Div1D, 2-161
function tree, 1-2 to 1-3
LinEv1D, 2-276
MaxMin1D, 2-288
Mul1D, 2-296
Neg1D, 2-299
PolyEv1D, 2-318 to 2-319
Prod1D, 2-329
QScale1D, 2-339
Scale1D, 2-353 to 2-354
Sub1D, 2-383
Subset1D, 2-385
Sum1D, 2-386
Sum2D, 2-387
one-dimensional complex operation functions
CxAdd1D, 2-103
CxDiv1D, 2-112
CxLinEv1D, 2-121 to 2-122
CxMul1D, 2-134
CxSub1D, 2-150
definition, 1-9
function tree, 1-4
ToPolar1D, 2-395
ToRect1D, 2-397
OuterProduct function, 2-314

P

PeakDetector function, 2-315 to 2-317
performance considerations, analysis functions, 1-10
PolyEv1D function, 2-318 to 2-319
PolyEv2D function, 2-320 to 2-321
PolyFit function, 2-322 to 2-323
PolyInterp function, 2-324 to 2-325
PowerFrequencyEstimate function, 2-326 to 2-328
probability distribution functions
definition, 1-9
F_Dist, 2-192
function tree, 1-6
InvF_Dist, 2-255 to 2-256
InvN_Dist, 2-262
InvT_Dist, 2-263
InvXX_Dist, 2-264
N-Dist, 2-298
T_Dist, 2-393
XX_Dist, 2-418
Prod1D function, 2-329
PseudoInverse function, 2-330 to 2-331
Pulse function, 2-332 to 2-333
PulseParam function, 2-234 to 2-336

Q

QR function, 2-337 to 2-338
QScale1D function, 2-339
QScale2D function, 2-340

R

Radix-4 and Radix-8 algorithms, 1-12
Ramp function, 2-341 to 2-342
RatInterp function, 2-343 to 2-344
real matrix functions. *See* vector and matrix algebra functions.
ReFFT function, 2-345
ReInvFFT function, 2-346
ResetIIRFilter function, 2-347 to 2-348
Reverse function, 2-349
RMS function, 2-350

S

SawtoothWave function, 2-351 to 2-352
Scale1D function, 2-353 to 2-354
Scale2D function, 2-355 to 2-356
ScaledWindow function, 2-357 to 2-358
Set1D function, 2-359
Shift function, 2-360 to 2-361
signal generation functions
    ArbitraryWave, 2-42 to 2-43
    Chirp, 2-81
    definition, 1-9
    function tree, 1-2
    GaussNoise, 2-205
    Impulse, 2-238
    Pulse, 2-332 to 2-333
    Ramp, 2-341 to 2-342
    SawtoothWave, 2-351 to 2-352
    Sinc, 2-362
    SinePattern, 2-363 to 2-364
    SineWave, 2-365 to 2-366
    SquareWave, 2-380 to 2-381
    Triangle, 2-402
    TriangleWave, 2-403 to 2-404
    Uniform, 2-406
    WhiteNoise, 2-409
signal processing functions. *See* FIR digital filter functions; frequency domain functions; IIR digital filter functions; time domain functions; windows functions.
Sinc function, 2-362
SinePattern function, 2-363 to 2-364
SineWave function, 2-365 to 2-366
Sort function, 2-367
source file code stored in Interactive window, 1-10
SpecialMatrix function, 2-368 to 2-370
Spectrum function, 2-371
SpectrumUnitConversion function, 2-372 to 2-375
SpInterp function, 2-376 to 2-377
Spline function, 2-378 to 2-379
Split-Radix algorithm, 1-12
SquareWave function, 2-380 to 2-381
statistics functions
    ANOVA1Way, 2-10 to 2-15
    ANOVA2Way, 2-16 to 2-26
    ANOVA3Way, 2-27 to 2-41
    Contingency_Table, 2-88 to 2-91
    definition, 1-9
    F_Dist, 2-192
    function tree, 1-6
    GenLSFit, 2-215 to 2-223
    Histogram, 2-232 to 2-233
    InvF_Dist, 2-255 to 2-256
    InvN_Dist, 2-262
    InvT_Dist, 2-263
    InvXX_Dist, 2-264
    Mean, 2-291
    Median, 2-292
    Mode, 2-293
    Moment, 2-294 to 2-295
    N-Dist, 2-298

RMS, 2-350
Sort, 2-367
StdDev, 2-382
T_Dist, 2-393
Variance, 2-408
XX_Dist, 2-418
StdDev function, 2-382
Sub1D function, 2-383
Sub2D function, 2-384
Subset1D function, 2-385
Sum1D function, 2-386
Sum2D function, 2-387
SVD function, 2-388 to 2-389
SVDS function, 2-390
SymEigenValueVector function, 2-391 to 2-392

T

T_Dist function, 2-393
technical support, B-1 to B-2
telephone and fax support numbers, B-2
time domain functions
Clip, 2-85
Convolve, 2-92 to 2-93
Correlate, 2-95 to 2-96
Decimate, 2-156
Deconvolve, 2-157
definition, 1-9
Difference, 2-159 to 2-160
function tree, 1-4
Integrate, 2-241 to 2-242
PulseParam, 2-234 to 2-336
Reverse, 2-349
Shift, 2-360 to 2-361
ToPolar function, 2-394
ToPolar1D function, 2-395
ToRect function, 2-396
ToRect1D function, 2-397
Trace function, 2-398
TransferFunction function, 2-399
Transpose function, 2-401
Triangle function, 2-402
TriangleWave function, 2-403 to 2-404
TriWin function, 2-405
two-dimensional array operation functions
Add2D, 2-5
definition, 1-8
Div2D, 2-162
function tree, 1-3
LinEv2D, 2-277
MaxMin2D, 2-289 to 2-290
Mul2D, 2-297
PolyEv2D, 2-320 to 2-321
QScale2D, 2-340
Scale2D, 2-355 to 2-356
Sub2D, 2-384

U

Uniform function, 2-406
UnWrap1D function, 2-407

V

Variance function, 2-408
vector and matrix algebra functions
BackSub, 2-46 to 2-47
CheckPosDef, 2-80
Cholesky, 2-82 to 2-83
ConditionNumber, 2-86 to 2-87
CxCheckPosDef, 2-104
CxCholesky, 2-105 to 2-106
CxConditionNumber, 2-107 to 2-108
CxDeterminant, 2-109 to 2-110
CxDotProduct, 2-113
CxEigenValueVector, 2-114 to 2-115
CxGenInvMatrix, 2-117 to 2-118
CxGenLinEqs, 2-119 to 2-120
CxLU, 2-125 to 2-126
CxMatrixMul, 2-127 to 2-128
CxMatrixNorm, 2-129 to 2-130

CxMatrixRank, 2-131 to 2-132
CxOuterProduct, 2-135 to 2-136
CxPolyRoots, 2-137 to 2-138
CxPseudoInverse, 2-140 to 2-141
CxQR, 2-142 to 2-143
CxSpecialMatrix, 2-145 to 2-147
CxSVD, 2-151 to 2-152
CxSVDS, 2-153
CxTrace, 2-154
CxTranspose, 2-155
definition, 1-10
Determinant, 2-158
DotProduct, 2-163
ForwSub, 2-201 to 2-202
function tree, 1-7 to 1-8
GenDeterminant, 2-207 to 2-208
GenEigenValueVector, 2-209 to 2-210
GenInvMatrix, 2-211 to 2-212
GenLinEqs, 2-213 to 2-214
InvMatrix, 2-261
LinEqs, 2-275
LU, 2-280 to 2-281
MatrixMul, 2-282 to 2-283
MatrixNorm, 2-284 to 2-285
MatrixRank, 2-286 to 2-287
NonLinearFitWith MaxIters, 2-305
Normal1D, 2-307 to 2-308
Normal2D, 2-309 to 2-310
OuterProduct, 2-314
PseudoInverse, 2-330 to 2-331
purpose and use, 1-21
QR, 2-337 to 2-338
SpecialMatrix, 2-368 to 2-370
SVD, 2-388 to 2-389
SVDS, 2-390
SymEigenValueVector, 2-391 to 2-392
Trace, 2-398
Transpose, 2-401

W

WhiteNoise function, 2-409
Wind_BPF function, 2-410 to 2-411
Wind_BSF function, 2-412 to 2-413
Wind_HPF function, 2-414 to 2-415
Wind_LPF function, 2-416 to 2-417
windowing, 1-13 to 1-15
windows functions
    BkmanWin, 2-52
    BlkHarrisWin, 2-53
    CosTaperedWin, 2-97
    definition, 1-9
    ExBkmanWin, 2-188
    ExpWin, 2-191
    FlatTopWin, 2-199
    ForceWin, 2-200
    function tree, 1-5 to 1-6
    GenCosWin, 2-206
    HamWin, 2-228
    HanWin, 2-229
    KsrWin, 2-273 to 2-274
    TriWin, 2-405
windType parameter, 1-16

X

XX_Dist function, 2-418