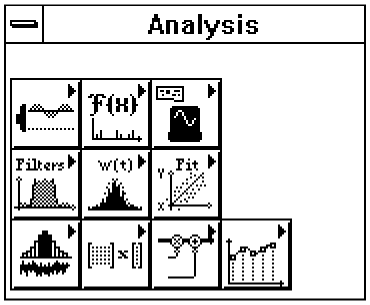
Analysis VIs

The following illustration shows the options that are available on the **Analysis** palette. For general information about analysis VIs, see Analysis VIs Overview. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Additional Numerical Method VIs
Array Operations VIs
Curve Fitting VIs
Digital Filter VIs
Digital Signal ProcessingVIs
Linear Algebra VIs
Measurement VIs
Probability and StatisticS VIs
Signal Generation VIs
Window VIs

Analysis VIs Overview

This topic contains general information about the LabVIEW Analysis VIs. For descriptions of specific Analysis functions, see Analysis VI Descriptions. See the topic, Analysis Examples, for a discussion of the theoretical and practical aspects of using analysis VIs in actual applications.

Both the LabVIEW base system (Windows Only) and the full development system contain the following VI groups:

Probability and Statistics VI Descriptions contains VIs that perform descriptive statistics functions, such as identifying the mean or the standard deviation of a set of data as well as inferential statistics functions for probability and analysis of variance (ANOVA).

Linear Algebra Overview contains general information about VIs that perform algebraic functions for real and complex vectors and matrices.

Array Operation VI Descriptions contains VIs that perform common, one- and two-dimensional numerical array operations, such as linear evaluation and scaling.

The full development system also includes:

Signal Generation VI Descriptions contains VIs that generate digital patterns and waveforms.

Digital Signal Processing VI Descriptions contains VIs that perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms such as the Hartley and Hilbert transforms.

Filters Overview contains general information about VIs that perform IIR, FIR, and nonlinear digital filtering functions.

Window VIs Overview contains general information about VIs that perform data windowing.

Curve Fitting Overview contains general information about VIs that perform curve fitting functions and interpolations.

Measurement Overview contains general information about VIs that perform measurement-oriented functions such as single-sided spectrums, scaled windowing, and peak power and frequency estimation.

Additional Numerical Method VIs Descriptions contains VIs that perform root-finding, numerical integration, and peak detection.

For additional information, see the following topics:

Getting Information about a VI
Analysis Error Reporting
Notation and Naming Conventions
Sampling Signals

Probability and Statistics Subpalette

Probability and Statistics contains VIs that perform descriptive statistics functions, such as identifying the mean or the standard deviation of a set of data, as well as inferential statistics functions for probability and analysis of variance (ANOVA).

Linear Algebra Subpalette

Linear Algebra contains VIs that perform algebraic functions for real and complex vectors and matrices.

Array Operations Subpalette

Array Operations contains VIs that perform common, one- and two-dimensional numerical array operations, such as linear evaluation and scaling.

Signal Generation Subpalette

Signal Generation contains VIs that generate digital patterns and waveforms.

Digital Signal Processing Subpalette

Digital Signal Processing contains VIs that perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms such as the Hartley and Hilbert transforms.

Filters Subpalette

Filters contains VIs that perform IIR, FIR, and nonlinear digital filtering functions.

Windows Subpalette

Windows contains VIs that perform data windowing.

Curve Fitting Subpalette

Curve Fitting contains VIs that perform curve fitting functions and interpolations.

Measurement Subpalette

Measurement contains VIs that perform measurement-oriented functions such as single-sided spectrums, scaled windowing, and peak power and frequency estimation.

Additional Numerical Methods Subpalette

Additional Numerical Methods contains VIs that perform root-finding, numerical integration, and peak detection.

Analysis Error Codes

Code	Name	Description
0	NoErr	No error; the call was successful.
-20001	OutOfMemErr	There is not enough memory left to perform the specified routine.
-20002	EqSamplesErr	The input sequences must be the same size.
-20003	SamplesGTZeroErr	The number of samples must be greater than zero.
-20004	SamplesGEZeroErr	The number of samples must be greater than or equal to zero.
-20005	SamplesGEOneErr	The number of samples must be greater than or equal to one.
-20006	SamplesGETwoErr	The number of samples must be greater than or equal to two.
-20007	SamplesGEThreeErr	The number of samples must be greater than or equal to three.
-20008	ArraySizeErr	The input arrays do not contain the correct number of data values for this VI.
-20009	PowerOfTwoErr	The size of the input array must be a power of two: size = 2^m, 0<m<23.
-20010	MaxXformSizeErr	The maximum transform size has been exceeded.
-20011	DutyCycleErr	The duty cycle must meet the condition: $0 \leq$ duty cycle ≤ 100.
-20012	CyclesErr	The number of cycles must be greater than zero and less than or equal to the number of samples.
-20013	WidthLTSamplesErr	The width must meet the condition: 0<width<samples.
-20014	DelayWidthErr	The following condition must be met: 0_(delay+width)<samples.
-20015	DtGEZeroErr	dt must be greater than or equal to zero.
-20016	DtGTZeroErr	dt must be greater than zero.
-20017	IndexLTSamplesErr	The index must meet the condition: 0_index<samples.
-20018	IndexLengthErr	The following condition must be met: $0 \leq$ (index+length)<samples.
-20019	UpperGELowerErr	The upper value must be greater than or equal to the lower value.
-20020	NyquistErr	The cutoff frequency, f_c, must meet the condition: $0 \leq f_c \leq \frac{f_s}{2}$.
-20021	OrderGTZeroErr	The order must be greater than zero.
-20022	DecFactErr	The decimating factor must meet the condition:

		$0 < \text{decimating} \leq \text{samples}$.
-20023	BandSpecErr	The following condition must be met: $0 \leq f_{flow} \leq f_{high} \leq \frac{f_s}{2}$
-20024	RippleGTZeroErr	The ripple amplitude must be greater than zero.
-20025	AttenGTZeroErr	The attenuation must be greater than zero.
-20026	WidthGTZeroErr	The width must be greater than zero.
-20027	FinalGTZeroErr	The final value must be greater than zero.
-20028	AttenGTRippleErr	The attenuation must be greater than the ripple amplitude.
-20029	StepSizeErr	The step-size, μ, must meet the condition: $0 \leq \mu \leq 0.1$.
-20030	LeakErr	The leakage coefficient must meet the condition: $0 \leq \text{leak} \leq \mu$.
-20031	EqRplDesignErr	The filter cannot be designed with the specified input values.
-20032	RankErr	The rank of the filter must meet the condition: $1 \leq (2\ \text{rank} + 1) \leq \text{size}$.
-20033	EvenSizeErr	The number of coefficients must be odd for this filter.
-20034	OddSizeErr	The number of coefficients must be even for this filter.
-20035	StdDevErr	The standard deviation must be greater than zero for normalization.
-20036	MixedSignErr	The elements of the Y Values array must be nonzero and either all positive or all negative.
-20037	SizeGTOrderErr	The number of data points in the Y Values array must be greater than two.
-20038	IntervalsErr	The number of intervals must be greater than zero.
-20039	MatrixMulErr	The number of columns in the first matrix is not equal to the number of rows in the second matrix or vector.
-20040	SquareMatrixErr	The input matrix must be a square matrix.
-20041	SingularMatrixErr	The system of equations cannot be solved because the input matrix is singular.
-20042	LevelsErr	The number of levels is out of range.
-20043	FactorErr	The level of factors is out of range for some data.
-20044	ObservationsErr	Zero observations were made at some level of a factor.
-20045	DataErr	The total number of data points must be equal to the product of the levels for each factor and the observations per cell.
-20046	OverflowErr	There is an overflow in the calculated F-value.

-20047	BalanceErr	The data is unbalanced. All cells must contain the same number of observations.
-20048	ModelErr	The Random Effect model was requested when the Fixed Effect model was required.
-20049	DistinctErr	The x values must be distinct.
-20050	PoleErr	The interpolating function has a pole at the requested value.
-20051	ColumnErr	All values in the first column in the X matrix must be one.
-20052	FreedomErr	The degrees of freedom must be one or more.
-20053	ProbabilityErr	The probability must be between zero and one.
-20054	InvProbErr	The probability must be greater than or equal to zero and less than one.
-20055	CategoryErr	The number of categories or samples must be greater than one.
-20056	TableErr	The contingency table must not contain a negative number.
-20061	InvSelectionErr	One of the input selections is invalid.
-20062	MaxIterErr	The maximum iterations have been exceeded.
-20063	PolyErr	The polynomial coefficents are invalid.
-20064	InitStateErr	This VI has not been initialized correctly.
-20065	ZeroVectorErr	The vector cannot be zero.

References for Analysis VIs

This section lists the reference material used to produce Analysis VIs . These references contain more information on the theories and algorithms implemented in the analysis library.

1. Baher, H. *Analog & Digital Signal Processing*. New York: John Wiley & Sons. 1990.
2. Bates, D.M. and Watts, D.G. *Nonlinear Regression Analysis and its Applications*. New York : John Wiley & Sons. 1988.
3. Bracewell, R.N. "Numerical Transforms." Science. Science-248. 11 May 1990.
4. Burden, R.L. & Faires, J.D. *Numerical Analysis. Third Edition*. Boston: Prindle, Weber & Schmidt. 1985.
5. Chen, C.H. et al. *Signal Processing Handbook*. New York: Marcel Dekker, Inc. 1988.
6. DeGroot, M. Probability and Statistics 2nd ed. Reading, Massachusetts : Addison-Wesley Publishing Co. 1986.
7. Dowdy, S. and Wearden, S. *Statistics for Research* 2nd ed. New York : John Wiley & Sons. 1991.
8. Dudewicz, E.J. and Mishra, S.N. *Modern Mathematical Statistics* New York: John Wiley & Sons, 1988.
9. Duhamel, P. et al. "On Computing the Inverse DFT." IEEE Transactions on ASSP. ASSP-34 (1986): 1 (February).
10. Dunn, O. and Clark, V. *Applied Statistics: Analysis of Variance and Regression* 2nd ed. New York : John Wiley & Sons. 1987.
11. Elliot, D.F. *Handbook of Digital Signal Processing Engineering Applications*. San Diego: Academic Press. 1987.
12. Harris, Fredric J. "On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform," Preceedings of the IEEE-66 (1978)-1.
13. Maisel, J.E. "Hilbert Transform Works With Fourier Transforms to Dramatically Lower Sampling Rates." Personal Engineering and Instrumentation News. PEIN-7 (1990): 2 (February).
14. Miller, I. and Freund, J.E. *Probability and Statistics for Engineers*. Englewood Cliffs, N.J. : Prentice-Hall, Inc. 1987.
15. Neter, J. et al. *Applied Linear Regression Models*. Richard D. Irwin, Inc. 1983.
16. Neuvo, Y., Dong, C.-Y., and Mitra, S.K. "Interpolated Finite Impulse Response Filters," IEEE Transactions on ASSP. ASSP-32 (1984): 6 (June).
17. O'Neill, M.A. "Faster Than Fast Fourier." BYTE. (1988) (April).
18. Oppenheim, A.V. & Schafer, R.W. *Discrete-Time Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1989.
19. Parks, T.W. and Burrus, C.S. *Digital Filter Design*. John Wiley & Sons, Inc.: New York. 1987.
20. Pearson, C.E. *Numerical Methods in Engineering and Science*. New York: Van Nostrand Reinhold Co. 1986.

21. Press, W.H. et al. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge: Cambridge University Press. 1988.

22. Rabiner, L.R. & Gold, B. *Theory and Application of Digital Signal Processing*. Englewood Cliffs, New Jersey: Prentice Hall. 1975.

23. Sorensen, H.V. et al. "On Computing the Split-Radix FFT." IEEE Transactions on ASSP. ASSP-34 (1986):1 (February).

24. Sorensen, H.V. et al. "Real-Valued Fast Fourier Transform Algorithms." IEEE Transactions on ASSP. ASSP-35 (1987): 6 (June).

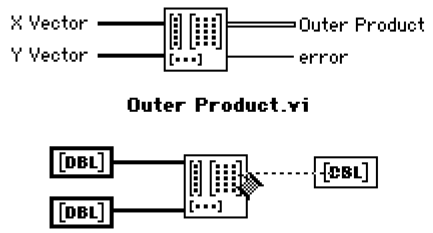
25. Stoer, J. and Bulirsch, R. *Introduction to Numerical Analysis*. New York : Springer-Verlag. 1987.

26. Vaidyanathan, P.P. Multirate Systems and Filter Banks. Englewood Cliffs, New Jersey: Prentice Hall. 1993.

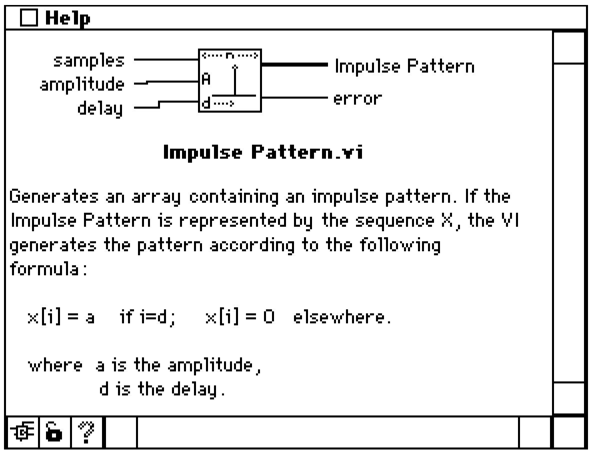
27. Wichman, B. and Hill, D. "Building a Random-Number Generator: A pascal routine for very-long-cycle random-number sequences." BYTE, March 1987, pp 127-128.

Getting Information about a VI

There are two quick ways to obtain information about a VI while in LabVIEW. To see the name of a VI and its parameters, go to the block diagram window and choose. Then place the cursor over the VI icon, as shown in the following illustration. The information in the Help window changes when you move the cursor to another VI or function.



You can resize the Help window by using the Positioning tool. Labview automatically adjusts the Help window size if the information does not fit on your resized screen. Once you resize the window, LabVIEW defaults to that size any time you access the Help window. The following illustration shows an example Help window.



Impulse Pattern.vi

Generates an array containing an impulse pattern. If the Impulse Pattern is represented by the sequence X, the VI generates the pattern according to the following formula:

x[i] = a   if i=d;   x[i] = 0   elsewhere.

where a is the amplitude,
d is the delay.

You can also access information about a VI through the Project menu. You use options from this menu to display a VIs hierarchy, callers, subVIs, unopened subVIs, and unopened typedefs. In addition, you can use the Find options on this menu to perform searches for objects or text in a particular VI, VI library, or all VIs in memory. If you choose the Show Profile Window option from this menu you can learn information about memory usage, the number of runs for a VI or subVI, average time spent on a particular VI or subVI, and so on.

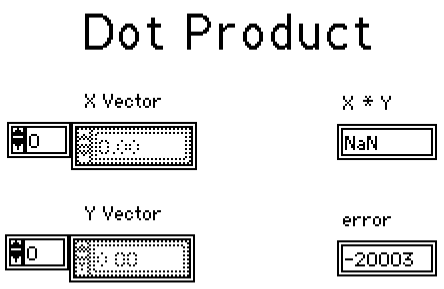
If you need more information, choose **Windows»Show VI Info...** to obtain a brief description of the operation performed by the VI.

You can open the front panel of a VI through the pop-up menu of the VIs icon or by double-clicking on the icon with the Operating or Positioning tools. The front panel shows the controls and indicators that the VI uses. When you display the VI connector pane and click on a control or indicator, the corresponding connector terminal turns black.

Analysis Error Reporting

Each analysis VI has an error output parameter, which returns a signed, 32-bit integer when invalid input conditions occur. Refer to Analysis Error Codes for the error condition that corresponds to the code.

In general, if the VI cannot resolve the error conflicts, or if it cannot complete the operation, the VI sets output arrays to empty arrays and floating-point, output scalars to undefined, which it displays as. The following figure shows the front panel of the Dot Product VI, which returns NaN and an error code of -20003, meaning the input arrays must not be empty, when you try to calculate the dot product of X Y.



Notation and Naming Conventions

To help you identify the type of parameters and operations, this manual uses the following notation and naming conventions unless otherwise specified in a VI description. Although there are a few scalar functions and operations, most of the analysis VIs process large blocks of data in the form of one-dimensional arrays (or vectors) and two-dimensional arrays (or matrices).

Normal lower case letters represent scalars or constants. For example,

$$a,$$
$$p,$$
$$b = 1.234.$$

Capital letters represent arrays. For example,

$$X,$$
$$A,$$
$$Y = a X + b.$$

In general, X and Y denote 1D arrays, and *A*, *B*, and *C* represent matrices.

Array indexes in LabVIEW are zero-based. The index of the first element in the array, regardless of its dimension, is zero. The following sequence of numbers represents a 1D array X containing n elements.

$$x = \{x_0, x_1, x_2, \ldots, x_{n-1}\}$$

The following scalar quantity represents the *i*th element of the sequence *X*.

$$x_i = 0 \leq i \leq n$$

The first element in the sequence is x_0 and the last element in the sequence is

x_{n-1} for a total of n elements.

The following sequence of numbers represents a 2D array containing n rows and m columns.

$$A = \begin{array}{ccccc} a_{00} & a_{01} & a_{02} & \cdots & a_{0m-1} \\ a_{10} & a_{11} & a_{12} & \cdots & a_{1m-1} \\ a_{20} & a_{21} & a_{22} & \cdots & a_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1m-1} \end{array}$$

The total number of elements in the 2D array is the product of n and m. The first index corresponds to the row number, and the second index corresponds to the column number. The following scalar quantity represents the element located on the i^{th} row and the

j^{th} column.

$$a_{ij}, 0 \leq i \leq n \text{ and } 0 \leq j \leq m$$

The first element in A is a_{00} and the last element is

a_{n-1m-1}.

Unless otherwise specified, this manual uses the following simplified array operation notations.

Setting the elements of an array to a scalar constant is represented by

$X = a,$

which corresponds to the sequence

$X = \{a, a, a, \ldots, a\}$

and is used instead of

$x_i = a, \qquad \text{for } i = 0,1,2,\ldots,n-1$

Multiplying the elements of an array bya a scalar constant is represented by

$Y = a X,$

which corresponds to the sequence

$Y = \left\{ax_0, ax_1, ax_2, \ldots, ax_{n-1}\right\}$

and is used instead of

$y_i = ax_i, \qquad \text{for } i = 0,1,2,\ldots,n-1.$

Similarly, multiplying a 2D array by a scalar constant is represented by

$B = k A,$

which corresponds to the sequence

$$A = \begin{array}{ccccc} ka_{00} & ka_{01} & ka_{02} & \cdots & ka_{0m-1} \\ ka_{10} & ka_{11} & ka_{12} & \cdots & ka_{1m-1} \\ ka_{20} & ka_{21} & ka_{22} & \cdots & ka_{2m-1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ ka_{n-10} & ka_{n-11} & ka_{n-12} & \cdots & ka_{n-1m-1} \end{array}$$

and is used instead of

$b_{ij} = ka_{ij}, \text{for } i = 0,1,2,\ldots,n-1 \text{ and } j = 0,1,2,\ldots,m-1$

Empty arrays are possible in LabVIEW. An array with no elements is an empty array and is represented by

Empty = NULL = $\varnothing$= { } .

In general, operations on empty arrays result in empty, output arrays or undefined results.

Sampling Signals

To use digital signal processing techniques, you must convert an analog signal into its digital representation. This section includes only a brief discussion of the notation that represents a digital signal. This section does not discuss the mathematical background or problems associated with sampling techniques.

Consider an analog signal $x(t)$ and the sampling interval Δt. The signal $x(t)$ can be represented by the discrete sequence of samples

$\{x(0), x(\Delta t), x(2\Delta t), x(3\Delta t), \ldots, x(k\Delta t), \ldots\}$.

Because Δt establishes only the sampling rate and has no bearing on the actual sampled (digitized) value, the sample at

$t = i\Delta t$, for $i = 0, 1, 2, \ldots$

corresponds to the $\leq$ element in the sequence.

Thus,

$x_i = x(i\Delta t)$

and $x(t)$ can be represented by the sequence X whose values are

$X = \{x_0, x_1, x_2, x_3 \ldots, x_k, \ldots\}$

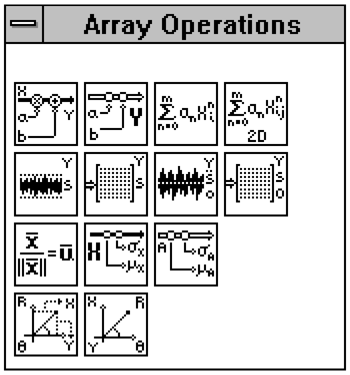
If *n* samples are obtained from the signal $x(t)$, en the sequence

$X = \{x_0, x_1, x_2, x_3 \ldots, x_{n-1}\}$

is the digital representation or the sampled version of $x(t)$.

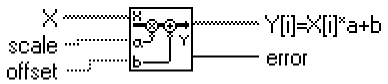
Array Operation VIs

This topic describes the VIs that perform common, one- and two-dimensional numerical analysis. The following illustration shows the options that are available on the **Array Operation** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



1D Linear Evaluation

Performs a linear evaluation of the input array **X**.



[DBL] **X** consists of the input array.
DBL **scale** is the multiplicative constant.
DBL **offset** is the additive constant.
[DBL] **Y[i] = X[i] * a + b** is the output array.
I32 **error**. See Analysis Error Codes for a description of the error.

The output array **Y[i] = X[i]*a + b** is given by

$$Y=aX+b$$

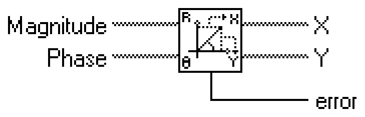
where a is the multiplicative **scale** constant, and b is the additive constant **offset**.

1D Polar To Rectangular (Advanced Only)

Converts two arrays of polar coordinates into two arrays of rectangular coordinates, according to the following formulas:

x = **magnitude** cos(**phase**)

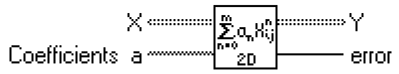
y = **magnitude** sin(**phase**).



≤ **Magnitude** is a one-dimensional array of polar coordinates.
≤ **Phase** is a one-dimensional array of polar coordinates. You must express Phase in radians.
DBL **X** is a one-dimensional array of rectangular coordinates.
DBL **Y** is a one-dimensional array of rectangular coordinates.
≤ **error.** See Analysis Error Codes for a description of the error.

1D Polynomial Evaluation

Performs a polynomial evaluation of **X** using **Coefficients: a**.



≤ **X** is the input data to be used in the polynomial evaluation.
≤ **Coefficients: a**. The total number of elements in **Coefficients: a** is the polynomial order plus one.
≤ **Y** is the output data.
≤ **error**. See Analysis Error Codes for a description of the error.

The output array **Y** is given by

$$Y = \sum_{n=0}^{m} a_n X^n$$

where m denotes the polynomial order.

1D Rectangular To Polar (Advanced Only)

Converts two arrays of rectangular coordinates into two arrays of polar coordinates, according to the following formulas:

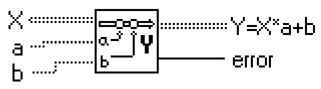
$$\text{magnitude} = \sqrt{x^2 + y^2}$$

$$\text{phase} = \tan^{-1}\frac{y}{x}$$

≤ **X** is a one-dimensional array of rectangular coordinates.
≤ **Y** is a one-dimensional array of rectangular coordinates.
≤ **Magnitude** is a one-dimensional array of polar coordinates.
≤ **Phase** is a one-dimensional array of polar coordinates. Phase is expressed in radians.
≤ **error**. See Analysis Error Codes for a description of the error.

2D Linear Evaluation

Performs a linear evaluation of the two-dimensional input array **X**.



≤ **X** is the two-dimensional input array.
≤ **a** is the multiplicative constant.
≤ **b** is the additive constant.
≤ **Y = X * a + b** is the two-dimensional output array.
≤ **error**. See Analysis Error Codes for a description of the error.

The two-dimensional output array **Y = X*a + b** is given by

$$Y=aX+b,$$

where a denotes the multiplicative constant, and b denotes the additive constant.

2D Polynomial Evaluation

Performs a polynomial evaluation of the two-dimensional input array **X** using **Coefficients a**.

≤
≤ **X** is the two-dimensional input array.
≤ **Coefficients a**. The total number of elements in **Coefficients a** is the polynomial order plus one: m + 1.
≤ **Y** is the two-dimensional output array.
≤ **error**. See Analysis Error Codes for a description of the error.

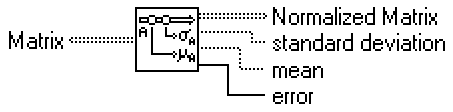
The two-dimensional output array **Y** is given by

$$Y = \sum_{n=0}^{m} a_n X^n$$

where m denotes the polynomial order.

Normalize Matrix (Advanced Only)

Normalizes the 2D input **Matrix** using its statistical profile (μ, s), where μ is the **mean** and s is the **standard deviation**, to obtain a **Normalized Matrix** whose statistical profile is (0,1).



≤ **Matrix**. If **Matrix** is an empty array, **Normalized Matrix** is also an empty array, and **mean** and **standard deviation** are NaN.
≤ **Normalized Matrix** is the output normalized matrix.
≤ **standard deviation** is the standard deviation of **Matrix**.
≤ **mean** is the mean of **Matrix**.
≤ **error**. See Analysis Error Codes for a description of the error.

The VI obtains **Normalized Matrix** using

$$B = \frac{A - \mu}{\sigma},¶$$

$$\mu = \frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} a_{ig}}{n \bullet m},$$

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n-1} \sum_{j=0}^{m-1} \left(a_{ij} - \mu\right)^2}{n \bullet m}},$$

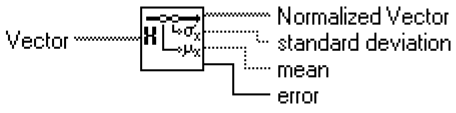
where B represents the 2D output sequence **Normalized Matrix**, A represents the 2D input sequence **Matrix** with n rows and m columns, and a_{ij} is the element of A on the

≤ row and
≤ column.

Normalize Vector (Advanced Only)

Normalizes the input **Vector** using its statistical profile (≤,s), where

≤ is the **mean** and s is the **standard deviation**, to obtain a **Normalized Vector** whose statistical profile is (0,1).



≤ **Vector**. If **Vector** is an empty array, **Normalized Vector** is also an empty array, and **mean** and **standard deviation** are NaN.
≤ **Normalized Vector** is the output normalized vector.
≤ **standard deviation** is the standard deviation of **Vector**.
≤ **mean** is the mean of **Vector**.
≤ **error**. See Analysis Error Codes for a description of the error.
The VI obtains **Normalized Vector** using

$$Y = \frac{X - \mu}{\sigma},$$

$$\mu = \frac{\sum_{i=0}^{n-1} x_i}{n},$$

$$\sigma = \sqrt{\frac{\sum_{i=0}^{n-1} \left(x_i - \mu\right)^2}{n}},$$

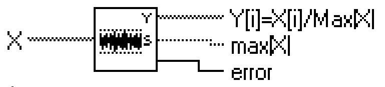
where Y represents the output sequence **Normalized Vector**, and *X* represents the input sequence

Vector of length n, and x_i is the

≤ element of X.

Quick Scale 1D (Advanced Only)

Determines the maximum absolute value of the input array **X** and then scales **X** using this value.



≤ **X** is the input array.
≤ **Y[i] = X[i]/Max|X|** is the output array.
≤ **max{X}** is the maximum absolute value in the input array.
≤ **error**. See Analysis Error Codes for a description of the error.

The output array **Y[i] = X[i]/Max|X|** is given by

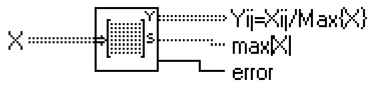
$$Y = \frac{X}{s},$$

where s is the maximum absolute value in **X**.

You can use this VI to normalize sequences within the range [-1:1]. This VI is particularly useful if the sequence is a zero mean sequence.

Quick Scale 2D (Advanced Only)

Determines the maximum absolute value of the input array **X** and then scales **X** using this value.



≤ **X** is the two-dimensional input array.
≤ **Yij = Xij/Max{X}** is the two-dimensional output array.
≤ **max|X|** is the maximum absolute value in **X.**
≤ **error**. See Analysis Error Codes for a description of the error.

The output array **Yij = Xij/Max{X}** is given by

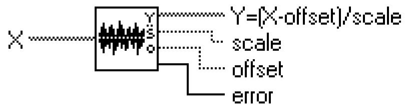
$$Y = \frac{X}{s},$$

where s denotes the maximum absolute value in **X**.

You can use this VI to normalize sequences within the range [-1:1]. This VI is particularly useful if the sequence is a zero mean sequence.

Scale 1D (Advanced Only)

Determines **scale** and **offset** and then scales the input array **X** using these values.



≤ **X** is the input array.
≤ **Y = (X-offset)/scale** is the output array.
≤ **scale** is the scaling factor.
≤ **offset** is the offset factor**.**
≤ **error**. See Analysis Error Codes for a description of the error.

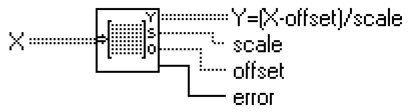
The output array Y is given by

$$Y = \frac{X - \text{offset}}{\text{scale}},$$

scale = 0.5(max - min), and
offset = min + **scale**,
where max denotes the maximum value in **X**, and min denotes the minimum value in **X**.
You can use this VI to normalize any numerical sequence with the assurance that the range of the output sequence is [-1:1].

Scale 2D (Advanced Only)

Determines **scale** and **offset** and then scales **X** using these values.



- ≤ **X** is the two-dimensional input array.
- ≤ **Y = (X - offset)/scale** is the two-dimensional output array.
- ≤ **scale** is the scaling factor.
- ≤ **offset** is the offset factor.
- ≤ **error**. See Analysis Error Codes for a description of the error.

The two-dimensional output array **Y = (X - offset)/scale** is given by

$$Y = \frac{X - \text{offset}}{\text{scale}},$$

scale = 0.5(max - min), and

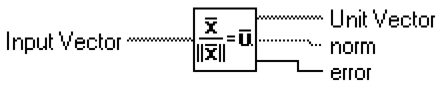
offset = min + 0.5 **scale**,

where max denotes the maximum value in **X**, and min denotes the minimum value in **X**.

You can use this VI to normalize any numerical sequence with the assurance that the range of the output sequence is [-1:1].

Unit Vector (Advanced Only)

Finds the **norm** of the **Input Vector** and obtains its corresponding **Unit Vector** by normalizing the original **Input Vector** with its **norm**.



- ≤ **Input Vector**. If **Input Vector** is an empty array, **Unit Vector** is also an empty array, and **norm** is NaN.
- ≤ **Unit Vector** is the output, normalized vector.
- ≤ **norm**.
- ≤ **error**. See Analysis Error Codes for a description of the error.

Let X represent the input **Input Vector**; **norm** is given by/

$$\|X\| = \sqrt{x_0^2 + x_1^2 + \ldots + x_{n-1}^2},$$

where $\|X\|$ is **norm**, and the VI calculates **Unit Vector,** U, using

$$U = \frac{X}{\|X\|}.$$

1D Linear Evaluation.vi

1D Linear Evaluation

1D Polar To Rectangular.vi

1D Polar To Rectangular

1D Polynomial Evaluation.vi

1D Polynomial Evaluation

1D Rectangular To Polar.vi

1D Rectangular To Polar

Normalize Matrix.vi

Normalize Matrix

Normalize Vector.vi

Normalize Vector

Quick Scale 1D.vi

Quick Scale 1D

Quick Scale 2D.vi

Quick Scale 2D

Scale 1D.vi

Scale 1D

Scale 2D.vi

Scale 2D

Unit Vector.vi

Unit Vector

2D Linear Evaluation.vi

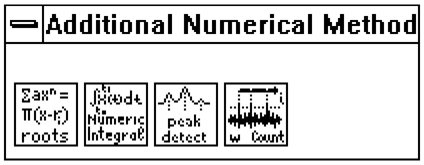
2D Linear Evaluation

2D Polynomial Evaluation.vi

2D Polynomial Evaluation

Additional Numerical Method VIs

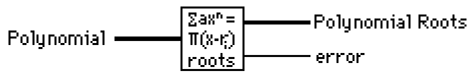
This topic describes the VIs that use numerical methods to perform root-finding, numerical integration, and peak detection. The following illustration shows the options that are available on the **Additional Numerical Method** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Complex Polynomial Roots
Numeric Integration
Peak Detector
Threshold Peak Detector

Complex Polynomial Roots (Advanced Only)

Finds the complex roots of a complex polynomial.



[CDB] **Polynomial** is the array of complex coefficients from the lowest to the highest order. For example, the second order polynomial as described by the 3-element array

$A = \{a_0, a_1, a_2\}$ would be:

$A = a_0 + a_1 x + a_2 x^2$,

This VI finds the two complex roots of the above polynomial.

Polynomial Roots is the array of complex roots of the complex Polynomial. The array always contains the same number of roots as the polynomial order, which is one less than the number of coefficients in the input Polynomial. For example, for the second order polynomial:

$a_0 + a_1 x + a_2 x^2$

Polynomial Roots would contain two complex roots {r0,r1}, such that

$$a_0 + a_1 x + a_2 x^2 = (x - r_0)(x - r_1)$$

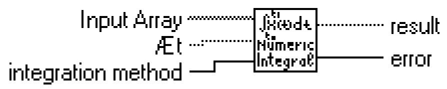
error. See Analysis Error Codes for a description of the error.

This VI uses a modified, complex Newton method to determine the n complex roots (some of which may be real, with a zero imaginary part), of the general complex polynomial:

$$a_0 + a_1 x + a_2 x^2 + \ldots a_{n-1} x^{n-1} + a_n x^n.$$

Numeric Integration (Advanced Only)

Performs a numeric integration on the input array of data using one of four, popular numeric integration methods.



≤ **Input Array** contains the data to be integrated, which is obtained from sampling some function f(t) at multiples of dt, that is, f(0), f(dt), f(2dt),.....

≤ **dt** is the interval size, which represents the sampling step size used in obtaining data in **Input Array** from the function. If you supply a negative dt, the VI uses its absolute value.

I32 **integration method** specifies the method used in performing the numeric integration.
0: Trapezoidal Rule
1: Simpsons' Rule
2: Simpsons' 3/8 Rule
3: Bode Rule

≤ **result** contains the result of the numeric integration.

≤ **error** reports any error encountered during execution.

Note: If the number of points provided for a certain chosen method does not contain an integral number of partial sums, then the method is applied for all possible points. For the remaining points, the next possible lower order method is used. For example, if the Bode method is selected, the following table shows what this VI evaluates for different numbers of points:

Number of Points	Partial Evaluations Performed
224	56 Bode
225	56 Bode, 1 Trapezoidal
226	56 Bode, 1 Simpsons'
227	56 Bode, 1 Simpsons' 3/8
228	57 Bode

So, if 227 points were provided and the Bode Method was chosen, the VI would arrive at the result by performing 56 Bode Method partial evaluations and one Simpsons' 3/8 Method evaluation.

Each of the methods depend on the sampling interval (**dt**) and compute the integral using successive applications of a basic formula in order to perform partial evaluations, which depend on some number of adjacent points. The number of points used in each partial evaluation represents the order of the method. The result is the summation of these successive partial evaluations.

$$\text{result} = \int_{t0}^{t1} f(t)\,dt \approx \sum_{j} \text{partial sums}$$

where *j* is a range dependent on the number of points and the method of integration.

The basic formulas for the computation of the partial sum of each rule in ascending method order are:

Trapezoidal: (x[i] + x[i+1])*dt, k = 1

Simpsons': (x[2i] + 4x[2i+1] + x[2i+2])*dt/3, k = 2

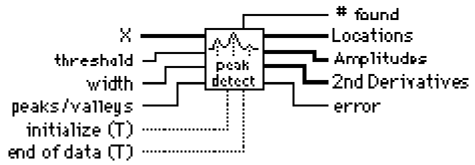
Simpsons' 3/8: (3x[3i] + 9x[3i+1] + 9x[3i+2] + 3x[3i+3]) * dt/8, k = 3

Bode: (14x[4i] + 64x[4i+1] + 24x[4i+2] + 64x[4i+3] + 14x[4i+4])*dt/45, k = 4 for i = 0, k, 2k, 3k, 4k..., Integral Part of [(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.

Peak Detector (Advanced Only)

Finds the location, amplitude, and second derivative of peaks or valleys in the input array.



≤ **X** is the input that holds the data to be processed. The data can be a single array or consecutive blocks of data. Consecutive blocks of data are useful for large, data arrays or for real time processing. Notice that in real time processing, **peaks/valleys** are not detected until approximately **width**/2 data points past the peak or valley.
≤ **threshold** is the input that rejects **peaks/valleys** that are too small. For peaks, any peak found with a fitted amplitude that is less than **threshold** is ignored. Valleys are ignored if the fitted trough is greater than **threshold**.
≤ **width** is the input that specifies the number of consecutive data points to use in the quadratic least squares fit. The value should be no more than about 1/2 of the half-width of the **peaks/valleys** and can be much smaller for noise-free data. Large widths can reduce the apparent amplitude of peaks and shift the apparent location.
≤ **peaks/valleys**. You use this control to choose between looking for peaks (positive-going bumps) and valleys (negative-going bumps). The settings for this control are 0 (peaks) and 1 (valleys).
TF **initialize**. Set this control to TRUE to process the first block of data. The VI requires some internal setup at the beginning for proper operation. If you only want to process one block of data, leave **initialize** unwired, or set its default state to TRUE.If you want to process consecutive blocks of data, set **initialize** to TRUE for the first block and FALSE for all other blocks of data.
≤ **end of data**. Set this control to TRUE to process the last block of data. After processing the last block of data, the VI manages internal data. If you only want to process one block of data, leave **end of data** unwired, or set its default state to TRUE. If you want to process consecutive blocks of data, set **end of data** to FALSE for all but the last block of data.
≤ **Locations** is an array containing the locations of **peaks/valleys** found in the current block of data. **Locations** are reported in indices from the beginning of processing.
≤ **Amplitudes** is an array containing the amplitudes of **peaks/valleys** found in the current block of data.
≤ **2nd Derivatives** is an array containing the second derivatives of **peaks/valleys** found in the current block of data.
≤ **# found** is the number of **peaks/valleys** found in the current block of data. **# found** is the size of the arrays **Locations**, **Amplitudes**, and **2nd Derivatives**.
≤ **error**. See Analysis Error Codes for a description of the error.
The data set can be passed to the VI as a single array or as consecutive blocks of data.

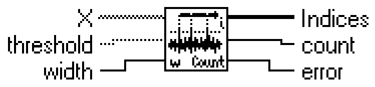
This VI is based on an algorithm that fits a quadratic polynomial to sequential groups of data points. The number of data points used in the fit is specified by **width**.

For each peak or valley, the quadratic fit is tested against the threshold level: peaks with heights lower than the threshold or valleys with troughs higher than the threshold are ignored. **peaks/valleys** are detected only after approximately **width**/2 data points have been processed beyond **peaks/valleys** locations. This delay has implications only for real time processing.

The VI must be notified when the first and last blocks are passed into the VI, so that the VI can initialize and then release data internal to the peak detection algorithm.

Threshold Peak Detector (Advanced Only)

Analyzes the input sequence **X** for valid peaks and keeps a **count** of the number of peaks encountered and a record of **Indices**, which locates the points that exceed the **threshold** in a valid peak. A peak is valid where the elements of X exceed the threshold and then return to a value less than or equal to the threshold, and the number of elements that exceed the **threshold** is at least equal to **width**.



≤ **X**. The number of samples in **X** must be greater than the specified **width**. If **X** is less than or equal to **width**, the VI sets **count** to zero and returns an error.

≤ **threshold** defaults to 0.0.

≤ **width** must be greater than zero. If **width** is less than or equal to zero, the VI sets **count** to zero and returns an error. **width** defaults to 1.

[I32] **Indices**.

≤ **count**.

≤ **error**. See Analysis Error Codes for a description of the error.

Complex Polynomial Roots.vi

Complex Polynomial Roots

Numeric Integration.vi

Numeric Integration

Peak Detector.vi

Peak Detector

Threshold Peak Detector.vi

Threshold Peak Detector

Analysis Examples

This topic discusses the theoretical and practical aspects of using the analysis VIs in actual applications and provides examples of common uses for the VIs. These examples illustrate how easy it is to use the analysis VIs and can help you develop more sophisticated applications.

Several of these examples address common user questions. With these examples, novice users can quickly understand the principles of digital data processing and apply them to their applications, while advanced users can learn more about high-level analysis functions.

Following is a list of Analysis VI examples by category:

Curve Fitting Examples
Echo Detection
Filtering Examples
Frequency Information Displayed
Parseval's Theorem
Pulse Demo
Statistical Examples
Windowing Example

These examples require no special code, files, techniques, applications, or dedicated VIs. These examples are included in the `examples directory` in the LabVIEW distribution disks.

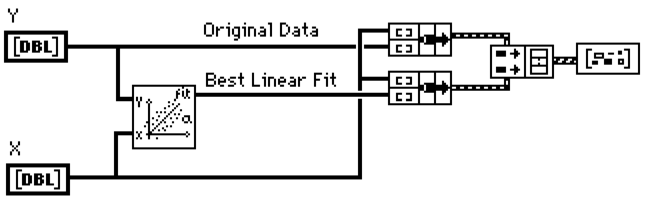
Curve Fitting Examples

In some applications, parameters such as humidity, temperature, and pressure can affect data you collect. You can model the statistical data by performing regression analysis and gain insight into the parameters that affect the data.

Two examples in this section illustrate how to use curve fitting VIs. The first example demonstrates how to use the Linear Fit, Exponential Fit, General Polynomial Fit, and related VIs. The second example illustrates how to use the General LS Linear Fit VI.

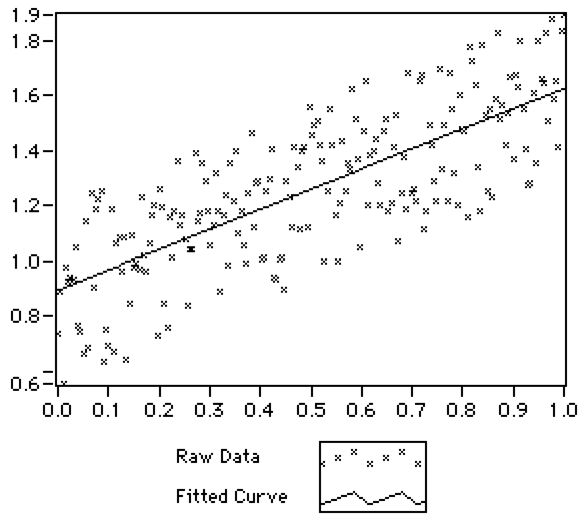
Fitting a Line to Data

The following block diagram shows how to fit a line to a set of data points using the Linear Fit VI.



You can modify the example to fit exponential and polynomial curves by replacing the Linear Fit VI with the Exponential Fit VI or the General Polynomial Fit VI.

The following multiplot graph shows the result of fitting a line to the noisy data set.



General LS Linear Fit

The following example demonstrates not only how to use the General LS Linear Fit VI to obtain the set of least square coefficients **a** and the fitted values, but also how to set up the input parameters.

The purpose is to find the set of least square coefficients **a** that best represent the set of data points (x,y). The relationship between x and y is of the form

$$y = f(a,x) = \sum_{i=0}^{n-1} a_i f_i(x) = a_0 f_0(x) + a_1 f_1(x) + \ldots + a_{n-1} f_{n-1}(x)$$

where

$$a = \{a_0, a_1, a_2, \ldots, a_{n-1}\}$$

and n is the total number of functions.

Assume the data is generated using the relationship

$$y = 2h_0(x) + 3h_1(x) + 4h_2(x) + \text{noise}$$

where

$$h_0(x) = \sin(x^2),$$

$$h_1(x) = \cos(x),$$

$$h_2(x) = \frac{1}{x+1}, \text{ and}$$

noise is a random value.

Also, assume that you think the relationship between **x** and **y** is of the form

$$y = a_0 f_0(x) + a_1 f_1(x) + a_2 f_2(x) + a_3 f_3(x) + a_4 f_4(x)$$

where

$$f_0(x) = 1.0,$$

$$f_1(x) = \sin(x^2),$$

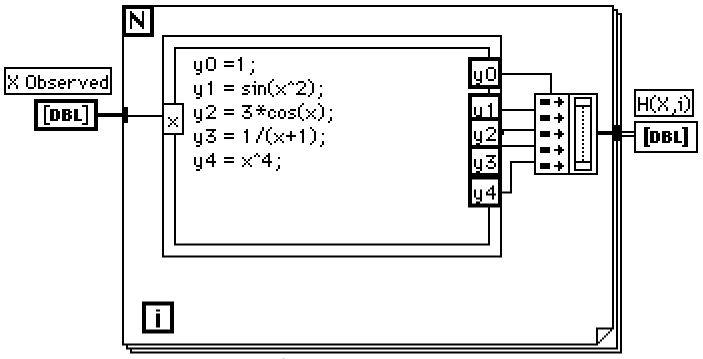
$$f_2(x) = 3\cos(x),$$

$$f_3(x) = \frac{1}{x+1},$$

$$f_4(x) = x^4.$$

To obtain the coefficients **a**, you must supply the set of (x,y) points in the arrays **X** and **Y** and you must also supply the basis function **H(X,i)**, which is a 2D array, to the General LS Linear Fit VI.

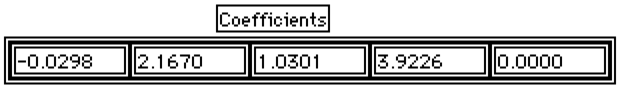
The arrays **X** and **Y** are the values observed in your experiment. A simple way to provide the basis function **H(X,i)** is shown in the following diagram.



You can easily edit the formula node to change, add, or delete functions. At this point, we have all the necessary inputs to use the General LS Linear Fit VI to solve for **a**. The expected set of coefficients are

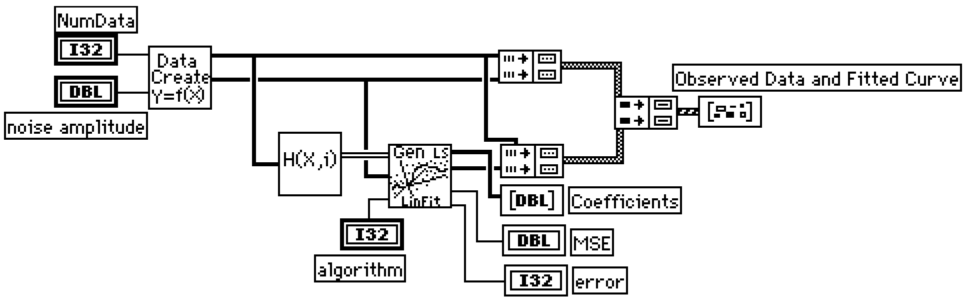
a={0.0, 2.0, 1.0, 4.0, 0.0}

Executing the General LS Linear Fit VI with the values of **X**, **Y**, and **H(X,i)** returns the following set of coefficients.



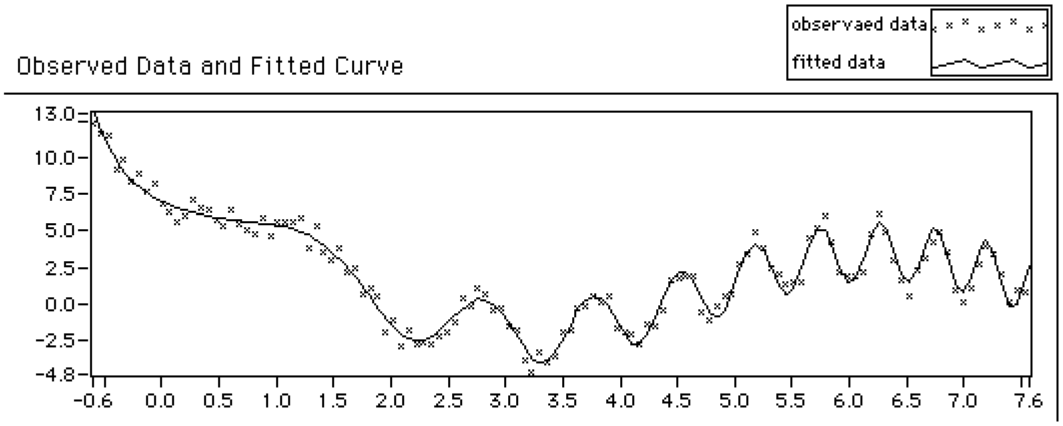
Coefficients				
-0.0298	2.1670	1.0301	3.9226	0.0000

The block diagram below demonstrates how to set up the General LS Linear Fit application to obtain the coefficients and a new set of y values.



The icon labelled Data Create generates the **X** and **Y** arrays. You can replace this icon with one that actually collects the data in your experiments. The icon labelled H(X,i) generates the 2D **H(X,i)** basis function.

The last portion of the diagram overlays the original and the estimated data points and produces a visual record of the General LS Linear Fit. The results are shown in the following graph.



General LS Linear Fit VI has six different algorithms to obtain the set of coefficients and the fitted values. In this example, there is no significant difference among different algorithms. You can select different algorithms from the front panel to see the results. In some cases, different algorithms may have significant differences, depending on your observed data set.

Echo Detection

Hilbert transforms are used extensively in the analysis of modulation systems, such as echo detection.

Consider the time signal of the form

$$x(t) = Ae^{-t/\tau}\cos(2\pi f_0 t)$$

and its Hilbert transform

$$x_H(t) = -Ae^{-t/\tau}\sin(2\pi f_0 t)$$

where A is the amplitude, f_0 is the natural resonant frequency, and

τ is the time decay onstant.

The natural logarithm of the magnitude of the analytic signal $x_A(t)$ is given by

$$\ln|x_A(t)| = \ln|x(t) + jx_H(t)| = -\frac{t}{\tau} + \ln A,$$

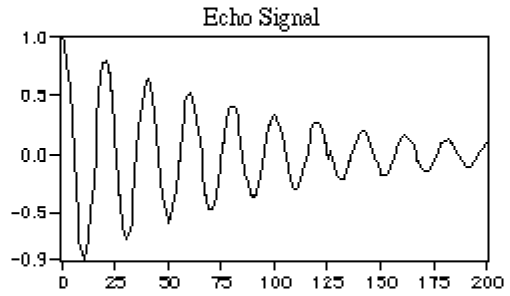
which has the form of a line with slope

$$m = -\frac{1}{\tau}$$

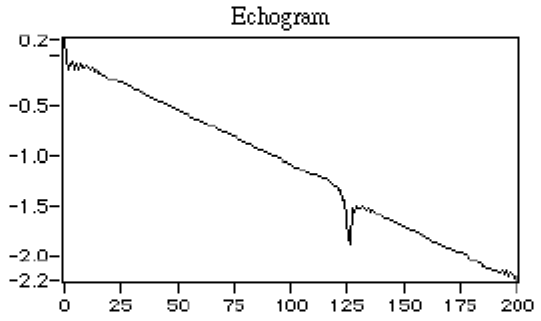
Thus, you can extract the time constant of the system by graphing

$$\ln|x_A(t)|$$

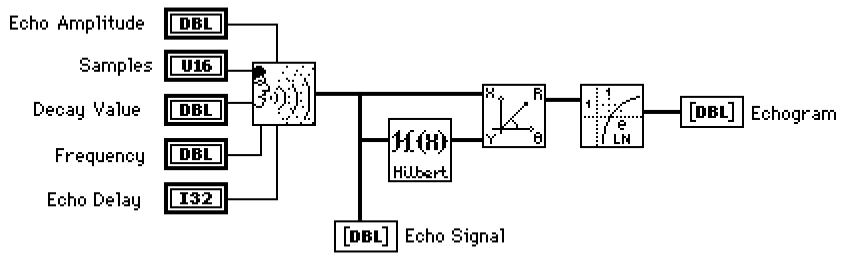
Consider the echo signal shown in the following graph. The echo signal is difficult to locate because the time delay between the source and the echo signal is short relative to the time decay constant of the system and because the echo amplitude is small compared to the source.



You can make the echo signal visible by plotting the magnitude of xA(t) on a logarithmic scale. The discontinuity that now appears in the following graph indicates the location of the time delay of the echo.



The following block diagram shows a simple echo detector. All the necessary nodes are either LabVIEW functions or analysis VIs.



This example generates the analytic signal using the Fast Hilbert Transform VI, finds its magnitude with the 1D Rectangular To Polar VI, and computes the natural log to detect the presence of an echo.

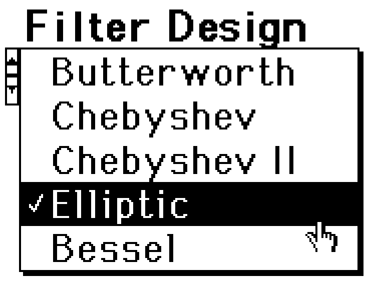
Filtering Examples

FIR Filter Design
Noisy Pulse Analyzed with a Median Filter

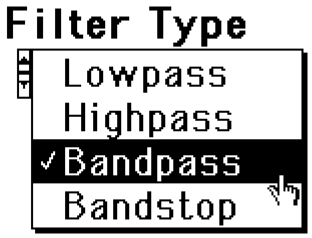
IIR Filter Design

When using a digital filter to process data, you should know its spectral characteristics. You can use the following example to examine the spectral properties of the IIR filter VIs.

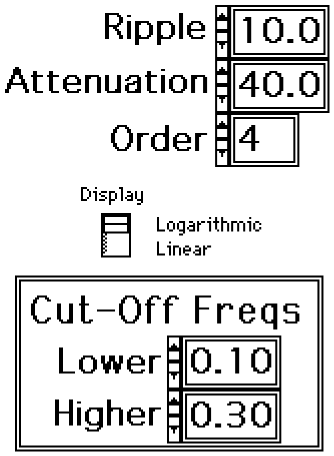
LabVIEW has five types of recursive filter VIsButterworth, Chebyshev, Chebyshev II, Elliptic, and Bessel.



Each type of filter has four basic, often-used configurations: lowpass, highpass, bandpass, and bandstop (or notch).



The filter parameters you can control are the lower and higher cut-off frequencies, the filter order, the passband ripple in decibels, and the stopband attenuation ripple. This example displays the spectral function linearly or in decibels.



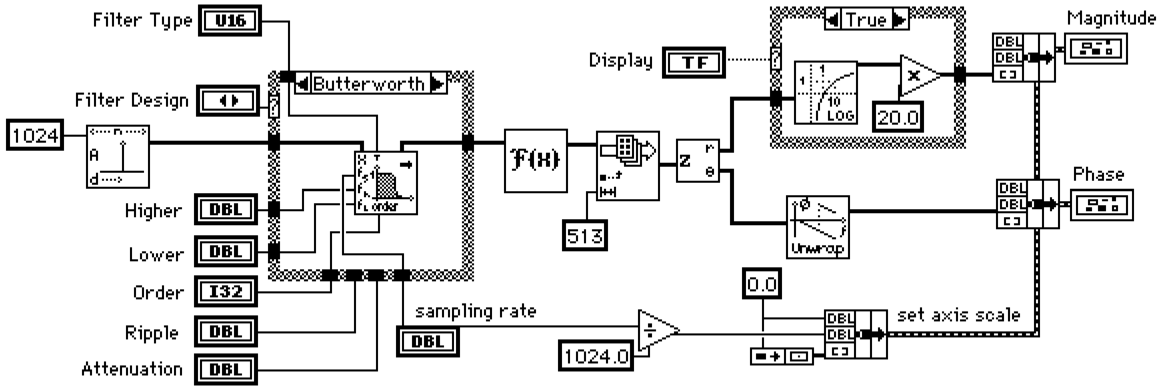
The following block diagram determines the spectral response of the filters. The diagram passes an impulse signal through a filter to obtain the magnitude and phase response of that filter. The case structures immediately to the right of the Impulse Pattern VI select the filter design (Butterworth, Chebyshev, Chebyshev II, Elliptic, or Bessel) and type (lowpass, highpass, bandpass, or bandstop). The signal obtained from the case structure is the impulse response of the system.

The transfer function of the system corresponds to the impulse response via the Fourier transform such that the impulse response and the transfer function are Fourier transform pairs

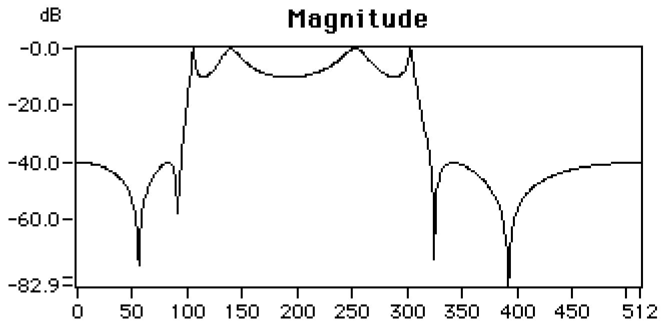
$h(t) \Leftrightarrow H(f)$,

where $h(t)$ is the impulse response, and $H(f)$ is the transfer function (frequency response).
Because the case structure output signal is the impulse response, you can derive the transfer function with a Fourier transform. Half of the information is redundant, so you need to process only half of the information after the FFT VI. Magnitude and phase information are much easier to interpret than the real and imaginary component of the FFT, and you can therefore use the 1D Rectangular to Polar VI to obtain the magnitude and phase. Finally, unwrap the phase and convert it to degrees and convert the magnitude

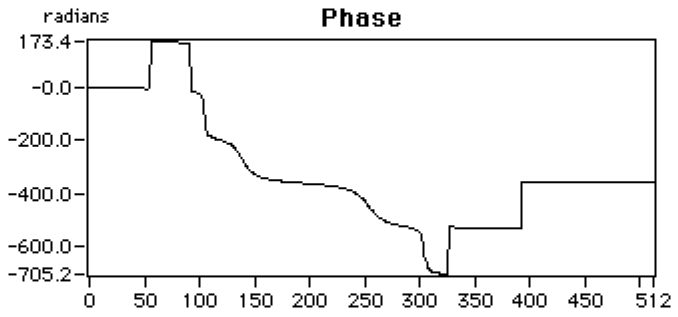
to decibels.



The following graph shows the magnitude of an elliptic bandpass filter.



The following graph shows the phase response of an elliptic bandpass filter.

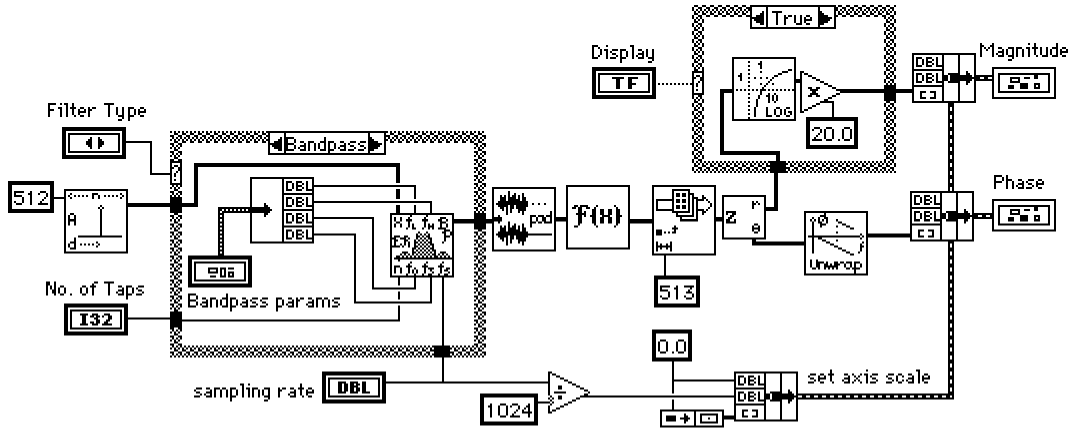


Notice that the phase information is clearly nonlinear and should be considered when selecting IIR or FIR filters to process data.

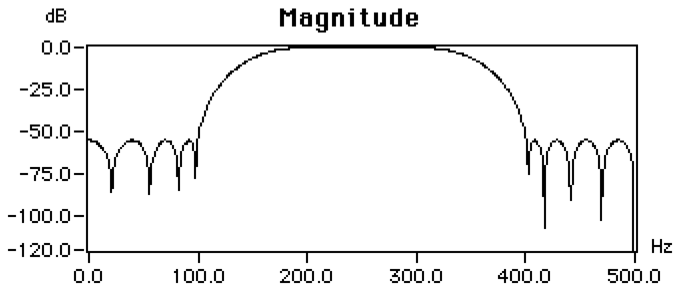
FIR Filter Design

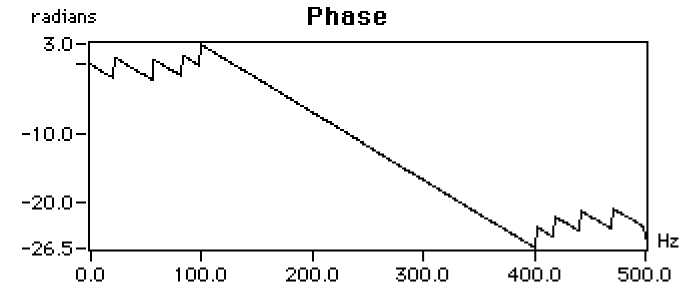
You should use FIR filters to produce linear phase responses. Linear phase response implies that all frequencies in the system have the same propagation delay.

The following block diagram displays the frequency response of Equi-Ripple FIR filters. The diagram is similar to that used in the IIR Filter Design example because the same mathematical theory applies to this type of filter.



The following graphs show the magnitude and phase response of a bandpass FIR filter.





The discontinuities in the phase are a result of using the absolute value of the magnitude. Still, it is linear response as described in the Finite Impulse Response Filters section.

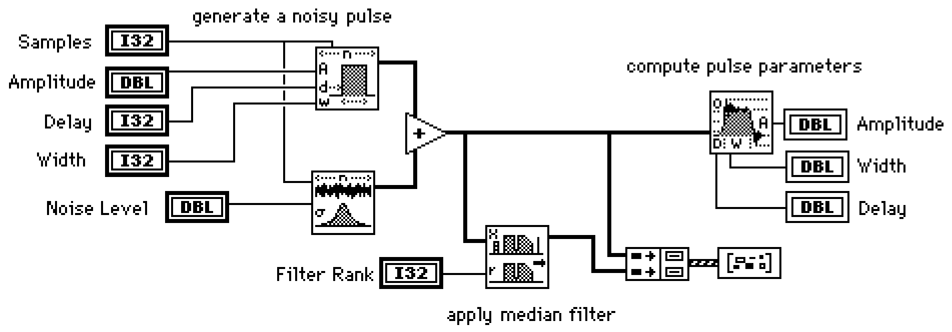
Noisy Pulse Analyzed with a Median Filter

One of the conditions the Pulse Parameters VI imposes is that the expected peak amplitude of the noise portion of the signal be less than or equal to 50% of the expected pulse amplitude. This condition is necessary because after the VI completes the modal analysis to determine the baseline and the top, it is difficult to discriminate between noise and signal without more information. In some practical applications, this kind of pulse-to-noise ratio is difficult to achieve, and you must do some preprocessing to extract pulse information.

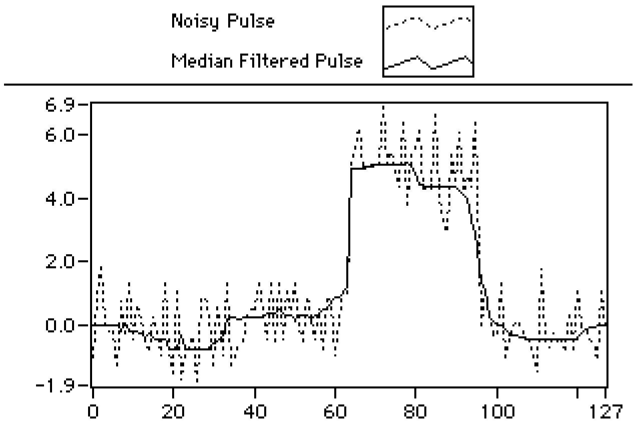
If the pulse is buried in noise whose expected peak amplitude exceeds 50% of the expected pulse amplitude, using a lowpass filter removes some of the unwanted noise. However, the filter also shifts the signal in time and smears the edges of the pulse because these transition edges contain high frequency information.

A median filter can extract the pulse more effectively. A median filter is a nonlinear filter that removes high frequency noise while preserving edge information.

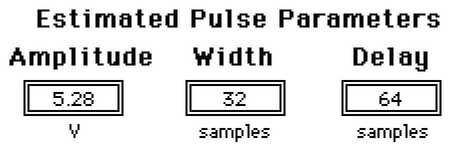
The following block diagram demonstrates how to use the Median Filter VI to successfully analyze a noisy pulse whose expected peak noise amplitude is greater than 100% of the expected pulse amplitude.



The following multiplot graph shows how you can easily track the pulse signal with the aid of a median filter in spite of the fact that the pulse is deeply buried in noise.



By removing the high frequency noise with the Median Filter VI, you can attain the condition for the Pulse Parameter VI and complete the analysis correctly. The results shown in the following figure correspond to the analyzed pulse in the previous multiplot graph.



Estimated Pulse Parameters

Amplitude	Width	Delay
5.28	32	64
V	samples	samples

The ideal pulse values when the signal was generated were as follows.

Amplitude:	5.0 V
Delay:	64 samples
Width:	32 samples

Frequency Information Displayed

Two of the principal challenges in scientific analysis are mathematically describing the Fourier transfer and understanding its properties. The most common applications of the Fourier transform are the analysis of linear time-invariant systems and spectral analysis, but this transform is most important because it gives the scientist a way to examine a relationship from the frequency domain point of view.

Two Sided DC Centered Fast Fourier Transforms
Frequency Information Obtained from Transforms

Most introductory textbooks in linear systems, digital signal processing, image processing, and other related applications discuss the two-sided mathematical description of the Fourier transform

$$X(f) = F\{x(t)\} = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt,$$

and its inverse

$$X(t) = F^{-1}\{X(f)\} = \int_{-\infty}^{\infty} e^{j2\pi ft}dt.$$

Two-sided information means that all the negative and positive frequencies and time are considered in the mathematical implementation of the forward and inverse Fourier transform. Single-sided or one-sided information considers only the positive frequencies and time history of the signal.

A Fourier transform pair consists of the signal represented in both the time and frequency domain. The notation

$$x(t) \leq X(f)$$

is commonly used to represent a Fourier transform pairfor example, tri(t)≤ sinc^2
Recent advances in the computation of FFTs, Discrete Fourier Transforms (DFTs), and their inverse operations led to the adoption of the frequency information presentation format used in the FFT VI description in the Digital Processing Signal VIs section of this manual. This format is used principally for speed and processing convenience.
When dealing with test and measurement applications, frequency information in the format used in the above topic often appears awkward to scientists. There are two other common ways to present this informationdisplaying the DC component in the center and displaying one-sided spectrums.

To convert to a DC-centered Fourier transform, the quick and obvious solution is to copy buffers from one place to another. In the case of converting single-side band information, the solution involves extracting the frequency information and multiplying each value by 2 (excluding DC and Nyquist components). A special case occurs when the length of the sequence is even, which is also a requirement for the computation of the FFT because its length has to be a power of 2.

The three examples that follow illustrate how to preprocess time sequences to obtain the desired frequency display format. The first two examples briefly discuss alternate and simpler methods for obtaining DC-centered and single-sideband Fourier transforms without manipulating buffers. The third example discusses how to obtain and display correct frequency information of the FFT for a given sampling interval. You can extend this example to incorporate it into the DC-centered and single-sideband FFT cases.

Two-Sided (DC-Centered) Fast Fourier Transforms

Most introductory textbooks that discuss the Fourier transform and its properties present a table of two-sided Fourier transform pairs. You can use the frequency shifting property of the Fourier transform to obtain a two-sided (the DC component in the middle of the buffer) representation. If x(t) ¤ X(f) is a Fourier transform pair, then

$$x(t)e^{j2\pi f_o t} \Leftrightarrow X(f - f_0)$$ ¶

Let¶

$$\Delta t = \frac{1}{f_s} = 1 \cdots (f_s \text{ is·the·sampling·frequency})$$

in the discrete representation of the time signal, and set $\leq$ to the index corresponding to the Nyquist component f_Nyq

$$f_0 = f_{Nyq} = \frac{f_s}{2} = \frac{1}{2\Delta t},$$

because that is how much frequency shifting is required for the DC component to appear in the location of the Nyquist component.

The discrete representation is

$$x_i e^{ji\pi} \Leftrightarrow X_{k-\frac{n}{2}},$$

where n is the number of elements in the discrete sequence.

Expanding the exponential term in the time sequence

$$e^{ji\pi} = \cos(i\pi) + j\sin(i\pi) = \begin{cases} 1 & \text{if } i \text{ is even} \\ -1 & \text{if } i \text{ is odd} \end{cases}$$

which means that to obtain an FFT whose DC component appears in the center of the sequence, you need only negate the odd elements of the original sequence. Thus, if

$$X = \left\{x_0, x_1, x_2, x_3, \ldots, x_{n-1}, x_{n-1}\right\}$$

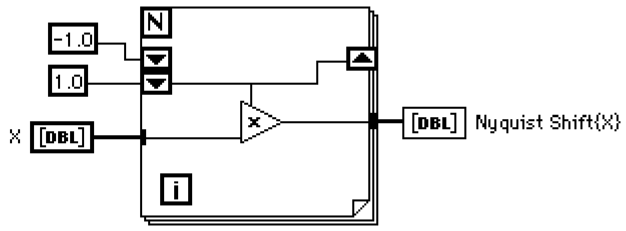
is the original input sequence, then the sequence

$$Y = \left\{x_0, x_1, x_2, x_3, \ldots, x_{n-1}, x_{n-1}\right\}$$

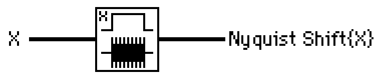
generates a two-sided FFT.

The following is a simple VI implementation. Notice that you can implement this calculation in place without extra buffers.

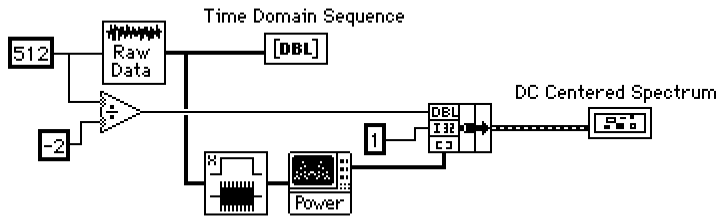
The following block diagram is one of many possible implementations. In this particular implementation, the For Loop includes a shift register with two left terminals. The constants wired to the shift register terminals initialize them with the values -1.0 and 1.0. When the For Loop executes, the first element of the array does not change because the element is multiplied by 1.0. The next iteration swaps the contents of the left terminals of the shift register and multiplies the second element of the array by -1.0, negating the original value. The For Loop repeats until it processes the whole array, at which time the contents of the output array are ready to generate a two-sided FFT.



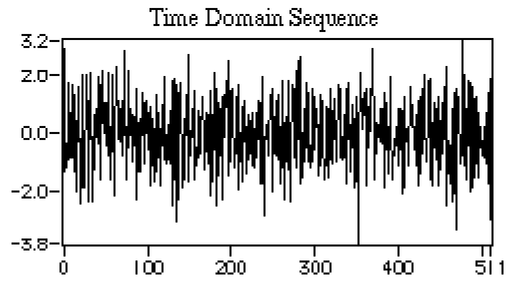
Finally, the following figure shows the input and output terminals of the Nyquist Shift VI. You can use this VI in an application that requires two-sided information.



The following block diagram shows a simple way to use the Nyquist Shift VI.



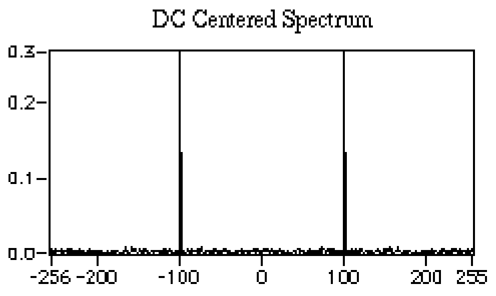
The upper branch of this block diagram graphs the raw data. The data is shown in the following graph.



In the previous block diagram, the Nyquist Shift VI preprocesses the raw data by negating every other element in it. The Power Spectrum VI then analyzes the data. To display the processed data correctly, you must supply the index corresponding to the harmonic in question. In this case, the initial value for this graph ≤ must be

$$x_0 = -\frac{n}{2}.$$

The following graph displays the result of processing the data in this manner to obtain two-sided FFTs. Notice that the DC component appears in the center of the display indicated by the zero index and that the overall format is just like that commonly found in tables of Fourier transform pairs.



You can apply this technique to sequences that generate real and complex FFTs, even-sized real and complex DFTs, power and cross power spectra, fast Hartley transforms (FHTs), and all functions related to these functions. You cannot apply this technique directly to odd-sized arrays because leakage will occur. However, you can derive a similar technique for odd-sized arrays.

Frequency Information Obtained from Transforms

The discrete implementation of the Fourier transform maps a digital signal into its Fourier series coefficients or harmonics. Unfortunately, neither a time nor a frequency stamp is directly associated with the FFT operation. Modern acquisition systems, whether they use add-on boards or instruments to capture data, allow you to control or specify the sampling interval ≤.

Because an acquired array of samples represents a progression of equally-spaced samples in time, you can determine the corresponding frequency in Hertz. The sampling frequency fs for ≤ is

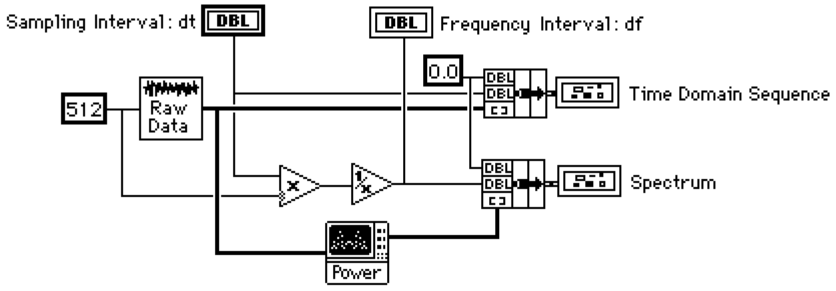
$$f_s = \frac{1}{\Delta t},$$

and the frequency interval is

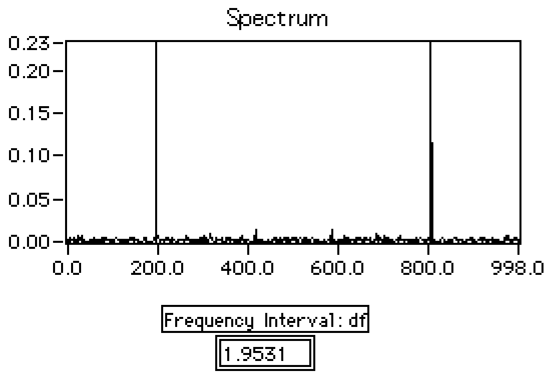
$$\Delta f = \frac{f_s}{n} = \frac{1}{n\Delta t}$$

where n is the number of samples in the sequence.

Given the sampling interval 1.000E-3, the following block diagram displays a graph with the correct frequency information.



Thus, for the signal $x(t)$, the resulting power spectrum graph with the correct frequency axis and the resulting frequency interval appear as in the following illustration.



The sampling interval is the smallest frequency that the system can resolve via FFT or related routines. A simple way to increase the resolution is to increase the number of samples or increase the sampling interval.

Parseval's Theorem

Parseval's Theorem states that the total energy computed in the time domain must equal the total energy computed in the frequency domain. It is a simple statement of conservation of energy. Parseval's relationship in its continuous form is given by

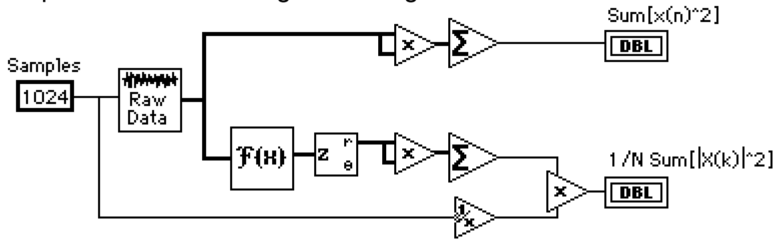
$$\int_{-\infty}^{\infty} x(t)x(t)\,dt = \int_{-\infty}^{\infty} |X(f)|^2\,df.$$

You can express the discrete version of Parseval's relationship as

$$\sum_{i=0}^{n-1} |x_i|^2 = \frac{1}{n}\sum_{k=0}^{n-1} |X_k|^2$$

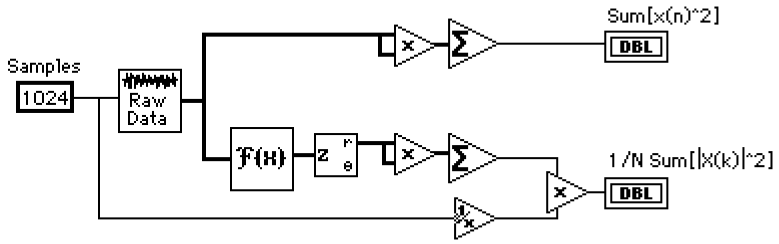
where $x_i \Leftrightarrow X_k$ is a discrete FFT pair, and n is the number of elements in the sequence.

You can implement Parseval's relationship using the Real FFT VI to compute the FFT of a real input sequence. The following block diagram demonstrates Parsevals theorem.



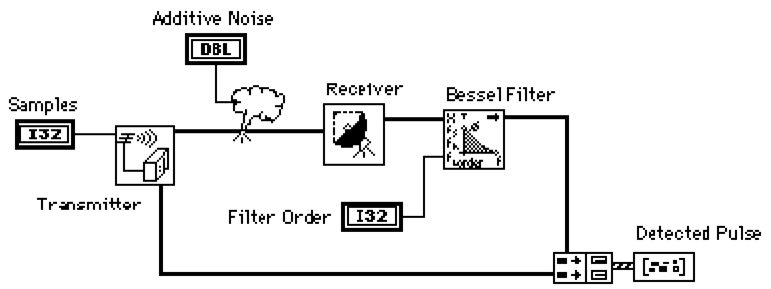
The upper branch in the block diagram evaluates the left side of Parsevals relationship, and the lower branch evaluates the right side.

Applying Parseval's relationship to the time signal and the corresponding FFT, the total computed energy in the time domain signal is the same total computed energy in the frequency domain, as shown in the following figure.



Pulse Demo

The Pulse Demo VI was one of the first examples created using LabVIEW, and it still provides a good example of the high-level capabilities of block diagram programming. The VI simulates the transmission of a pulse in a noisy environment based upon basic communication theory. Consider the following block diagram and a signal x(t), which represents the pulse in question.



The Transmitter VI generates the signal x(t) and modulates it to create a signal of the form

$$x_m(t) = x(t)\cos(2\pi f_c t)$$

where ≤ is the carrier frequency.

The Noise VI, whose icon is a lightning bolt hitting the wire, then corrupts the transmitted signal with a noise signal. You can control the amount of noise through the front panel of the Noise VI. The signal produced by this VI is

$$x_n(t) = x_m(t) + n(t) = x(t)\cos(2\pi f_c t) + n(t),$$

where n(t) is an uncorrelated noise signal.

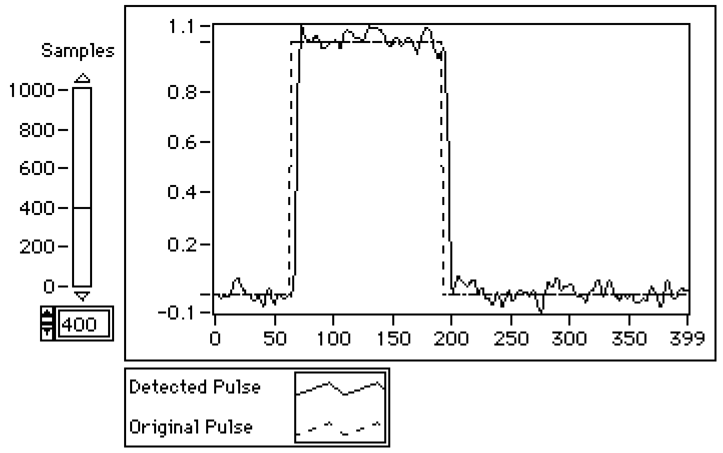
The Receiver VI, whose icon is a parabolic antenna, receives and demodulates the noisy pulse. Because the carrier frequency is known, you can demodulate the signal through a heterodyning procedure. Expanding the following terms,

$$\begin{aligned} y(t) &= 2x_n(t)\cos(2\pi f_c t) + n(t) \\ &= 2[x(t)\cos(2\pi f_c t) + n(t)]\cos(2\pi f_c t) \\ &= 2x(t)\cos^2(2\pi f_c t) + n(t)\cos(2\pi f_c t) \\ &= x(t) + x(t)\cos(4\pi f_c t) + 2n(t)\cos(2\pi f_c t) \end{aligned}$$

The signal y(t) contains the original signal x(t), which contains the pulse information plus two modulated signals in the upper portion of the spectrum, one at fc and the other one at 2fc. To extract x(t) from y(t), you can use a low pass filter with a cutoff frequency less than the carrier frequency:

$$x(t) \quad \text{Lowpass}[y(t)].$$

The following graph displays the results.



The block diagram describing this process is much like the block diagram a researcher, scientist, or engineer would use to explain and document the process. It is simple, elegant, and self-documenting.

Statistical Examples

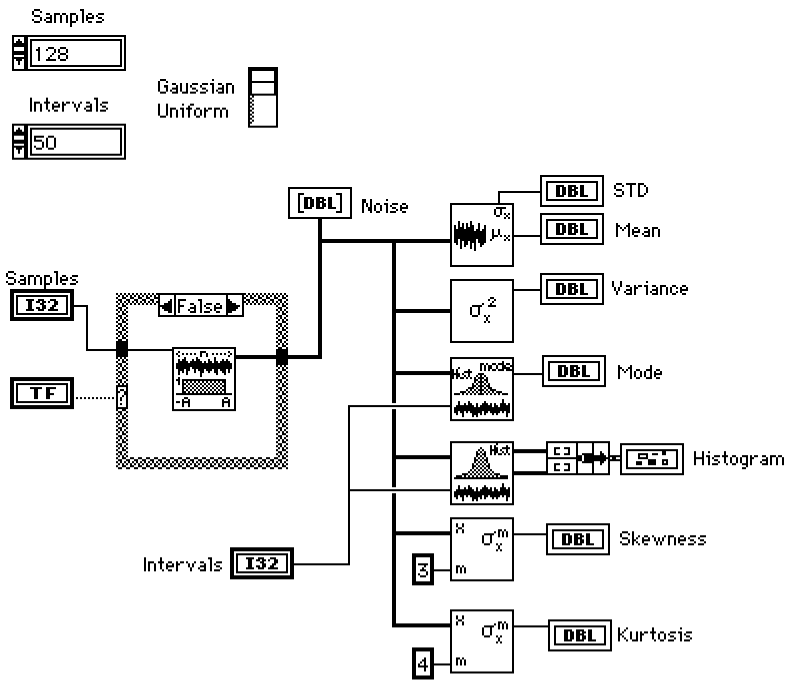
Probability Density and Distribution Functions

Statistical Analysis of Simulated Noise Signals

When developing analysis algorithms, you may need to generate and simulate noise signals with specific statistical properties to observe the behavior of the algorithm under deviant conditions. The robustness of the analysis algorithm and its successful implementation depends upon the development of a good model for noise signals.

LabVIEW contains an efficient random number generator you can use to model simple probabilistic events. In addition, a subset of the analysis VIs can generate random patterns with specific statistical characteristics. The Uniform White Noise and the Gaussian White Noise VIs, for example, generate two common random patterns for modeling noisy environments.

The VI whose front panel and block diagram appear below generates a noisy signal whose expected statistical properties are known beforehand. The VI then uses descriptive statistical analysis VIs to verify the model and the properties of the noise signal.



The desired statistical properties in this example are the mean, standard deviation, skewness, kurtosis, histogram, and mode. The mean and standard deviation are well-known statistics. Skewness is a measure of symmetry, and kurtosis is a measure of peakedness, and they correspond to the third and fourth order moments about the mean, respectively. The histogram is an indication of the distribution, and the mode is the value that occurs most often.
The expected values for a (0:1) Gaussian-distributed, white-noise signal are as follows.

E{μ} = 0.0

E{σ} = 1.0

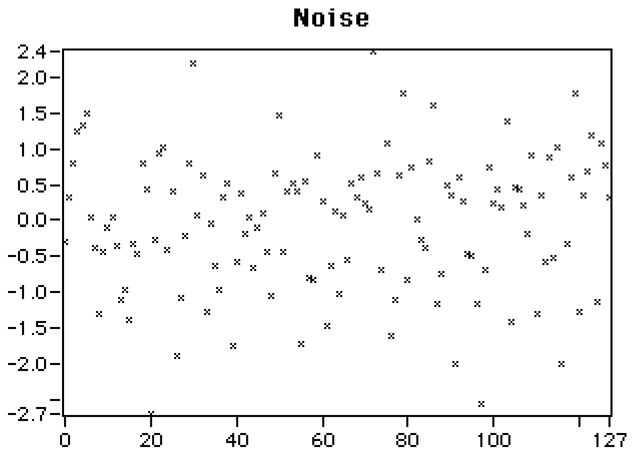
E{σ3} = 0.0

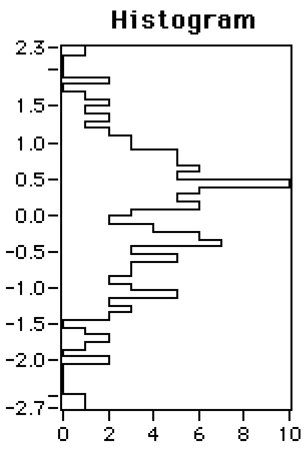
E{σ4} = 3.0

E{mode} = 0.0

The histogram should resemble a bell-shaped curve.

The following graphs show the result of generating a (0:1) Gaussian-distributed noise signal, as well as its histogram, which resembles a bell-shaped curve corresponding to a Gaussian distribution.





The following controls show the computed values for the mean, standard deviation, skewness, kurtosis, and mode to be compared to the actual expected values.

Mean	STD	Variance	Mode	Skewness	Kurtosis
-0.04	0.95	0.91	0.39	-0.20	2.48

Generating random numbers using a digital computer is impossible because you cannot create a true random sequence with a deterministic machine. However, the analysis VIs generate finite length sequences (at least 290 samples) that closely mimic noise signals.

Probability Density and Distribution Functions

The previous example extracted some statistical properties of random patterns using descriptive statistics VIs. The example also mentioned that the histogram is an indication of the probability density function.

The probability distribution function F(x) is defined as

$$F(x) = \int_{-\infty}^{x} f(t)\,dt,$$

where $f(x)$ is the probability density function, and the following conditions on the probability density function have been imposed:

$$f(x)) \geq 0 \quad \forall x \varepsilon \text{ domain of } f,$$

and

$$\int_{-\infty}^{\infty} f(x)dx = 1.$$

It follows from calculus theorems that

$$\frac{dF(x)}{dx} = f(x)$$

To obtain the probability density and distribution functions of the white noise pattern generator VIs, you can use the Histogram VI because it is a denormalized discrete representation of the probability density function. The discrete representation is

$$\sum_{1=0}^{n-1} x_i \Delta x = 1 \ ,$$

and the sum of the elements of the histogram is of the form

$$\sum_{1=0}^{n-1} h_1 = n \ ,$$

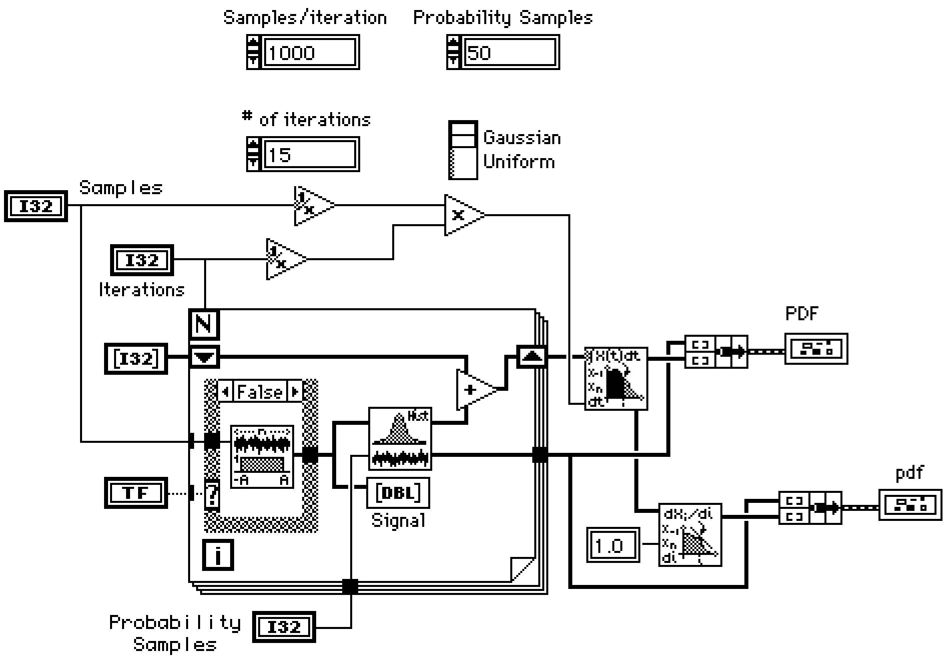
where m is the number of samples in the histogram, and n is the number of samples in the input sequence representing the function.

Thus, to obtain an estimate of the probability distribution function, it is only necessary to normalize the histogram by

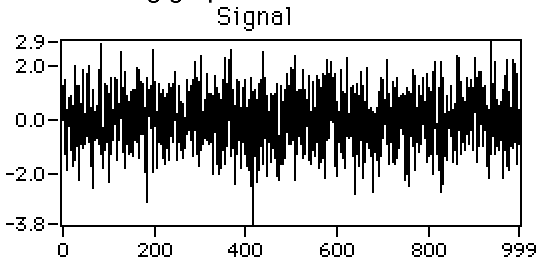
$$\Delta x = \frac{1}{n}$$

factor and letting $h_j = x_j$.

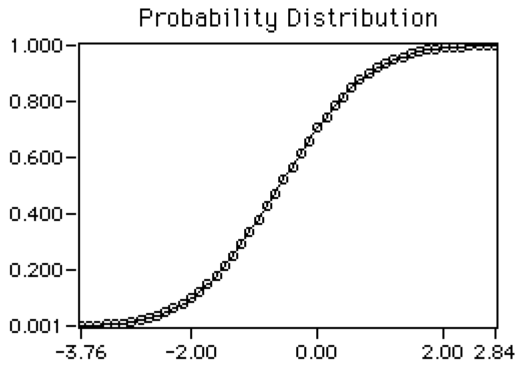
Consider the following set of front panel controls and the block diagram below them, which uses 25,000 samples (2,500 in each of 10 loop iterations) to generate the probability distribution function. The output array of the Integral x(t) VI is the probability distribution function, and the differentiation of the distribution is the probability density function.



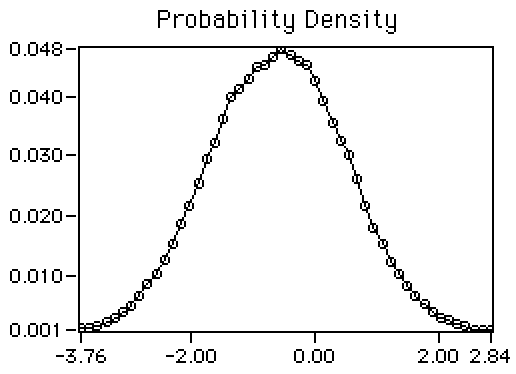
The following graph shows the last block of Gaussian-distributed noise samples.



The following graphs show the results of executing the previous block diagram. Notice that the curve corresponding to the probability distribution function is monotonically increasing and is limited to the maximum value of 1.00 as the value of the X axis increases.



Also notice that the probability density function shows a Gaussian distribution that conforms to the specific pattern selected when the VI generated the noise signal.



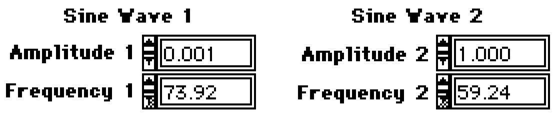
Windowing Example

Minimizing Leakage Using Smoothing Windows

The Sampling Theorem states that you can completely reconstruct a continuous-time signal from discrete equally spaced samples if the highest frequency in the time signal is less than half the sampling frequency. Half the sampling frequency equals the Nyquist frequency. This theorem bridges the gap between continuous time signals and digital time signals. When you digitize a time signal in practical applications, side effects occur even when the data meets the Nyquist criterion. One of the most common side effects is energy leakage caused by the finite observation window.

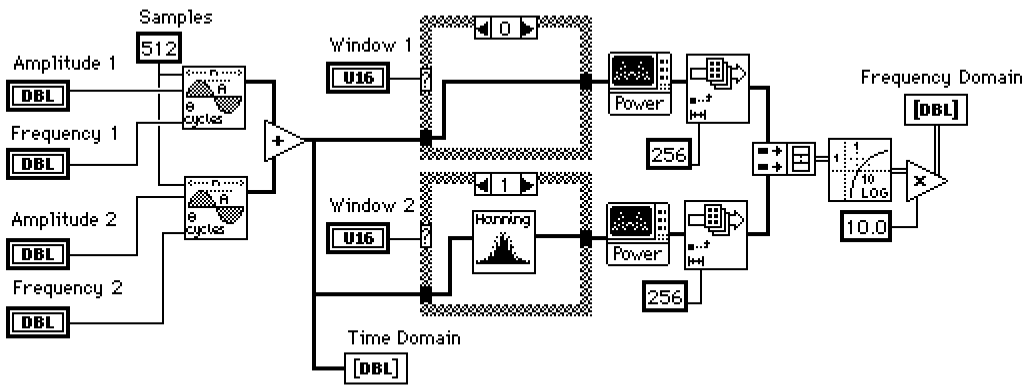
When you use the FFT or DFT to measure the frequency content of your data, these transforms assume that the finite window of data is one period of a periodic signal. The observation window, then, can cause sharp transition changes to be introduced into your measured data.

You can minimize the effects of these transition edges by applying smoothing windows. You can think of these windows, which modify the spectral contents of the digitized waveform, as simple filtering operations. The type of window you should use depends upon your application requirements. The following example demonstrates the windowed and non-windowed spectrums of a signal composed of the sum of two sinusoids. The two sinusoids have amplitudes and frequencies (measured in cycles) as shown.

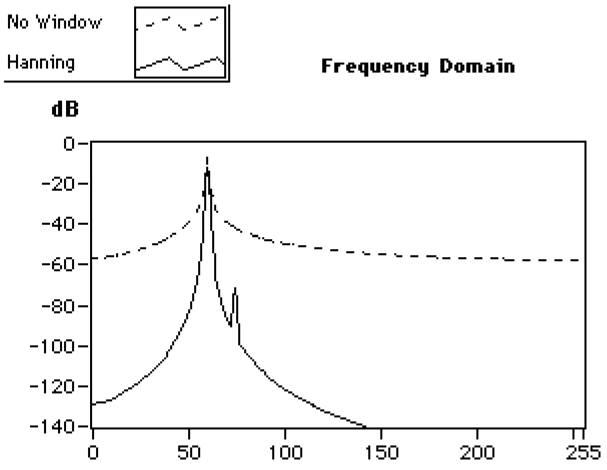


Sine Wave 1		Sine Wave 2	
Amplitude 1	0.001	Amplitude 2	1.000
Frequency 1	73.92	Frequency 2	59.24

The block diagram for this example demonstrates how to use smoothing windows to reduce spectral leakage:



The following graph displays the results. The dashed line represents the spectrum of the digitized signal with no window applied, and the solid line represents the windowed spectrum. Notice how the nonwindowed spectrum shows leakage that is more than 20 dB greater than the smaller sinusoid.

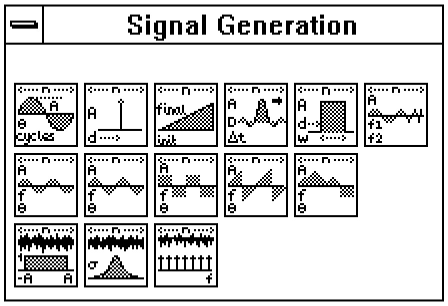


You can apply more sophisticated techniques to get a more accurate description of the original time-continuous signal in the frequency domain. However, in most applications, applying a smoothing window is sufficient to obtain a better frequency representation of the signal.

Signal Generation VIs

This topic describes the VIs that generate one-dimensional arrays with specific waveform patterns. For information about Signal Generation VIs that use normalized input frequencies, see Normalized Frequency.

The following illustration shows the options that are available on the **Signal Generation** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

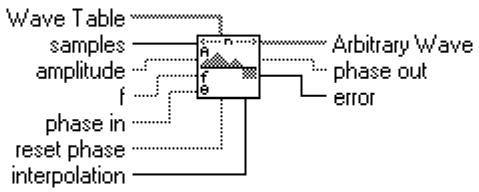


Arbitrary Wave
Chirp Pattern
Gaussian White Noise
Impulse Pattern
Periodic Random Noise
Pulse Pattern
Ramp Pattern
Sawtooth Wave
Sinc Pattern
Sine Pattern
Sine Wave VI
Square Wave
Triangle Wave
Uniform White Noise

For examples of how to use the signal generation VIs, see the examples located in `examples\analysis\sigxmpl.llb`.

Arbitrary Wave (Advanced Only)

Generates an array containing an arbitrary wave.



≤ **Wave Table** is one cycle of the waveform used in creating the output **Arbitrary Wave**.
≤ **samples** is the number of samples of the Arbitrary Wave. samples must be greater than or equal to 0. If **samples** is less than zero, the VI sets **Arbitrary Wave** to an empty array and returns an error. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Arbitrary Wave**. **amplitude** defaults to 1.0.
≤ **f** is the frequency of **Arbitrary Wave** in normalized units of cycles/sample. **f** defaults to 1 cycle/128 samples, or 7.8125E-3 cycles/sample.

≤ **phase in** is the initial phase, in degrees, of **Arbitrary Wave** when **reset phase** is true.
≤ **reset phase** determines the initial phase of **Arbitrary Wave**. If **reset phase** is true, the initial phase is set to **phase in**. If **reset phase** is false, the initial phase is set to the value of **phase out** when the VI last executed. **reset phase** defaults to true.
≤ **interpolation** determines the type of interpolation the VI uses to generate **Arbitrary Wave** from the **Wave Table** array. If **interpolation** is 0, the VI does not use interpolation. If **interpolation** is 1, the VI uses linear interpolation. **interpolation** defaults to 0 (no interpolation).
≤ **Arbitrary Wave** is the output arbitrary wave.
≤ **phase out** is the phase, in degrees, of the next sample of **Arbitrary Wave**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Arbitrary Wave**, the VI generates the pattern according to the following formula:

$$y[i] = a * \mathrm{arb}(\mathrm{phase}[i]), \qquad \text{for } i = 0, 1, 2, ..., n-1,$$

where a is the **amplitude**, n is the number of **samples**,

$$\mathrm{arb}(\mathrm{phase}[i]) = \mathrm{WT}(\mathrm{phase}[i] \text{ modulo } 360)*m/360)$$

where m is the size of the Wave Table array

If **interpolation** = 0 (no interpolation), then $WT(\mathrm{x})$ = **Wave Table**[int(x)].

If **interpolation** = 1 (linear interpolation), then *WT*(*x*) is equal to the linearly interpolated value of **Wave Table**[int(x)] and **Wave Table**[(int(x)+1) modulo *m*].

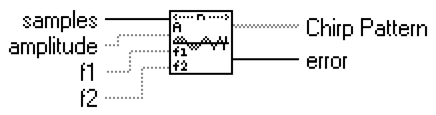
phase[i] = initial_phase + **f***360.0**i*, where **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from an arbitrary wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of this VI produce the output **Arbitrary Wave** array containing the next **samples** of the arbitrary wave.

phase out is set to phase[n], and this reentrant VI uses this value as its new **phase in** if **reset phase** is false the next time the VI executes.

Chirp Pattern (Advanced Only)

Generates an array containing a chirp pattern.



≤ **samples** is the number of samples of the **Chirp Pattern**. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Chirp Pattern**. **amplitude** defaults to 1.0.
≤ **f1** is the beginning frequency of **Chirp Pattern** in normalized units of cycles/sample.
≤ **f2** is the ending frequency of **Chirp Pattern** in normalized units of cycles/sample.
≤ **Chirp Pattern**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Chirp Pattern**, the VI generates the pattern according to the following formula:

$$y_i = a^* \sin\left(\left(a/2i + b\right)i\right)$$

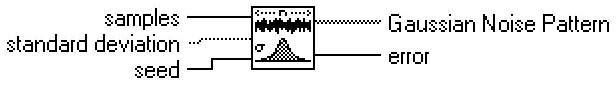
for i = 0, 1, 2, ..., n-1,

where a is the **amplitude**, a = 2π (**f2**-**f1**)/n, $b = 2$

π**f1**, **f1** is the beginning frequency in normalized units of cycles/sample, **f2** is the ending frequency in normalized units of cycles/sample, and n is the number of **samples**.

Gaussian White Noise (Advanced Only)

Generates a Gaussian-distributed, pseudorandom pattern whose statistical profile is $(\mu, \sigma) = (0, s)$, where s is the absolute value of the specified **standard deviation**.



≤ **samples** is the number of samples of the **Gaussian Noise Pattern**. **samples** must be greater than or equal to 0. If **samples** is less than zero, the VI sets **Gaussian Noise Pattern** to an empty array but does not return an error. **samples** defaults to 128.
≤ **standacrd deviation** defaults to 1.0.
≤ **seed**. If **seed** is a prime number, the VI generates a much longer random sequence.
≤ **Gaussian Noise Pattern**. The largest **Gaussian Noise Pattern** that the VI can generate depends upon the amount of memory in your system and is theoretically limited to

2,147,483,647 $(2^{31} - 1)$ elements.
≤ **error**. See Analysis Error Codes for a description of the error.
The VI generates the Gaussian-distributed pseudorandom sequence using a modified version of the Very-Long-Cycle random number generator algorithm based upon the Central Limit Theorem. Given that the probability density function, f(x), of the Gaussian-distributed **Gaussian Noise Pattern** is:

$$f(x) = \frac{1}{\sqrt{2ps}} e^{\left(\frac{1}{2}\right)\left(\frac{x}{s}\right)^2}$$

where *s* is the absolute value of the specified **standard deviation** and that you can compute the expected values, $E\{\bullet\}$, , using the formula:

$$E(x) = \int_{-\infty}^{\infty} x(f(x))dx$$

then the expected mean value, ≤, and the expected standard deviation value,

σ, of the pseudorandom sequence are:

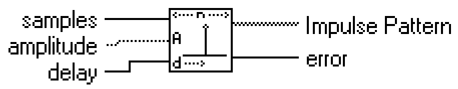
$$\mu = E\{x\} = 0$$

$$\sigma = \left[E\left\{(x - \mu)^2\right\}\right]^{1/2} = s.$$

The pseudorandom sequence produces approximately 2^{90} samples before the pattern repeats itself.

Impulse Pattern (Advanced Only)

Generates an array containing an impulse pattern.



≤ **samples** is the number of samples of the **Impulse Pattern**. **samples** must be greater than **delay**. If **samples** is negative or zero, the VI sets **Impulse Pattern** to an empty array and returns an error. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Impulse Pattern**. **amplitude** defaults to 1.0.
≤ **delay** must be greater than or equal to 0. If **delay** is less than zero, or greater than or equal to the number of **samples**, the VI sets **Impulse Pattern** to zero and returns an error.
≤ **Impulse Pattern.** The largest **Impulse Pattern** the VI can generate depends on the amount of memory in your system and is theoretically limited to
≤ elements.
≤ **error**. See Analysis Error Codes for a description of the error.
If the **Impulse Pattern** is represented by the sequence X, the VI generates the pattern according to the following formula:

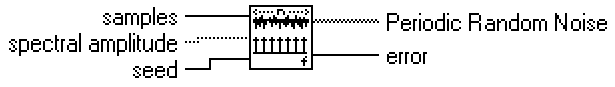
$$x_i = \begin{cases} \text{amp if } i = d \\ 0 \quad \text{elsewhere} \end{cases}$$

for i = 0, 1, 2, ..., n-1

where a is the **amplitude**, d is the **delay**, and n is the number of **samples**.

Periodic Random Noise (Advanced Only)

Generates an array containing periodic random noise (PRN).



≤ **samples** is the number of samples of the **Periodic Random Noise**. **samples** defaults to 128.
≤ **spectral amplitude** is the magnitude of the frequency domain components of the periodic random noise.
≤ **seed** is the seed value used in generating the random phase of the periodic random noise. If **seed** < 0, the random phase generator is not reseeded. The default value for **seed** is -1.
≤ **Periodic Random Noise** is the output array containing periodic random noise.
≤ **error**. See Analysis Error Codes for a description of the error.
The output array contains all frequencies which can be represented with an integral number of cycles in the requested number of **samples**. Each frequency-domain component has a magnitude of **spectral amplitude** and random phase.

Another way of thinking of the output array of PRN, is that it is a summation of sinusoidal signals with the same amplitudes but with random phases. The unit of spectral amplitude is the same as the output Periodic Random Noise, and is a linear measure of amplitude, similar to other signal generation VIs.

The VI generates the same periodic random sequence for a given positive **seed** value. The VI does not reseed the random phase generator if **seed** is negative.

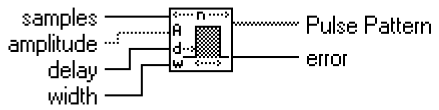
The output sequence is bounded by an amplitude of **spectral amplitude** $* \frac{\text{samples}}{2}$.

You can use PRN to compute the frequency response of a linear system in one time record instead of averaging the frequency response over several time records, as you must for nonperiodic random noise sources.

You do not need to window PRN before performing spectral analysis because PRN is self-windowing and therefore has no spectral leakage. This is because PRN contains only integral-cycle sinusoids.

Pulse Pattern (Advanced Only)

Generates an array containing a pulse pattern.



≤ **samples** is the number of samples of the **Pulse Pattern**. **samples** must be greater than or equal to **delay** + **width**. If **samples** is less than zero, the VI sets **Pulse Pattern** to an empty array and returns an error. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Pulse Pattern**. **amplitude** defaults to 1.0.
≤ **delay** must be greater than or equal to 0. **delay** defaults to 0.
≤ **width** must be greater than or equal to 0. **width** defaults to 1.
≤ **Pulse Pattern**. The largest **Pulse Pattern** the VI can generate depends on the amount of memory in your system and it is theoretically limited to
≤ elements.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence X represents **Pulse Pattern**, the VI generates the pattern according to the following formula:

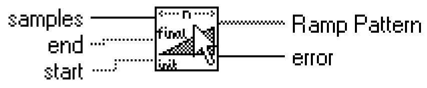
$$x_i = \begin{cases} a & \text{if } d \le i < (d + w) \\ 0.0 & \text{elsewhere} \end{cases}$$

for i= 0, 1, 2, ..., n-1.

where a is the **amplitude**, d is the **delay**, w is the **delay**, and n is the number of **samples**.

Ramp Pattern (Advanced Only)

Generates an array containing a ramp pattern.



≤ **samples** is the number of samples of the **Ramp Pattern**. If **samples** is less than two, the VI sets **Ramp Pattern** to an empty array and returns an error. **samples** defaults to 128.
≤ **end** is the ending value, or final value of **Ramp Pattern**.
≤ **start** is the starting value, or first value of **Ramp Pattern**.
≤ **Ramp Pattern**. The largest **Ramp Pattern** the VI can generate depends on the amount of memory in your system and it is theoretically limited to
≤ elements.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence X represents **Ramp Pattern**, the VI generates the pattern according to the formula:

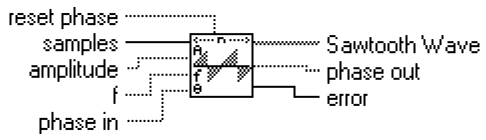
$$x_i = x_0 + iDx \,, \qquad \text{for } i = 0,\ 1,\ 2,\ \ldots,\ n-1$$

where $\Delta x = \dfrac{x_{n-1} - x_0}{n-1}$, x_{n-1} is the **end**,

≤ is the **start**, and n is the number of samples.
The Ramp Pattern VI does not impose conditions on the relationship between **start** and **end**. The VI can therefore generate ramp-up and ramp-down patterns.

Sawtooth Wave (Advanced Only)

Generates an array containing a sawtooth wave.



≤ **reset phase** determines the initial phase of **Sawtooth Wave**. If **reset phase** is true, the initial phase is set to **phase in**. If **reset phase** is false, the initial phase is set to the value of **phase out** when the VI last executed. **reset phase** defaults to true.
≤ **samples** is the number of samples of the **Sawtooth Wave**. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Sawtooth Wave**. **amplitude** defaults to 1.0.
≤ **f** is the frequency of **Sawtooth Wave** in normalized units of cycles/sample. **f** defaults to 1 cycle/128 samples, or 7.8125E-3 cycles/sample.
≤ **phase in** is the initial phase, in degrees, of **Sawtooth Wave** when **reset phase** is true.
≤ **Sawtooth Wave** is the output sawtooth wave.
≤ **phase out** is the phase, in degrees, of the next sample of **Sawtooth Wave**.
≤ **error**. See Analysis Error Codes for a description of the error.

If the sequence Y represents **Sawtooth Wave**, the VI generates the pattern according to the following formula:

$$y[i] = a * \text{sawtooth}(\text{phase}[i]), \text{ for } i = 0, 1, 2, ..., n-1,$$

where a is the **amplitude**, n is the number of **samples**,

$$\text{sawtooth}(\text{phase}[i]) = \begin{cases} \dfrac{p}{180.0} & 180 \le p < 180 \\ \dfrac{p}{180.0} & 0 \le p < 360 \end{cases}$$

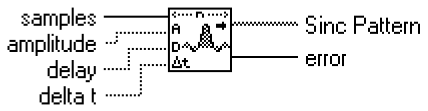
p = phase [i] modulo 360.0, phase[i] = initial_phase + **f***360.0* i, **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a sawtooth wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Sawtooth Wave** array containing the next **samples** of a sawtooth wave.

phase out is set to phase[n], and this reentrant VI uses this value as its new **phase in** if **reset phase** is false the next time the VI executes.

Sinc Pattern (Advanced Only)

Generates an array containing a sinc pattern.



≤ **samples** is the number of samples of the **Sinc Pattern**. **samples** must be greater than or equal to 0. If **samples** is less than zero, the VI sets **Sinc Pattern** to an empty array and returns an error. **samples** defaults to 128.
≤ **amplitude** is the amplitude of Sinc **Pattern**. **amplitude** defaults to 1.0.

≤ **delay** shifts the peak value within the **Sinc Pattern** as the VI generates the pattern. LabVIEW determines this condition from the preceding formula and shifts the peak value when $i\Delta t = d$. **delay** defaults to 0.0.

≤ **delta t** is the sampling interval. It is a floating-point number inversely proportional to the width of the main sinc lobe. That is, the smaller the sampling interval, the wider the main lobe; the larger the sampling interval, the smaller the main lobe. Notice that when **delta t** is 1, and d is an integer value, the VI sets **Sinc Pattern** to zero except at the point where i = d. At this point, the value is equal to **amplitude**. The recommended range of values for the sampling interval is 0 < **delta t** # 1. **delta t** must be greater than 0.0. If **delta t** is less than or equal to zero, the VI sets **Sinc Pattern** to an empty array and returns an error. **delta t** defaults to 1.0.

≤ **Sinc Pattern.** The largest **Sinc Pattern** the VI can generate depends on the amount of memory in your system and is theoretically limited to
≤ elements.

≤ **error**. See Analysis Error Codes for a description of the error.

If the sequence Y represents **Sinc Pattern**, the VI generates the pattern according to the following formula:

$$y_i = a\,\mathrm{sinc}(i\Delta t - d), \quad \text{for } i = 0, 1, 2, \ldots, n-1,$$

where $\mathrm{sinc}(x) = \dfrac{\sin(\pi x)}{\pi x}$, a is the **amplitude**,

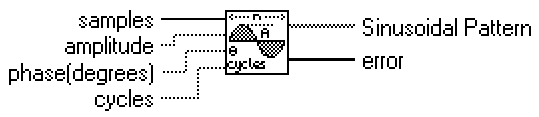
≤ is the sampling interval **delta t**, d is the **delay**, and n is the number of **samples**.

The main lobe of the sinc function, sinc(*x*), is the part of the sinc curve bounded by the region -1 # *x* # 1.

When $|x| = 1$, the sinc(x) = 0.0, and the peak value of the sinc function occurs when x = 0. Using l'Hopital's Rule, you can show that sinc(0) = 1 and that it is also its peak value. Thus, the main lobe is the region of the sinc curve encompassed by the first set of zeros to the left and the right of the sinc value.

Sine Pattern (Advanced Only)

Generates an array containing a sinusoidal pattern.

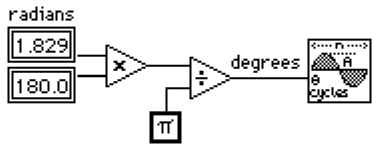


≤ **samples** is the number of samples of the **Sinusoidal Pattern**. **samples** must be greater than or equal to 0. If **samples** is less than zero, the VI sets **Sinusoidal Pattern** to an empty array and returns an error. **samples** defaults to 128.

≤ **amplitude** is the amplitude of **Sinusoidal Pattern**. **amplitud**e defaults to 1.0.

≤ **phase** defaults to 0.0.

Note: phase must be in degrees rather than radians. If phase is in radians, make sure you convert it to degrees, as shown in the following figure, before using the Sinusoidal Pattern VI.



≤ **cycles** defaults to 1.0.

Note: Because cycles is a floating-point number, fractional cycles are possible for the Sinusoidal Pattern. Furthermore, setting cycles to a negative number does not generate an error condition because it is mathematically correct and useful to consider negative frequencies in Fourier and spectral analysis.

≤ **Sinusoidal Pattern**. The largest **Sinusoidal Pattern** the VI can generate depends on the amount of memory in your system and is theoretically limited to
≤ elements.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Sinusoidal Pattern**, the VI generates the pattern according to the following formula:

$$y_i = a\sin(x_i) \qquad \text{for } i = 0, 1, 2, \ldots, n-1,$$

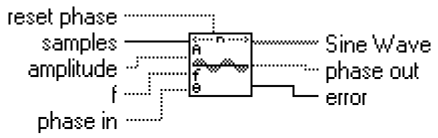
where

$$x_i = \frac{2\pi ik}{n} + \frac{\pi\phi_0}{180}$$

, a is the amplitude, k is the number of **cycles** in the pattern, ϕ_0 is the initial **phase** (in degrees), and n is the number of **samples**.

Sine Wave (Advanced Only)

Generates an array containing a sine wave.



≤ **reset phase** determines the initial phase of **Sine Wave**. If **reset phase** is true, the initial phase is set to **phase in**. If **reset phase** is false, the initial phase is set to the value of **phase out** when the VI last executed. **reset phase** defaults to true.
≤ **samples** is the number of samples of the **Sine Wave**. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Sine Wave**. **amplitude** defaults to 1.0.
≤ **f** is the frequency of **Sine Wave** in normalized units of cycles/sample. **f** defaults to 1 cycle/128 samples, or 7.8125E-3 cycles/sample.
≤ **phase in** is the initial phase, in degrees, of **Sine Wave** when **reset phase** is true.
≤ **Sine Wave** is the output sine wave.
≤ **phase out** is the phase, in degrees, of the next sample of **Sine Wave**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Sine Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * \sin(\text{phase}[i]), \qquad \text{for } i = 0, 1, 2, \ldots, n-1,$$

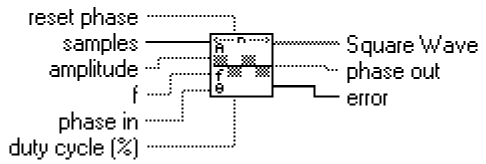
where a is the **amplitude** and phase[i] = initial_phase + **f***360.0*i, **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a sine wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Sine Wave** array containing the next **samples** of a sine wave.

phase out is set to phase[n], and this reentrant VI uses this value as the new **phase in** if **reset phase** is false the next time the VI executes.

Square Wave (Advanced Only)

Generates an array containing a square wave.



≤ **reset phase** determines the initial phase of **Square Wave**. If **reset phase** is true, the initial phase is set to **phase in**. If **reset phase i**s false, the initial phase is set to the value of **phase out** when the VI last executed. **reset phase** defaults to true.
≤ **samples** is the number of samples of the **Square Wave**. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Square Wave**. **amplitude** defaults to 1.0.
≤ **f** is the frequency of **Square Wave** in normalized units of cycles/sample. **f** defaults to 1 cycle/128 samples, or 7.8125E-3 cycles/sample.
≤ **phase in** is the initial phase, in degrees, of **Square Wave** when **reset phase** is true.
≤ **duty cycle** is the duty cycle, in percent, of the **Square Wave**.
≤ **Square Wave** is the output square wave.
≤ **phase out** is the phase, in degrees, of the next sample of **Square Wave**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Square Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * \text{square}(\text{phase}[i]), \quad \text{for } i = 0, 1, 2, \ldots, n-1$$

where a is the **amplitude**, n is the number of **samples**,

$$\text{square}(\text{phase}[i]) = \begin{cases} 1.0 & 0 \le p < \left(\frac{\text{duty}}{100}360\right) \\ -1.0 & \left(\frac{\text{duty}}{100}360\right) \le p < 360 \end{cases},$$

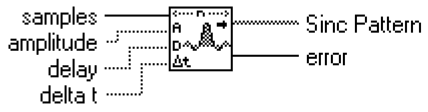
where p = phase[i] modulo 360.0, $duty$ = **duty cycle**,
phase[i] = initial_phase + **f***360.0*i, **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a square wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of this VI produce the output **Square Wave** array containing the next **samples** of a square wave.

phase out is set to phase[n], and this reentrant VI uses this value as its new **phase in** if **reset phase** is false the next time the VI executes.

Triangle Wave (Advanced Only)

Generates an array containing a triangle wave.



≤ **reset phase** determines the initial phase of **Triangle Wave**. If **reset phase** is true, the initial phase is set to **phase in**. If **reset phase** is false, the initial phase is set to the value of **phase out** when the VI last executed. **reset phase** defaults to true.
≤ **samples** is the number of samples of the **Triangle Wave**. **samples** defaults to 128.

≤ **amplitude** is the amplitude of **Triangle Wave**. **amplitude** defaults to 1.0.
≤ **f** is the frequency of **Triangle Wave** in normalized units of cycles/sample. **f** defaults to 1 cycle/128 samples, or 7.8125E-3 cycles/sample.
≤ **phase in** is the initial phase, in degrees, of **Triangle Wave** when **reset phase** is true.
≤ **Triangle Wave** is the output triangle wave.
≤ **phase out** is the phase, in degrees, of the next sample of **Triangle Wave**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the sequence Y represents **Triangle Wave**, the VI generates the pattern according to the following formula:

$$y_i = a * (\text{phase}[i]) \quad \text{for } i = 0, 1, 2, \ldots, n-1,$$

where a is the **amplitude**, n is the number of **samples**,

$$\text{tri}(\text{phase}[i]) = \begin{cases} \dfrac{p}{90} & 0 \le p < 90 \\ 2 - \dfrac{p}{90} & 90 \le p < 270 \\ \dfrac{p}{90} + 4 & 270 \le p < 360 \end{cases}$$

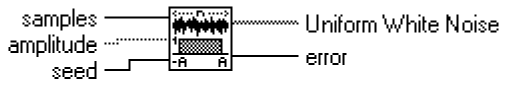
where p = (phase[i] modulo 360.0), phase[i] = initial_phase + **f***360.0*i, **f** is the frequency in normalized units of cycles/sample, initial_phase is **phase in** if **reset phase** is true, or initial_phase is the **phase out** from the previous execution of this instance of the VI if **reset phase** is false.

The VI is reentrant, so you can use it to simulate a continuous acquisition from a triangle wave function generator. If the input control **reset phase** is false, subsequent calls to a specific instance of the VI produce the output **Triangle Wave** array containing the next **samples** of a triangle wave.

phase out is set to phase[n], and this reentrant VI uses this value as its new **phase in** if **reset phase** is false the next time the VI executes.

Uniform White Noise (Advanced Only)

Generates a uniformly distributed, pseudorandom pattern whose values are in the range [-a:a], where a is the absolute value of **amplitude**.



≤ **samples** is the number of the samples of the **Uniform White Noise**. samples must be greater than or equal to 0. If **samples** is less than zero, the VI sets **Uniform White Noise** to an empty array but does *not* return an error. **samples** defaults to 128.
≤ **amplitude** is the amplitude of **Uniform White Noise**. **amplitude** defaults to 1.0.
≤ **seed**. If **seed** is a prime number, the VI generates a much longer random sequence.
≤ **Uniform White Noise**. The largest **Uniform White Noise** that the VI can generate depends upon the amount of memory in your system and is theoretically limited to
≤ elements.
≤ **error**. See Analysis Error Codes for a description of the error.

The VI generates the pseudorandom sequence using a modified version of the Very-Long-Cycle random number generator algorithm. Given that the probability density function, f(x), of the uniformly distributed **Uniform White Noise** is

$$f(x) = \begin{cases} \frac{1}{2a} & \text{if } -a \leq x \leq a \\ 0 & \text{elsewhere} \end{cases},$$

where a is the absolute value of the specified **amplitude**, and that you can compute the expected values, ≲, using the formula:

$$E(x) = \int_{-\infty}^{\infty} x(f(x))dx,$$

then the expected mean value, ≲, and the expected standard deviation value,

≲, of the pseudorandom sequence are:

$$\mu = E\{x\} = 0,$$

$$\sigma = \left[E\left\{(x-\mu)^2\right\}\right]^{1/2} = \frac{a}{\sqrt{3}} \approx 0.57735a$$

The pseudorandom sequence produces approximately ≲ samples before the pattern repeats itself.

Arbitrary Wave.vi

Arbitrary Wave

Chirp Pattern.vi

Chirp Pattern

Gaussian White Noise.vi

Gaussian White Noise

Impulse Pattern.vi

Impulse Pattern

Periodic Random Noise.vi

Periodic Random Noise

Pulse Pattern.vi

Pulse Pattern

Ramp Pattern.vi

Ramp Pattern

Sawtooth Wave.vi

Sawtooth Wave

Sinc Pattern.vi

Sinc Pattern

Sine Pattern.vi

Sine Pattern

Sine Wave.vi

Sine Wave

Square Wave.vi

Square Wave

Triangle Wave.vi

Triangle Wave

Uniform White Noise.vi

Uniform White Noise

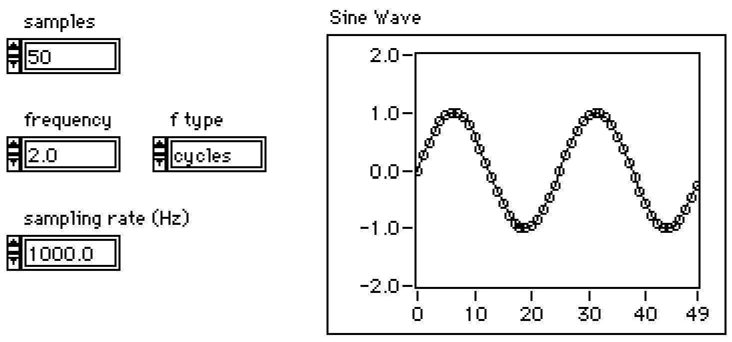
Normalized Frequency

Some of the Signal Generation VIs use an input frequency control that is assumed to use normalized frequency units of cycles per sample. This frequency ranges from 0 to 1.0, which corresponds to a real frequency range of 0 to the sampling rate. This frequency also wraps around 1.0, so that a normalized frequency of 1.1 is equivalent to 0.1.

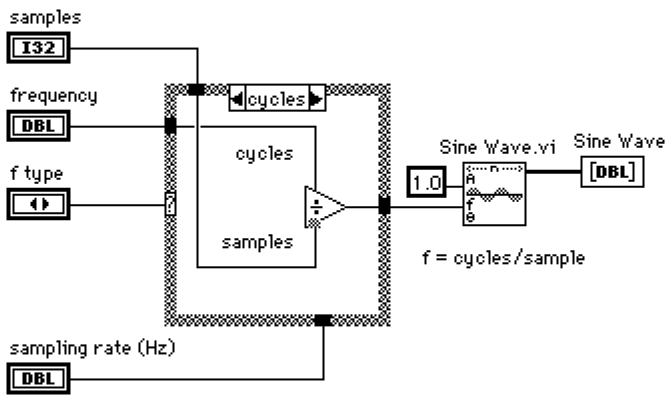
If you use some of these VIs, you must convert your frequency units to the normalized units of cycles/sample. You must use these normalized units with the following VIs.

Sine Wave
Square Wave
Sawtooth Wave
Triangle Wave
Arbitrary Wave
Chirp Pattern

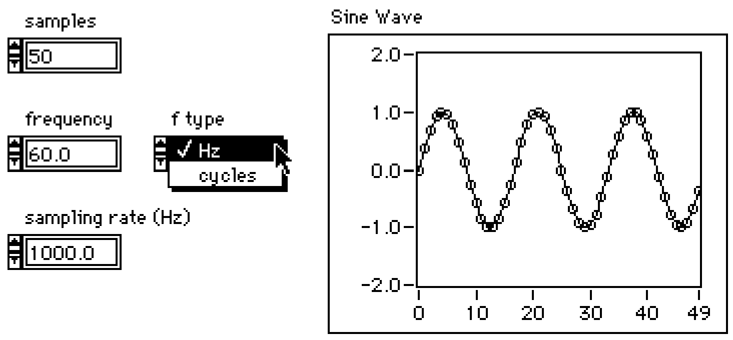
If you are used to working in frequency units of cycles, you can convert cycles to cycles/sample by dividing cycles by the number of samples generated. The following example illustration shows the Sine Wave VI, which is being used to generate two cycles of a sine wave.



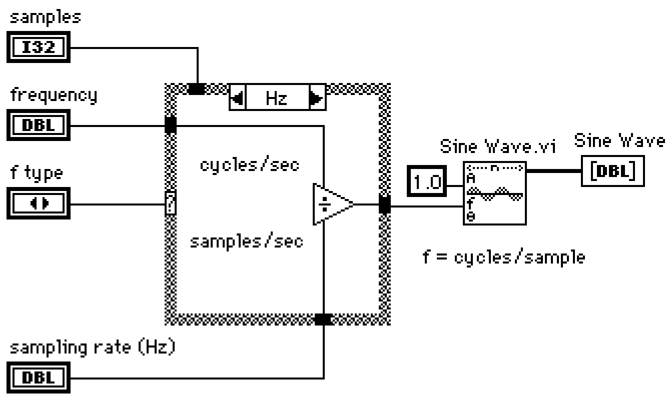
The following illustration shows the block diagram for converting cycles to cycles/sample.



However, you may need to use frequency units of Hz. If you need to convert to Hz or cycles/sec to cycles/sample, divide your frequency in cycles/sec by the sampling rate given in samples/sec. The following example illustration shows the Sine Wave VI, which is being used to generate a 60 Hz sine signal.



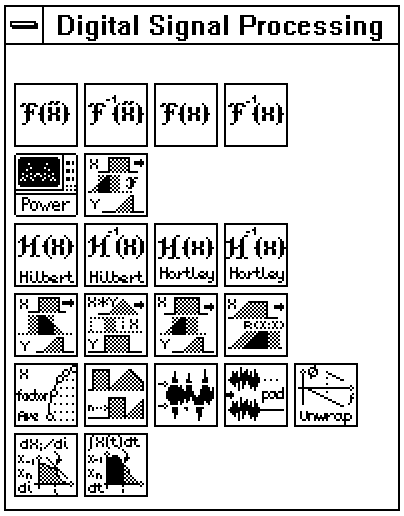
The following illustration shows the block diagram for generating a Hz sine signal.



Digital Signal Processing VIs

This topic describes the VIs that process and analyze an acquired or simulated signal. The digital signal processing VIs perform frequency domain transformations, frequency domain analysis, time domain analysis, and other transforms, such as the Fourier, Hartley, and Hilbert transforms. For more information about the Fast Fourier Transform, see the topic Fast Fourier Transform (FFT).

The following illustration shows the options that are available on the **Digital Processing Signal** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

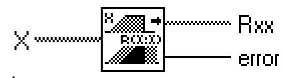


AutoCorrelation
Complex FFT
Convolution
Cross Power
CrossCorrelation
Decimate
Deconvolution
Derivative x(t)
Fast Hilbert Transform
FHT
Integral x(t)
Inverse Complex FFT
Inverse Fast Hilbert Transform
Inverse FHT
Inverse Real FFT
Power Spectrum
Real FFT
Unwrap Phase
Y[i] = Clip {X[i]}
Y[i] = X[i-n]
Zero Padder

For examples of how to use the digital signal processing VIs, see the examples located in `examples\analysis\dspxmpl.llb`.

AutoCorrelation (Advanced Only)

Computes the autocorrelation of the input sequence **x**.



≤ **X**.

≤ **Rxx**.

≤ **error**. See Analysis Error Codes for a description of the error.

The autocorrelation $Rxx(t)$ of a function $x(t)$ is defined as

$$R_{xx}(t) = x(t) \otimes x(t) = \int_{-\infty}^{\infty} x(\tau)x(t+\tau)dt\,,$$

where the symbol $\otimes$ denotes correlation.

For the discrete implementation of this VI, let Y represent a sequence whose indexing can be negative, let n be the number of elements in the input sequence **x**, and assume that the indexed elements of **x** that lie outside its range are equal to zero,

$xj = 0, \quad j < 0 \quad \text{or} \quad j \quad n.$

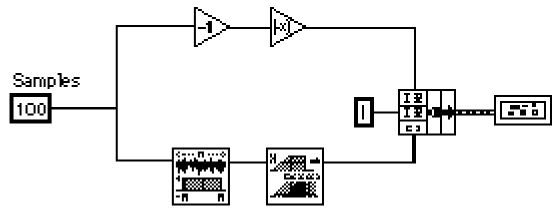
Then the VI obtains the elements of Y using

$$y_j = \sum_{k=0}^{n-1} x_k x_{j+k} \qquad \text{for } j = -(n-1,),(n-2,),\ldots,-2,-1,0,1,2,\ldots,n-1$$

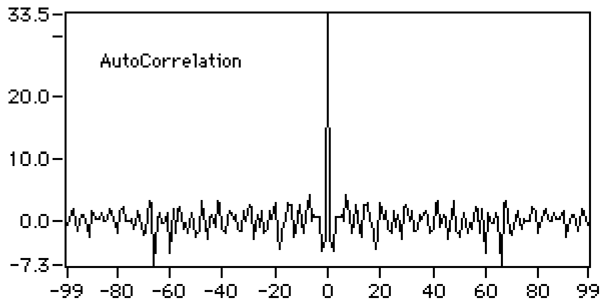
The elements of the output sequence **Rxx** are related to the elements in the sequence Y by

$$rxx_i = y_{i-(n-1)} \qquad \text{for } i = 0,\ 1,\ 2,\ \ldots,\ 2n\text{-}2.$$

Notice that the number of elements in the output sequence **Rxx** is $2n - 1$. Because you cannot use negative numbers to index LabVIEW arrays, the corresponding correlation value at t = 0 is the nth element of the output sequence **Rxx**. Therefore, **Rxx** represents the correlation values that the VI shifted n times in indexing. The following block diagram shows one way to display the correct indexing for the autocorrelation function.

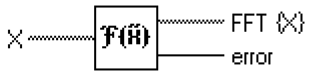


The following graph is the result of the preceding block diagram.



Complex FFT (Advanced Only)

Computes the Fourier transform of the input sequence **X**.



You can use this VI to perform an FFT on an array of complex numeric representations.

≤ **X**.
[CDB] **FFT {X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the complex output sequence, then

$$Y = F\{X\}.$$

You can use the Complex FFT VI to perform the following operations when **X** has one of the complex LabVIEW data types.

- The FFT of a complex-valued sequence X
- The DFT of a complex-valued sequence X

The Complex FFT VI first analyzes the input data, and based on this analysis, the VI calculates the Fourier transform of the data by executing one of the preceding options. All these routines take advantage of the concurrent processing capabilities of the CPU and FPU.

When the number of samples in the input sequence X is a valid power of 2,

$$n = 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$$

where n is the number of samples, the VI computes the fast Fourier transform by applying the split-radix algorithm. The largest complex FFT the VI can compute is $2^{23} = 8{,}388{,}608$ (8M).

When the number of samples in the input sequence X is *not* a valid power of 2,

≤

where n is the number of samples, the VI computes the discrete Fourier transform by applying the Chirp-Z algorithm. The largest complex DFT that can be computed is

$2^{22} - 1 = 4{,}194{,}303$ (4M - 1).

Note*:* **The advantages of the FFT include its speed and memory efficiency because the VI performs the transform in place. The size of the input sequence, however, must be a power of 2. The DFT can efficiently process any size sequence, but the DFT is slower than the FFT and uses more memory, because it must store intermediate results during processing.**

Let Y be the complex output sequence and n be the number of samples in it. Using equation (3-7), you can show that

$Y_{n-i} = Y_{-i}$

which means you can interpret the $(n-i)^{th}$ element of Y as the

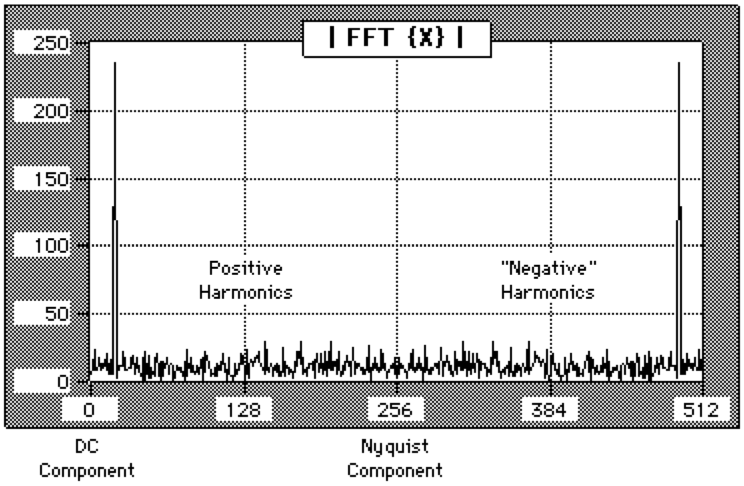
$-i^{th}$ element of the sequence, if it could be physically realized, which represents the negative $\leq$ harmonic.

If n is even, let k = n/2. The following table shows the format of the complex output sequence.

Array Element	**Interpretation**
Y_0	DC component
Y_1	1st harmonic or fundamental
Y_2	2nd harmonic
Y_3	3rd harmonic
.	.
.	.
.	.
Y_{k-2}	$(k-2)^{th}$ harmonic
Y_{k-1}	$(k-1)^{th}$ harmonic
Y_k	Nyquist harmonic
$Y_{k+1} = Y_{n-(k-1)} = Y_{-(k-1)}$	$-(k-1)^{th}$ harmonic*
$Y_{k+2} = Y_{n-(k-2)} = Y_{-(k-2)}$	$-(k-2)^{th}$ harmonic*
.	.
.	.
.	.
Y_{n-3}	-3rd harmonic*
Y_{n-2}	- 2nd harmonic*
Y_{n-1}	-1st harmonic*

*These entries represent negative harmonics

The following illustration represents this complex sequence.

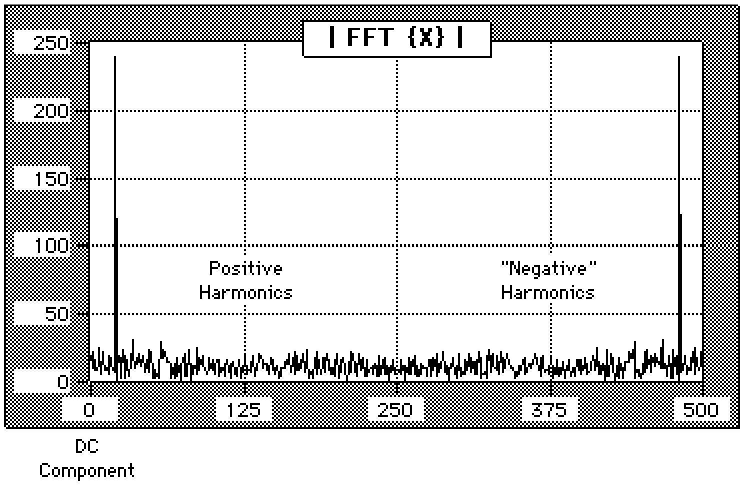


If n is odd, let . The following table shows the format of the complex output sequence Y.

Array Element	Interpretation
≤	DC component
≤	1st harmonic or fundamental
≤	2nd harmonic
≤	3rd harmonic
.	.
.	.
.	.
≤	$k^{th} - 1$ harmonic
≤	k^{th} harmonic
≤	$-k^{th}$ harmonic
≤	≤
.	.
.	.
.	.
≤	-3rd harmonic*
≤	- 2nd harmonic*
≤	-1st harmonic*

*These entries represent negative harmonics

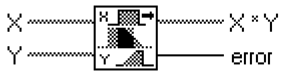
The following illustration represents the preceding table.



This format is an accepted standard in digital signal processing applications. It is convenient because it simplifies performing the inverse transform to obtain the final, processed result.

Convolution (Advanced Only)

Computes the convolution of the input sequences **X** and **Y**.



≤ **X**.
≤ **Y**.
≤ **X * Y**. The convolution of **X** and **Y**.
≤ **error**. See Analysis Error Codes for a description of the error.

The convolution $h(t)$, of the signals $x(t)$ and $y(t)$ is defined as

$$h(t) = x(t) * y(t) = \int_{-\infty}^{\infty} x(\tau) y(t - \tau) d\tau$$

where the symbol $*$ denotes convolution.

For the discrete implementation of the convolution, let h represent the output sequence **X * Y**, let n be the number of elements in the input sequence **X**, and let m be the number of elements in the input sequence **Y**. Assuming that indexed elements of **X** and **Y** that lie outside their range are zero,

$$x_i = 0, \ i < 0 \ \text{ or } \ i \geq n$$

and

$$y_i = 0, \ j < 0 \ \text{ or } \ j \geq m$$

then you obtain the elements of h using

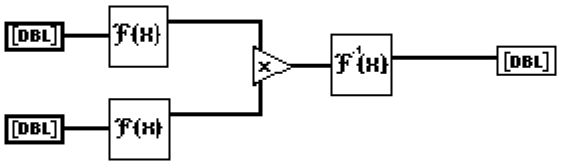
$$h_j = \sum_{k=0}^{n-1} x_k y_{i-k}$$

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

size = n + m - 1,

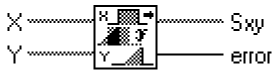
where size denotes the total number of elements in the output sequence **X** * **Y**.

Note: This is not a circular convolution. Because x(t) * Y(t) X(f) Y(f) is a Fourier transform pair, you can create a circular version of the convolution using a diagram similar to the following diagram.



Cross Power (Advanced Only)

Computes the cross power spectrum of the input sequences **X** and **Y**.



≤ **X**.
≤ **Y**.
≤ **Sxy**.
≤ **error**. See Analysis Error Codes for a description of the error.

The cross power, $S_{xy}(f)$ of the signals $x(t)$ and $y(t)$ is defined as

$$S_{xy}(f) = X*(f)\,Y(f)$$

where $X^*(f)$ is the complex conjugate of $X(f)$, $X(f) = F\{x(t)\}$, and $Y(f) = F\{y(t)\}$.

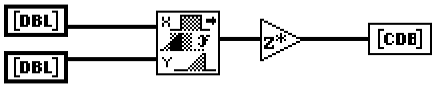
This VI uses the FFT or DFT routine to compute the cross power spectrum, which is given by

$$s_{xy} = \frac{1}{n^2} F^*\{X\} F\{Y\}$$

where s_{xy} represents the complex output sequence **Sxy**, and n is the number of samples that can accommodate both input sequences **X** and **Y**.

The largest cross power that the VI can compute via the FFT is 2^{23} (8,388,608 or 8M).

Note: Some textbooks define the cross power spectrum as $S'_{xy}(f) = X(f)Y^*(f)$. If you prefer this definition of cross power to the one specified in this VI, take the complex conjugate of the output sequence Sxy. Because the VI operates on the real and imaginary portions separately, you can use the following diagram to obtain the results for $S'_{xy}(f)$.



When the number of samples in **X** and **Y** are equal and are a valid power of 2,

$$n = m = 2^k \qquad \text{for } k = 1, 2, 3, \ldots, 23,$$

where n is the number of samples in **X**, and m is the number of samples in **Y**, the VI makes direct calls to the FFT routine to compute the complex, cross power sequence. This method is extremely efficient in both execution time and memory management because the VI performs the operations inplace.

When the number of samples in **X** and **Y** are not equal,

$n \neq m$,

where n is the number of samples in **X**, and m is the number of samples in **Y**, the VI first resizes the smaller sequence by padding it with zeros to match the size of the larger sequence. If this size is a valid power of 2,

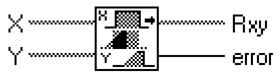
$\max(n,m) = 2k \qquad \text{for } k = 1, 2, 3, ..., 23,$

the VI computes the cross power spectrum using the FFT; otherwise the VI uses the slower DFT to compute the cross power spectrum. Thus, the size of the complex output sequence is

size = max(n,m).

CrossCorrelation (Advanced Only)

Computes the cross correlation of the input sequences **X** and **Y**.



≤ **X**.
≤ **Y**.
≤ **Rxy**.
≤ **error**. See Analysis Error Codes for a description of the error.

The cross correlation $Rxy(t)$ of the signals $x(t)$ and $y(t)$ is defined as

$$R_{xy}(t) = x(t) \otimes y(t) = \int_{-\infty}^{\infty} x(\tau) y(t+\tau) d\tau,$$

where the symbol ≤ denotes correlation.

The discrete implementation of this VI is as follows. Let h represent a sequence whose indexing can be negative, let n be the number of elements in the input sequence **X**, let m be the number of elements in the sequence **Y**, and assume that the indexed elements of **X** and **Y** that lie outside their range are equal to zero,

$$x_j = 0, \ j < 0 \ \text{ or } \ j \geq n$$

and

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

Then the VI obtains the elements of h using

$$h_j = \sum_{k=0}^{n-1} x_k y_{j+k}$$

for $j = -(n-1), -(n-2), ..., -2, -1, 0, 1, 2, ..., m-1$.

The elements of the output sequence **Rxy** are related to the elements in the sequence h by

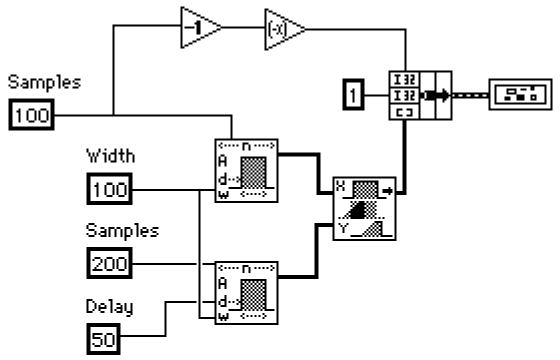
$rxy_i = h_{i-(n-1)} \qquad \text{for } i = 0, 1, 2, \ldots, \text{size}-1,$

$\text{size} = n + m - 1$

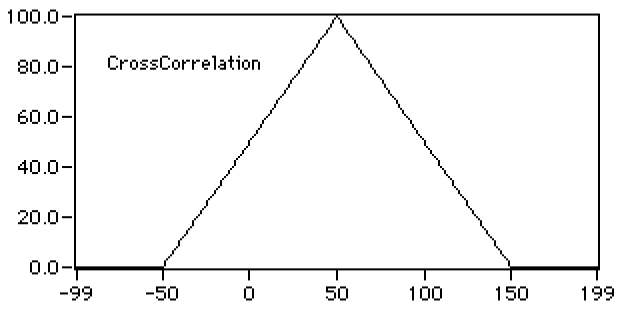
where size is the number of elements in the output sequence **Rxy**.

Because you cannot index LabVIEW arrays with negative numbers, the corresponding cross correlation value at $t = 0$ is the n^{th} element of the output sequence **Rxy**. Therefore, **Rxy** represents the correlation values that the VI shifted n times in indexing.

The following block diagram shows one way to index the CrossCorrelation VI.

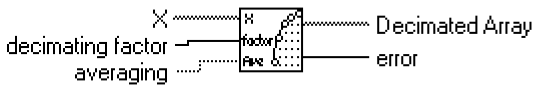


The following graph is the result of the preceding block diagram.



Decimate (Advanced Only)

Decimates the input sequence **X** by the **decimating factor** and the **averaging** binary control.



≤ **X**. The number of elements in **X** must be greater than or equal to the **decimating factor**.

≤ **decimating factor** must be greater than zero:

0 < **decimating factor** $\leq n$.

If **decimating factor** is greater than the number of samples in **X** or less than or equal to zero, the VI sets **Decimated Array** to an empty array and returns an error. **decimating factor** defaults to 2.

≤ **averaging** defaults to false.

≤ **Decimated Array**.

≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Decimated Array**, the VI obtains the elements of the sequence Y using

$$y_i = \begin{cases} x_{im} & \text{if ave is false} \\ \dfrac{1}{m}\displaystyle\sum_{k=0}^{m-1} x_{(im+k)} & \text{if ave is true} \end{cases}$$

for $i = 0, 1, 2, \ldots, \text{size}-1$

$$\text{size} = \text{trunc}\left(\frac{n}{m}\right),$$

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

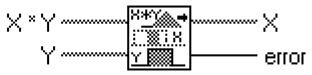
m is the **decimating factor**,

ave is the **averaging** option, and

size is the number of elements in the output sequence **Decimated Array**.

Deconvolution (Advanced Only)

Computes the deconvolution of the input sequences **X * Y** and **Y**.



≤ **X * Y**. The number of elements in **X * Y** must be greater than or equal to the number of elements in **Y**: n m. If the number of elements in **X * Y** is less than the number of elements in **Y**, the VI sets **X** to an empty array and returns an error.

≤ **Y**.

≤ **X**. The number of elements in **X** is

$$\text{size} = n - m + 1,$$

where n is the number of elements in **X * Y,** and m is the number of elements in **Y**.

≤ **error**. See Analysis Error Codes for a description of the error.

The VI can use Fourier identities to realize the convolution operation because

$$x(t) \quad y(t) \leq X(f)\,Y(f)$$

is a Fourier transform pair, where the symbol denotes convolution, and the deconvolution is the inverse of the convolution operation. If $h(t)$ is the signal resulting from the deconvolution of the signals $x(t)$ and $y(t)$, the VI obtains $h(t)$ using the equation

$$h(t) = F^{-1}\left(\frac{X(f)}{Y(f)}\right),$$

where $X(f)$ is the Fourier transform of $x(t)$, and

$Y(f)$ is the Fourier transform of $y(t)$.

The VI performs the discrete implementation of the deconvolution using the following steps.

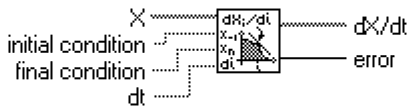
1. Compute the Fourier transform of the input sequence **X Y**.
2. Compute the Fourier transform of the input sequence **Y**.

3. Divide the Fourier transform of **X Y** by the Fourier transform of **Y**. Call the new sequence h.
4. Compute the inverse Fourier transform of H to obtain the deconvoluted sequence **X**.

Note: **The deconvolution operation is a numerically unstable operation, and it is not always possible to solve the system numerically. Computing the deconvolution via FFTs is perhaps the most stable generic algorithm that does not require sophisticated DSP techniques. However, it is not free of errors (for example, when there are zeros in the Fourier transform of the input sequence Y).**

Derivative x(t) (Advanced Only)

Performs a discrete differentiation of the sampled signal **X**.



≤ **X** is the sampled signal.
≤ **initial condition** defaults to 0.0.
≤ **final condition** defaults to 0.0.
≤ **dt** is the sampling interval and must be greater than zero. If **dt** is less than or equal to zero, the VI sets **d/dt X** to an empty array and returns an error. **dt** defaults to 1.0.
≤ **dX/dt**.
≤ **error**. See Analysis Error Codes for a description of the error.

The differentiation f(t) of a function F(t) is defined as

$$f(t) = \frac{d}{dt}F(t).$$

Let Y represent the sampled output sequence **d/dt X**. The discrete implementation is given by

$$y_i = \frac{1}{2dt}(x_{i+1} - x_{i-1})$$

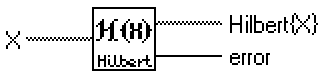
for $i = 0, 1, 2, ..., n-1$,

where n is the number of samples in **x(t)**,

x_{-1} is specified by **initial condition** when i = 0, and
$x_n$is specified by **final condition** when $i = n$-1.
The **initial condition** and **final condition** minimize the error at the boundaries.

Fast Hilbert Transform (Advanced Only)

Computes the fast Hilbert transform of the input sequence **X**.



≤ **X**.
≤ **Hilbert {X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

The Hilbert transform of a function $x(t)$ is defined as

$$h(t) = H\{x(t)\} = -\frac{1}{\pi}\int_{-\infty}^{\infty}\frac{x(\tau)}{t-\tau}d\tau.$$

Using Fourier identities, you can show the Fourier transform of the Hilbert transform of $x(t)$ is

$h(t) \quad H(f) = - j \operatorname{sgn}(f) X(f)$

where $x(t)$ ≤ $X(f)$ is a Fourier transform pair and

$$\operatorname{sgn}(f) = \begin{cases} 1 & f > 0 \\ 0 & f = 0 \\ -1 & f < 0 \end{cases}$$

The VI performs the discrete implementation of the Hilbert transform with the aid of the FFT routines based upon the $h(t)$ ≤ $H(f)$ Fourier transform pair by taking the following steps. Refer the output format of the FFT VI for more information.

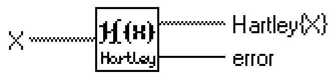
1. Fourier transform the input sequence **X**: $Y = F\{X\}$.
2. Set the DC component to zero:≤=0.
3. If the sequence Y is an even size, set the Nyquist component to zero: Y_{Nyq}=0.
4. Multiply the positive harmonics by -j.
5. Multiply the negative harmonics by j. Call the new sequence H, which is of the form $H_k = -j\operatorname{sgn}(k)\,Y_k$.
6. Inverse Fourier transform H to obtain the Hilbert transform of **X**.

You use the Hilbert transform to extract instantaneous phase information, obtain the envelope of an oscillating signal, obtain single-sideband spectra, detect echoes, and reduce sampling rates.

Note: Because the VI sets the DC and Nyquist components to zero when the number of elements in the input sequence is even, you cannot always recover the original signal with an inverse Hilbert transform. The Hilbert transform works well with bandpass limited signals, which exclude the DC and the Nyquist components.

FHT (Advanced Only)

Computes the fast Hartley transform (FHT) of the input sequence **X**.



≤ **X**. To properly compute the FHT of **X**, the number of elements, n, in the sequence must be a valid power of 2:

$m = 2_m \qquad \text{for } m = 1, 2, 3, \ldots, 23.$

If the number of elements in **X** is not a valid power of 2, the VI sets **Hartley{X}** to an empty array and returns an error.

≤ **Hartley {X}**.

≤ **error**. See Analysis Error Codes for a description of the error.

The Hartley transform of a function $x(t)$ is defined as

$$X(f) = \int_{-\infty}^{\infty} x(t)\operatorname{cas}(2\pi ft)\,dt$$

where $\operatorname{cas}(x) = \cos(x) + \sin(x)$.

If Y represents the output sequence **Hartley{X}** obtained via the FHT, then Y is obtained through the discrete implementation of the Hartley integral:

$$Y_k = \sum_{i=0}^{n-1} X_i \operatorname{cas}\left(\frac{2\pi ik}{n}\right), \qquad \text{for } k = 0, 1, 2, \ldots, n-1.$$

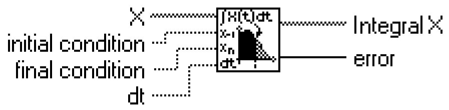
where n is the number of elements in **X**.

The Hartley transform maps real-valued sequences into real-valued frequency domain sequences. You can use it instead of the Fourier transform to convolve signals, deconvolve signals, correlate signals, and find the power spectrum. You can also derive the Fourier transform from the Hartley transform.

When the sequences to be processed are real-valued sequences, the Fourier transform produces complex-valued sequences in which half of the information is redundant. The advantage of using the Hartley transform instead of the Fourier transform is that the Hartley transform uses half the memory to produce the same information the FFT produces. Further, the FHT is calculated in place and is as efficient as the Fourier transform. The disadvantage of the FHT is that the size of the input sequence must be a valid power of 2.

Integral x(t) (Advanced Only)

Performs the discrete integration of the sampled signal **X**.



≤ **X** is the sampled signal.
≤ **initial condition** defaults to 0.0.
≤ **final condition** defaults to 0.0.
≤ **dt** is the sampling interval, and must be greater than zero. If **dt** is less than or equal to zero, the VI sets **Integral X** to an empty array and returns an error. **dt** defaults to 1.0.
≤ **Integral X** is the sampled output sequence**.**
≤ **error**. See Analysis Error Codes for a description of the error.

The integral $F(t)$ of a function $f(t)$ is defined as

$$F(t) = \int f(t)\,dt$$

Let Y represent the sampled output sequence **Integral X**. The VI obtains the elements of Y using

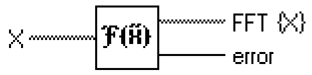
$$y_i = \frac{1}{6}\sum_{j=0}^{i}\left(x_{j-1} + 4x_j + x_{j+1}\right)dt \qquad \text{for } i = 0, 1, 2, \ldots, n-1,$$

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

x_1 is specified by **initial condition** when $i = 0$, and
≤is specified by **final condition** when $i = n-1$.
The **initial condition** and **final condition** minimize the overall error by increasing the accuracy at the boundaries, especially when the number of samples is small. Determining boundary conditions before the fact enhances accuracy.

Inverse Complex FFT (Advanced Only)

Computes the inverse Fourier transform of the complex input sequence **FFT X**.



You can use this VI to perform an inverse FFT on an array of one of the LabVIEW complex numeric representations.

≤ **FFT {X}**.
≤ **X**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence, then

$$Y = F_{-1}\{X\}$$

You can use this VI to perform the following operations when **FFT X** has one of the complex LabVIEW data types.

- The inverse FFT of a complex-valued sequence X
- The inverse DFT of a complex-valued sequence X

The Inverse Complex FFT VI first analyzes the input data and, based on this analysis, inverse Fourier transforms the data by executing one of the preceding options. All these routines take advantage of the concurrent processing capabilities of the CPU and FPU.

When the number of samples in the input sequence *X* is a valid power of 2,

$$n = 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$$

where n is the number of samples, the VI computes the inverse FFT by applying the split-radix algorithm. The longest sequence with an inverse complex FFT that the VI can compute is ≤.

When the number of samples in the input sequence X is *not* a valid power of 2,

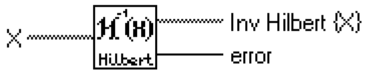
$$n \neq 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$$

where n is the number of samples, the VI computes the inverse DFT by applying the Chirp-Z algorithm. The longest sequence with an inverse complex DFT that the VI can compute is $2^{22} - 1$ (4,194,303 or 4M - 1).

Note: **The advantages of the inverse FFT include its speed and memory efficiency because the transform is performed in place. The size of the input sequence, however, must be a power of 2. The inverse DFT can efficiently process any size sequence, but the inverse DFT is slower than the inverse FFT and uses more memory because it must store intermediate results during processing.**

Inverse Fast Hilbert Transform (Advanced Only)

Computes the inverse fast Hilbert transform of the input sequence **X**.



≤ **X**.
≤ **Inv Hilbert {X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

The inverse Hilbert transform of a function $h(t)$ is defined as

$$h(t) = H^{-1}\{h(t)\} = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{h(\tau)}{t-\tau}d\tau.$$

Using the definition of the Hilbert transform

$$h(t) = H\{h(t)\} = \frac{1}{\pi}\int_{-\infty}^{\infty}\frac{x(\tau)}{t-\tau}d\tau,$$

you can obtain the inverse Hilbert transform by negating the forward Hilbert transform

$$x(t) = H^{-1}\{h(t)\} = H\{h(t)\}.$$

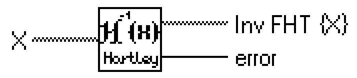
Therefore, the VI performs the discrete implementation of the inverse Hilbert transform with the aid of the Hilbert transform by taking the following steps.

1. Hilbert transform the input sequence **X**: $Y = H\{X\}$.
2. Negate Y to obtain the inverse Hilbert transform: $H^{-1}\{X\} = -Y$.

For more information on the algorithm this VI uses, refer to the description of the Fast Hilbert Transform VI in this chapter.

Inverse FHT (Advanced Only)

Computes the inverse fast Hartley transform of the input sequence **X**.



≤ **X**. To properly compute the inverse FHT of **X**, the number of elements, n, in the sequence must be a valid power of 2:

$n = 2^m$ for $m = 1, 2, 3, \ldots, 23$.

If the number of elements in **X** is not a valid power of 2, the VI sets **Inv FHT{X}** to an empty array and returns an error.

≤ **Inv FHT {X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

The inverse Hartley transform of a function $X(f)$ is defined as

$$x(t) = \int_{-\infty}^{\infty} X(f)\,\mathrm{cas}(2\pi ft)\,df$$

where $\mathrm{cas}(x) = \cos(x) + \sin(x)$.

If Y represents the output sequence **Inv FHT{X}**, the VI calculates Y through the discrete implementation of the inverse Hartley integral:

$$Y_k = \frac{1}{n}\sum_{i=0}^{n-1} X_i \,\mathrm{cas}\frac{2\pi ik}{n}, \qquad \text{for } k = 0, 1, 2, \ldots, n\text{-}1.$$

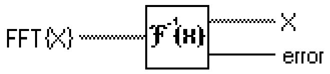
where n is the number of elements in **X**.

The inverse Hartley transform maps real-valued frequency sequences into real-valued sequences. You can use it instead of the inverse Fourier transform to convolve, deconvolve, and correlate signals. You can also derive the Fourier transform from the Hartley transform.

See the description of the FHT VI for a comparison of the Fourier and Hartley transforms.

Inverse Real FFT (Advanced Only)

Computes the Inverse Real Fast Fourier Transform (FFT) or the Inverse Real Discrete Fourier Transform (DFT) of the input sequence **FFT{X}**.



≤ **FFT{X}** is the complex input sequence.
≤ **X** is the Inverse Real FFT of **FFT{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.
The input sequence is complex-valued. The Inverse Real FFT VI automatically determines the options which are

1. Inverse Real FFT of a complex-valued sequence if the size is a power of 2.
2. Inverse Real DFT of a complex-valued sequence if the size is not a power of 2.

The Inverse Real FFT VI executes Inverse FFT routines if the size of the input sequence is a valid power of 2:

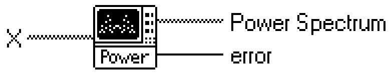
$$\text{size} = 2^m, m = 1, 2, \ldots, 23.$$

If the size of the input sequence is not a power of 2, the Inverse Real FFT VI calls an efficient Inverse DFT routine.

The output sequence **X** = Inverse Real FFT[**FFT{X}**] is real and it returns in one real array.

Power Spectrum (Advanced Only)

Computes the **Power Spectrum** of the input sequence **X**.



≤ **X**.
≤ **Power Spectrum** computes the harmonic power content of periodic signals. If **X** represents actual measurements in volts, the VI expresses the normalized units of the output sequence **Power Spectrum** in watts on a 1-
Ω basis.
≤ **error**. See Analysis Error Codes for a description of the error.
The **Power Spectrum** $S_{xx}(f)$ of a function $x(t)$ is defined as

$$S_{xx}(f) = X*(f)X(f) = |X(f)|2$$

where $X(f) = F\{x(t)\}$, and $X^*(f)$ is the complex conjugate of $X(f)$.

This VI uses the FFT and DFT routines to compute the power spectrum, which is given by

$$S_{xx} = \frac{1}{n^2 |F\{X\}|^2},$$

where S_{xx} represents the output sequence **Power Spectrum**, and n is the number of samples in the input sequence **X**.

When the number of samples, n, in the input sequence X is a valid power of 2,

$$n = 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$$

the **Power Spectrum** VI computes the fast Fourier transform of a real-valued sequence using the split-radix algorithm and efficiently scales the magnitude square. The largest **Power Spectrum** the VI can compute using the FFT is 223 (8,388,608 or 8M).

When the number of samples in the input sequence *X* is *not* a valid power of 2,

$$n \neq 2^m \qquad \text{for } m = 1, 2, 3, \ldots, 23,$$

where n is the number of samples, the Power Spectrum VI computes the discrete Fourier transform of a real-valued sequence using the Chirp-Z algorithm and scales the magnitude square. The largest **Power Spectrum** the VI can compute using the fast DFT is $\leq$.

The FFT computation of the **Power Spectrum** is extremely fast and memory efficient because the transform is real and done in the same space. However, the size of the input sequence must be exactly a power of 2. The DFT version efficiently computes the **Power Spectrum** of any size sequence. The DFT version is slower than the FFT version, uses more memory, and is not as efficient in scaling.

Let Y be the Fourier transform of the input sequence **X** and n be the number of samples in it. Using equation (3-7), you can show that

$$|Y_{n-i}|^2 = |Y_{-i}{}^2|.$$

You can interpret the power in the $(n - 1)^{th}$ element of Y as the power in the

$\leq$element of the sequence, which represents the power in the *negative*
$\leq$harmonic. You can find the total power for the
$\leq$harmonic (DC and Nyquist component not included) using

$$\text{Power in } i^{th} \text{ harmonic} = 2|Y_i|^2 = |Y_i|^2 + |Y_{n-1}|^2, \quad 0 < i < \frac{n}{2}.$$

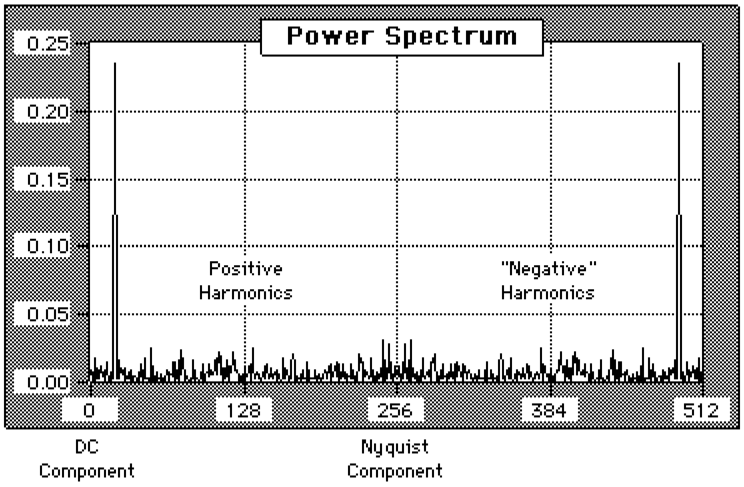
The total power in the DC and Nyquist components are $|Y_0|^2$ and $|Y_{n/2}|^2$, respectively.

If n is even, let . The following table shows the format of the output sequence $\leq$ corresponding to the **Power Spectrum**.

Array Element	Interpretation
Sxx_0	Power in DC component
$Sxx_1 = Sxx_{(n-1)}$	Power in 1st harmonic or fundamental
$Sxx_2 = Sxx_{(n-2)}$	Power in 2nd harmonic
$Sxx_3 = Sxx_{(n-3)}$	Power in 3rd harmonic
.	.
.	.
.	.
$Sxx_{(k-2)} = Sxx_{n-(k-2)}$	Power in $\leq$harmonic

$Sxx_{(k-1)} = Sxx_{n-(k-1)}$	Power in ≤harmonic
Sxx_k	Power in Nyquist harmonic

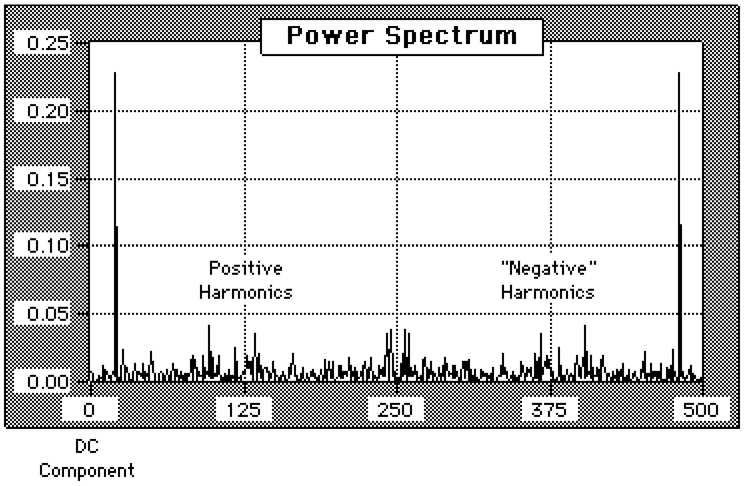
The following illustration represents the preceding table information.



If n is odd, let $k = \frac{n-1}{2}$. The following table shows the format of the output sequence *Sxx* corresponding to the **Power Spectrum**.

Array Element	**Interpretation**
≤	Power in DC component
≤	Power in 1st harmonic or fundamental
≤	Power in 2nd harmonic
≤	Power in 3rd harmonic
.	.
.	.
.	.
≤	Power in ≤harmonic
≤	Power in ≤harmonic
$Sxx_k = Sxx_{n-k}$	Power in ≤ harmonic

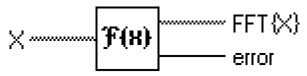
The following illustration represents the preceding table information.



The format described in the preceding tables is an accepted standard in digital signal processing applications. The topic, Analysis Examples , shows several simple methods you can use to manipulate the data and display the results in a more familiar way.

Real FFT (Advanced Only)

Computes the Real Fast Fourier Transform (FFT) or the Real Discrete Fourier Transform (DFT) of the input sequence **X**.



≤ **X** is the real input sequence.
≤ **FFT{X}** is the FFT of **X**.
≤ **error**. See Analysis Error Codes for a description of the error.

The input sequence is real-valued. The Real FFT VI automatically determines the options, which are

1. FFT of a real-valued sequence
2. DFT of a real-valued sequence

The Real FFT VI executes FFT routines if the size of the input sequence is a valid power of 2:

size = 2m, m = 1, 2,..., 23.

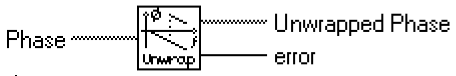
If the size of the input sequence is not a power of 2, the Real FFT VI calls an efficient Real DFT routine.

The output sequence Y = Real FFT[**X**] is complex and returns in one complex array:

Y = YRe + jYIm

Unwrap Phase (Advanced Only)

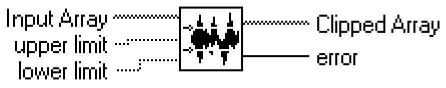
Unwraps the **Phase** array by eliminating discontinuities whose absolute values exceed ¹.



≤ **Phase** is expressed in radians.
≤ **Unwrapped Phase** is expressed in radians.
≤ **error**. See Analysis Error Codes for a description of the error.

Y[i] = Clip {X[i]} (Advanced Only)

Clips the elements of **Input Array** to within the bounds specified by **upper limit** and **lower limit**.



≤ **Input Array**.
≤ **upper limit** must be greater than or equal to **lower limit**. If **upper limit** is less than **lower limit**, the VI sets the sequence **Clipped Array** to an empty array and returns an error. **upper limit** defaults to 1.0.
≤ **lower limit** must be less than or equal to **upper limit. lower limit** defaults to 0.0.
≤ **Clipped Array**.
≤ **error**. See Analysis Error Codes for a description of the error.

Let the sequence Y represent the output sequence **Clipped Array**; then the elements of Y are related to the elements of **Input Array** by

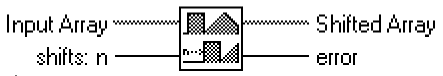
$$y_i = \begin{cases} a & x_i > a \\ x_i & b \le x_i \le a \\ b & x_i < b \end{cases}$$

for i = 0, 1, 2, ..., n-1,

where n is the number of elements in **Input Array**, *a* is the **upper limit**, and *b* is the **lower limit**.

Y[i] = X[i-n] (Advanced Only)

Shifts the elements in the **Input Array** by the specified number of shifts.



≤ **Input Array**.
≤ **shifts: n**. The VI shifts **Input Array** to the right if **shifts: n** is positive and to the left if **shifts: n** is negative. To properly shift **Input Array** without setting the output sequence **Shifted Array** to zero, the absolute value of **shifts: n** must be less than the number of elements in **Input Array**:

$|\textbf{shifts: n}| < n.$

If the absolute value of **shifts: n** is greater than or equal to the number of samples in **Input Array**, the VI sets **Shifted Array** to zero and returns an error. **shifts: n** defaults to 0.

≤ **Shifted Array**.
≤ **error**. See Analysis Error Codes for a description of the error.

Let the sequence Y represent the output sequence **Shifted Array**; then the elements of Y are related to the elements of **X** by

$$y_i = \begin{cases} x_{i-\text{shifts}} & \text{if } 0 \le i - \text{shifts} < n \\ 0 & \text{elsewhere} \end{cases}$$

for i = 0, 1, 2, ..., n-1

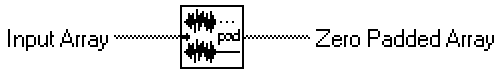
where n is the number of elements in **Input Array**.

Note: This VI does not rotate the elements in the array. The VI disposes of the elements of the

input sequence shifted outside the range, and you cannot recover them by shifting the array in the opposite direction.

Zero Padder (Advanced Only)

Resizes the input sequence **Input Array** to the next higher valid power of 2, sets the new trailing elements of the sequence to zero, and leaves the first n elements unchanged, where n is the number of samples in the input sequence.



≤ **Input Array**. If **Input Array** is not a power of 2, the VI resizes the sequence to the next size that is a valid power of 2. If **Input Array** is already a power of 2, the VI resizes the sequence to the next valid power of 2. For instance, if **Input Array** contains 500 elements, the size of the zero padded output sequence is 512(2^9). If **Input Array** contains 1,024(2^10) elements, the size of the zero padded output sequence is 2,048(2^11).
Special Case: If Input Array is empty, Zero Padded Array is also empty.

≤ **Zero Padded Array**.
This VI is useful when the size of the acquired data buffers is not a power of 2, and you want to take advantage of fast processing algorithms in the analysis VIs. These algorithms include Fourier transforms, Power Spectrum, and fast Hartley transforms, which are extremely efficient for buffer sizes that are a power of 2.

AutoCorrelation.vi

AutoCorrelation

Complex FFT.vi

Complex FFT

Convolution.vi

Convolution

Cross Power.vi

Cross Power

CrossCorrelation.vi

CrossCorrelation

Decimate.vi

Decimate

Deconvolution.vi

Deconvolution

Derivative x(t).vi

Derivative x(t)

Fast Hilbert Transform.vi

Fast Hilbert Transform

FHT.vi

FHT

Integral x(t) .vi

Integral x(t)

Inverse Complex FFT.vi

Inverse Complex FFT

Inverse Fast Hilbert Transform.vi

Inverse Fast Hilbert Transform

Inverse Real FFT.vi

Inverse Real FFT

Power Spectrum.vi

Power Spectrum

Real FFT.vi

Real FFT

Unwrap Phase.vi

Unwrap Phase

Y[i] = Clip {X[i]}.vi

Y[i] = Clip {X[i]}

Y[i] = X[i-n].vi

Y[i] = X[i-n]

Zero Padder.vi

Zero Padder

Inverse FHT.vi

Inverse FHT

Fast Fourier Transform (FFT)

The Fourier transform establishes the relationship between a signal and its representation in the frequency domain. The Fourier transform is a powerful analysis tool for spectral analysis, applied mechanics, acoustics, medical imaging, numerical analysis, instrumentation, and telecommunications.

The definition of the Fourier transform of a signal $x(t)$ is

$$X(f) = F\left(x\{t\}\right) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}dt \qquad (3\text{-}1)$$

and the inverse Fourier transform of a signal $X(f)$ is

$$x(t) = F^{-1}\left(X\{f\}\right) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}dt. \qquad (3\text{-}2)$$

A notation often used to indicate that the signals $x(t)$ and $X(f)$ are a Fourier transform pair and are related via the Fourier transform is

$$x(t) \leq X(f) \qquad (3\text{-}3)$$

You can derive the discrete representation of the Fourier transform equations, equations (3-1) and (3-2), by sampling the Fourier transform pair in equation (3-3) using the following sampling relationships:

$$\Delta t = \frac{1}{f_s} \Delta f = \frac{f_s}{n}$$

where $\lesssim$ is the sampling interval,

Δf is the frequency resolution,
f_s is the sampling frequency, and n is the number of samples in both the time and frequency domain. Thus, the discrete transform pair

$$x_i \Leftrightarrow X_k \qquad (3\text{-}4)$$

is obtained and the discrete Fourier transform is given by

$$X_k = \sum_{i=0}^{n-1} x_i e^{-j2\pi ik/n} \Delta t \qquad (3\text{-}5)$$

and the inverse by

$$x_i = \sum_{i=0}^{n-1} X_k e^{j2\pi ik/n} \Delta f. \qquad (3\text{-}6)$$

X_k in equation (3-5) represents an amplitude spectral density. By multiplying the right-hand side of equation (3-5) by the frequency resolution
$\lesssim$, we arrive at the amplitude spectrum. This amplitude spectrum is the final form of the DFT and inverse DFT, given by equations (3-7) and (3-8), respectively. Notice that the DFT is independent of the sampling rate.

$$X_k = \sum_{i=0}^{n-1} x_i e^{-j2\pi ik/n} \qquad \text{for } k = 0, 1, 2, \ldots, n-1 \qquad (3\text{-}7)$$

$$x_i = \frac{1}{n}\sum_{k=0}^{n-1} x_k e^{j2\pi ik/n} \qquad \text{for } i = 0,1,2,\ldots,n-1 \tag{3-8}$$

Direct implementation of the DFT requires approximately n^2 complex operations, and until recently, it was a time-consuming process. However, when the size of the sequence is

$$n=2^m \qquad \text{for } m = 1, 2, 3, \ldots$$

you can implement the computation of the DFT with approximately $n\log2(n)$ operations. DSP literature refers to these algorithms as fast Fourier transforms (FFTs). Furthermore, with the aid of the FFT, you can find the DFT of any size sequence in approximately $3n\log2(n)$ operations where n is the next power of 2 that accommodates intermediate results. You can find a more detailed explanation of FFT theory in most introductory texts on DSP.

The algorithm implemented in the LabVIEW analysis VIs is known as the Split-Radix algorithm. This algorithm has a form similar to the Radix-4 algorithms with the efficiency of Radix-8 algorithms. The Split-Radix algorithm requires the least number of multiplications among the Radix-2, Radix-4, and Mixed-Radix algorithms.

This manual uses the following notation to denote the discrete Fourier transform of a sequence x

$$X = F\{x\},$$

and

$$x = F^{-1}\{X\}$$

to denote the discrete inverse Fourier transform. The Fourier transform always results in a complex output sequence, and the input sequence can be either *real* or *complex*. Unless otherwise specified, two real sequences represent the complex sequences. If X is a complex sequence, then

$$X_{Re} = \mathrm{Re}\{X\}$$

represents the real part of the complex sequence X,

$$X_{Im} = \mathrm{Im}\{X\}$$

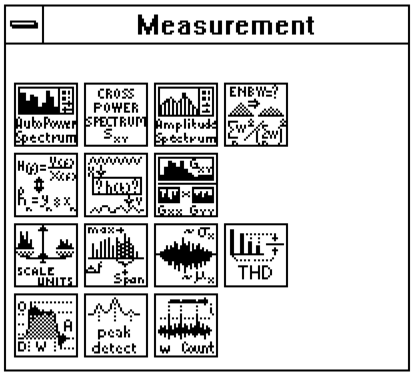
represents the imaginary part of the complex sequence X, and

$$X = X_{Re} + jX_{Im} = \mathrm{Re}\{X\} + j\,\mathrm{Im}\{X\}.$$

Measurement VIs

This topic describes the measurement VIs, which are streamlined to 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. For general information about Measurement VIs, see the Measurement VIs Overview.

The following illustration shows the options that are available on the **Measurement** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

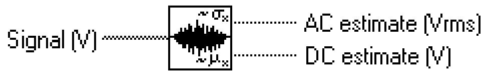


AC & DC Estimator
Amplitude and Phase Spectrum
Auto Power Spectrum
Cross Power Spectrum
Harmonic Analyzer
Impulse Response Function
Network Functions (avg)
Peak Detector
Power & Frequency Estimate
Pulse Parameters
Scaled Time Domain Window
Spectrum Unit Conversion
Threshold Peak Detector
Transfer Function

For examples of how to use the measurement VIs, see the examples located in `examples\analysis\measure\daqmeas.llb` and in `examples\analysis\measure\measxmpl.llb`.

AC & DC Estimator (Advanced Only)

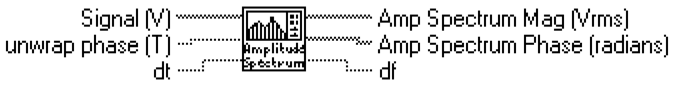
Computes an estimation of the AC and DC levels of the input signal.



≤ **Signal** is the input, time-domain signal, usually in volts. At least three cycles of the signal must be contained in the time-domain record for a valid estimate.
≤ **AC estimate** is the estimate of the input signal AC level, usually in volts rms if the input signal is in volts.
≤ **DC estimate** is the estimate of the input signal DC level, usually in volts if the input signal is in volts.

Amplitude and Phase Spectrum (Advanced Only)

Computes the single-sided, scaled amplitude spectrum magnitude and phase of a real time-domain signal.



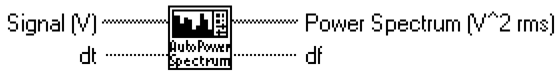
≤ **Signal** is the input, time-domain signal, usually in volts. At least three cycles of the signal must be contained in the time-domain record for a valid estimate.
≤ **unwrap phase**. Set to TRUE to enable phase unwrapping on the output phase Amp Spectrum Phase (radians). If you set **unwrap phase** to FALSE, the VI does not perform unwrapping. The default setting is TRUE.
≤ **dt** is the sample period of the time-domain signal, usually in seconds. **dt** is also
$\frac{1}{f}$ where
≤ is the sampling frequency of the time-domain signal.
≤ **Amp Spectrum Mag** is the single-sided, amplitude spectrum magnitude in volts rms if the input signal is in volts. If the input signal is not in volts, the results are in input signal units rms.
≤ **Amp Spectrum Phase** is the single-sided, amplitude spectrum phase in radians.
≤ **d**f is the line frequency interval of the power spectrum, in Hertz, if **dt** is in seconds.
The VI computes the amplitude spectrum as

$$\frac{\mathrm{FFT(Signal)}}{\mathrm{N}}$$

where N is the number of points in the signal array. The VI then converts the amplitude spectrum to single-sided rms magnitude and phase spectra.

Auto Power Spectrum (Advanced Only)

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



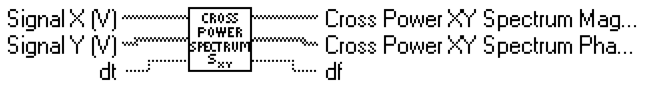
≤ **Signal** is the input, time-domain signal, usually in volts. At least three cycles of the signal must be contained in the time-domain record for a valid estimate.
≤ **dt** is the sample period of the time-domain signal, usually in seconds. **dt** is also
$\frac{1}{f_s}$ where
≤ is the sampled frequency of the time-domain signal.
≤ **Power Spectrum** is the single-sided, power spectrum in volts rms squared if the input signal is in volts. If the input signal is not in volts, the results are in input signal units rms squared.
≤ **df** is the line frequency interval of the power spectrum, in Hertz, if **dt** is in seconds.
This VI computes the power spectrum as

$$\frac{\mathrm{FFT*(Signal)\,x\,FFT(Signal)}}{\mathrm{N}^2}$$

where N is the number of points in the signal array and * denotes complex conjugate. The VI then converts the power spectrum into a single-sided power spectrum result.

Cross Power Spectrum (Advanced Only)

Computes the single-sided, scaled, cross power spectrum of two real-time signals. The cross power spectrum gives the product of the amplitude of the signals X and Y and the difference between their phases (phase of Y minus phase of X).


Signal X [V]
Signal Y [V]
dt
CROSS POWER SPECTRUM S_{XY}
Cross Power XY Spectrum Mag...
Cross Power XY Spectrum Pha...
df

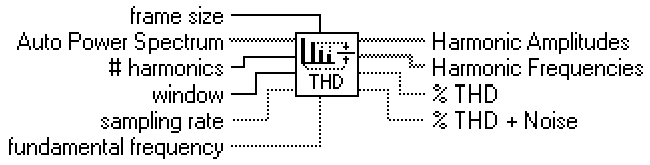
≤ **Signal X** is the input, time-domain signal X, usually in volts. At least three cycles of the signal must be contained in the time-domain record for a valid estimate.
≤ **Signal Y** is the input, time-domain signal Y, usually in volts. At least three cycles of the signal must be contained in the time-domain record for a valid estimate.
≤ **dt** is the sample period of the time-domain signal, usually in seconds. **dt** is also
≤ where
≤ is the sampled frequency of the time-domain signal.
≤ **Cross Power XY Spectrum Mag** is the single-sided, cross power spectrum between signals X and Y in volts rms squared if the input signals are in volts. If the input signals are not in volts, the results are in input signal units rms squared.
≤ **Cross Power XY Spectrum Phase** is the phase spectrum in radians showing the difference between the phases of signal Y and signal X.
≤ **df** is the line frequency interval of the power spectrum, in Hertz, if **dt** is in seconds.
This VI computes the cross power spectrum as

$$\frac{\mathrm{FFT}*(\mathrm{Signal\ X})\mathrm{x}\,\mathrm{FFT}(\mathrm{Signal\ Y})}{\mathrm{N}^2}$$

where N is the number of points in array **Signal X** or **Signal Y**. The VI then converts the cross power spectrum to single-sided magnitude and phase spectra.

Harmonic Analyzer (Advanced Only)

Finds the fundamental and harmonic components (amplitude and frequency) present in the input **Auto Power Spectrum**, and computes the percent of total harmonic distortion (**%THD)** and the total harmonic distortion plus noise (**%THD + Noise)**.



≤ **Auto Power Spectrum** is the single-sided, auto power spectrum of the windowed signal. This array can be the output of a frequency-domain averaging process for improved harmonic estimation.
≤ **frame size** is the number of samples in the time-domain, signal array before it was passed to the Auto Power Spectrum VI. The **frame size** is typically the number of samples in one block of data from a data acquisition operation. If this control is unwired, the frame size used in this VI is set to twice the size of the **Auto Power Spectrum** input array.
≤ **# harmonics** is the number of harmonic components that you want this VI to approximate and use in the THD measurement. This number includes the fundamental component. For example, if you want to compute the second harmonic distortion in your signal, this number should be two: find the fundamental frequency component (say at
f_1 Hz) and its second harmonic (at
f_2= 2
f_1 Hz).
≤ **window** is the window selection that you used for the Scaled Time Domain Window VI. If you did not use a window function (not recommended for an accurate THD estimation), this selector defaults to zero (no window).
≤ **sampling rate** is the input sampling rate in Hz.

≤ **fundamental frequency** is an estimate of the fundamental frequency that you want this VI to use in the harmonic search and in the THD computation. If this control is set to zero (default), then the frequency of the largest non-DC component found in **Auto Power Spectrum** is used as the fundamental frequency.

≤ **Harmonic Amplitudes** is the array of amplitudes of the fundamental component and its harmonics. These values are always positive and are in units of Vrms if the input Auto Power Spectrum values are given in V^2rms.

≤ **Harmonic Frequencies** is the array of frequencies of the fundamental component and its harmonics. These value are in units of Hz if the input sampling rate is given in Hz.

≤ **% THD** is the percent total harmonic distortion present in the input **Auto Power Spectrum**. The THD computation is made using the following equation:

$$\%\mathrm{THD} = \frac{100\sqrt{A(f_2)^2 + A(f_3)^2 + \ldots + A(f_N)^2}}{A(f_1)}$$

where

$A(f_1)$ is the amplitude of the fundamental component

$A(f_n)$ is the amplitude of the

N^{th} harmonic

N is the **# harmonics**

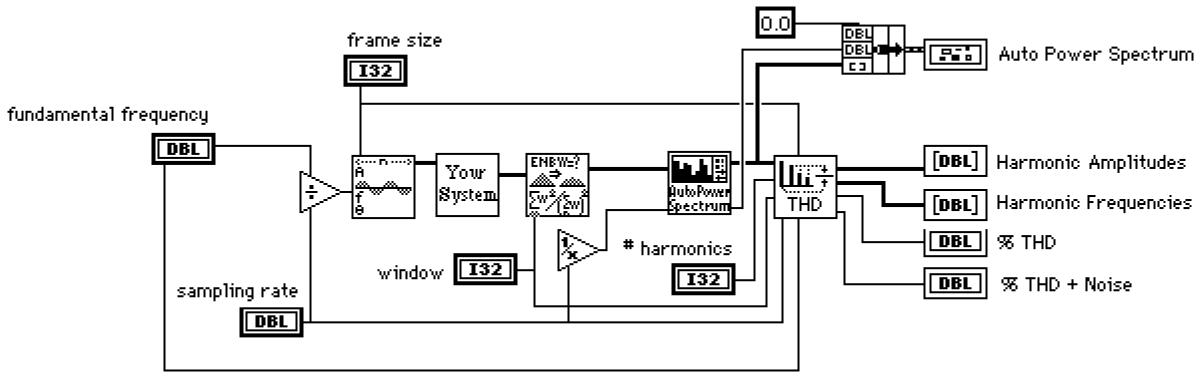
≤ **% THD + Noise** is the percent total harmonic distortion plus noise present in the input **Auto Power Spectrum**.**% THD + Noise** is computed using the following equation:

$$\%\mathrm{THD}+\mathrm{Noise} = \frac{100\sqrt{\mathrm{sum}(\mathbf{APS})}}{A(f_1)}$$

where sum(**APS**) is the sum of the **Auto Power Spectrum** elements minus the elements near DC and near the **fundamental frequency** index.

You must pass the windowed, auto power spectrum of your signal to this VI for it to function correctly. You should pass your time-domain signal through the Scaled Time Domain Window and then through the Auto Power Spectrum, connecting the Auto Power Spectrum output to this VI.

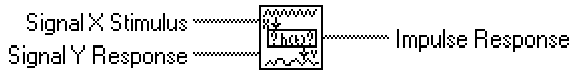
The following illustration shows an example using the Harmonic Analyzer VI.



Impulse Response Function (Advanced Only)

Computes the impulse response of a network based on real signals X (**Signal X Stimulus**) and Y (**Signal

Y Response).



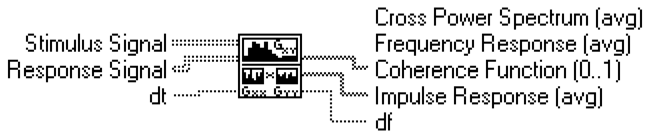
≤ **Signal X Stimulus** is a time-domain signal, usually the network stimulus.
≤ **Signal Y Response** is a time-domain signal, usually the network response.
≤ **Impulse Response** is the impulse response function computed from the averaged transfer function. This parameter is unitless.
The impulse response is in the time domain, so you do not need to convert time units to frequency units. The impulse response is the inverse transform of the transfer function.

This VI computes **Impulse Response** as

$$\text{Inverse FFT}\left[\frac{\text{Cross Power}\left(\text{Stimulus, Response}\right)}{\text{Power Spectrum}\left(\text{Stimulus}\right)}\right]$$

Network Functions (avg) (Advanced Only)

Computes several network response functions of two, real time-domain signals X (**Stimulus Signal**) and Y
(**Response Signal**).



≤ **Stimulus Signal** is an array of time-domain signals, usually the network stimulus.
≤ **Response Signal** is an array of time-domain signals, usually the network response.
≤ **dt** is the sample period of the time-domain signal, usually in seconds. **dt** is also
$\frac{1}{f_s}$ where
≤ is the sampled frequency of the time-domain signal.
Cross Power Spectrum contains the rms averaged, cross power spectrum.
≤ **Magnitude** is the single-sided, cross power spectrum between signals X and Y in volts rms squared if the input signals are in volts. If the input signals are not in volts, the results are in input signal units rms squared.
≤ **Phase** is the phase spectrum in radians showing the difference between the phases of signal X and signal Y.
≤ **Frequency Response** contains the transfer function computed from the rms averaged, cross power spectrum and auto power spectrums.
≤ **Magnitude** is the single-sided, frequency response specimen of the network. This is the averaged amplitude gain of the network.
≤ **Phase** is the phase spectrum in radians showing the difference between the phases of signal X and signal Y.
≤ **Impulse Response** is the impulse response function computed from the averaged transfer function. This parameter is unitless.
≤ **Coherence Function** is the single-sided, coherence function spectrum. This is unitless and ranges from 0 (no coherence) to 1 (complete coherence). The VI computes this value as

$$\frac{\left|\text{averaged }_{xy(f)}S\right|^2}{\left[\text{averaged } S_{xx}(f)\right]\left[\text{averaged } S_{yy}(f)\right]}$$

≤ **df** is the line frequency interval of the coherence function spectrum, in Hertz, if **dt** is in seconds. The signals X (**Stimulus Signal**) and Y (**Response Signal**) include coherence, averaged cross power spectrum magnitude and phase, averaged transfer function (frequency response), and averaged impulse response.

You usually compute these functions on the stimulus and response signals from a network under test. The coherence function shows the frequency content of the **Response Signal** Y due to **Stimulus Signal** X and measures the validity of the network frequency response measurement.

You can use this VI to measure the coherence between any two signals. The VI averages multiple stimulus and response signals to get valid coherence measurements. **Cross Power Spectrum** and **Impulse Response** are the rms averaged versions of the similarly named VIs. **Frequency Response** is the rms averaged version of the frequency response outputs of the Transfer Function VI.

Peak Detector (Advanced Only)

Finds the location, amplitude, and second derivative of peaks or valleys in the input array.

≤
≤ **X** is the input that holds the data to be processed. The data can be a single array or consecutive blocks of data. Consecutive blocks of data are useful for large, data arrays or for real time processing. Notice that in real time processing, **peaks/valleys** are not detected until approximately **width**/2 data points past the peak or valley.
≤ **threshold** is the input that rejects **peaks/valleys** that are too small. For peaks, any peak found with a fitted amplitude that is less than **threshold** is ignored. Valleys are ignored if the fitted trough is greater than **threshold**.
≤ **width** is the input that specifies the number of consecutive data points to use in the quadratic least squares fit. The value should be no more than about 1/2 of the half-width of the **peaks/valleys** and can be much smaller for noise-free data. Large widths can reduce the apparent amplitude of peaks and shift the apparent location.
≤ **peaks/valleys**. You use this control to choose between looking for peaks (positive-going bumps) and valleys (negative-going bumps). The settings for this control are 0 (peaks) and 1 (valleys).
≤ **initialize**. Set this control to TRUE to process the first block of data. The VI requires some internal setup at the beginning for proper operation. If you only want to process one block of data, leave **initialize** unwired, or set its default state to TRUE.If you want to process consecutive blocks of data, set **initialize** to TRUE for the first block and FALSE for all other blocks of data.
≤ **end of data**. Set this control to TRUE to process the last block of data. After processing the last block of data, the VI manages internal data. If you only want to process one block of data, leave **end of data** unwired, or set its default state to TRUE. If you want to process consecutive blocks of data, set **end of data** to FALSE for all but the last block of data.
≤ **Locations** is an array containing the locations of **peaks/valleys** found in the current block of data. **Locations** are reported in indices from the beginning of processing.
≤ **Amplitudes** is an array containing the amplitudes of **peaks/valleys** found in the current block of data.
≤ **2nd Derivatives** is an array containing the second derivatives of **peaks/valleys** found in the current block of data.
≤ **# found** is the number of **peaks/valleys** found in the current block of data. **# found** is the size of the arrays **Locations**, **Amplitudes**, and **2nd Derivatives**.
≤ **error**. See Analysis Error Codes for a description of the error.
The data set can be passed to the VI as a single array or as consecutive blocks of data.

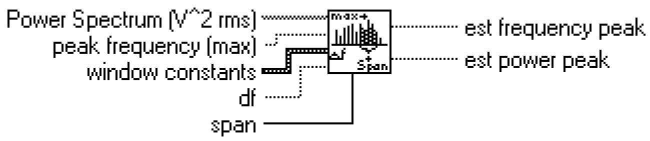
This VI is based on an algorithm that fits a quadratic polynomial to sequential groups of data points. The number of data points used in the fit is specified by **width**.

For each peak or valley, the quadratic fit is tested against the threshold level: peaks with heights lower than the threshold or valleys with troughs higher than the threshold are ignored. **peaks/valleys** are detected only after approximately **width**/2 data points have been processed beyond **peaks/valleys** locations. This delay has implications only for real time processing.

The VI must be notified when the first and last blocks are passed into the VI, so that the VI can initialize and then release data internal to the peak detection algorithm.

Power & Frequency Estimate (Advanced Only)

Computes the estimated power and frequency around a peak in the power spectrum of a time-domain signal.



≤ **Power Spectrum** is the power spectrum of a time domain signal (the output of the Auto Power Spectrum VI).

≤ **peak frequency** is the frequency (usually in Hertz) of the frequency peak around which you want to estimate the frequency and power. The default is -1. If you do not wire this parameter, the VI automatically searches for the maximum peak in the power spectrum array and estimates the frequency and power around it.

window constants contains the window constants of the window that was used to compute the power spectrum. This is usually the output of the Scaled Time Domain Window VI. The default values are set to those of the uniform window (no window).

≤ **eq noise BW** is the equivalent noise bandwidth (ENBW) of the selected window. You can use this value to divide a sum of individual power spectra of the power spectrum or to compute the power in a given frequency span. The **eq noise BW** defaults to 1.0.

≤ **coherent gain** is the inverse of the scaling factor that was applied to the window. The **coherent gain** defaults to 1.0.

≤ **df** is the line frequency interval of the input spectrum. You need this input only when you use the spectral density output formats (the last four display unit selections). **df** defaults to 1.0.

≤ **span** is the number of frequency lines around the peak to be included in the peak frequency and power estimation. The default is 7, which means that the power in three frequency lines before the peak frequency line, the peak frequency line itself, and three frequency lines after the peak are included in the estimation. This is adequate for most windows.

≤ **est frequency peak** is computed as

$$\text{Est Freq} = \frac{\Sigma\left(\text{Power Spectrum}(j)*(j*df)\right)}{\Sigma\left(\text{Power Spectrum}(j)\right)}$$

for j = i - span/2,...i + span/2

where i = peak index, Power Spectrum (j) = power in bin j, and **df** = frequency bin width.

≤ **est power peak** is computed as

$$\text{Est Freq} = \frac{\Sigma\left(\text{Power Spectrum}(j)\right)}{\text{ENBW}}$$

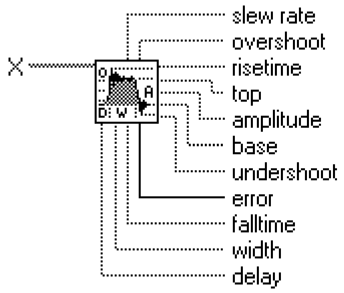
for j = i - span/2,...i + span/2

where i = peak index, Power Spectrum (j) = power in bin j, and ENBW = equivalent noise bandwidth of the window.

With this VI, you can achieve good frequency estimates for measured frequencies that lie between frequency lines on the spectrum. The VI makes corrections for the window function you use.

Pulse Parameters (Advanced Only)

Analyzes the input sequence **X** for a pulse pattern and determines the best set of pulse parameters that describes the pulse.



The waveform-related parameters are **slew rate**, **overshoot**, topline (**top**), **amplitude**, baseline (**base**), and **undershoot**. The time-related parameters are **risetime**, **falltime**, **width** (or duration), and **delay**.

≤ **X** must meet the following conditions.

- The number of samples in X must be greater than or equal to 3.
- **X** should have a rising edge, a plateau, and a falling edge.
- The expected peak noise amplitude must be less than 50% of the expected amplitude.

If the number of samples in **X** is less than 3, the VI sets all the pulse parameters to NaN and returns an error.

If **X** does not contain a rising edge, plateau, and falling edge, the VI analyzes the data, assigns values to as many pulse parameters as it can identify, and sets parameters it cannot identify to NaN. The VI does not report this condition as an error.

If the data in **X** is noisy beyond the expected 50% amplitude, the VI does not have enough information to differentiate between glitches and pulse data and may assign incorrect values to the pulse parameters. Because it cannot be detected, the VI does not report this condition as an error.

Note: If the data is noisy, you can apply a median filter to the data before passing it to the Pulse Parameters VI. See the Noisy Pulse Analyzed with a Median Filter section.

Special Case: When the **X** data is a constant value, c, the VI sets the pulse parameters to the following values.

amplitude = **overshoot** = **undershoot** = **delay** = **width** = 0.

top = **base** = c.

risetime = **falltime** = **slew rate** = NaN.

≤ **slew rate** is the ratio between (90% amplitude - 10% amplitude) and the risetime.
≤ **overshoot** is the difference between the maximum value in the pulse and the topline.
≤ **risetime** is the time required to rise from 10% amplitude to 90% amplitude on the rising edge of the pulse.
≤ **top** is the line that best represents the values when the pulse is active, high, or on.
≤ **amplitude** is the difference between the topline and the baseline.
≤ **base** is the line that best represents the values when the pulse is inactive, low, or off.
≤ **undershoot** is the difference between the baseline and the minimum value in the pulse.

≤ **error**. See Analysis Error Codes for a description of the error.
≤ **falltime** is the time required to fall from 90% amplitude to 10% amplitude on the falling edge of the pulse.
≤ **width** is the difference between the falling edge time and the rising edge time at which 50% amplitude occurs.
≤ **delay** is the difference between the time origin and the time at which 50% amplitude occurs on the rising edge of the pulse.
The VI uses the following steps to calculate the output parameters.

1. Find the maximum and minimum values in the input sequence **X**.
2. Generate the histogram of the pulse with 1% range resolution.
3. Determine the upper and lower modes to establish the **top** and **base** values.
4. Find the **overshoot**, **amplitude**, and **undershoot** from **top**, **base**, maximum, and minimum values.
5. Scan **X** and determine the **slew rate**, **risetime**, **falltime**, **width**, and **delay**.

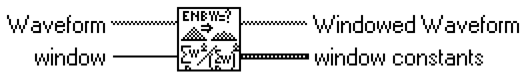
The VI interpolates **width** and **delay** to obtain a more accurate result not only of **width** and **delay,** but also of **slew rate**, **risetime**, and **falltime**.

If **X** contains a train of pulses, the VI uses the train to determine **overshoot**, **top**, **amplitude**, **base**, and **undershoot**, but uses only the first pulse in the train to establish **slew rate**, **risetime**, **falltime**, **width**, **and delay**.

Note: Because pulses commonly occur in the negative direction, this VI can discriminate between positive and negative pulses and can analyze the X sequence correctly. You do not need to preprocess the sequence before analyzing it.

Scaled Time Domain Window (Advanced Only)

Applies the selected window to the time-domain signal.



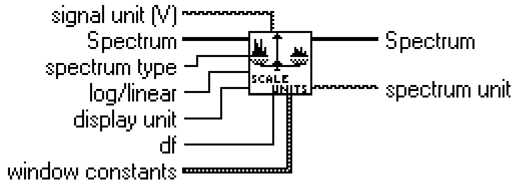
≤ **Waveform** is the time-domain signal.
≤ **window** is the time-domain window to be used.
0: Uniform
1: Hanning
2: Hamming
3: Blackman-Harris
4: Exact Blackman
5: Blackman
6: Flat Top
7: Four Term Blackman-Harris
8: Seven Term Blackman-Harris

≤ **Windowed Waveform** is the time-domain signal, multiplied by the scaled window.
≤ **window constants** contains the window constants for the selected window. The default values are set to those of the uniform window (no window).
≤ **eq noise BW** is the equivalent noise bandwidth of the selected window. You can use this value to divide a sum of individual power spectra of the power spectrum or to compute the power in a given frequency span.
≤ **coherent gain** is the inverse of the scaling factor that was applied to the window.
The VI scales the result so that when the power or amplitude spectrum of the windowed waveform is computed, all windows provide the same level within the accuracy constraints of the window. This VI also returns important window constants for the selected window. These constants are useful when you use

VIs that perform computations on the power spectrum, such as the Power & Frequency Estimate VI and Spectrum Unit Conversion VI.

Spectrum Unit Conversion (Advanced Only)

Converts either the power, amplitude, or gain (amplitude ratio) spectrum to alternate formats including Log (decibel and dbm) and spectral density.



≤ **Spectrum** is the spectrum of a time domain signal, and can be a power spectrum (Vrms^2), amplitude spectrum (Vrms), or gain (amplitude ratio).

≤ **spectrum type** specifies the type of spectrum wired to **Spectrum**.

0: power spectrum (Vrms^2)
1: amplitude spectrum (Vrms)
2: gain (amplitude ratio)

The default setting is 0: power spectrum.

≤ **log/linear** specifies linear or log spectrum output.

0: linear
1: dB
2: dBm

When you specify dB for power or amplitude spectrum input, the reference is 1 Vrms (dBV). For amplitude ratio input, the VI does not apply a reference. When you specify dBm for power or amplitude spectrum input, the reference is 0.78 Vrms. The default setting is 1.

≤ **display unit** is the output unit for the spectrum. You can choose one of the following output units.

0: Vrms volts rms
1: Vpk volts peak
2: Vrms^2 volts squared rms
3: Vpk^2 volts squared peak
4: Vrms/ÃHz volts rms per root Hz
5: Vpk/ÃHz volts peak per root Hz
6: Vrms^2/Hz volts squared rms per Hz
7: Vpk^2/Hz volts squared peak per Hz

The last four selections are amplitude spectral density (4,5) and power spectral density (6,7).

≤ **window constants** contains the window constants for the selected window (from the Scaled Time Domain Window VI). You need this input only when you use the spectral density output formats (the last four **display unit** selections). The default values are set to those of the uniform window (no window).

≤ **eq noise BW** is the equivalent noise bandwidth (ENBW) of the selected window. You can use this value to divide a sum of individual power spectra of the power spectrum or to compute the power in a given frequency span. The **eq noise BW** defaults to 1.0.

≤ **coherent gain** is the inverse of the scaling factor that was applied to the window. The **coherent gain** defaults to 1.0.

≤ **df** is the line frequency interval of the input spectrum. You DBLC this input only when you use the spectral density output formats (the last four display unit selections). **df** defaults to 1.0.

abc **signal unit** is a string containing the unit of the input time domain signal. If the original signal unit was in volts, then this input should contain the letter V for volts. The default setting is in volts.

≤ **Spectrum** is the output spectrum in the form specified by the log /linear and display unit inputs.
≤ **spectrum unit** is a string containing the unit of the output spectrum. If the output spectrum is in decibels form, the unit is prepended by dB.

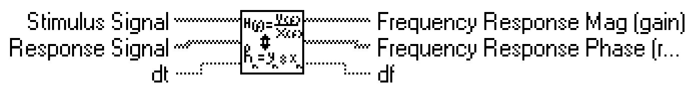
Threshold Peak Detector (Advanced Only)

Analyzes the input sequence **X** for valid peaks and keeps a **count** of the number of peaks encountered and a record of **Indices**, which locates the points that exceed the **threshold** in a valid peak. A peak is valid where the elements of X exceed the threshold and then return to a value less than or equal to the threshold, and the number of elements that exceed the **threshold** is at least equal to **width**.

≤
≤ **X**. The number of samples in **X** must be greater than the specified **width**. If **X** is less than or equal to **width**, the VI sets **count** to zero and returns an error.
≤ **threshold** defaults to 0.0.
≤ **width** must be greater than zero. If **width** is less than or equal to zero, the VI sets **count** to zero and returns an error. **width** defaults to 1.
≤ **Indices**.
≤ **count**.
≤ **error**. See Analysis Error Codes for a description of the error.

Transfer Function (Advanced Only)

Computes the transfer function (also known as the frequency response) from the time-domain Stimulus Signal and Response Signal from a network under test.



≤ **Stimulus Signal** is an array of time-domain signals, usually the network stimulus.
≤ **Response Signal** is an array of time-domain signals, usually the network response.
≤ **dt** is the sample period of the time-domain signal, usually in seconds. It is also
≤ where
≤ is the sampling frequency of the time-domain signal.
≤ **Frequency Response Mag** is the single-sided frequency response spectrum of the network. This is the amplitude gain of the network.
≤ **Frequency Response Phase** is the single-sided phase response spectrum of the network. This is the network phase in radians.
≤ **df** is the line frequency interval of the frequency response spectra, in Hertz, if dt is in seconds.
This VI computes the transfer function of a system based on the real signals X (**Stimulus Signal**) and Y (**Response Signal**). The output is the amplitude gain of the network, which is unitless.

The VI computer frequency response is:

$$\frac{\text{Cross Power (Stimulus,Response)}}{\text{Power Spectrum (Stimulus)}}$$

AC & DC Estimator.VI

AC & DC Estimator

Amplitude and Phase Spectrum.vi

Amplitude and Phase Spectrum

Auto Power Spectrum.vi

Auto Power Spectrum

Cross Power Spectrum.vi

Cross Power Spectrum

Harmonic Analyzer.vi

Harmonic Analyzer

Impulse Response Function.vi

Impulse Response Function

Network Functions (avg) .vi

Network Functions (avg)

Peak Detector.vi

Peak Detector

Power & Frequency Estimate.vi

Power & Frequency Estimate

Pulse Parameters.vi

Pulse Parameters

Scaled Time Domain Window.vi

Scaled Time Domain Window

Spectrum Unit Conversion.vi

Spectrum Unit Conversion

Threshold Peak Detector.vi

Threshold Peak Detector.vi

Transfer Function.vi

Transfer Function

Measurement VIs Overview

For detailed VI descriptions, see Measurement VIs.

Several measurement VIs 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 measurement VIs interact with VIs that perform such functions as scaled time domain windowing and power and frequency estimation.

You can use the measurement VIs for the following applications.

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

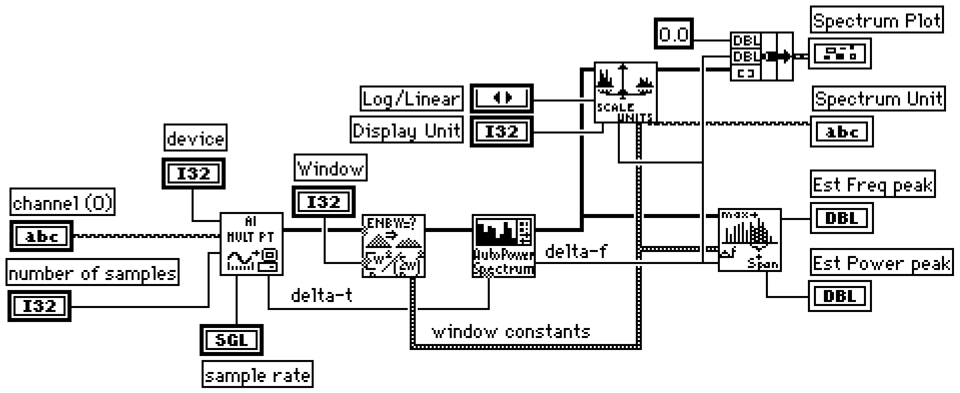
The DFT, FFT, and power spectrum are useful for measuring the frequency content of stationary or transient signals. The FFT provides the average frequency content of the signal over the entire time that the signal was acquired. For this reason, you use the FFT mostly for stationary signal analysis (when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

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

- Real-world, time-domain signal input is assumed.
- Outputs are in magnitude and phase, scaled, and in units where appropriate, ready for immediate graphing.
- Single-sided spectrums from DC to $\frac{\text{Sampling Frequency}}{2}$.
- Sampling period to frequency interval conversion for graphing with appropriate X-axis units (in Hertz).
- Corrections for the windows being used are applied where appropriate.
- Windows are scaled so that each window gives the same peak spectrum amplitude result within its amplitude accuracy constraints.

Views power or amplitude spectrums in various unit formats, including decibels and spectral density units $V^2/\text{Hz}, V/\sqrt{\text{Hz}}$, and so on.

In general, you can directly connect the measurement VIs to the output of data acquisition VIs and to graphs through the axis cluster, as the following spectrum analyzer diagram shows.



The following examples are included with the library.

- Amplitude Spectrum Example
- Simulated Dynamic Signed Analysis Example
- Total Harmonic Distortion (THD) Example

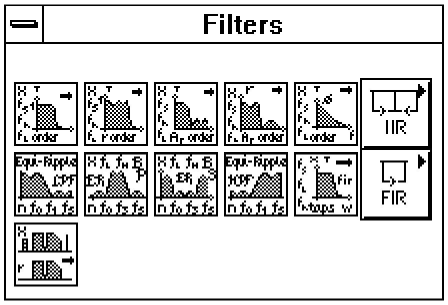
(Windows and Macintosh) You can use the following examples with National Instruments hardware.

- Simple Spectrum Analyzer and Spectrum Analyzer - both work with any analog input hardware (use dynamic signal acquisition hardware for good quality measurements).
- Dynamic Signal Analyzer and Network Analyzer - both work with dynamic signal acquisition hardware. The Network Analyzer requires the AT-DSP2200 board.

Filter VIs

This topic describes the VIs that implement IIR, FIR, and nonlinear filters. For general information about Filter VIs, see Digital Filtering VIs Overview .

The following illustration shows the options that are available on the **Filter** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Bessel Filter
Butterworth Filter
Chebyshev Filter
Elliptic Filter
Equi-Ripple BandPass
Equi-Ripple BandStop
Equi-Ripple HighPass
Equi-Ripple LowPass
FIR Windowed Filter
Inverse Chebyshev Filter
Median Filter

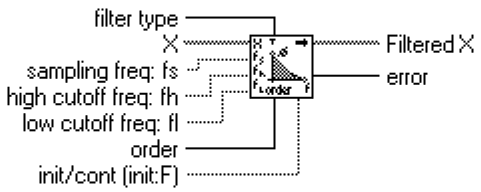
Subpalettes

Advanced IIR Filtering
Advanced FIR Filtering

For examples of how to use the filter VIs, see the examples located in `examples\analysis\fltrxmpl.llb`.

Bessel Filter (Advanced Only)

Generates a digital, Bessel filter using the filter type, sampling frequency, high cutoff frequency, low cutoff frequency, and order by calling the Bessel Coefficients VI. The VI then calls the IIR filter to filter the **X** sequence using this model to obtain a Bessel **Filtered X** sequence.



filter type specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass

3: Bandstop

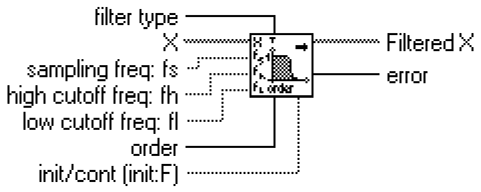
≤ **X** is the input signal to be filtered.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fl** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
$0 \leq f_1 \leq 0.5f_s$

where ≤is the cutoff frequency, and

≤ is the sampling frequency. If **low cutoff freq : fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered X** to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.
≤ **order** must be greater than zero. If **order** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **order** defaults to 2.
≤ **init/cont** controls the initialization of the internal states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.

Butterworth Filter (Advanced Only)

Generates a digital Butterworth filter using the sampling frequency, low cutoff frequency, high cutoff frequency, order, and filter type by calling the Butterworth Coefficients VI. The Butterworth Filter VI then calls the IIR Filter VI to filter the **X** sequence using this model to get a Butterworth **Filtered X** sequence.



≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **X** is the input signal to be filtered.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤,
where ≤is the cutoff frequency and
≤ is the sampling frequency. If **low cutoff freq: fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered X** to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.

≤ **order** must be greater than zero. If **order** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **order** defaults to 2.

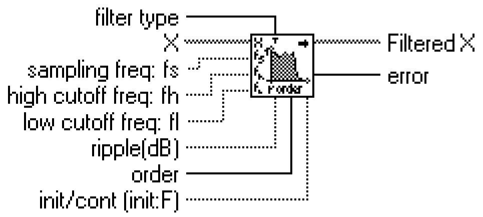
≤ **init/cont** controls the initialization of the internal states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.

≤ **Filtered X** is the output array of filtered samples.

≤ **error**. See Analysis Error Codes for a description of the error.

Chebyshev Filter (Advanced Only)

Generates a digital, Chebyshev filter using the sampling frequency, lower cutoff frequency, upper cutoff frequency, ripple, order, and filter type by calling the Chebyshev Coefficients VI. The Chebyshev Filter VI filters the **X** sequence using this model to obtain a Chebyshev **Filtered X** sequence by calling the IIR Filter VI.



≤ **filter type** specifies the passband of the filter according to the following values.

0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **X** is the input signal to be filtered.

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.

≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).

≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion

≤,

where ≤ is the cutoff frequency, and

≤ is the sampling frequency. If **low cutoff freq : fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered X** to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.

≤ **ripple** is the ripple in the passband. **ripple** must be greater than zero, and you must express it in decibels. If **ripple** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **ripple** defaults to 0.1.

≤ **order** must be greater than zero. If **order** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **order** defaults to 2.

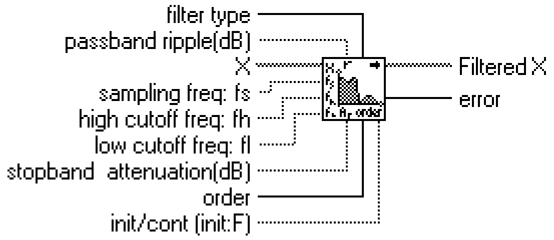
≤ **init/cont** controls the initialization of the internal states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.

≤ **Filtered X** is the output array of filtered samples.

≤ **error**. See Analysis Error Codes for a description of the error.

Elliptic Filter (Advanced Only)

Generates a digital, elliptic filter using the sampling frequency, lower cutoff frequency, upper cutoff frequency, **filter type**, **passband ripple**, **stopband attenuation**, and order by calling the Elliptic Coefficients VI. The Elliptic Filter VI then calls the IIR Filter VI to filter the **X** sequence using this model to obtain an elliptic **Filtered X** sequence.

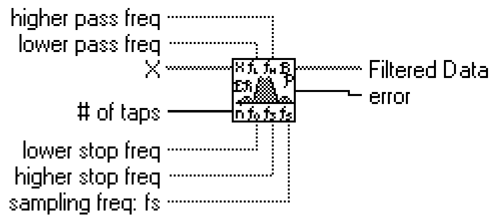


≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **passband ripple** is the ripple in the passband. **ripple** must be greater than zero, and you must express it in decibels. If **passband rippl**e is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **passband ripple** defaults to 1.0.
≤ **X** is the input signal to be filtered.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency, and
≤ is the sampling frequency. If **low cutoff freq : fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered X** to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.
≤ **stopband attenuation** is the attenuation in the stopband. **stopband attenuation** must be greater than zero and you must express it in decibels. If **stopband attenuation** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **stopband attenuation** defaults to 60.0.
≤ **order** is the order of the IIR filter and must be greater than zero. If order is less than or equal to zero, the VI sets Filtered X to an empty array and returns an error. order defaults to 2.0.
≤ **init/cont** controls the initialization of the internal states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.

Equi-Ripple BandPass (Advanced Only)

Generates a bandpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and the higher pass frequency, lower pass frequency**, # of taps, lower stop frequency, higher stop frequency, and sampling frequency.** The VI then filters the input sequence **X** to obtain the bandpass, filtered, linear-phase sequence **Filtered Data**.



≤ **higher pass freq** must be greater than **lower pass freq** frequency. If **higher pass freq** is less than or equal to **lower pass freq**, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **higher pass freq** defaults to 0.35.

≤ **lower pass freq** must be greater than the **lower stop freq**. If **lower pass freq** is less than or equal to **lower stop freq**, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **lower pass freq** defaults to 0.25.

≤ **X** is the input signal to be filtered.

≤ **# of taps** must be greater than zero. If # of taps is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. # **of taps** defaults to 32.

≤ **lower stop freq** must be greater than zero. If **lower stop freq** is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **lower stop freq** defaults to 0.20.

≤ **higher stop freq** must be greater than **higher pass freq** and must observe the Nyquist criterion:

$$0 \le f_0 \le f_1 \le f_2 \le f_3 \le 0.5f_s,$$

where ≤is **lower stop freq**,
≤is **lower pass freq**,
≤ is **higher pass freq**,
f_3 is the **higher stop freq**, and
≤is the sampling frequency. If any of these conditions are violated, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **higher stop freq** defaults to 0.4.

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.

≤ **Filtered Data**. Because the VI filters via convolution, the number of elements, k, in **Filtered Data** is

$$k = n + m - 1,$$

where n is the number of elements in **X**, and m is the number of taps.

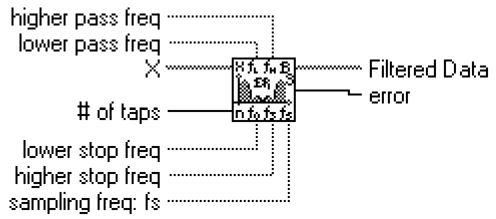
A delay is also associated with the output sequence delay $= \frac{m-1}{2}$.

≤ **error**. See Analysis Error Codes for a description of the error.

The first stopband of the filter region goes from zero (DC) to the lower stop frequency. The passband region goes from the lower pass frequency to the higher pass frequency, and the second stopband region goes from the higher stop frequency to the Nyquist frequency.

Equi-Ripple BandStop (Advanced Only)

Generates a bandstop FIR digital filter with equi-ripple characteristics using the Parks-McClellan algorithm and higher pass frequency, lower pass frequency**, # of taps, lower stop frequency, higher stop frequency, and sampling frequency**.The VI then filters the input sequence **X** to obtain the bandstop, filtered, linear-phase sequence **Filtered Data**.



≤ **higher pass freq** must be greater than **higher stop freq** and observe the Nyquist criterion ≤,
where ≤ is the lower pass frequency,
≤ is the lower stop frequency,
≤is the higher stop frequency,
≤ is the higher pass frequency, and
≤ is the sampling frequency. If any of these conditions are violated, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **higher pass freq** defaults to 0.4.
≤ **lower pass freq** must be greater than zero. If **lower pass freq** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error via the Parks-McClellan VI. **lower pass freq** defaults to 0.2.
≤ **X** is the input signal to be filtered.
≤ **# of taps** must be greater than zero. If the number of taps is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. The VI does not place restrictions on **# of taps**, but **# of taps** should be odd. **# of taps** defaults to 31.

Note: The Parks-McClellan algorithm introduces a large error when you design a bandstop filter for an even number of taps. To avoid this error, the Equi-Ripple BandStop VI adjusts the number of taps to the next higher odd value if # of taps is even.

≤ **lower stop freq** must be greater than **lower pass freq**. If **lower stop freq** is less than or equal to **lower pass freq**, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **lower stop freq** defaults to 0.25.
≤ **higher stop freq** must be greater than **lower stop freq**. If **higher stop freq** is less than or equal to **lower stop freq**, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **higher stop freq** defaults to 0.35.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.
≤ **Filtered Data**. Because the VI filters via convolution, the number of elements, k, in **Filtered Data** is

$$k = n + m - 1,$$

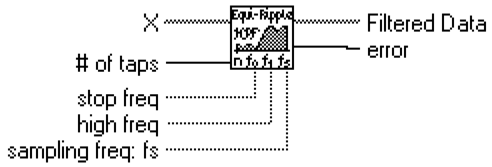
where n is the number of elements in **X**, and m is the number of taps.
A delay is also associated with the output sequence,

$$\text{delay} = \frac{m-1}{2}$$

≤ **error**. See Analysis Error Codes for a description of the error.
The first passband region of the filter goes from zero (DC) to the lower pass frequency. The stopband region goes from the lower stop frequency to the higher stop frequency, and the second passband region goes from the higher pass frequency to the Nyquist frequency.

Equi-Ripple HighPass (Advanced Only)

Generates a highpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and **the # of taps, stop frequency, high frequency, and sampling frequency**. The VI then filters the input sequence **X** to obtain the highpass, filtered, linear-phase sequence **Filtered Data**.



≤ **X** is the input signal to be filtered.
≤ **# of taps** must be greater than zero. If the number of taps is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. The VI does not place restrictions on the **# of taps**, but **# of taps** should be odd. **# of taps** defaults to 31.

Note: The Parks-McClellan algorithm introduces a large error when designing a highpass filter for an even number of taps. To avoid this error, the Equi-Ripple HighPass VI adjusts the number of taps to the next higher odd value if # of taps is even.

≤ **stop freq** must be greater than zero. If **stop freq** is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **stop freq** defaults to 0.2.

≤ **high freq** must be greater than **stop freq** and observe the Nyquist criterion

≤,

where ≤ is the **stop freq**,

≤ is the **high freq**, and

≤ is the sampling frequency. If any of these conditions are violated, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **high freq** defaults to 0.3.

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.

≤ **Filtered Data**. Because the VI filters via convolution, the number of elements, *k*, in **Filtered Data** is

$$k = n + m - 1,$$

where n is the number of elements in **X**, and m is the adjusted number of taps.

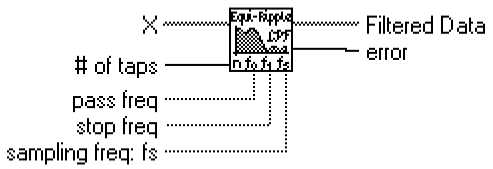
A delay equal to is also associated with the output sequence.

≤ **error**. See Analysis Error Codes for a description of the error.

The stopband of the filter goes from zero (DC) to the stop frequency. The transition band goes from the stop frequency to the high frequency, and the passband goes from the high frequency to the Nyquist frequency.

Equi-Ripple LowPass (Advanced Only)

Generates a lowpass FIR filter with equi-ripple characteristics using the Parks-McClellan algorithm and the **# of taps, pass frequency, stop frequency, and sampling frequency**. The VI then filters the input sequence **X** to obtain the lowpass filtered, linear-phase sequence **Filtered Data**.



≤ **X** is the input signal to be filtered.

≤ **# of taps** must be greater than zero. If the number of taps is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **# of taps** defaults to 32.

≤ **pass freq** must be greater than zero. If **pass freq** is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **pass freq** defaults to 0.2.

≤ **stop freq** must be greater than the **pass freq** and observe the Nyquist criterion

≤,

where ≤ is the **pass freq**,

≤ is the **stop freq**, and

≤ is the sampling frequency.

If any of these conditions are not met, the VI sets **Filtered Data** to an empty array and returns an error via the Parks-McClellan VI. **stop freq** defaults to 0.3.

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.

≤ **Filtered Data**. Because the VI filters via convolution, the number of elements, k, in **Filtered Data** is

$$k = n + m - 1,$$

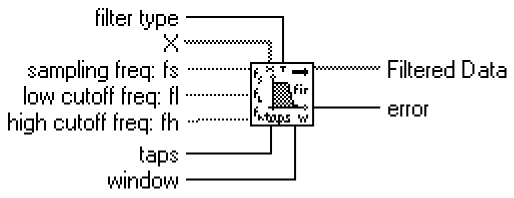
where n is the number of elements in **X**, and m is the number of taps.

A delay is also associated with the output sequence $\text{delay} = \frac{m-1}{2}$.

≤ **error**. See Analysis Error Codes for a description of the error.
The passband of the filter goes from zero (DC) to **pass freq**. The transition band goes from **pass freq** to **stop freq**, and the stopband goes from **stop freq** to the Nyquist frequency.

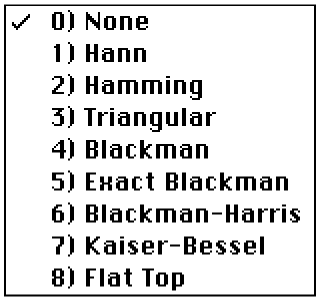
FIR Windowed Filter (Advanced Only)

Filters the input data sequence, **X**, using the set of windowed FIR filter coefficients specified by the sampling frequency, cutoff frequency, and number of **taps**.



≤ **X** is the input signal to be filtered.
≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling frequency: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error. **sampling frequency : fs** defaults to 1.0.
≤ **low cutoff frequency: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency, and
≤ is the sampling frequency. If **low cutoff frequency: fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered Data** to an empty array and returns an error. **low cutoff frequency: fl** defaults to 0.125.
≤ **high cutoff frequency: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **taps** determines the total number of FIR coefficients and must be greater than zero. If **taps** is less than or equal to 0, the VI sets **FIR Windowed Filter** to an empty array and returns an error. **taps** must be odd for highpass and bandstop filters. **taps** defaults to 25.
≤ **window**. With the **window** control you can select the following smoothing window options.


✓ 0) None
1) Hann
2) Hamming
3) Triangular
4) Blackman
5) Exact Blackman
6) Blackman-Harris
7) Kaiser-Bessel
8) Flat Top

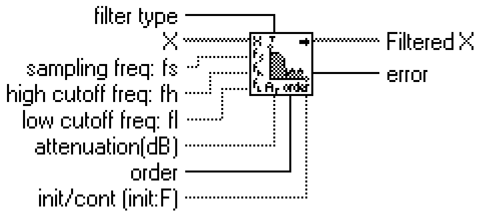
≤ **Filtered Data**.
Note: Filtered Data has an associated index delay caused by the convolution operation. The

delay is given by $\mathrm{delay} = \dfrac{\mathrm{taps} - 1}{2}$ **.**

≤ **error**. See Analysis Error Codes for a description of the error.

Inverse Chebyshev Filter (Advanced Only)

Generates a digital, Chebyshev II filter using the specified sampling frequency, cutoff frequencies, **attenuation** in decibels, **filter type**, and filter **order** by calling the Inv Chebyshev Coefficients VI. The Inverse Chebyshev Filter VI filters the **X** sequence using this model to obtain a Chebyshev II **Filtered X** sequence by calling the IIR Filter VI.

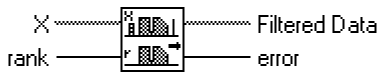


≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **X** is the input signal to be filtered.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency and
≤ is the sampling frequency. If **low cutoff freq : fl** is less than zero or greater than half the sampling frequency, the VI sets **Filtered X** to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.
≤ **attenuation** is the attenuation in the stopband. attenuation must be greater than zero, and you must express it in decibels. If **attenuation** is less than or equal to zero, the VI sets **Filtered X** to an empty array and returns an error. **attenuation** defaults to 60.0.
≤ **order** must be greater than zero. If **order** is less than or equal to zero, the VI sets **Filtered Data** to an empty array and returns an error. **order** defaults to 2.0.
≤ **init/cont** controls the initialization of the internal states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.

Median Filter (Advanced Only)

Applies a median filter of **rank** to the input sequence **X**.



If Y represents the output sequence **Filtered Data**, and if J_i represents a subset of the input sequence **X** centered about the

≤element of **x**

$$J_i = \{x_{i-r}, x_{i-r+1} \dots, x_{i-1}, x_i, x_{i+1} \dots, x_{i+r-1}, x_{i+r}\},$$

and if the indexed elements outside the range of **X** equal zero, the VI obtains the elements of Y using

$$y_i = \mathrm{Median}(J_i) \qquad \text{for } i = 0, 1, 2, \dots, n-1,$$

where n is the number of elements in the input sequence **X**, and r is the filter **rank**.

≤ **X** is the input signal to be filtered. The number of elements in **X** must be greater than the **rank**, $n > r \geq 0$.

If the number of elements in **X** is less than or equal to **rank**, the VI sets the output **Filtered Data** to an empty array and returns an error.

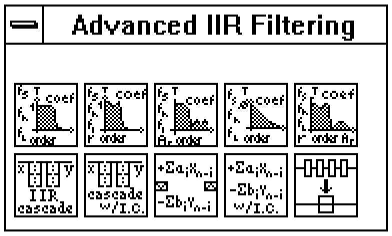
≤ **rank** must be greater than or equal to zero. If **rank** is less than zero, the VI sets the output **Filtered Data** to an empty array and returns an error. **rank** defaults to 2.

≤ **Filtered Data**.

≤ **error**. See Analysis Error Codes for a description of the error

Advanced IIR Filtering

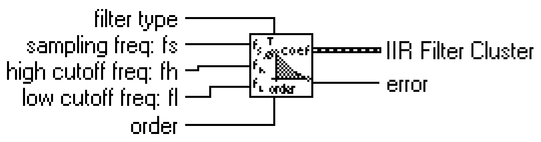
For general information about Advanced FIR Filtering VIs, Digital Filtering VIs Overview. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Bessel Coefficients
Butterworth Coefficients
Cascade>Direct Coefficients
Chebyshev Coefficients
Elliptic Coefficients
IIR Cascade Filter
IIR Cascade Filter with I.C.
IIR Filter
IIR Filter with I.C.
Inverse Chebyshev Coefficients

Bessel Coefficients (Advanced Only)

Generates the set of filter coefficients to implement an IIR filter as specified by the Bessel filter model. You can then pass these coefficients to the IIR Filter VI.



The Bessel Coefficients VI is a subVI of the Bessel Filter VI.

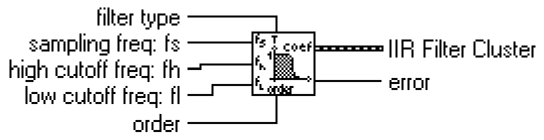
≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency, and
≤ is the sampling frequency.
≤ **order** is the order of the IIR filter and must be greater than zero.
≤ **IIR Filter Cluster** contains three elements.
filter structure indicates either IIR second-order or IIR fourth-order filter stages.

≤ **Reverse Coefficients** of the IIR cascade filter.
≤ **Forward Coefficients** of the IIR cascade filter.
≤ **error**. See Analysis Error Codes for a description of the error

Butterworth Coefficients (Advanced Only)

Generates the set of filter coefficients to implement an IIR filter as specified by the Butterworth filter model. You can pass these filter coefficients (IIR Filter Cluster) to the IIR Cascade Filter VI to filter a sequence of data.



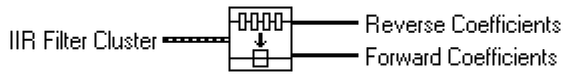
The Butterworth Coefficients VI is a subVI of the Butterworth Filter VI.

≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency, and
≤ is the sampling frequency.
≤ **order** is the order of the IIR filter and must be greater than zero.
≤ **IIR Filter Cluster** contains three elements.
≤ **filter structure** indicates either IIR second-order or IIR fourth-order filter stages.
≤ **Reverse Coefficients** for the IIR cascade filter.
≤ **Forward Coefficients** for the IIR cascade filter.
≤ **error**. See Analysis Error Codes for a description of the error.

Cascade >Direct Coefficients (Advanced Only)

Converts IIR filter coefficients from the cascade form to the direct form.



≤ **IIR Filter Cluster** contains the cascade form of IIR filter coefficients. It contains three elements.
filter structure selects IIR second-order or IIR fourth-order filter stages
≤ **Reverse Coefficients** of the IIR cascade filter.
≤ **Forward Coefficients** of the IIR cascade filter.
This cluster is the output from one of the IIR coefficient design VIs: Butterworth Coefficients, Bessel Coefficients, Chebyshev Coefficients, Elliptic Coefficients, or Inv Chebyshev Coefficients.

≤ **Forward Coefficients** contain the direct form, forward coefficients.
≤ **Reverse Coefficients** contain the direct form, reverse coefficients.
As an example, you can convert a cascade filter, composed of two second-order stages, to a direct form filter as follows.

Reverse Coefficients:

$\{a_{11}, a_{21}, a_{12}, a_{22}\} \rightarrow \{1.0, a_1, a_2, a_3, a_4\}$

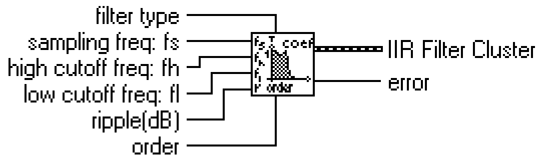
Forward Coefficients:

≤

See the IIR Cascade Filter VI for information on cascade form filtering, the IIR Filter VI for information on direct form filtering, and the Digital Filtering VIs Overview section for a discussion of both filter forms.

Chebyshev Coefficients (Advanced Only)

Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev filter model. You can pass these coefficients to the IIR Filter VI to filter a sequence of data.



The Chebyshev Coefficients VI is a subVI of the Chebyshev Filter VI.

≤ **filter type** specifies the passband of the filter according to the following values.

0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.

≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).

≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion

≤

where ≤ is the cutoff frequency and

≤ is the sampling frequency.

≤ **ripple** is the ripple in the passband. ripple must be greater than zero, and you must express it in decibels. **ripple** defaults to 0.1.

≤ **order** is the order of the IIR filter and must be greater than zero.

≤ **IIR Filter Cluster** contains three elements.

≤ **filter structure** indicates either IIR second-order or IIR fourth-order filter stages.

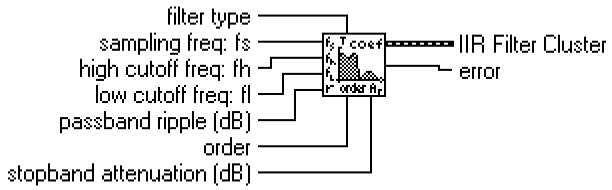
≤ **Reverse Coefficients** for the IIR cascade filter.

≤ **Forward Coefficients** for the IIR cascade filter.

≤ **error**. See Analysis Error Codes for a description of the error

Elliptic Coefficients (Advanced Only)

Generates the set of filter coefficients to implement a digital elliptic IIR filter. You can pass these coefficients to the IIR Filter VI.



The Elliptic Coefficients VI is a subVI of the Elliptic Filter VI.

≤ **filter type** specifies the passband of the filter according to the following values.

0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

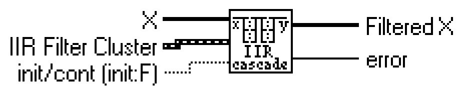
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If **sampling freq: fs** is less than or equal to zero, the VI sets IIR Filter Cluster to an empty array and returns an error. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤

where ≤ is the cutoff frequency, and

≤ is the sampling frequency. If **low cutoff freq: fl** is less than zero or greater than half the sampling frequency, the VI sets IIR Filter Cluster to an empty array and returns an error. **low cutoff freq: fl** defaults to 0.125.
≤ **passband ripple** is the ripple in the passband. **ripple** must be greater than zero, and you must express it in decibels. If **passband rippl**e is less than or equal to zero, the VI sets **IIR Filter Cluster** to an empty array and returns an error. **passband ripple** defaults to 1.0.
≤ **order** is the order of the IIR filter and must be greater than zero. If order is less than or equal to zero, the VI sets IIR Filter Cluster to an empty array and returns an error. order defaults to 2.0.
≤ **stopband attenuation** is the attenuation in the stopband. **stopband attenuation** must be greater than zero and you must express it in decibels. If **stopband attenuation** is less than or equal to zero, the VI sets **IIR Filter Cluster** to an empty array and returns an error. **stopband attenuation** defaults to 60.0.
≤ **IIR Filter Cluster** contains three elements.
≤ **filter structure** indicates either IIR second-order or IIR fourth-order filter stages.
≤ **Reverse Coefficients** for the IIR cascade filter.
≤ **Forward Coefficients** for the IIR cascade filter.
≤ **error**. See Analysis Error Codes for a description of the error.

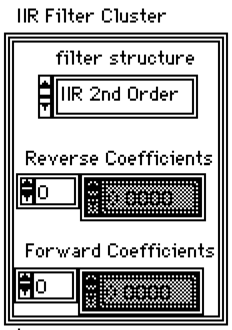
IIR Cascade Filter (Advanced Only)

Filters the input sequence X using the cascade form of the IIR filter specified by the IIR Filter Cluster.



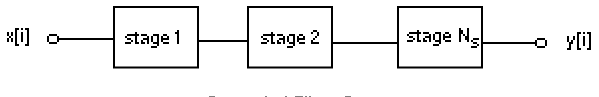
Second-Order Filtering
Fourth-Order Filtering

≤ **X** is the input array of samples to be filtered.
≤ **IIR Filter Cluster** contains three elements.



≤ **filter structure** selects IIR second-order or IIR fourth-order filter stages.
≤ **Reverse Coefficients** of the IIR cascade filter.

≤ **Forward Coefficients** of the IIR cascade filter.
≤ **init/cont** controls the initialization of the internal filter states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.
This IIR implementation is called cascade because it is a cascade of second or fourth-order filter stages. The output of one filter stage is the input to the next filter stage for all Ns filter stages:



Cascaded Filter Stages

Second-Order Filtering

Each second-order stage (stage number $k = 1,2,\ldots,N_s$) has two reverse coefficients (a_{1k}, a_{2k}) and three forward coefficients (b_{0k}, b_{1k}, b_{2k}). The total number of reverse coefficients is 2Ns and the total number of forward coefficients is 3Ns. The Reverse Coefficients and the Forward Coefficients array contain the coefficients for one stage followed by the coefficients for the next stage, and so on. For example, an IIR filter composed of two second-order stages must have a total of four reverse coefficients and six forward coefficients, as follows:

$$\text{Reverse Coefficients} = \{a_{11}, a_{21}, a_{22}\}$$

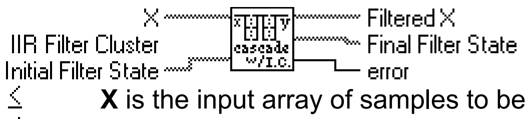
$$\text{Forward Coefficients} = \{b_{01}, b_{11}, b_{21}, b_{02}, b_{12}, b_{22}\}$$

Fourth-Order Filtering

For fourth order cascade stages, the filtering is implemented in the same manner as in the second-order stages, but each stage must have four reverse coefficients (a1k...a4k) and five forward coefficients $\{b_{0k} \ldots, b_{4k}\}$.

IIR Cascade Filter with I.C. (Advanced Only)

Filters the input sequence X using the cascade form of the IIR filter specified by the IIR Filter Cluster.

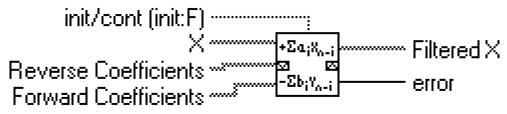


≤ **X** is the input array of samples to be filtered.
≤ **IIR Filter Cluster** contains three elements.
≤
≤ **filter structure** selects IIR second-order or IIR fourth-order filter stages.
≤ **Reverse Coefficients** of the IIR cascade filter.
≤ **Forward Coefficients** of the IIR cascade filter.
≤ **Initial Filter State**. This array should be the same size as the Reverse Coefficients array in the **IIR Filter Cluster**. If this array is not the correct size (empty, for example), the array is resized internally and is initialized to zero before the IIR filtering operation.
The filtering occurs internally and the filter state is maintained until all samples in array X have

been passed through the filter. The final filter state array is then passed to the array indicator, **Final Filter State**.
≤ **Filtered X** is the output array of filtered samples.
≤ **Final Filter State** contains the final, internal filter states, which can be passed as the **Initial Filter State** to the next call to the IIR Cascade Filter with I.C. to filter samples continuously.
≤ **error**. See Analysis Error Codes for a description of the error.

IIR Filter (Advanced Only)

Filters the input sequence **X** using the direct form IIR filter specified by **Reverse Coefficients** and **Forward Coefficients**.



≤ **init/cont** controls the initialization of the internal filter states. When **init/cont** is FALSE (default), the internal states are initialized to zero. When **init/cont** is TRUE, the internal filter states are initialized to the final filter states from the previous call to this instance of this VI. To filter a large data sequence that has been split into smaller blocks, set this control to FALSE for the first block, and to TRUE for continuous filtering of all remaining blocks.
≤ **X** is the input array of samples to be filtered.
≤ **Reverse Coefficients**. This VI does not place any restrictions on the coefficient arrays. If both coefficient arrays are empty, the VI performs no filtering and sets **Filtered X** to the value of **X**.

Note: You can use the IIR Filter VI to perform FIR filtering by passing an empty array into Reverse Coefficients.

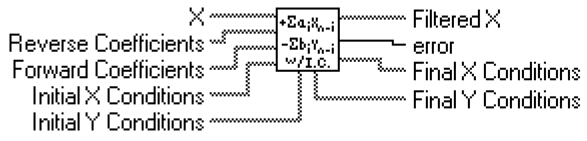
≤ **Forward Coefficients**.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.
If Y represents the output sequence **Filtered X**, the VI obtains the elements of Y using

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

where n is the number of **Forward Coefficients** (represented by b_j), and m is the number of **Reverse Coefficients** (represented by
a_k).

IIR Filter with I.C. (Advanced Only)

Filters the input sequence **X** using the direct form IIR filter specified by **Reverse Coefficients** and **Forward Coefficients**.



≤ **X** is the input array of samples to be filtered.
≤ **Reverse Coefficients**. This VI does not place any restrictions on the coefficient arrays. If both coefficient arrays are empty, the VI performs no filtering and sets **Y** to the value of **X**.

Note: You can use the IIR Filter VI to perform FIR filtering by passing an empty array into Reverse Coefficients or by leaving Reverse Coefficients unwired.

≤ **Forward Coefficients**.
≤ **Initial X Conditions** contains the initial conditions for the input array **X**. The most recent prior

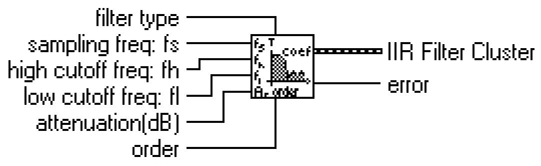
input should be the last element in the array. The number of elements in this array should be one less than the number of elements in the Forward Coefficients array.
≤ **Initial Y Conditions** contains the initial conditions for the output array **Y**. The most recent output should be the last element in the array. The number of elements in this array should be one less than the number of elements in the Reverse Coefficients array.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.
≤ **Final X Conditions** contains the most recent inputs that may be used as **Initial X Conditions** on the next call to this VI.
≤ **Final Y Conditions** contains the most recent outputs that may be used as **Initial Y Conditions** on the next call to this VI.
If *Y* represents the output sequence **Filtered X**, the VI obtains the elements of *Y* using

$$y_i = \frac{1}{a_0}\left(\sum_{j=0}^{n-1} b_j x_{i-j} - \sum_{k=1}^{m-1} a_k y_{i-k}\right),$$

where n is the number of **Forward Coefficients** (represented by ≤), and m is the number of **Reverse Coefficients** (represented by
≤).

Inverse Chebyshev Coefficients (Advanced Only)

Generates the set of filter coefficients to implement an IIR filter as specified by the Chebyshev II Filter model. You can pass these coefficients to the IIR Filter VI to filter a sequence of data.



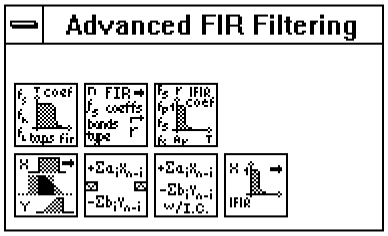
The Inv Chebyshev Coefficients VI is a subVI of the Inverse Chebyshev Filter VI.

≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.
≤ **high cutoff freq: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff freq: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤
where ≤ is the cutoff frequency and
≤ is the sampling frequency.
≤ **attenuation** is the attenuation in the stopband. attenuation must be greater than zero, and you must express it in decibels. **attenuation** defaults to 60.0.
≤ **order** is the order of the IIR filter and must be greater than zero.
≤ **IIR Filter Cluster** contains three elements.
≤ **filter structure** indicates either IIR second-order or IIR fourth-order filter stages.
≤ **Reverse Coefficients** for the IIR cascade filter.
≤ **Forward Coefficients** for the IIR cascade filter.
≤ **error**. See Analysis Error Codes for a description of the error.

Advanced FIR Filtering

For general information about Advanced FIR Filtering VIs, Digital Filtering VIs Overview. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Convolution
FIR Narrowband Coefficients
FIR Narrowband Filter
FIR Windowed Coefficients
Parks-McClellan

Convolution (Advanced Only)

Computes the convolution of the input sequences **X** and **Y**.

≤
≤ **X**.
≤ **Y**.
≤ **X * Y**. The convolution of **X** and **Y**.
≤ **error**. See Analysis Error Codes for a description of the error.

The convolution $h(t)$, of the signals $x(t)$ and $y(t)$ is defined as

$$h(t) = x(t) * y(t) = \int_{-\infty}^{\infty} x(\tau)y(t-\tau)d\tau$$

where the symbol $*$ denotes convolution.

For the discrete implementation of the convolution, let h represent the output sequence **X * Y**, let n be the number of elements in the input sequence **X**, and let m be the number of elements in the input sequence **Y**. Assuming that indexed elements of **X** and **Y** that lie outside their range are zero,

$$x_i = 0, \ i < 0 \ \text{or} \ i \geq n$$

and

$$yj = 0, \ j < 0 \ \text{or} \ j \geq m$$

then you obtain the elements of h using

$$h_j = \sum_{k=0}^{n-1} x_k y_{i-k}$$

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

size = n + m - 1,

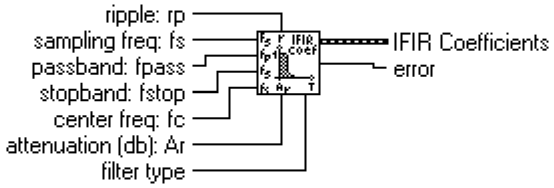
where size denotes the total number of elements in the output sequence **X * Y**.

Note: This is not a circular convolution. Because x(t) * Y(t) ≤X(f) Y(f) is a Fourier transform pair, you can create a circular version of the convolution using a diagram similar to the following diagram.

≤

FIR Narrowband Coefficients (Advanced Only)

Generates a set of filter coefficients to implement a digital interpolated FIR filter. You can pass these coefficients to the FIR Narrowband Filter VI to filter the data.



≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets the coefficients to an empty cluster and returns an error. **sampling freq: fs** defaults to 1.0.

≤ **passband: fpass** is the passband bandwidth. See figures 4-1 through 4-4 for the definition of different filters. **passband: fpass** defaults to 0.01.

≤ **stopband: fstop** is the stopband bandwidth. See figures 4-1 through 4-4 for the definition of different filters. **stopband: fstop** defaults to 0.02.

≤ **center freq: fc** is the center frequency of the filter. See figures 4-1 through 4-4 for the definition of different filters. **center freq: fc** defaults to 0.1.

≤ **attenuation: Ar** is the attenuation in the stopband of the filter. See figures 4-1 through 4-4 for the definition. **attenuation: Ar** defaults to 60 decibels.

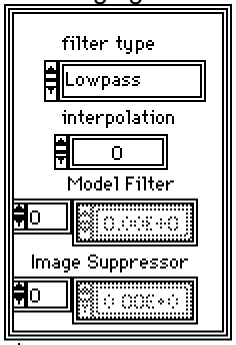
≤ **ripple: rp** is the ripple in the passband of the filter. See figures 4-1 through 4-4 for the definition. **ripple: rp** defaults to 0.01.

≤ **filter type** specifies the passband of the filter according to the following values.

0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

filter type defaults to lowpass.

≤ **IFIR Coefficients** is a cluster that contains IFIR coefficients. It has four elements, as shown in the following figure.



≤ **filter type** is the filter type that you use to determine how to filter the data.

0: Lowpass
1: Highpass

2: Bandpass
3: Bandstop
4: Wideband-Lowpass *
5: Wideband-Highpass **
* cutoff frequencies near Nyquist
** cutoff frequencies near zero

≤ **interpolation** is the interpolation factor M. The model filter is stretched by **interpolation** times.
≤ **Model Filter** contains the coefficients of the model filter.
≤ **Image Suppressor** contains the coefficients of the filter image suppressor.
≤ **error**. See Analysis Error Codes for a description of the error.

The following figures show how the narrowband filter parameters define the lowpass, highpass, bandpass, and bandstop filters. The filter response on the y axis is shown on a linear scale. For this reason, the stopband attenuation Ar was mapped to a linear attenuation using the following equations:

$$A_r = -20\log\langle\delta_A\rangle$$

$$\delta_A = 10\frac{-Ar}{20}.$$

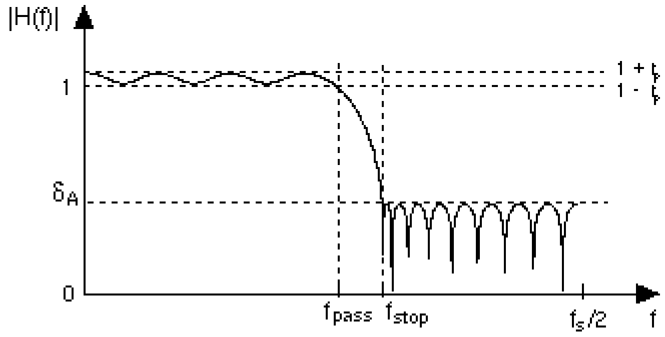


Figure 5-1. Lowpass Filter

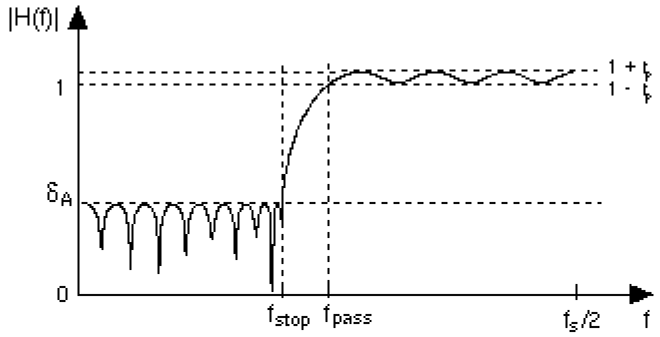


Figure 5-2. Highpass Filter

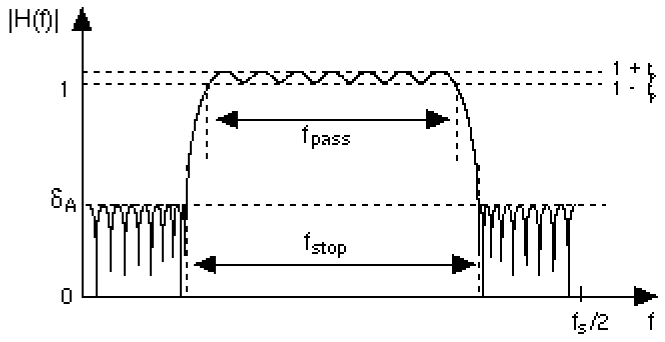


Figure 5-3. Bandpass Filter

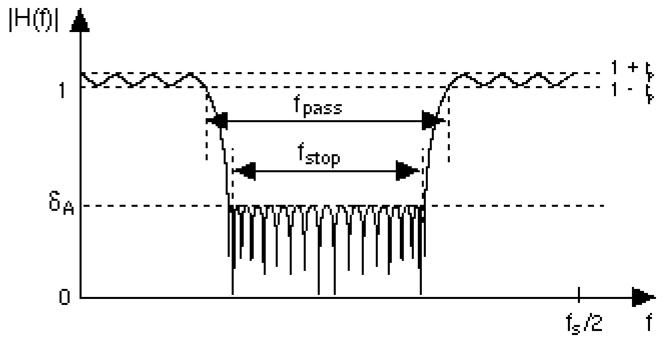
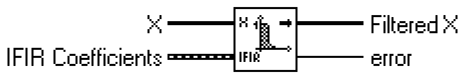


Figure 5-4. Bandstop Filter

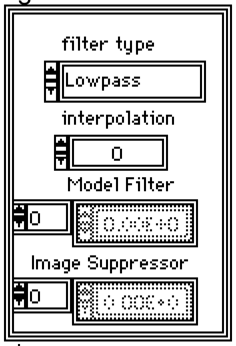
FIR Narrowband Filter (Advanced Only)

Filters the input sequence **X** using the IFIR filter specified by **IFIR Coefficients** as designed by the FIR Narrowband Filter Coefficients VI.



≤ **X** is the input signal to be filtered.

≤ **IFIR Coefficients** is a cluster that contains IFIR coefficients. It has four elements, as the following figure shows.



≤ **filter type** is the filter type that you use to determine how to filter the data.

0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop
4: Wideband-Lowpass *
5: Wideband-Highpass **

* cutoff frequencies near Nyquist
** cutoff frequencies near zero

≤ **interpolation** is the interpolation factor M. The model filter is stretched by **interpolation** times.
≤ **Model Filter** contains the coefficients of the model filter.
≤ **Image Suppressor** contains the coefficients of the filter image suppressor.
≤ **Filtered X** is the output array of filtered samples.
≤ **error**. See Analysis Error Codes for a description of the error.

Note: The overall filter is a linear-phase FIR filter. The delay for this filter is

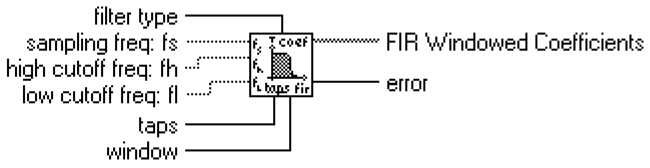
$$\frac{\left[(N_G - 1) \bullet M + N_I\right]}{2}$$

where

N_G is the number of elements in the array **Model Filter**.
N_I is the number of elements in the array **Image Suppressor**.
M is the value of **interpolation** in the cluster **IFIR Coefficients**.

FIR Windowed Coefficients (Advanced Only)

Generates the set of filter coefficients you need to implement a FIR windowed filter.

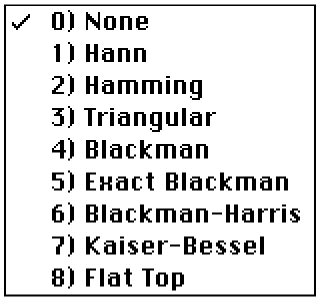


≤ **filter type** specifies the passband of the filter according to the following values.
0: Lowpass
1: Highpass
2: Bandpass
3: Bandstop

≤ **sampling frequency: fs** is the sampling frequency and must be greater than zero. If it is less than or equal to zero, the VI sets **FIR Windowed Coefficients** to an empty array and returns an error. **sampling frequency: fs** defaults to 1.0.
≤ **high cutoff frequency: fh** is the high cutoff frequency. The VI ignores this parameter when **filter type** is 0 (lowpass) or 1 (highpass).
≤ **low cutoff frequency: fl** is the low cutoff frequency and must observe the Nyquist criterion
≤

where f_l is the cutoff frequency, and

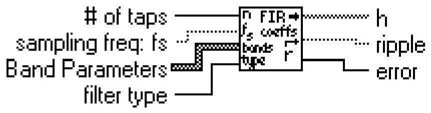
f_s is the sampling frequency. If **low cutoff frequency: fl** is less than zero or greater than half the sampling frequency, the VI sets **FIR Windowed Coefficients** to an empty array and returns an error. **low cutoff frequency: fl** defaults to 0.125.
≤ **taps** determines the total number of FIR coefficients and must be greater than zero. If **taps** is less than or equal to 0, the VI sets **FIR Windowed Coefficients** to an empty array and returns an error. **taps** must be odd for highpass and bandstop filters. **taps** defaults to 25.
≤ **window**. With the **window** control you can select the following smoothing window options.


✓ 0) None
1) Hann
2) Hamming
3) Triangular
4) Blackman
5) Exact Blackman
6) Blackman-Harris
7) Kaiser-Bessel
8) Flat Top

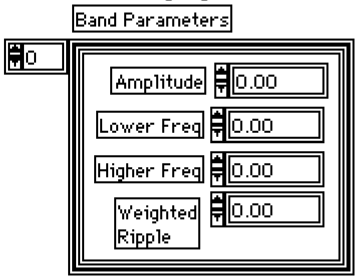
≤ **FIR Windowed Coefficients**.
≤ **error**. See Analysis Error Codes for a description of the error.

Parks-McClellan (Advanced Only)

Generates a set of linear-phase FIR multiband digital filter coefficients using the number of taps, sampling frequency, **Band Parameters**, and filter type.



≤ **# of taps** contains the total number of coefficients in **h**. A tap corresponds to a multiplication and an addition. If there are n taps, every filtered sample requires n multiplications and n additions. **# of taps** must be greater than zero. If **# of taps** is less than or equal to zero, the VI sets **h** to an empty array, sets **ripple** to NaN, and returns an error. **# of taps** defaults to 32.
≤ **sampling freq: fs** is the sampling frequency and must be greater than zero. **sampling freq: fs** defaults to 1.0.
≤ **Band Parameters** is an array of clusters. Each cluster element contains the necessary information associated with each band for the FIR design. Each cluster contains four elements, as shown in the following figure.


Band Parameters
0
Amplitude 0.00
Lower Freq 0.00
Higher Freq 0.00
Weighted Ripple 0.00

The **Band Parameters** cluster array must contain at least one element, that is, one band.

≤ **Amplitude** is the amplitude you want for the band. A value of 1.0 corresponds to a passband, and a value of 0.0 corresponds to a stopband. This VI does not place restrictions on the value of the amplitude.
≤ **Lower Freq** is the frequency at which the band begins.
≤ **Higher Freq** is the frequency at which the band ends.
≤ **Weighted Ripple** is the weighted ripple error that this VI minimizes. The higher the weight, the smaller the error in the band.
For each band, **Higher Freq** must be greater than **Lower Freq**, and for adjacent bands, **Lower Freq** in the higher band must be greater than **Higher Freq** in the lower band,

$f_{h_i} > f_{l_i}$, for i = 0, 1, 2, ..., m-1,

$f_{l_{i+1}} > f_{h_i}$, for i = 1, 2, ..., m-1,

where f_{l_i} represents the **Lower Freq** in the i^{th} band, and

f_{h_i} represents the **Higher Freq** in the

i^{th} band. The **Higher Freq** in the last band must observe the Nyquist criterion

$f_{h_{m-1}}$ £ $0.5f_s$,

where f_s is the sampling frequency.

If **Band Parameters** does not contain any elements, or if any of the preceding frequency conditions is violated, the VI sets **h** to an empty array, sets **Weighted Ripple** to NaN, and returns an error. **Band Parameters** defaults to an empty array.

≤ **filter type**. You can select three distinct filter types.

0: Multiband
1: Differentiator
2: Hilbert

filter type defaults to multiband.

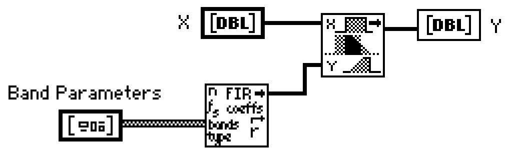
≤ **h** is an array of FIR coefficients, which the VI computes using the Parks-McClellan algorithm with the Remes exchange technique.

≤ **ripple** is the optimal ripple the VI computes and is a measure of deviation from the ideal filter specifications.

≤ **error**. See Analysis Error Codes for a description of the error.

Note: The Parks-McClellan VI finds the coefficients using iterative techniques based upon an error criterion. Although you specify valid filter parameters, the algorithm may fail to converge.

The Parks-McClellan VI generates only the filter coefficients. It does not perform the filtering function. To filter a sequence X using the set of FIR filter coefficients **h**, use the Convolution VI with **X** and **h** as the input sequences.



The equi-ripple filters use a similar technique to filter the data.

Bessel Coefficients.vi

Bessel Coefficients

Bessel Filter.vi

Bessel Filter

Butterworth Coefficients.vi

Butterworth Coefficients

Butterworth Filter.vi

Butterworth Filter

Cascade>Direct Coefficients.vi

Cascade>Direct Coefficients

Chebyshev Coefficients.vi

Chebyshev Coefficients

Chebyshev Filter.vi

Chebyshev Filter

Convolution.vi

Convolution

Elliptic Coefficients.vi

Elliptic Coefficients

Elliptic Filter.vi

Elliptic Filter

Equi-Ripple BandPass.vi

Equi-Ripple BandPass

Equi-Ripple HighPass.vi

Equi-Ripple HighPass

Equi-Ripple LowPass.vi

Equi-Ripple LowPass

FIR Narrowband Coefficients.vi

FIR Narrowband Coefficients

FIR Narrowband Filter.vi

FIR Narrowband Filter

FIR Windowed Coefficients.vi

FIR Windowed Coefficients

FIR Windowed Filter.vi

FIR Windowed Filter

IIR Cascade Filter.vi

IIR Cascade Filter

IIR Cascade Filter with I.C. .vi

IIR Cascade Filter with I.C.

IIR Filter.vi

IIR Filter

IIR Filter with I.C. .vi

IIR Filter with I.C.

Inv Chebyshev Coefficients.vi

Inv Chebyshev Coefficients

Inverse Chebyshev Filter.vi

Inverse Chebyshev Filter

Median Filter.vi

Median Filter.vi

Parks-McClellan.vi

Parks-McClellan

Equi-Ripple BandStop.vi

Equi-Ripple BandStop

Advanced FIR Filtering Subpalette

Advanced FIR Filtering

Advanced IIR Filtering Subpalette

Advanced IIR Filtering

Digital Filtering VIs Overview

For detailed Measurement VI reference information, see Filter VIs.

Following is a list of Digital Filtering Overview topics:

Infinite Impulse Response Filters
Finite Impulse Response Filters
Nonlinear Filters

Overview

Analog filter design is one of the most important areas of electronic design. Although analog filter design books featuring simple, well- tested filter designs exist, filter design is often reserved for specialists because it requires advanced mathematical knowledge and understanding of the processes involved in the system affecting the filter.

Modern sampling and digital signal processing tools have made it possible to replace analog filters with digital filters in applications that require flexibility and programmability. These applications include audio, telecommunications, geophysics, and medical monitoring.

Digital filters have the following advantages over their analog counterparts.

- They are software programmable.
- They are stable and predictable.
- They do not drift with temperature or humidity or require precision components.
- They have a superior performance-to-cost ratio.

You can use digital filters in LabVIEW to control parameters such as filter order, cutoff frequencies, amount of ripple, and stopband attenuation.

The digital filter VIs described in this section follow the virtual instrument philosophy. The VIs handle all the design issues, computations, memory management, and actual data filtering internally, and are transparent to the user. You do not have to be an expert in digital filters or digital filter theory to process the data.

The following discussion of sampling theory is intended to give you a better understanding of the filter parameters and how they relate to the input parameters.

The sampling theorem states that you can reconstruct a continuous-time signal from discrete, equally spaced samples if the sampling frequency is at least twice that of the highest frequency in the time signal. Assume you can sample the time signal of interest at $\lesssim$ equally spaced intervals without losing information. The

$\lesssim$ parameter is the sampling interval.
You can obtain the sampling rate or sampling frequency f_s from the sampling interval

$$f_s = \frac{1}{\Delta t},$$

which means that, according to the sampling theorem, the highest frequency that the digital system can process is

$$f_{Nyq} = \frac{f_s}{2}.$$

The highest frequency the system can process is known as the Nyquist frequency. This also applies to digital filters. For example, if your sampling interval is

≤= 0.001 sec,
then the sampling frequency is

≤ = 1000 Hz,
and the highest frequency that the system can process is

$f_{Nyq} = 500$ Hz.
The following types of filtering operations are based upon filter design techniques.

- Smoothing windows
- Infinite impulse response (IIR) or recursive digital filters
- Finite impulse response (FIR) or nonrecursive digital filters
- Nonlinear filters

The rest of this chapter presents a brief theoretical background on the IIR, FIR, and nonlinear techniques and discusses the digital filter VIs corresponding to each technique. Refer to the Window VIs topic for information about the VIs that implement smoothing windows.

Infinite Impulse Response Filters

Cascade Form IIR Filtering
Chebyshev Filters
Butterworth Filters
Chebyshev II or Inverse Chebyshev Filters
Elliptic or Cauer Filters
Bessel Filters

Infinite impulse response filters (IIR) are digital filters with impulse responses that can theoretically be infinite in length (duration). The general difference equation characterizing IIR filters is

≤
where N_b is the number of forward coefficients (

≤) and
N_a is the number of reverse coefficients (
≤).
In most IIR filter designs (and in all of the LabVIEW IIR filters), coefficient a_0 is 1. The output sample at the present sample index i is the sum of scaled present and past inputs (
≤ and
x_{i-j} when j
≤0) and scaled past outputs {bmc filov-4.bmp}.

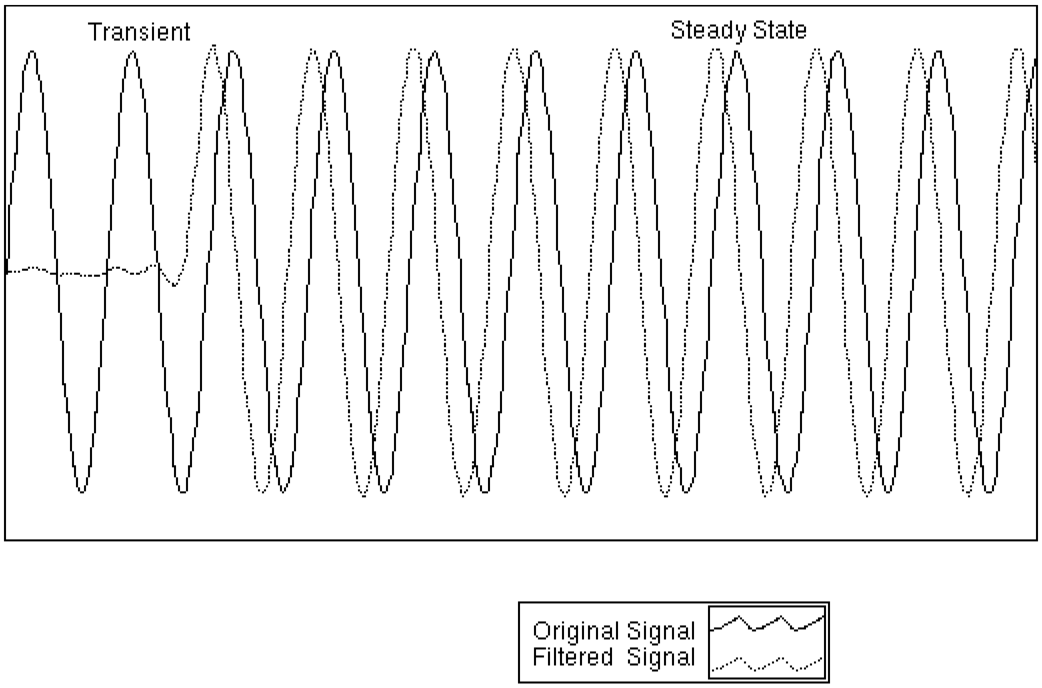
The response of the general IIR filter to an impulse $(x_0 = 1 \text{ and } x_{i-j} = 0 \text{ for all } i \neq 0)$ is called the impulse response of the filter. The impulse response of the filter is indeed of infinite length for nonzero coefficients. In practical filter applications, however, the impulse response of stable IIR filters decays to near zero in a finite number of samples.
IIR filters in LabVIEW contain the following properties.

- Negative indices the above equation are assumed to be zero the first time you call the VI.
- Because the initial filter state is assumed to be zero (negative indices), a transient proportional to the

filter order occurs before the filter reaches a steady state. The duration of the transient response, or delay, for lowpass and highpass filters is equal to the filter order

- delay = order.
- The duration of the transient response for bandpass and bandstop filters is twice the filter order
- delay = 2 * order.

You can eliminate this transient response on successive calls by enabling state memory. To enable state memory, set the **init/cont** control of the VI to TRUE (continuous filtering).



The number of elements in the filtered sequence equals the number of elements in the input sequence.

The filter retains the internal filter state values when the filtering completes.

The advantage of digital IIR filters over finite impulse response (FIR) filters is that IIR filters usually require fewer coefficients to perform similar filtering operations. Thus, IIR filters execute much faster and do not require extra memory, because they execute in place.

The disadvantage of IIR filters is that the phase response is nonlinear. If the application does not require phase information, such as simple signal monitoring, IIR filters may be appropriate. You should use FIR filters for those applications requiring linear phase responses.

IIR filters are also known as recursive filters or autoregressive moving-average (ARMA) filters. See References for Analysis VIs for more information on this topic.

Cascade Form IIR Filtering

Filters implemented using the structure defined by equation (4-1) directly are known as direct form IIR filters. Direct form implementations are often sensitive to errors introduced by coefficient quantization and by computational, precision limits. Additionally, a filter designed to be stable can become unstable with increasing coefficient length, which is proportional to filter order.

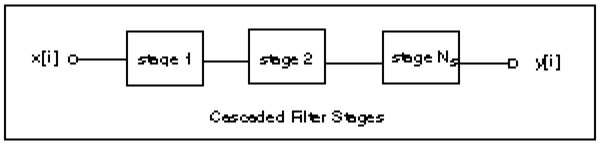
A less sensitive structure can be obtained by breaking up the direct form transfer function into lower order sections, or filter stages. The direct form transfer function of the filter given by equation (4-1) (with ≤=1) can be written as a ratio of z transforms, as follows:

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

By factoring equation (4-2) into second-order sections, the transfer function of the filter becomes a product of second-order filter functions

$$H(z) = \prod_{k=1}^{N_s} \frac{b_{0k} + b_{1k} z^{-1} + b_{2k} z^{-2}}{1 + a_1 z^{-1} + \ldots + a_{2k} z^{-2}}$$

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



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

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

$y_0[i] = x[i]$,¶

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

$$y_k[i] = b_0 k_{sk}[i] + b_{1k} s_k[i-1] + b_{2k} s_k[i-2], k = 1, 2, \ldots, N_s$$

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

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

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

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

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

$$y_k[i] = b_0 k_{sk}[i] + b_{1k} s_k[i-1] + b_{2k} s_k[i-2] - b_{3k} s_k[i-3]$$
$$- b_{4k} s_k[i-4], k = 1, 2, \ldots, N_s$$

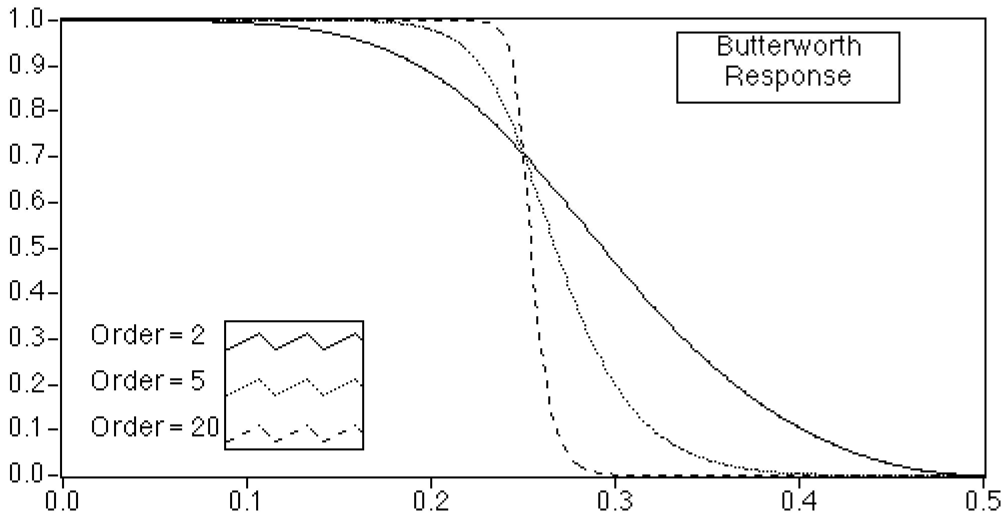
$$y[i] = y_{Ns}[i]$$

Notice that in the case of fourth-order filter stages $N_s = [(N_a + 1)/4]$.

Butterworth Filters

A smooth response at all frequencies and a monotonic decrease from the specified cutoff frequencies characterize the frequency response of Butterworth filters. Butterworth filters are maximally flat--the ideal response of unity in the passband and zero in the stopband. The half power frequency or the 3-dB down frequency corresponds to the specified cutoff frequencies.

The following illustration shows the response of a lowpass Butterworth filter. The advantage of Butterworth filters is a smooth, monotonically decreasing frequency response. Once you set the cutoff frequency, LabVIEW sets the *steepness* of the transition proportional to the filter order. Higher order Butterworth filters approach the ideal lowpass filter response.

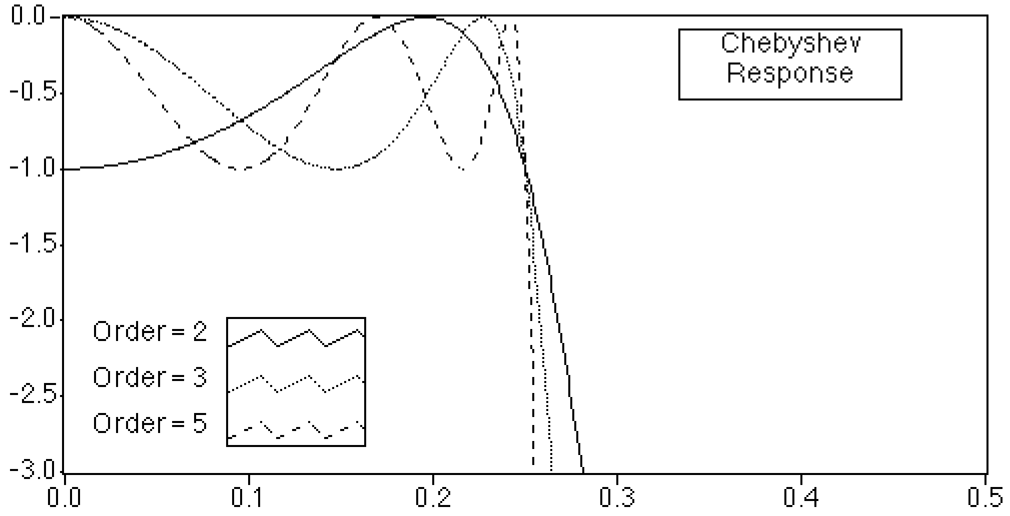


Chebyshev Filters

Butterworth filters do not always provide a good approximation of the ideal filter response because of the slow rolloff between the passband (the portion of interest in the spectrum) and the stopband (the unwanted portion of the spectrum).

Chebyshev filters minimize peak error in the passband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want (the maximum tolerable error in the passband). The frequency response characteristics of Chebyshev filters have an equi-ripple magnitude response in the passband, monotonically decreasing magnitude response in the stopband, and a sharper rolloff than Butterworth filters.

The following graph shows the response of a lowpass Chebyshev filter. Notice that the equi-ripple response in the passband is constrained by the maximum tolerable ripple error and that the sharp rolloff appears in the stopband. The advantage of Chebyshev filters over Butterworth filters is that Chebyshev filters have a sharper transition between the passband and the stopband with a lower order filter. This produces smaller absolute errors and higher execution speeds.

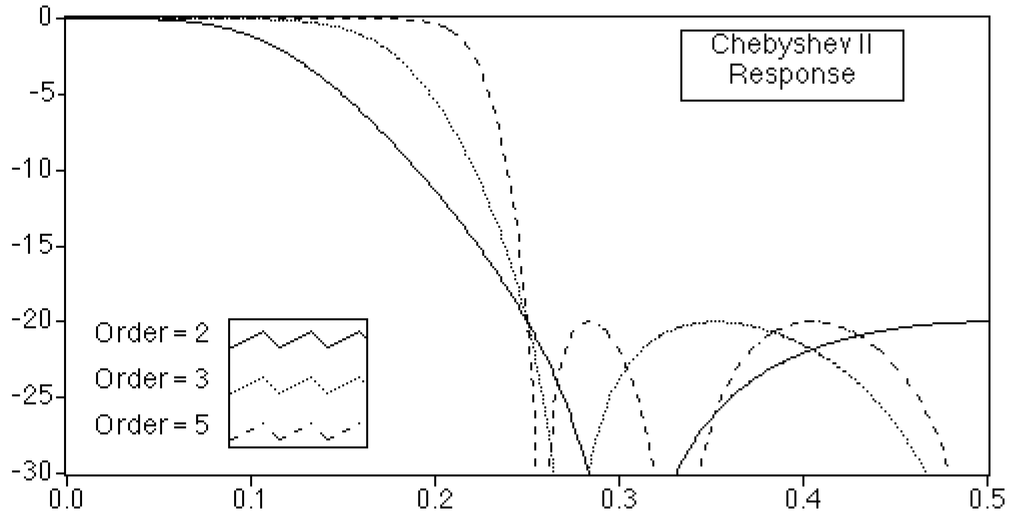


Chebyshev II or Inverse Chebyshev Filters

Chebyshev II, also known as inverse Chebyshev or Type II Chebyshev filters, are similar to Chebyshev filters, except that Chebyshev II filters distribute the error over the stopband (as opposed to the passband), and Chebyshev II filters are maximally flat in the passband (as opposed to the stopband).

Chebyshev II filters minimize peak error in the stopband by accounting for the maximum absolute value of the difference between the ideal filter and the filter response you want. The frequency response characteristics of Chebyshev II filters are equi-ripple magnitude response in the stopband, monotonically decreasing magnitude response in the passband, and a rolloff sharper than Butterworth filters.

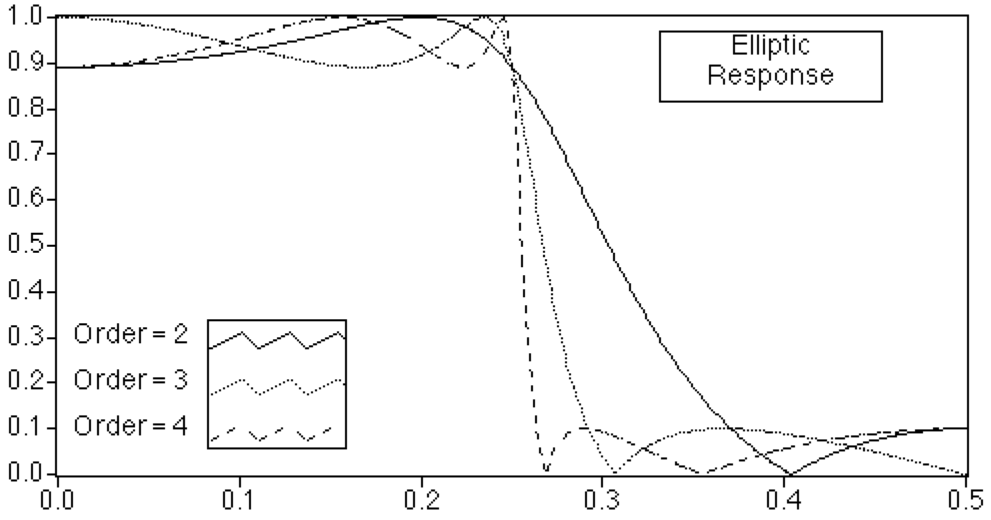
The following graph plots the response of a lowpass Chebyshev II filter. Notice that the equi-ripple response in the stopband is constrained by the maximum tolerable error and that the smooth monotonic rolloff appears in the stopband. The advantage of Chebyshev II filters over Butterworth filters is that Chebyshev II filters give a sharper transition between the passband and the stopband with a lower order filter This difference corresponds to a smaller, absolute error and higher execution speed. One advantage of Chebyshev II filters over regular Chebyshev filters is that Chebyshev II filters distribute the error in the stopband instead of the passband.



Elliptic or Cauer Filters

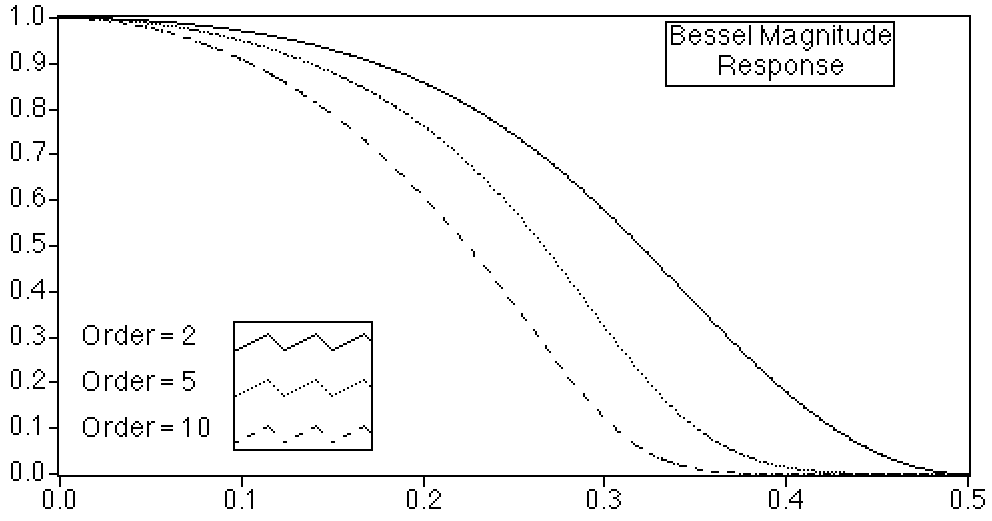
Elliptic filters minimize the peak error by distributing it over the passband and the stopband. Equi-ripples in the passband and the stopband characterize the magnitude response of elliptic filters. Compared with the same order Butterworth or Chebyshev filters, the elliptic design provides the sharpest transition between the passband and the stopband. For this reason, elliptic filters are quite popular.

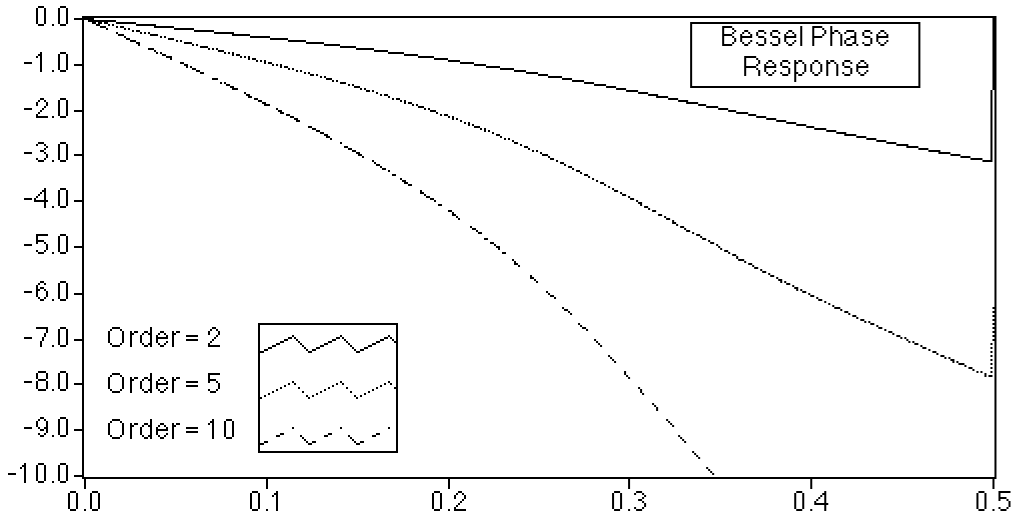
The following graph plots the response of a lowpass elliptic filter. Notice that the ripple in both the passband and stopband is constrained by the same maximum tolerable error (as specified by ripple amount in dB). Also, notice the sharp transition edge for even low-order elliptic filters.



Bessel Filters

You can use Bessel filters to reduce nonlinear phase distortion inherent in all IIR filters. In higher order filters and those with a steeper rolloff, this condition is more pronounced, especially in the transition regions of the filters. Bessel filters have maximally flat response in both magnitude and phase. Furthermore, the phase response in the passband of Bessel filters, which is the region of interest, is nearly linear. Like Butterworth filters, Bessel filters require high-order filters to minimize the error and, for this reason, are not widely used. You can also obtain linear phase response using FIR filter designs.The following graphs plot the response of a lowpass Bessel filter. Notice that the response is smooth at all frequencies, as well as monotonically decreasing in both magnitude and phase. Also, notice that the phase in the passband is nearly linear.





Finite Impulse Response Filters

Finite impulse response (FIR) filters are digital filters, which have a finite impulse response. FIR filters are also known as nonrecursive filters, convolution filters, or moving-average (MA) filters because you can express the output of an FIR filter as a finite convolution

$$y_i = \sum_{k=0}^{n-1} h_k x_{i-k}$$

where X represents the input sequence to be filtered, y represents the output filtered sequence, and h represents the FIR filter coefficients.

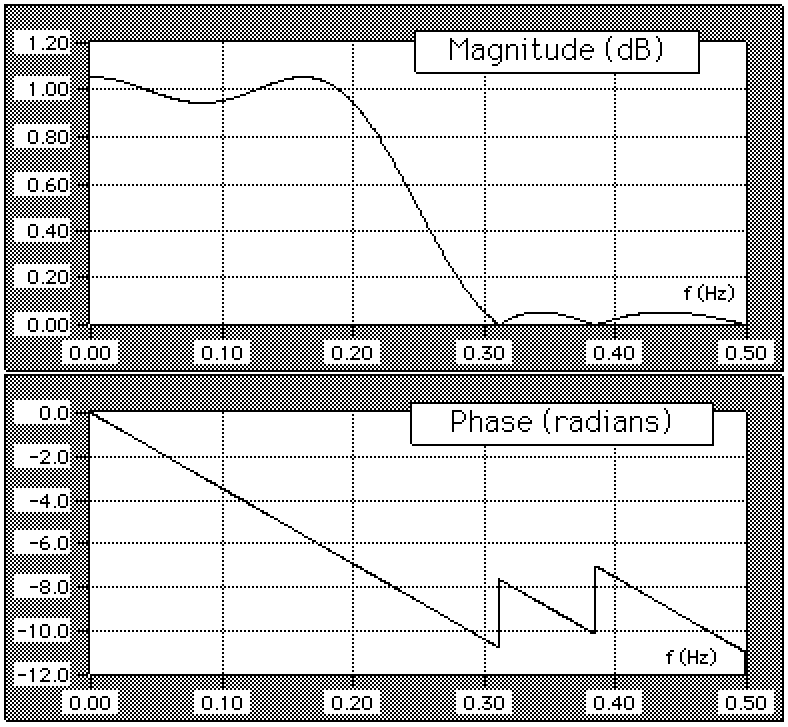
The following list gives the most important characteristics of FIR filters.

- They can achieve linear phase due to filter coefficient symmetry in the realization.
- They are always stable.
- You can perform the filtering function using the convolution and, as such, generally associate a delay with the output sequence

$$\text{delay} = \frac{n-1}{2}$$

- where n is the number of FIR filter coefficients

The following graphs plot a typical magnitude and phase response of FIR filters versus normalized frequency. These graphs were generated using the FIR Filter Design example in the topic, Analysis Examples.



The discontinuities in the phase response arise from the discontinuities introduced when you compute the magnitude response using the absolute value. Notice that the discontinuities in phase are on the order of π. The phase, however, is clearly linear. See References for Analysis VIs for material that can give you more information on this topic.

You design FIR filters by approximating a specified, desired frequency response of a discrete-time system. The most common techniques approximate the desired magnitude response while maintaining a linear phase response.

FIR Filter Design by Windowing

The simplest method for designing linear-phase FIR filters is the *window design* method. To design a FIR filter by windowing, you start with an ideal frequency response, calculate its impulse response, and then truncate the impulse response to produce a finite number of coefficients. You meet the linear phase constraint by maintaining symmetry about the center point of the coefficients. The truncation of the ideal impulse response results in the effect known as the Gibbs phenomenonÐoscillatory behavior near abrupt transitions (cutoff frequencies) in the FIR filter frequency response.

You can reduce the effects of the Gibbs phenomenon by smoothing the truncation of the ideal impulse response using a smoothing window function. By tapering the FIR coefficients at each end, you can diminish the height of the side lobes in the frequency response. The disadvantage to this method, however, is that the main lobe widens, resulting in a wider transition region at the cutoff frequencies. The selection of a window function, then, is similar to the choice between Chebyshev and Butterworth IIR filters in that it is a trade-off between side lobe levels near the cutoff frequencies and width of the transition region.

The design of FIR filters by windowing is simple and computationally inexpensive. It is therefore the fastest way to design FIR filters. It is not necessarily, however, the best FIR filter design method.

Optimum FIR Filter Design of using the Parks-McClellan Algorithm

The Parks-McClellan algorithm offers an optimum FIR filter design technique that attempts to design the best filter possible for a given number of coefficients. Such a design reduces the adverse effects at the cutoff frequencies. It also offers more control over the approximation errors in different frequency bands−control that is not possible with the window method.

Using the Parks-McClellan algorithm to design FIR filters is computationally expensive. This method, however, produces optimum FIR filters by applying time-consuming iterative techniques.

Narrowband FIR Filter Design

When you use conventional techniques to design FIR filters with especially narrow bandwidths, the resulting filter lengths may be very long. FIR filters with long filter lengths often require lengthy design and implementation times, and are more susceptible to numerical inaccuracy. In some cases, conventional filter design techniques, such as the Parks-McClellan algorithm, may fail the design altogether.

You can use a very efficient algorithm, called the Interpolated Finite Impulse Response (IFIR) filter design technique, to design narrowband FIR filters. Using this technique produces narrowband filters that require far fewer coefficients (and therefore fewer computations) than those filters designed by the direct application of the Parks-McClellan algorithm. LabVIEW also uses this technique to produce wideband, lowpass (cutoff frequency near Nyquist) and highpass filters (cutoff frequency near zero). For more information on IFIR filter design, see Multirate Systems and Filter Banks by P.P. Vaidyanathan, or the paper on interpolated finite impulse response filters by Neuvo, et al., listed in References for Analysis VIs .

Windowed FIR Filters

You use the **filter type** parameter of the FIR VIs to select the type of windowed FIR filter you wantÐlowpass, highpass, bandpass, or bandstop. The following list gives the two related FIR VIs.

- FIR Windowed CoefficientsÐgenerates the windowed (or unwindowed) coefficients.
- FIR Windowed FiltersÐfilters the input using windowed (or unwindowed) coefficients.

Optimum FIR Filters

You can use the Parks-McClellan algorithm to design optimum, linear-phase, FIR filter coefficients in the sense that the resulting filter optimally matches the filter specifications for a given number of coefficients. The Parks-McClellan VI takes as input an array of band descriptions, each containing information describing the response you want for the given band. The VI outputs the FIR coefficients along with computed ripple, which is a measure of the deviation of the resulting filter from the ideal filter specifications.

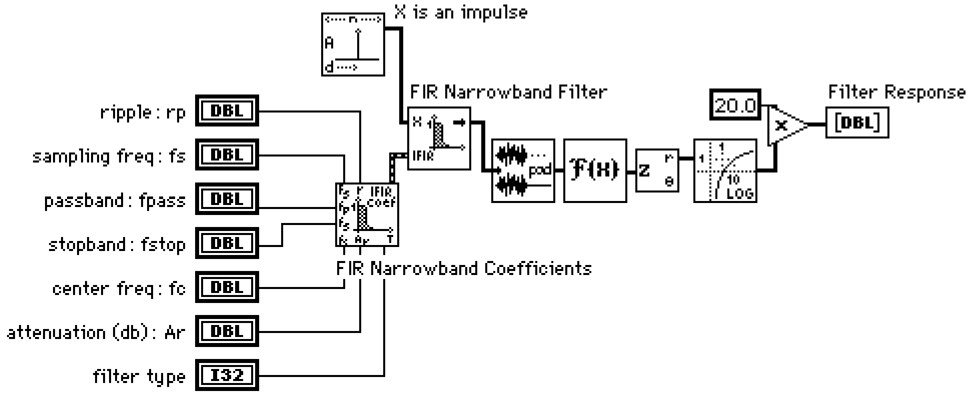
Four VIs use the Parks-McClellan VI to implement filters whose stopband and passband ripple level are equal. These VIs are Equi-Ripple LowPass, Equi-Ripple HighPass, Equi-Ripple BandPass, and Equi-Ripple BandStop.

FIR Narrowband Filters

You can design narrowband FIR filters using the FIR Narrowband Coefficients VI, and then implement the filtering using the FIR Narrowband Filter VI. The design and implementation are separate operations because many narrowband filters require lengthy design times, while the actual filtering process is very fast and efficient. Keep this in mind when creating your narrowband filtering diagrams.

The parameters required for narrowband filter specification are filter type, sampling rate, passband and stopband frequencies, passband ripple (linear scale), and stopband attenuation (decibels). For bandpass and bandstop filters, passband and stopband frequencies refer to bandwidths, and you must specify an additional center frequency parameter. You can also design wideband lowpass filters (cutoff frequency near Nyquist) and wideband highpass filters (cutoff frequency near zero) using the narrowband filter VIs.

The following illustration shows how to use the FIR Narrowband Coefficients VI and the FIR Narrowband Filter VI to estimate the response of a narrowband filter to an impulse.



Nonlinear Filters

Smoothing windows, IIR filters, and FIR filters are linear because they satisfy the superposition and proportionality principles

$$L\{ax(t) + by(t)\} = aL\{x(t)\} + bL\{y(t)\},$$

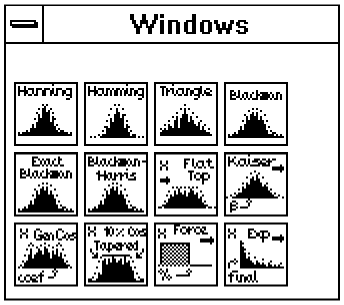
where a and b are constants, $x(t)$ and $y(t)$ are signals, $L\{\bullet\}$ is a linear filtering operation, and their inputs and outputs are related via the convolution operation.

A nonlinear filter does not meet the preceding conditions and you cannot obtain its output signals via the convolution operation, because a set of coefficients cannot characterize the impulse response of the filter. Nonlinear filters provide specific filtering characteristics that are difficult to obtain using linear techniques. The median filter is a nonlinear filter that combines lowpass filter characteristics (to remove high-frequency noise) and high-frequency characteristics (to detect edges).

Window VIs

This topic describes the VIs that implement smoothing windows. For general information about Linear Algebra VIs, see Window VIs Overview .

The following illustration shows the options that are available on the **Windows** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

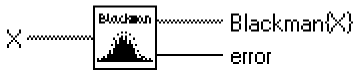


Blackman Window
Blackman-Harris Window
Cosine Tapered Window
Exact Blackman Window
Exponential Window
Flat Top Window
Force Window
General Cosine Window
Hamming Window
Hanning Window
Kaiser-Bessel Window
Triangle Window

For examples of how to use the window VIs, see the examples located in `examples\analysis\windxmpl.llb`.

Blackman Window (Advanced Only)

Applies a Blackman window to the input sequence **X**.



X.
Blackman{X}.
error. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Blackman{X}**, the VI obtains the elements of Y from

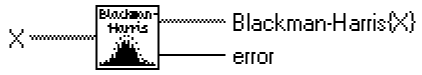
$$y_i = x_i [0.42 - 0.50 \cos(w) + 0.08 \cos(2w)] \quad \text{for } i = 0, 1, 2, ..., n-1$$

$$w = \frac{2\pi i}{n},$$

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

Blackman-Harris Window (Advanced Only)

Applies a three-term, Blackman-Harris window to the input sequence **X**.



≤ **X**.
≤ **Blackman-Harris{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Blackman-Harris{X}**, the VI obtains the elements of Y from

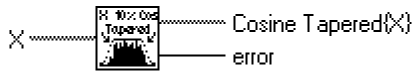
≤ [0.422323 - 0.49755 cos(w) + 0.07922 cos(*2w*)]
for i = 0, 1, 2, ..., n-1,

≤

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

Cosine Tapered Window (Advanced Only)

Applies a cosine tapered window to the input sequence **X**.



≤ **X**.
≤ **Cosine Tapered{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Cosine Tapered{X}**, the VI obtains the elements of Y from

$$y_i = \begin{cases} 0.5x_i(1-\cos w) & \text{for } i = 0,1,2,\ldots,m-1, \quad \text{and for } i = n-m, n-m+1,\ldots,n-1 \\ x_i & \text{elsewhere} \end{cases}$$

where ≤
and

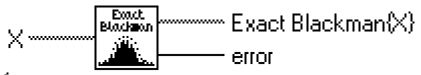
$$m = \text{round}\left(\frac{n}{10}\right),$$

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

Using this window is the equivalent of applying the Hanning window to the first and last 10% of the input sequence **X**.

Exact Blackman Window (Advanced Only)

Applies an Exact Blackman window to the input sequence **X**.



≤ **X**.
≤ **Exact Blackman{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Exact Blackman{X}**, the VI obtains the elements of Y from

$$y_i = x_i\left[a_0 - a_1\cos(w) + a_2\cos(2w)\right],$$

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

$$w = \frac{2\pi i}{n},$$

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

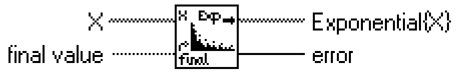
≤= 7938/18608

a_1= 9240/18608, and

a_2= 1430/18608.

Exponential Window (Advanced Only)

Applies an exponential window to the input sequence **X**.



≤ **X**.
≤ **final value**.
≤ Exponential{X}.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Exponential{X}**, the VI obtains the elements of Y from

$$y_i = x_i \exp(ai) \qquad \text{for } i = 0, 1, 2, \ldots, n-1$$

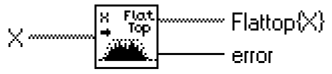
$$a = \frac{\ln(f)}{n-1},$$

where f is the **final value**, and n is the number of samples in **X**.

You can use the Exponential Window VI to analyze transients.

Flat Top Window (Advanced Only)

Applies a flat top window to the input sequence **X**.



≤ **X**.
≤ **Flattop{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Flattop{X}**, the VI obtains the elements of Y from

$$y_i = x_i \quad 0.2810639 - 0.5208972\cos(w) + 0.1980399\cos(2w)]$$

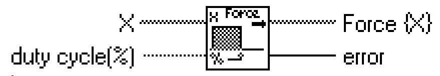
for $i = 0, 1, 2, \ldots, n-1,$

$$w = \frac{2\pi i}{n},$$

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

Force Window (Advanced Only)

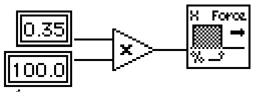
Applies a force window to the input sequence **X**.



≤ **X**.

≤ **duty cycle**.

Note: **duty cycle must be a percentage. If your duty cycle is expressed as a fraction of a completed record, you must convert the duty cycle to a percentage, as shown in the following figure, before using the Force Window VI.**



≤ **Force{X}**.

≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Force{X}**, the VI obtains the elements of *Y* from

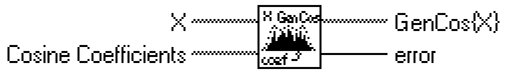
$$w = \begin{cases} x_i & (\text{if } 0 \le i \le d) \\ 0 & \text{elsewhere} \end{cases}$$

$d = (0.01)(n)$(**duty cycle**), where n is the number of elements in **X**.

You can use the Force Window VI to analyze transients.

General Cosine Window (Advanced Only)

Applies a general, cosine window to the input sequence **X**.



≤ **X**.

≤ **Cosine Coefficients**.

Special Case: If **Cosine Coefficients** is an empty array, the VI sets **GenCos{x(t)}** to an empty array, even if **X** is not empty.

≤ **GenCos{X}**.

≤ **error**. See Analysis Error Codes for a description of the error.

If A represents the **Cosine Coefficients** input sequence and Y represents the output sequence **GenCos{X}**, the VI obtains the elements of Y from

$$y_i = x_i \sum_{k-0}^{m-1} (-1)^k a_k \cos(kw)$$

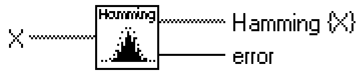
for $i = 0, 1, 2, \ldots, n-1$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in **X**, and m is the number of **Cosine Coefficients**.

Hamming Window (Advanced Only)

Applies a Hamming window to the input sequence **X**.



≤ **X**.
≤ **Hamming{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Hamming{X}**, the VI obtains the elements of Y from

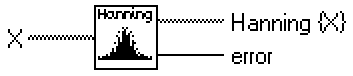
$$y_i = x_i\ [0.54 - 0.46\cos(w)] \text{ for } i = 0, 1, 2, \ldots, n-1,$$

$$w = \frac{2\pi i}{n},$$

where n is the number of elements in the input sequence X.

Hanning Window (Advanced Only)

Applies a Hanning window to the input sequence **X**.



≤ **X**.
≤ **Hanning {X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Hanning {X}**, the VI obtains the elements of Y using

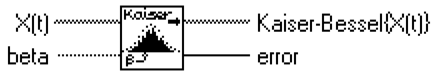
$$y_i = 0.5x_i\ [1 - \cos(w)] \qquad \text{for } i = 0, 1, 2, \ldots, n-1,$$

$$w = \frac{2\pi i}{n},$$

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

Kaiser-Bessel Window (Advanced Only)

Applies a Kaiser-Bessel window to the input sequence **X(t)**.



≤ **X(t)**.
≤ **beta** is proportional to the side lobe attenuationÐthat is, the larger the **beta** is, the greater the side lobe attenuation. **beta** defaults to 0.0.
≤ **Kaiser-Bessel{X(t)}**.
≤ **error**. See Analysis Error Codes for a description of the error.

If Y represents the output sequence **Kaiser-Bessel{X(t)}**, the VI obtains the elements of Y from

$$y_i = x_i \frac{I_o\left(\beta\sqrt{1.0-a^2}\right)}{I_o(\beta)}$$

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

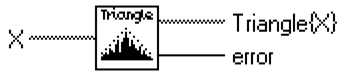
$$a = \frac{i-k}{k},$$

$$k = \frac{n-1}{2},$$

where n is the number of elements in **X(t)**, and *I*o() is the zero-order modified Bessel function.

Triangle Window (Advanced Only)

Applies a triangular window to the input sequence **X**.



≤ **X**.
≤ **Triangle{X}**.
≤ **error**. See Analysis Error Codes for a description of the error.

Note: The triangle smoothing window is also known as the Bartlett smoothing window.

If Y represents the output sequence **Triangle{X}**, the VI obtains the elements of Y from

$$y_i = x_i \ \mathrm{tri}(w) \qquad \text{for } i = 0, 1, 2, \ldots, n-1,$$

$$w = \frac{2i-n}{n},$$

where $\mathrm{tri}(w) = 1 - |w|$, and n is the number of elements in **X**.

Blackman Window.vi

Blackman Window

Blackman-Harris Window.vi

Blackman-Harris Window

Cosine Tapered Window.vi

Cosine Tapered Window

Exact Blackman Window.vi

Exact Blackman Window

Exponential Window.vi

Exponential Window

Flat Top Window.vi

Flat Top Window

Force Window.vi

Force Window

General Cosine Window.vi

General Cosine Window

Hamming Window.vi

Hamming Window

Hanning Window.vi

Hanning Window

Kaiser-Bessel Window.vi

Kaiser-Bessel Window

Triangle Window.vi

Triangle Window

Window VIs Overview

For detailed VI descriptions, see Window VIs .

Smoothing Windows
Windows for Spectral Analysis versus Windows for Coefficient Design

Smoothing Windows

In practical, signal-sampling applications, you can obtain only a finite record of the signal, even when you carefully observe the sampling theorem and sampling conditions. Unfortunately for the discrete-time system, the finite sampling record results in a truncated waveform that has different spectral characteristics from the original continuous-time signal. These discontinuities produce leakage of spectral information, resulting in a discrete-time spectrum that is a smeared version of the original continuous-time spectrum.

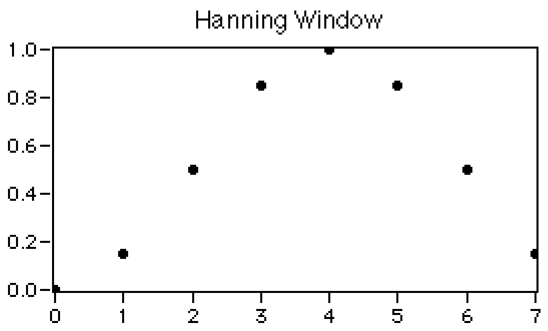
A simple way to improve the spectral characteristics of a sampled signal is to apply smoothing windows. When performing Fourier or spectral analysis on finite-length data, you can use windows to minimize the transition edges of your truncated waveforms, thus reducing spectral leakage. When used in this manner, smoothing windows act like predefined, narrowband, lowpass filters.

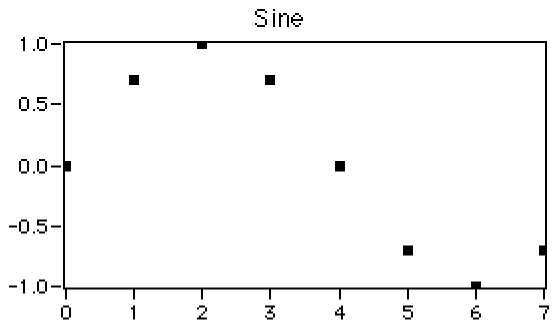
Windows for Spectral Analysis versus Windows for Coefficient Design

The window VIs implemented in the Analysis library in LabVIEW are designed for spectral analysis applications. In these applications, the input signal is windowed by passing it through one of the window VIs. The windowed signal is then passed to a DFT-based VI for frequency-domain display and analysis.

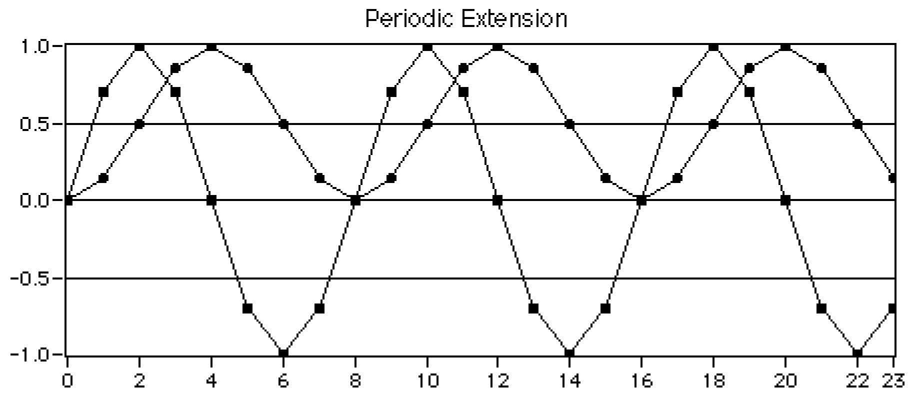
The window functions designed for spectral analysis must be DFT-even, a term defined by Fredric J. Harris in his paper On the Use of Windows for Harmonic Analysis with the Discrete Fourier Transform (Proceedings of the IEEE, Volume 66, No.1, January 1978). A window function is DFT-even if its dot product (inner product) with integral cycles of sine sequences is identically zero. Another way to think of a DFT-even sequence is that its DFT has no imaginary component.

The following figures illustrate the Hanning window and one cycle of a sine pattern for a sample size of 8. It is clear from the figures that the DFT-even Hanning window is not symmetric about its midpoint and its last point is not equal to its first point, much like one complete cycle of a sine pattern.





Finally, the DFT considers input sequences to be periodic--that the signal being analyzed is actually a concatenation of the input signal. The following illustration shows three such cycles of the above sequences, demonstrating the smooth periodic extension of the DFT-even window and the single-cycle sine pattern.



Another type of window application is that of FIR filter design (see the descriptions of FIR Windowed Coefficients and FIR Windowed Filter). This application requires windows that are symmetric about their midpoint.

The difference between the DFT-even window function (spectral analysis) and the symmetrical window function (coefficient design) can be clearly seen in the following equations of the Hanning window function.

Hanning window function for spectral analysis:

$$w[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N}\right)\right)$$

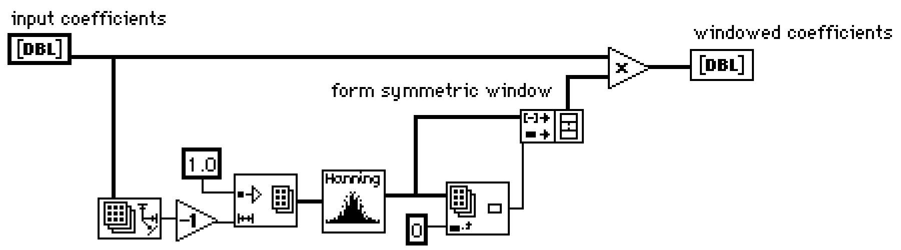
for i=0,1,2,...,N-1

Hanning window function for symmetrical coefficient design:

$$v[i] = 0.5\left(1 - \cos\left(\frac{2\pi i}{N-1}\right)\right)$$

for i=0,1,2,...,N-1

It is clear from the above equations that you can implement the symmetrical window functions by slightly modifying the use of the DFT-even window functions. The following illustration shows a block diagram that uses the Hanning Window VI to implement symmetrical windowing of filter coefficients.

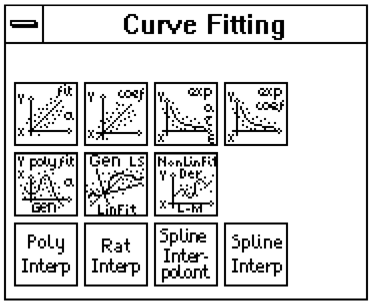


An example in the section Analysis Examples , demonstrates how to use smoothing windows, and the LabVIEW distribution disks contain an example that shows the spectral characteristics of the smoothing windows. See References for Analysis VIs   for material that can give you more information on this topic.

Curve Fitting VIs

This topic describes the VIs that perform curve fitting analysis or regression. For general information about Filter VIs, see Curve Fitting VIs Overview .

The following illustration shows the options that are available on the **Curve Fitting** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.

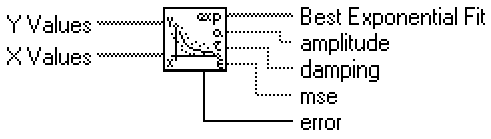


Exponential Fit
Exponential Fit Coefficients
General LS Linear Fit
General Polynomial Fit
Linear Fit
Linear Fit Coefficients
Nonlinear Lev-Mar Fit
Polynomial Interpolation
Rational Interpolation
Spline Interpolant
Spline Interpolation

For examples of how to use the regression VIs, see the examples located in `examples\analysis\regressn.llb`.

Exponential Fit (Advanced Only)

Finds the exponential curve values and the set of exponential coefficients **amplitude** and **damping**, which describe the exponential curve that best represents the input data set.



≤ **Y Values** must have the same sign and must contain at least two points: that is,

$y_i>0$ for i=0, 1,...n-1 or

$y_i<0$ for i=0, 1,...n-1, n

≥ 2, where Y represents the input sequence **Y Values**, and n is the number of data points. If the signs are inconsistent or there are less than two sample points, the VI sets **Best Exponential Fit** to an empty array, sets **amplitude**, **damping**, and **mse** to NaN, and returns an error via the Exponential Fit Coefficients VI.

Note: This VI performs an exponential fit even when the elements of Y Values are negative. It

performs the fit under the assumption that the amplitude coefficient is also negative and returns a negative amplitude. Y Values cannot contain both positive and negative elements.

≤ **X Values** must contain at least two points.
≤ **Best Exponential Fit**.
≤ **amplitude**.
≤ **damping**.
≤ **mse** is the mean squared error.
≤ **error**. See Analysis Error Codes for a description of the error.

The general form of the exponential fit is given by

$$F = ae^{\tau x}$$

where F is the output sequence **Best Exponential Fit**, X is the input sequence **X Values**, *a* is the **amplitude**, and t is the **damping** constant.

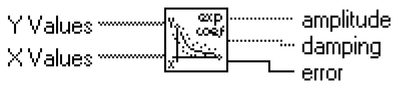
The VI obtains **mse** using the formula

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

where f is the output sequence **Best Exponential Fit**, y is the input sequence **Y Values**, and n is the number of data points.

Exponential Fit Coefficients (Advanced Only)

Finds the set of exponential coefficients **amplitude** and **damping**, which describe the exponential curve that best represents the input data set.



This VI is a subVI of the Exponential Fit VI.

≤ **Y Values** must have the same sign and must contain at least two points: that is,
≤>0 for i=0, 1,...n-1 or
≤<0 for i=0, 1,...n-1, n
≤ 2, where Y represents the input sequence **Y Values**, and n is the number of data points. If the signs are inconsistent or there are less than two sample points, the VI sets **Best Exponential Fit** to an empty array, sets **amplitude**, **damping**, and **mse** to NaN, and returns an error via the Exponential Fit Coefficients VI.

Note: This VI performs an exponential fit even when the elements of YÊValues are negative. The VI performs the fit under the assumption that the amplitude coefficient is also negative and returns a negative amplitude. YÊValues cannot be a mixture of positive and negative elements.

≤ **X Values** must contain at least two points.
≤ **amplitude**.
≤ **damping**.
≤ **error**. See Analysis Error Codes for a description of the error.

The general form of the exponential fit is given by

≤

where F is the sequence representing the best fitted values, X represents the input sequence **X Values**, *a* is the **amplitude**, and t is the **damping** constant.

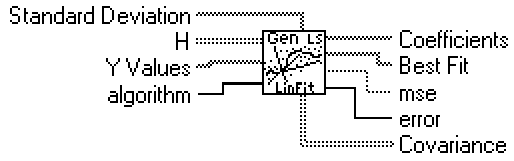
General LS Linear Fit (Advanced Only)

For examples of General LS Linear Fit see:

Example 1: Predicting Cost
Example 2: Linear Combinations

General LS Linear Fit finds the Best Fit k-dimensional plane and the set of linear coefficients using the least chi-square method for observation data sets

$\{x_{i0}, x_{i1}, \dots x_{ik-1}, y_i\}$ where i = 0, 1, ..., n - 1. n is the number of your observation data sets.



≤ **H** is an n-by-k matrix, which contains the observation data i=0,1,...,n-1, where n is the number of rows in **H**, k is the number of columns in H.
≤ $\mathbf{Y}$ **Values**. The number of elements in $\mathbf{Y}$ **Values** should be equal to the number of rows in **H**.
≤ **algorithm** has six selections:
0: SVD
1: Givens
2: Givens2
3: Householder
4: LU decomposition
5: Cholesky

algorithm defaults to SVD.

≤ **Standard Deviation** is the standard deviation
$\sigma_i$for data point
$(x_i y_i)$. If they are equal or you do not know, leave this array empty. Internally, LabVIEW sets all to 1.0.
≤ **Coefficients** is the set of coefficients that minimize
χ^2, which is defined in equation (7-5).
≤ **Best Fit** is the fitted data computed by using the Coefficients.
≤ **mse** is the mean squared error.
≤ **error**. See Analysis Error Codes for a description of the error.
≤ **Covariance** is the matrix of covariances C with k-by-k elements.
c_{jk} is the covariance between
a_i and
≤ and
c_{jj} is the variance of
a_j.
You can use the General LS Linear Fit VI to solve multiple linear regression problems. You can also use it to solve for the linear coefficients in a multiple-function equation. Before beginning the formal description of this VI, consider both of the following, simple examples. The first example uses the General LS Linear Fit VI to perform multiple regression analysis based entirely on tabulated observation data. The second solves for the linear coefficients in a multiple-function equation.

Example 1: Predicting Cost

Suppose you want to estimate the total cost (in dollars) of a production of baked scones; using the quantity produced, X1, and the price of one pound of flour, X2. To keep things simple, the following five data points form this sample data table.

Cost (dollars)	**Quantity**	**Flour Price**
Y	**X1**	**X2**
$150	295	3.00
$75	100	3.20
$120	200	3.10
$300	700	2.80
$50	60	2.50

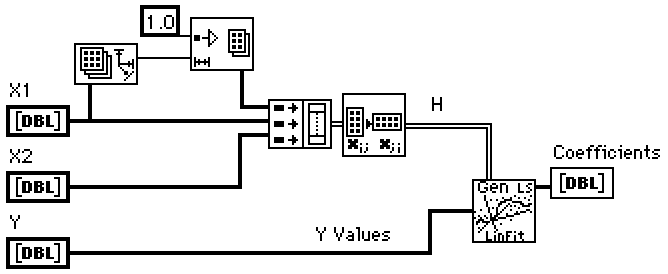
You want to estimate the coefficients to the equation:

$$y = b_0 + b_1X_1 + b_2X_2.$$

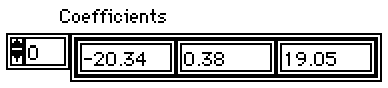
The only parameters that you need to build are **H** (observation matrix) and Y Values. Each column of **H** is the observed data for each independent variable: the first column is one because the coefficient b0 is not associated with any independent variable. **H** should be filled in as:

$$H = \begin{bmatrix} 1 & 295 & 3.00 \\ 1 & 100 & 3.20 \\ 1 & 200 & 3.10 \\ 1 & 700 & 2.80 \\ 1 & 60 & 2.50 \end{bmatrix}$$

In LabVIEW, the observed data would normally appear in three arrays (Y, X1, and X2). The following block diagram demonstrates how to build H using the General LS Linear Fit VI.



After running the General LS Linear Fit VI, the following Coefficients are obtained.



and the resulting equation for the total cost of scone production is therefore:

Y=-20.34 + 0.38X1 + 19.05X2.

Example 2: Linear Combinations

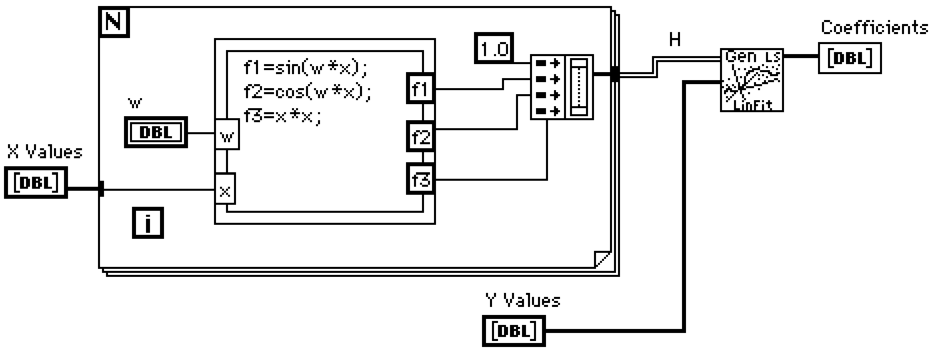
Suppose that you have collected samples from a transducer (Y Values) and you want to solve for the coefficients of the model:

$$y = b_0 + b_1 \sin(\omega x) + b_2 \cos(\omega x) + b_3 x^3$$

To build **H**, you set each column to the independent functions evaluated at each *x* value. Assuming there are 100 x values, **H** would be:

$$H = \begin{bmatrix} 1 & \sin(\omega x_0) & \cos(\omega x_0) & x_0{}^2 \\ 1 & \sin(\omega x_1) & \cos(\omega x_1) & x_1{}^2 \\ 1 & \sin(\omega x_2) & \cos(\omega x_2) & x_2{}^2 \\ \cdots & \cdots & \cdots & \cdots \\ 1 & \sin(\omega x_{99}) & \cos(\omega x_{99}) & x_{99}{}^2 \end{bmatrix}$$

Given that you have the independent X Values and observed Y Values, the following block diagram demonstrates how to build H and use the General LS Linear Fit VI.



The General LS Linear Fit Problem can be described as follows.

Given a set of observation data, find a set of coefficients that fit the linear ÒmodelÓ.

$$y_i = b_0 x_{i0} + \ldots + b_{k-1} x_{ik-1} = \sum_{j=0}^{k-1} b_j x_{ij} \qquad i = 0, 1, \ldots, n-1 \tag{7-4}$$

where

B is the set of **Coefficients**.
n is the number of elements in **Y Values** and the number of rows of **H**.
k is the number of **Coefficients**.

x_{ij} is your observation data, which is contained in **H**.

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

Equations (7-4) can also be written as $Y=HB$.

This is a multiple linear regression model, which uses several variables $x_{i0}, x_{i1}, \ldots, x_{ik-1}$, to predict one variable

≤. In contrast, the Linear Fit, Exponential Fit, and Polynomial Fit VIs are all based on a single predictor variable, which uses one variable to predict another variable.
In most cases, we have more observation data than coefficients. The equations in (7-4) may not have the solution. The fit problem becomes to find the coefficients B that minimizes the difference between the observed data, ≤ and the predicted value

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

This VI uses the least chi-square plane method to obtain the coefficients in (7-4), that is, finding the solution, B, which minimizes the quantity:

$$\chi^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2 = \sum_{i=0}^{n-1}\left(\frac{y_i - \sum_{j=0}^{k-1} b_j x_{ij}}{\sigma_i}\right)^2 = \left|H_0B - Y_0\right|^2 \qquad (7\text{-}5)$$

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

In this equation,   is the **Standard Deviation**. If the measurement errors are independent and normally distributed with constant standard deviation, $\sigma_i = \sigma$ the preceding equation is also the least square estimation.

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

$$\begin{cases} \dfrac{\partial\chi^2}{\partial b_0} = 0 \\ \dfrac{\partial\chi^2}{\partial b_1} = 0 \\ . \\ . \\ . \\ . \\ \dfrac{\partial\chi^2}{\partial b_{k-1}} = 0 \end{cases}$$

The preceding equations can be derived to: $H_0^T H_0 B = H_0^T Y$ (7-6)

H_0^T is the transpose of
H_0.
Equations (7-6) are also called normal equations of the least-square problems. You can solve them using LU or Cholesky factorization algorithms, but the solution from the normal equations is susceptible to roundoff error.
An alternative, and preferred way to minimize $\leq$ is to find the least square solution of equations
$H_0 B = Y_0$
You can use QR or SVD factorization to find the solution, B. For QR factorization, you can choose Householder, Givens, and Givens2 (also called fast Givens) .

Different algorithms can give you different precision, and in some cases, if one algorithm cannot solve the equation, perhaps another algorithm can. You can try different algorithms to find the best one based on your observation data.

The **Covariance** matrix C is computed as

$$C = \left(H_0^T H_0\right)^{-1}.$$

The **Best Fit** Z is given by

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

The **mse** is obtained using the following formula.

$$mse = \frac{1}{n}\sum_{i=0}^{n-1}\left(\frac{y_i - z_i}{\sigma_i}\right)^2$$

The polynomial fit that has a single predictor variable can be thought of as a special case of multiple regression. If the observation data sets are $\{x_i, y_i\}$ where i = 0, 1, ..., n-1, the model for polynomial fit is:

$$y_i = \sum_{j=0}^{k-1} b_j x_i^j = b_0 + b_1 x_i + b_2 x_i^2 + \ldots + b_{k-1} x_i^{k-1} \qquad (7\text{-}7)$$

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

Comparing equations (7-4) and (7-7) shows that . In other words,

$$x_{i0} = x_i^0,\ x_{i1} = x_i,\ x_{i2} = x_1^2, \ldots x_{ik-1} = x_i^{k-1}$$

In this case, you can build **H** as follows:

$$H = \begin{Bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{k-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{k-1} \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{k-1} \end{Bmatrix}$$

Instead of using $x_{ij} = x_j^i$, you can also choose another function formula to fit the data sets

$\leq$. In general, you can select

$x_{ij} = f_j(x_i)$. Here,

$f_j(x_i)$ is the function model that you choose to fit your observation data. In polynomial fit,

$f_j(x_i) = x_i^j$.

In general, you can build **H** as follows:

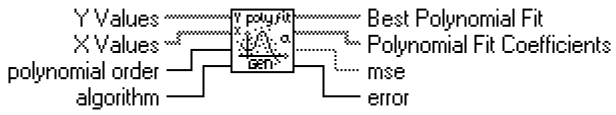
$$H = \begin{Bmatrix} f_0(x_0) & f_1(x_0) & f_2(x_0) & \cdots & f_{k-1}(x_0) \\ f_0(x_1) & f_1(x_1) & f_2(x_1) & \cdots & f_{k-1}(x_1) \\ . & & & & \\ . & & & & \\ . & & & & \\ . & & & & \\ f_0(x_{n-1}) & f_1(x_{n-1}) & f_2(x_{n-1}) & \cdots & f_{k-1}(x_{n-1}) \end{Bmatrix}$$

Your fit model is:

$$y_i = b_0 f_0(x) + b_1 f_1(x) + \ldots + b_{k-1} f_{k-1}(x)$$

General Polynomial Fit (Advanced Only)

Finds the polynomial curve values and the set of Polynomial Fit Coefficients, which describe the polynomial curve that best represents the input data set.



≤ **Y Values.** The number of sample points in **Y Values** must be greater than **polynomial order**. If the number of sample points is less than or equal to **polynomial order**, the VI sets **Polynomial Fit Coefficients** to an empty array and returns an error.

≤ **X Values.** The number of sample points in **X Values** must be greater than **polynomial order**. If the number of sample points is less than or equal to **polynomial order**, the VI sets **Polynomial Fit Coefficients** to an empty array and returns an error.

≤ **polynomial order** must be greater than or equal to zero: 0 ≤ $m < n-1$, where n is the number of sample points, and m is the **polynomial order.** If polynomial order is less than zero, the VI sets **Polynomial Fit Coefficients** to an empty array and returns an error. polynomial order defaults to 2.

≤ **algorithm** has six selections:

0: SVD
1: Givens
2: Givens2
3: Householder
4: LU decomposition
5: Cholesky

It defaults to SVD.

≤ **Best Polynomial Fit.**

≤ **Polynomial Fit Coefficients.** The total number of elements in **Polynomial Fit Coefficients** is $m + 1$. where m is the polynomial order.

≤ **mse** is the mean squared error.

≤ **error.** See Analysis Error Codes for a description of the error.

The general form of the polynomial fit is given by

$$f_i = \sum_{j=0}^{m} a_j x_i^j$$

where

F represents the output sequence **Best Polynomial Fit**,
X represents the input sequence **X Values**,
A represents the **Polynomial Fit Coefficients**, and m is the **polynomial order**.

The VI obtains **mse** using the formula

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

where Y represents the input sequence **Y Values,** and n is the number of data points.

General Polynomial Fit is a special case of the General LS Linear Fit. The General Polynomial Fit VI uses the General LS Linear Fit VI as a subVI. This VI builds the H matrix internally using input X Values for the General LS Linear Fit VI.

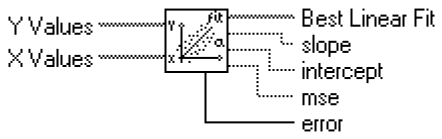
The formula used to build H is as follows:

$$h_{ij} = f_j(x_i) = x_i^j \qquad \text{For example, } H = \begin{bmatrix} 1 & x_0 & \dots x_0^m \\ 1 & x_1 & \dots x_1^m \\ & & . \\ & & . \\ 1 & & . \\ & x_{n-1} & \dots x_{n-1}^m \end{bmatrix}$$
$$i = 0,1,\dots,n-1$$
$$j = 0,1,\dots,m$$

For more information about the General LS Linear Fit VI and the difference among different algorithms, please refer to the description of JUMP General LS Linear Fit VI

Linear Fit

Finds the line values and the set of linear coefficients **slope** and **intercept**, which describe the line that best represents the input data set.



≤ Y **Values** must contain at least two points. If there are fewer than two sample points, the VI sets **Best Linear Fit** to an empty array, sets **slope**, **intercept**, and **mse** to NaN, and returns an error via the Linear Fit Coefficients VI.
≤ **X Values** must contain at least two points. If there are fewer than two sample points, the VI sets **Best Linear Fit** to an empty array, sets **slope**, **intercept**, and **mse** to NaN, and returns an error via the Linear Fit Coefficients VI.
≤ **Best Linear Fit**.
≤ **slope**.
≤ **intercept**.
≤ **mse** is the mean squared error.
≤ **error**. See Analysis Error Codes for a description of the error.
The general form of the linear fit is given by

$$F = mX + b,$$

where F represents the output sequence **Best Linear Fit**, X represents the input sequence **X Values**, m is the **slope**, and b is the **intercept**.

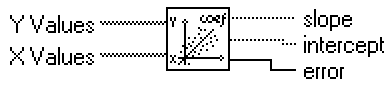
The VI obtains **mse** using the formula

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

where F represents the output sequence **Best Linear Fit**, Y represents the input sequence $\mathbf{Y}$ **Values**, and n is the number of data points.

Linear Fit Coefficients (Advanced Only)

Finds the set of linear coefficients **slope** and **intercept**, which describe the line that best represents the input data set.



This VI is a subVI of the Linear Fit VI.

≤ **Y Values** must contain at least two points. If there are less than two sample points, the VI sets **slope** and **intercept** to NaN and returns an error.

≤ **X Values** must contain at least two points. If there are less than two sample points, the VI sets **slope** and **intercept** to NaN and returns an error.

≤ **slope**.

≤ **intercept**.

≤ **error**. See Analysis Error Codes for a description of the error.

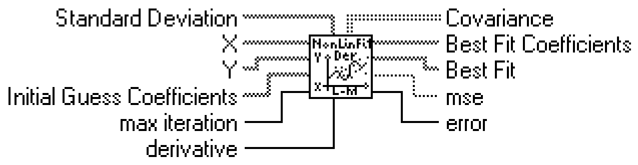
The general form of the linear fit is given by

$$F = mX + b,$$

where F is the sequence representing the best fitted values. X represents the input sequence **X Values**, m is the **slope**, and b is the **intercept.**

Nonlinear Lev-Mar Fit (Advanced Only)

Uses the Levenberg-Marquardt method to determine a nonlinear set of coefficients that minimize a chi-square quantity.



≤ **X** is the input array. The number of valid input points must be greater than zero and greater than the number of specified coefficients.

≤ **Y** is the input array. The number of valid input points must be greater than zero and greater than the number of specified coefficients.

≤ **Standard Deviation** is the standard deviation
σ_i for data point
$(x_i y_i)$. If they are equal or you do not know, leave this array empty. Internally, LabVIEW sets all to 1.0.

≤ **Initial Guess Coefficients** denotes your initial guessed solution.

[I32] **max iteration** is the maximum executing iteration. If the VI reaches maximum iteration without finding a solution, the function returns an error. You have to increase the **max iteration** or adjust the **Initial Guess Coefficients** to get a solution.

≤ **derivative** specifies the method used to calculate the Jacobian. You can choose from the following two methods.

0: numerical calculation. Use numerical approximation to compute the Jacobian.

1: formula calculation. Use the formula you specified in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI to compute the Jacobian.

≤ **Best Fit Coefficients** is the set of coefficients that minimize
≤ , which is defined in equation (7-8).

≤ **Best Fit** is the fitted data
$z_i = f(x_i)A$, computed by using **Best Fit Coefficients** for A. For a definition of
$z_i = f(x_i)A$, see the information below the error parameter.

≤ **Covariance** is the matrix of covariances C.
≤is the covariance between
≤and

≤, and
≤is the variance of
≤.
≤ **mse** is the value of
≤ in equation (7-8), computed by using the **Best Fit Coefficients**.
≤ **error**. See Analysis Error Codes for a description of the error.
This VI determines the set of coefficients that minimize the chi-square quantity:

$$\chi^2 = \sum_{i=0}^{N-1}\left(\frac{y_i - f(x_i;a_1 \ldots a_M)}{\sigma_i}\right)^2 \qquad (7\text{-}8)$$

In this equation, $(x_i y_i)$ are the input data points, and is the nonlinear function

$f(x_i;a_1 \ldots a_M) = f(X, A)$ where
$a_1 \cdots a_M$ are coefficients. If the measurement errors are independent and normally distributed with constant, standard deviation
$\sigma_i = \sigma$ this is also the least-square estimation.
You must specify the nonlinear function $f = f(X, A)$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, which is a subVI of the Nonlinear Lev-Mar Fit VI. You can access the Target Fnc & Deriv NonLin VI by selecting it from the menu that appears when you select **Windows»This VI's SubVIs**.
This VI provides two ways to calculate the Jacobian (partial derivatives with respect to the coefficients) needed in the algorithm. These two methods are:

- numerical calculation--Uses a numerical approximation to compute the Jacobian.
- formula calculation--Uses a formula to compute the Jacobian. You need to specify the Jacobian function $\partial f / \partial A$ in the Formula Node on the block diagram of the Target Fnc & Deriv NonLin VI, as well as the nonlinear function $f = f(X, A)$ This is a more efficient computation than the numerical calculation, because it does not require a numerical approximation to the Jacobian.

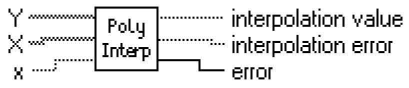
The input arrays **X** and **Y** define the set of input data points. The VI assumes that you have prior knowledge of the nonlinear relationship between the x and y coordinates. That is, $f = f(X, A)$, where the set of coefficients, A, is determined by the Levenberg-Marquardt algorithm.

Using this function successfully sometimes depends on how close your initial guess coefficients are to the solution. Therefore, it is always worth taking effort and time to obtain good initial guess coefficients to the solution from any available resources before using the function.

Polynomial Interpolation (Advanced Only)

Interpolates or extrapolates the function f at x, given a set of n points ≤, where

$f(x_i) = y_i$, f is any function, and given a number, x. The VI calculates output **interpolation value** $P_{n-1}(x)$, where
$P_{n-1}$is the unique polynomial of degree n-1 that passes through the n points
≤.



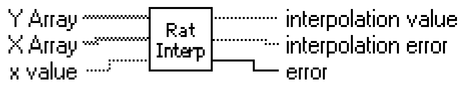
≤ **Y** is the input array.
≤ **X**. If the number of elements in **X** is different from the number of elements in Y, the VI sets the output **interpolation value** and **interpolation error** to NaN and returns an error.
≤ **x**. If the value of **x** is in the range of **X**, the VI performs interpolation. Otherwise, the VI performs extrapolation.

Note: If the x value is too far from the range of X, the interpolation error may be large. It is not a satisfactory extrapolation.

≤ **interpolation value**.
≤ **interpolation error** is an estimate of the error in the interpolation.
≤ **error**. See Analysis Error Codes for a description of the error.

Rational Interpolation (Advanced Only)

Interpolates or extrapolates f at **x** using a rational function.



≤ **Y Array** is the input array.
≤ **X Array**. If the number of elements in the **X Array** is different from the number of elements in the **Y Array**, the VI sets the output **interpolation value** and **interpolation error** to NaN and returns an error.
≤ **x**. If the value of x is in the range of **X**, the VI performs interpolation. Otherwise, the VI performs extrapolation.

Note: If the x value is too far from the range of X, the interpolation error may be large. It is not a satisfactory extrapolation.

≤ **interpolation value**.
≤ **interpolation error** is an estimate of the error in the interpolation.
≤ **error**. See Analysis Error Codes for a description of the error.

The rational function

$$\frac{P(x_i)}{Q(x_i)} = \frac{p_0 + p_1 x_i + \dots p_m x_i^m}{q_0 + q_1 x_i + \dots q_m x_i^m}$$

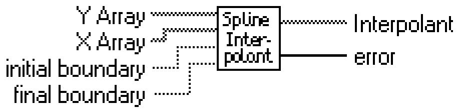
passes through all the points formed by ***Y* Array** and **X Array**. P and Q are polynomials, and the rational function is unique, given a set of n points ≤, where,

≤, f is any function, and given a number **x** in the range of the ≤ values. This VI calculates the output **interpolation value** y using $\frac{P(x_i)}{Q(x_i)}$. If the number of points is odd, the degrees of freedom of P and Q are using $\frac{n-1}{2}$. If the number of points is even, the degrees of freedom of P are $\frac{n}{2}-1$, and the degrees of freedom of Q are $\frac{n}{2}-1$, where n is the total number of points formed by **Y Array** and **X Array**.

Spline Interpolant (Advanced Only)

Returns an array **Interpolant** of length n, which contains the second derivatives of the spline interpolating function g(x) at the tabulated points ≤, where i = 0, 1,..., n-1. Input arrays **X Array** and **Y Array** are of length n and contain a tabulated function,

≤, with
$x_0 < x_1 < \ldots x_{n-1}$ **initial boundary** and **final boundary** are the first derivative of the interpolating function $g(x)$ at points 0 and n-1, respectively.



≤ **Y Array** is the input array.
≤ **X Array**. If the number of elements in the **X Array** is different from the number of elements in the **Y Array**, the VI sets the output Interpolant to an empty array and returns an error.
≤ **initial boundary** is the first derivative of interpolating function $g(x)$ at
≤. It defaults to 10^30, which causes this VI to set the initial boundary condition for a natural spline. For a definition of $g(x)$, see the discussion below.
≤ **final boundary** is the first derivative of interpolating function $g(x)$ at
$x_{n-1}, g'(x_{n-1})$. It defaults to 10^30 which causes this VI to set the final boundary condition for a natural spline.
≤ **Interpolant** is the second derivative of interpolating function g(x) at points
≤, i = 0, 1,..., n-1.
≤ **error**. See Analysis Error Codes for a description of the error.
If **initial boundary** and **final boundary** are equal to or greater than 10^30, the VI sets the corresponding boundary condition for a natural spline, with zero second derivative on that boundary.

The interpolating function $g(x)$ passes through all the points

$\{x_i, y_i\}\, g(x_i) = y_i$
where i = 0, 1,..., n-1.

The VI obtains the interpolating function $g(x)$ by interpolating every interval $[x_i, x_i + 1]$ with a cubic polynomial function

$p_i(x)$ that meets the following conditions:
1. $p_i(x_i) = y_i + 1$
2. $p_i(x_i + 1) = y_i + 1$
3. $g(x)$ has continuous first and second derivatives everywhere in the range $[x_0, x_{n-1}]$:
   a. $p_i'(x_i) = p_{i+1}'(x_i)$
   b. $p_i''(x_i) = p_{i+1}''(x_i)$

For the preceding conditions, i = 0, 1,..., n-2.

From condition 3(b), we derive the following equations:

$$\frac{x_i - x_{i-1}}{6} g''(x_i - 1) + \frac{x_{i+1} - x_{i-1}}{3} g''(x_i) + \frac{x_{i+1} - x_i}{6} g''(x_{i+1})$$
$$= \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \qquad i=1, 2, \ldots n-2 \qquad (7\text{-}9)$$

These are n -2 linear equations with n unknowns $g''(x_i)$

i = 0, 1,É, n - 1. This VI computes

$g''_{(x_0)}, g''_{(x_{n-1})}$ from **initial boundary** and **final boundary** using the formula

$$g'_{(x)} = \frac{y_{i+1} - y_i}{x_{i+1} - x_i} + \frac{3A^2 - 1}{6}(x_{i+1} - x_i)g''_{(x_i)}$$

$$+ \frac{3B^2 - 1}{6}(x_{i+1} - x_i)g''_{(x_{i+1})}.$$

Here

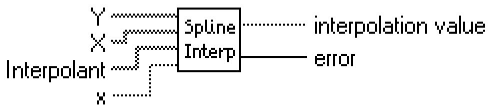
$$A = \frac{x_{i+1} - x_i}{x_{i+1} - x_i} \qquad B = 1 - A = \frac{x - x_i}{x_{i+1} - x_i}$$

You can derive this formula from the preceding conditions numbered 1Ð3. This VI then uses $\leq$ in equation (6-1) to solve all the

$\leq$, for i = 1, É n-2.
$\leq$ is the output **Interpolant**. You can use **Interpolant** as an input to the Spline Interpolation VI to interpolate *y* at any value of
$x_0 \leq x \leq x_{n-1}$.

Spline Interpolation (Advanced Only)

Performs a cubic spline interpolation of f at x, given a tabulated function.



$\leq$ **Y** is the array of tabulated values of the dependent variable.
$\leq$ **X** is the array of tabulated values of the independent variable.
$\leq$ **Interpolant** is the second derivative of the cubic spline interpolating function. You can obtain **Interpolant** from the Spline Interpolant VI.

Note: The number of elements in the three input arrays X, Y, and Interpolant should be the same. Otherwise, the VI sets the output interpolation value to NaN and returns an error.

$\leq$ **x** should be in the range of **X** values.
$\leq$ **interpolation value**.
$\leq$ **error**. See Analysis Error Codes for a description of the error.
This VI performs cubic spline interpolation using a tabulated function in the form of

$y_i = f(x_i)$ for i = 0, 1,..., n-1, and given the second derivatives **Interpolant** that the VI obtains from the Spline Interpolant VI. The value of **x** must be in the range of **X** values. The points are formed by the input arrays **X** and $\mathbf{Y}$, and n is the total number of points.

On the interval $[x_i, x_{i+1}]$, the output **interpolation value** y is defined by
$y = Ay_i + By_i + 1 + Cy''_i + Dy''_i + 1$
and

$$A = \frac{x_{i+1} - x}{x_{i+1} - x_i},$$

$$B = 1 - A,$$

$$C = \frac{1}{6}\left(A^3 - A\right)\left(x_{i+1} - x_i\right)^2,$$

$$D = \frac{1}{6}\left(B^3 - B\right)\left(x_{i+1} - x_i\right)^2$$

Exponential Fit.vi

Exponential Fit

Exponential Fit Coefficients.vi

Exponential Fit Coefficients

General LS Linear Fit.vi

General LS Linear Fit

General Polynomial Fit.vi

General Polynomial Fit

Linear Fit.vi

Linear Fit

Linear Fit Coefficients.vi

Linear Fit Coefficients

Nonlinear Lev-Mar Fit.vi

Nonlinear Lev-Mar Fit

Polynomial Interpolation.vi

Polynomial Interpolation

Rational Interpolation.vi

Rational Interpolation

Spline Interpolant.vi

Spline Interpolant

Spline Interpolation.vi

Spline Interpolation

Curve Fitting VIs Overview

For detailed VI descriptions, see Curve Fitting VIs.

Curve fitting analysis is a technique for extracting a set of curve parameters or coefficients from the data set to obtain a functional description of the data set. The algorithm that fits a curve to a particular data set is known as the Least Squares method and is discussed in most introductory textbooks in probability and statistics. The error is defined as

$$e(a) = \left[f(x,a) - y(x)\right]^2, \qquad (7\text{-}1)$$

where $e(a)$ is the error, $y(x)$ is the observed data set, $f(x,a)$ is the functional description of the data set, and a is the set of curve coefficients which best describes the curve.

For example, let $\{a_0, a_1\}$. Then the functional description of a line is

$$f(x,a) = \{a_0 + a_1 x\}$$

The least squares algorithm finds a by solving the system

$$\frac{\partial}{\partial a} e(a) = 0 \qquad (7\text{-}2)$$

To solve this system, you set up and solve the Jacobian system generated by expanding equation (7-2). After you solve the system for *a*, you can obtain an estimate of the observed data set for any value of *x* using the functional description $f(x,a)$.

In LabVIEW, the curve fitting VIs automatically set up and solve the Jacobian system and return the set of coefficients that best describes your data set. You can concentrate on the functional description of your data and not worry about solving the system in equation (7-2).

Two input sequences, Y Values and X Values, represent the data set $y(x)$. A sample or point in the data set is

$\leq$

where $\leq$ is the

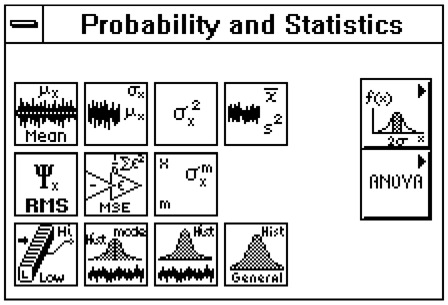
$\leq$ element of the sequence X Values, and
$\leq$ is the
$\leq$ element of the sequence Y Values.
In general, for each predefined type of curve fit, there are two types of VIs, unless otherwise specified. One type returns only the coefficients, so that you can further manipulate the data. The other type returns the coefficients, the corresponding expected or fitted curve, and the mean squared error (MSE). Because it is a discrete system, the VI calculates the MSE, which is a relative measure of the residuals between the expected curve values and the actual observed values, using the formula

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=0}^{n-1}\left(f_i - y_i\right)^2 \qquad (7\text{-}3)$$

where f is the sequence representing the fitted values, Y is the sequence representing the observed values, and n is the number of sample points observed.

Probability and Statistics VIs

This topic describes the VIs that perform probability, descriptive statistics, analysis of variance, and interpolation functions. The following illustration shows the options that are available on the **Probability and Statistics** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



General Histogram
Histogram
Mean
Median
Mode
Moment About Mean
MSE
RMS
Sample Variance
Standard Deviation
Variance

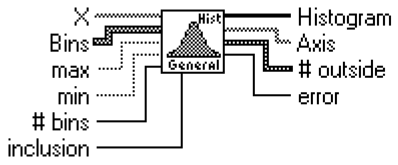
Subpalettes

Probability
Analysis of Variance

For examples of how to use the statistics VIs, see the examples located in `examples\analysis\statxmpl.llb`.

General Histogram (Advanced Only)

Finds the discrete histogram of the input sequence **X** based on the given bin specifications.



X represents the input data.

Bins specifies the boundaries of each bin of the histogram.
The input **Bins** is an array of clusters where each cluster defines the range of values for a bin. The cluster includes the following elements.

lower specifies the lower boundaries of the bin.
upper specifies the upper boundaries of the bin.
bin inclusion specifies how to treat the boundaries of each bin. The acceptable values for inclusion are listed below:

0: lower
1: upper
2: both
3: neither

Choosing 0 causes the lower boundary to be part of the bin but not the upper boundary. Choosing 1 is exactly opposite. Both boundaries can be included by choosing 2. Both boundaries can be excluded by choosing 3.

If no bin specifications are provided in the input **Bins**, the inputs **max**, **min**, **# bins**, and **inclusion** are used to specify a set of uniformly spaced bins.

≤ **max** specifies the maximum value to include in the histogram. This parameter is optional as explained below.
≤ **min** specifies the minimum value to include in the histogram. This parameter is optional as explained below.
If you leave the inputs **max** and **min** unwired, the maximum and minimum values in the input sequence **X** are used.

≤ **# bins** specifies the number of bins in the histogram. This parameter is optional as explained below.
If you leave **# bins** unwired, the number of bins is determined according to Sturges' Rule (number of bins = 1 + 3.3log(sizeof(X))).

≤ **inclusion** specifies how to handle the boundaries of each bin. The valid values for inclusion are:

0:	lower	includes the lower boundary
1:	upper	includes the upper boundary

Note: If array Bins is not empty, the max, min, # bins, and inclusion parameters are ignored.

≤ **Histogram** specifies the resulting histogram.
≤ **Axis** specifies the center values for each bin of Histogram.
The centers of each bin are set according to the following equation and returned in the output array **Axis**:

center[i] = (lower + upper)/2.

lower is the lower boundary of bin i.
upper is the upper boundary of bin i.

≤ **# outside** is the output cluster. **# outside** contains three elements.
≤ **total**. Upon successful execution, the element **total** contains the total number of points in **X** not falling in any bin.
The elements **above** and **below** have meaning only if **Bins** are specified such that **Bins**[0].upper =< **Bins**[1].lower < **Bins**[1].upper,...-<**Bins**[k-1].lower, and <**Bins**[k-1].upper

where k is the number of elements in **Bins**.

≤ **above** represents the number of values in **X** above **Bins**[sizeof(**Bins**)-1].upper.
≤ **below** represents the number of values in **X** below **Bins**[0].lower.
≤ **error**. See Analysis Error Codes for a description of the error.
The VI obtains the **Histogram** as follows. The VI establishes all the intervals (also called bins) based on the information in the input array **Bins** first. The intervals (bins) are:

Δ_i = (**Bins**[i].lower: **Bins**[i].upper) $i = 0, 1, 2, \ldots, k-1$
where

Bins[i].lower is the value lower in the $\leq$ cluster of array **Bins**, **Bins**[i].upper is the value upper in the

$\leq$ cluster of array **Bins**, k is the number of elements in **Bins**, which consists of the number of total intervals (bins).
Whether the two ending points **Bins**[i].lower and **Bins**[i].upper of each interval (bin) are included in the interval (bin) $\leq$ depends on the value of bin inclusion in the corresponding cluster i of the **Bins**.

If the array **Bins** is empty, the VI uses inputs max, min, and # bins to establish the intervals (bins). Each interval (bin) width Δ_x is the same. Use

$$\Delta x = \frac{max - min}{\#bins}$$

to calculate each interval (bin) width $\leq$. The intervals (bins) are as follows:

if **bin inclusion** = lower (including lower boundary).

$$\Delta_0 = [\text{minimum} + \Delta x), \Delta_1 = [\min + \Delta x : \min + 2\Delta x), \ldots, \Delta_i =$$

$$[\min + i\Delta x : \min + (i+1)\Delta x), \ldots, \Delta_{k-1} = [\min + (k-1)\Delta x : \max]$$

if **bin inclusion** = upper (including upper boundary).

$$\Delta_0 = [\min : \min + \Delta x], \Delta_1 = (\min + \Delta x : \min + 2\Delta x], \ldots, \Delta_i =$$

$$(\min + i\Delta x : \min + (i+1)\Delta x], \ldots, \Delta_{k-1} = (\min + (k-1)\Delta x : \max]$$

Note: The first start point min and last end point max are always included in the first and last intervals (Bins).

After establishing the intervals (**Bins**), the VI obtains the **Histogram** using the following formula.

Define the function to be

$$y_i(x) = \begin{cases} 1 & \text{if } x \in\supseteq \Delta_i \\ 0 & \text{elsewhere} \end{cases}$$

For example, if x falls into the interval (bin) $\leq$, then

$y_i(x) = 1$.
Finally, the VI evaluates the histogram sequence H using

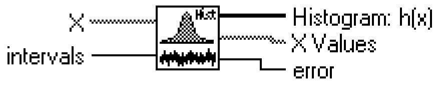
$$h_i = \sum_{j=0}^{n-1} y_i(x_j)$$

where H represents the elements of the output sequence **Histogram**, and n is the number of elements in the input sequence **X**. h_i is the total number of points in the input array **X** that fall into the interval (bin)

$\leq$, where i=0,1,...k-1. k is the number of bins.

Histogram

Finds the discrete histogram of the input sequence **X**. The histogram is a frequency count of the number of times that a specified interval occurs in the input sequence.



≤ **X** must contain at least one sample. If **X** is empty, the histogram is undefined, and the VI sets **Histogram: h(x)** and **X Values** to empty arrays and returns an error.
≤ **intervals** must be greater than zero. If **intervals** is less than or equal to zero, the histogram is undefined, and the VI sets **Histogram: h(X)** and **X Values** to empty arrays and returns an error. **intervals** defaults to 1.
≤ **Histogram: h(x)**.
≤ **X Values**.
≤ **error**. See Analysis Error Codes for a description of the error.
If the input sequence is

X = {0, 1, 3, 3, 4, 4, 4, 5, 5, 8},

then 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 VI obtains **Histogram: h(x)** as follows. The VI scans the input sequence **X** to determine the range of values in it. Then the VI establishes the interval width, *x*, 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**, and m is the specified number of **intervals**.

Let χ represent the output sequence **X Values**, because the histogram is a function of **X**. The VI evaluates elements of c using

$$\chi_i = \min + 0.5\Delta x + i\Delta x \qquad \text{for } i = 0,1,2,\ldots,m-1$$

The VI defines the ≤ interval to be the range of values from up to but not including,

$$\Delta_i = [\chi_i - 0.5\Delta x : \chi_i + 0.5\Delta x) \qquad \text{for } i = 0,1,2,\ldots,m-1$$

and defines the function to be

≤.

The function has unity value if the value of x falls within the specified interval. Otherwise it is zero. Notice that the interval ≤ is centered about

χ_i, and its width is
≤.

The last interval, Δ_{m-1}, is defined .In other words, if a value is equal to max, it is counted as belonging to the last interval.

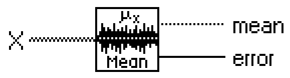
Finally, the VI evaluates the histogram sequence H using

$$h_i = \sum_{j=0}^{n-1} y_i(x_j) \qquad \text{for } i = 0,1,2,\ldots,m-1$$

where ≤ represents the elements of the output sequence **Histogram: h(x)**, and n is the number of elements in the input sequence **X**.

Mean

Computes the mean (average) of the values in the input sequence **X**.



≤ **X**. If the input sequence **X** is empty, **mean** is NaN.
≤ **mean**.
≤ **error**. See Analysis Error Codes for a description of the error.

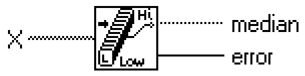
The VI computes **mean** (μ) using the following formula:

$$\mu = \frac{1}{n}\sum_{i=0}^{n-1} x_i$$

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

Median

Finds the median value of the input sequence **X** by sorting the values of **X** and selecting the middle element(s) of the sorted array.



≤ **X**. If the input sequence **X** is empty, **median** is NaN.
≤ **median**.
≤ **error**. See Analysis Error Codes for a description of the error.

Let n be the number of elements in the input sequence **X,** and let *S* be the sorted sequence of *X*. The VI finds **median** using the following identity:

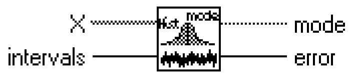
$$\text{median} = \begin{cases} s_i & \text{if n is odd} \\ 0.5\left(s_{k-1} + s_k\right) & \text{if n is even} \end{cases}$$

$$\text{where } i = \frac{n-1}{2},$$

$$\text{and } k = \frac{n}{2}.$$

Mode

Finds the **mode** of the input sequence **X**.



≤ **X** must contain at least one sample. If the **X** is empty, the histogram is undefined, the error returns via the Histogram VI, and the Mode VI sets **mode** to NaN.

Special Case: If the input sequence has a constant value, the Mode VI ignores the number of intervals and sets **mode** to the constant value in the inputsequence:

if **X** = a ⇒ **mode** = a.

≤ **intervals**. The number of **intervals** must be greater than zero. If the number of **intervals** is less than or equal to zero, the histogram is undefined, the error returns via the Histogram VI, and the Mode VI sets **mode** to NaN. **intervals** defaults to 1.

≤ **mode** is the value that occurs most often in a sequence of values. For example, if the input sequence is

X = {0, 1, 3, 3, 4, 4, 4, 5, 5, 7},

then the **mode** of **X** is 4 because that is the value that most often occurs in **X**.

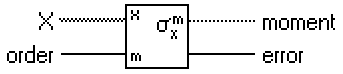
Because the VI finds **mode** with the aid of a histogram, you should read the Histogram VI description. The VI obtains **mode** as follows. The VI generates a discrete histogram $h(x)$ with the specified number of **intervals** of the input sequence **X** and then scans $h(x)$ for the interval ≤ that has the maximum count. Once the VI identifies the interval, the VI selects the center value of the interval as the **mode** of the input sequence **X**

$h(\text{mode}) = \max[h(x)]$.

≤ **error**. See Analysis Error Codes for a description of the error.

Moment About Mean (Advanced Only)

Computes the moment about the mean of the input sequence **X** using the specified **order**.



≤ **X**. If the input sequence **X** is empty, **moment** is NaN.
≤ **order** must be greater than zero. If **order** is less than or equal to zero, the VI sets **moment** to NaN and returns an error. **order** defaults to 2.
≤ **moment**.
≤ **error**. See Analysis Error Codes for a description of the error.
Let m be the desired **order**. The VI computes the m^{th}-order **moment** using the formula:

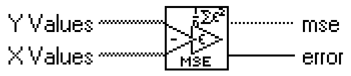
$$\sigma_x^m = \frac{1}{n}\sum_{i=0}^{n-1}\left(x_i - \mu\right)^2$$

where σ_x^m is the

≤ -order **moment**, and n is the number of elements in the input sequence **X**.

MSE (Advanced Only)

Computes the mean squared error (mse) of the input sequences **X Values** and **Y Values**.



≤ **Y Values**.
≤ **X Values**. If the number of elements in **X Values** is different from the number of elements in **Y Values**, the VI computes **mse** based on the sequence that contains the fewest elements.
≤ **mse** is the mean squared error. If one of the input sequences is an empty array, the value of **mse** is NaN.
≤ **error**. See Analysis Error Codes for a description of the error.
The VI uses the following formula to find **mse**:

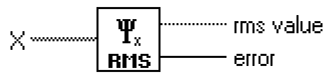
$$mse = \frac{1}{n}\sum_{i=0}^{n-1}\left(x_i - y_i\right)^2 ,$$

,

where n is the number of data points.

RMS (Advanced Only)

Computes the root mean square (rms) of the input sequence **X**.



≤ **X**. If the input sequence **X** is empty, **rms value** is NaN.

≤ **rms value**.

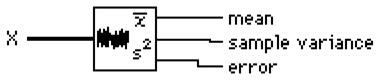
≤ **error**. See Analysis Error Codes for a description of the error.The VI computes the rms value (ψ_x) using the following formula:

$$\psi_x = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1} x_i^2},$$

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

Sample Variance (Advanced Only)

Computes the mean and sample variance of the values in the input sequence X.



≤ **X** contains the input array of samples.

≤ **mean** the mean of the input array of samples in **X**. If n is the number of samples **in** X, then the sample mean is computed as

$$\textbf{mean} = \frac{\sum_{i=0}^{n-1} x_i}{n}$$

≤ **sample variance** is the sample variance of the input array of samples in **X**. If n is the number of samples in **X**, then the sample variance is computed as

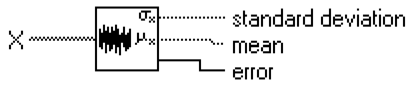
$$\textbf{samplevariance} = \frac{\sum_{i=0}^{n-1}(x_i - \text{mean})^2}{n-1}$$

≤ **error**. See Analysis Error Codes for a description of the error.

Note: If you need to compute the sample standard deviation of X, simply take the square root of sample variance.

Standard Deviation

Computes the mean value and the standard deviation of the values in the input sequence **X**.



≤ **X**. If the input sequence **X** is empty, **standard deviation** and **mean** are NaN.

≤ **standard deviation**.

≤ **mean**.

≤ **error**. See Analysis Error Codes for a description of the error.

The VI computes **standard deviation** (σ_x) and **mean**

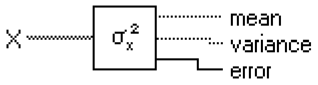
(μ) using the following formula:

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2} \; ,$$

where $\mu = \frac{1}{n}\sqrt{\sum_{i=0}^{n-1} x_i}$, and n is the number of elements in **X**.

Variance (Advanced Only)

Computes the variance and the mean value of the input sequence **X**.



≤ **X**. If the input sequence **X** is empty, **variance** and **mean** are NaN.
≤ **mean**.
≤ **variance**.
≤ **error**. See Analysis Error Codes for a description of the error.

The VI computes **variance** σ_x^2 and **mean**

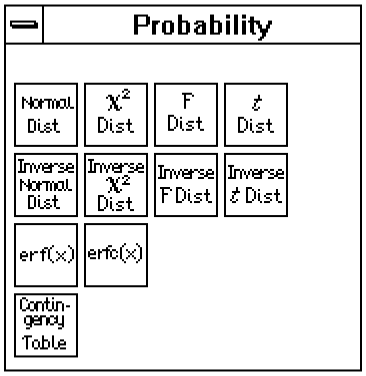
≤ using the following formula:

$$\sigma_x = \sqrt{\frac{1}{n}\sum_{i=0}^{n-1}(x_i - \mu)^2} \; ,$$

where $\mu = \frac{1}{n}\sqrt{\sum_{i=0}^{n-1} x_i}$, and n is the number of elements in **X**.

Probabiliity

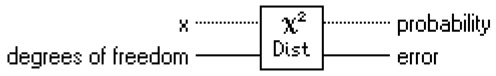
The following illustration shows the options that are available on the **Probability** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Chi Square Distribution
Contingency Table
erf(x)
erfc(x)
F Distribution
Inverse Chi Square Distribution
Inverse F Distribution
Inverse Normal Distribution
Inverse T Distribution
Normal Distribution
T Distribution

Chi Square Distribution (Advanced Only)

Computes the one-sided **probability**, p, of the distributed random variable, *x*, with the specified **degrees of freedom**.



$$p = \Pr ob\{X \leq \mathbf{x}\}$$

where X is ≤ distributed with n **degrees of freedom**, p is the **probability**, n is **degrees of freedom**, and **x** is the value.

≤ **x**.
≤ **degrees of freedom** must be greater than zero: n > 0. If **degrees of freedom** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **probability** must be greater than or equal to zero and less than or equal to one:

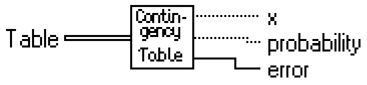
$$0.0 \leq p \leq 1.0$$

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

≤ **error**. See Analysis Error Codes for a description of the error.

Contingency Table (Advanced Only)

Classifies and tallies objects of experimentation according to two schemes of categorization.



$\leq$ **Table**.

$\leq$ **x** specifies the value at which you wish to interpolate a corresponding y value.

$\leq$ **probability** must be greater than or equal to zero and less than or equal to one:

$\leq$

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

$\leq$ **error**. See Analysis Error Codes for a description of the error.

With the $\leq$ test of homogeneity, the VI takes a random sample of some fixed size from each of the categories in one categorization scheme. For each of the samples, the VI categorizes the objects of experimentation according to the second scheme, and tallies them. The VI tests the hypothesis to determine whether the populations from which each sample is taken are identically distributed with respect to the second categorization scheme.

With the $\leq$ test of independence, the VI takes only one sample from the total population. The VI then categorizes each object and tallies it in two categorization schemes. The VI tests the hypothesis that the categorization schemes are independent.

You must choose a level of significance for each test. This is how likely you want it to be that the VI rejects the hypothesis when it is true. Ordinarily, you do not want it to be very likely. So you should use a small number (0.05 or 5 percent is a common choice) to determine the level of significance. The output parameter **probability** is the level of significance at which the hypothesis is rejected. Thus, if **probability** is less than the level of significance, you must reject the hypothesis.

Formulas

Let $y_{p,q}$ be the number of occurrences in the

$(pq)^{th}$ cell of the contingency table for
p = 0, 1,..., (s-1) and q = 0, 1,..., (k-1),

where s is the number of rows in the Contingency **Table**, and k is the number of columns in the Contingency **Table**.

Let

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

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

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

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

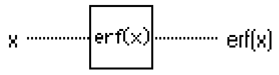
$$x = \sum_{p=0}^{s-1} \sum_{q=0}^{k-1} \frac{\left[y_{p,q} - e_{p,q}\right]^2}{e_{p,q}}$$

The VI uses **x** to calculate the **probability** $p = \text{Prob}\{X \geq \mathbf{x}\}$, where X is a random variable from the

≤ distribution. If the hypothesis is true, **x** came from a
≤ distribution with (s-1) and (k-1) degrees of freedom.

erf(x) (Advanced Only)

Evaluates the error function at the input value.

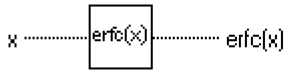


≤ **x**.
≤ **erf(x)** is accurate to 15 decimal places.

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x \exp\left(-t^2\right) dt$$

erfc(x) (Advanced Only)

Evaluates the complementary error function at the input value.



≤ **x**.
≤ **erfc (x)** is accurate to 15 decimal places**.**

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_x^\infty \exp\left(-t^2\right) dt$$

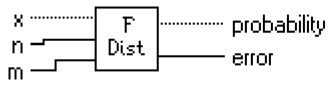
erfc (x) = 1 - erf (x).

F Distribution (Advanced Only)

Computes the one-sided **probability**, p, of the F-distributed random variable, F, with the specified **n** and **m** degrees of freedom

$$p = \text{Prob}\left\{F_{n,m} \leq x\right\}$$

where F is F-distributed, p is the **probability**, n specifies the first degree of freedom, **m** specifies the second degree of freedom, and **x** is the value.



≤ **x**.
≤ **n**. The degrees of freedom **n** must be greater than zero: n > 0. If **n** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **m**. The degrees of freedom **m** must be greater than zero: m > 0. If **m** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **probability** must be greater than or equal to zero and less than or equal to one:
≤

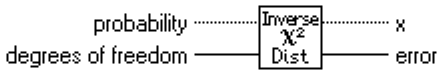
If **probability** is out of range, the VI sets **x** to NaN and returns an error.

≤ **error**. See Analysis Error Codes for a description of the error.

Inverse Chi Square Distribution (Advanced Only)

Computes the value of **x** such that the condition

≤
is satisfied, given the **probability** value, p, of a ≤-distributed random variable, X, with n **degrees of freedom**.



≤ **probability** must be greater than or equal to zero and less than or equal to one:
≤

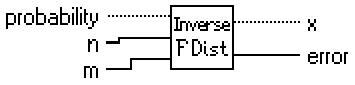
If **probability** is out of range, the VI sets **x** to NaN and returns an error.

I16 **degrees of freedom** must be greater than zero: n > 0. If **degrees of freedom** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **x**.
≤ **error**. See Analysis Error Codes for a description of the error.

Inverse F Distribution (Advanced Only)

Computes the value of **x** such that the condition

$p = \mathrm{Prob}_{n,m} < X$
is satisfied, given the **probability** value p of an F-distributed random variable, F, with **n** and **m** degrees of freedom.



≤ **probability** must be greater than or equal to zero and less than or equal to one:
≤

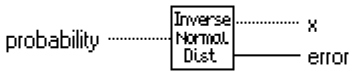
If **probability** is out of range, the VI sets **x** to NaN and returns an error.

≤ **n**. The degrees of freedom **n** must be greater than zero: n > 0. If **n** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **m**. The degrees of freedom **m** must be greater than zero: m > 0. If **m** is less than or equal to zero, the VI sets **x** to NaN and returns an error.
≤ **x**.
≤ **error**. See Analysis Error Codes for a description of the error.

Inverse Normal Distribution (Advanced Only)

Computes the value of **x** such that the condition

≤
is satisfied, given the **probability** value, p, of a Normally distributed random variable, X.



≤ **probability** must be greater than or equal to zero and less than or equal to one:
≤

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

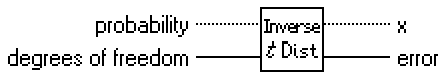
≤ **x**.
≤ **error**. See Analysis Error Codes for a description of the error.

Inverse T Distribution (Advanced Only)

Computes the value of **x** such that the condition

$$p = \text{Prob}\{T_n \le \mathbf{x}\}$$

is satisfied, given the **probability** value, p, of a T-distributed random variable, T, with n **degrees of freedom**.



≤ **probability** must be greater than or equal to zero and less than or equal to one:

≤

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

≤ **degrees of freedom** must be greater than zero: n > 0. If **degrees of freedom** is less than or equal to zero, the VI sets **x** to NaN and returns an error.

≤ **x**.

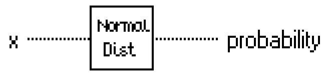
≤ **error**. See Analysis Error Codes for a description of the error.

Normal Distribution (Advanced Only)

Computes the one-sided **probability**, p, of the normally distributed random variable, **x**,

≤

where X is standard Normally distributed, p is the **probability**, and **x** is the value.



≤ **x**.

≤ **probability** must be greater than or equal to zero and less than or equal to one:

≤

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

This function computes only the one-sided probability. You can obtain the two-sided probability $(p = \text{Prob}\{-x \le X \le \mathbf{x}\})$ p_1, using the following formula:

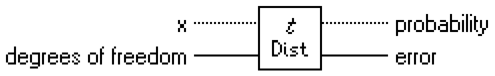
$$p_1 = 1 - 2(1 - p) = 2p - 1 = 2\,\text{Prob}\{X \le x\} - 1$$

T Distribution (Advanced Only)

Computes the one-sided **probability**, p, of the t-distributed random variable, Tn, with the specified **degrees of freedom**

≤

where T is t-distributed, p is the **probability**, n is **degrees of freedom**, and **x** is the value.



≤ **x**.

≤ **degrees of freedom** must be greater than zero: n > 0. If **degrees of freedom** is less than or equal to zero, the VI sets **x** to NaN.

≤ **probability** must be greater than or equal to zero and less than or equal to one:

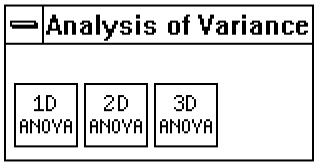
≤

If **probability** is out of range, the VI sets **x** to NaN and returns an error.

error. See Analysis Error Codes for a description of the error.

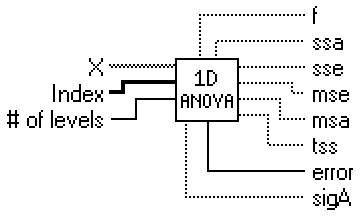
Analysis of Variance

The following illustration shows the options that are available on the **Probability and Statistics** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



1D ANOVA (Advanced Only)

2D ANOVA takes an array, **X**, of experimental observations made at various **levels** of a factor, with at least one observation per level, and performs a one-way analysis of variance in the fixed effect model. In the one-way analysis of variance, the VI tests whether the level of the factor has an effect on the experimental outcome.



- **X** contains all the observational data.
- **Index** contains the level to which the corresponding observation belongs.
- **# of levels** is the total number of levels.
- **f**, **ssa**, **sse**, **mse**, **msa**, **tss**, **sigA**. See the following discussion of this VI for more information about these return values.
- **error**. See Analysis Error Codes for a description of the error.

A factor is a basis for categorizing data. For example, if you count the number of sit-ups individuals can do, one basis of categorization is age. For age, you might have the following levels.

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

Now, suppose that you make a series of observations to see how many sit-ups people can do. If you take a random sampling of five people, you might find the following results.

Person 1	8 years old (level 0)	10 sit-ups

Person 2	12 years old (level 1)	15 sit-ups
Person 3	16 years old (level 2)	20 sit-ups
Person 4	20 years old (level 2)	25 sit-ups
Person 5	13 years old (level 1)	17 sit-ups

Notice that you have made at least one observation per level. To perform an analysis of variance, you must make at least one observation per level.

To perform the analysis of variance, you specify an array **X** of observations, with values 10, 15, 20, 25, and 17. The array **Index** specifies the level (or category) to which each observation applies. In this case, Index has the values 0, 1, 2, 2, and 1. Finally, there are three possible levels, so you pass in a value of 3 for the **# of levels** parameter.

The Statistical Model

Performing the analysis of variance, you express each experimental outcome as the sum of three parts. Let x_{im} be the

≤ observation from the
≤ level. Then each observation is written
$x_{im} = \mu + \alpha_i + \varepsilon_{im}$
where ≤ is a standard effect, called the overall mean.

α_i is the effect of the
≤ level of the factor, which is the difference between the mean of the
≤ level
α_i and the overall mean
$\mu\left(\mu_i = \mu + \alpha_i\right)$, and
ε_{im} is a random fluctuation.

Assumptions

Assume that the populations of measurements at each level are Normally distributed with mean μ_i and variance

α_A^2, and assume that
≤sum to zero. Finally, assume that for each i and m,
≤is Normally distributed with mean 0 and variance
α_A^2.

The Hypothesis

This VI tests the hypothesis that ≤= 0 for i = 0, 1,..., k-1 (where k is **# of levels**). In other words, this hypothesis, referred to as the null hypothesis, states that no level affects the experimental outcome, and then looks for evidence to the contrary.

The General Method

This VI computes the total sum of squares, **tss**, which is a measure of the total variation of the data from the overall population mean.

tss consists of two parts: ssa, a measure of variation attributed to the factor, and sse, a measure of variation attributed to random fluctuation. In other words, **tss** = **ssa** + **sse**,

The VI computes the two mean square quantities msa and mse from ssa and sse by dividing ssa and sse by their own degrees of freedom. The larger msa is relative to mse, the more significant effect the factor has on the experimental outcome.

In particular, if the null hypothesis is true, then the ratio **f**, f = msa/mse, 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**, **sigA** is the probability that you get a value larger than **f** when sampling from this distribution.

Testing the Hypothesis

How do you know when to reject the null hypothesis? You decide how likely you want it to be that you mistakenly reject the null hypothesis. This is the level of significance (a common choice is 0.05.). The output **sigA** is compared to the chosen level of significance to determine whether to accept or reject the null hypothesis. If **sigA** is less than the chosen level of significance, you should reject the null hypothesis. If you reject the null hypothesis, you must acknowledge that at least one level has some effect on the experimental outcome.

Formulas

Let ≲= the

≲ observation made at the
≲ level for $m = 0, 1, ...,$
$n_i - 1$ and $i = 0, 1, ..., k$-1, where
$n_i$is the number of observations at the
≲level and k = **# of levels**.

$$X_{i\bullet} = \sum_{m=0}^{n_i-1} x_{im}$$

$$X_{\bullet\bullet} = \sum_{i=0}^{k-1} \sum_{m=0}^{n_i-1} x_{im}$$

$$n = \sum_{i=0}^{k-1} n_i$$

then

$$ssa = \sum_{i=0}^{k-1}\left(\frac{X_{i\bullet}^2}{n_i}\right) - \frac{X_{\bullet\bullet}^2}{n}$$

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

$$sse = \sum_{i=0}^{k-1}\sum_{m=0}^{n_i-1} x_{im}^2 - \sum_{i=0}^{k}\left(\frac{X_{i\bullet}^2}{n_i}\right)$$

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

$$tss = \sum_{i=0}^{k-1}\sum_{m=0}^{n_i-1} x_{im}^2 - \frac{X_{\bullet\bullet}^2}{n}$$

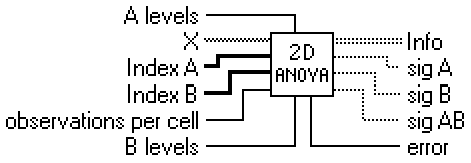
$$f = \frac{msa}{mse}$$

$$SigA = Prob\left\{F_{k-1,n-k} > f\right\}$$

$F_{k-1,n-k}$ is the F distribution with k-1 and n-k degrees of freedom.

2D ANOVA (Advanced Only)

2D ANOVA takes an array of experimental observations made at various levels of two factors and performs a two-way analysis of variance.



≤ **A levels** contains the number of levels in factor A. The sign of **A levels** is set to positive if A is fixed, and negative if A is random.
≤ **X** contains all the observation data.
≤ **Index A** contains the level of factor A to which the corresponding observation belongs.
≤ **Index B** contains the level of factor B to which the corresponding observation belongs.
≤ **observations per cell** is the number of observations in each cell. It is the same for all cells.
≤ **B levels** contains the number of levels in factor B. The sign of **B levels** is set to positive if B is fixed, and negative if B is random.
≤ **Info** is a 4-by-4 matrix organized as follows:

$$\text{Info} = \begin{bmatrix} ssa & dofa & msa & fa \\ ssb & dofb & msb & fb \\ ssab & dofab & msab & fab \\ sse & dofe & mse & 0.0 \end{bmatrix}$$

where

- The first column corresponds to the sum of squares associated with factor A, factor B, AB interaction, and residual error.
- The second column corresponds to the respective degrees of freedom.
- The third column corresponds to the respective mean squares.
- The fourth column corresponds to the respective F values.

≤ **sig A** is the computed level of significance associated with factor A.
≤ **sig B** is the computed level of significance associated with factor B.
≤ **sig AB** is the computed level of significance associated with the interaction of factors A and B.
≤ **error**. See Analysis Error Codes for a description of the error.

Factors, Levels, and Cells

A factor is a basis for categorizing data. For example, if you count the number of sit-ups individuals can do, one basis of categorization is age. For age, you might have the following levels.

level 0:	6 years old to 10 years old
level 1:	11 years old to 15 years old

Another possible factor is weight, with the following levels.

level 0:	less than 50 kg
level 1:	between 50 and 75 kg
level 2:	more than 75 kg

Now, suppose that you made a series of observations to see how many sit-ups people could do. If you took a random sampling of n people, you might find the following results:

Person 1	8 years old (level 0)	30 kg (level 0)	10 sit-ups
Person 2	12 years old (level 1)	40 kg (level 0)	15 sit-ups
Person 3	15 years old (level 1)7	6 kg (level 2)	20 sit-ups
Person 4	14 years old (level 1)	60 kg (level 1)	25 sit-ups
Person 5	9 years old (level 0)	51 kg (level 1)	17 sit-ups
Person 6	10 years old (level 0)	80 kg (level 2)	4 sit ups

and so on.

If you plot observations as a function of factor A and factor B, they fall into cells of a matrix with factor A as rows and factor B as columns. Each cell must contain at least one observation, and each cell must contain the same number of observations.

To perform the analysis of variance, you specify an array **X** of observations, with values 10, 15, 20, 25, 17, and 4. The array **Index A** specifies the level (or category) of factor A to which each observation applies. In this case, the array would have the values 0, 1, 1, 1, 0, and 0.

The array **Index B** specifies the level (or category) of factor B to which each observation applies. In this case, the array would have the values
0, 0, 2, 1, 1, and 2. Finally, there are two possible levels for factor A and three possible levels for factor B, so you pass in a value of 2 for the **A levels** parameter, and a value of 3 for the **B levels** parameter.

You can apply any one of the following models, where L is the specified **observations per cell**.

- Model 1: Fixed-effects with no interaction and one observation per cell (per specified levels x and y 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 with interaction and L>1 observations per cell.
- Model 4: Random-effects with interaction and L>1 observations per cell.

Random and Fixed Effects

A factor is a random effect if it has a large population of levels about which you want to draw conclusions, but such that you cannot sample from all levels. You thus pick levels at random and hope to generalize about all levels. A factor is a fixed effect if you can sample from all levels about which you want to draw conclusions.

The input parameters **A levels** and **B levels** represent the number of levels in factors A and B, respectively as well as whether the factors are random or fixed. If, for instance, factor A is random, you set **A levels** to be negative the number of levels in factor A. Notice that if there is only one observation per cell, both **A levels** and **B levels** must be positive. That is, you use model 1.

The General Method

In each of the models, the VI breaks up the total sum of squares, tss, a measure of the total variation of the data from the overall population mean, into some number of component sums of squares. In model 1 tss = ssa + ssb + sse,

whereas in models 2 through 4

tss = ssa + ssb + ssab + sse.

Each component sum in tss is a measure of variation attributed to a certain factor or interaction among the factors. Here ssa is a measure of the variation due to factor A, ssb is a measure of the variation due to factor B, ssab is a measure of the variation due to the interaction between factors A and B, and sse is a measure of the variation due to random fluctuation. Notice that with model 1 you have no ssab term. This is what no interaction means.

The VI divides each of the values ssa, ssb, ssab, and sse by their own degrees of freedom to compute the mean square quantities msa, msb, msab, and mse. If one factor, such as factor A, has a strong effect on the experimental observations, the respective mean square quantity msa will be relatively large.

The Statistical Model

Let x_{pqr} be the

r^{th} observation at the
p^{th} and
q^{th} levels of A and B respectively, where r = 0,1,...,L-1.
Model 1 expresses each observation as the sum of four components.

$x_{pqr} = \mu + \alpha_p + \beta_q + \varepsilon_{pqr}$
$x_{pqr} = \mu + \alpha_p + \beta_q + (\alpha\beta)_{pq} + \varepsilon_{pqr}$
Models 2, 3, and 4 express each observation as the sum of five components.

$\leq$
where β_q $\mu_q - \mu$

- $\leq$ is the overall mean response (the average of the mean response for all the populations).
- α_p is the effect of the
$\leq$ level of factor A (equal to
$\mu_p - \mu$ where
μ_p is the average of the
$\leq$ level of factor A over all levels of factor B).
- β_q is the effect of the
$\leq$level of factor B (equal to
$\mu_q - \mu$ where
μ_q is the average of the
$\leq$ level of factor B over all levels of factor A).
- $(\alpha\beta)_{pq}$ is the interaction between the
$\leq$ level of factor A and the
$\leq$ level of factor B (equal to
$\mu_{pq} - (\mu + \alpha_p + \beta_q)$ where
μ_{pq} is the population mean of the
pq^{th} cell).
- ε_{pqr} is the deviation of

≤from the population mean response for the
≤ population.

Assumptions

- Assume that for each p, q, and r, ≤ is Normally distributed with mean 0 and variance

σ^2_{e}.
- If a factor such as A is fixed, assume that the populations of measurements at each level of A are Normally distributed with mean $\alpha_p + \mu$ and variance

≤, and that all the populations at each of the levels have the same variance. In addition, assume that
≤ sum to zero. Analogous assumptions are made for B.
- If a factor such as A is random, assume that the effect of the level of A itself, ≤, is a random variable Normally distributed with mean 0 and variance

≤. Analogous assumptions are made for B.
- If all of the factors such as A and B associated with the effect of an interaction ≤ are fixed, assume that the populations of measurements at each level are Normally distributed with mean

≤ and variance
σ^2_{AB}. For any fixed p,
≤ sum to zero, when summing over all q. Similarly, for any fixed q, the 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 ≤ are random, assume the effect is a random variable Normally distributed with mean 0 and variance

≤. If A is fixed but B is random, then also assume that for any fixed q the means
≤ sum to zero, when summing over all p. Similarly, if B is fixed but A is random, assume that for any fixed p the means
≤ sum to zero, when summing over all q.
- Assume that all effects taken to be random variables are mutually independent.

The Hypotheses

Each of the following hypotheses is a different way of saying that a factor or an interaction among factors has no effect on experimental outcomes. This VI assumes that there are no effects and then seeks evidence to contradict this assumption. The three hypotheses are:

- (A) that $\alpha_p = 0$ for all levels p if factor A is fixed, and that

$\sigma^2_A = 0$ if factor A is random.
- (B) that $\beta_q = 0$ for all levels q if factor B is fixed, and that

$\sigma^2_B = 0$ if factor B is random.
- (AB) that $(\alpha\beta)_{pq} = 0$ for all levels p and q if both factors A and B are fixed, and that

≤ if either factor A or factor B is random. (This does not apply to model 1. In model 1 there is no interaction, and the associated output parameters are superfluous.)

The Testing of Hypotheses

For each hypothesis, the VI computes a number f that is used to calculate the associated sig probability. For example, for the hypothesis (A), that ≤ for all the levels p, (fixed A), the VI computes

$$fa = \frac{msa}{mse}$$

then $sigA = \Pr ob\left\{F_{a-1,(a-1)(b-1)} > fa\right\}$

where $F_{a-1,(a-1)(b-1)}$ is an F distribution with degrees of freedom a-1 and (a-1)(b-1). You can then use the probabilities **sigA**, **sigB**, and **sigAB** to determine when you should reject the associated hypotheses (A), (B), and (AB).

How do you know when to reject the null hypothesis? For each hypothesis, you choose a level of significance. This level of significance is how likely you want it to be that you mistakenly reject the hypothesis (a common choice is 0.05). Compare your chosen level of significance with the associated sig probability output. If the sig probability is less than your chosen level of significance, you should reject the null hypothesis.
For example, if A is a random effect, your chosen level of significance is 0.05, and the output **sigA** is 0.03, then you must reject the hypothesis $\alpha A^2 = 0$ and conclude that factor A has an effect on the experimental observations.

Formulas

Let $\leq$ be the

$\leq$ observation at the
$\leq$and
$\leq$levels of A and B respectively, where r= 0,1,...,L-1.
Let

$$a = |A\ levels|$$

$$b = |B\ levels|$$

$$T_{pq\bullet} = \sum_{r=0}^{L-1} X_{pqr}$$

$$T_{p\bullet\bullet} = \sum_{q=0}^{b-1} X_{pq\bullet}$$

$$T_{\bullet q\bullet} = \sum_{p=0}^{a-1} X_{pq\bullet}$$

$$T = \sum_{p=o}^{a-1} \sum_{q=0}^{b-1} \sum_{r=0}^{L-1} X_{pqr}^2$$

$$A = \sum_{p=0}^{a-1} \frac{T_{p\bullet\bullet}^2}{bL}$$

$$B = \sum_{q=0}^{b-1} \frac{T_{\bullet q\bullet}^2}{aL}$$

$$S = \sum_{p=0}^{a-1} \sum_{q=0}^{b-1} T_{pq\bullet}^2$$

$$CF = \frac{(\text{total sum of all observations})^2}{abL}$$

then

$$ssa = A - CF \quad msa = \frac{ssa}{a-1} \quad dofa = a - 1$$

$$ssb = B - CF \quad msb = \frac{ssb}{b-1} \quad dofb = b - 1$$

$$ssab = S - A - B - CF \quad msab = \frac{ssab}{(a-1)(b-1)}$$

dofab = (a-1)(b-1) if L>1

dofab = 0 if L=1

$$sse = T - s \quad mse = \frac{sse}{(ab(L-1))}$$

dofe = ab(L-1) if L>1

dofe = (a-1)(b-1) if L=1

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

$$fa = \frac{msa}{mse} \quad \text{if B is fixed}$$

$$fa = \frac{msa}{msab} \quad \text{if B is random}$$

$$fb = \frac{msb}{mse} \quad \text{if A is fixed}$$

$$fb = \frac{msb}{msab} \quad \text{if A is random}$$

$$sigA = \Pr ob\left\{F_{a-1,ab(L-1)} > fa\right\} \quad \text{if B is fixed}$$

$$sigA = \Pr ob\left\{F_{a-1,(a-1)(b-1)} > fa\right\} \quad \text{if B is random}$$

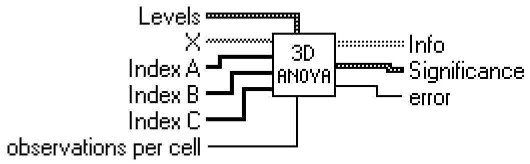
$$sigB = \Pr ob\left\{F_{a-1,ab(L-1)} > fb\right\} \quad \text{if A is fixed}$$

$$sigB = \Pr ob\left\{F_{a-1,(a-1)(b-1)} > fb\right\} \quad \text{if A is random}$$

$$sigAB = \Pr ob\left\{F_{(a-1)(b-1),ab(L-1)} > fab\right\}$$

3D ANOVA (Advanced Only)

3D ANOVA takes an array of experimental observations made at various levels of three factors and performs a three-way analysis of variance. In any ANOVA, you look for evidence that the factors or interactions among factors have a significant effect on experimental outcomes. What varies with each model is the method used to do this. The three-way ANOVA models are as follows, where L is the number of **observations**.



- Fixed-effects with interaction and L>1 observations per cell
- Any of the six mixed-effects models with interaction and L>1 observations per cell, and
- Random-effects with interaction and L>1 observations per cell

A factor is a basis for categorizing data. A cell of data consists of all those experimental observations that fall in particular levels of the three factors. The number of observations that fall in a cell must be some constant number L, which does not vary between cells. See the description of factors, levels, and cells in the 2D ANOVA VI description. Remember that a cell in this 3D ANOVA VI is the intersection of three factors instead of two as described in the 2D ANOVA VI description.

≤ **Levels** is a cluster of three numeric values corresponding to number of levels in the A, B, and C factors, as well as the effects of the A, B, and C factors (fixed or random).
≤ **Level A** is the number of levels in A if A is fixed, or the negative number of levels in A if A is random.
≤ **Level B** is the number of levels in B if B is fixed, or the negative number of levels in B if B is random.
≤ **Level C** is the number of levels in C if C is fixed, or the negative number of levels in C if C is random.
≤ **X** contains all the observation data.
≤ **Index A** contains the level of factor A to which the corresponding observation belongs.
≤ **Index B** contains the level of factor B to which the corresponding observation belongs.
≤ **Index C** contains the level of factor C to which the corresponding observation belongs.
≤ **observations per cell** is the number of observations in every cell. It is the same for all the cells.
≤ **Info**. The output 2D array **Info** is an 8 by 4 matrix organized as follows:

$$\text{Info} = \begin{bmatrix} ssa & dofa & msa & fa \\ ssb & dofb & msb & fb \\ ssc & dofc & msc & fbc \\ ssab & dofab & msab & fab \\ ssac & dofac & msac & fac \\ ssbc & dofbc & msbc & fbc \\ ssabc & dofabc & msabc & fabc \\ sse & dofe & mse & 0.0 \end{bmatrix}$$

where the first column corresponds to the sums of squares associated with the respective factors (A, B, C), the respective interactions (AB, AC, BC, ABC), and residual error,

the second column corresponds to the respective degrees of freedom,

the third column corresponds to the respective mean squares, and

the fourth column corresponds to the respective F values.

≤ **Significance** is a cluster of seven numerical values corresponding to the significance levels.
≤ **sigA** is the computed level of significance associated with factor A.
≤ **sigB** is the computed level of significance associated with factor B.
≤ **sigC** is the computed level of significance associated with factor C.

≤ **sigAB** is the computed level of significance associated with the interaction of factors A and B.
≤ **sigAC** is the computed level of significance associated with the interaction of factors A and C.
≤ **sigBC** is the computed level of significance associated with the interaction of factors B and C.
≤ **sigABC** is the computed level of significance associates the interaction of factors A, B and C.
≤ **error**. See Analysis Error Codes for a description of the error.

Random and Fixed Effects

A factor is a random effect if it has a large population of levels about which you want to draw conclusions, but such that you cannot sample from all levels. You thus pick levels at random and hope to generalize about all levels. A factor is a fixed effect if you can sample from all levels about which you want to draw conclusions.

The General Method

In each of the models, the VI breaks up the total sum of squares, tss, a measure of the total variation of the data from the overall population mean, into a number of component sums of squares.

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

Each component in the sum tss is a measure of variation attributed to a certain factor or interaction among the factors. Here ssa is a measure of the variation due to factor A; ssb is a measure of the variation due to factor B; ssc is a measure of the variation due to factor c; ssab is a measure of the variation due to the interaction between factors A and B; and so on for ssac, ssbc, and ssabc. Also, sse is a measure of the variation due to random fluctuation. The VI divides each by its own degrees of freedom to obtain the corresponding averages msa, msb, msc, msab, msac, msbc, msabc, and mse. If, for instance, factor A has a strong effect on the experimental observations, then msa will be relatively large.

The Statistical Model

Let x_{pqrs} equation be the

s^{th} observation at the
≤,
≤, and
≤ levels of A, B, and C respectively, where s = 0, 1,..., L-1. Express each observation as the sum of eight components. Thus,
$$x_{pqrs} = \mu + \alpha_p + \beta_q + \gamma_r + (\alpha\beta)_{pq} + (\alpha\gamma)_{pr} + (\beta\gamma)_{qr} + (\alpha\beta\gamma)_{pqr} + \varepsilon_{pqrs}$$
where

- ≤ is the overall mean.
- ≤ is the average effect of the
≤ level of factor A.
- ≤ is the average effect of the
≤ level of factor B.
- γ_r is the average effect of the
≤ level of factor C.
- $(\alpha\beta)_{pq}$is the two-factor interaction of the
≤level of factor A with the
≤ level of factor B.
- $(\alpha\gamma)_{pr}$ is the two-factor interaction of the
≤ level of factor A with the rthlevel of factor C.

- $(\beta\gamma)_{qr}$is the two-factor interaction of the

≤ level of factor B with the rth level of factor C.
- $(\alpha\beta\gamma)_{pqr}$ is the three-factor interaction of the

≤ level of factor A, the
≤ level of factor B, and the
≤level of factor C.
- $\varepsilon_{pqrs}$is random fluctuation.

Assumptions

- Assume that for each p, q, and r, ≤ is Normally distributed with mean 0 and variance

≤.
- If a factor, for instance, A, is fixed, assume the populations of measurements at each level of A are Normally distributed with mean $\alpha_p + \mu$ and variance

≤ and that all the populations at each of the levels 7have the same variance. In addition, assume that
≤ sum to zero. Analogous assumptions are made for B and C.
- If a factor, for instance, A, is random, assume the effect of the level of A itself, ≤, is a random variable Normally distributed with mean 0 and variance

$\sigma_A{}^2$. Analogous assumptions are made for B and C.
- If some of the factors, for instance, A and B, associated with the effect of an interaction are ≤ fixed, then assume that the populations of measurements at each level of A and B are Normally distributed with mean

$\mu + \alpha_p + \beta_q + (\alpha\beta)_{pq}$ and variance

$\sigma_{AB}{}^2$. For any fixed p, the means
≤ sum to zero when summing over all q. Similarly, for any fixed q,
≤ sum to zero when summing over all p.
- If any of the factors, for instance, A and B, associated with the effect of an interaction ≤ are random, assume the effect is a random variable Normally distributed with mean 0 and variance

≤. If A is fixed but B is random, assume that for any fixed q, the means
≤ sum to zero when summing over all p. Similarly, if B is fixed but A is random, assume that for any fixed p the means
≤ sum to zero when summing over all q.
- Assume all effects taken to be random variables are mutually independent.

The Hypotheses

Each of the following hypotheses is a different way of saying that a factor or an interaction among factors has no effect on experimental outcomes. This VI assumes that there are no effects and then seeks evidence to contradict these assumptions. The seven hypotheses are as follows.

- (A) that ≤ for all levels p if factor A is fixed, and that

$\sigma_A{}^2 = 0$ if factor A is random.
- (B) that ≤ for all levels q if factor B is fixed, and that

$\sigma_B{}^2 = 0$ if factor B is random.
- (C) that $\gamma_r = 0$ for all levels r if factor C is fixed, and that

$\sigma_C{}^2 = 0$ if factor B is random.

- (AB) that $\alpha\beta_{pq} = 0$ for all levels p and q if factors A and B are fixed, and that

$\sigma_{AB}^2 = 0$ if either factor A or B is random.
- (AC) that $\alpha\gamma_{pr} = 0$ for all levels p and q if factors A and C are fixed, and that

$\sigma_{AC}^2 = 0$ if either factor A or C is random.
- (BC) that $\beta\gamma_{qr} = 0$ for all levels p and q if factors B and C are fixed, and that

$\sigma_{BC}^2 = 0$ if either factor B or C is random.
- (ABC) that $\alpha\beta_{pqr} = 0$ for all levels p, q, and r if factors A, B, and C are fixed, and that

$\sigma_{ABC}^2 = 0$ if any of factors A, B, or C is random.

The Testing of Hypotheses

For each hypothesis, the VI computes number f that is used to calculate the associated sig probability.

For example, for the hypothesis (A), that a*p*= 0 for all the levels p, (fixed A), the VI computes $fa = \frac{msa}{mse}$, then

$sigA = Prob\left\{F_{a-1,abc(L-1)} > fa\right\}$ where

$F_{a-1,abc(L-1)} > fa$ is an F distribution with degrees of freedom a-1 and abc(L-1). You can then use the probabilities **sigA**, **sigB**, **sigC**, **sigAB**,..., **sig ABC** to determine when you should reject the associated hypotheses (A), (B), (C), (AB),..., (ABC).
How do you know when to reject the null hypothesis? For each hypothesis, you choose a level of significance. This level of significance is how likely you want it to be that you mistakenly reject the hypothesis (a common choice is 0.05). Compare your chosen level of significance with the associated sig probability output. If the sig probability is less than your chosen level of significance, you should reject the null hypothesis.

If, for instance, A is a random effect, your level of significance is 0.05, and **sigA** = 0.03, you must reject the hypothesis that $\leq$ and conclude that factor A has an effect on the experimental observations.

With some models there are no appropriate tests for certain hypotheses. If such is the case, the output parameters directly involved with the testing of these hypotheses are -1.0.

Formulas

Let $\leq$ be the

$\leq$ observation at the
$\leq$,
$\leq$, and
$\leq$ levels of A, B, and C respectively, where s = 0, 1,..., L-1.
Let

$a = |A\ levels|$

$b = |B\ levels|$

$c = |C\ levels|$
then

$$T_{pqr\bullet} = \sum_{s=0}^{c-1} T_{pqrs} \quad T_{pq\bullet\bullet}$$

$$T_{pq\bullet\bullet} = \sum_{r=0}^{c-1} X_{pqr\bullet}$$

$$T_{p\bullet r\bullet} = \sum_{q=0}^{b-1} X_{pqr\bullet}$$

$$T_{\bullet qr\bullet} = \sum_{p=0}^{a-1} X_{pqr\bullet}$$

$$T_{p\bullet\bullet\bullet} = \sum_{q=0}^{b-1} X_{pq\bullet\bullet}$$

$$T_{\bullet q\bullet\bullet} = \sum_{p=0}^{a-1} X_{pq\bullet\bullet}$$

$$T_{\bullet\bullet r\bullet} = \sum_{p=0}^{a-1} X_{p\bullet r\bullet}$$

$$T = \sum_{p=o}^{a-1} \sum_{q=0}^{b-1} \sum_{r=0}^{c-1} \sum_{s=0}^{L-1} X_{pqrs}^2$$

$$A = \sum_{p=0}^{a-1} \frac{T_{p\bullet\bullet\bullet}^2}{bcL}$$

$$B = \sum_{q=0}^{b-1} \frac{T_{\bullet q\bullet\bullet}^2}{acL}$$

$$C = \sum_{r=0}^{C-1} \frac{T_{\bullet\bullet r\bullet}^2}{abL}$$

$$AB = \sum_{p=0}^{a-1} \sum_{q=0}^{b-1} \frac{T_{pq\bullet\bullet}^2}{cL}$$

$$AC = \sum_{p=0}^{a-1} \sum_{r=0}^{c-1} \frac{T_{p\bullet r\bullet}^2}{bL}$$

$$BC = \sum_{q=0}^{b-1} \sum_{r=0}^{c-1} \frac{T_{\bullet qr\bullet}^2}{aL}$$

$$S = \sum_{p=0}^{a-1} \sum_{q=0}^{b-1} \sum_{r=0}^{c-1} \frac{T_{pqr\bullet}^2}{L}$$

$$CF = \frac{(\text{total sum of all observations})^2}{abcL}$$

$$ssa = A - CF \quad msa = \frac{ssa}{a-1}$$

$$ssb = B - CF \quad msb = \frac{ssb}{b-1}$$

$$ssc = C - CF \quad msc = \frac{ssc}{c-1}$$

$$ssab = AB - A - B + CF \quad msab = \frac{ssab}{(a-1)(b-1)}$$

$$ssac = AC - A - C + CF \quad msab = \frac{ssac}{(a-1)(c-1)}$$

$$ssbc = BC - A - C + CF \quad msbc = \frac{ssbc}{(b-1)(c-1)}$$

$$ssabc = S - AB - AC - BC + A + B + C - CF \quad mssbc = \frac{sssbc}{(a-1)(b-1)(c-1)}$$

General Histogram.vi

General Histogram

Histogram.vi

Histogram

Mean.vi

Mean

Median.vi

Median

Mode.vi

Mode

Moment About Mean.vi

Moment About Mean

MSE.vi

MSE

RMS.vi

RMS

Sample Variance.vi

Sample Variance

Standard Deviation.vi

Standard Deviation

Variance.vi

Variance

Probability Subpalette

Probabiliity

Chi Square Distribution.vi

Chi Square Distribution

Contingency Table.vi

Contingency Table

erf(x) .vi

erf(x)

erfc(x) .vi

erfc(x)

F Distribution.vi

F Distribution

Inv Chi Square Distribution.vi

Chi Square Distribution

Inv F Distribution.vi

Inv F Distribution

Inv Normal Distribution.vi

Inv Normal Distribution

Inv T Distribution.vi

Inv T Distribution

Normal Distribution.vi

Normal Distribution

T Distribution.vi

T Distribution

Analysis of Variance Subpalette

Analysis of Variance

1D ANOVA.vi

1D ANOVA

2D ANOVA.vi

2D ANOVA

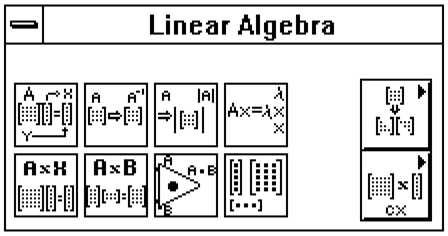
3D ANOVA.vi

3D ANOVA

Linear Algebra VIs

This topic describes the VIs that perform matrix related computation and analysis. It includes both real and complex matrices. For general information about Linear Algebra VIs, see Linear Algebra VIs Overview.

The following illustration shows the options that are available on the **Linear Algebra** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



A x B
A x Vector
Determinant
Dot Product
EigenValues & Vectors
Inverse Matrix
Outer Product
Solve Linear Equations

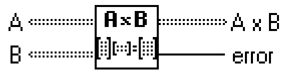
Subpalettes

Advanced Linear Algebra
Complex Linear Algebra
Advanced Complex Linear Algebra

Linear Algebra VI Descriptions

A x B

Performs the matrix multiplication of two input matrices.



≤ **A**. The number of columns in A must match the number of rows in **B** and must be greater than zero. If the number of columns in **A** does not match the number of rows in **B**, the VI sets **A x B** to an empty array and returns an error.
≤ **B** is the second matrix. If the number of rows in **B** does not match the number of columns in **A**, the VI sets **A x B** to an empty array and returns an error.
≤ **A x B** is the matrix containing the result of the matrix multiplication **A x B**.
≤ **error**. See Analysis Error Codes for a description of the error.
If **A** is an n-by-k matrix and **B** is a k-by-m matrix, the matrix multiplication of A and B, C = AB, results in a matrix, C, whose dimensions are n-by-m. Let A represent the 2D input array **A** matrix, B represent the 2D input array **B** matrix, and C represent the 2D output array **A x B**. The VI obtains the elements of C using the formula

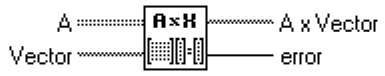
$$c_{ij} = \sum_{l=0}^{k-1} a_{il} b_{lj} \quad \text{for} \begin{cases} i = 0,1,2,\ldots,n-1 \\ j = 0,1,2,\ldots,m-1 \end{cases}$$

where n is the number of rows in **A** matrix, k is the number of columns in **A** matrix and the number of rows in **B** matrix, and m is the number of columns in **B** matrix.

Note: The A x B VI performs a strict matrix multiplication and not an element-by-element 2D multiplication. To perform an element-by-element multiplication, you must use the LabVIEW Multiply function. In general, ABBA.

A x Vector

Performs the multiplication of an input matrix and an input vector.



≤ **A.** The number of columns in **A** must match the number of elements in X and must be greater than zero. If the number of columns in **A** does not match the number of elements in X, the VI sets **A x Vector** to an empty array and returns an error.
≤ **Vector** is the input vector.
≤ **A x Vector** is the output vector containing the result of **A** multiplied by **Vector**.
≤ **error**. See Analysis Error Codes for a description of the error.

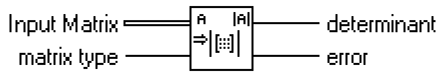
If **A** is an n-by-k matrix, and X is a vector with k elements, the multiplication of **A** and X, Y = AX, results in a vector Y with n elements. Let Y represent the output **A x Vector**. The VI obtains the elements of Y using the formula

$$y_i = \sum_{j=0}^{k-1} a_{ij} x_j \quad \text{for } i = 0,1,2,\ldots,n-1$$

where n is the number of rows in **A**, and k is the number of columns in **A** and the number of elements in X.

Determinant

Computes the **determinant** of a real, square matrix **Input Matrix**.



≤ **Input Matrix** must be a square, real matrix.
≤ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **determinant** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy. **matrix type** has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≤ **determinant**.
Special Case: The **determinant** of a singular matrix is zero. This is a valid result and is not an error. $|A| = 0.0$ if A is singular.

≤ **error**. See Analysis Error Codes for a description of the error.

Let A be a square matrix that represents the **Input Matrix**, and let L and U represent the lower and upper triangular matrices, respectively, of A such that

A = LU,

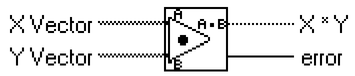
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U

$$|A| = \prod_{i=0}^{n-1} u_{ii} ,$$

where $|A|$ is the **determinant** of **X**, and n is the dimension of **X**.

Dot Product

Computes the dot product of **X Vector** and **Y Vector**.



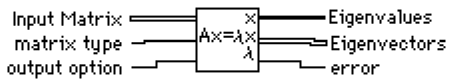
≤ **X Vector** is the first input vector. If the number of elements in **X Vector** is different from the number of elements in **Y Vector**, the VI computes the dot product based on the sequence that contains the fewest elements. If **X Vector** is an empty array, the dot product is NaN.
≤ **Y Vector** is the second input vector. If **Y Vector** is an empty array, the dot product is NaN.
≤ **X*Y** is the dot product.
≤ **error**. See Analysis Error Codes for a description of the error.
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product **X*Y** using the formula:

$$X * Y = \sum_{i=0}^{n-1} x_i y_i ,$$

where n is the number of data points. Notice that the output value **X*Y** is a scalar value.

EigenValues & Vectors (Advanced Only)

Finds the eigenvalues and eigenvectors right of a square, real **Input Matrix**.



≤ **Input Matrix** is an n-by-n square, real matrix, where n is the number of rows and columns of **Input Matrix**.
≤ **matrix type** is the type of **Input Matrix**. A symmetric matrix needs less computation than an unsymmetrical matrix. A symmetric matrix always has real eigenvectors and eigenvalues. **matrix type** has two possible options.

0:	general
1:	symmetric

matrix type defaults to general.

≤ **output option** determines what needs to be computed. The **output option** has two possible options.

0:	eigenvalues--computes eigenvalues
1:	both eigenvalues and eigenvectors--computes eigenvalues and eigenvectors

output option defaults to eigenvalues and eigenvectors.

≤ **Eigenvalues** is a complex vector of n elements, which contains all of the computed **Eigenvalues** of the **Input Matrix**. The **Input Matrix** could have complex **Eigenvalues** if it is not symmetric.

≤ **Eigenvectors** is a n-by-n complex matrix containing all of the computed **Eigenvectors** of the **Input Matrix**. The ith column of **Eigenvectors** is the eigenvector corresponding to the ith component of the vector, **Eigenvalues**. Each eigenvector is normalized so that its largest component is always unified. The **Input Matrix** could have complex **Eigenvectors** if it is not symmetric.

If **the output** option sets to **Eigenvalues**, **Eigenvectors** sets to empty.

≤ **error**. See Analysis Error Codes for a description of the error.

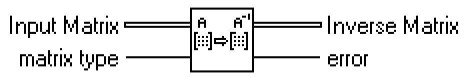
The eigenvalue problem is to determine the nontrivial solutions to the equation:

$$AX=\lambda X$$

where A is a n-by-n **Input Matrix**, X is a vector with n elements, and λ is a scalar. The n values of λ that satisfy the equation are the **Eigenvalues** of A and the corresponding values of X are the right **Eigenvectors** of A. A symmetric, real matrix always has real eigenvalues and eigenvectors.

Inverse Matrix

Finds the Inverse Matrix of the Input Matrix.



≤ **Input Matrix** must be a nonsingular, square matrix. If the **Input Matrix** is singular or is not square, the VI sets the **Inverse Matrix** to an empty array and returns an error.

≤ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **Inverse Matrix** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy.

matrix type has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≤ **Inverse Matrix** is the inverse matrix of the **Input Matrix**.

≤ **error**. See Analysis Error Codes for a description of the error.

Let A be the **Input Matrix** and I be the identity matrix. You obtain the **Inverse Matrix** value by solving the system AB = I for B.

If **A** is a nonsingular matrix, you can show that the solution to the preceding system is unique and that it corresponds to the **Inverse Matrix** of A:

$$B=A^{-1},$$

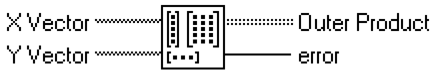
and **B** is therefore an **Inverse Matrix**. A nonsingular matrix is a matrix in which no row or column contains a linear combination of any other row or column, respectively.

Note: The numerical implementation of the matrix inversion is not only numerically intensive but, because of its recursive nature, is also highly sensitive to round-off errors introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.

You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.

Outer Product

Computes the outer product of **X Vector** and **Y Vector**.



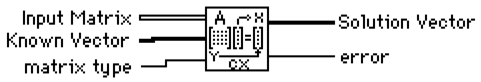
≤ **X Vector** is the first input vector.
≤ **Y Vector** is the second input vector.
≤ **Outer Product**. If one of the input sequences is an empty array, **Outer Product** is an empty array.
≤ **error**. See Analysis Error Codes for a description of the error.
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula:

$$a_{ij} = x_i y_i \qquad \text{for} \begin{cases} i = 0, 1, 2, \ldots, n-1 \\ j = 0, 1, 2, \ldots, m-1 \end{cases},$$

where A represents the 2D output sequence **Outer Product**, n is the number of elements in the input sequence **X Vector**, and m is the number of elements in the input sequence **Y Vector**.

Solve Linear Equations (Advanced Only)

Solves a real linear system AX=Y.



≤ **Input Matrix** is a square or rectangular, real matrix.
≤ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **Solution Vector** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy. **matrix type** has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≤ **Known Vector**. The number of elements in the **Known Vector** must be equal to the rows of the **Input Matrix**. If the number of elements in the **Known Vector** does not match the rows of the **Input Matrix**, the VI sets the **Solution Vector** to an empty array and returns an error.
≤ **Solution Vector** is the solution X to AX=Y.
≤ **error**. See Analysis Error Codes for a description of the error.
Let A be an m-by-n matrix that represents the **Input Matrix**, Y be the set of m coefficients in **Known Vector** and X be the set of n elements in **Solution Vector** that solves the system

AX = Y.

When $m>n$, the system has more equations than unknowns, so it is an overdetermined system. The solution that satisfies AX=Y may not exist, so the VI finds the least square solution X, which minimizes $\|AX - Y\|$.

When $m<n$, the system has more unknowns than equations, so it is an underdetermined systems. It may have infinite solutions that satisfy AX=Y. The VI finds one of these solutions.

In the case of $m=n$, if A is a nonsingular matrix--no row or column is a linear combination of any other row or column, respectively--then you can solve the system for X by decomposing the input matrix A into its lower and upper triangular matrices, L and U, such that

AX = LZ = Y,

and

Z = UX

can be an alternate representation of the original system. Notice that Z is also an n element vector.

Triangular systems are easy to solve using recursive techniques. Consequently, when you obtain the L and U matrices from A, you can find Z from the LZ = Y system and X from the UX = Z system.

In the case of $m \leq n$ can be decomposed to an orthogonal matrix Q and an upper triangular matrix R, so that A=QR. The linear system can then be represented by QRX=Y. You can then solve RX=QTY.

You can easily solve this triangular system to get x using recursive techniques.

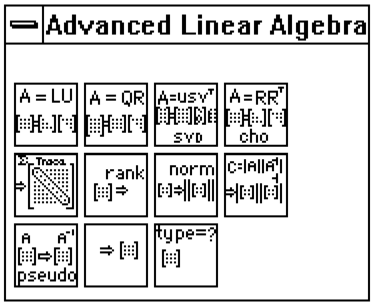
Note: You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.

The numerical implementation of the matrix inversion is numerically intensive and, because of its recursive nature, is also highly sensitive to round-off error introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve the system.

Advanced Linear Algebra

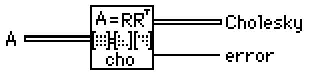
For general information about Advanced Linear Algebra VIs, see Linear Algebra VIs Overview.

The following illustration shows the options that are available on the **Advanced Linear Algebra** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Cholesky Factorization (Advanced Only)

Performs Cholesky factorization for a real, positive definite matrix **A**.



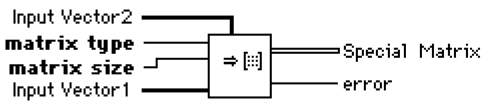
≤ **A** must be a positive definite matrix. If **A** is not positive definite, the VI returns an error code.
≤ **Cholesky** contains the factored, upper triangular matrix **R**.
≤ **error**. See Analysis Error Codes for a description of the error.

If the real, square matrix **A** is positive definite, you can factor it as $A=R^TR$, where R is an upper triangular matrix, and

R^T is the transpose of R.

Create Special Matrix (Advanced Only)

Generates a real, special matrix based on the **matrix type**.



Let n represent matrix size, X represent **Input Vector1**, nx represent the size of X, and Y represent **Input Vector2**, ny represent the size of Y, and B represent the output **Special Matrix**.

≤ **matrix type** specifies the type of special matrix that is generated output **Special Matrix**. **matrix type** has five possible options.

0: Identity Matrix--generate a n-by-n identity matrix.

1: Diagonal Matrix--generate a nx-by-nx diagonal matrix whose diagonal elements are the elements of X.

2: Toeplitz Matrix--generate a 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, the first element of X is used.

3: Vandermonde Matrix--generate a nx-by-nx Vandermonde matrix whose columns are powers of the elements of X. The elements of a Vandermonde matrix are:

$b_{i,j} = x_i^{nx-j-1}$, where i,j=0...nx-1.

4: Companion Matrix--generate a nx-1-by-nx-1 companion matrix. If vector X is a vector of a polynomial coefficient, the first element of X is the coefficient of the highest order, the last element of X is the constant term in the polynomial, the corresponding companion matrix is constructed as follows:

the first row is $b_{0,j-1} = -\frac{x_j}{x_0}$, j=1,2...nx-1 the rest of B from the second row is an identity matrix.

The eigenvalues of a companion matrix contain the roots of the corresponding polynomial.

≤ **matrix size** determines the dimension size of the output **Special Matrix** in some options.

≤ **Input Vector2** used as the input to construct a special matrix in some options.

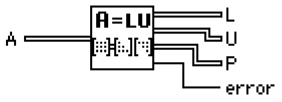
≤ **Input Vector1** used as the input to construct a special matrix in some options.

≤ **Special Matrix** is the generated matrix.

≤ **error**. See Analysis Error Codes for a description of the error.

LU Factorization (Advanced Only)

Performs the LU factorization of a real, square matrix **A**.



≤ **A** must be a square, real matrix.

≤ **L** is a lower triangular matrix.

≤ **U** is an upper triangular matrix.

≤ **P** is a permutation matrix.

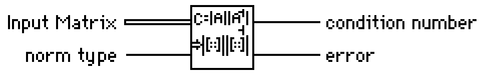
≤ **error**. See Analysis Error Codes for a description of the error.

LU factorization factors the square matrix **A** into two triangular matrices; one is a lower triangular matrix **L** with ones on the diagonal, and the other is an upper triangular matrix **U**, so that PA=LU, where **P** is a permutation matrix, which serves as the identity matrix with some rows exchanged.

Factorization serves as a key step for inverting a matrix, computing the **determinant** of a matrix, and solving a linear equation.

Matrix Condition Number (Advanced Only)

Computes the **condition number** of a real matrix **Input Matrix**.



≤ **Input Matrix** can be a rectangular matrix when **norm type** is 2-norm. If norm type is not 2-norm, **Input Matrix** must be a square matrix.

≤ **norm type** indicates what type of norm is used to compute the **condition number**. **norm type** has four possible options.

0: 2-norm

1: 1-norm
2: F-norm
3: inf-norm

norm type defaults to 2-norm. See the description for Matrix Norm for a definition of a matrix norm.

≤ **condition number** defines c as

$c = \|A\|_p \|A^{-1}\|_p$, where $\|A\|_p$ is the norm of **Input Matrix A**.

Different values of p define the different types of norms, therefore p defines different types of computations of condition numbers.

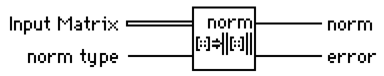
For the 2-norm condition number, c is the ratio of the largest, singular value of A to the smallest, singular value of A.

≤ **error**. See Analysis Error Codes for a description of the error.

The **condition number** of a matrix measures the sensitivity of a system solution of linear equations to errors in the data. It gives an indication of the accuracy of the results from a matrix inversion and a linear equation solution.

Matrix Norm (Advanced Only)

Computes the **norm** of a real matrix **Input Matrix**.



≤ **Input Matrix** can be a square or rectangular, real matrix.

≤ **norm type** indicates what type of norm is used to compute the norm. norm type has four possible options.

0: 2-norm--$\|A\|_2$ is the largest singular value of the **Input Matrix**.

1: 1-norm--$\|A\|_1$ is the largest column sum of the **Input Matrix**.

2: F-norm-- $\|A\|_f$ is equal to

$\sqrt{\sum \mathrm{diag}(A^T A)}$ where diag $A^T A$ means the diagonal elements of matrix $A^T A$, A^T is the transpose of A.

3: inf-norm--$\|A\|_\infty$ is the largest row sum of the **Input Matrix**.

≤ **norm**.

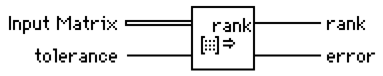
≤ **error**. See Analysis Error Codes for a description of the error.

The norm of a matrix is a scalar that gives some measure of the magnitude of the elements in the matrix. Let **A** represent the **Input Matrix**, the norm of A is represented by $\|A\|_p$, where p can be 1,2,F,

∞. Different values of p mean different types of norms that are computed.

Matrix Rank (Advanced Only)

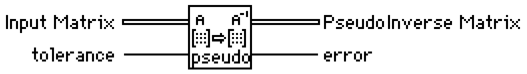
Computes the rank of a rectangular, real matrix **Input Matrix**.



≤ **Input Matrix** must be a real matrix.
≤ **tolerance** defaults to -1. All of the negative **tolerance** causes an internal tol=max (m,n)*
≤* eps to be used, where A represents the **Input Matrix**, m represents the number of rows in A, n represents the number of columns in A,
≤ is the 2-norm of A, eps is the smallest, floating point number that can be represented by type double, eps = 2^(-52)=2.22e-16.
≤ **rank**.
≤ **error**. See Analysis Error Codes for a description of the error.
Matrix rank is the number of singular values in the **Input Matrix** that are larger than the **tolerance**. rank is the maximum number of independent rows or columns in the **Input Matrix**.

PseudoInverse Matrix (Advanced Only)

Finds the Pseudo**Inverse Matrix** of a rectangular, real matrix **Input Matrix**.

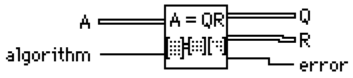


≤ **Input Matrix** is a rectangular, real matrix.
≤ **tolerance** defaults to -1. All of the negative **tolerance** causes an internal tol=max (m,n)*
≤* eps to be used, where A represents the **Input Matrix**, m represents the number of rows in A, n represents the number of columns in A, EMBED is the 2-norm of A, eps is the smallest, floating point number that can be represented by type double,
eps = 2^(-52)=2.22e-16.
≤ **PseudoInverse Matrix**.
≤ **error**. See Analysis Error Codes for a description of the error.
You compute Pseudo**Inverse Matrix** A^+ by using the SVD algorithm and any singular value less than the **tolerance**, which are set to zero. For a definition of the PseudoInverse of a matrix, see the Solving Linear Equations and Matrix Inverses section

If Input matrix A is square and not singular, ≤ is the same as
A^{-1}, but using the Inverse Matrix VI to compute
A^{-1} is more efficient than using this VI.

QR Factorization (Advanced Only)

Performs the QR factorization of a real matrix A.



≤ **A** is an m-by-n real matrix, where m is the number of rows in **A** and n is the number of columns in **A**. It can be either a square or rectangular matrix.
≤ **algorithm** has three possible options:
0: householder
1: givens
2: fast givens

algorithm defaults to the householder

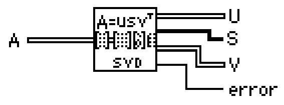
≤ **Q** is an m-by-m, orthogonal matrix.
≤ **R** is an m-by-n, upper triangular matrix.
≤ **error**. See Analysis Error Codes for a description of the error.
QR factorization is also called orthogonal-triangular factorization. It factors a real matrix **A** into two matrices. One is an orthogonal matrix **Q**, and the other is an upper triangular matrix **R**, so that A=QR. This

VI 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.

SVD Factorization (Advanced Only)

Performs the singular value decomposition (SVD) of a given m-by-n real matrix **A**, with $m>n$.



≤ **A** is an m-by-n matrix with $m>n$, where represents the number of rows in A**,** and represents the number of columns in **A**. If A has $m<n$, transpose **A** before you call this VI. Or, you can create rows of zeros underneath the nonzero rows in **A**, until **A** becomes square, and then call this VI.

≤ **U** is an m-by-n matrix, which contains n orthogonal columns.

≤ **S** is an array, which contains the number of n singular values of **A** in decreasing order.

≤ **V** is an n-by-n orthogonal matrix.

≤ **error**. See Analysis Error Codes for a description of the error.

SVD produces three matrices U, S_0, and V so that

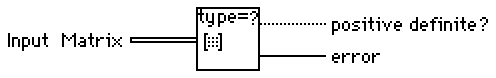
$A = US_0V^T$, where U and

V^T are orthogonal matrices,

$S_0$is an n-by-n diagonal matrix with the elements of array **S** on the diagonal in decreasing order.

Test Positive Definite (Advanced Only)

Tests whether the **Input Matrix** is a Positive Definite matrix.



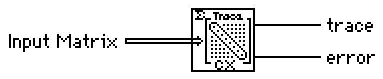
≤ **Input Matrix** is a square, real matrix.

TF **positive definite?** contains the test result. If the **Input Matrix** is Positive Definite, **positive definite?**=TRUE, otherwise, it equals FALSE.

≤ **error**. See Analysis Error Codes for a description of the error.

Trace (Advanced Only)

Finds the trace of Input Matrix.



≤ **Input Matrix** must have as many rows as columns, and its dimensions must be greater than zero. If **Input Matrix** is an empty array or is not square, the VI sets **trace** to NaN and returns an error.

≤ **trace**.

≤ **error**. See Analysis Error Codes for a description of the error.

Let A be a square matrix that represents **Input Matrix** and tr(A) be **trace**. The **trace** of A is the sum of the main diagonal elements of A

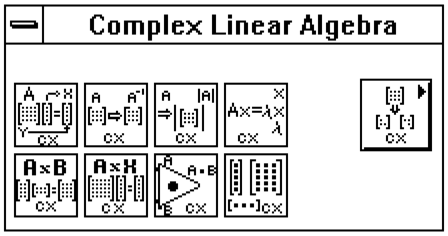
$$tr(A) = \sum_{i=0}^{n-1} a_{ii},$$

where n is the dimension of **Input Matrix**.

Complex Linear Algebra

For general information about Complex Linear Algebra VIs, see Linear Algebra VIs Overview.

The following illustration shows the options that are available on the **Complex Linear Algebra** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



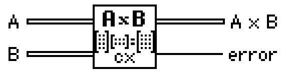
Complex A x B
Complex A x Vector
Complex Determinant
Complex Dot Product
Complex EigenValues & Vectors
Complex Inverse Matrix
Complex Outer Product
Solve Complex Linear Equations

Subpalette

Advanced Complex Linear Algebra

Complex A x B (Advanced Only)

Performs the matrix multiplication of two input complex matrices.



≤ **A**. The number of columns in **A** must match the number of rows in **B** and must be greater than zero. If the number of columns in **A** does not match the number of rows in **B**, the VI sets **A x B** to an empty array and returns an error.
≤ **B** is the second matrix. If the number of rows in **B** does not match the number of columns in **A**, the VI sets **A x B** to an empty array and returns an error.
≤ **A x B** is the matrix containing the result of the matrix multiplication **A x B**.
≤ **error**. See Analysis Error Codes for a description of the error.

If A is an n-by-k matrix and B is a k-by--m matrix, the matrix multiplication of A and B, C = AB, results in a matrix, C, whose dimensions are n-by-m. Let A represent the 2D input array **A** matrix, B represent the 2D input array **B** matrix, and C represent the 2D output array **A x B**. The VI obtains the elements of C using the formula

$$c_{ij} = \sum_{l=0}^{k-1} a_{il} b_{lj} \quad \text{for} \begin{cases} i = 0, 1, 2, \ldots, n-1 \\ j = 0, 1, 2, \ldots, m-1 \end{cases},$$

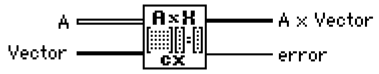
where n is the number of rows in **A** matrix, k is the number of columns in **A** matrix and the number of rows in **B** matrix, and m is the number of columns in **B** matrix.

Note: The Complex A x B VI performs a strict matrix multiplication and not an element-by-element 2D multiplication. To perform an element-by-element multiplication, you must use

the LabVIEW Multiply function. In general, ABBA.

Complex A x Vector (Advanced Only)

Performs the multiplication of a complex input matrix and a complex input vector.



≤ **A.** The number of columns in **A** must match the number of elements in vector and must be greater than zero. If the number of columns in **A** does not match the number of elements in vector, the VI sets **A x Vector** to an empty array and returns an error.
≤ **Vector** is the input vector.
≤ **A x Vector** is the output vector containing the result of **A** multiplied by **Vector**.
≤ **error**. See Analysis Error Codes for a description of the error.

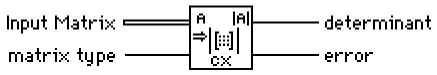
If **A** is an n-by-k matrix, and X is a vector with k elements, the multiplication of **A** and X, Y = AX, results in a vector Y with n elements. Let Y represent the output **A x Vector**, X represents the input vector. The VI obtains the elements of Y using the formula

$$y_i = \sum_{l=0}^{k-1} a_{ij} s_j \quad \text{for } i = 0, 1, 2, \ldots, n-1$$
,

where n is the number of rows in **A**, and k is the number of columns in **A** and the number of elements in X.

Complex Determinant (Advanced Only)

Finds the **determinant** of a complex, square matrix **Input Matrix**.



≤ **Input Matrix** must be a square matrix.
≤ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **determinant** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy. **matrix type** has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≤ **determinant**.
Special Case: The **determinant** of a singular matrix is zero. This is a valid result and is not an error. $|A| = 0.0$ if A is singular.

≤ **error**. See Analysis Error Codes for a description of the error.
Let A denote a square matrix that represents the **Input Matrix**, and let L and U be the lower and upper triangular matrices, respective, of A such that

A = LU,

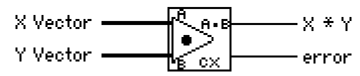
where the main diagonal elements of the lower triangular matrix L are arbitrarily set to one. The VI finds the **determinant** of A by the product of the main diagonal elements of the upper triangular matrix U:

$$|A| = \prod_{i=0}^{n-1} u_{ii} \,,$$

where $|A|$ is the **determinant** of A**,** and n is the dimension of A**.**

Complex Dot Product (Advanced Only)

Computes the dot product of complex **X Vector** and **Y Vector**.



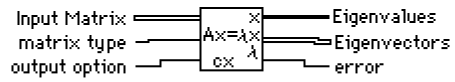
≤ **X Vector**. If the number of elements in **X Vector** is different from the number of elements in **Y Vector**, the VI computes the dot product based on the sequence that contains the fewest elements. If **X Vector** is an empty array, the dot product is NaN.
≤ **Y Vector**. If **Y Vector** is an empty array, the dot product is NaN.
CDB **X*Y** is the dot product.
≤ **error**. See Analysis Error Codes for a description of the error.
Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains the dot product **X*Y** using the formula:

$$X * Y = \sum_{i=0}^{n-1} x_i y_i \,,$$

where n is the number of data points. Notice that the output value **X*Y** is a complex scalar value.

Complex EigenValues & Vectors (Advanced Only)

Finds the **Eigenvalues** and right **Eigenvectors** of a square complex **Input Matrix A**.



≤ **Input Matrix** must be an n-by-n square matrix, where n is the number of rows or columns of **Input Matrix**.
≤ **matrix type** is the type of **Input Matrix**. A Hermitian matrix needs less computation than a general matrix. A Hermitian matrix always has real eigenvalues. **matrix type** has two possible options.

0: general
1: Hermitian matrix

matrix type defaults to general.

≤ **output option** determines what needs to be computed. The **output option** has two possible options.

0: eigenvalues--computes eigenvalues
1: eigenvalues and eigenvectors--computes eigenvalues and eigenvectors

output option defaults to eigenvalues and eigenvectors.

≤ **Eigenvalues** is a complex vector of n elements, which contains all of the computed eigenvalues of the **Input Matrix**. The **Input Matrix** could have complex **Eigenvalues** if it is not a Hermitian matrix.
≤ **Eigenvectors** is an n-by-n complex matrix containing all the computed eigenvectors of the **Input Matrix**. The
≤ column of **Eigenvectors** is the eigenvector corresponding to the ith component of the vector, **Eigenvalues**. Each eigenvector is normalized so that its largest component is always unity.
If you set the **output option** to **Eigenvalues**, the VI sets **Eigenvectors** to empty.

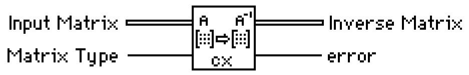
≲ **error**. See Analysis Error Codes for a description of the error.
The eigenvalue problem is to determine the nontrivial solutions for the equation:

AX=≲X

where A represents an n-by-n **Input Matrix**, X represents a vector with n elements, and is a scalar. The n values of ≲ that satisfy the equation are the **Eigenvalues** of A and the corresponding values of X are the right **Eigenvectors** of A. A Hermitian matrix always has real eigenvalues.

Complex Inverse Matrix (Advanced Only)

Finds the **Inverse Matrix** of a complex matrix **Input Matrix**.



≲ **Input Matrix** must be a nonsingular, square matrix. If the **Input Matrix** is singular or is not square, the VI sets the **Inverse Matrix** to an empty array and returns an error.
≲ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **Inverse Matrix** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy. **matrix type** has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≲ **Inverse Matrix** is the inverse matrix of the **Input Matrix**.
≲ **error**. See Analysis Error Codes for a description of the error.
Let A be the **Input Matrix** and I be the identity matrix. You obtain the **Inverse Matrix** by solving the system AB = I for B.

If A is a nonsingular matrix, you can show that the solution to the preceding system is unique and that it corresponds to the inverse matrix of A

B=≲

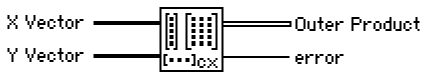
and B is therefore the **Inverse Matrix**. A nonsingular matrix is a matrix in which no row or column contains a linear combination of any other row or column, respectively.

Note: You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Complex Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.T

The numerical implementation of the matrix inversion is not only numerically intensive but, because of its recursive nature, it is also highly sensitive to round-off error introduced by the floating point, numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.

Complex Outer Product (Advanced Only)

Computes the outer product of a complex **X Vector** and **Y Vector**.



≤ **X Vector** is the first input vector.
≤ **Y Vector** is the second input vector.
≤ **Outer Product**. If one of the input sequences is an empty array, the **Outer Product** is an empty array.
≤ **error**. See Analysis Error Codes for a description of the error.

Let X represent the input sequence **X Vector** and Y represent the input sequence **Y Vector**. The VI obtains **Outer Product** using the formula:

$$a_{ij} = x_i y_j \qquad \text{for} \begin{cases} i = 0,1,2,\ldots,n-1 \\ j = 0,1,2,\ldots,m-1 \end{cases},$$

where A represents the 2D output sequence **Outer Product**, n is the number of elements in the input sequence **X Vector**, and m is the number of elements in the input sequence **Y Vector**.

Solve Complex Linear Equations (Advanced Only)

Solves a complex, linear system AX=Y.

≤
≤ **Input Matrix** must be a nonsingular, square or rectangular matrix.
≤ **Known Vector**. The number of elements in the **Known Vector**, must match the row size of the **Input Matrix**. If the number of elements in the **Known Vector** does not match the row size of matrix, the VI sets the **Known Vector** to an empty array and returns an error.
≤ **matrix type** is the type of **Input Matrix**. Knowing the type of **Input Matrix** can speed up the computation of the **Solution Vector** and can help you to avoid unnecessary computation, which could introduce numerical inaccuracy. **matrix type** has four possible options.

0: general
1: positive definite
2: lower triangular
3: upper triangular

matrix type defaults to general.

≤ **Solution Vector** is the solution X to AX = Y.
≤ **error**. See Analysis Error Codes for a description of the error.

Let A represent the m-by-n **Input Matrix**, Y represent the set of m elements in the **Known Vector**, and X represent the set of n elements in the **Solution Vector** that solves for the system

AX = Y.

When $m>n$, the system has more equations than unknowns, so it is an overdetermined system. Since the solution that satisfies AX=Y may not exist, the VI finds the least square solution X, which minimizes ≤.

When $m<n$, the system has more unknowns than equations, so it is an underdetermined system. It might have infinite solutions that satisfy AX=Y. The VI then selects one of these solutions.

When $m=n$, if A is a nonsingular matrix--no row or column is a linear combination of any other row or column, respectively--then you can solve the system for X by decomposing the **Input Matrix** A into its lower and upper triangular matrices, L and U, such that

AX = LZ = Y,

and

Z = UX

can be an alternate representation of the original system. Notice that Z is also an n element vector.

Triangular systems are easy to solve using recursive techniques. Consequently, when you obtain the L and U matrices from A, you can find Z from the LZ = Y system and X from the UX = Z system.

When $m \leq n$, A can be decomposed to an orthogonal matrix Q, and an upper triangular matrix R, so that A=QR, and the linear system can be represented by QRX=Y. You can then solve

$$RX = Q^H Y.$$

You can easily solve this triangular system to get X using recursive techniques.

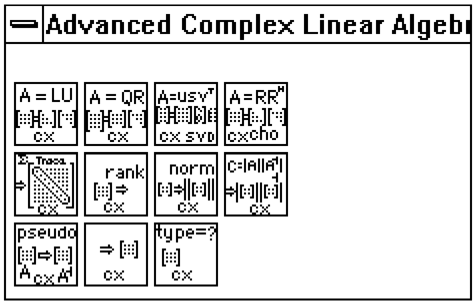
Note: You cannot always determine beforehand whether the matrix is singular, especially with large systems. The Inverse Matrix VI detects singular matrices and returns an error, so you do not need to verify whether you have a valid system before using this VI.

The numerical implementation of the matrix inversion is numerically intensive and, because of its recursive nature, is also highly sensitive to round-off error introduced by the floating-point numeric coprocessor. Although the computations use the maximum possible accuracy, the VI cannot always solve for the system.

Advanced Complex Linear Algebra

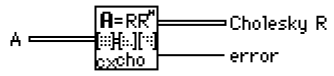
For general information about Advanced Complex Linear Algebra VIs, see Linear Algebra VIs Overview.

The following illustration shows the options that are available on the **Advanced Complex Linear Algebra** subpalette. Click on one of the icons below for function description information. You can also click on the text jumps below the icons to access function descriptions.



Complex Cholesky Factorization (Advanced Only)

Performs Cholesky factorization of a complex, positive definite matrix **A**.



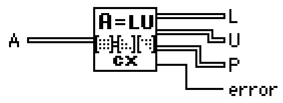
≤ **A** must be a positive definite, complex matrix. If **A** is not positive definite, the VI returns an error code.
≤ **Cholesky R** contains the factored upper triangular matrix **R**.
≤ **error**. See Analysis Error Codes for a description of the error.

If the complex square matrix **A** is positive definite, it can be factored as $A = R^H R$, where R is an upper triangular matrix and

R^H is the complex conjugate transpose of R.

Complex LU Factorization (Advanced Only)

Performs the LU factorization of a complex, square matrix **A**.



≤ **A** is a square, complex matrix.
≤ **L** is a complex, lower triangular matrix.

≤ **U** is a complex, upper triangular matrix.
≤ **P** is a permutation matrix.
≤ **error**. See Analysis Error Codes for a description of the error.
LU factorization factors the square matrix **A** into two triangular matrices; one is a lower triangular matrix **L** with ones on the diagonal, and the other is an upper triangular matrix **U**, so that

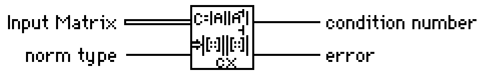
PA = LU

where **P** is a permutation matrix, which consists of the identity matrix with some rows exchanged.

Factorization is the key step for inverting a matrix, computing the **determinant** of a matrix, and solving a linear equation.

Complex Matrix Condition Number (Advanced Only)

Computes the **condition number** of a complex matrix **Input Matrix**.



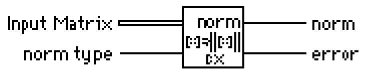
≤ **Input Matrix** can be a rectangular matrix when **norm type** is 2-norm. If norm type is not 2-norm, **Input Matrix** must be a square matrix.
≤ **norm type** indicates what type of norm is used to compute the **condition number**. **norm type** has four possible options.
0: 2-norm
1: 1-norm
2: F-norm
3: inf-norm

norm type defaults to 2-norm. See the description of the Complex Matrix Norm VI for a definition of a matrix norm.

≤ **condition number** defines c as
$c = |A|_p \|A^{-1}\|_p$, where
≤ is the norm of the **Input Matrix** A.
Different values of p define different types of norms as well as defining different types of computations of a **condition number**.
For the 2-norm condition number, c is the ratio of the largest, singular value of A to the smallest, singular value of A.
≤ **error**. See Analysis Error Codes for a description of the error.
The **condition number** of a matrix measures the sensitivity of the solution of a system of linear equations to errors in the data. It gives an indication of the accuracy of the results from the matrix inversion and linear equation solutions.

Complex Matrix Norm (Advanced Only)

Computes the **norm** of a complex matrix **Input Matrix**.



≤ **Input Matrix** can be a square or rectangular, complex matrix.
≤ **norm type** indicates what type of norm is used to compute the **norm**. **norm type** has four possible options.
0: 2-norm--≤ is the largest singular value of the **Input Matrix**.

1: 1-norm--≤ is the largest column sum of the **Input Matrix**.

2: F-norm--≤ is equal to

$\sqrt{\sum \mathrm{diag}\left(A^H A\right)}$, where diag $\left(A^H A\right)$ means the diagonal elements of matrix $A^H A$,

A^H is the transpose of A.

3: inf-norm--≤ is the largest row sum of the **Input Matrix**.

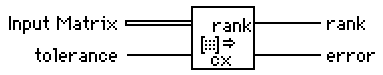
≤ **norm**.

≤ **error**. See Analysis Error Codes for a description of the error.

The **norm** of a matrix is a scalar that gives some measure of the magnitude of the elements of the matrix. Let **A** represent the **Input Matrix**, represent the norm of A, where p can be 1,2,f,. Different values of p mean different types of norms that are computed.

Complex Matrix Rank (Advanced Only)

Computes the rank of a rectangular, complex matrix **Input Matrix**.



≤ **Input Matrix** is a rectangular matrix.

≤ **tolerance** defaults to -1. All of the negative **tolerance** causes an internal tol=max (m,n)*

≤* eps to be used, where A represents the **Input Matrix**, m represents the number of rows in A, n represents the number of columns in A,

≤ is the 2-norm of A, eps is the smallest, floating point number that can be represented by type double, eps = 2^(-52)=2.22e-16.

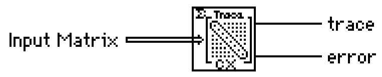
≤ **rank**.

≤ **error**. See Analysis Error Codes for a description of the error.

rank is the number of singular values of the **Input Matrix** that are larger than the **tolerance**. **rank** is the maximum number of independent rows or columns of the **Input Matrix**.

Complex Matrix Trace (Advanced Only)

Finds the **trace** of **Input Matrix**.



≤ **Input Matrix** must have as many rows as columns, and its dimensions must be greater than zero. If **Input Matrix** is an empty array or is not square, the VI sets **trace** to NaN and returns an error.

≤ **trace**.

≤ **error**. See Analysis Error Codes for a description of the error.

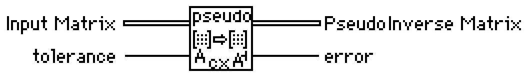
Let A be a square matrix that represents **Input Matrix** and tr(A) be **trace**. The **trace** of A is the sum of the main diagonal elements of A

$$\mathrm{tr}(A) = \sum_{i=0}^{n-1} a_{ii}$$

where n is the dimension of **Input Matrix**.

Complex PseudoInverse Matrix (Advanced Only)

Finds the **PseudoInverse Matrix** of a rectangular, complex matrix **Input Matrix**.



≤ **Input Matrix** is a rectangular matrix.

≤ **tolerance** defaults to -1. All of the **tolerance** causes an internal tol=max (m,n)*
≤* eps to be used, where A represents the **Input Matrix**, m represents the number of rows in A, n represents the number of columns in A,
≤ is the 2-norm of A, eps is the smallest, floating point number that can be represented by type double, eps = 2^(-52)=2.22e-16.

≤ **PseudoInverse Matrix**.

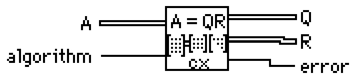
≤ **error**. See Analysis Error Codes for a description of the error.

An SVD algorithm computes **PseudoInverse Matrix** ≤, and treats any singular values less than the **tolerance** as zeros. For a definition of the PseudoInverse of a matrix, see the Solving Linear Equations and Matrix Inverses section at the beginning of this chapter.

If Input matrix A is square and not singular, ≤ is the same as
≤, but using the Complex Inverse Matrix VI to compute
≤ is more efficient than using this VI.

Complex QR Factorization (Advanced Only)

Performs QR factorization for a complex matrix A.



≤ **A** is an m-by-n complex matrix, where m is the number of rows in A and n is the number of columns in A. It can be either a square or rectangular matrix.

≤ **algorithm** has three possible options.

0: Householder
1: Givens
2: fast Givens

algorithm defaults to Householder.

≤ **Q** is an m-by-m, orthogonal matrix.

≤ **R** is an m-by-n, upper triangular matrix.

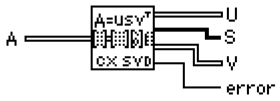
≤ **error**. See Analysis Error Codes for a description of the error.

QR factorization is also called orthogonal-triangular factorization. It factors a complex matrix **A** into two matrices; one is an orthogonal matrix **Q**, the other is an upper triangular matrix **R**, so that A = QR. This VI provides three methods for the factorization: Householder, Givens, and Fast Givens.

You can use QR factorization to solve linear systems that contain less or more equations than unknowns.

Complex SVD Factorization (Advanced Only)

Performs the singular value decomposition (SVD) of a given m-by-n, complex matrix A with m>n.



≤ **A** is a complex matrix of m-by-n with m>n, where m represents the number of rows in **A** and n represents the number of columns in **A**. If A has m<n, transpose **A** before you call this VI. Or, you can

create rows of zeros underneath the nonzero rows in **A**, until **A** becomes square, and then call this VI.

≤ **U** is an m-by-n matrix, which contains n orthogonal columns.

≤ **S** is an array, which contains the number of n singular values of **A** in decreasing order.

≤ **V** is an n-by-n orthogonal matrix.

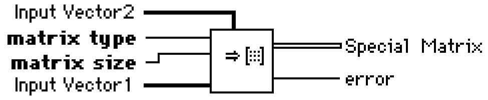
≤ **error**. See Analysis Error Codes for a description of the error.

SVD produces three matrices **U**, **S**, and **V**, so that $A = US_{o}V^{H}$, where U and V are orthogonal matrices,

≤ is an n-by-n diagonal matrix with the elements of array **S** on the diagonal in decreasing order. The diagonal elements are the singular values of A.

Create Special Complex Matrix (Advanced Only)

Generates a special, complex matrix based on the **matrix type**.



Let n represent **matrix size**, X represent **Input Vector1**, nx represent the size of X, and Y represent **Input Vector2**, ny represent the size of Y, and B represent the output **Special Matrix**.

≤ **matrix type** specifies the type of special matrix that is generated output **Special Matrix**. **matrix type** has five possible options.

0: Identity Matrix--generate a n-by-n identity matrix.
1: Diagonal Matrix--generate a nx-by-nx diagonal matrix whose diagonal elements are the elements of X.
2: Toeplitz Matrix--generate a 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, the first element of X is used.
3: Vandermonde Matrix--generate a nx-by-nx Vandermonde matrix whose columns are powers of the elements of X. The elements of a Vandermonde matrix are:

$b_{i,j} = x_i^{nx-j-1}$, where i,j=0…nx-1.

4: Companion Matrix--generate a nx-1-by-nx-1 companion matrix. If vector X is a vector of a polynomial coefficient, the first element of X is the coefficient of the highest order, the last element of X is the constant term in the polynomial, the corresponding companion matrix is constructed as follows:
the first row is, $b_{0,j-1} = \frac{x_j}{x_0}$, j=1,2…nx-1, the rest of B from the second row is an identity matrix.

The eigenvalues of a companion matrix contain the roots of the corresponding polynomial.

≤ **matrix size** determines the dimension size of the output **Special Matrix** in some options.

≤ **Input Vector2** used as the input to construct a special matrix in some options.

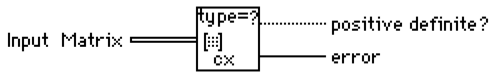
≤ **Input Vector1** used as the input to construct a special matrix in some options.

≤ **Special Matrix** is the generated matrix.

≤ **error**. See Analysis Error Codes for a description of the error.

Test Complex Positive Definite (Advanced Only)

Tests whether the **Input Matrix** is a Positive Definite matrix.



≤ **Input Matrix** is a complex, real matrix.

- **positive definite?** contains the test result. If the **Input Matrix** is Positive Definite, **positive definite?**=TRUE, otherwise, it equals FALSE.
- **error**. See Analysis Error Codes for a description of the error.

A x B.vi

A x B

A x Vector.vi

A x Vector

Determinant.vi

Determinant

Dot Product.vi

Dot Product

EigenValues & Vectors.vi

EigenValues & Vectors

Inverse Matrix.vi

Inverse Matrix

Outer Product.vi

Outer Product

Solve Linear Equations.vi

Solve Linear Equations

Advanced Linear Algebra Subpalette

Advanced Linear Algebra

Cholesky Factorization.vi

Cholesky Factorization

Create Special Matrix.vi

Create Special Matrix

LU Factorization.vi

LU Factorization

Matrix Condition Number.vi

Matrix Condition Number

Matrix Norm.vi

Matrix Norm

Matrix Rank.vi

Matrix Rank

PseudoInverse Matrix.vi

PseudoInverse Matrix

QR Factorization.vi

QR Factorization

SVD Factorization.vi

SVD Factorization

Test Positive Definite.vi

Test Positive Definite.vi

Trace.vi

Trace

Complex Linear Algebra Subpalette

Complex Linear Algebra

Complex A x B.vi

Complex A x B

Complex A x Vector.vi

Complex A x Vector

Complex Determinant.vi

Complex Determinant

Complex Dot Product.vi

Complex Dot Product

Complex EigenValues & Vectors.vi

Complex EigenValues & Vectors

Complex Inverse Matrix.vi

Complex Inverse Matrix

Complex Outer Product.vi

Complex Outer Product

Solve Complex Linear Equations.vi

Solve Complex Linear Equations

Advanced Complex Linear Algebra Subpalette

Advanced Complex Linear Algebra

Complex Cholesky Factorization.vi

Complex Cholesky Factorization

Complex LU Factorization.vi

Complex LU Factorization

Complex Matrix Condition Number.vi

Complex Matrix Condition Number

Complex Matrix Norm.vi

Complex Matrix Norm

Complex Matrix Rank.vi

Complex Matrix Rank

Complex Matrix Trace.vi

Complex Matrix Trace

Complex PseudoInverse Matrix.vi

Complex PseudoInverse Matrix

Complex QR Factorization.vi

Complex QR Factorization

Complex SVD Factorization.vi

Complex SVD Factorization

Create Special Complex Matrix.vi

Create Special Complex Matrix

Test Complex Positive Definite.vi

Test Complex Positive Definite

Linear Algebra VIs Overview

For detailed VI descriptions, see Linear Algebra VIs.

The VIs in this topic can be divided into the following groups:

Basic Matrix Manipulations
Common Matrices
Matrix Factorization
Solving Linear Equations and Matrix Inverses
Eigenvalues and EigenVectors
Matrix Analysis

A matrix is represented by a 2D array:

$$A = \begin{bmatrix} a_{00} & a_{00} & \cdots & a_{0n-01} \\ a_{10} & a_{11} & \cdots & a_{1n-1} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m-10} & a_{m-11} & \cdots & a_{m-1n-1} \end{bmatrix}$$

A is an m-by-n matrix that contains m rows and n columns.

A matrix is called a rectangular matrix in general. When $m=n$, it is called square matrix.

Basic Matrix Manipulations

Addition

$$C = A + B \Rightarrow c_{ij} = a_{ij} + b_{ij}$$

A, **B**, and **C** have the same dimension size.

Matrix-Matrix Multiplication

$$C = AB \Rightarrow c_{ij} = \sum_{k=0}^{r-1} a_{ik} b_{kj}$$

If **A** is a n-by-r matrix, **B** is a r-by-m matrix, then **C** is a n-by-m matrix.

Scalar-Matrix Multiplication

$$C = \alpha A \Rightarrow c_{ij} = \alpha a_{ij}$$

C and **A** have the same dimension size.

Transposition

for a real matrix:

$$C = A^T \Rightarrow c_{ij} = a_{ji}$$

for a complex matrix, it is the complex conjugate transposition:

$C = A^H \Rightarrow c_{ij=a^*{}_{ji}}$

complex conjugate: if $a = x + iy$, then conjugate $a^* = x - iy$. If **A** is a m-by-n matrix, then **C** is an n-by-m matrix and is called the transpose of **A**.

Common Matrices

Identity Matrix

$$A = \begin{bmatrix} 1 & 0 & \dots 0 \\ 0 & 1 & \dots 0 \\ 0 & 0 & \dots 1 \end{bmatrix}, a_{ij} = 0$$ when

$i \neq j, a_{ij} = 1 \text{ when } i = j$

A is a square matrix.

Diagonal Matrix

$$A = \begin{bmatrix} a_{\infty} & 0 & \dots & 0 \\ 0 & a_{11} & \dots & 0 \\ 0 & 0 & \dots & a_{m} - 1n - 1 \end{bmatrix}$$ when {bmc DIAM-1.BMP}.

Hermitian Matrix

If a complex matrix **A** satisfies$A = A^H$, **A** is called a Hermitian matrix.

Symmetric Matrix

Matrix **A** is called a symmetric matrix if $a_{ij} = a_{ji}$, that is

$A = A^T$.

Upper Triangular Matrix

$$A = \begin{bmatrix} a_{\infty} & a_{01} & \dots & a_{0n-1} \\ 0 & a_{11} & \dots & a_{1n-1} \\ 0 & 0 & \dots & a_{m-1n-1} \end{bmatrix}, a_{ij} = 0,$$ when i>j.

Lower Triangular Matrix

$$A = \begin{bmatrix} a_{\infty} & a_{01} & \dots & a_{0n-1} \\ 0 & a_{11} & \dots & a_{1n-1} \\ 0 & 0 & \dots & a_{m-1n-1} \end{bmatrix}, a_{ij} = 0,$$ when i<j.

Orthogonal Matrix

Matrix **A** is said to be orthogonal if $A^T A = 1$ **I** is an identity matrix.

Permutation Matrix

A permutation matrix is an identity matrix with some rows or columns exchanged. **A** permutation matrix is an orthogonal matrix.

Positive Definite Matrix

A real matrix is positive definite if and only if it is symmetric; that is, $\leq$, and the quadratic form

$X^T AX > 0$ for all nonzero vectors X.
A complex matrix is positive definite if and only if it is Hermitian; that is, $A = A^H$ and
$X^H AX > 0$ for all nonzero, complex vectors X.

Matrix Factorization

A Matrix can be factored into the multiplication of several, simpler matrices. You can use these factored, simple matrices to solve some matrix problems, such as solving a linear equation, inverting a matrix, and finding the **determinant** of a matrix.

The common factorization methods include LU, Cholesky, QR, and Singular Value Decomposition (SVD).

- LU Factorization--factors a square matrix into two matrices. One is an upper triangular matrix U, and the other is a lower triangular matrix L that has ones on the diagonal, so that PA=LU. P is a permutation matrix.
- When a square matrix is positive definite, you can factor it into $A = R^T R$, if A is a real matrix, and
$A = R^H R$, if A is a complex matrix, where R is an upper triangular matrix. This is called Cholesky factorization. Cholesky factorization only needs half of the operations of LU factorization.
- QR Factorization--factors a matrix as the product of an orthogonal matrix Q and an upper triangular matrix R: A=QR. QR factorization is useful for both square and rectangular matrices.
- SVD--decomposes a matrix into the product of three matrices:
$A = USV^T$, where U and V are orthogonal matrices and S is a diagonal matrix whose diagonal values are called the singular values of A. SVD is useful for solving analysis problems involving matrices. In addition to its common uses, you can use SVD for operations such as pseudoinverse, rank, norm, and condition number.

Solving Linear Equations and Matrix Inverses

To Solve the linear equation AX=Y, you must find solution X when you know the given values of A and Y. A is a m-by-n matrix, X is a vector with n elements, and Y is a vector with m elements.

Using LU factorization, if $m=n$ and **A** is a square matrix, **A** can be factored into triangular matrices **L** and **U**, so that A=LU. AX=Y becomes LUX=Y and you can solve Z for LZ=Y where Z=UX. You can then solve for X in UX=Z.

In the Cholesky case, $L = R^T$ and $U = R$

Triangular systems are easy to solve using recursive techniques.
If $m \leq n$, the number of equations are different from the number of unknowns and **A** is not a square matrix, **A** can be factored into an orthogonal matrix **Q** and an upper triangular matrix **R**, so that A=QR. AX=Y becomes QRX=Y and you can solve for X by using
$RX = Q^T Y$.
When $m>n$, and the system has more equations than unknowns, it is called an overdetermined system. The solution that satisfies AX=Y may not exist. The solution above finds the least square solution that minimizes $\|AX - Y\| = \sum\left[(AX)_i - y_i\right]^2$.

When m<n, and the system has more unknowns than equations, it is called an underdetermined system. It may have infinite solutions that satisfy AX=Y. The solution above finds one of these solutions.
Inverting a square matrix **A** means that you find $\leq$ that satisfies
$AA^{-1} = I$, where I is an identity matrix.
$\leq$ is called the inverse of matrix **A**. You can solve for
$\leq$ by solving n linear equations
$AA^{-1} = I$.
When **A** is not a square matrix, or when **A** is singular, $\leq$does not exist. You can compute the pseudoinverse of A instead. If the m-by-n matrix
$\leq$ satisfies the following four Moore-Penrose conditions:
$AA^+A = A$
$A^+AA^+ = A^+$
AA^+ is a Hermitian matrix if A is a complex matrix.
AA^+ is a symmetric matrix if A is real matrix.

$\leq$ is a Hermitian matrix if A is a complex matrix.
$\leq$ is a symmetric matrix if A is real matrix.
Then,
$\leq$ is called the pseudoinverse of matrix **A**. You can compute for
$\leq$ using SVD.

Eigenvalues and EigenVectors

This eigenvalue problem is to determine the nontrivial solutions to the equation $AX = \lambda X$, where A is an n-by-n matrix, X is a vector with elements, and

$\leq$ is a scalar. The n values of
$\leq$ that satisfy the equation are called eigenvalues of A, and the corresponding values of X are called the right eigenvectors of A.

Matrix Analysis

Matrix Analysis VIs can compute the matrix **determinant**, condition number, norm, and rank. Typically, you use these parameters to analyze a matrix property.