



IDL Tutorial

Advanced Signal Processing

**Copyright © 2008 ITT Visual Information Solutions
All Rights Reserved
<http://www.ittvis.com/>**

IDL[®] is a registered trademark of ITT Visual Information Solutions for the computer software described herein and its associated documentation. All other product names and/or logos are trademarks of their respective owners.



The IDL Intelligent Tools (iTools)

The IDL Intelligent Tools (**iTools**) are a set of interactive utilities that combine data analysis and visualization with the ability to produce presentation quality graphics. The iTools allow users to continue to benefit from the control of a programming language, while enjoying the convenience of a point-and-click environment. There are 7 primary iTool utilities built into the IDL software package. Each of these seven tools is designed around a specific data or visualization type :

- Two and three dimensional plots (line, scatter, polar, and histogram style)
- Surface representations
- Contours
- Image displays
- Mapping
- Two dimensional vector flow fields
- Volume visualizations

The iTools system is built upon an **object-oriented** component framework architecture that is actually comprised of only a single tool, which adapts to handle the data that the user passes to it. The pre-built *iPlot*, *iSurface*, *iContour*, *iMap*, *iImage*, *iVector* and *iVolume* procedures are simply shortcut configurations that facilitate ad hoc data analysis and visualization. Each pre-built tool encapsulates the functionality (data operations, display manipulations, visualization types, etc.) required to handle its specific data type. However, users are not constrained to work with a single data or visualization type within any given tool. Instead, using the iTools system a user can combine multiple dataset visualization types into a single tool creating a hybrid that can provide complex, composite visualizations.

What is Signal Processing?

A **signal** is basically the record of a process that occurs in relation to an independent variable. This independent variable can be any of a number of things, but in most cases it is **time**, in which case the signal is actually called a "**time-series**". Consequently, this exercise will only work with time-series signals. In addition, although IDL's digital signal processing tools can work in more than one dimension, this exercise will only work with one-dimensional signals (i.e. vectors).

A **digital signal** is essentially a sequence of real values observed at discrete points in time that is stored as numbers on your computer. The term "digital" actually describes two different properties of the signal :

- The values of the signal are only measured at discrete points in time as a result of **sampling**. In general most signals have a constant **sampling interval**.
- The signal can take only discrete values as defined by the dynamic range of the instrument and the precision at which the data is stored on the computer.

Quite often it is difficult or even impossible to make sense of the information contained in a digital signal by looking at it in its raw form. In addition, any signal obtained from an instrument measuring a physical process will almost always contain noise. **Signal processing** is a technique that involves using computer algorithms to analyze and transform the signal in an effort to create natural, meaningful, and alternate representations of the useful information contained in the signal while suppressing the effects of noise. In most cases signal processing is a multi-step process that involves both numerical and graphical methods.

Curve Fitting

The problem of curve fitting can be summarized as follows :

Given a tabulated set of data values $\{x_i, y_i\}$ and the general form of a mathematical model (i.e. a function $f(x)$ with unspecified parameters), determine the parameters of the model that minimize an error criterion.

In the following exercise, the signal data from the example data file "curve.csv" will be input into the current IDL session using the Import ASCII macro. This example data file is located in the "data" subfolder.

The file "curve.csv" is standard ASCII text containing 2 columns of data that represent 2 variables ("x" and "y"). This ASCII text file uses comma-separated values as the file format, which can be easily viewed in spreadsheet form within Microsoft Excel.

1. IDL> import_ascii
2. Within the file selection dialog choose the "curve.csv" file and hit "Open".

This will launch the standard ASCII Template wizard which was used in previous exercises to load data into the iTools system.

3. Within Step 1 of 3, make sure to change the "Data Starts at Line:" field to 2 and hit "Next >>".
4. In Step 2 of 3 all settings can be left as their defaults, so simply hit "Next >>".
5. In Step 3 of 3, change the "Name:" of *FIELD1* and *FIELD2* to X and Y, respectively.
6. Once this is accomplished, press "Finish".

Once the ASCII Template wizard has finished running the user will be returned to the main IDL development environment window where a new structure variable named "curve_ascii" now exists. This variable stores the data that was input from the file, and the *HELP* procedure can be used to acquire information on this variable :

```
7. IDL> HELP, curve_ascii, /ST
** Structure <14a2b70>, 2 tags, length=400, data length=400, refs=1:
   X                FLOAT      Array[50]
   Y                FLOAT      Array[50]
```

The *POLY_FIT* function within IDL performs a least-squares polynomial fit with optional error estimates and returns a vector of coefficients. The *POLY_FIT* function uses matrix inversion. A newer version of this routine, *SVDFIT*, uses Singular Value Decomposition (SVD), which is more flexible but slower. Another version of this routine, *POLYFITW*, performs weighted least-squares fitting. In addition, the *CURVEFIT* function uses a gradient-expansion algorithm to compute a non-linear least squares fit to a user supplied function with an arbitrary number of parameters. The user supplied function may be any non-linear function where the partial derivatives are known or can be approximated.

Using the tabulated polynomial data now stored in the "curve_ascii" variable, use the *POLY_FIT* fitting algorithm to fit a 3rd degree polynomial to the points :

8. IDL> coeff = POLY_FIT (curve_ascii.x, curve_ascii.y, 3, yFit)
9. IDL> iPlot, curve_ascii.x, curve_ascii.y, LINESTYLE=6, SYM_INDEX=2
10. IDL> iPlot, curve_ascii.x, yFit, /OVERPLOT

The resulting *IDL iPlot* visualization window should look similar to Fig. 1.

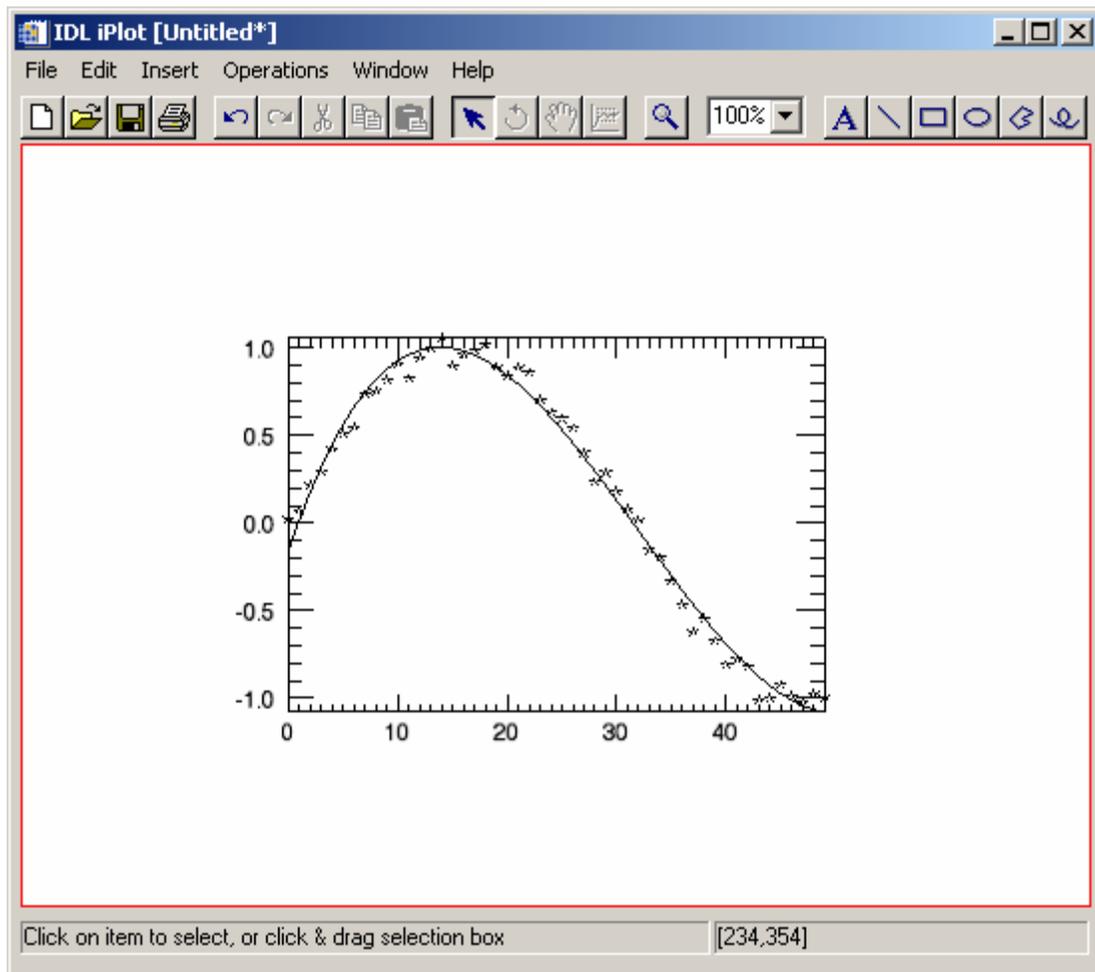


Figure 1: Simulated instrument response data (* symbols) and the fitted function (solid line)

Notice that POLY_FIT returns a vector containing the 4 estimated coefficients for the 3rd degree polynomial :

```
11. IDL> PRINT, coeff
      -0.154124
      0.183942
     -0.00846933
     8.83802e-005
```

12. Once finished viewing the fitted curve, close the *IDL iPlot* window.

Simple Noise Removal

A general discussion of time-series analysis assumes that a time-series is comprised of four components :

- A trend or long-term movement (a constant or uniformly varying background level)
- A cyclical fluctuation about the trend (a superposition of sinusoidal variations, each with a definite amplitude, frequency, and phase)
- A pronounced seasonal effect (sinusoidal variation with a large frequency value)
- A residual, irregular, or random effect (additive noise)

It is the identification and extraction of these components that can be of interest when performing analysis of signals. In addition, adjacent observations are likely to be correlated, especially as the time intervals between them become shorter. The analysis of the correlation between signals can also be of interest and will be covered later within this exercise.

Start by creating a test signal that can be used for analysis. An IDL function named "*test_signal.pro*" has been supplied within the "*lib*" subfolder that can be used to create a test signal.

```
1. IDL> void = test_signal (100, X=x, S=s)
```

Note: The *TEST_SIGNAL* routine is a custom program that is included with the distribution materials for this tutorial. If executing this statement results in the error "% Error opening file.", then the "*lib*" subfolder included with this tutorial that contains this program file is not included in IDL's path. To solve this problem simply open the program file into IDL and select "*Run > Compile*" from the main menu.

Once this batch file has executed there will be 2 new variables called "*x*" (the independent variable time) and "*s*" (the actual signal data) within the IDL session :

```
2. IDL> HELP, x, s
      X          FLOAT      = Array[100]
      S          FLOAT      = Array[100]
```

Both of these variables are vectors of 32-bit floating-point precision data that are 100 elements in length. To visualize this time-series signal simply make a call to the *iPlot* procedure and maximize the display window :

3. IDL> `iPlot, x, s, VIEW_GRID=[2,2], VIEW_TITLE='Original Signal'`
4. Select "Window > Zoom on Resize" from the IDL iPlot window menu system.
5. Press the  button in the upper-right hand corner to maximize the window.

The resulting *IDL iPlot* window will contain 4 view sub-windows. The line plot for the independent variable "x" as the X component and the signal variable "s" as the Y component will appear in the upper-left hand corner view window. The resulting visualization should look similar to Fig. 2.

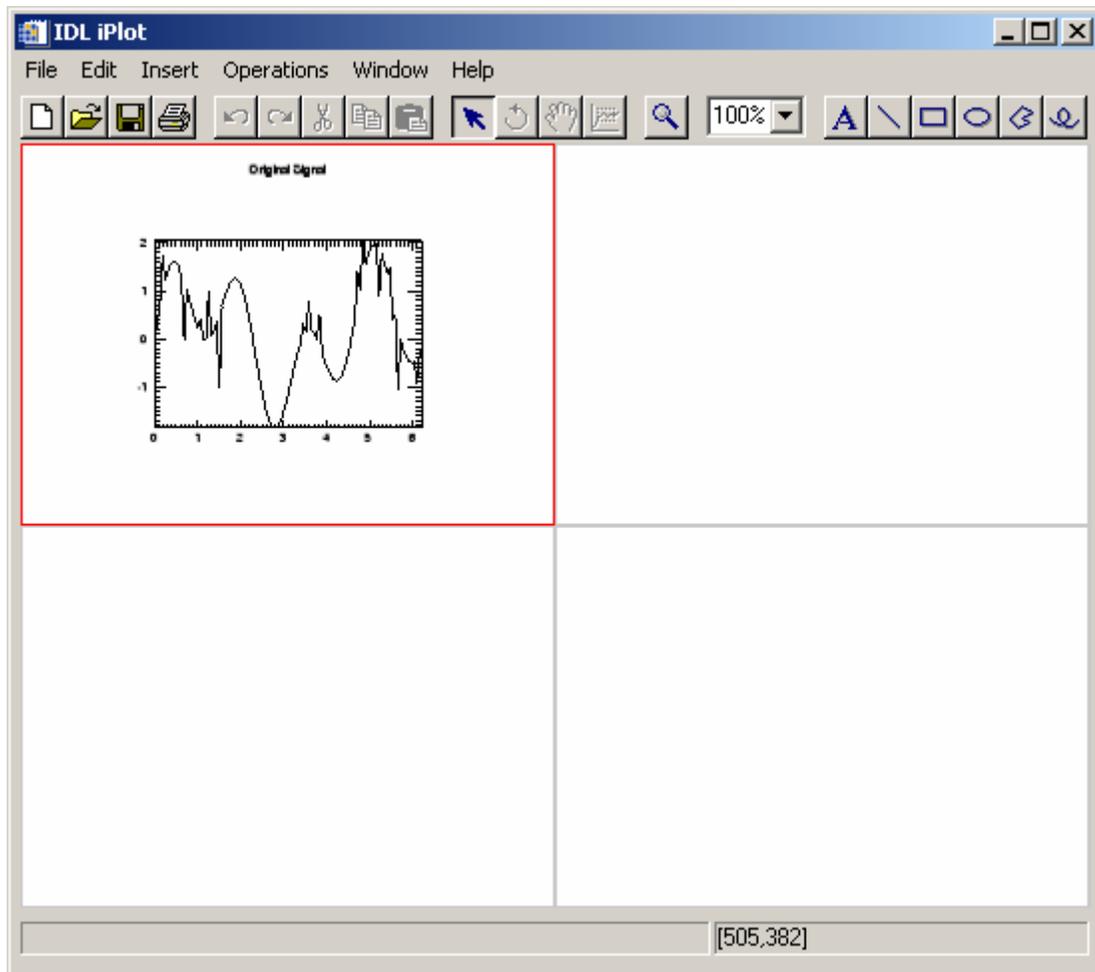


Figure 2: Simple line plot of test signal data

This signal has very sharp components that are analogous to noise. One way to attenuate certain types of noise in a signal is by using the *SMOOTH* function within IDL. The *SMOOTH* function filters an input array using a running-mean top hat kernel of a user-specified width (boxcar average). The *SMOOTH* function moves the kernel over the input array, element-by-element. At each step, the mean of the elements of the input array that fall under the window of the kernel is calculated, and this value is placed at the location of the centroid of the kernel at its current array element location. For example, consider a simple vector with five elements :

3.00	1.00	7.00	2.00	4.00
------	------	------	------	------

When applying a *SMOOTH* function the output will be the average across the user-specified kernel size. For instance, when using a kernel size of 3 the element in the exact center of this vector would be replaced with the value $(1.00 + 7.00 + 2.00) / 3 = 3.33$. This is the output that results from smoothing the above vector using a kernel with a size of 3 :

3.00	3.66	3.33	4.33	4.00
------	------	------	------	------

Note: By default, the endpoints outside the kernel window are replicated and placed in the output array unchanged. To override this effect use the *EDGE_TRUNCATE* keyword to the *SMOOTH* function.

Apply the *SMOOTH* function to the test signal and visualize the results :

```
6. IDL> smoothed = SMOOTH (s, 3)
7. IDL> iPlot, x, smoothed, /VIEW_NEXT, VIEW_TITLE='Smoothed'
```

The smoothed signal will be displayed in the upper-right hand corner view window. Smoothing a signal in this manner tends to distort the phase of the signal. Another interesting way to visualize the effect is by viewing the residual signal that is left after you remove the mean signal generated by the *SMOOTH* function from the original signal :

```
8. IDL> residual = s - smoothed
9. IDL> iPlot, x, residual, /VIEW_NEXT, VIEW_TITLE='Residual Signal'
```

The residual signal will contain information about the noise that the smoothing process removed.

Another way to remove certain types of unwanted noise from a signal is by using the *MEDIAN* function. The filtering process used by *MEDIAN* is similar to *SMOOTH*, but the application of the kernel is quite different. In *MEDIAN*, the median value (instead of the mean value) of the elements under the kernel is placed in the output array at the current centroid location. For example, if a *MEDIAN* function is applied to the same example vector :

3.00	1.00	7.00	2.00	4.00
------	------	------	------	------

With a kernel of size 3 the *MEDIAN* function will return :

3.00	3.00	2.00	4.00	4.00
------	------	------	------	------

Apply the *MEDIAN* function to the test signal and visualize the results :

```
10. IDL> median_filt = MEDIAN (s, 3)
11. IDL> iPlot, x, median_filt, /VIEW_NEXT, VIEW_TITLE='Median Filter'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 3.

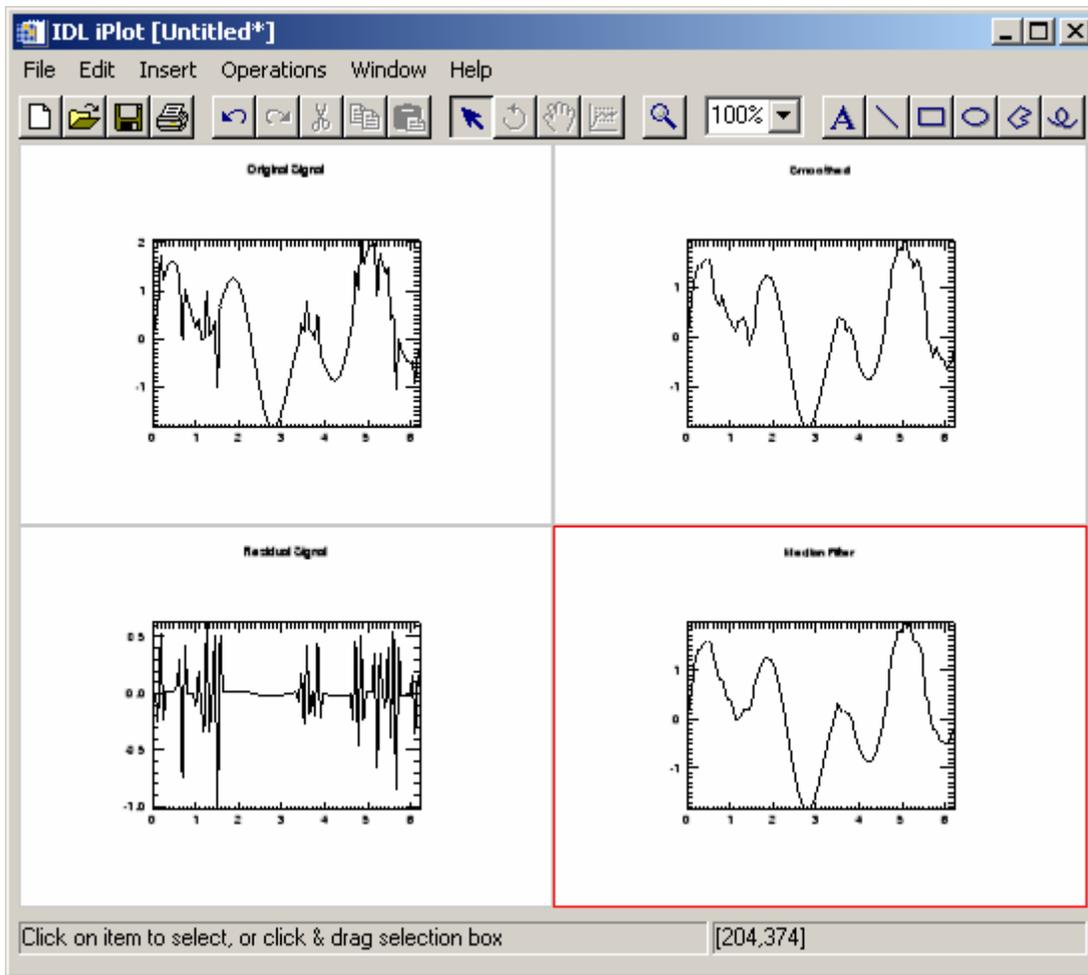


Figure 3: Result of applying SMOOTH and MEDIAN functions to test signal

12. Once finished viewing the modified signal line plots, close the *IDL iPlot* window.

In addition, the *CONVOL* function in IDL also provides the ability to convolve a signal with a user-defined kernel array. Convolution is a general process that can be used for various types of smoothing, shifting, differentiation, edge detection, etc..

Digital filters can also be used to remove unwanted frequency components (e.g. noise) from a sampled signal. Two broad classes of filters are Finite Impulse Response (a.k.a. Moving Average) filters and Infinite Impulse Response (a.k.a. Autoregressive Moving Average) filters.

Digital filters that have an impulse response that reaches zero in a finite number of steps are called Finite Impulse Response (FIR) filters. An FIR filter is implemented by convolving its impulse response with the data sequence it is filtering. IDL's *DIGITAL_FILTER* function computes the impulse response of an FIR filter based on Kaiser's window, which in turn is based on the Bessel function. *DIGITAL_FILTER* constructs lowpass, highpass, bandpass, or bandstop filters.



Obtain the coefficients of a non-recursive lowpass filter for the equally spaced data points and convolve it with the original signal :

```
13.IDL> coeffs = DIGITAL_FILTER (0.0, 0.25, 50, 12)
14.IDL> digital_filt = CONVOL (s, coeffs, /EDGE_WRAP)
15.IDL> iPlot, x, s, VIEW_GR=[3,1], VIEW_TI='Original Signal'
16.IDL> iPlot, x, digital_filt, /VIEW_NE, VIEW_TI='Digital Filter'
```

Another filtering technique, known as the Savitzky-Golay smoothing filter (a.k.a. least-squares or “DISPO”), also provides a digital smoothing that can be used to process a noisy signal. IDL’s *SAVGOL* function returns the coefficients of a Savitzky-Golay smoothing filter, which can then be applied using the *CONVOL* function. The filter is defined as a weighted moving average with weighting given as a polynomial of a certain degree. The returned coefficients, when applied to a signal, perform a polynomial least-squares fit within the filter window.

Use the *SAVGOL* function to obtain the coefficients of a Savitzky-Golay smoothing filter and visualize the results of both the *DIGITAL_FILTER* and *SAVGOL* functions :

```
17.IDL> coeffs = SAVGOL (5, 5, 0, 2)
18.IDL> savgol = CONVOL (s, coeffs, /EDGE_WRAP)
19.IDL> iPlot, x, savgol, /VIEW_NE, VIEW_TI='Savitzky-Golay'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 4.

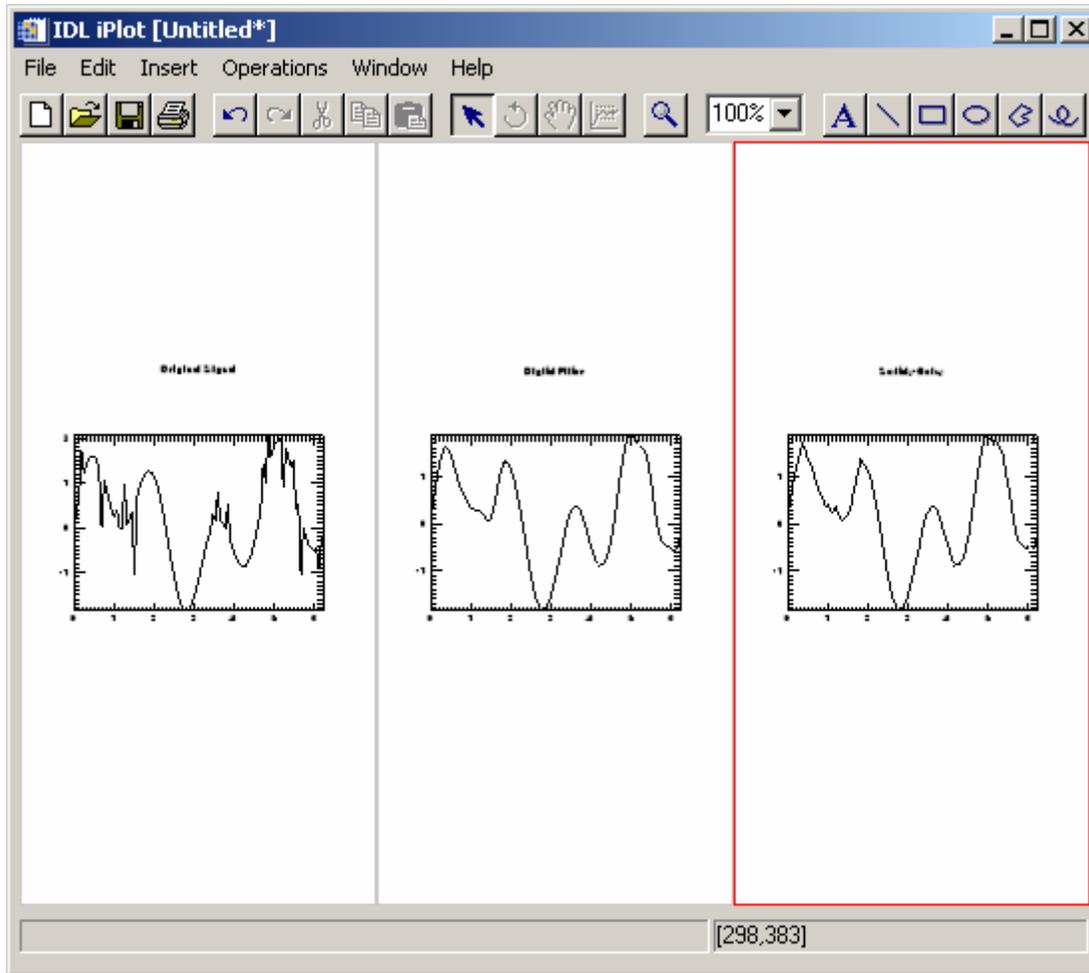


Figure 4: Result of applying `DIGITAL_FILTER` and `SAVGOL` functions to test signal

20. Once finished viewing the modified signal line plots, close the *IDL iPlot* window.

Correlation Analysis

The first step in the analysis of a time-series is the transformation to a stationary series. A stationary series exhibits statistical properties that are unchanged as the period of observation is moved forward or backward in time. Specifically, the mean and variance of a stationary series remain fixed in time. The sample autocorrelation function is commonly used to determine the stationarity of a time-series. The autocorrelation of a time-series measures the dependence between observations as a function of their time differences or **lag**. In other words, the autocorrelation of a function shows how a function is related to itself as a function of a lag value. A plot of the sample autocorrelation coefficients versus corresponding lags can be very helpful in determining the stationarity of a time-series.

Compute the autocorrelation of the synthesized signal data versus a set of user-defined lag values and display the results :

1. IDL> n = 100
2. IDL> lag = INDGEN (n/2) * 2
3. IDL> acorr = A_CORRELATE (s, lag)
4. IDL> iPlot, lag, acorr, VIEW_TI='Auto-Correlation'
5. IDL> iPlot, [0,n], [0,0], LINESYLE=1, YRANGE=[-1,1], /OVER

The resulting *IDL iPlot* visualization window should look similar to Fig. 5.

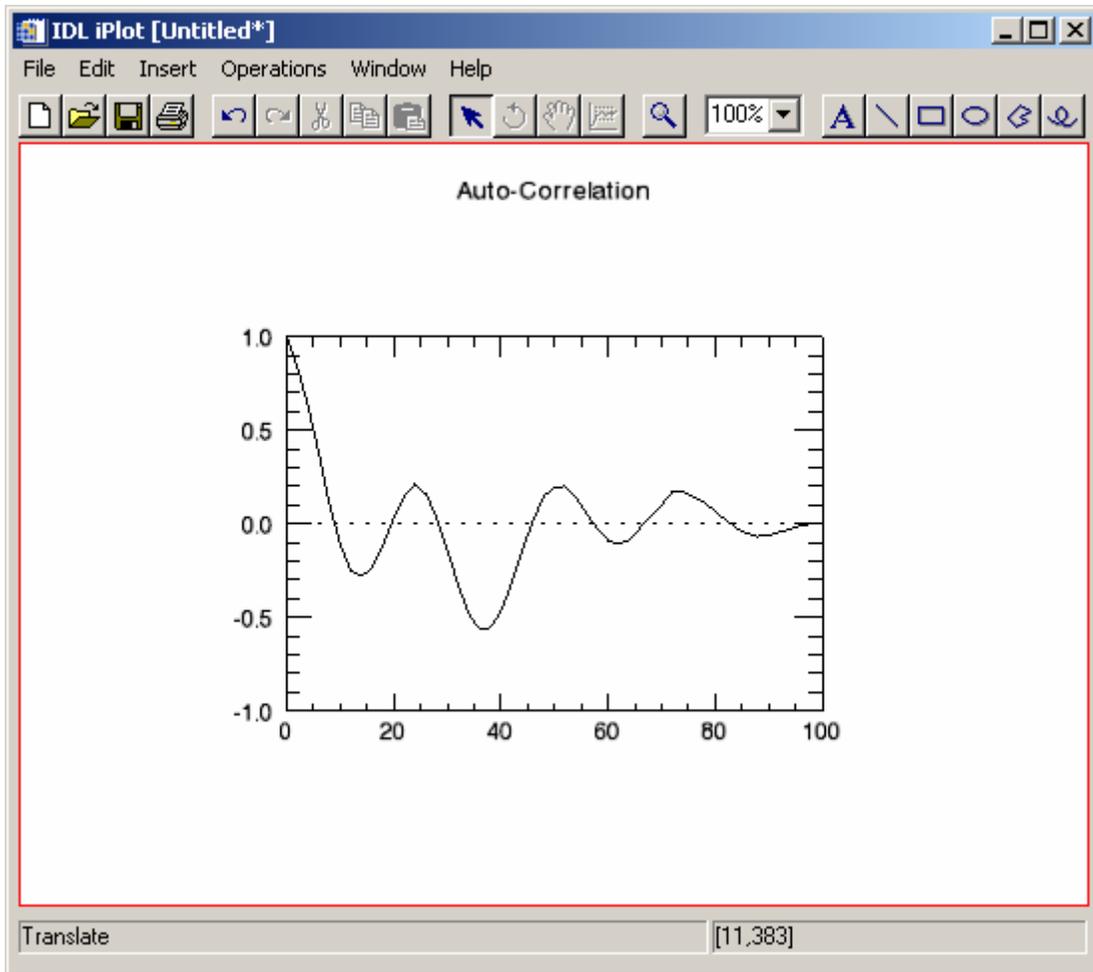


Figure 5: Autocorrelation function for test signal

The autocorrelation measures the persistence of a wave within the whole duration of a time-series. When the autocorrelation goes to zero, the process is becoming random, implying there are no regularly occurring structures. An autocorrelation of 1 means the signal is perfectly correlated with itself. An autocorrelation of -1 means the signal is perfectly anticorrelated with itself.

6. Once finished viewing the autocorrelation results, close the *IDL iPlot* window.

The `C_CORRELATE` function in IDL can be used to compute the cross-correlation of two sample populations as a function of the lag. To demonstrate the use of this function, construct another signal that is simply the reverse of the original :

```

7. IDL> lag = INDGEN (n) - n/2
8. IDL> t = REVERSE (s)
9. IDL> ccorr = C_CORRELATE (s, t, lag)
10.IDL> iPlot, lag, ccorr, VIEW_TI='Cross-Correlation'
11.IDL> iPlot, [-n/2,n/2], [0,0], LINESYLE=1, YRANGE=[-1,1], /OVER

```

The resulting *IDL iPlot* visualization window should look similar to Fig. 6.

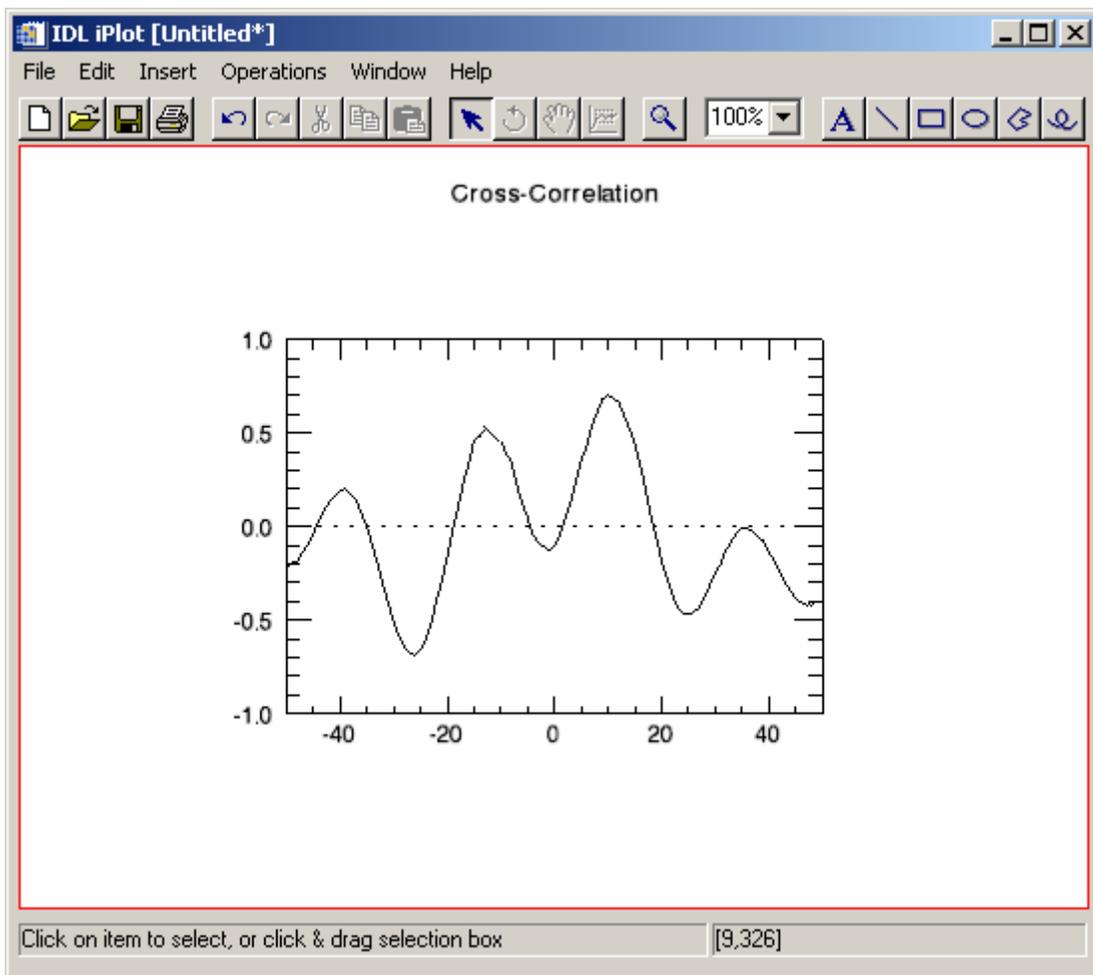


Figure 6: Cross-correlation between two series at a set of user-defined lag values

12. Once finished viewing the cross-correlation results, close the *IDL iPlot* window.

Signal Analysis Transforms

Most signals can be decomposed into a sum of discrete (usually sinusoidal) signal components. The result of such decomposition is a frequency spectrum that can uniquely identify the signal. IDL provides three main transforms to decompose a signal and prepare it for analysis :

- The Discrete Fourier Transform
- The Hilbert Transform
- The Wavelet Transform

The **Discrete Fourier Transform** (DFT) is the most widely used method for determining the frequency spectra of digital signals (and is the only transform that will be covered in this tutorial). This is due in part to the development of an efficient computer algorithm for computing DFTs known as the **Fast Fourier Transform** (FFT). IDL implements the Fast Fourier Transform in its *FFT* function. The DFT is essentially a way of estimating a Fourier transform at a finite number of discrete points, which works well with a digital signal since it is also sampled at discrete intervals.

The Fourier transform is a mathematical method that converts an input signal from the physical (time or space) domain into the frequency (or Fourier) domain. Functions in physical space are plotted as functions of time or space, whereas the same function transformed to Fourier space is plotted versus its frequency components. What the Fourier transform will help illustrate is that real signals can often be comprised of multiple sinusoidal waves (with the appropriate amplitude and phase) that when added together will re-create the real signal. In other words, the FFT algorithm decomposes the input time-series into an alternative set of basis functions, in this case sines and cosines.

The result of a FFT analysis is complex-valued coefficients, and *spectra* are derived from these coefficients in order to visualize the transform. Spectra are essentially the amplitude of the coefficients (or some product or power of the amplitudes) and are usually plotted versus frequency. There are a number of different ways to calculate spectra, but the primary forms are :

- Real and imaginary parts of Fourier coefficients
- Magnitude spectrum
- Phase spectrum
- Power spectrum

The most direct way to visualize spectra of a Fourier transformation of a signal is to plot the **real and imaginary parts** of the spectrum as a function of frequency. For example, calculate and display a Fourier transform for a new signal that consists of a sine wave. For simplicity assume the vector "x" contains time-series values in units of seconds :

1. IDL> sine = SIN (3.15 * x)
2. IDL> dft = FFT (sine)
3. IDL> real = FLOAT (SHIFT (dft, n/2-1))
4. IDL> imaginary = IMAGINARY (SHIFT (dft, n/2-1))

5. IDL> iPlot, x, sine, VIEW_GR=[2,2], VIEW_TI='Original Signal'
6. IDL> iPlot, real, /VIEW_NE, VIEW_TI='Real Part'
7. IDL> iPlot, imaginary, /VIEW_NE, VIEW_TI='Imaginary Part'

Although the real and imaginary parts contain some useful information, it is more common to display the **magnitude spectrum** since it actually has physical significance. Since there is a one-to-one correspondence between a complex number and its magnitude (and phase for that matter), no information is lost in the transformation from a complex spectrum to its magnitude. In IDL, the magnitude can be easily computed using the absolute value (*ABS*) function :

8. IDL> magSpec = ABS (dft)
9. IDL> iPlot, magSpec, /VIEW_NE, XRANGE=[0,n/2], XTITLE='Mode', \$
YTITLE='Spectral Density'

The resulting *IDL iPlot* visualization window should look similar to Fig. 7.

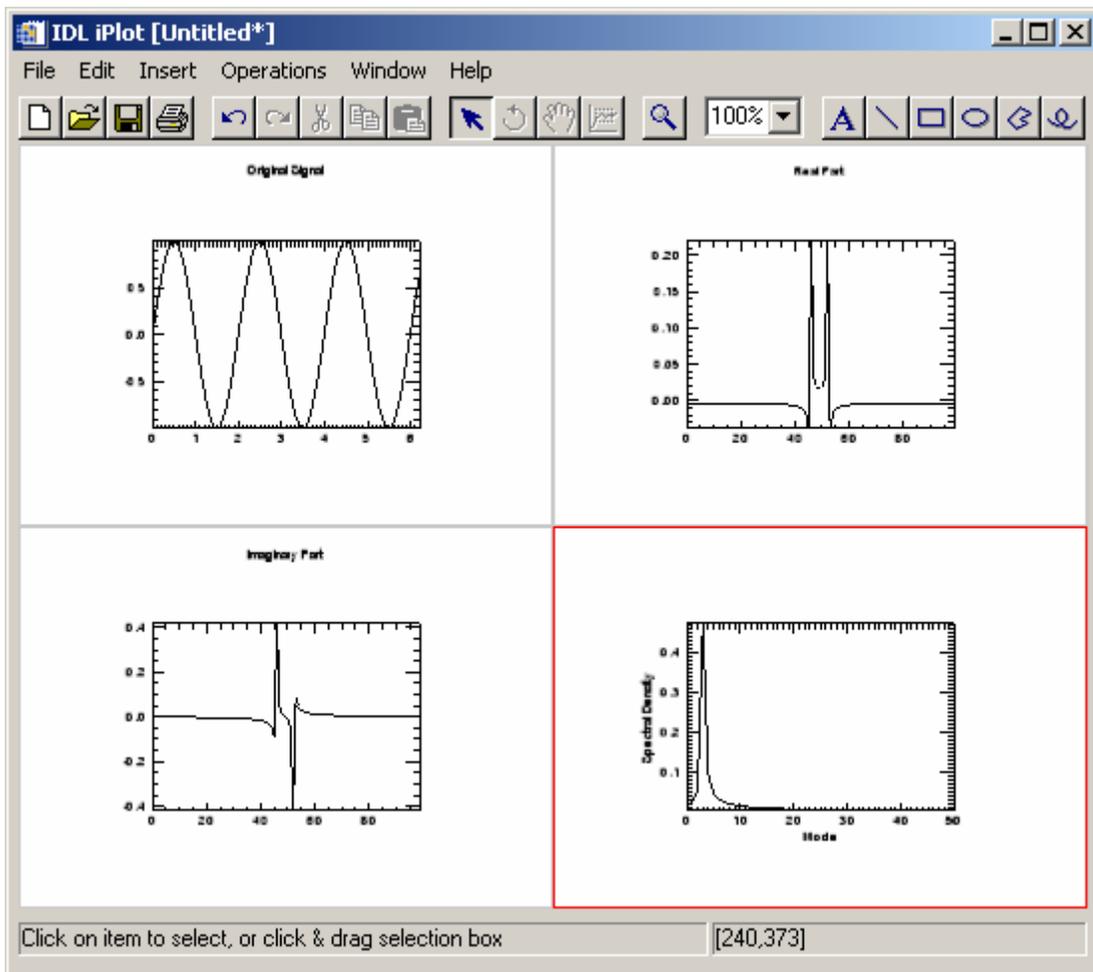


Figure 7: Real and imaginary parts of DFT plus the magnitude spectrum for the signal with sinusoidal component

From this magnitude spectrum the mode at which the sinusoidal pattern occurs matches the peak of the spectral density (in this case Mode=3.15). Assuming the independent time variable "x" is in units of seconds, the plot of the original signal on top shows that this sinusoidal component (SIN(3.15*x)) has a wavelength (X-distance of one sine cycle) of approximately 2 seconds. Consequently, the frequency of this sinusoidal component is approximately (1 / 2 seconds) = 0.5 Hz.

10. Once finished viewing the magnitude spectrum, close the *IDL iPlot* window.

In addition to the sine component, add a cosine variation of a sinusoidal pattern to the signal with a wavelength of 0.5 seconds and a frequency of 2 Hz :

11. IDL> cosine = COS (12.5 * x)
12. IDL> y = sine + cosine
13. IDL> iPlot, x, sine, COLOR=[255,0,0], NAME='SIN (3.15 * x)'
14. IDL> iPlot, x, cosine, COLOR=[0,0,255], NAME='COS (12.5 * x)', \$
/OVER
15. IDL> iPlot, x, y, COLOR=[0,128,0], NAME='Combined Signal', \$
/OVER
16. Using the mouse cursor, click and drag a rectangular selection box that encompasses the entire plot dataspace so that everything is highlighted.
17. From the *iPlot* menu system select "*Insert > New Legend*" in order to insert a legend.

The resulting *IDL iPlot* visualization window should look similar to Fig. 8.

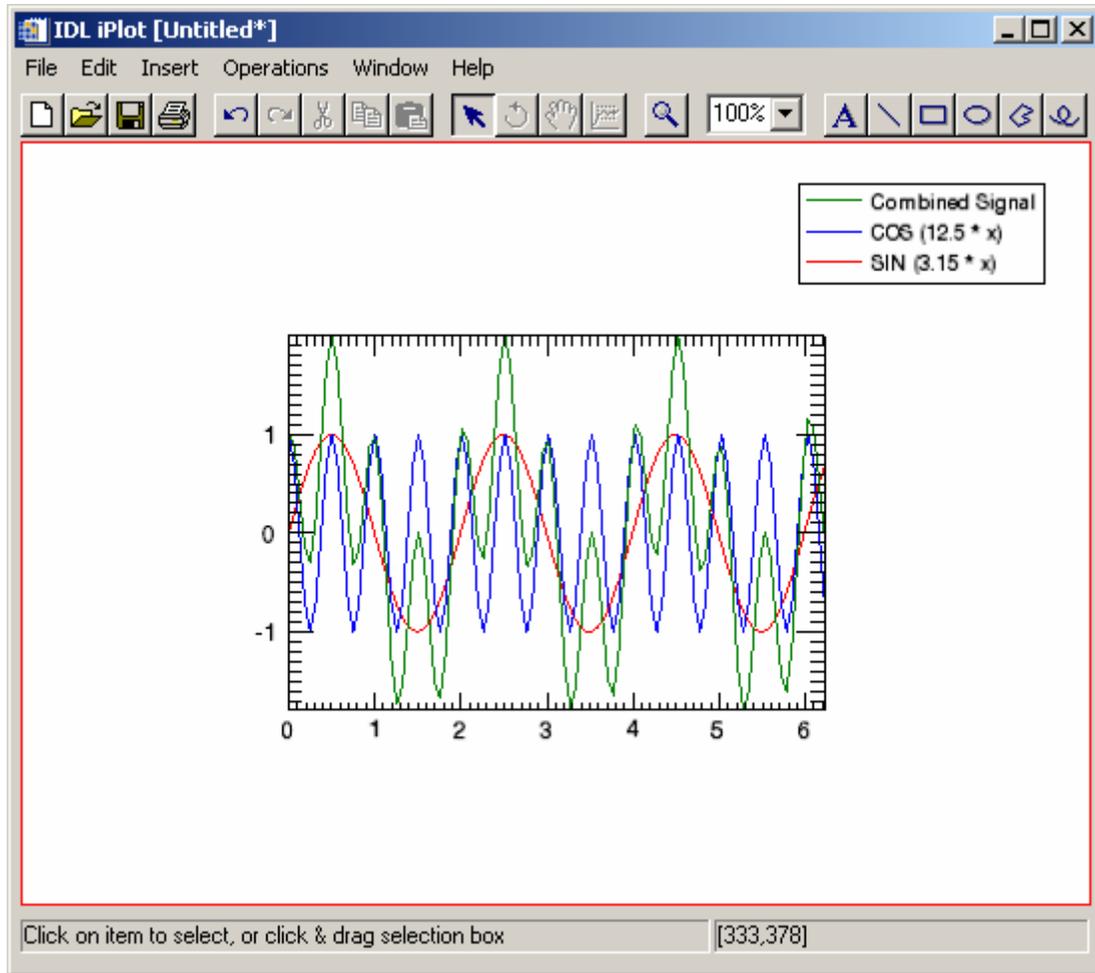


Figure 8: Synthetic signal created from addition of two sinusoidal components

18. Once finished viewing the combined signal, close the *IDL iPlot* window.

Signals are usually sampled at regularly spaced intervals in time. For example, the test signal dataset currently contained in the IDL variable "y" is a discrete representation of three superposed sine and cosine waves at $n=100$ points. In this case, the independent variable "x" that was used to create the test signal "y" is sampled at regularly spaced intervals of time, which is assumed to be in units of seconds. Consequently, the difference between any two neighboring elements in this vector "x" is known as the **sampling interval** (δ) :

```
19. IDL> samplingInterval = x[1] - x[0]
20. IDL> HELP, samplingInterval
    SAMPLINGINTERVAL  FLOAT      =      0.0628319
```

Since it is assumed that the time values are in units of seconds, it follows that the sampling frequency for this test signal is ~ 16 Hz :

```
21. IDL> samplingFreq = 1 / samplingInterval
```



```
22.IDL> HELP, samplingFreq
      SAMPLINGFREQ      FLOAT      =      15.9155
```

The **Nyquist frequency** (or “folding frequency”) is a special property of the sampling interval. The Nyquist frequency is basically the critical sampling interval at which measurements must be made in order to accurately portray the signal with only discrete values. For example, the critical sampling of a simple sine wave is two sample points per cycle. In other words, a digital signal has to have a sampling interval that is equal to or smaller than $\frac{1}{2}$ the sinusoidal component’s wavelength. Consequently, the Nyquist Frequency is defined as $(1/2*\delta)$:

```
23.IDL> Nyquist = 1 / (2 * samplingInterval)
24.IDL> HELP, Nyquist
      NYQUIST      FLOAT      =      7.95775
```

If a discretely sampled signal is not bandwidth-limited to frequencies less than the Nyquist frequency then **aliasing** occurs. When aliasing occurs the power spectrum will have a feature that has been falsely translated back across the Nyquist frequency. For example, construct another signal that adds a sinusoidal component with a frequency greater than the Nyquist frequency (7.96 Hz in this example). The new component will be $(\text{SIN}(70*x))$, which has a wavelength of approximately 0.089 seconds and a frequency of 11.14 Hz :

```
25.IDL> frq = FINDGEN (n) * samplingFreq / (n-1)
26.IDL> y = sine + cosine + SIN (70*x)
27.IDL> dft = FFT (y)
28.IDL> magSpec = ABS (dft)
29.IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
30.IDL> iPlot, frq, magSpec, /VIEW_NE, XRANGE=[0,Nyquist], $
      XTITLE='Frequency (Hz)', YTITLE='Spectral Density'
```

The resulting *IDL iPlot* visualization window should look similar to Fig. 9.

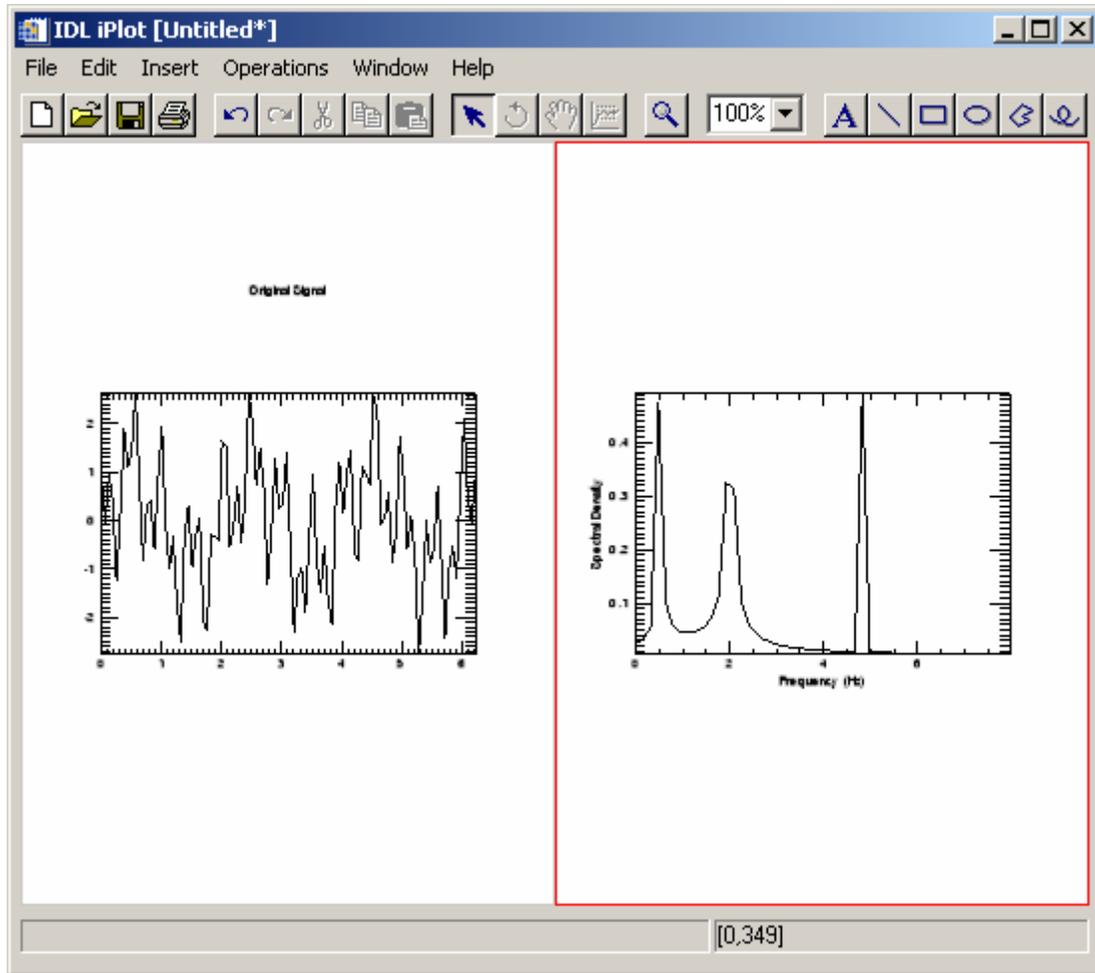


Figure 9: Magnitude spectrum for signal with sinusoidal components

The original “sine” and “cosine” components (at 0.5 Hz and 2 Hz respectively) are readily apparent in this magnitude spectrum, but there is also an aliased feature located at 4.77 Hz (Nyquist - (11.14-Nyquist)). Furthermore, the shape of the magnitude spikes vary from relatively narrow spikes to more diffuse features. For instance, the peak at 2 Hz is more spread-out, and this is due to an effect known as smearing or **leakage**. The leakage effect is a direct result of the definition of the DFT and is not due to any inaccuracy in the FFT.

Note: Leakage can be reduced by increasing the length of the time sequence, or by choosing a sample size that includes an integral number of cycles of the frequency component of interest.

31. Once finished viewing the magnitude spectrum, close the *IDL iPlot* window.

The **phase spectrum** of a Fourier transform can also be easily computed in IDL using the arc-tangent (ATAN) function. By convention, the phase spectrum is usually plotted versus frequency on a logarithmic scale :

```
32. IDL> y = sine + cosine
33. IDL> dft = FFT (y)
34. IDL> phiSpec = ATAN (dft)
```

```

35. IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
36. IDL> iPlot, frq, phiSpec, /VIEW_NE, VIEW_TI='Phase Spectrum', $
    XRANGE=[1,Nyquist], YRANGE=[0,0.006]

```

The resulting *IDL iPlot* visualization window should look similar to Fig. 10.

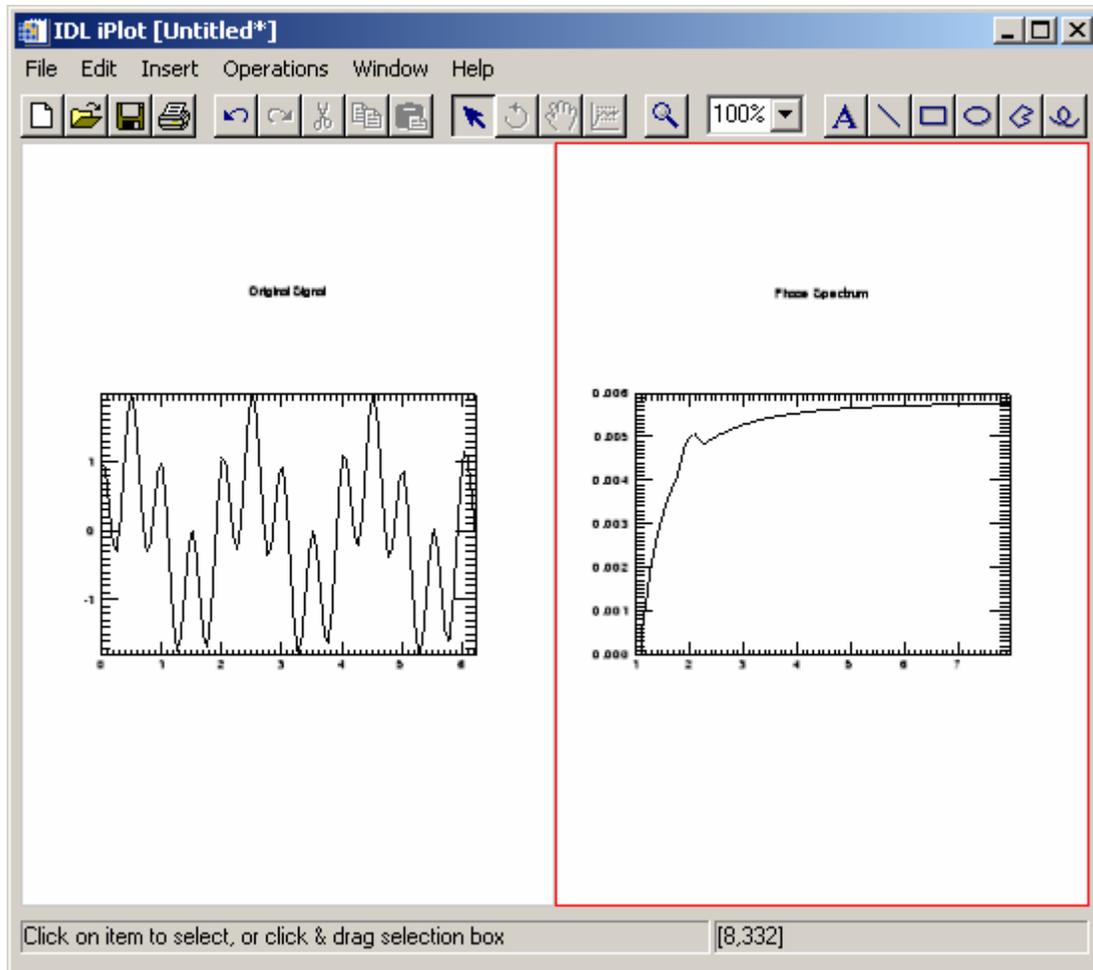


Figure 10: Phase spectrum for signal with sinusoidal components

The same sinusoidal frequency component in the signal at 2 Hz shows up as a discontinuity in this phase plot.

37. Once finished viewing the phase spectrum, close the *IDL iPlot* window.

Finally, another way to visualize the same information is with the *power spectrum*, which is the square of the magnitude of the complex spectrum :

```

38. IDL> powSpec = (ABS (dft) ) ^ 2
39. IDL> iPlot, x, y, VIEW_GR=[2,1], VIEW_TI='Original Signal'
40. IDL> iPlot, frq, powSpec, /VIEW_NE, XRANGE=[0,Nyquist], $
    VIEW_TI='Power Spectrum'

```

The resulting *IDL iPlot* visualization window should look similar to Fig. 11.

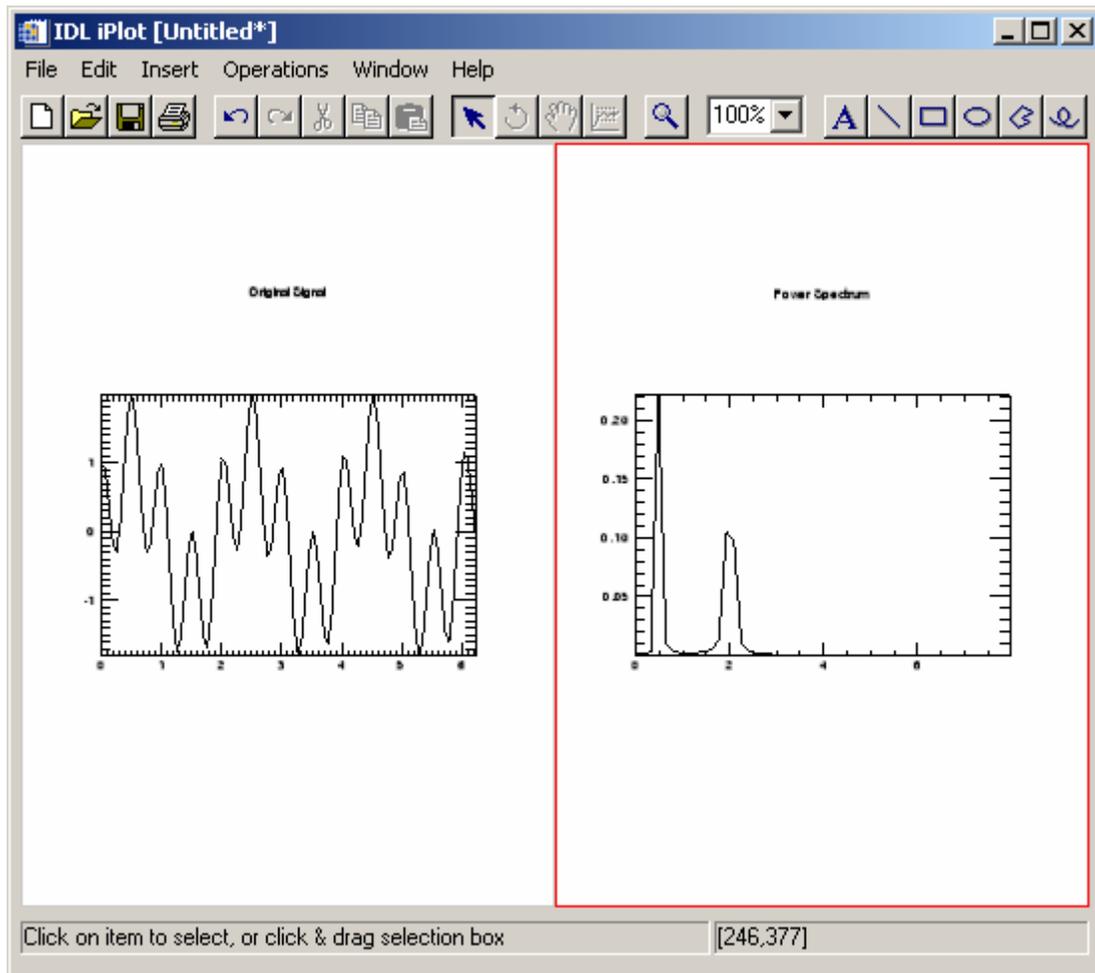


Figure 11: Power spectrum for signal with sinusoidal components

41. Once finished viewing the power spectrum, close the *IDL iPlot* window.

Windowing

The leakage effect mentioned previously is a direct consequence of the definition of the Discrete Fourier Transform, and of the fact that a finite time sample of a signal often does not include an integral number of some of the frequency components in the signal. The effect of this leakage can be reduced by increasing the length of the time sequence or by employing a windowing algorithm. IDL's HANNING function computes two windows that are widely used in signal processing :

- Hanning Window
- Hamming Window

For example, compare the power spectrum from the original signal to one that has the Hanning window applied :

1. IDL> hann = FFT (HANNING (n) * y)
2. IDL> hannPowSpec = (ABS (hann)) ^ 2
3. IDL> iPlot, frq, powSpec, XRANGE=[0,3], XTITLE='Frequency (Hz)', \$
YTITLE='PSD', LINESSTYLE=2
4. IDL> iPlot, frq, hannPowSpec, XRANGE=[0,3], /OVER

The resulting *IDL iPlot* visualization window should look similar to Fig. 12.

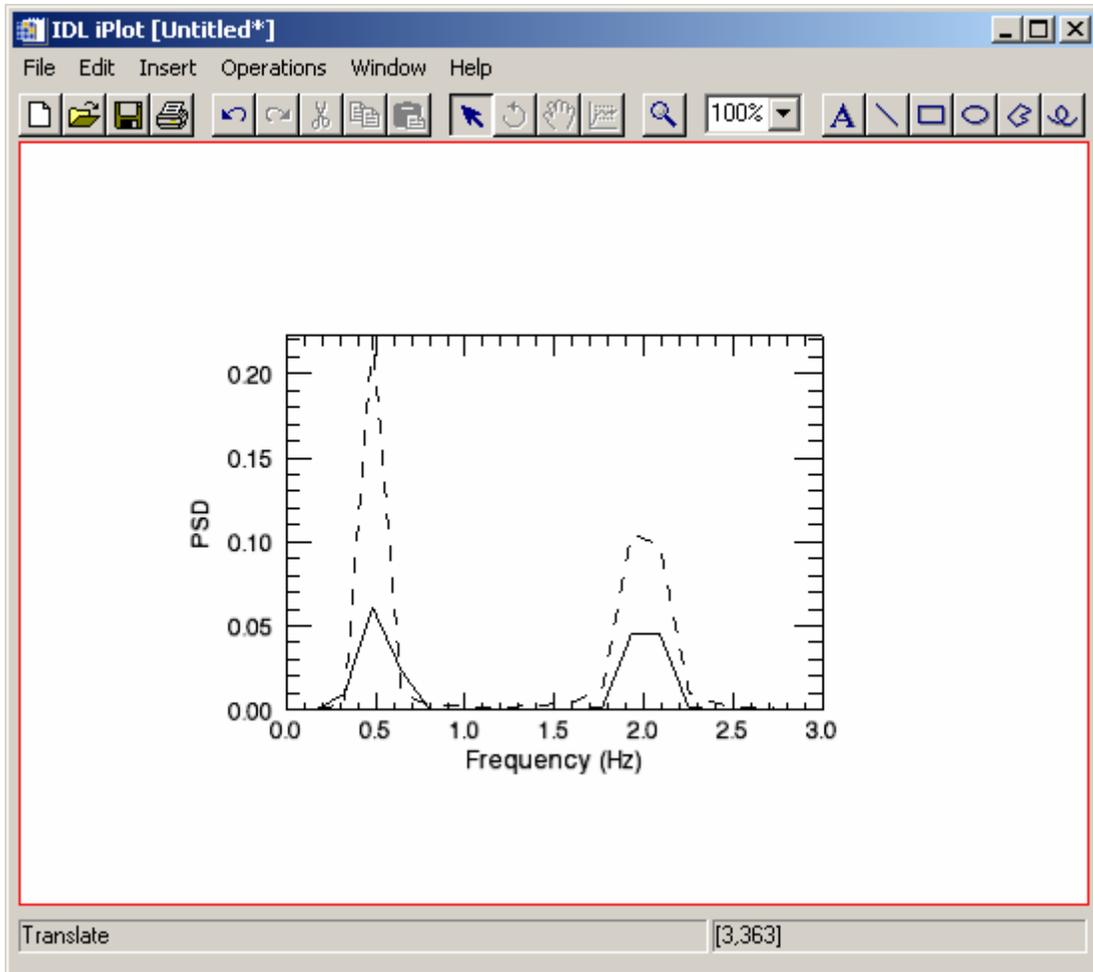


Figure 12: Power spectrum of original (dashed) and Hanning windowed (solid) signal

The power spectrum of the Hanning windowed signal shows some mitigation of the leakage effect.

5. Once finished viewing the power spectrums, close the *IDL iPlot* window.

Wavelet Analysis

Wavelet analysis is becoming a popular technique for both signal and image analysis. By decomposing a signal using a particular wavelet function, one can construct a picture of the energy within the signal as a function of both spatial dimension (or time) and wavelet scale (or frequency). The wavelet transform is used in numerous fields such as geophysics (seismic events), medicine (EKG and medical imaging), astronomy (image processing), and computer science (object recognition and image compression).

The *WV_CWT* function within IDL can be utilized to calculate the continuous wavelet transform for a signal. Use the *WV_CWT* function to compute the transform for the signal currently stored in the variable "y" using a Gaussian wavelet function :

```
1. IDL> cwt = WV_CWT (y, 'Gaussian', 2)
```

Next, create the power spectrum for this continuous wavelet transform :

```
2. IDL> ps = ABS (cwt) ^ 2
```

An effective method for visualizing the continuous wavelet power spectrum is by using an image display with a color palette applied. The *LOADCT* routine within IDL can be used to automatically create the appropriate color table vectors for a number of pre-defined color palettes. Use the *LOADCT* routine in conjunction with *TVLCT* to create the color table for a rainbow paletter ranging from blue to red :

```
3. IDL> ct = BYTARR (3, 256)
4. IDL> LOADCT, 34
5. IDL> TVLCT, r, g, b, /GET
6. IDL> ct[0,*] = r
7. IDL> ct[1,*] = g
8. IDL> ct[2,*] = b
```

Finally, display the continuous wavelet power spectrum within the *iImage* utility :

```
9. IDL> iImage, ps, RGB_TABLE=ct, VIEW_TI='Wavelet Power Spectrum'
```

The resulting *IDL iImage* visualization window should look similar to Fig. 13.

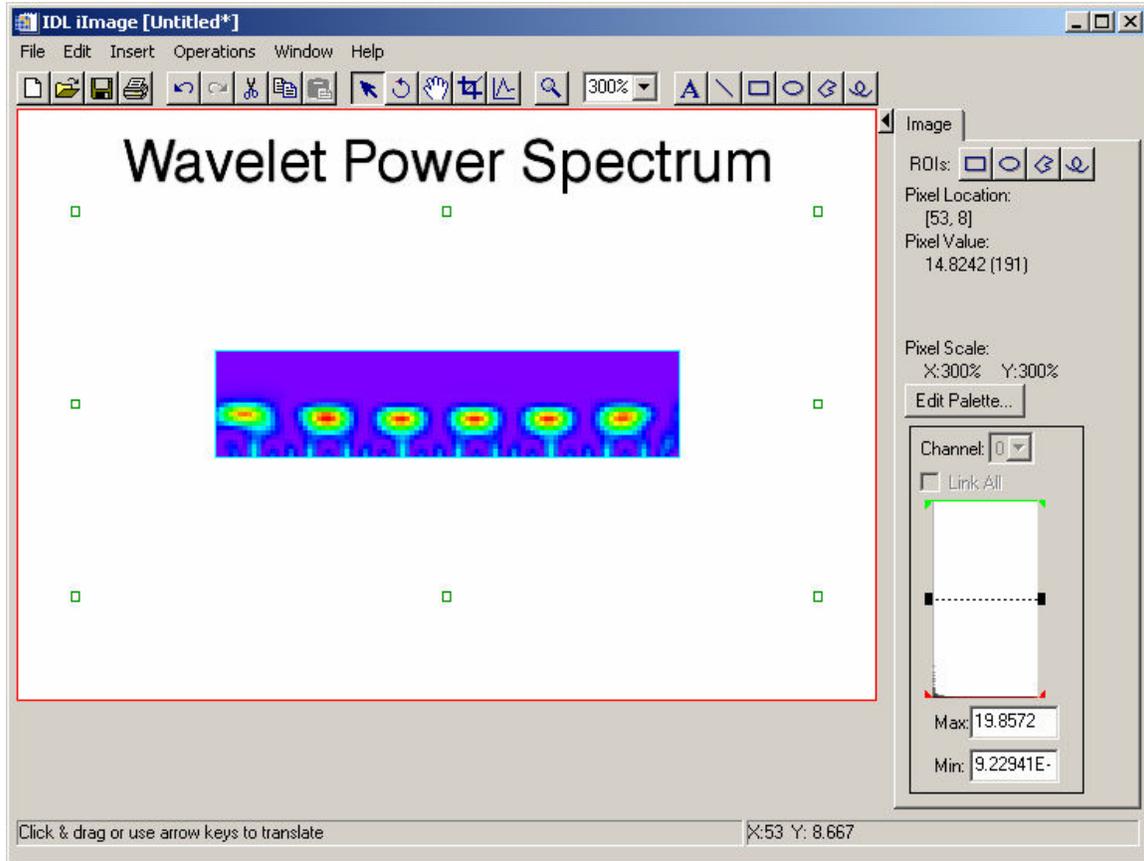


Figure 13: Continuous wavelet power spectrum displayed as an image

The display of the wavelet power spectrum clearly illustrates the sinusoidal components contained in the input signal.

10. Once finished viewing the continuous wavelet power spectrum, close the *IDL iImage* window.

© 2008 ITT Visual Information Solutions
All Rights Reserved

IDL® is a registered trademark of ITT Visual Information Solutions for the computer software described herein and its associated documentation. All other product names and/or logos are trademarks of their respective owners.

The information contained in this document pertains to software products and services that are subject to the controls of the Export Administration Regulations (EAR). All products and generic services described have been classified as EAR99 under U.S. Export Control laws and regulations, and may be re-transferred to any destination other than those expressly prohibited by U.S. laws and regulations. The recipient is responsible for ensuring compliance to all applicable U.S. Export Control laws and regulations.