C# FFT signal-processing .NET4.8 VS2022

DSPLib - FFT / DFT Fourier Transform Library for .NET 4

Steve Hageman

4.93/5 (69 votes)

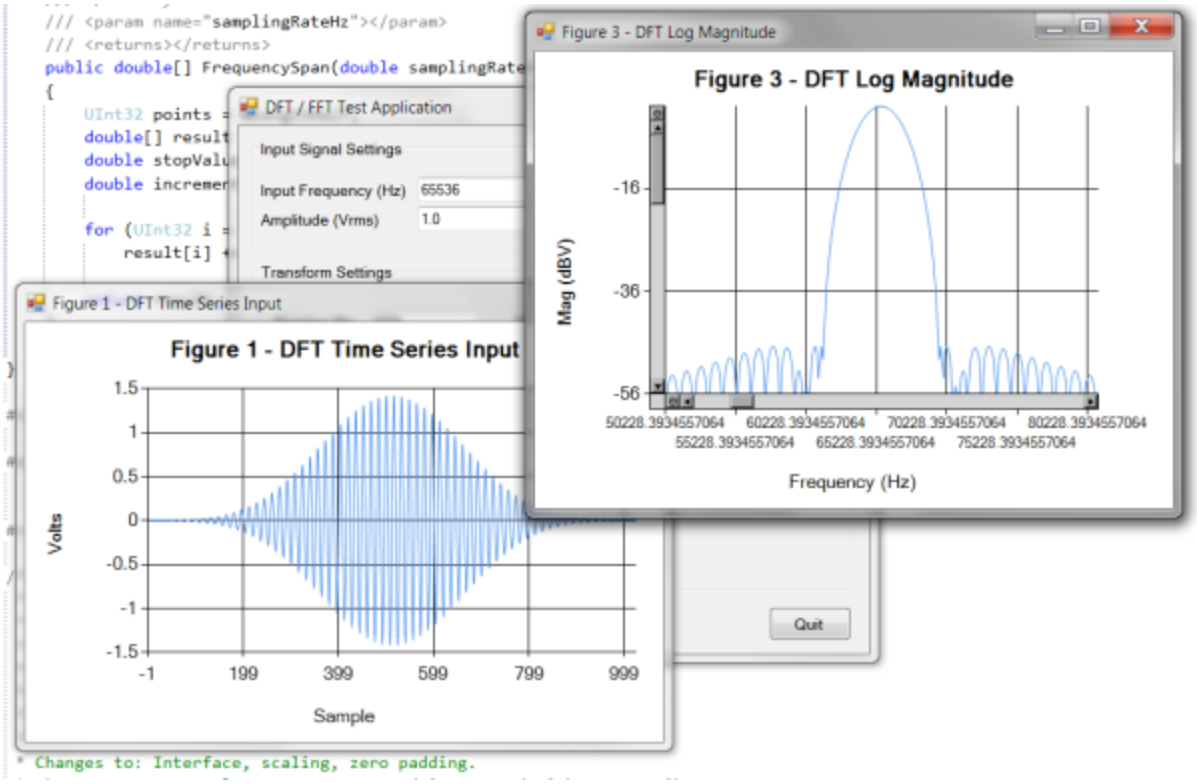
11 Jul 2023 MIT 26 min read 203.2K 17.9K

DSPLib is a complete DSP Library that is an end to end solution for performing FFT's with .NET 4

In this post, you will find a practical, organized and complete .NET 4+ Open Source library of DSP oriented routines released under the very non-restrictive MIT License.

Download DSPLib Library Files V2.0 - 12.2 KB

Download Test Project + Examples + Library Files V2.0 - 33 KB



Introduction

There is a real need for a ready to use Fourier Transform Library that users can take right out of the box and perform Fast Fourier Transforms (FFT) or Discrete Fourier Transforms (DFT) and get a classical spectrum versus frequency plot.

The vast majority of code that you will find in Commercial packages, Open Source libraries, Textbooks and on the Web are simply unsuited for this task and takes hours of further tweaking to get a classic and properly scaled spectrum plot.

The library presented here is a practical, organized and complete .NET 4+ Open Source library of DSP oriented routines released under the very non-restrictive MIT License.

What DSPLib Does

DSPLib has several main parts, but its basic goal is to allow a real Fourier Transform to be preformed on a time series input array, resulting in a usable classic spectrum output without any further tweaking required by the user.

Basic Fourier Transforms (FT) come in two basic types: the most general form can produce a spectrum output from any length of input data. This type of transform is called the Discrete Fourier Transform or DFT. The code is simple and brute force.

- The pros are: The input data can be any length.
- The cons are: Since it is a general method, it is computationally intensive and large input data sets can take a very long time to calculate.

A more specific type of FT is called the Fast Fourier Transform or FFT.

- The pros are: It is much, much faster to compute than the DFT.
- The cons are: The input data length is constrained to be power of twos. The code is more complex to understand.

As an example: A 8192 point FFT takes: less than 1 Millisecond on my i7 Computer. A DFT of the same length takes 360 Milliseconds. Hence you can see why the FFT is much more popular than the brute force DFT in real time applications.

DSPLib implements both kinds of Fourier Transform.

***Note**: For the remainder of this article, the generalized Fourier Transform will be referred to as a 'FT' when the discussion can apply to either a 'FFT' or 'DFT' as they both produce the same result for equivalent input data.*

All FTs can take a real or complex number input and naturally produce a complex number result. All FTs produced to date have implemented their own Complex Number type and this naturally leads to incompatibilities between libraries. .NET 4.0 finally includes (in the `System.Numerics` namespace) a Complex number structure and many math methods for operating on complex numbers. `DSPLib` incorporates the .NET 4 Complex Number Type.

To have real speed improvements and to automatically scale the speed on processors with multiple cores and/or threads, `DSPLib` also implements the Task Parallel Extensions built into .NET 4. This leads to a real improvement in execution time that automatically scales with the number of processor cores / threads available. For instance, on a 2 core / 4 thread i7 processor, using the Task Parallel extensions decreases the execution time of a 10,000 point DFT by more than 3X.

Smart caching of Sine and Cosine multiplication terms on smaller DFTs also increases performance by around 3X.

Both of these easy to implement features increase the raw DFT Speed by around 9 times even on a low end i7 processor.

Real and Imaginary Spectrum Parts

All FT implementations naturally produce an output data array that is the same length as the input array. The output however consists not only of complex numbers, but Real and Imaginary parts of the Spectrum itself – sometimes called negative frequencies depending on how one wants to think about it. The Imaginary part of the spectrum is just the mirror image of the Real part and does not contain any additional information about the spectrum.

`DSPLib` returns only the real part of the spectrum output data array (one half the length of the input data array, not including the FS/2 data point). All the other libraries that you will find give you the entire Real and Imaginary parts as a return value. This is a source of confusion for many people and it is not useful for generating the usual real spectrum data representation.

***Note**: If you need the Imaginary and Real parts of the complex output spectrum for some mathematical reason, that's fine, you know why you need it. DSPLib is geared towards generating scaled classical spectrum plots, not general FT mathematics.*

Fourier Transform Scaling

Most of the FT implementations you will find are not scaled properly for classical spectrum displays. That is, the amplitude of a transformed signal varies with the number of transformed points: N. This is disconcerting to most users. Most users expect a properly scaled output independent of the number of points used in the FT.

For instance, if a user inputs a 10 Hz signal of 1 Vrms amplitude, then they expect to see a 1 Vrms peak at 10 Hz in the spectrum plot. `DSPLib` is the only library to date that properly scales the FT outputs so that they are accurate, no matter how you change the number of points in the transform.

***Note**: All DFT and FFT implementations also have scaling differences at DC (Bin 0) and at half the Sampling Frequency (FS/2). These points have no imaginary part, only a real part. Hence, the required scaling is different at these points also. DSPLib applies the proper scale to the DC point. The FS/2 point is considered imaginary and by convention it is not included in the resulting real part of the spectrum output.*

Scaling with Added Data Windowing

You may know that all FTs assume that the chunk of time limited data that you input actually extends for infinite time both before the chunk and afterwards. If you transform a pure sine wave with an even number of cycles, the spectrum display will be a perfect peak centered in one bin of the output spectrum.

In real life, it is usually not possible to generate exactly even input sequences. Any fractional cycles and the output will be smeared across the entire resulting spectrum. The amplitude of fractional cycles will also appear to be lower than they actually are.

This is where using a window comes in to play. A window is applied to the input data and it tapers the beginning and end of the time series to zero or near zero. The use of a window, thus tricks the DFT into not noticing the discontinuities in the input data.

Windows are normally picked based on four criteria:

1. How low in amplitude they push the undesired sidelobes
2. How much they reduce the amplitude uncertainty
3. How many FT Bins they smear the data into
4. Historical reasons, i.e., "Everybody uses Window: xyz, in this situation.", etc.

Picking the optimum window involves many tradeoffs.

`DSPLib`'s windowing is based on an excellent article written by the staff at the Max Planck Institute [1] and includes all the usual types of windows like: Welch, Hamming and Hann (or Hanning as it is commonly called) and 27 other very useful but perhaps less well known windows. Also included are 17 very unique Flat Top windows that give very low amplitude error even if the input frequency is in between the FT output Bins.

Since Windowing the input data changes it physically, it can also change the resulting FT Spectrum output amplitudes. `DSPLib` includes two functions that allow the window gain (or loss) to be accounted for so that the properly scaled FT output can be found and displayed. One of the scale factors relates to the peak of discrete signals. The other type of scale factor must be used when noise is being measured.

After a set of window coefficients is generated, they are input to the appropriate Scale Factor functions(s) and a scalar scale factor number is calculated. This scale factor is applied by multiplication to the resulting spectrum to correct the amplitude.

More information on the need and selection of windows may be found in reference [2].

Zero Padding and Scaling

Zero padding is a very useful trick that is used with FFTs and DFTs. One use is to round the length of input data up to the next power of two so that a faster FFT can be used for the transform method. For instance, if you had 1000 points of input data, you can zero pad an additional 24 points onto the data so that a 1024 point FFT can be preformed instead of a slower 1000 point DFT (nearly a 9X speed improvement on my computer).

Another use for zero padding is to make the display look more like an expected spectrum display. Zero padding broadens any peak(s) by interpolating points on the display and this makes a plot look better on a display screen or a plot. Zero padding also interpolates the space between calculated points reducing amplitude error when signals are not directly at the bin centers of the FT.

Finally, zero padding allows a very fine look at the sidelobes and sideband suppression added by the window.

Zero padding has an influence on the resulting spectrum output amplitudes even when the signals are at the bin centers. The FT routines presented here take an initialization parameter that includes the desired zero padding number so that the proper re-scaling factors can automatically be taken into account. Since the FT routines know about the desired zero padding for scaling reasons, they

add the requested amount of zeros to the input data so the user does not have to.

Reference [2] has more information on Zero Padding and even more uses for it.

Test Data Generation

Real data to analyze isn't always available during program development and debugging. `DSPLib` helps by including two calibrated Sine Wave generators and a calibrated Gaussian random noise generator for making lifelike test signals that can also include noise.

General Spectrum Output Manipulation Methods

As mentioned, the raw output of any FT is an array of complex numbers. In addition to the .NET 4 Complex number type operators, `DSPLib` includes a number of easy to use conversion routines for converting the Complex result into a scalar array of either Linear or Log Magnitude formats.

Additionally, a number of math routines that operate on scalar arrays are included. These allow addition, subtraction, multiplication and division between two arrays or between an array and a constant.

Examples

Enough talk about the general aspects of `DSPLib`. The examples below will show how easy it is to apply in practice.

Example 1

C# Shrink ▲

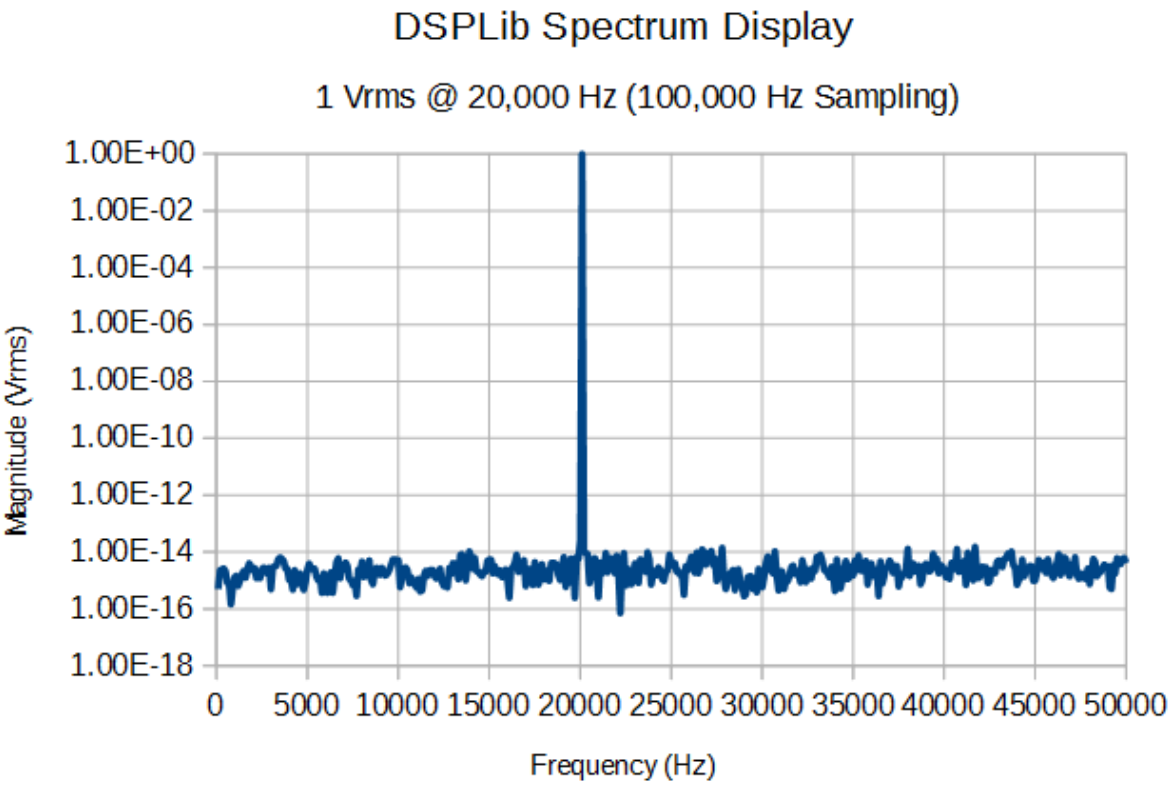
```
void example1()
{
  // Generate a test signal,
  //  1 Vrms at 20,000 Hz
  //  Sampling Rate = 100,000 Hz
  //  DFT Length is 1000 Points
  double amplitude = 1.0;
  double frequency = 20000;
  UInt32 length = 1000;
  double samplingRate = 100000;

  double[] inputSignal =
           DSP.Generate.ToneSampling(amplitude, frequency, samplingRate, length);

  // Instantiate a new DFT
  DFT dft = new DFT();

  // Initialize the DFT
  // You only need to do this once or if you change any of the DFT parameters.
```

```
  dft.Initialize(length);

  // Call the DFT and get the scaled spectrum back
  Complex[] cSpectrum = dft.Execute(inputSignal);

  // Convert the complex spectrum to magnitude
  double[] lmSpectrum = DSP.ConvertComplex.ToMagnitude(cSpectrum);

  // Note: At this point, lmSpectrum is a 501 byte array that
  // contains a properly scaled Spectrum from 0 - 50,000 Hz (1/2 the Sampling Frequency)

  // For plotting on an XY Scatter plot, generate the X Axis frequency Span
  double[] freqSpan = dft.FrequencySpan(samplingRate);

  // At this point a XY Scatter plot can be generated from,
  // X axis => freqSpan
  // Y axis => lmSpectrum

  // In this example, the maximum value of 1 Vrms is located at bin 200 (20,000 Hz)
}
```

Example 1 – Basic Setup for Measuring a Signal with a DFT

In the example above, a test signal is generated, a DFT is instantiated and initialized and the DFT is executed. Scaling is done on the result and the magnitude result is available in the array: `lmSpectrum`.

Note: To use this code in your own project, see the section: "Adding DSPLib to your project" below.



Example 1 – Plot of output from the Example 1 code snippet

What about FFTs?

The procedure to use a FFT is exactly the same as in the DFT of Example 1, except: the input data length, plus any zero padding must be an exact power of two.

Actual code that can be substituted to instantiate an FFT instead:

C#

```
DSPLib.FFT fft = new DSPLib.FFT();
fft.Initialize(inputDataArray.Length);
Complex[] cSpectrum = fft.Execute(inputDataArray);
```

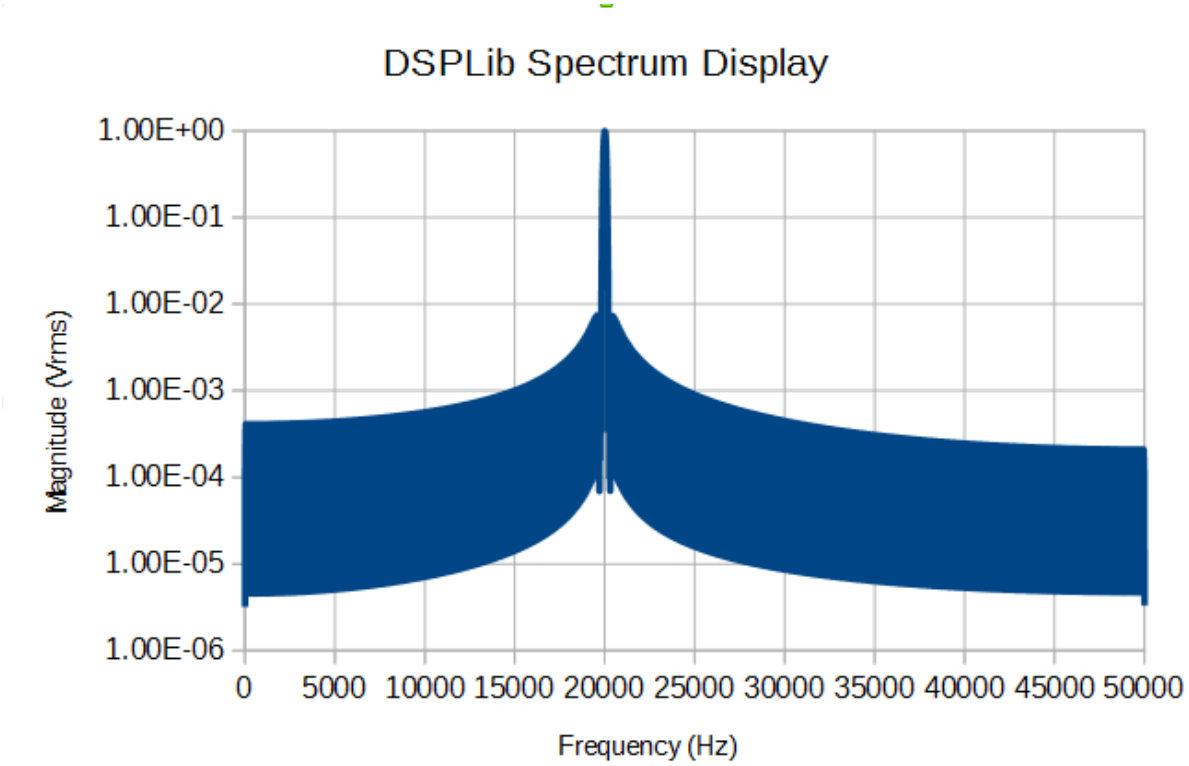
Example 2

C# Shrink ▲

```
void example2()
{
  // Same Input Signal as Example 1 - Except a fractional cycle for frequency.
  double amplitude = 1.0; double frequency = 20000.5;
  UInt32 length = 1000; UInt32 zeroPadding = 9000; // NOTE: Zero Padding
  double samplingRate = 100000;

  double[] inputSignal = DSPLib.DSP.Generate.ToneSampling
                         (amplitude, frequency, samplingRate, length);

  // Apply window to the Input Data & calculate Scale Factor
  double[] wCoefs = DSP.Window.Coefficients(DSP.Window.Type.Hamming, length);
  double[] wInputData = DSP.Math.Multiply(inputSignal, wCoefs);
  double wScaleFactor = DSP.Window.ScaleFactor.Signal(wCoefs);

  // Instantiate & Initialize a new DFT
  DSPLib.DFT dft = new DSPLib.DFT();
  dft.Initialize(length, zeroPadding); // NOTE: Zero Padding

  // Call the DFT and get the scaled spectrum back
  Complex[] cSpectrum = dft.Execute(wInputData);

  // Convert the complex spectrum to note: Magnitude Format
  double[] lmSpectrum = DSPLib.DSP.ConvertComplex.ToMagnitude(cSpectrum);

  // Properly scale the spectrum for the added window
  lmSpectrum = DSP.Math.Multiply(lmSpectrum, wScaleFactor);

  // For plotting on an XY Scatter plot generate the X Axis frequency Span
  double[] freqSpan = dft.FrequencySpan(samplingRate);

  // At this point a XY Scatter plot can be generated from,
  // X axis => freqSpan
  // Y axis => lmSpectrum
}
```

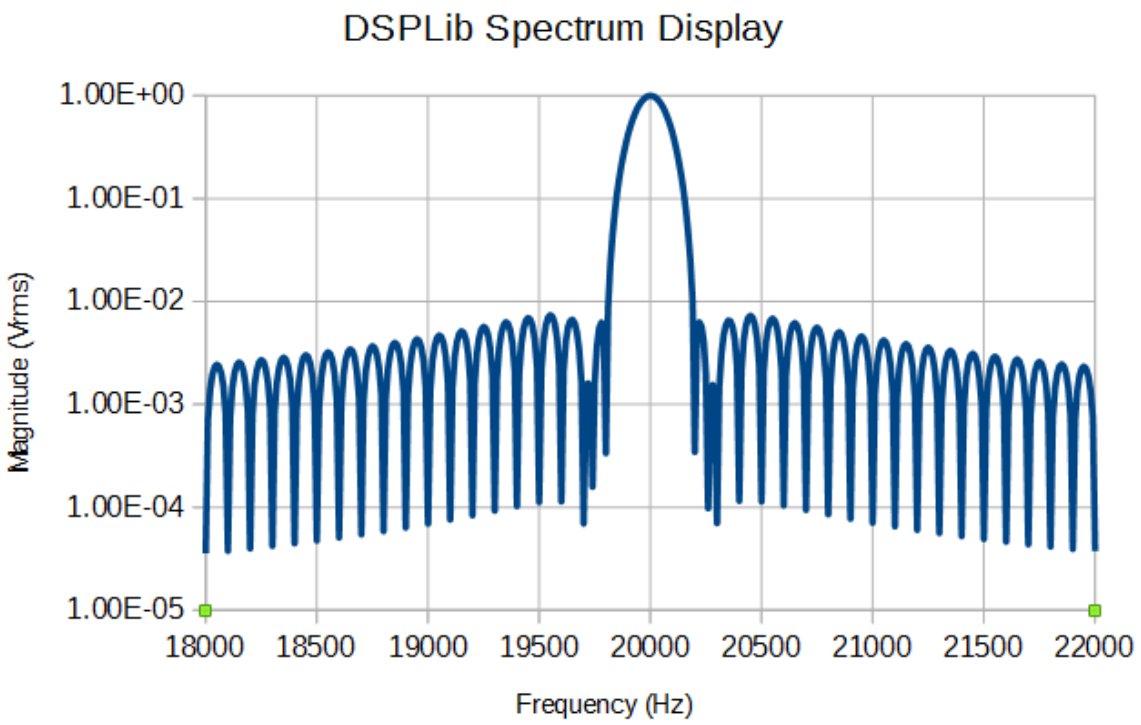
Example 2 – Basic setup for measuring a signal with a DFT & Windowing

In the example above, a test signal is generated, but this time with a fractional cycle (20000.5 Hz), a DFT is instantiated and initialized with zero padding to show the fine window detail. Finally, the DFT is executed. Scaling is done on the result and the magnitude result is available in the array: `lmSpectrum`.



Example 2 A – Plot of output from the Example 2 code snippet

Zero padding shows fine spectrum detail.



Example 2 B – Plot of output from the Example 2 code snippet

Zoomed in X-Axis shows fine spectrum detail due to the Window applied and the addition of zero padding. Also note that the amplitude is still correct even though the DFT length in this case is 1000 points + 9000 Zero Padding = 10,000 points total. Amplitude corrections are easily taken care of in `DSPLib`: Zero Padding is taken care of in the FT themselves, amplitude errors from applying windows are corrected with the windows scale factor functions.

Review of Basic Usage Steps for Using DSPLib

`DSPLib` was designed to be consistent in its use for both DFTs and FFTs. Additionally, Signals and Noise like signals can be analyzed following the same basic steps, as outlined below.

Basic Usage Steps - Analysis of Signal Magnitude

1. Instantiate & initialize FT, calculate window scale factor for signals
2. Get or generate input data
3. Apply window to input data
4. Execute the FT on windowed input data
5. Convert to magnitude format
6. If required: Average bin by bin, then repeat steps 2-6 for all averages
7. Apply window scale factor to data

Result: Properly scaled Spectrum for Signals

Basic Usage Steps - Analysis of Noise Signal

1. Instantiate & initialize FT, calculate window scale factor for noise
2. Get or generate input data
3. Apply window to input data
4. Execute the FT on windowed input data
5. Convert to magnitude squared format
6. If required: Average magnitude squared data bin by bin, then repeat steps 2-6 for all averages
7. Convert to magnitude format
8. Apply window scale factor to data.

Result: Properly scaled Spectrum for Noise Signals.

***Note**: The reasons why signals and noise must be treated differently is outlined in reference [2].*

Description of Test Application & Examples

The code download includes a sample test application that includes several test function code snippets and also includes a GUI based application that allows DFTs and FFTs to be preformed on a user definable test signal. This should be very helpful in understanding the use of `DSPLib`.

Example code snippets using `DSPLib` are contained in the "`DSPLib_Test`" solution, in a file called: "*DSPLib_ExampleCode.cs*" and included there are these examples:

- Example 1 – A basic DFT Example that generates data and produces a spectrum output
- Example 2 – Same as example 1, but also includes adding window and scaling factors
- Example 3 – Same as example 2, but also includes adding zero padding
- Example 4 – Same as example 2, but final result is Log Magnitude dBV units
- Example 5 – FFT Example with windowing
- Example 6 – DFT with noise signal and windowing, demonstrates the proper way to average and measure noise
- Example 7 - FFT with windowing and zero padding
- Example 8 - DFT with signal and noise floor added (Simulates a real system)
- Example 9 - DFT Phase extraction test
- Example 10 - FFT Phase extraction test with windowing, zero padding and automatic peak detection
- Example 11 - DFT Showing that the Task Parallel DFT can also be run on a Back Ground Worker thread
- Example 12 - DFT of complex input data (I&Q), also shows how to window IQ data and how to apply the scale factors

Complete Library Documentation

A complete Library Description of each section and function of `DSPLib` is presented below. The library also implements XML Comments that enables IntelliSense Help in the Visual Studio Environment.

Fourier Transforms

C#

```
void DSPLib.DFT()

void DSPLib.FFT()
```

Instantiates the DFT or FFT classes.

Note: These transforms are instantiated as objects because many times, I find it convenient to have many FTs running at the same time with different lengths for real time applications. By using separate objects, time does not have to be wasted redefining the FT length parameters between executing transforms.

C#

```
void DSPLib.DFT.Initialize
  (UInt32 inputDataLength, UInt32 zeroPaddingLength = 0, bool forceNoCache = false)

void DSPLib.FFT.Initialize(UInt32 inputDataLength, UInt32 zeroPaddingLength = 0)
```

- Initializes internal variables of the DFT or FFT class before use
- Must be called after instantiating the DFT or FFT object
- Only needs to be called once or when the FT definition changes
- `InputDataLength` - The length of the time series that is to be transformed
- `(Optional) zeroPaddingLength` - The optional number of zeros to add to the input data array before transformation. The FT "`Execute`" function will add the requested number of zeros to the input data array so the user does not have to.
- DFT specific parameters & properties

The DFT attempts to allocate some common Sine and Cosine multiplication terms in memory to speed the DFT calculations up. For small DFTs, this won't make much difference. For very large DFTs, there won't be enough memory to allocate the arrays anyway so this won't make any difference then either.

The DFT initialize function includes an optional parameter: `forceNoCache`. When set to `true`, this parameter will prevent the DFT from using pre-allocated memory for the DFT calculations. This may save stack space memory, but it may also result in a longer execution time. Default is:

`forceNoCache = false`.

Normally, if you are working with several DFTs in a program, you should initialize the biggest ones first to make sure that they have enough program memory to allocate arrays to speed up the DFT execution.

The DFT object implements a boolean read only property called `IsUsingCached` This property can be queried to see if the DFT as it is currently initialized is using cached values.

C#

```
Complex[] DSPLib.DFT.Execute(double[] inputData)
Complex[] DSPLib.DFT.Execute(Complex[] inputData)

Complex[] DSPLib.FFT.Execute(double[] inputData)
Complex[] DSPLib.FFT.Execute(Complex[] inputData)
```

- Executes a FT on the `inputData` array supplied
- Returns a `System.Numerics.Complex[]` array of the spectrum result
- Only the real part of the spectrum is returned (one half the input data length). This corresponds to: DC through the Sampling Frequency / 2.
- Note: In V2.0 an overload to the DFT and FFT Execute methods was added that accepts Complex[] data input types.

C#

```
double[] DSPLib.DFT.FrequencySpan(double samplingRateHz)

double[] DSPLib.FFT.FrequencySpan(double samplingRateHz)
```

A helper function to return an array of frequency points that correspond to each value in the FT output array. This function is useful in applications where an amplitude versus frequency plot needs to be made. This function depends on the '`Initialized`' function being called with the proper parameters first.

This array only needs to be regenerated when the number of points and or zero padding length changes in the FT or the sampling rate changes. If you call **`.Initialize()`** again, you probably need to call this function again.

Fourier Transform Format Conversion

C#

```
double[] DSPLib.ConvertComplex.ToMagnitude(Complex[] cSpectrum)

double[] DSPLib.ConvertComplex.ToMagnitudeDBV(Complex[] cSpectrum)

double[] DSPLib.ConvertComplex.ToMagnitudeSquared(Complex[] cSpectrum)
```

```
double[] DSPLib.ConvertComplex.ToPhaseDegrees(Complex[] cSpectrum)

double[] DSPLib.ConvertComplex.ToPhaseRadians(Complex[] cSpectrum)
```

This group of functions convert the `Complex[]` array of the FT output to more convenient end user formats. Several output formats are available:

- `ToMagnitude`: Represents the magnitude of the complex quantity at each bin
- `ToMagnitudeDBV`: Represents the magnitude in log10 format referenced to 1 volt RMS. Ex: 1 Vrms = 0 dBV, 0.1 Vrms = -20 dBV
- `ToMagnitudeSquared`: Represents the magnitude squared value of the spectrum. Magnitude squared (V^2) is proportional to power. Since noise is proportional to power, magnitude squared is proper format when averaging noise.
- `ToPhaseDegrees`: Represents the phase in degrees of the complex quantity at each point
- `ToPhaseRadians`: Represents the phase in radians of the complex quantity at each point

C#

```
double[] DSPLib.ConvertMagnitude.ToMagnitudeSquared(double[] spectrum)

double[] DSPLib.ConvertMagnitude.ToMagnitudeDBV(double[] spectrum)
```

This group of functions convert the Magnitude `double[]` array of the FT converted spectrum output to more convenient end use formats. Several output formats are available:

- `ToMagnitude`: Represents the magnitude of the complex quantity at each bin.
- `ToMagnitudeDBV`: Represents the magnitude in Log10 format referenced to 1 volt RMS. Ex: 1 Vrms = 0 dBV, 0.1 Vrms = -20 dBV
- `ToMagnitudeSquared`: Represents the magnitude squared value of the spectrum. Magnitude squared (V^2) is proportional to power. Since noise is proportional to power, magnitude squared is proper format when averaging noise.

C#

```
double[] DSPLib.ConvertMagnitudeSquared.ToMagnitude(double[] spectrum)

double[] DSPLib.ConvertMagnitudeSquared.ToMagnitudeDBV(double[] spectrum)
```

Functions to convert the `MagnitudeSquared double[]` array of the FT converted spectrum output to more convenient end use formats. Several output formats are available:

- `ToMagnitude`: Represents the magnitude of the complex quantity at each point
- `ToMagnitudeDBV`: Represents the magnitude in Log10 format referenced to 1 volt RMS. Ex: 1 Vrms = 0 dBV, 0.1 Vrms = -20 dBV

Spectrum Data Analysis Functions

C#

```
double DSPLib.Analyze.FindMaxAmplitude(double[] inData)

double DSPLib.Analyze.FindMaxPosition(double[] inData)

double DSPLib.Analyze.FindMaxFrequency(double[] inData, double[] fSpan)
```

Given an input array, these functions find the maximum quantity specified. Either, Maximum Amplitude found in the array, bin position where the maximum amplitude was found or frequency where maximum amplitude was found. These functions are designed to be used with the spectrum output and directly accept the output from the `ConvertTo` functions in any format.

C#

```
double DSPLib.Analyze.FindMean(double[] inData, UInt32 startBin=10, UInt32 stopBin=10)

double DSPLib.Analyze.FindRms(double[] inData, UInt32 startBin=10, UInt32 stopBin=10)
```

Given an input array, these functions determine the Mean value (Average) or RMS Value of the array. Since there is leakage from either DC or Fs/2 (the last data point) into the spectrum, optional parameters allow the first and last specified bins to be excluded from the result. The default of 10 bins from either end is more than enough to exclude all the leakage for any of the windows supplied with `DSPLib`.

C#

```
double[] DSPLib.Analyze.UnwrapPhaseDegrees(double[] inPhaseDeg)

double[] DSPLib.Analyze.UnwrapPhaseRadians(double[] inPhaseRad)
```

Calculated phase always has discontinuities at +/-PI intervals. These functions attempt to unwrap the input phase arrays to remove jump type discontinuities. This works well for clean signals, but as the signal to noise ratio declines, it is harder to determine the true discontinuities from noise. These functions are a good starting point, but with real world signals and low signal to noise ratios, best performance is when sufficient signal averaging is used. The behavior is modeled off of the equivalent Octave / Matlab functionality.

Window Coefficient Generation

C#

```
enum DSPLib.Window.Type.{WindowName}

double[] DSPLib.Window.Coefficients(DSPLib.Window.Type windowName, UInt32 points)
```

The windows are specified with an `enum` value of type: "`DSPLib.Window.Type`." This is passed into the function Coefficients along with the number of points desired. The result is an array of doubles

that can be applied by multiplying with the input data and also used by one of the scaling factor routines to determine the window gain or loss. The window names and parameters are specified in Appendix A.

Window Analysis and Scaling

C#

```
double DSPLib.Window.Signal(double[] winCoeffs)

double DSPLib.Window.Noise(double[] winCoeffs, double samplingFrequencyHz)

double DSPLib.Window.NENBW(double[] winCoeffs)
```

Given any set of window coefficients, these routines will determine the gain or loss of the supplied window data array. The window coefficients can be from any window, they are not limited to only the windows supplied by `DSPLib`.

The `Noise` function also takes into account the bin width in Hz to correct for multiple or fractional hertz bin-widths. See Example 6 for more information.

The function `NENBW` calculates the normalized, equivalent noise bandwidth of the supplied window coefficients in bins.

Signal Generation

These routines can be used to generate test signals for simulation purposes.

C#

```
DSPLib.Generate.LinSpace(double startVal, double stopVal, UInt32 points)
```

LinSpace generates a data array with linear spaced values between `startVal`, `StopVal` with number of points: points. (Similar to the Octave / Matlab function of the same name).

C#

```
double[] DSPLIb.Generate.ToneSampling(double amplitudeVrms, double frequencyHz,
  double samplingFrequencyHz, UInt32 points, double dcV = 0.0, double phaseDeg = 0)

double[] DSPLIb.Generate.ToneCycles(double amplitudeVrms, double cycles,
         UInt32 points, double dcV = 0.0, double phaseDeg = 0)
```

`ToneSampling` and `ToneCycles` generate a real `double[]` array of a sine wave tone signal with the specified number of points.

- `ToneSampling` generates the signal based on typical sampling properties.
- `ToneCycles` generates the tone based on point values.

Optional parameters are:

- `dcV`: A DC Voltage to add to the signal, may be positive or negative
- `PhaseDeg`: A phase angle in degrees to add to the generated signal. Multiple signals may be generated with different phase angles then added together using the included array math functions for further analysis.

C#

```
double[] DSPLIb.Generate.NoisePsd(double amplitudePsd,
                                  double samplingFrequencyHz, UInt32 points)

double[] DSPLIb.Generate.NoiseRms(double amplitudeRms, double samplingFrequencyHz,
                                  UInt32 points, double dcV)
```

The noise routines generate calibrated noise signals depending on the application requirements. Either a Power Spectral Density may be specified (PSD) or an RMS value for noise may be used.

The units for PSD are Vrms / rt-Hz (Volts RMS / Root-Hertz).

`NoiseRMS` has an optional value dcV for adding a DC Voltage to the resulting noise signal. Default = 0.0 Volts. **Note**: The DC Value is NOT included in the RMS Voltage calculation.

Array Math Helper Functions

***Note**: All functions defined as static.*

C#

```
double[] DSPLib.Math.Add(double[]a, double[]b)

double[] DSPLib.Math.Subtract(double[]a, double[]b)

double[] DSPLib.Math.Multiply(double[]a, double[]b)

double[] DSPLib.Math.Divide(double[]a, double[]b)

double[] DSPLib.Math.Add(double[]a, double b)

double[] DSPLib.Math.Subtract(double[]a, double b)

double[] DSPLib.Math.Multiply(double[]a, double b)

double[] DSPLib.Math.Divide(double[]a, double b)

double[] DSPLib.Math.RemoveMean(double[] a)

double[] DSPLib.Math.Sqrt(double[] a)

double[] DSPLib.Math.Square(double[] a)

double[] DSPLib.Math.Multiply(Complex[] a, double[] b)
```

`DSPLib` defines a set of useful math functions that allow `double[]` arrays to be added, subtracted, multiplied or divided by another `double[]` array or a constant value.

`RemoveMean` will remove the mean or average value from an input data array. This is useful for removing the DC value from an input data array.

`Sqrt` and `Square` are general purpose routines to either square or square root an input data array.

***Note**: In Debug build mode, asserts warn if the double[] a array does not match the length of the double[] b array.*

Adding DSPLib to Your Project

`DSPLib` requires .NET 4 and above.

- Download the *DSPLib_source.zip* from the link above. This zip file contains just the *DSPLib.cs* source file.
- Add the *DSPLib.cs* source file as an existing file to your project.
- `DSPlib` requires a reference to the `System.Numerics` libraries for the complex data types, so `System.Numerics` must be added as a reference to your project as well.
- For convenience, add "`using DSPLib;`" to the top of any file that needs access to the `DSPLib` Namespace.

The sample, test project can be downloaded using the link above. The Test Application (and solution) was written in VS2022. It should compile right out of the box. If you have any reference issues, add `System.Numerics` and `System.Windows.Forms.DataVisulization` (for the .NET Charting control) to the `DSPLib_Test` project.

The examples are located in the *DSPLib_Test.zip* file as a separate file called *DSPLib_ExampleCode.cs*. This file contains many sample usage scenarios as detailed above and will answer many common usage questions.

Optimization

It is a 64 bit world now with most Windows computers running some form of a 64 bit operating system. Based on my testing: Compiling with the CPU type set to x64 will not speed up any basic `DSPLib` operations, but it will increase the maximum size of a DFT with caching. Caching is where the `DFT.Initialize()` routine pre-calculates and stores Sine and Cosine terms in arrays for use in the DFT Execution, giving a 3x reduction in calculation time. Caching happens in either x32 or x64 bit mode, but the memory available for caching can be doubled when operating in x64 bit mode (see `DFT.Initialize()` description above).

Similarly, setting the compiler option to `Release` (or `Optimize`) instead of `Debug` provides an

immediate a 3x speed improvement in either x32 or x64 modes.

Further Notes On Optimization: Since the library makes extensive use of the .NET 4 "Parallel Task Threading" procedures, and these procedures are "Core Aware", any addition of Cores / Threads in the processor running the code will get an immediate speed increase in execution without any user intervention or re-compiling needed. As an example (see figure below), I ran the same DFT code with various DFT sizes on three different Intel I7 processors from 15 to 45 Watts and from 2 Cores to 6 Cores. As can be seen, from the slowest processor to the fastest processor is a calculation speed increase of over 4x with no changes in the compiled code at all.

dftspeed

Speed comparison of three different processors running the same executable. The DFT size was changed from 2000 to 60000 points (X-Axis) and the calculation time in MilliSeconds was recorded (Y-Axis). The Blue trace is a 2 Core / 4 Thread processor, the Orange trace is a 4 Core / 8 Thread processor and the Yellow trace is 6 Core / 12 Thread processor. The Speed Increase happens automatically with no re-compilation or user intervention required.

References

- [1] G. Heinzel, A. Rudiger and R. Schilling, “Spectrum and spectral density estimation by the Discrete Fourier transform (DFT), including a comprehensive list of window functions and some new flat-top windows.”, Max-Planck-Institut fur Gravitationsphysik, February 15, 2002
- [2] Steve Hageman, "The practicing instrumentation engineer's guide to the Fourier Transforms", Published on EDN.com, June 2012.
    - Part 1: Understand DFT and FFT implementations
    - Part 2: Spectral leakage and windowing
    - Part 3: Other window types, averaging DFTs & more
    - Measuring small signals accurately: A practical guide
- [3] The Core FFT used in DSPLib was derived from a blog article written in 2010 by Gerald T. Beauregard. It is used here under the provisions of the MIT license that it was published under.
    - https://gerrybeauregard.wordpress.com/2011/04/01/an-fft-in-c/

Appendix A – Included Windows

Below are the tabulated performance parameters of the windows implemented in `DSPLib`.

For more information on these windows, see Reference [1].

Name	Peak Sideband Level (dB)	Flatness (dB)	3dB BW (bins)	NENBW (bins)
Rectangular or None	13.3	3.9224	0.8845	1
Welch	21.3	2.2248	1.1535	1.2
Bartlett	26.5	1.8242	1.2736	1.3333
Hanning or Hann	31.5	1.4236	1.4382	1.5
Hamming	42.7	1.7514	1.3008	1.3628
Nuttall3	46.7	0.863	1.8496	1.9444
Nuttall4	60.9	0.6184	2.1884	2.31
Nuttall3A	64.2	1.0453	1.6828	1.7721
Nuttall3B	71.5	1.1352	1.6162	1.7037
Nuttall4A	82.6	0.7321	2.0123	2.1253
BH92	92	0.8256	1.8962	2.0044
Nuttall4B	93.3	0.8118	1.9122	2.0212

Table1 - Normal Windows implemented in DSPLib

Name	Peak Sideband Level (dB)	Flatness (dB)	3dB BW (bins)	NENBW (bins)
SFT3F	31.7	0.0082	3.1502	3.1681
SFT3M	44.2	0.0115	2.9183	2.9452
FTNI	44.4	0.0169	2.9355	2.9656
SFT4F	44.7	0.0041	3.7618	3.797
SFT5F	57.3	0.0025	4.291	4.3412
SFT4M	66.5	0.0067	3.3451	3.3868
FTHP	70.4	0.0096	3.3846	3.4279
HFT70	70.4	0.0065	3.372	3.4129
FTSRS	76.6	0.0156	3.7274	3.7702
SFT5M	89.9	0.0039	3.834	3.8852
HFT90D	90.2	0.0039	3.832	3.8832
HFT95	95	0.0044	3.759	3.8112

Name	Peak Sideband Level (dB)	Flatness (dB)	3dB BW (bins)	NENBW (bins)
HFT116D	116.8	0.0028	4.1579	4.2186
HFT144D	144.1	0.0021	4.4697	4.5386
HFT169D	169.5	0.0017	4.7588	4.8347
HFT196D	196.2	0.0013	5.0308	5.1134
HFT223D	223	0.0011	5.3	5.3888
HFT248D	248.4	0.0009	5.5567	5.6512

Table 2 - Flat Top Windows implemented in DSPLib

The windows are referenced to an `ENUM` with the same name suffix. For instance, to call a window of '`None`', the `ENUM` would be called as `DSPLib.DSP.Window.Type.None`.

A window of type HFT248D would be called as `DSPLib.DSP.Window.Type.HFT248D`.

C# has a number of methods for converting `ENUM`s to `string`s and visa versa. Consult the C# online help for more information.

***Note**: Tabulated values in Table 1 and 2 are from reference [1].*

History

- 18th June, 2016
  - Initial release
- 16th July, 2016
  - Added more examples
  - Added '`PhaseUnwrap`' methods
  - Fixed DFT phase sign
  - Fixed FFT with multiple zero padded executions
  - See the *DSPLib.cs* file header for complete list of changes
  - Download files updated.
- 15th August, 2016
  - Added notes on Optimization
  - Added Example 11
  - Test Solution download updated
- 4th July, 2017
  - Added <= zero check in Log calculations to trap possible exception

    - V1.03 of C# source
- 15th October, 2017
    - Slight improvement to the way in which V1.03 worked
    - Same results, different design pattern
- 9th February, 2018
    - Sync'd Example Test Download Project Solution with Latest Library Version
    - Updated Test project to VS2017
    - No underlying library or operational changes
- 19th September, 2020
    - Added further notes on Optimizations
- 11th July, 2023
    - Added Complex[] data input type FFT and DFT (See Example 12)
    - Fixed issue with the returned length of the FT's to be in-line with academic norms. This could be a breaking issue when compared to previous version of this library as the number of returned spectrum points may be different than in Version 1.x
    - Updated project to .NET Framework 4.8
    - Updated project to VS2022

License