Background: Go over all background sections before going to your actual assigned lab session.

Verification: When you have completed a step and want verification, simply demonstrate the step to the instructor.

Lab Report: It is only necessary to turn in a report on Sections that require you to make graphs and to give short explanations. You are asked to label the axes of your plots and include a title for every plot. If you are unsure about what is expected, ask the instructor.

1 Background

This Lab is about frequency analysis in Matlab.

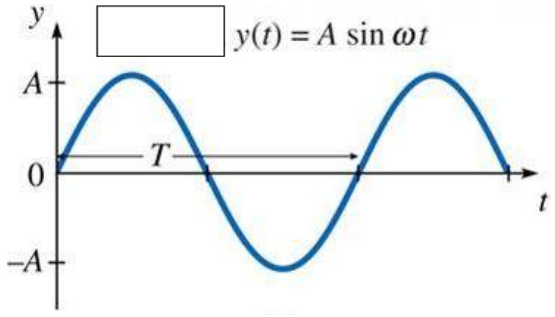
Goal

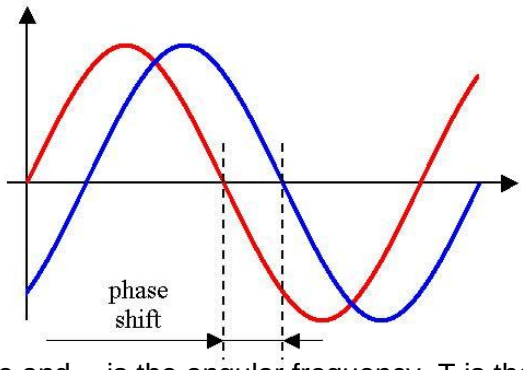
The goal of the laboratory project is to introduce the Fast Fourier Transform (FFT) algorithm for efficient computer calculation of the Fourier transform and to investigate some of the Fourier Transform's properties. The specific goals are:

1. Understanding discrete sampling and the Nyquist frequency.
2. Understanding Discrete Fourier transform.
3. Performing FFT and visualization of the result.

Pure Tones or Sine waves.

A sine wave or pure tone has (I) infinite duration, but only one frequency, (II) is periodic and has a phase and (III) is known as the "harmonic function". A sine wave can be represented by a sine function: $y(t) = A \cdot \sin(2\pi f \cdot t + \phi)$ or $y(t) = A \cdot \sin(\omega \cdot t + \phi)$



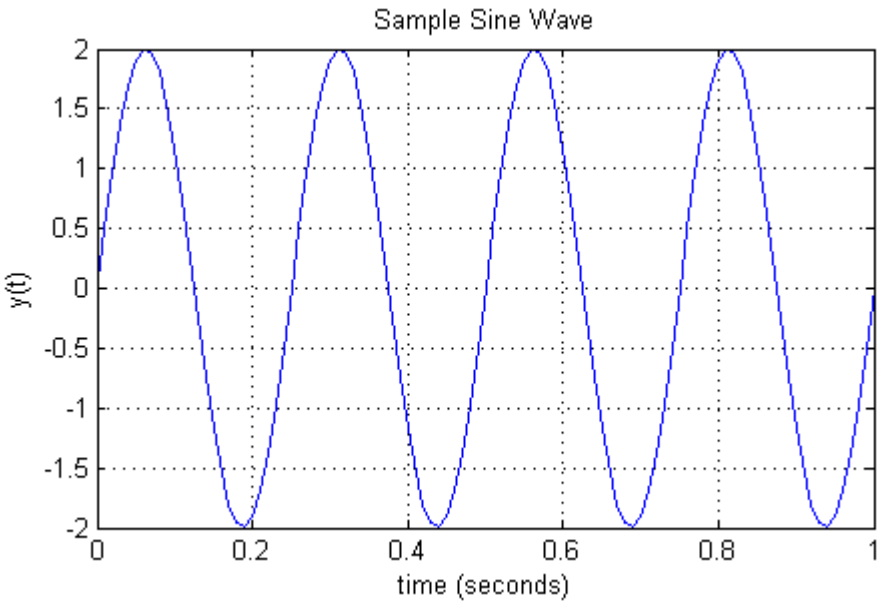


A represents the Amplitude, *f* frequency, ϕ phase and ω is the angular frequency. T is the period of the sine wave, were *f* = 1 / T.

A simple Matlab implementation of example of a pure tone:

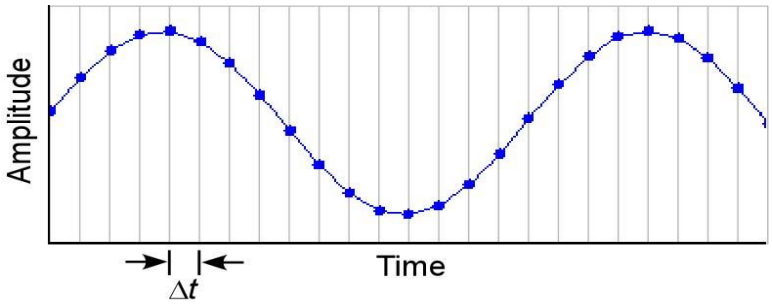
```
fo = 4;   %frequency of the sine wave
Fs = 100; %sampling rate
Ts = 1/Fs; %sampling time interval
t = 0:Ts:1-Ts; %sampling period
n = length(t); %number of samples
y = 2*sin(2*pi*fo*t); %the sine curve

%plot the cosine curve in the time domain
sinePlot = figure;
plot(t,y)
xlabel('time (seconds)')
ylabel('y(t)')
title('Sample Sine Wave')
grid
```

Here's what we get:

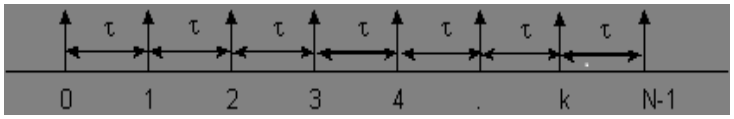


Discrete Sampling and Digital Representation of Sound (Nyquist, Aliasing).

This section provides a brief explanation of how sound is represented digitally. An understanding of the basic principles introduced here will be helpful in creating sound signals in Matlab. Before a continuous, time-varying signal such as sound can be manipulated or analysed with a digital computer, the signal must be *acquired* or *digitized* by a hardware device called an analogue-to-digital (A/D) converter, or *digitizer*. The digitizer repeatedly measures or *samples* the instantaneous voltage amplitude of a continuously varying (analogue) input signal at a particular sampling rate, typically thousands or tens of thousands of times per second (see figure below). In the case of an audio signal, this time-varying voltage is proportional to the sound pressure at a device such as a microphone. The digital representation of a signal created by the digitizer thus consists of a sequence of numeric values representing the amplitude of the original waveform at discrete, evenly spaced points in time.



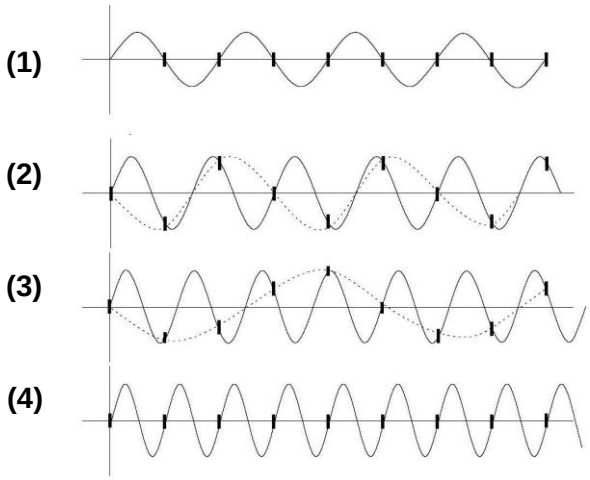
Sampling to create digital representation of a pure tone signal. The blue sinusoidal curve represents the continuous analog waveform being sampled. Measurements of the instantaneous amplitude of the signal are taken at a sampling rate of 1/Δt. The resulting sequence of amplitude values is the digitized signal.



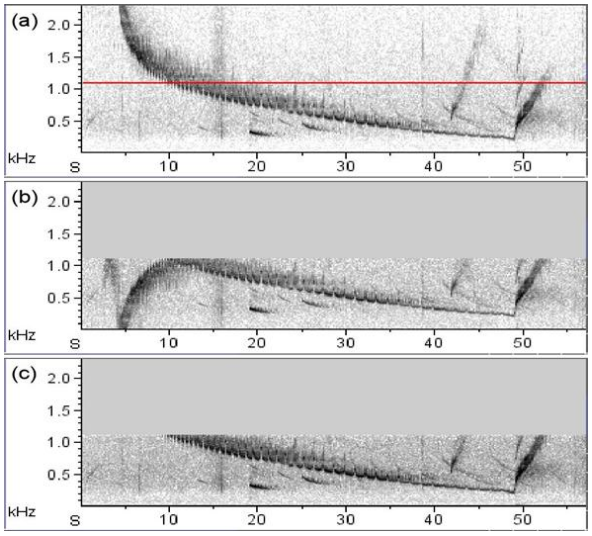
Sampling time Δt or **τ**. Time between each sample. It is also the inverse of the sampling frequency. If sampling frequency, Fs = 20 samples/sec, then t = 1/Fs = .05 sec Total Number of Samples: N. The total number of samples collected or available = N. Sequence Time Length: T, the time length of the collected sequence and is equal to the product of the sample time and the total number of samples = N Δt Sample index: k. The sequence time is no longer continuous so instead of Δt, we use a discrete time measure called k^{th} sample. This is an index of the samples. Its range extends from 0 to N-1, where N is the last sample. Each k^{th} sample of total N samples, is located at time k times Δt .

The precision with which the digitized signal represents the continuous signal depends on two parameters of the digitizing process: the rate at which amplitude measurements are made (the *sampling rate* or *sampling frequency, Fs*), and the number of bits used to represent each amplitude measurement (the *sample size* N or *bit depth*). The more frequently a signal is sampled, the more precisely the digitized signal represents temporal changes in the amplitude of the original signal. The sampling rate that is required to make an acceptable representation of a waveform depends on the signal's frequency. More specifically, the sampling rate must be more than twice as high as the highest frequency contained in the signal. Otherwise, the digitized signal will have frequencies represented in it that were not actually present in the original at all. This appearance of phantom frequencies as an artefact of inadequate sampling rate is called *aliasing.*

The highest frequency that can be represented in a digitized signal without aliasing is called the *Nyquist frequency*, and is equal to half the frequency at which the signal was digitized. If the only energy above the Nyquist frequency in the analogue signal is in the form of low-level, broadband noise, the effect of aliasing is to increase the noise in the digitized signal. However, if the spectrum of the analogue signal contains any peaks above the Nyquist frequency, the spectrum of the digitized signal will contain spurious peaks below the Nyquist frequency as a result of aliasing.



Upper trace: A sine wave is sampled near or at the Nyquist frequency. In the example given above, 4 complete sine waves are shown but 8 samples are taken. High-frequency components of the data above the Nyquist frequency will be lost. Traces 2, 3 and 4 show how sampling looks like when occurring below the Nyquist frequency. The dotted lines show the effect of under sampling. Can you explain why we can't see a dotted line in trace 4?

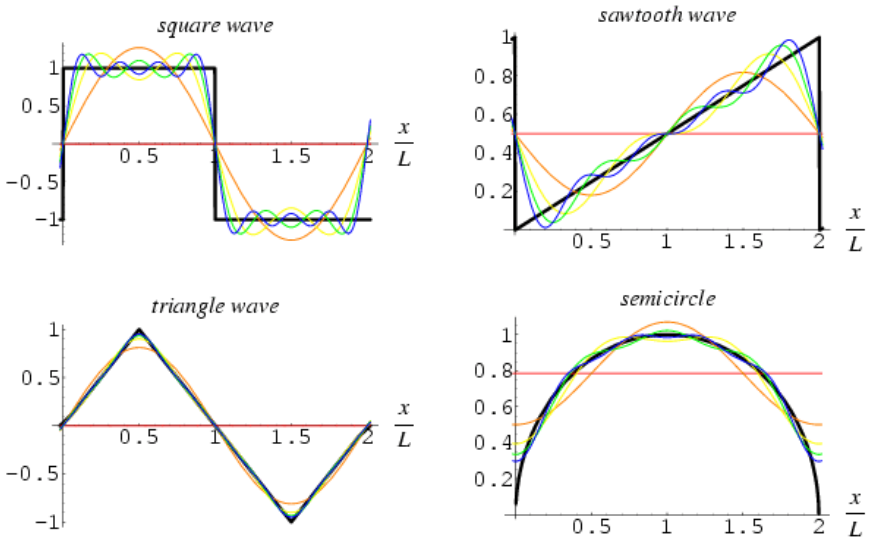


Aliasing in ***spectrograms***. **(a)** Spectrogram of a bearded seal song signal digitized at 11025 Hz. All of the energy in the signal is below the Nyquist frequency (5512.5 Hz); only the lowest 2300 Hz is shown. The red line is at 1103 Hz, one-fifth of the Nyquist frequency. **(b)** The same signal sampled at 2205 Hz (one-fifth of the original rate; Nyquist frequency, 1102.5 Hz) without an anti-aliasing filter. The frequency down sweep in the first ten seconds of the original signal appears in inverted form in this under sampled signal, due to aliasing. **(c)** Same signal as in (b), but now passed through a low-pass (anti-aliasing) filter with a cutoff of 1100 Hz before being digitized. The down sweep in the first 10 sec of the original signal, which exceeds the Nyquist frequency, does not appear because it was blocked by the filter.

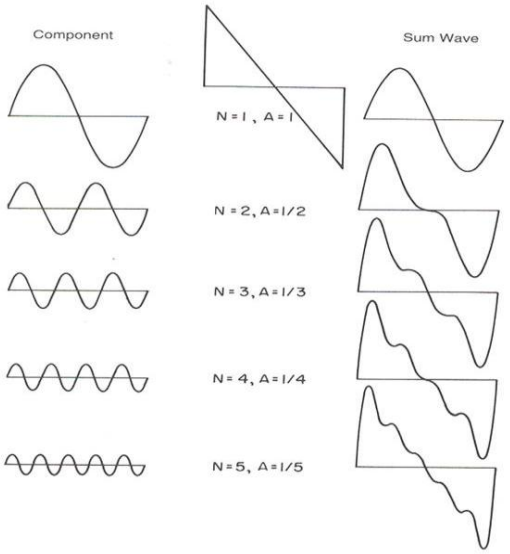
In spectrograms, aliasing is recognizable by the appearance of one or more inverted replicates of the real signal, offset in frequency from the original. Thus, when an analogous signal is sampled at a fixed rate, there is a maximum frequency called the Nyquist frequency (Fn) at which sampling begins to lose information. This critical frequency is two sample points per cycle Fn = 1 / (2 Fs) of the highest frequency in the original sound signal.

Fourier series and Synthesis (Why use $e^{(j n \omega_o t)}$?).

Up to now we only considered pure tones. Nonetheless, real sounds are broadband and thus contain many pure tones. A Fourier series is an expansion of a harmonic function in terms of an infinite sum of sines and cosines. Fourier series make use of the orthogonal relationships of the sine and cosine functions. The computation and study of Fourier series is known as harmonic analysis and is extremely useful as a way to break up an *arbitrary* periodic function or sound into a set of pure tones. Examples of successive approximations to common functions using Fourier series are illustrated below.



In music, if a note has an fundamental frequency, f, integer multiples of that frequency, $2f, 3f, 4f$ and so on, are known as *harmonics*. As a result, the mathematical study of overlapping waves is called harmonic analysis. An example of a sawtooth wave Fourier series is given below.



The fundamental frequency (or sine wave in this case), N=1, of the sawtooth wave is given by the component (or harmonic) with the lowest frequency. Al other components have higher frequencies and are called higher harmonics of the sawtooth wave, which have higher frequencies. That is, component N=4 has a frequency that is four times higher than that of component N=1. Also notice that a similar relationship occurs for the amplitude of each component. In this case, the fundamental component, N=1, has the largest amplitude, A=I, and e.g., component N=4, has an amplitude that is four times smaller, A=I/4. Finally, a similar relationship occurs for the phase of each component. In this case, however, the phase is zero for all components.

Thus, any sound can be "synthesized" by adding its harmonics (i.e., "pure tones") together with the proper amplitudes and phases. This can be formalized as follows:

$$x(t) = A_0 + \sum_{n=1}^{\infty} A_n \cos(\omega_n t + \phi_n)$$

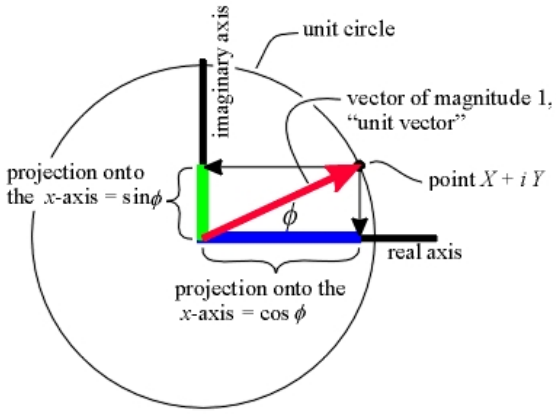
However, you should be aware that there are other ways of formalization, and that you might get more information or insight using those other forms. Actually, you are forced to deal with this other representation because this is the representation in which results are returned when using FFT algorithms in applications like Matlab.

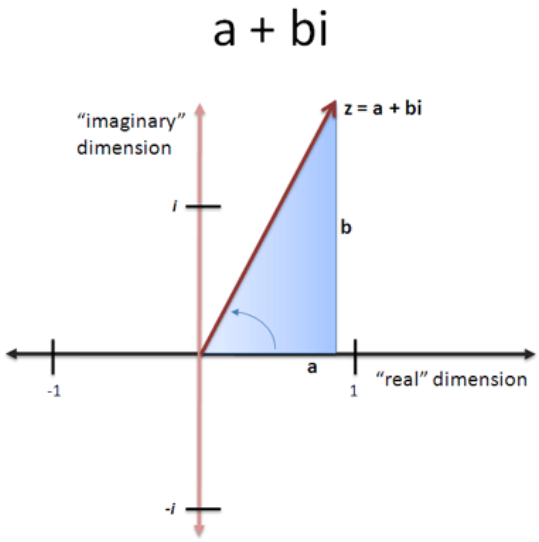
In a Fourier Series used in FFT there are two terms in the **n**th harmonic - a cosine term and a sine term. Together they give you the components of the signal at that frequency, i.e. the **n**th harmonic. Writing them out gives:

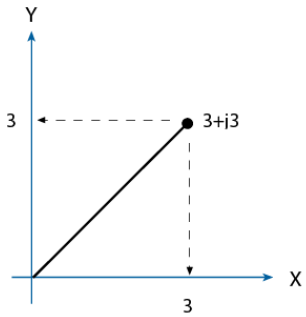
$$a_n \cos(n\, \omega_o t) + b_n \sin(n\, \omega_o t)$$

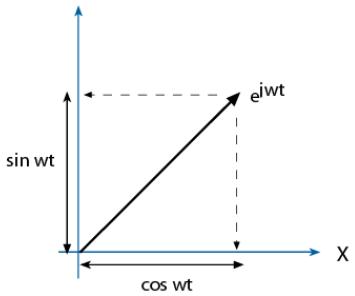
representing the total component at the n^{th} harmonic of the wave form.

Consider the following diagrams:









Here, we have:

$$a_n = c_n \cos(\phi_n)$$

$$b_n = c_n \sin(\phi_n) \text{ and } c_n \bullet c_n = a_n + b_n$$

Using these relations, we can write the component at the **n**th harmonic.

Total component at the n^{th} harmonic

$= a_n \cos(n \phi_o t) + b_n \sin(n \phi_o t)$

$= [c_n \cos(\phi_n)\cos(n \omega_o t)] + [c_n \sin(\phi_n) \sin(n \phi_o t)]$

$= c_n [\cos(\phi_n) \cos(n \omega_o t) + \sin(\phi_n) \sin(n \omega_o t)]$

$= c_n \cos(n \omega_o t + \phi_n)$ where $\omega_o = 2\pi / T$

Notice the resemblance with the equation at the top of this page. Now, we can interpret this result as follows:

(1) c_n is actually the size of the component at the **n**th harmonic, when using FFT. Also notice that c_n is the complex modulus or magnitude. In Matlab the magnitude of a complex number can be computed by a function called `abs`.
(2) You can get a_n and b_n from c_n if you know the phase angle ϕ_n of the harmonic.
(3) Different phase angles will produce different a's and b's, but the size of the harmonic would stay the same.
(4) It is important to obtain at least one period of the wave form when performing Fourier transformation using FFT

Also notice that It is possible to compute the a's and b's by approximating the following integrals.

$$a_k = (2/T) \int_0^T f(t)\cos(2\pi kt/T)\, dt \qquad b_k = (2/T) \int_0^T f(t)\sin(2\pi kt/T)\, dt$$

However, that isn't necessarily the way it is done. In most cases a different representation is used. Consider the following:

$$(2/T)\int_0^T f(t)e^{n2\pi t/T}\,dt = (2/T)\int_0^T f(t)\cos(n2\pi t/T)\,dt + (2j/T)\int_0^T f(t)\sin(n2\pi t/T)\,dt$$

In the integral above, the function, f(t), is multiplied by a complex exponential. However, the complex exponential can be represented as a complex sum of the cosine and the sine.

$$e^{(j\,n\,\omega_o\,t)} = \cos(n\,\omega_o t) + j\sin(n\,\omega_o t) \text{ and } \omega_o = 2\pi / T.$$

Using that representation gives the two integrals above. (And, note the presence of "j" multiplying the sine in the integral above, and in the expression for the complex exponential. That is what makes it complex.) Notice also the resemblances with the equations given earlier (previous page).

Now, the neat result from this is that you can do one integral and get both the a's and b's simultaneously. In the computer implemented FFT algorithm that is what happens.

Notice also, the Fourier coefficients a_n and b_n of the complex notation are multiplied by 2 / T. We will see that something similar happens with the FFT in Matlab. This is important, because this factor is needed to scale amplitude so that it gives the correct values.

Basic properties of Discrete Time Fourier Transform (DTFT).

By applying the Fourier Transform algorithm on these N samples, we have made an implicit assumption. We have assumed that the signal is periodic over N samples, so we have assumed that the fundamental frequency of our signal is equal to the inverse of time T of the N samples. We express the fundamental frequency as

$$f_0 = \frac{1}{T}$$

We can rewrite T as a function of sample time, τ and total number of samples chosen, N to alternately express the fundamental frequency in terms of f_s and N.

$$f_0 = \frac{1}{\tau N} = \frac{1}{\frac{\sec s}{sample} x\, Total\ no.\ of\ sample}$$

$$f_0 = \frac{f_s}{N}$$

This is a very important concept to understand. It says that you have artificially set the fundamental frequency to the sampling frequency divided by the total number of samples observed. It is a strange idea seemingly having nothing to do with the target signal and in fact this is true. This frequency referred to as the *fundamental frequency* of the signal really is kind of a resolution frequency and has nothing to do with the target signal. It just means that we resolve the target signal components in integer multiples of this resolution frequency.

Let's say that we sampled the above signal at sample time of 1/20th sec and observed 60 samples. Then the fundamental frequency is

$$f_0 = \frac{f_s}{N} = \frac{20}{60} = .333\,Hz$$

Now when we compute the Fourier Transform we will be stepping this fundamental frequency by integer multiples. With f_0 = .333, the next harmonic would be f_1 = .666 and so on. The harmonics used in the analysis are not integers but *integer multiple* of the fundamental frequency of the signal as determined by the sampling frequency and the N samples observed. An alternate way to see these harmonics is see them as bins which collect energy. In DFT they are also called cells.

Now the n_{th} harmonic can be expressed as n times f_0.

$$f_n = nf_0$$

this is also equal to

$$f_n = \frac{n}{\tau N} = \frac{n f_s}{N}$$

DFT is a special case of the Fourier Transform and is actually an approximation of the real thing. The validity of the approximation is effected by the type of waveform we are dealing with as well as the parameters f_s and N.

Computing the DFT

1. What is the sampling frequency of the target signal? Is the sampling frequency large enough so that it covers all significant frequencies in the signal?

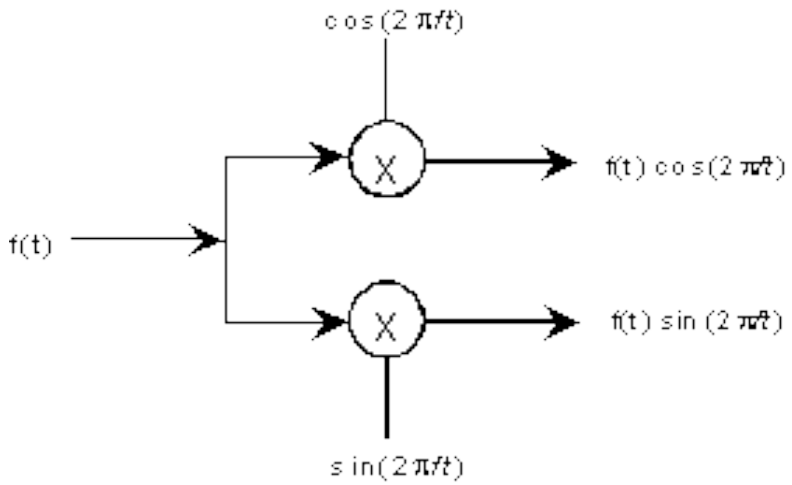
2. How many samples do we need?

First compute the fundamental frequency, and starting with the fundamental frequency we multiply the discrete signal by a complex exponential and perform summation on the result.

Do you recall what it means to multiply by a complex exponential? How do you interpret the following equation?

$$e^{(j_n \omega_o t)} = \cos(n \omega_o t) + j \sin(n \omega_o t)$$

The figure below shows what is happening in real-life.



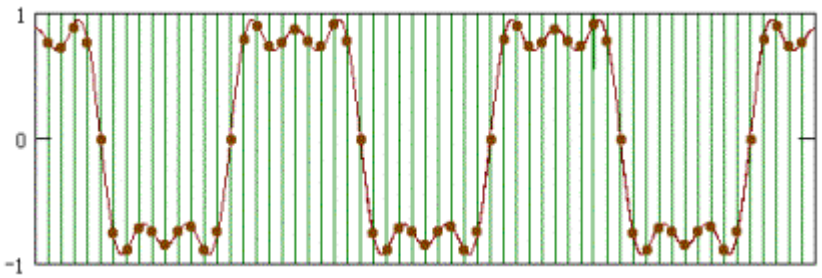
What we are really doing when we multiply by a complex exponential

The signal in fact is being split into two parts, 1. multiplied by a sine wave and the other by a cosine of the same frequency. The resulting two signals are orthogonal and are the result of multiplication with the complex exponential or phasor.

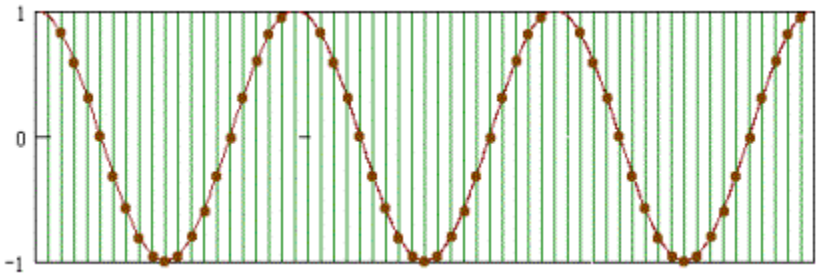
DFT Step by step

Now we compute the DFT of the signal in Fig 1.

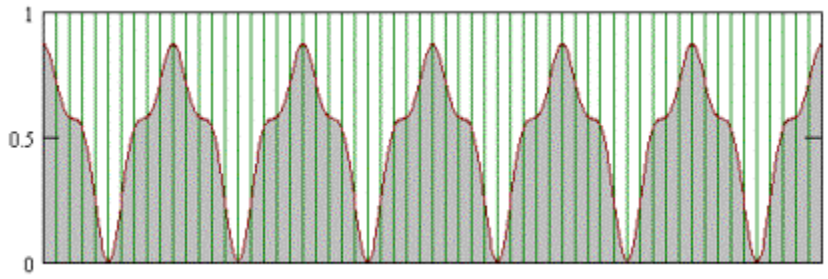
Step 1 - Multiply the target signal in by a cosine wave in (First Harmonic) of frequency f_0. For this demonstration, we assume that $f_0 = 1$. (Although only cosines are shown, we do this for both sines and cosines and keep track of the results separately.)



The sampled target signal



First Harmonic $f_1 = 1 * f_0$

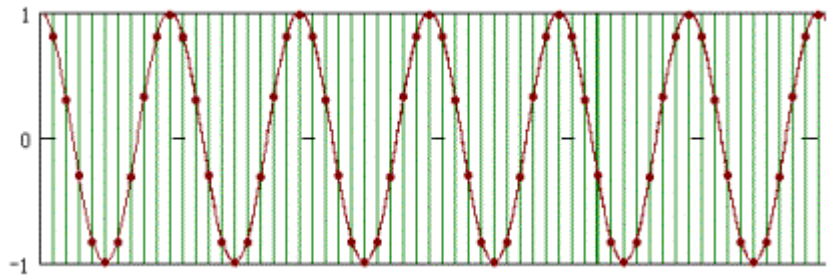


Result of multiplying the first harmonic with the target signal. The waveform has positive area.

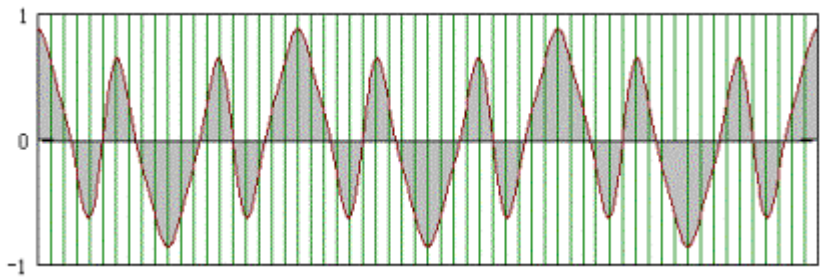
The multiplication gives us the waveform in 7cNow integrate this waveform over the N samples. In a discrete case, we integrate by multiplying the sample amplitude by the width of the base which is equal to t, the sample time, using the trapezoidal rule. We are in effect adding up the areas of all the small gray rectangles. Each sample value is multiplied by t and these areas are summed.

The result of the multiplication tells us something interesting. We see that the resulting waveform is not even, so it has net area under it. This means that there is a signal hiding in this frequency. What is the amplitude of this frequency? That we know only when we complete the summation. The result of the summation gives us the amplitude of this harmonic in the target signal.

Step 2: Now multiply the original Signal with the second.

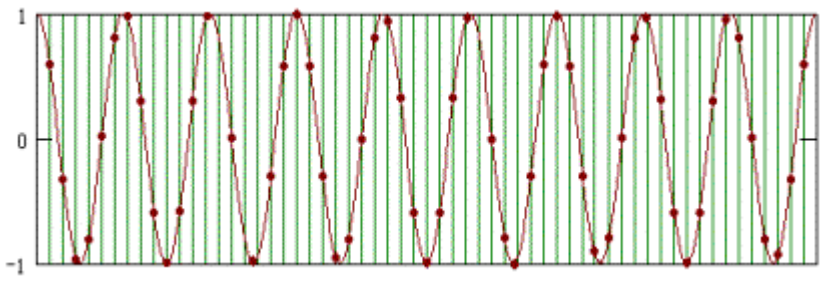


2nd Harmonic $f_2 = 2 * f_0$ The multiplication gives the waveform:



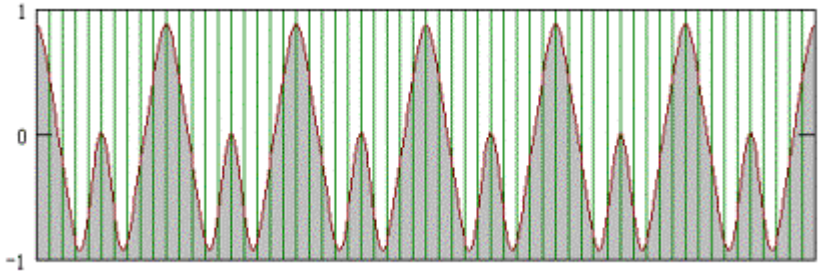
Result of multiplying the 2nd harmonic with the target signal.

The waveform is even, which means that the summation of the little gray rectangles will give zero area. Since it has no net area means there is nothing of interest here. Let's go to the next harmonic. Now multiply the target signal with the 3rd harmonic.



3nd Harmonic $f_2 = 3 * f_0$

The resulting waveform is shown below.



Result of multiplying the 3rd harmonic with the target signal. The waveform has positive area.

Once again, we get a waveform that has a DC component, so we will do the summation to figure out the amplitude. We will stop here because the original signal only has three frequencies. And we have found them all.

Fast Fourier Transform (FFT)

The DFT although clear and easy to compute requires a good many calculations. For each harmonic, we have N+1 multiplications and we do this N times, giving us N^2 + N calculations. A 256 point DFT would require 65792 calculations. The algorithm for Fourier Transform existed for more than 200 years before it came into widespread use mostly because we could not cope with such large number of computations. The algorithm waited for the development of the microprocessor.

In 1948, Cooley and Tukey and came up with a computational breakthrough called the Fast Fourier Transform algorithm. This was an ingenious manipulation of the inherent symmetry of the calculations. It now allowed the computation of a N point DFT as a function only 2N instead of the N^2. So a 256 point DFT would only require 512 calculations, a huge improvement from 65792 calculations doing it the laborious way. The algorithm was quickly and widely adopted and is the basis of all modern signal processing.

Most DSP books spend a lot of time going through the mechanism of the FFT and how it is computed. Huge butterfly figures in our books help to confuse and confound us as to what is actually going on. Most of us just give up on it. Let me alleviate your guilt. Although extremely important in itself, the understanding of the mechanism of the FFT is not all that important. So, I am skipping over the details of its implementation.

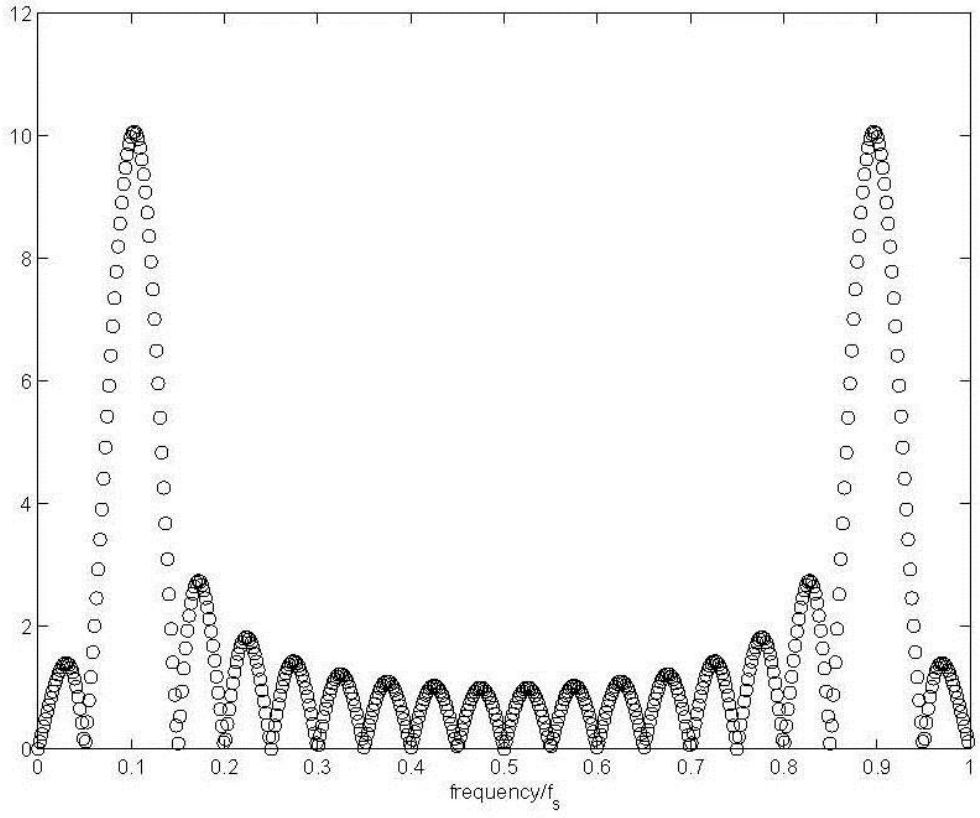
The method has been programmed in all sort of software and we can safely skip it without impacting our understanding of the ***application*** of the DFT and the FFT.

The main thing one needs to know about the FFT is that it works only with sample numbers that are powers of 2, such as 16, 32, 64 etc.. We cannot do a FFT on an arbitrary number of samples as we can with DFT which is a generic process. The FFT is a DFT with constraints on the number of samples.

The other thing about the FFT process to know is that it allows zero-padding. Let's say we have 28 samples and we wish to do a DFT via the FFT, we can do two things, 1. we can do a 16 point FFT and discard the remaining 12 points or we can insert four zeros at the end so we have 32 points. Now we can do a 32 point FFT. The zero-padding provides us better resolution but does not provide any extra information. The frequency detected is still a function of the original N samples and not the zero-padded length, although the FFT does *look* a lot better. And looks count.

Periodicity: The DTFT is periodic. One period extends from f = 0 to f_s, where f_s is the sampling frequency. Taking advantage of this redundancy, the DFT is only defined in the region between 0 and f_s.

Symmetry: When the region between 0 and f_s is examined, it can be seen that there is even symmetry around the center point, 0.5 f_s, and the Nyquist frequency. This symmetry adds redundant information. The Figure below shows the DFT (implemented with Matlab's FFT function) of a cosine with a frequency one tenth the sampling frequency, fs. Notice that the data between 0.5 f_s and f_s is a mirror image of the data between 0 and 0.5 f_s.



How MATLAB transforms discrete data

Here's MATLAB's definition for the Fast Fourier Transfrom (FFT). You can get this by typing `help fft`

```
 ---------------------------------------------
  FFT Discrete Fourier transform.
     FFT(X) is the discrete Fourier transform (DFT) of vector X.  If the
     length of X is a power of two, a fast radix-2 fast-Fourier
     transform algorithm is used.  If the length of X is not a
     power of two, a slower non-power-of-two algorithm is employed.
     For matrices, the FFT operation is applied to each column.
     For N-D arrays, the FFT operation operates on the first
     non-singleton dimension.

     FFT(X,N) is the N-point FFT, padded with zeros if X has less
     than N points and truncated if it has more.

     FFT(X,[],DIM) or FFT(X,N,DIM) applies the FFT operation across the
     dimension DIM.

     For length N input vector x, the DFT is a length N vector X,
     with elements
                      N
        X(k) =       sum  x(n)*exp(-j*2*pi*(k-1)*(n-1)/N), 1 <= k <= N.
                     n=1
     The inverse DFT (computed by IFFT) is given by
                      N
        x(n) = (1/N) sum  X(k)*exp( j*2*pi*(k-1)*(n-1)/N), 1 <= n <= N.
                     k=1

     The relationship between the DFT and the Fourier coefficients a and b in

                 N/2
   x(n) = a0 + sum a(k)*cos(2*pi*k*t(n)/(N*dt))+b(k)*sin(2*pi*k*t(n)/(N*dt))
                 k=1
   is
        a0 = X(1)/N, a(k) = 2*real(X(k+1))/N, b(k) = -2*imag(X(k+1))/N,
   where x is a length N discrete signal sampled at times t with spacing dt.

   See also IFFT, FFT2, IFFT2, FFTSHIFT.
```

Notice that here Matlab uses the notation j (not i) to denote the complex part of a complex number. Next notice that Matlab uses a numbering system from 1 to N (not zero to N). But overall, we can make some sense of the information given by the Matlab's help function because of the explanations given on the previous pages.

Lab 02: The Fast Fourier Transform

As mentioned earlier, Matlab's FFT function is an effective tool for computing the discrete Fourier transform of a signal.The following code examples will help you to understand the details of using the FFT function.

Example 1: The typical syntax for computing the FFT of a signal is FFT(x,N) where x is the signal, x[n], you wish to transform, and N is the number of points in the FFT. N must be at least as large as the number of samples in x[n]. To demonstrate the effect of changing the value of N, sythesize a cosine with 30 samples at 10 samples per period.

```
n = [0:29];
x = cos(2*pi*n/10);
```

Define 3 different values for N. Then take the transform of x[n] for each of the 3 values that were defined. The `abs` function finds the magnitude of the transform, as we are not concerned with distinguishing between real and imaginary components.

```
N1 = 64;
N2 = 128;
N3 = 256;

X1 = abs(fft(x,N1));
X2 = abs(fft(x,N2));
X3 = abs(fft(x,N3));
```

The frequency scale begins at 0 and extends to N ¡ 1 for an N-point FFT. We then normalize the scale so that it extends from 0 to 1 - 1/N.

```
F1 = [0 : N1 - 1]/N1;
F2 = [0 : N2 - 1]/N2;
F3 = [0 : N3 - 1]/N3;
```

Plot each of the transforms one above the other.

```
figure(1)
subplot(3,1,1)
plot(F1,X1,'-x'),title('N =  64'),axis([0 1 0 20])
subplot(3,1,2)
plot(F2,X2,'-x'),title('N = 128'),axis([0 1 0 20])
subplot(3,1,3)
plot(F3,X3,'-x'),title('N = 256'),axis([0 1 0 20])
```

Upon examining the plot one can see that each of the transforms adheres to the same shape, differing only in the number of samples used to approximate that shape. What happens if N is the same as the number of samples in x[n]? To find out, set N1 = 30. What does the resulting plot look like? Why does it look like this?

Example 2: In the last example the length of x[n] was limited to 3 periods in length.
Now, let's choose a large value for N (for a transform with many points), and vary the number of repetitions of the fundamental period.

```
n = [0:29];
x1 = cos(2*pi*n/10);          % 3 periods
x2 = [x1 x1];                 % 6 periods
x3 = [x1 x1 x1];              % 9 periods

N = 2048;

X1 = abs(fft(x1,N));
X2 = abs(fft(x2,N));
X3 = abs(fft(x3,N));

F = [0:N-1]/N;

Figure(2)
subplot(3,1,1)
plot(F,X1),title('3 periods'),axis([0 1 0 50])
subplot(3,1,2)
plot(F,X2),title('6 periods'),axis([0 1 0 50])
subplot(3,1,3)
plot(F,X3),title('9 periods'),axis([0 1 0 50])
```

The previous code will produce three plots. The first plot, the transform of 3 periods of a cosine, looks like the magnitude of 2 since with the center of the first since at 0.1 fs and the second at 0.9 fs. The second plot also has a sinc-like appearance, but it has a larger magnitude at 0.1 fs and 0.9 fs. Similarly, the third plot has a larger magnitude compared to the previous two plots. As x[n] is extended to an large number of periods, the sincs will begin to look more and more like impulses.

But sinusoid should transform to an impulse in the frequency domain, why do we have sincs in the frequency domain?

When the FFT is computed with an N larger than the number of samples in x[n], it fills in the samples after x[n] with zeros.

Example 2 had an x[n] that was 30 samples long, but the FFT had an N = 2048. When Matlab computes the FFT, it automatically fills the spaces from n = 30 to n = 2047 with zeros. This is like taking a sine wave and multiplying it with a rectangular box of length 30.

A multiplication of a box and a sine wave in the time domain should result in the convolution of a sinc with impulses in the frequency domain. Furthermore, increasing the width of the box in the time domain should increase the frequency of the sinc in the frequency domain. The previous Matlab experiment supports this conclusion.

2 Spectrum Analysis with the FFT and Matlab

It should be clear that FFT does not directly give you the spectrum of a signal. As we have seen, the outcome of FFT can vary dramatically depending on the number of points (N) assigned to the FFT, and the number of periods of the signal that are represented. There is another problem as well.

The FFT contains information between 0 and Fs, however, we know that the sampling frequency, must be at least twice the highest frequency component. Therefore, the signal's spectrum should be entirely below Fs / 2, the Nyquist frequency.

Recall also that when the region between 0 and F_s is examined, there is even symmetry around the center point, 0.5 F_s, and the Nyquist frequency. This symmetry adds redundant information.

Finally, recall that the Fourier coefficients a_n and b_n of the complex notation are multiplied by 2 / N, as indicated by the Matlab help on FFT.

The exercises below involve performing FFT of "complex" wave forms and visualization of the result in a correct fashion.

Getting comfortable with sinusoids and FFT and effects of noise

Include a short summary of this Section with plots in your Lab report. Write a MATLAB script file to do steps (a) through (i) below. Include a listing of the script file with your report.

(a) Use the following equation to compute the wave y from time line t.

$$y_n = \sin (2\pi f_n \ t + \phi_n)$$

Graphically display two sine waves (i.e., "pure tones") with frequencies, f1 and f2 [in Hz] , phases phi1 and phi2, and their sum; with timeline t running from `T1` up to `T2 [in msec]` in three separate subplots. Use a sampling frequency, `Fs [in Hz]`. The sine waves y1 and y2 should differ in their Frequency, Amplitude and Phase.
L denotes the length of the signal in sample number (use `length` to determine this).

It is your task to create a plot that shows amplitude as a function of frequency for the summed y1 + y2 signal by following the steps as described below.

(b) First show what happens if you perform the Fast Fourier Transform of the summed signal. Explain what you see, what is wrong with this plot? Is the sampling frequency, Fs, high enough, why is this important?

(c) How can you get rid of the redundant information?

(d) Now make sure that the frequency axis (x-axis) is correctly converted into Hz. How can you do this? Keep in mind that in this respect signal length and sampling frequency are important.

(e) Finally you have to make sure that the amplitude range (Y-axis) is correct. Remember that the help on FFT explained that the relationship between DTF and the Fourier coefficients a_n and b_n of the complex notation is obtained by multiplying them by 2 and dividing by N, the length of the signal (i.e., number of samples). So how would you propose to convert the transform of y1 + y2 to get the correct amplitude values?

(f) By now you should be able to produce both a time-domain plot and a frequency (i.e., a single sided amplitude spectrum) plot of the y1 + y2 signal with the correct ranges and labels.

Note that the amplitude spectrum is basically as we expect, with most of the signal amplitude concentrated at F1 and F2, and just a bit of leakage (sincs) at other frequencies; note also that the amplitudes at those two frequencies correspond to the amplitudes definied in our sine wave equations

(g) What happens if you use `semilogx` instead of the `plot` function of Matlab?

(h) If the signal length, L, becomes shorter than the period, T, of the summed signal y1 + y2, is it still possible to get a good Amplitude spectrum. If not, can you explain this.

(i) Finally, under normal circumstances, the signal will not contain only coherent information (i.e., the sine waves or harmonics) but noise as well. Show how you could introduce (or add noise) to the y1 + y2 signal in Matlab (Noise refers to the randomness of the values in a given measurement). What happens with the amplitude spectrum (show this in a plot).