Notes on Fourier transforms

The Fourier transform is something we all toss around like we understand it, but it is often discussed in an offhand way that leads to confusion for those just learning their way around DSP. I'm not ready to write a comprehensive manual on the thing (Smith devotes 14 chapters to it), but here are some assorted bits of trivia that may clear the air some.

Buzzwords, or fftspeak:

Time domain representation means a graph with time along the bottom. Waveforms are usually represented this way.

Frequency domain representation means a graph with frequency along the bottom. Spectral plots are like this.

Polar representation means graphing in terms of an angle and radius. Since sine waves are inherently angular this can be useful. To map the frequency domain onto a polar plot, the angle π radians represents 1/2 the sampling rate. The region on the bottom of the circle represents negative frequency plotted from 0 to $-\pi$. Points on a polar plot can also be indicated by their rectangular (Cartesian) coordinates.

The **unit** means one. The unit circle on a polar plot is a circle of radius 1. Setting things to equal 1 often makes math clearer.

The letter **j** is the imaginary square root of -1. Some mathematicians use i for this, but i means something else in electronics. We aren't interested in roots of negative numbers per se, but complex numbers are handy.

Complex numbers are the sum of a real and imaginary part such as (a + bj). The math works as if the imaginary part were at right angles to the real part, which is the way a lot of audio phenomena behave. Some authors indicate variables that represent complex numbers with a capital letter. Numbers whose imaginary parts are zero are called **real** numbers.

The letter ω (Greek lower case omega) is often used to refer to angles. You will see $\omega=2\pi f$ as a way to convert a frequency f to an angle. When derived this way, ω may be called angular frequency.

The letter **e** means Euler's constant, a number like π that is one of the fundamental features of the universe. It is used to calculate interest, and is the base of natural logarithms. The interesting item here is that you can represent a

sine wave as $e^{j\omega}$. The j makes this a complex number, the ω makes it polar notation. $e^{j\omega} = \cos(\omega) + j\sin(\omega)$.

A **function** is the mathematical representation of any sort of curve. A sine wave is a function that could be written f(t) = ksin(ωt). Functions are always functions of some thing, in this case t. Some texts make the distinction that f(x) is a continuous function (i.e. an unbroken curve) and f[x] is a discreet function, made up of a lot of points. (Like a sampled waveform.) f(x) is pronounced "f of x".

A **coefficient** is a value that adjusts the overall value of a term in a function. For ksin(ωt), k is a coefficient. Most of the hard part of DSP design is finding the coefficients that make a function work the way you want it to.

The **delta function** is a 1 followed by as many 0s as you want. It's the mathematical equivalent of hitting something with a stick to see what it will do. When you apply a delta function to digital filter, you get its **impulse response**. The Fourier transform of the impulse response is the frequency response.

A **transform** is a method for converting a function of time into a function of frequency (or back). In audio, it converts a chunk of waveform into a spectral representation.

You **multiply** two **functions** by multiplying the value of each point on one curve by the value of the equivalent point of the other curve.

Correlation of functions is performed by multiplying each point on one curve by all of the other curve. This gives one complete curve per point. You then add all of these together.

Convolution of functions is performed by multiplying each point on one curve by the reverse of the other curve (that's the other curve backwards) and adding all the results. Convolution of two time domain functions is equivalent to multiplying the frequency domain versions, and vice versa.

Transforms

There are several transforms out there - Laplace, Z-transform, and Fourier being the big names.

The **Laplace** transform converts a waveform into a series of exponentially changing sinusoids. This results in a whole family of curves of amplitude vs. frequency (one curve for each possible exponent) These are represented by parallel curves in a three dimensional space called the s domain. This is very

useful for designing analog filters, whose response is a combination of exponential shapes.

The **Z transform** converts waveforms into something similar to the s plane, but in a polar scheme known as the Z plane. The frequency is represented by the angle, and the exponent by the radius, so the amplitude vs. frequency curves are wrapped in circles. This is needed for designing digital filters.

The **Fourier transform** converts waveforms into a series of sinusoids. A sinusoid is a waveform shaped like a sine wave, but not necessarily starting at 0.0. The Fourier transform actually results in two curves, one that represents the amplitude of the sinusoids and another that shows the phases.

Types of Fourier transform

There are four types of signal encountered in audio. These signal types are:

- Non-periodic analog signal
- Periodic analog signal
- Non-periodic digitized (discrete) signal
- Periodic digitized (discrete) signal

Different variants of the Fourier transform are appropriate to each. Some of these are referred to with initials.

- In the case of the non-periodic analog signal, the sinusoids may have any frequency, and there may be an infinite number of them. The original Fourier transform deals with this.
- In the case of the periodic analog signal, the sinusoids take frequencies that are multiples of the fundamental established by the period (The **Fourier series**). There still may be an infinite number. (You need an infinite number of sinusoids to represent a corner in a waveform, so square waves and triangles have infinite Fourier representations.)
- In the case of the non-periodic digitized signal, the sinusoids are discrete themselves, and are limited in frequency to the range of one half of the sampling rate. The **D**iscrete **T**ime **F**ourier **T**ransform is used here.
- In the case of the periodic digitized signal, the sinusoids are discrete themselves, and limited in frequency to one half of the sampling rate. This can be analyzed by the **D**iscrete **F**ourier **T**ransform, or the **F**ast **F**ourier **T**ransform.

All of these transforms have inverse transforms, which take us back to the original waveform.

The transforms for analog signals are the theoretical basis for all of the math, but you can't actually do any of them in a computer. Computers deal with discrete signals only, because all they know is lists of numbers. These are processed by the DFT. The waveform goes in as a list of numbers, and two lists come out. What numbers do we need in the lists?

Details of the output

A sinusoid has frequency, amplitude and phase. These three parameters must be included for each of the components of a waveform. When we are dealing with periodic waveforms, the frequency of a component can be implied by its position in the list. The first component is DC, the next the fundamental, then the second harmonic, and so on. So lists with values for each component will be sufficient. What do the two values mean?

Amplitude and phase are one possibility. In that case each pair of numbers is a **polar representation** of the component (phase is an angle, and a number and an angle define a point on a polar plot.) This is very nice, because the list of amplitudes can be charted as a graph of the frequency response. This is what a spectral plot actually is. We generally ignore phase when we are just looking, since phase has no audible effect.[1] For mathematical convenience, the amplitude range is normalized from 0 to 1.0, and the phase is from $-\pi$ to π.[2]

Phase is hard to do math with. The fact that the phase wraps around from $-\pi$ to π, and that math leads to division by 0 make the code awkward. For computation, a **rectangular representation** of amplitude and phase is handier. This can be done by specifying each component as a sum of a cosine wave and a sine wave. This is the most common definition of the Fourier series:

$$1/2a_0 + (a_1\cos(x) + b_1\sin(x)) + (a_2\cos(2x) + b_2\sin(2x)) + \ldots.$$

This is usually output as two lists, one with the a values and the other with the b values. To keep the lists the same length, a b_0 of value 0 is included in the b list. The lists are called the cosine and sine parts or sometimes the real and imaginary parts. (Even if there are no complex numbers around). If there are N points in the input sample, there will be $N/2 + 1$ points in each list. The frequency x is the sample rate divided by N. A 512 point DFT with a sample rate of 44.1 khz gives a fundamental frequency of 86.13 hz. The highest frequency represented is half the sample rate.

---

[1] The inaudible effects of phase are significant, and forgetting about phase when you are processing audio and not just listening is a serious mistake.

[2] We usually do angles in radians. It's nicer for the computers. And yes, negative phase is as likely as positive.

This is called the real DFT. The algorithm that does this is rather inefficient (there's a correlation with every potential sinusoid), so a streamlined version called the **F**ast **F**ourier **T**ransform is preferred. The FFT also outputs two lists, real and imaginary, which actually are complex numbers.

$$(a_0\cos(0) - b_0 j\sin(0)) + (a_1\cos(x) - b_1 j\sin(x)) + (a_2\cos(2x) - b_2 j\sin(2x))$$

These can be changed to amplitude and phase pairs with a simple Cartesian to polar conversion.

There are N components in each list from the FFT . They still represent multiples of $x = SR / N$. The values beyond half the sampling rate represent negative frequency components running from $-(N/2 - 1)x$ to $-x$.[3]

The **FFT** takes complex numbers as it input[4]. If the input is a not a complex signal, the imaginary parts of the input are zero and the output curves will be symmetrical about the $N/2$ point. This implies a lot of wasted computation. This can be skipped, giving the **real FFT**. The output looks just like the complex FFT.

At this point, many may be worrying about how waveforms that are harmonic series on other fundamentals can be accurately represented by components based on the arbitrary frequency SR / N. Consider an example: a 100 hz tone. This falls between the 86.13 hz and the 166.32 hz components of the transform, so would show as an increase in each. These, plus the phase information are enough for the iFFT to give a 100 hz tone back. Of course, the more points in the analysis, the more accurate the reconstructed signal.

Transforming continuous signals

The FFT works with a chunk of signal N points long. The algorithm requires that N be a power of two. If a recording is too short, we can just add zeros, but more likely, we are interested in recordings much longer than N samples. We are also interested in how the transform changes over time, not just the overall average frequency content.

This problem is overcome with a system of windowing, which is similar to the practice of showing moving pictures by projecting a series of still pictures. In essence, the incoming signal is broken into chunks of N points and analyzed as a series of frames. To smooth out the errors caused by this arbitrary chopping,

---

[3] This makes the outputs symmetrical curves. For many purposes the negative frequencies can be ignored, but the iFFT uses them when converting back to waveforms, so the result would be a loss in amplitude.

[4] Most waveforms are real, not complex. Where would you find a complex waveform? As the output of an inverse FFT.

overlapping chunks are processed-- when the signal is reconstituted, the overlapped frames are mixed together. For further smoothing, the frames are faded in and out before the analysis. There are several schemes for doing this. Luckily for Max users, all of this is hidden in the pfft~ object.

Putting the FFT to work.

So, whats the point? Well, there are several pretty neat tricks we can do with Fourier transforms of signals. In MSP the output of fft~ is a pair of synchronized signals, one with the coefficients of the real terms and the other with the coefficients of the imaginary terms. These can be routed and processed just like any other signal. (But you wouldn't want to listen to them!)

Viewing Spectra.

The patch in figure 1 will show the spectrum of a sound

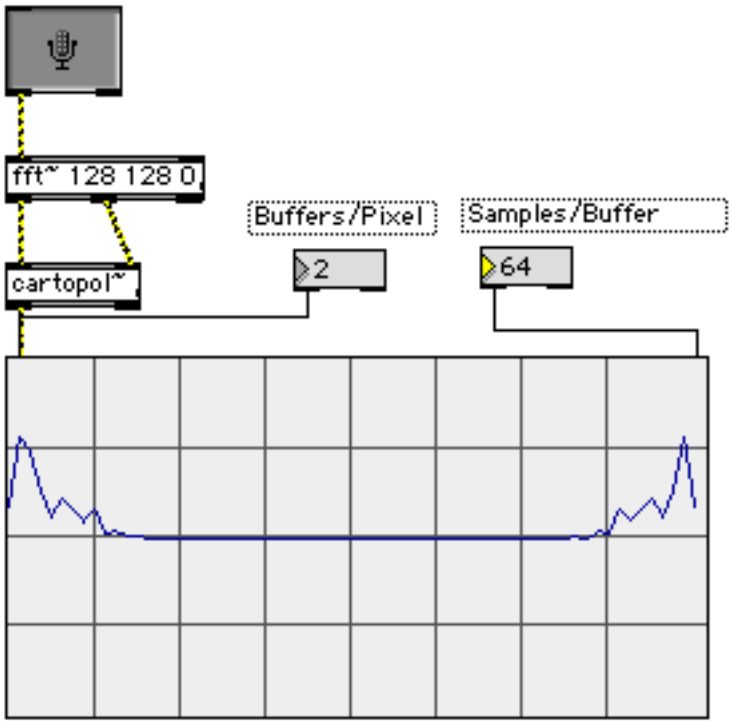

Fig 1. The magnitude spectrum of fft~

Note the magic settings for fft~ and scope~ to get a stable image. The scope max is set to 16.

Cartopol~ is used to convert the rectangular output of fft~ into magnitude and phase format. The spectrum is mirrored because fft~ produces the full complex

spectrum-- those are the negative frequencies at the right. It's hard to see much detail because of the limitations of scope~. Perhaps the future will see a display optimized for this.

Filtering by Convolution

When the fft signal has been converted into magnitude and phase format, it should be clear that you can change the amplitude of the reconstructed signal just by changing the magnitude part of the fft signal. You can also do this by changing both the real and imaginary signals in rectangular notation. If you could isolate particular bands in the fft output, you could change just that part of the spectrum. A method for doing this is shown in the demonstration patch "forbidden planet".

First a work about pfft~[5]. This is a wrapper for fft operations. To use it you specify a subpatch to process the fft data, a frame size, and the number of overlap windows to use. Pfft~ will perform the fft and pass the data to the subpatch, and will reconstruct whatever the subpatch gives back.

In the sub patch, the fft ouputs come from an fftin~ object. This specifies the inlet number on pfft~ this will be connected to, and a window shape if you like to tweak things. fftin~ has three outlets: one for the real fft signal, one for the imaginary fft signal, and one for synchronization - this output is an index to the bin[6] number in the frame. Likewise, what is sent to fftout~ will be reconstructed at the outlet of pfft~.

In "Forbidden Planet" a table containing a shaped spectrum is converted into a signal by loading it into a buffer~. This signal is played in sync with the fft via the index~ object connect to the third output of fftin~. The signal is multiplied by both the real and imaginary parts of the fft and the results sent to fftout~. This will superimpose the shape of the table contents (i.e. the filter curves drawn by the user) on the spectrum of the reconstructed signal.

This is convolution in the time domain by multiplication in the frequency domain.

Signal morphing

The next step is to use an input signal for the modification source. This is illustrated by the convolution workshop patch. In the supatcher, the signal from fftin~ 2 is converted to magnitude and phase, and the magnitude is multiplied by the fftin~1 signal. This superimposes the spectrum of input 2 onto input 1. The

[5] Poly fft~. It's like the Poly~ object, running multiple copies of the same subpatcher.
[6] The something of the A*something* terms.

result is something of both. A gradual transition from one to the other creates a smooth change of timbre.

Cross synthesis

The generalized cross synthesis patch shows the effects of adding spectra. It lets you choose the amplitudes and/or phase signal from either of the two sources or combine the two. You can hear that the amplitude has the most effect, panning from the noise to the rhythm sound. Swapping phase is more subtle, but is noticeable in the case of noise phase and rhythm amplitude. The blue fader brings in a layer of magnitude convolution so you can compare the effects.

To learn more about the Fourier transforms, I suggest you brush up on your math, then study

Smith, Steven; The Scientist and Engineer's Guide to Digital Signal Processing, 1997. (download in pdf or order from www.DSPguide.com)

Roads, Curtis; The Computer Music Tutorial 1996. MIT Press