The Discrete Fourier Transform and Its Use

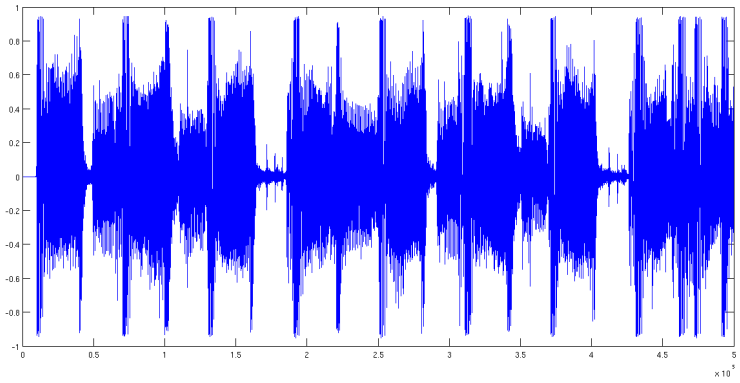
5.35.01 Fall 2011

22 September 2011

Contents

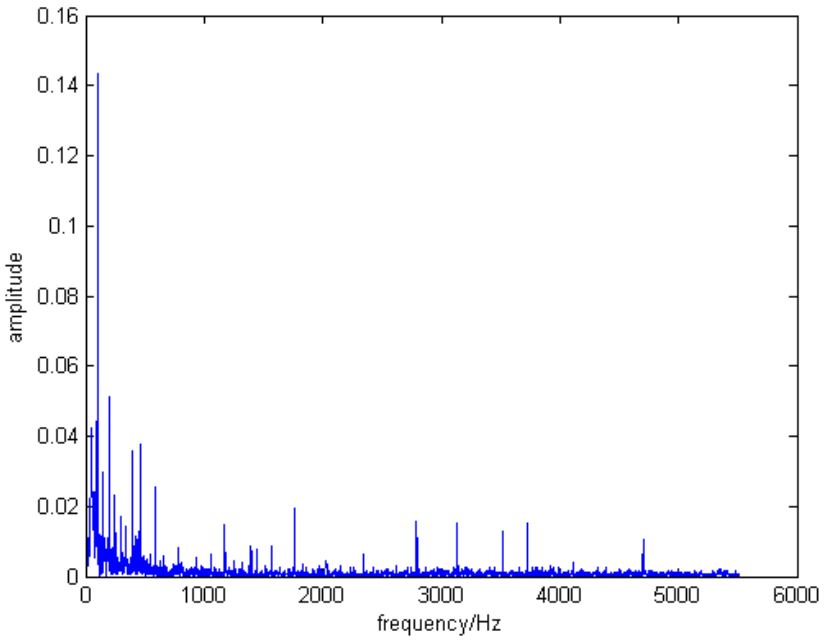
1 Motivation

In physics and engineering, we frequently encounter signals–that is, continuous streams of data with a fixed sampling rate–representing some information of interest. For example, consider the waves generated by an AM or FM transmitter, the crashing of waves on a beach, a piece of music, or raindrops hitting the ground. In each case, we can define some event to monitor, such as the number of raindrops hitting a certain region, and record that data over time. But just having this data does not tell us much, because we are not used to unraveling such complicated and seemingly noisy patterns. For example, below is the waveform (signal intensity, 44.1kHz sampling) of an excerpt from a song:



At first glance, we can see that some patterns emerge. For example, there appears to be a strong signal every few thousand samples with more complicated details in between, but beyond that we cannot easily say much about this signal. But what if we can reduce these data to something more managable, such as the rhythm of each instrument? Certainly, we could sit and listen to the music to discern which instruments play at which frequency if we want to recreate the song, but this requires significant time and effort by a dedicated observer. What if we want to automate the process for practical reasons, such as voice transcription or song identification? In this case, we would need to establish an automated, well-characterized method for analyzing the signal.

One such tool is the Fourier transform, which converts a signal varying in time, space, or any other dimension into one varying in a conjugate dimension. For example, a time-varying signal may be converted into a signal varying in frequency, such that we are able to view monochromatic time oscillations as single peaks in the frequency signal. Applied to our comlex music waveform, we see:



Suddenly, we start seeing that a few key rhythms and instruments are present in our clip, as represented by the tallest peaks. Instead of worrying about the whole signal, we can focus our attention on just the most promienent contributors to the signal, allowing us to address practically these signals, rather than wallowing in a vast sea of numbers.

But what about applications to physical chemistry? We mentioned earlier the concept of interferometry, the technique of examining the intensity of a signal at various points in space as a method for determining the spatial frequencies present in the incoming light, and therefore the temporal frequencies. This is mathematically no different from the clip of music: we sample the intensity at different positions, plot the signal, and use a Fourier transform to discern the contribution of each frequency of light. The result is a spectrum of frequencies (or wavelengths, depending on which way we prefer to label the horizontal axis), which is then available for interpretation using the laws of physics. The two situations are very different, yet both can be viewed in a sufficiently similar way that the mathematics governing the behavior of the two systems becomes identical. The recognition of these isomorphisms in the structure of physical and mathematical phenomena revolutionizes science and technology, because they permit us to solve many problems using identical techniques, meaning that the work put into understanding a technique in one context suddenly pays off in all fields.

2 The Continuous Fourier Transform

2.1 A Brief Introduction to Linear Algebra

Before delving into the mechanics of the Fourier transform as implemented on a computer, it is important to see the origin of the technique and how it is constructed. Therefore, we will start with the continuous Fourier transform, seek an understanding of its structure, and exploit that understanding to see how we might transform discrete signals, as nearly all real data are.

To begin with, recall the concept of a vector. We typically encounter such structures as arrows in Euclidean space which are endowed with direction and magnitude, and we consider two vectors equal when these two properties are the same for each. You may also be familiar with the idea that a vector in space can be expressed as the sum of other vectors, such as the $\hat{\mathbf{i}}$, $\hat{\mathbf{j}}$, and $\hat{\mathbf{k}}$ vectors, representing the x, y, and z axes, respectively. These vectors are orthonormal, that is, each has a magnitude of 1 and sits at a right angle (is orthogonal) to the others. So, together they form what we call an orthonormal basis set, which means that all vectors in this three-dimensional space can be expressed as some linear combination of the basis vectors. More concisely, for basic vectors $\mathbf{e_j}$, constant coefficients c_j, and some vector $\mathbf{v}$ in an N-dimensional vector space spanned by $\{\mathbf{e_j}\}$:

$$\mathbf{v} = \sum_{j=1}^{N} c_j \mathbf{e_j} \tag{1}$$

With a bit of math, we can define some fundamental behavior for abstract vectors and vector spaces, including the abstract idea of an inner product, which you may know as a dot product. Briefly, an inner product acts a measure of the spatial overlap of two vectors, and therefore can be used as a test for orthogonality. The inner product of a vector with itself gives the magntitude of the vector squared, and so we can express any vector for some basic $\{e_j\}$ as:

$$\mathbf{v} = \sum_{j=1}^{N} \frac{\mathbf{v} \cdot \mathbf{e_j}}{\mathbf{e_j} \cdot \mathbf{e_j}} \mathbf{e_j} \tag{2}$$

So, for any vector in a finite vector space, we have a method for expressing the vector as a sum over any basis, meaning that we are free to choose one which is convenient to work with. For continuous vector spaces we must be a little more careful with out definitions, but the basic ideas still hold.

2.2 The Wave Formulation of the Fourier Transform

Recall the Fourier series, which expresses an integrable function with period 2π as a sum of sinusoidal functions:

$$f(t) = \frac{a_0}{2} + \sum_{j=1}^{\infty} a_j \cos(jt) + b_j \sin(jt) \tag{3}$$

We can view the various sinusoids and the constant coefficient as vectors, and define the inner product as an integral over the period (with a constant factor for normalization):

$$f(t) \cdot g(t) = \frac{1}{\pi} \int_{-\pi}^{\pi} f(t)g(t)dt \tag{4}$$

Under this operation, we can see that each sine, cosine, and the constant are mutually orthogonal, and with a bit more work we can prove that they span all integrable periodic functions. Perhaps more importantly, this method allows us a simple way to compute the expansion of the function on the basic set, freeing us to note the contribution of each frequency. Building on this, the extension to an arbitrary integrable function involves the relaxation of the restriction of periodicity and allow any wave frequency in our basis set. The basic structure of the space has changed somewhat, but the inner product we defined holds if we extend the limits to be infinite and introduce a factor to correct for volume[1]. Therefore, for $f(t)$ to be expressed as a sum of waves with frequencies ω and coefficients $F(\omega)$, the projection of $f(t)$ onto the wave basis set will look something like:

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t) \cos(\omega t)dt \tag{5}$$

But, this is not quite complete, as we need to consider functions which are not at their maximum at $t = 0$. So, we could write a similar expression for $\sin(\omega t)$, or we could be more clever.

Consider the complex numbers, which we can express in a number of equivalent ways:

$$z = a + ib = |z|e^{i\theta} = |z| \cos(\theta) + i|z| \sin(\theta) \tag{6}$$

So, what if we recast our view of waves in light of this? We can express compactly the idea of a wave propagating along an axis by viewing the wave as extending onto the complex axis, as a sort of helix of fixed diameter which coils around the axis of propagation. Since (in most real-world examples) our original wave is a real function, we can just take the real part of this wave function, which, with complex coefficients allowed, becomes some sum of sine and cosine functions, leading to the phase offset we were originally seeking. So, we can express the more complete transform:

$$F(\omega) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} f(t)e^{-i\omega t}dt \tag{7}$$

where we allow $F(\omega)$ to take on complex values, to account for this phase shift[2]. We can also invert this expression:

$$f(t) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{\infty} F(\omega)e^{i\omega t}dt \tag{8}$$

This we call the Fourier transform and its inverse.

[1] This is related to the conversion between units in the transform, from angular frequency to regular frequency.

[2] $\mathrm{Re}\left((a+ib)e^{i\omega t}\right) = \mathrm{Re}\left((a+ib)\cos(\omega t) + (ia-b)\sin(\omega t)\right) = a\cos(\omega t) - b\sin(\omega t)$

3 The Discrete Fourier Transform (DFT)

The continuous Fourier transform is itself a great feat of mathematics, and an enormous number of interesting problems can be expressed and solved efficiently in this form. However, there is one problem: while we can work on continuous intervals without issue in mathematics, the real world tends to be a little more discrete. We cannot really have π apples on the table, unless you are the sort of person who likes to work in unusual bases, and likewise the clock mechanisms in timepieces and computers count using some division of seconds. Similarly, computer memory is divided into discrete blocks, and we store our streams of data using these discrete blocks. In such a world, it becomes highly impractical to think about varying continuously over frequencies, but what if we instead somehow choose only a limited number of frequencies, such that we can approximate the true result in a way which balances practicality and accuracy?

3.1 An Efficient Algorithm Exists

The good news is, we can. An algorithm for computing the Fourier transform of a discrete function (the discrete Fourier transform or DFT) efficiently does exist, and exploits the mathematical structure of the polynomials, which can also be viewed using the linear algebra framework we used. Using certain tricks, the fast Fourier transform (FFT) can be used to calculate the DFT much more rapidly, with the running time required being $\mathrm{O}(n \ln(n))$ instead of $\mathrm{O}(n^2)$, where n is the signal length and O represents the way the running time of the operation scales with the paramater n. What this means is that doubling the signal only slightly more than doubles the time needed to compute the transform, rather than quadrupling it, and so we are able to work with rather large signals in a more reasonable length of time.

The details of the FFT are far too gory to be spelled out here, but the basic idea is that we can express the inner product of each reference frequency with the signal as a sum over complex divisions of unity ($e^{i\omega t}$). These products can in turn be expressed as a matrix product of with the signal vector, meaning our problem becomes a matter of efficiently multiplying a matrix with a vector. This can be broken down into subproblems by quartering the matrix repeatedly, which is how we achieve the reduction to $O(n \ln(n))$ growth. A full derivation of the method can be found in any good book on algorithm design. See, for example, chapter 30 of the second edition of Carmen, Leiserson, Rivest, and Stein's *Introduction to Algorithms*. Weisstein gives a rather technical overview of the mathematics of the algorithm on his website at `http://mathworld.wolfram.com/FastFourierTransform.html`, with plenty of references to more complete manuscripts on the subject. To see how the algorithm is implemented on a computer, the GNU Scientific Library (`http://www.gnu.org/software/gsl/`) is open-source and includes a few different FFT algorithms.

4 Using the FFT in Matlab

Now that we have seen a bit of where the Fourier transform comes from and have satisfied ourselves that an efficient method for computing the DFT of a signal exists, we can exploit our knowledge for practical gain. Normally, this would require that we actually sit down and program a computer to execute the FFT, but the good folks writing mathematics packages such as Matlab, Mathematica, the GNU Scientific Library, and countless others have already done this for us. As a result, any numerical analysis package worth using should include this function, but today we will be using Matlab.

Why Matlab? For one, Matlab acts as a friendly interface to the numerical workhorse that is Fortran. Another good reason is that MIT pays for us to have a license to use the software, and so we might as well use something that is well-written, has a nice enough interface, and costs nothing to use (at least until graduation). There are alternatives from the open source community which are free regardless of who you are (GSL, for one), but most of these require a little more computing expertise than is practical for this class[3]. In short, Matlab provides us the most power for the least effort required to understand how a computer works, leaving us free to worry more about our methods than their low-level implementation.

So, how do we use Matlab? There are a number of texts on this subject, including Mathwork's tutorial on their product and dozens of "Matlab for x" textbooks, where x is your field of study. So, here we will cover only the most important aspects of the operation of the software.

The first step is to download and install Matlab, following the directions available at `http://ist.mit.edu/services/software/math/tools`. Packages are available for Windows, Macintosh, and Linux, all of which have somewhat different interfaces and commands shortcuts. However, the basic functionality of the package remains the same, so we will talk mostly about the Matlab commands of interest.

Once you install and run the package, you should be greeted with an interpreter with all sorts of subwindows. The one we care about most is in the center, as this is our command line interface to the Matlab engine. We will enter our commands and see results here when working interactively, and when running scripts and errors or results will also be displayed here.

Matlab's data structure is the array, which can have essentially any number of dimensions, limited by the memory in your computer. We fill this array with a single data type, such as 32-bit integers or floating-point numbers, and can access the elements by choosing the appropriate indices. We can even use a nice shorthand to create certain types of arrays:

[3] If you find that you are interested in physical chemistry more than just as a hobby, it would be wise to acquire a working knowledge of computer science fundamentals, such as those taught in Abelson and Sussman's *Structure and Interpretation of Computer Programs*, as well as a solid understanding of algorithmic design. Even if you never end up writing much code yourself, the discipline necessary to understand the analysis will prove highly useful in experimental design.

```
>> 1:10
ans =
     1  2  3  4  5  6  7  8  9  10
```

In the array above, we created an array of ascending numbers, by default separated by 1. We can also change the spacing of the array elements by inserting a third term:

```
>> 1:0.5:3
ans =
    1.0000  1.5000  2.0000  2.5000  3.0000
```

Notice that we now have an array of floating-point numbers, rather than integers. The representation of floating point numbers on a computer is approximate, so when performing calculations we must make sure that the precision of the representation (32-bit, 64-bit, etc) is sufficient for our needs[4]. In this case we are happy with just a few significant figures, so the standard 32-bit floating point number is adequate.

Now that we can create arrays, we can store and address them (a semicolon suppresses the output of a command):

```
>> a = 1:10;
>> a(2:5)
ans =
    2  3  4  5
```

We can even create multidimensional arrays and address each dimension separately:

```
>> a = rand(3,3)
a =
    0.9649    0.9572    0.1419
    0.1576    0.4854    0.4218
    0.9706    0.8003    0.9157
>> a(:,3)
ans =
    0.1419
    0.4218
    0.9157
```

Here, the : is used as shorthand for the whole dimension. So, we have asked Matlab for every row of the third column, but we could easily mix and match the indices to get the exact window desired.

The real strength of Matlab is that it contains some high-quality algorithms for calculating quantities of great interest to us. For example, we can rapidly

[4]If you will ever have to work with computers in anything more than a passive capacity, learn about machine structures and their implications. Knowing the limitations of your instruments is absolutely essential to performing good experimental work.

calculate the eigenvalues and eigenvectors of large matrices, take Fourier transforms of long signals, perform statistical analysis of large data sets, make plots of thousands of data points, and so on. The breadth of what Matlab can do is documented in the manual for the package, although many books also exist to introduce a reader to some of the most relevant commands. For now, let us turn our attention to a simple script for taking a Fourier transform and plotting it.

4.1 A Mixture of Sinusoids

Here is a brief script to compute a signal, take its Fourier transform, and then invert the transform to show that we recover the original signal:

```
Fs = 1000; % frequency of sampling
T = 1/Fs; % sample time
L = 1000; % length of sampling
t = (0:L-1)*T;

% mix up the signal
frequencies = [1, 120; 0.7, 100; 0.9, 60];
data = frequencies(:,1)'*sin(2*pi*frequencies(:,2)*t);

data = data + 0.5*(1-2*rand(1,L));

subplot(3,1,1);
plot(Fs*t,data);
xlabel('t/ms');
ylabel('intensity');

NFFT = 2^nextpow2(L);
Y = fft(data, NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);

subplot(3,1,2);
plot(f,2*abs(Y(1:NFFT/2+1)));
xlabel('frequency/Hz');
ylabel('|F(\omega)|');

subplot(3,1,3);
recovered = L*ifft(Y);
plot(Fs*t,recovered(1:L),'b',
     Fs*t,data-recovered(1:L), 'r');
xlabel('t/ms');
ylabel('intensity');
```

Let us walk through the script and figure out what each part does. First, we set some parameters to give meaning to the arrays, which are otherwise unitless[5]:

```
Fs = 1000; % frequency of sampling
T = 1/Fs; % sample time
L = 10000; % length of sampling
t = (0:L-1)*T;
```

Next, we create the signal by amplifying the value of each sinusoid by some amplitude, as stored as two columns in our array frequencies.

```
% mix up the signal
frequencies = [1, 120; 0.7, 100; 0.9, 60];
data = frequencies(:,1)'*sin(2*pi*frequencies(:,2)*t);
```

Now that we have created the signal, we introduce some random noise:

```
data = data + 0.5*(1-2*rand(1,L));
```

Now, `data` contains values calculated for various values of t according to the function:

$$f(t) = \sin(2\pi(120)t) + 0.7\sin(2\pi(100)t) + 0.9\sin(2\pi(60)t) + \eta(t) \qquad (9)$$

Next, we plot the signal as a function of time:

```
subplot(3,1,1);
plot(Fs*t,data);
xlabel('t/ms');
ylabel('intensity');
```

Now we take the FFT of the data, scaling the length of the signal to a power of 2 to optimize the accuracy of the result (see a description of the algorithm to understand why this works):

```
NFFT = 2^nextpow2(L);
Y = fft(data, NFFT)/L;
f = Fs/2*linspace(0,1,NFFT/2+1);
```

Plot the FFT below the signal, using the magnitude of each coefficient to show the contribution of the frequency only, ignoring the phase shift. Normally the FFT gives us a double-sided result, but we ignore the high-frequency signals because they are a mirror of the lower frequencies in amplitude:

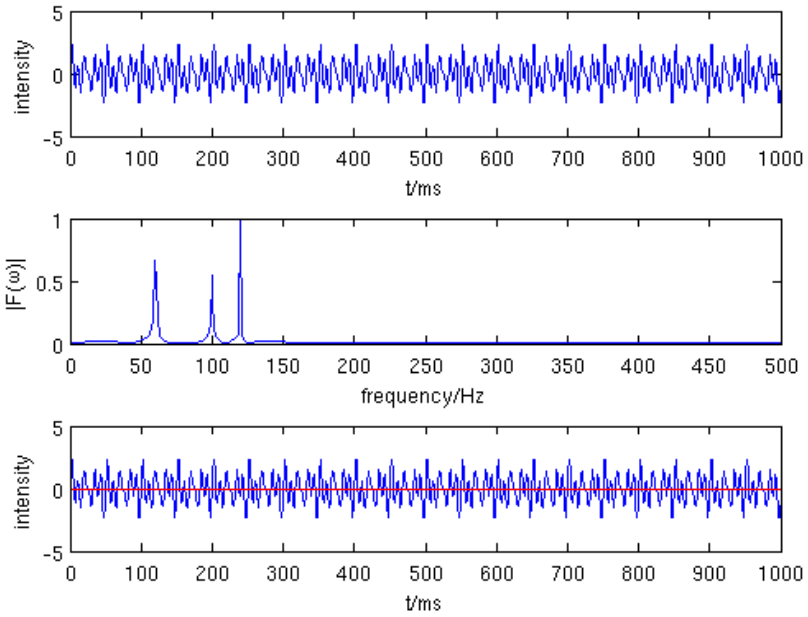
```
subplot(3,1,2);
plot(f,2*abs(Y(1:NFFT/2+1)));
xlabel('frequency/Hz');
ylabel('|F(\omega)|');
```

[5]To a computer, numbers do not inherently have units. It is up to us to keep track of them ourselves and verify that they are, in fact, reasonable values.

Finally, perform the inverse of the FFT, take the difference with the orignal signal, and plot both:

```
subplot(3,1,3);
recovered = L*ifft(Y);
plot(Fs*t,recovered(1:L),'b',
     Fs*t,data-recovered(1:L), 'r');
xlabel('t/ms');
ylabel('intensity');
```

Running this script gives us a nice plot with some useful information and demonstrates that the algorithm does, in fact, work:



Note that our signal peaks, while clearly present, are broadened. This is an artifact of the discrete nature of our transform, because we cannot perform an infinitely precise inner product to obtain an infinitessimally wide peak. This finite resolution can be improved by increasing our sampling frequency, but in some cases this becomes impractical or impossible. It is an important skill to recognize these limitations before committing time and effort to an experiment, so play around with the script to understand the limitations of this idealized situation. For example, try adding a high-frequency component to the signal, such as 550Hz. Where does this appear on the frequency spectrum?

Now that we have a working script, we can put in any signal we want, scale the units appropriately, and see the result. For example, we can replace our sinusoids with `sawtooth`, `square`, or even a signal of our choosing, such as an

interferogram. Algorithms even exist to perform a multidimensional FFT, so we can use this same technique to study diffraction patterns in crystals or solutions to gain insight as to the average structure of a material[6].

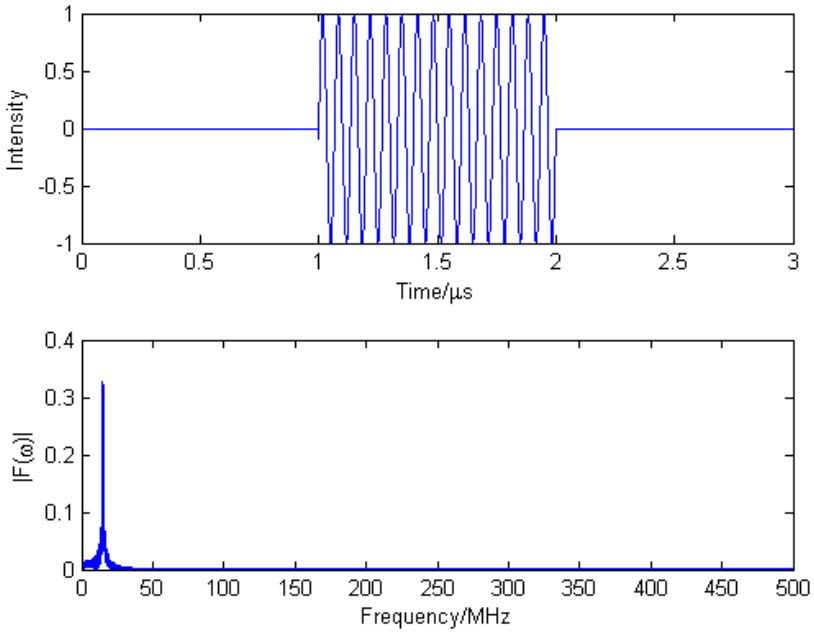
4.2 A Finite-Length Pulse

Below is a script for generating a pulse of a given length, time offset, and central frequency, and for computing its FFT:

```
Fs = 10^9;                    % Sampling frequency
T = 1/Fs;                      % Sample time
L = 3*10^-6;                     % Length of signal in time
L_counts = ceil(L/T);          % length of signal in samples
t = (0:L_counts-1)*T;          % Time vector
pw = 1*10^-6;                  % time that our packet exists
pw_counts = ceil(pw/T);        % length of the packet in samples
ps = 1*10^-6;                  % offset from zero for the packet
ps_counts = ceil(ps/T);        % offset of the packet in samples
f = 15*10^6;                   % packet frequency

y = sin(2*pi*(t-ps)*f);
y(1:(ps_counts-1)) = 0;
y((ps_counts+pw_counts):L_counts) = 0;

subplot(2,1,1);
plot(t*10^6,y)
xlabel('Time/\mus');
ylabel('Intensity');

NFFT = 2^nextpow2(L_counts); % Next power of 2 from length of y
Y = fft(y,NFFT)/L_counts;
f = Fs/2*linspace(0,1,NFFT/2+1);

% Plot single-sided amplitude spectrum.
subplot(2,1,2);
plot(f,2*abs(Y(1:NFFT/2+1)))
xlabel('Frequency/Hz')
ylabel('|F(\omega)|')
```

And an example result:

[6]Some theoretical examples are discussed by Chandler in his *Introduction to Modern Statistical Mechanics*, and Guinier's *X-Ray Diffraction* has a more complete discussion.



These sorts of pulses are important in NMR spectroscopy, where you might hear them referred to as π or $\pi/2$ pulses. Think about these ideas when performing your NMR experiments for this class.

5 Conclusions and Further Directions

Because of the general way we can apply the Fourier transform to a class of problems, it is considered one of the most practical and beautiful products of mathematics. Because of its efficiency, the transform has been used as a building point for other techniques vital to classical computation, signal analysis, high-frequency design, precision instrumentation, and a variety of other fields. Even emerging fields make use of the Fourier transform: Shor's quantum factoring algorithm[7] is fundamentally a Fourier transform of a signal composed of values computed in a cyclic group, and most efficient quantum algorithms employ a quantum Fourier transform to achieve their complexity reduction. Coherent control of light for spectroscopy and communication is governed by the mixing of optical frequencies, and a number of important projects are being carried out to understand the behavior of increasingly short and well-defined pulses of light in various systems. No matter where you look in physics, engineering, and computer science, the Fourier transform is an absolutely essential tool, and learning to recognize situations where it is applicable is a skill which will serve well over time.

[7] http://arxiv.org/abs/quant-ph/9508027

5.35 / 5.35U Introduction to Experimental Chemistry
Fall 2012