E E 2 7 5 Lab June 30, 2006

Lab 2. Spectral Analysis in Matlab

Introduction

This lab will briefly describe the following topics:
• Signal statistics
• Discrete Fourier Transform
• Power Spectral Density estimation
• Time-varying spectra
• Wavelets

There are a number of important statistical signal processing concepts in this Lab. We will fully appreciate some of them as we proceed in class. You can certainly read about them in this Lab, but do not have to do the simulations. (Some functions are missing).

For this Lab. do the simulations up to **Q2** on page 3. Then go to "Lab 19 The Fast Fourier Transform" and do that Lab.

For your report, you need to answer **Q1, Q2** and the 7 questions on page 3 of Lab.19.

Statistical Signal Processing

Repeated measurements of a signal x under identical environmental conditions typically yield different waveforms. The value $x(n)$ of a signal at sample n is not a constant, but rather a random variable with a certain distribution.

A signal is *stationary* if its distribution at any n is independent of time shifts, i.e., if $x(n)$ and $x(n+N)$ have the same distribution for every integer N. Stationary signals are associated with steady states in the system producing the signal, and are characterized by constant averages and variances along the signal. When statistics computed along any instance of a signal are identical to those computed across ensembles of different instances of the signal, the system is *ergodic*. Ergodicity is assumed in most practical situations.

Matlab has functions for computing the mean (*mean*), variance (*var*), standard deviation (*std*), covariance (*cov*), and correlation coefficient (*corrcoeff*) of signal values sampled across time or across ensembles. The Statistics Toolbox provides many additional statistical functions.

The Signal Processing Toolbox provides a *crosscorrelation* function (*xcorr*) and a *crossco-*

variance function (*xcov*). Crosscorrelation of two signals is equivalent to the convolution of the two signals with one of them reversed in time. The *xcorr* function adds several options to the standard Matlab *conv* function. The *xcov* function simply subtracts the mean of the inputs before computing the crosscorrelation. The crosscorrelation of a signal with itself is called its *autocorrelation*. It measures the *coherence* of a signal with respect to time shifts.

Try:

```
x = randn(1,100);
w = 10;
y = conv(ones(1,w)/w,x);
avgs = y(10:99);
plot(avgs)
```

Ensemble averages:

```
w = 10;
for i = 1:w;
X(i,:)  = randn(1,100);
end
AVGS = mean(X);
plot(AVGS)
```

```
x = [-1 0 1];
y = [0 1 2];
xcorr(x,y)
conv(x,fliplr(y))
xcov(x,y)
xcov(x,x)
xcov(x,y-1)
```

Q1: What do you notice about these vectors?
Explain why you see these results.

Example: Crosscorrelation

In a simple target ranging system, an outgoing signal x is compared with a returning signal y. We can model y by

$$y(n) = \alpha x(n-d) + \beta$$

where α is an attenuation factor, d is a time delay, and β is channel noise. If T is the return time for the signal, then x and y should be correlated at $n = T$. The target will be located at a distance of vT, where v is the channel speed of the signal.

Try:

```
x = [zeros(1,25),1,zeros(1,25)];
subplot(311), stem(x)
y = 0.75*[zeros(1,20),x] + 0.1*randn(1,71);
subplot(312), stem(y)
[c lags] = xcorr(x,y);
subplot(313), stem(lags,c)
```

Q2: Does this example show the expected behavior?
Why or why not?

Discrete Fourier Transform (DFT)

It is often useful to deompose data into component frequencies. *Spectral analysis* gives an alternative view of time or space-based data in the *frequency domain*. The computational basis of spectral analysis is the *discrete Fourier transform* (DFT).

The DFT of a vector y of length n is another vector Y of length n:

$$Y_{k+1} = \sum_{j=0}^{n-1} \omega^{jk} y_{j+1}$$

where ω is a complex n^{th} root of unity:

$$\omega = e^{-2\pi i/n}$$

This notation uses i for the complex unit, and j and k for indices that run from 0 to $n-1$. The subscripts $j+1$ and $k+1$ run from 1 to n, corresponding to the range usually associated with Matlab vectors.

Data in the vector y are assumed to be separated by a constant interval in time or space $dt = 1/Fs$. Fs is called the *sampling frequency* of y. The coefficient Y_{k+1} measures the amount of the frequency $f = k(Fs/n)$ that is present in the data in y. The vector Y is called the *spectrum* of y.

The midpoint of Y (or the point just to the right of the midpoint, if n is even), corresponding to the frequency $f = Fs/2$, is called the *Nyquist point*. The real part of the DFT is symmetric about the Nyquist point.

The graphical user interface `fftgui` allows you to explore properties of the DFT. If y is a vector,

```
fftgui(y)
```

plots `real(y)`, `imag(y)`, `real(fft(y))`, and `imag(fft(y))`. You can use the mouse to move any of the points in any of the plots, and the points in the other plots respond.

Try:

Roots of unity

```
edit z1roots
z1roots(3);
z1roots(7);
```

Explore the DFT:

```
delta1 = [1 zeros(1,11)];
fftgui(delta1)


delta2 = [0 1 zeros(1,10)];
fftgui(delta2)


deltaNyq = [zeros(1,6),1,zeros(1,5)];
fftgui(deltaNyq)


square = [zeros(1,4),ones(1,4),zeros(1,4)];
fftgui(square)


t = linspace(0,1,50);
periodic = sin(2*pi*t);
fftgui(periodic)
```

Fast Fourier Transform (FFT)

The Matlab function `fft`, called by `fftgui`, uses a *fast* Fourier transform algorithm to compute the DFT.

DFTs with a million points are common in applications. For modern signal and image processing applications, and many other applications of the DFT, the key is the ability to do such computations rapidly. Direct application of the definition of the DFT requires n multiplications and n additions for each of the n coefficients - a total of $2n^2$ floating-point operations. This number does not include the generation of the powers of ω. To do a million-

point DFT, a computer capable of doing one multiplication and addition every microsecond would require a million seconds, or about 11.5 days.

Modern FFT algorithms have computational complexity $O(nlog_2n)$ instead of $O(n^2)$. If n is a power of 2, a one-dimensional FFT of length n requires less than $3nlog_2n$ floating-point operations. For $n = 2^{20}$, that's a factor of almost 35,000 times faster than $2n^2$.

When using the FFT, a distinction is often made between a *window length* and an *FFT length*. The window length is the length of the input. It might be determined by, say, the size of an external buffer. The FFT length is the length of the output, the computed DFT. The command

```
Y=fft(y)
```

returns the DFT Y of y. The window length `length(y)` and the FFT length `length(Y)` are the same.

The command

```
Y=fft(y,n)
```

returns the DFT Y with length n. If the length of y is less than n, y is padded with trailing zeros to length n. If the length of y is greater than n, the sequence y is truncated. The FFT length is then the same as the padded/truncated version of the input y.

Try:

Vector data interpolation and the origins of the FFT

```
edit fftinterp
fftinterp
hold off
```

Note: Several people discovered fast DFT algorithms independently, and many people have since joined in their development, but it was a 1965 paper by John Tukey of Princeton University and John Cooley of IBM Research that is generally credited as the starting point for the modern usage of the FFT. The Matlab `fft` function is based on FFTW, "The fastest Fourier Transform in the West", developed by MIT graduate students Matteo Frigo and Steven G. Johnson. (http://www.fftw.org)

Spectral Analysis with the FFT

The FFT allows you to estimate efficiently the component frequencies in data from a discrete set of values sampled at a fixed rate. The following list shows the basic relationships among the various quantities involved in any spectral analysis. References to time can be replaced by references to space.

`y` Sampled data

`n = length(y)`	Number of samples
`Fs`	Samples/unit time
`dt = 1/Fs`	Time increment
`t = (0:n-1)/Fs`	Time range
`Y = fft(y)`	Discrete Fourier Transform (DFT)
`abs(Y)`	Amplitude of the DFT
`abs(Y).^2/n`	Power of the DFT
`Fs/n`	Frequency increment
`f = (0:n-1)*(Fs/n)`	Frequency range
`Fs/2`	Nyquist frequency

A plot of the power spectrum is called a *periodogram*. The first half of the principal frequency range (from 0 to the Nyquist frequency $Fs/2$ is sufficient, because the second half is a reflection of the first half.

Spectra are sometimes plotted with a principal frequency range from $-Fs/2$ to $Fs/2$. Matlab provides the function `fftshift` to rearrange the outputs of `fft` and convert to a 0-centered spectrum.

Try:

Periodograms

```
edit pgrams
pgrams
```

Whale call

```
edit whalefft
whalefft
```

FFT Demos

```
sigdemo1
playshow fftdemo
phone
playshow sunspots
```

If you're not familiar with DTMF, try the following with a touch-tone telephone. While listening to the receiver, press two keys in the same row simultaneously, then press two keys in the same column (e.g., 1 & 2, then 1 & 4).

Aliasing

Discrete time signals sampled from time-periodic analog signals need not be time periodic, but they are always periodic in frequency, with a period equal to the sampling frequency. The resulting *harmonics* show up as spectral copies in frequency domain plots like the periodogram. If the sampling rate is too low, these spectral copies can overlap within the principal range, confusing the frequency analysis.

Power Spectral Density (PSD)

The *power spectral density* (PSD) of an analog signal y is a function of frequency, $R_{yy}(f)$, whose area equals the total signal power. Its units are, e.g., watts/hertz, and $R_{yy}(f)\Delta f$ approximates signal power over a small range of frequencies Δf centered at f. The Wiener-Khintchine theorem states that $R_{yy}(f)$ is the DFT of the autocorrelation function $r_{yy}(t)$ of y. The value $R_{yy}(0) = r_{yy}(0)$ gives the average power in the signal.

For signals sampled over a finite interval of time, the best we can do is estimate the PSD. This result is because the spectra of finite sequences suffer from both poor resolution and *leakage* (nonzero spectral components at frequencies othe than harmonics of y due to sampling over noninteger multiples of the signal period).

PSD estimates of noisy analog signals from a finite number of its samples are based on three fundamentally different approaches:

• Non-parametric methods
Make no assumptions about the data in the sample and work directly with the DFT.
Welch: `pwelch`
Multitaper: `pmtm`

• Parametric methods
Model the data in the sample as the output of a linear system excited by white noise (noise with zero mean and constant PSD), estimate the filter coefficients, and use these to estimate the PSD.
Burg: `pburg`
Yule-Walker: `pyulear`

• Subspace methods
Based on an eigenanalysis or eigendecomposition of the correlation matrx associated with the data in the sample.
EV: `peig`
MUSIC: `pmusic`

Try:

```
Fs = 100;
t = 0:1/Fs:10;
y = sin(2*pi*15*t) + sin(2*pi*30*t);
nfft = 512;
Y = fft(y,nfft);
f = Fs*(0:nfft-1)/nfft;
Power = Y.*conj(Y)/nfft;
plot(f,Power)
title('Periodogram')


figure
ryy = xcorr(y,y);
Ryy = fft(ryy,512);
plot(f, abs(Ryy))
title('DFT of Autocorrelation')
```

Non-parametric Methods

Non-parametric methods estimate the PSD directly from the signal itself. The simplest such method is the periodogram. An improved version of the periodogram is Welch's method. A more modern technique is the multitaper method (MTM).

The following functions estimate the PSD Pxx in units of power per radians per sample. The corresponding vector of frequencies w is computed in radians per sample, and has the same length as Pxx.

• Periodogram method

`[Pxx w] = periodogram(x)`

Estimates the PSD using a periodogram. Optional inputs specify windows (default is rectangular), FFT length, PSD sample frequencies, and output frequency range.

• Welch method

`[Pxx w] = pwelch(x)`

Estimates the PSD using Welch's averaged periodogram method. The vector x is segmented into equal-length sections with overlap. Trailing entries not included in the final segment are discarded. Each segment is windowed. Optional inputs specify windows (default is Hamming), overlap, FFT length, and PSD sample frequencies.

• **Multitaper method**

```
[Pxx w] = pmtm(x, nw)
```

Estimates the PSD using a sequence of $2*nw-1$ orthogonal tapers (windows in the frequency domain). The quantity nw is the time-bandwidth product for the discrete prolate spheroidal sequences specifying the tapers. Optional inputs specify taper frequencies, FFT length, and PSD sample frequencies.

Try:

```
t = 0:1/100:10-1/100;
x = sin(2*pi*15*t) + sin(2*pi*30*t);
periodogram(x,[],512,100);
figure
pwelch(x,[],512,100);
figure
pmtm(x,[],512,100);
```

Comment on the three methods.
Is one preferable to the other two?
Why or why not?

Parametric Methods

Parametric methods can yield higher resolution than non-parametric methods in cases where the signal length is short. These methods use a different approach to spectral estimation: instead of estimating the PSD directly from the data, they model the data as the output of a linear system driven by white noise (an adaptive filter), and then attempt to estimate the parameters of that linear system.

The most commonly used linear system model is the all-pole model, a system with all of its zeros at the origin in the z-plane. The output of such a system for white noise input is an autoregressive (AR) process. These methods are sometimes referred to as *AR methods*.

AR methods give accurate spectra for data that is "peaky," that is, data with a large PSD at certain frequencies. The data in many practical applications (such as speech) tends to have peaky spectra, so that AR models are often useful. In addition, the AR models lead to a system of linear equations that is relatively simple to solve.

The following methods are summarized on the next page. The input p specifies the order of the autoregressive (AR) prediction model.

• **Yule-Walker AR method**

```
[Pxx f] = pyulear(x,p,nfft,fs)
```

• **Burg method**

```
[Pxx f] = pburg(x,p,nfft,fs)
```

• **Covariance and modified covariance methods**

```
[Pxx f] = pcov(x,p,nfft,fs)
[Pxx f] = pmcov(x,p,nfft,fs)
```

Try:

```
edit pmethods
pmethods('pyulear',25,1024)
pmethods('pburg',25,1024)
pmethods('pcov',5,512)
pmethods('pmcov',5,512)
```

Some of the factors to consider when choosing among parametric methods are summarized in the following table. See the documentation for further details.

Burg	**Covariance**	**Modified Covariance**	**Yule-Walker**
Characteristics			
Does not apply window to data	Does not apply window to data	Does not apply window to data	Applies window to data
Minimizes forward and backward prediction errors	Minimizes forward prediction error	Minimizes forward and backward prediction errors	Minimizes forward prediction error
Advantages			
High resolution for short data records	Better resolution than Y-W for short data records	High resolution for short data records	As good as other methods for large data records
Always produces a stable model	Extract frequencies from mix of p or more sinusoids	Extract frequencies from mix of p or more sinusoids	Always produces a stable model
		No spectral line-splitting	
Disadvantages			

Peak locations highly dependent on initial phase	Can produce unstable models	Can produce unstable models	Performs relatively poorly for short data records
Spectral line-splitting for sinusoids in noise, or when order is very large	Frequency bias for estimates of sinusoids in noise	Peak locations slightly dependent on initial phase	Frequency bias for estimates of sinusiods in noise
Frequency bias for sinusoids in noise		Minor frequency bias for sinusiods in noise	

Conditions for Nonsingularity			
	p must be $\leq 1/2$ input frame size	p must be $\leq 2/3$ input frame size	Autocorrelation matrix always positive-definite, nonsingular

Subspace Methods

Subspace methods, also known as *high-resolution methods* or *super-resolution methods*, generate PSD estimates based on an eigenanalysis or eigendecomposition of the correlation matrix. These methods are best suited for *line spectra* - i.e., spectra of sinusoidal signals - and are effective for detecting sinusoids buried in noise, especially when signal to noise ratios are low.

The following functions estimate the *pseudospectrum* S (an indicateor of the presence of sinusoidal components in a signal) of the input signal x, and a vector w of normalized frequencies (in rad/sample) at which the pseudospectrum is evealuated. The input p controls the dimensions of the signal and noise subspaces used by the algorithms.

• Eigenvector method

```
[S f] = peig(x,p,nfft,fs)
```

• Multiple Signal Classification (MUSIC) method

```
[S f] = pmusic(x,p,nfft,fs)
```

The MUSIC algorithm uses Schmidt's eigenspace analysis method. The eigenvector uses a weighted version of the MUSIC algorithm.

Try:

```
edit ssmethod
ssmethod(3)
ssmethod(4)
```

Spectrum Viewer in SPTool

The SPTool allows you to view and analyze spectra using different methods. To create a spectrum in SPTool,

1. Select the signal in the **Signals** list in SPTool.
2. Select the **Create** button under the **Spectra** list.

Use the **View** button under the **Spectra** list in the SPTool GUI to display one or more selected spectra.

Try:

```
t = 0:1/100:10-1/100;
x = sin(2*pi*15*t) + sin(2*pi*30*t);
```

Import this signal into SPTool and view the spectrum using various methods.

Time-Varying Spectra

The spectral estimation methods described so far are designed for the analysis of signals with a constant spectrum over time. In order to find time-varying spectra, different methods of analysis and visualization must be used.

The *time-dependent Fourier transform* of a signal is a sequence of DFTs computed using a sliding window. A *spectrogram* is a plot of its magnitude versus time.

```
[B f t] = specgram(x,nfft,fs,window,numoverlap)
```

calculates the time-dependent Fourier transform for the signal in vector x and returns the DFT values B, the frequency vectors f, and the time vectors t. The spectrogram is computed as follows:

1. The signal is split into overlapping sections and applies to the window specified by the *window* parameter to each section.
2. It computes the discrete-time Fourier transform of each section with a length $nfft$ FFT to produce an estimate of the short-term frequency content of the signal; these transforms make up the columns of B. The quantity $length(window) - numoverlap$ specifies by how many samples *specgram* shifts the window.

3. For real input, *specgram* truncates to the first $nfft/2 + 1$ points for $nfft$ even and $(nfft + 1)/2$ for $nfft$ odd.

When called with no outputs, *specgram* displays the spectrogram.

Try:

Time-constant spectrum

```
t = 0:1/100:10-1/100;
x = sin(2*pi*15*t) + sin(2*pi*30*t);
specgram(x,256,100,hann(21),15)
colorbar
```

Time-varying spectrum

```
load handel
sound(y,Fs)
specgram(y,512,Fs,kaiser(100,5),75)
colorbar
```

Spectrogram Demos

```
specgramdemo
xpsound
```

Example: Reduced Sampling Rate

This example compares the sound and spectrogram of a speech signal sampled at progressively reduced rates.

If you resample a signal at a fraction of the original sampling frequency, part of the signal's original frequency content is lost. The down-sampled signal will contain only those frequencies less than the new Nyquist frequency. As down-sampling continues, the words in the signal remain recognizable long after the original spectrogram has become obscured. This is a tribute to the human auditory system, and is the basis of signal compression algorithms used in communications.

Try:

```
edit HAL9000
HAL9000
```

Wavelets

One of the drawbacks of the Fourier transform is that it captures frequency information about a signal without any reference to time. For stationary signals this is unimportant, but for time-varying or "bursty" signals, time can be critical to an analysis.

The time-dependent Fourier transform computed by the *specgram* function is one solution to this problem. By applying a DFT to a sliding window in the time domain, *specgram* captures a signal's frequency content at different times. The disadvantage of this method is that it is *uniform* on all time intervals: it does not adjust to local idiosyncrasies in the frequency content. As a result, more subtle nonstationary characteristics of a signal can go undetected.

Wavelet analysis uses a more adaptable method, and is capable of revealing trends, breakdown points, discontinuities in higher derivatives, and self-similarity that the DFT might miss.

A wavelet is a waveform of effectively limited duration that has an average value of zero.

The *discrete wavelet transform* (DWT) computes *coefficients of similarity* between a signal and a sliding wavelet. The coefficients are found with wavelets of different *scales* (widths that approximate different frequencies) to analyze the signal at different resolutions. In a wavelet analysis, a signal is iteratively decomposed into the sum of a lowpass *approximation* and a progressive sequence of highpass *details*.

The Wavelet Toolbox adds wavelet analysis techniques to the signal processing techniques available in the Signal Processing Toolbox.

```
wavemenu
```

brings up a menu for accessing graphical tools in the Wavelet Toolbox.

Try:

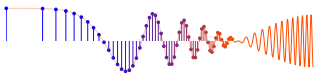
```
wavemenu
```

Select **Wavelet 1-D**
and then **File → Example Analysis → Basic Signals → Frequency breakdown**

S is the signal
a_5 is the approximation
d_n are the details

What is happening in this signal and how does the wavelet analysis show it?

(The material in this lab handout was put together by Paul Beliveau and derives principally from the MathWorks training document "MATLAB for Signal Processing", 2006.)



Pre-Lab and Warm-Up: You should read at least the Pre-Lab and Warm-up sections of this lab assignment and go over all exercises in the Pre-Lab section before going to your assigned lab session.

Verification: The Warm-up section of each lab must be completed **during your assigned Lab time** and the steps marked *Instructor Verification* must also be signed off **during the lab time**. One of the laboratory instructors must verify the appropriate steps by signing on the **Instructor Verification** line. When you have completed a step that requires verification, simply demonstrate the step to the TA or instructor. Turn in the completed verification sheet to your TA when you leave the lab.

Lab Report: It is only necessary to turn in a report on Section 4 with graphs and explanations. You are asked to **label** the axes of your plots and include a title for every plot. In order to keep track of plots, include your plot *inlined* within your report. If you are unsure about what is expected, ask the TA who will grade your report.

1 Introduction & Objective

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.

2 Background

2.1 The Fast Fourier Transform

Suppose `g` is an array of N values representing the time signal $g[n] = g(nT_s)$. The MATLAB command

```
>> G = fft(g);
```

causes MATLAB to compute the discrete Fourier transform of the time signal $g(nT_s)$ and place the result in array `G`. The array `G` represents a spectrum $G(n\Delta f)$, also of N values. Remember that MATLAB numbers its array elements starting with one. This means that $g[0]$ is stored in array element `g(1)` and $g((N-1)T_s)$ is stored in `g(N)`. Similarly, $G(0)$ is stored in array element `G(1)` and $G((N-1)\Delta f)$ is stored in `G(N)`. For greatest computational efficiency, N should be a power of two. If $g(nT_s), n = 0, ..., N-1$ is a time signal, the Fourier transform that MATLAB calculates is given by

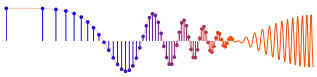
$$G(k\Delta f) = \sum_{k=0}^{N-1} g(nT_s)e^{-j2\pi kn/N}, \quad n = 0, ..., N-1. \tag{1}$$

Note that T_s represents the time between values of $g(kT_s)$. It turns out that the frequency interval Δf between values of $G(n\Delta f)$ is given by

$$\Delta f = \frac{1}{NT_s}$$

Given the array `G`, the MATLAB command

```
>> g = ifft(G);
```



calculates the inverse Fourier transform given by

$$g(nT_s) = \frac{1}{N}\sum_{k=0}^{N-1} G(k\Delta f)e^{+j2\pi nk/N}, \quad k = 0, ..., N-1. \tag{2}$$

The Fourier transform given by equation 1 is called a discrete Fourier transform or DFT. MATLAB uses the discrete transform because MATLAB cannot store continuous-time signals. MATLAB uses an efficient algorithm called the Fast Fourier Transform (FFT) to calculate the discrete Fourier transform. The discrete Fourier transform has properties that are similar to those of the familiar continuous Fourier transform. There is one important difference. The spectrum $G(k\Delta f)$ defined in equation 1 above is periodic in frequency with period $f_s = N\Delta f$. This periodicity is a consequence of the discrete-time nature of the time signal $g(nT_s)$. One period of the spectrum extends from frequency 0 to frequency $(N-1)\Delta f$. The positive frequency components lie between frequency 0 and frequency $(\frac{N}{2}-1)\Delta f$. The spectral components from frequency $N\Delta f/2$ to $(N-1)\Delta f$ are repeats of the negative frequency components that lie between frequencies $-(\frac{N}{2})\Delta f$ and $-\Delta f$ respectively. Because the spectrum $G(k\Delta f)$ is defined only at discrete values of frequency, the FFT algorithm considers the time function $g(nT_s)$ to be periodic with period NT_s. Consequently, the N-value array g you define will be interpreted as one period of an infinite-duration periodic signal. The spectrum $G(k\Delta f)$ defined in equation (1) is actually the Fourier transform of the periodic signal.

2.2 Plotting the Spectrum

If $G(k\Delta f)$ is plotted against frequency, zero hertz will appear on the left of the graph. The positive frequency components will appear to the right of zero, followed farther to the right by the negative frequency components. Because this is contrary to convention, MATLAB provides a function to rearrange the components of the array `G` to place the negative frequency components to the left of zero. The command

```
>> H = fftshift(G);
```

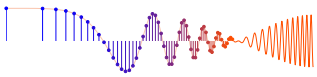
will create an array `H` that represents a spectrum $H(n\Delta f)$ whose DC component is in the center as expected. Before plotting `H` (or `G`), recall that these arrays may contain complex numbers. The command `plot(H)` will cause MATLAB to plot the imaginary part against the real part. This usually gives an interesting graph, but probably not the one you had in mind! You may obtain the magnitude spectrum by the command `M = abs(H)`, and the angle spectrum by `a = angle(H)`. You may also want to use the commands `real()` and `imag()` to find the real and imaginary parts of the signals you are examining. Tip: if `H` is real, plot it. If `H` is complex, plot `abs(H)` and `angle(H)`.

3 Pre-Lab

Sketch the Fourier Transform for each of the signals given in the procedure. Record them in your lab notebook and bring a photocopy of your notebook to the lecture before lab.

4 Lab Exercises

Because of the requirement that the number of samples be a power of two, we will let all of the time signals in this lab project consist of $N = 512$ samples having a total duration of $500\mu s$. (What does this make T_s? What is Δf?) You can generate a time axis and a frequency axis for your graphs by



```
>> tt = linspace(0,500e-6 - Ts,N);
>> ff = linspace(-(N/2)*deltaf,((N/2)-1)*deltaf,N);
```

Don't forget to do

```
>> G = fftshift(G);
```

so the spectrum matches the values on the axis.

1. A discrete time "unit impulse" is defined by the time signal

$$\delta[n] = \begin{cases} 1 & n = 0 \\ 0 & \text{otherwise} \end{cases}$$

   Let the time signal be a unit impulse. (Remember that $N = 512$.) Compute and plot the spectrum. Verify that your spectrum is correct by evaluating equation (1) by hand. MATLAB Hint: If you type:

   ```
   set(gcf, 'PaperPosition', [0.5, 0.5, 7.5,1 0])
   ```

   before printing, your plots will be better spaced on the page.

2. Let the time signal be $g[n] = g(nT_s) = 1$. Compute and plot the spectrum. Verify that your spectrum is correct by substituting the spectrum you obtain into equation (2) and showing by hand that you obtain $g(nT_s)$ back again.

3. Let the time signal be a single pulse extending from $t = -16T_s$ to $t = 16T_s$. (Remember, the time signal is interpreted as periodic!). Compute and plot the spectrum. Verify that it is correct by comparing with the conventional Fourier transform of a continuous-time pulse.

4. Let the time signal be the pulse of Step 3 above, but extending from $t = -32T_s$ to $t = 32T_s$. Compare its spectrum with the spectrum obtained in Step 3.

5. Let the time signal be a cosine of amplitude one whose frequency is chosen so that it has exactly 32 cycles in $500\mu s$. Compute and plot the spectrum. Now verify your result by substituting the spectrum you obtain into equation (2) and showing that you recover the original cosine. (This is much easier than substituting a cosine into equation (1) to verify the spectrum.)

6. Let the time signal be a cosine of amplitude one whose frequency is 65 kHz. Compute and plot the spectrum. Compare the result with the spectrum you obtained in Step 5 above.

7. Let the time signal be an "RF pulse" of frequency 64 kHz and duration $64T_s$.

Instructor Verification (separate page)

5 Report

Include the required spectra from Steps 1 through 7. Verify the correctness of your results as requested. Your report need not contain very much writing, but be sure that what you do write is correct, supports or explains the graphical data, and uses good English. Only one per group needs to do the report.