EE477 Digital Signal Processing

Spring 2006

Lab #11

Using a Fast Fourier Transform Algorithm

Introduction

The symmetry and periodicity properties of the discrete Fourier transform (DFT) allow a variety of useful and interesting decompositions. In particular, by clever grouping and reordering of the complex exponential multiplications it is possible to achieve substantial computational savings while still obtaining the exact DFT solution (no approximation required). Many "fast" algorithms have been developed for computing the DFT, and collectively these are known as Fast Fourier Transform (FFT) algorithms. Always keep in mind that an FFT algorithm is not a different mathematical transform: it is simply an efficient means to compute the DFT. In this experiment you will use the Matlab `fft()` function to perform some frequency domain processing tasks.

The statement about doing "real time" processing with an FFT algorithm is not really true, since the DFT requires an entire block of input samples to be available before processing can begin. So, rather than being able to run the processing algorithm on the fly as each waveform sample arrives as we might with a direct form digital filter, it will be necessary to buffer a block of samples and then start the first FFT while still receiving the new samples as they arrive from the A/D and providing the output samples to the D/A. This inherent delay, or processing *latency*, is separate from the time required to compute the FFT itself. The Matlab implementation will be run on stored data anyway, so the real time processing latency is not an issue here.

Short-time Fourier transform (STFT)

One interesting use of the FFT is to implement linear time-invariant systems. The idea is to break the input signal into blocks, perform the FFT on each block, multiply by a filter function in the frequency domain, then IFFT to reconstruct the filtered time domain signal. Because the FFT provides the means to reduce the computational complexity of the DFT from order (N^2) to order ($N \log_2(N)$), it is often feasible to do FFT-based processing for DSP systems. Even with the forward and inverse transform overhead, the computational cost of doing both the FFT and IFFT may be lower than doing the equivalent computation with conventional time domain methods.

The DFT is a frequency-sampled version of the Fourier transform, so multiplying the DFT by a filter function in the frequency domain is actually the equivalent of *circular* convolution, not linear convolution. This means that the resulting time domain signal may have "time domain aliasing" if the effects of the circular overlap are not accounted for. Refer to an authoritative DSP textbook for the details of this issue.

Nevertheless, for this experiment we are going to use a somewhat crude short-time processing algorithm. The concept is as follows.

(1) We will segment the input signal into overlapping blocks. The overlap will be 50%, and each block will be "*windowed*" by a smooth raised-cosine function. The window will be chosen so that the original signal can be reconstructed perfectly if no signal modification is done (see Figure 1).
(2) For each windowed block, the FFT will be calculated. This gives a spectral "snapshot" of what is going on during that short block of the input signal.
(3) We can now do some modification of the FFT data, such as multiplying by a spectral shape (filter) or some other type of frequency-domain processing. Assuming that we had a real-only input signal and we want a real-only output signal, we need to make sure that whatever manipulation is applied gets done in a way that will keep the complex DFT data in *conjugate symmetric* form. That is, if we change anything in the DFT bins between 0 and $N/2$, the same change needs to apply to the mirror image bins N-1 down to $N/2$, and the resulting modified DFT data must remain conjugate symmetric.
(4) After the spectral processing, the IFFT (inverse FFT) is calculated. The resulting block is then *overlap-added* to the output buffer, thereby reconstructing the desired signal.

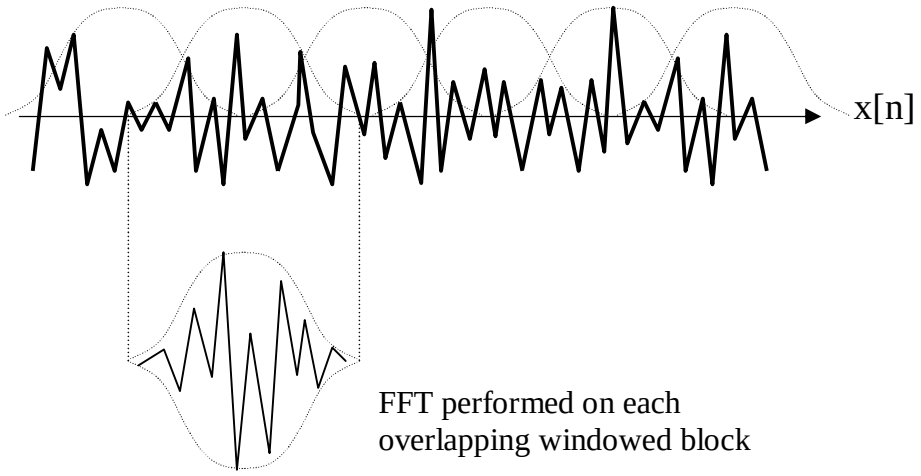

Figure 1: *Signal segmentation and overlap-add reconstruction.*

Why is it necessary to window and overlap the analysis blocks? There are several issues here. It is theoretically possible to do a huge FFT over the entire input signal, but for practical reasons we usually want to limit the time delay and storage memory of a real time system. Breaking the signal into shorter blocks provides this opportunity. Applying the smooth window function is helpful in reducing the truncation effects that otherwise would be evident in the DFT data, but this may or may not be important depending on the application. Finally, the overlapping

windows provide a smooth transition from one block to the next, and this is important if the frequency domain processing varies with time (see Exercise C below).

The choice of FFT length, overlap amount, window shape, etc., can be made with a solid theoretical basis. Consult a DSP text book for more information on the theoretical underpinnings of the short-time Fourier transform.

In a practical DSP system designed for real time operation, the STFT skeleton could be organized as shown in Figure 2. In the "background," the interrupt service routines (ISRs) collect the input data into a buffer, and play the processed data from an output buffer. In the "foreground" the FFT routine waits until a block of input data has arrived, then performs the window/FFT/modify/IFFT/overlap-add sequence for that block. The foreground routine then waits for the 50% overlap period, and runs again on the overlapped data. This process continues over and over as long as the program is running.

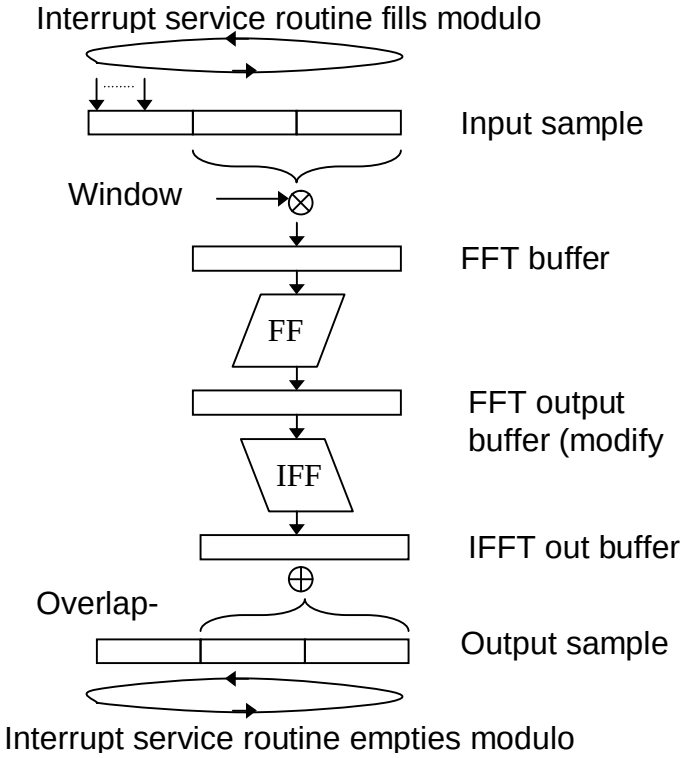

Figure 2: *STFT skeleton program flow.*

The application code would be placed in between the FFT and the IFFT. Your program would then have access to the complex DFT data for each block.

The STFT process in Matlab will be similar, except the input and output buffers will not have interrupt service routines filling and emptying them. Instead, you should write an STFT processing function. The outline of such a function is given here.

```
function xx=block_fft(in,fftlen)
% block_fft(in, fftlen)
% Calculates windowed FFT (length fftlen, with 50% overlap),
%  then inverse FFT with overlap-add.  If not processing is
inserted
%  into the function below, the output should be essentially
identical
%  to the input.
%
% in  = input data vector
% fftlen = desired fft length (e.g., 1024)
% xx  = output data vector
in=in(:);
% Create output buffer for overlap add
xx=zeros(1,length(in));

% Create length "fftlen" raised cosine window function
wind=0.5*(1-cos(2*pi*(0:fftlen-1)/fftlen));

% Stride through the input with 50% overlap (fftlen/2),
% calculate windowed length "fftlen" FFT of each block,
% then inverse transform and overlap-add.
for i=1:fftlen/2:(length(in)-fftlen)
    ff=fft(wind.*in(i:(i+fftlen-1))',fftlen);
% User processing of FFT data would go here…
    xx(i:(i+fftlen-1))= xx(i:(i+fftlen-1))+ ifft(ff,fftlen);
end
```

ð Exercise A: Look at FFT data

Using Matlab, show plots of the FFT magnitude and phase for the following signals.

Explain the results to the lab instructor (instructor check off A).

```
xx = [1 zeros(1,1023)];          (length 1024 FFT)

yy = [0 1 zeros(1,1022)];        (length 1024 FFT)

aa = 3*sin(2*pi*(0:1023)/64);    (length 1024 FFT)

bb = 3*sin(2*pi*(0:1023)/64);    (now use a length 4096 FFT)
```

Questions to think about:

- How do the numerical results compare to your theoretical expectations?
- What does the Matlab function `unwrap()` do, and how might that be helpful when interpreting the phase results?
- If the FFT magnitude is zero at a particular index, is the phase meaningful?

ð Exercise B: Run FFT/IFFT pass program

Create your own copy of the `block_fft()` function given in the introduction (see the course web site to download the text file).

Test you code with several different input signals. The skeleton should act like a "pass" program: the input data should pass through the window/FFT/IFFT/overlap-add procedure without deliberate modification. Verify that this is working.

Finally, insert some Matlab instructions in the "`% User processing of FFT data would go here`…" position to change the *gain* of the pass program: multiply the complex FFT data by 2.5. Comment on the results (instructor check off B).

Once you have the pass program with gain 2.5 working, take out the 2.5 multiply before working on Exercise C.

ð Exercise C: STFT for signal processing

Now that the basic pass program is working, you can consider some more interesting STFT-based processing. In this part you will do some signal-dependent processing. Since the DFT gives a complex view of the input signal's short-time spectrum, we can take advantage of the spectral analysis to do some signal enhancement.

It is common to have an input signal that is contaminated with unwanted broadband noise. One way to reduce the undesired noise is to use a spectral *threshold*. The algorithm compares the spectral magnitude in each FFT bin to a threshold value. If the magnitude is above the threshold, it is assumed to be "signal" and gets passed unaltered. On the other hand, if the measured magnitude in an FFT bin is below the threshold, it is assumed to be noise and the bin is set to zero. If the threshold is chosen carefully, the output signal will have less audible noise than the input signal. This process is known as a "de-hisser" or a spectral "noise gate."

In order to do the threshold test you will need to add another parameter to the `block_fft()` function: `xx=block_fft(in,fftlen,thresh)`, and use the threshold and the FFT magnitude to take out the FFT bins that are below the threshold. You will also need to experiment with various threshold values, and try different types of input signals to verify that your de-hisser is working. Demonstrate your code by choosing an appropriate threshold value for the noisy speech signal on the course web site (instructor check off C).

Check Off Sheet

Name _____

Date: _____

Exercise A: _____

Exercise B: _____

Exercise C: _____