ECE4703 B Term 2010 Laboratory Assignment 5

Project Code and Report Due by 3:00pm 09-Dec-2010

The goals of this laboratory assignment are:

- to develop an understanding of *frame-based* digital signal processing,
- to familiarize you with computationally efficient techniques that achieve performance gains not through compiler optimization or architectural optimization but, rather, through clever *algorithm optimization*,
- to allow you to experimentally verify *asymptotic complexity* predictions, and
- to allow you to apply the FFT to a practical signal parameter estimation problem.

1 Background

The problem of estimating the frequency of a noisy sinusoidal signal occurs in a variety of applications including radar and communications. In this assignment, we consider a system with a complex-valued discrete-time observation

$$z[n] = a\exp(j\Omega n + \phi) + w[n] = \underbrace{a\cos(\Omega n + \phi)}_{x[n]} + j\,\underbrace{a\sin(\Omega n + \phi)}_{y[n]} + w[n]$$

for $n = 0, 1, \ldots, N-1$ where $\Omega = \omega/f_s$ is the normalized frequency (radians/sec) that we wish to estimate, ϕ is the unknown phase (radians) of the observation at time $n = 0$, a is the amplitude of the complex exponential, $w[n]$ is complex additive noise, and N is the total number samples in the observation. In typical scenarios with uncorrelated zero-mean Gaussian distributed complex noise, the *maximum-likelihood* frequency estimator is

$$\hat{\Omega}_{\mathrm{ML}} = \arg\max_{\lambda\in[0,\pi)} |A(\lambda)|^2 \tag{1}$$

where

$$A(\lambda) = \sum_{n=0}^{N-1} z[n]\exp(-j\lambda n) \tag{2}$$

Note that $A(\lambda)$ can be efficiently computed for discrete $\lambda = \frac{k2\pi}{N}$ where $k = 0, \ldots, N-1$ using the N-point FFT when N is an integer power of two. These discrete frequencies, however, may not provide fine enough enough resolution for frequency estimates in some applications, hence it is often the case that the following four-step strategy is used to provide a more accurate approximation to the solution of (1):

1. Given N and $z[n]$ for $n = 0, \dots, N-1$, compute the N-point FFT

$$A[k] = A\left(\frac{k2\pi}{N}\right) = \sum_{n=0}^{N-1} z[n] \exp\left(-j\frac{k2\pi}{N}n\right) = \underbrace{B[k]}_{\text{real part}} + j \underbrace{C[k]}_{\text{imaginary part}} \tag{3}$$

   for $k = 0, \dots, N-1$.

2. Compute the squared magnitude of $A[k]$ as

$$U[k] = |A[k]|^2 = B^2[k] + C^2[k] \qquad k = 0, \dots, \frac{N}{2} - 1. \tag{4}$$

   Note that, by working with *squared* magnitudes, we avoid having to calculate any computationally intensive square roots. Also note that you only need to compute these squared magnitudes over the first half of $A[k]$ (why?).

3. Compute the *coarse frequency estimate* by finding the index of the maximum of $U[k]$, i.e.

$$\hat{k} = \arg \max_{k \in \{0, \dots, \frac{N}{2}-1\}} U[k]. \tag{5}$$

   Denote the coarse normalized frequency estimate as $\hat{\Omega}_{\text{coarse}} = \frac{2\pi\hat{k}}{N}$. Note that the coarse non-normalized frequency estimate is $\hat{\omega}_{\text{coarse}} = f_s \cdot \hat{\Omega}_{\text{coarse}}$.

4. Generate a *fine frequency estimate* using a refinement technique with the coarse estimate as the initial guess. In practice, a variety of refinement techniques can be used, e.g. Newton's method, the secant method, or the bisection method. In this assignment, we will use a relatively simple technique called *quadratic interpolation* to generate the fine frequency estimate. The basic idea is as follows. You provide three input/output pairs $\{x_1, y_1\}$, $\{x_2, y_2\}$, $\{x_3, y_3\}$, and compute the quadratic polynomial coefficients a, b, c such that

$$y_i = ax_i^2 + bx_i + c \tag{6}$$

   for $i = 1, 2, 3$. Intuitively, what you are doing here is finding a quadratic polynomial that interpolates the points around the maximum in the squared magnitude of the FFT as shown in Figure 1. The inputs are the normalized frequencies in the neighborhood of the maximum, i.e.

$$\{x_1, x_2, x_3\} = \left\{\frac{2\pi(\hat{k}-1)}{N}, \frac{2\pi\hat{k}}{N}, \frac{2\pi(\hat{k}+1)}{N}\right\}. \tag{7}$$

   The outputs are the squared FFT magnitudes at these frequencies, i.e.

$$\{y_1, y_2, y_3\} = \left\{U[\hat{k}-1], U[\hat{k}], U[\hat{k}+1]\right\}. \tag{8}$$

   Once you've determined the quadratic polynomial coefficients a, b, c that fit these input/output pairs, you can find the maximum of the function $f(x) = ax^2 + bx + c$ using standard calculus techniques. This maximum corresponds to our fine normalized frequency estimate, and can be expressed as

$$\hat{\Omega}_{\text{fine}} = \frac{-b}{2a} \approx \hat{\Omega}_{\text{ML}}. \tag{9}$$

   The fine non-normalized frequency estimate can then be computed as $\hat{\omega}_{\text{fine}} = f_s \cdot \hat{\Omega}_{\text{fine}}$.

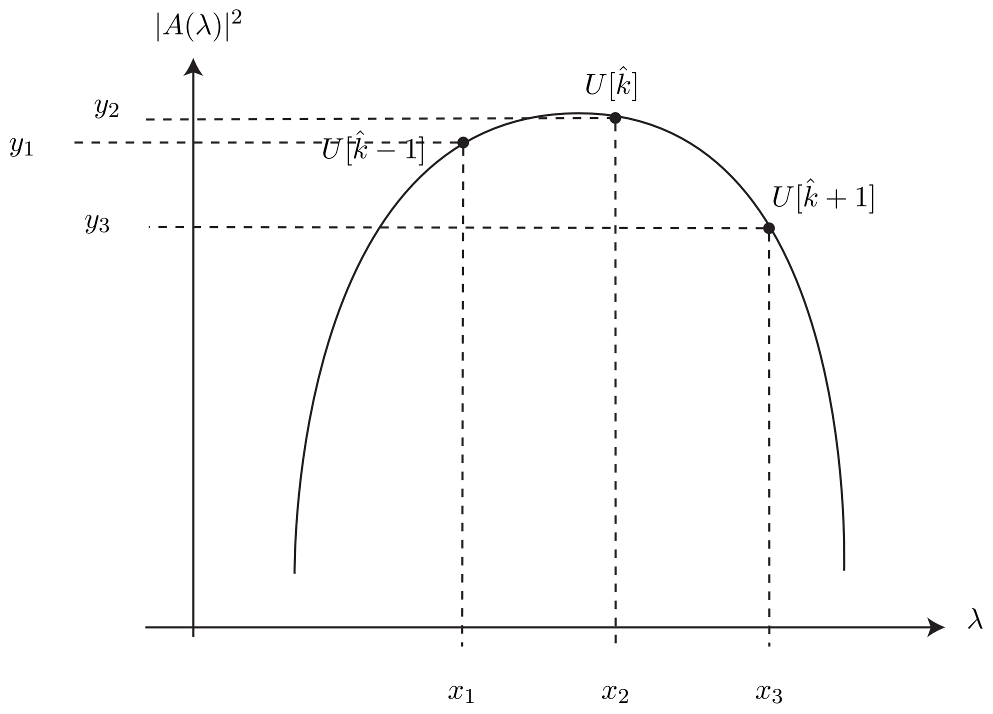

Figure 1: The principle of quadratic interpolation.

If done correctly, this procedure can yield very accurate results in which the RMS frequency error of the non-normalized fine estimates is much less than one part per million (depending on the length of the observation and the amount of background noise).

2 Problem Statement

In this assignment, you will perform frequency estimation in real-time by using TI's optimized FFT code to perform the first step of the maximum likelihood frequency estimator and then writing your own code to perform steps 2–4. All math in this assignment should be performed in single-precision floating point and all of your code will be written in C.

A suggested flow diagram for the assignment is shown in Figure 2. It is recommended that you use a double buffering technique here since all processing in this assignment is frame-based (your Kehtarnavaz textbook discusses a triple-buffering technique in Lab 6, but the third buffer can be eliminated since the FFT output overwrites the FFT input). Let the left channel represent the real part of each input sample ($x[n]$) and let the right channel represent the imaginary part of each input sample ($y[n]$). The (complex-valued) incoming samples are placed in one buffer (with $2N$ `float` elements) and the last complete frame of (complex-valued) samples is stored in another buffer (with $2N$ `float` elements). Your ISR will simply accept new input samples and fill up the incoming buffer; no processing other than incrementing a global index and checking for a full buffer is performed in the ISR. When the incoming buffer is full, you should swap buffers (the "incoming" buffer becomes the "complete" buffer, and vice versa) and begin estimating the frequency of the signal captured in the complete buffer in your main code. Note that, in step 1, the FFT will overwrite the current "complete" buffer. Your ISR will also be running while you perform the frequency estimation steps and will be filling up the new incoming buffer. To run in real-time, you need to complete all of your computations before the next incoming buffer is full. You do not need to send any output signals to the AIC23 codec.

2.1 Step 1

Due to the wide variety of applications for the FFT, TI provides an optimized linear assembly function to implement FFTs on the C6x. You will need three functions to use TI's optimized FFT routines. These functions are `cfftr2_dit`, `digitrev_index`, and `bitrev`. You should read the header comments in these files to make sure that you know how to use them correctly. Your FFT function call should work like:

```
void cfftr2_dit( float *z, const float *w, short N)

// z  Pointer to Array of Dimension 2*N elements holding
//    Input to and Outputs from the function
// w  Pointer to an array holding the complex twiddle factors
// N  Number of complex points in z
```

The input to the FFT is in the array z and the result of the FFT is returned in the array z (the input is overwritten by the FFT function). You should compute the sin/cos portions of the "twiddle factors" during the initialization part of your code (prior to the FFT function call) and store them in W. You will probably want to include the header file `math.h` to allow for pre-computation of the sin and cos terms needed in for the complex "twiddle-factors". All input/output arrays and twiddle factors should be single-precision floating point datatypes.

You should confirm that this part of your code is working correctly before proceeding. One way to do this is just provide some known input to the FFT function and compare the output to Matlab.

2.2 Step 2

This step should be straightforward.

2.3 Step 3

This step should also be straightforward. Note that $U[k]$ is a multimodal function, hence there is no efficient way to find the maximum except by brute-force search. The key here is to find the index of the maximum, i.e. to find $\hat{k}$.

2.4 Step 4

You may need to do a little bit of research on quadratic interpolation to determine an efficient algorithm for performing this step. You should be able to compute the quadratic polynomial coefficients $\{a, b, c\}$ using standard operations like addition, subtraction, multiplication, and division. There should be no need for any fancy calculations here like a square root. You are strongly encouraged to test this function independently by picking your own $\{a, b, c\}$ polynomial coefficients, generating three input/output pairs $\{x_1, y_1\}$, $\{x_2, y_2\}$, $\{x_3, y_3\}$, passing these into your function,

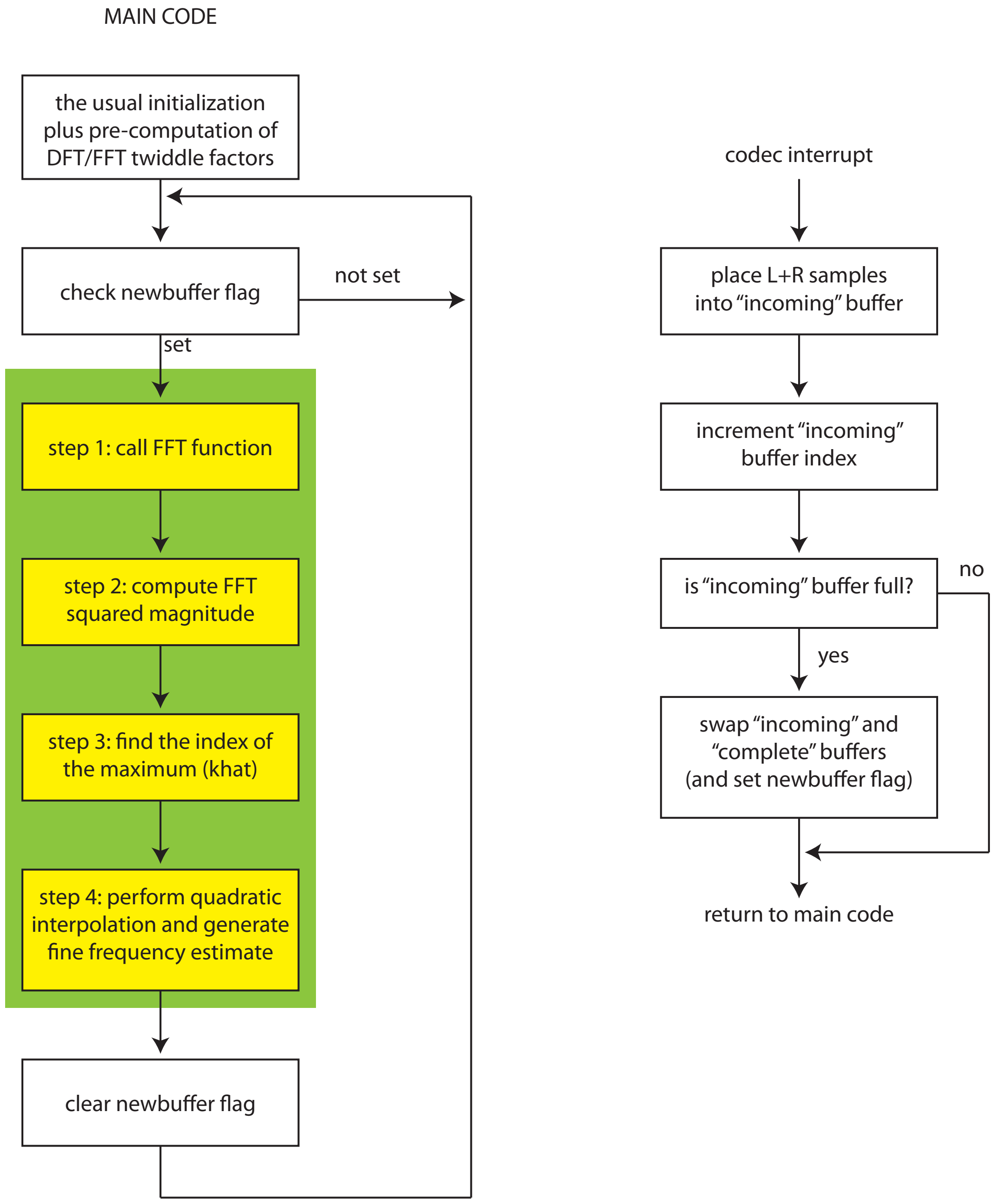
MAIN CODE
the usual initialization plus pre-computation of DFT/FFT twiddle factors
check newbuffer flag
not set
set
step 1: call FFT function
step 2: compute FFT squared magnitude
step 3: find the index of the maximum (khat)
step 4: perform quadratic interpolation and generate fine frequency estimate
clear newbuffer flag
codec interrupt
place L+R samples into "incoming" buffer
increment "incoming" buffer index
is "incoming" buffer full?
no
yes
swap "incoming" and "complete" buffers (and set newbuffer flag)
return to main code

Figure 2: Suggested logical flow of your frequency estimation code code. The green box represents the overall frequency estimation function and the yellow boxes represent the specific steps (which you put into subfunctions for testing if you like).

and confirming that your function returns the same quadratic polynomial coefficients that you chose.

2.5 Verification and Testing

Since part of this assignment is to explore the complexity trends of each step, you should use a sampling frequency of 8 kHz to allow for testing at larger values of N. To test that your frequency estimator is working correctly, you can use the following Matlab code to generate a 1 kHz complex exponential signal with some moderate background noise:

```
fs = 44100;        % sampling frequency
f = 1000;          % frequency of tone
p = 2*pi*rand;     % random phase of tone
s2 = 0.01;  % variance of the additive noise
T = 5;             % duration
t = [0:1/fs:T]';  % time vector
x = cos(2*pi*f*t+p)+sqrt(s2)*randn(size(t));  % real part
y = sin(2*pi*f*t+p)+sqrt(s2)*randn(size(t));  % imaginary part
m = max([x;y]);
x = 0.99*x/m;      % normalize to avoid clipping
y = 0.99*y/m;      % normalize to avoid clipping
sound([x y],fs);
```

Feel free to adjust these parameters to see how they affect the accuracy of your frequency estimator. You should not increase the frequency of the tone much beyond 3 kHz since the anti-aliasing filter on the DSK will block frequencies close to and above 4 kHz.

You may use the `printf` function to verify that your frequency estimates are accurate but please remember that this will prevent real-time operation of your code. You should probably display the non-normalized estimates since those will correspond to the actual frequency of the analog signal. Do not use the `printf` function when profiling your code and when determining the largest possible value of N that will run in real-time.

3 In Lab

You will work with the same lab partner as in the prior laboratory assignments. Please contact the instructor if your lab partner has dropped the course or if you have concerns about your lab partner's performance on the prior assignment.

4 Code Submission and Specific Items to Discuss in Your Report

Your code will be tested for correct functionality and profiled by the grader for select values of N to determine if the profiling results in your report are accurate. Please be sure to write structured C code and to comment your code liberally to facilitate testing by the grader.

You should test your frequency estimator at different values of N to determine how the length of the observation and the number of FFT points affect the accuracy of the frequency estimates. Each frequency estimate will be a little different from the last — this is caused by the noise and is to be expected. Nevertheless, you should be getting very accurate and closely repeatable frequency estimates for even moderate values of N as long as you don't add too much noise to the signal.

In addition to verifying the accuracy of your frequency estimator, you should profile each of the four steps in your frequency estimator for increasing values of N, up to the point where the code no longer runs in real time. Your double-buffer arrays might get very big, in which case you should put your data in external memory since it will allow much larger arrays than the internal memory. If the arrays get too big to store in external memory, you can increase the sampling rate to reduce the available time for computation. How large can you make N before real-time operation is no longer possible? What are the complexity trends of each step? You should be able to clearly demonstrate the $N \log_2(N)$ complexity trend of the FFT, but what about the other steps? What is the complexity trend in computing the squared magnitudes (step 2)? What is the complexity trend in the brute-force search for the maximum (step 3)? What is the complexity trend in the quadratic interpolation step (step 4)? Can you explain your profiling results? You should plot your profiling results to demonstrate the trends clearly and discuss what parts of the frequency estimator actually set the limit on N for real-time operation. In other words, is there a particular step in the frequency estimator that dominates the overall complexity at large N? Your plots should include distinct line types and clearly labeled legends. You may want to try generating semilog or loglog plots to see if that makes your results easier to interpret.