# ECE4703 Laboratory Assignment 5

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 reduce computational load 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 processing problem.

## 1 Problem Statement

In this assignment, you will develop a frame-by-frame "overlap and save" fast convolution method to perform FIR filtering and compare your results to standard sample-by-sample FIR filtering. All math in this assignment should be performed in single-precision floating point and all of your code will be written in C. The number of coefficients in your FIR filter is denoted as $M$. To allow testing at larger values of $M$, the sampling rate for all parts of this assignment should be set to $f_s = 8\text{kHz}$.

As discussed in lecture, all frame-based processing should be performed by a function called from `main()`. The ISR should only read input samples, fill buffers, set flags to indicate a new buffer has been filled, and write output samples. To run in real-time, you need to complete all of your computations before the next incoming buffer is full.

## 2 Part 1: Sample-by-Sample FIR Filtering

In this part of the assignment, you can re-use your code from Laboratory Assignment 2 to implement a sample-by-sample FIR filter with single-precision floating point arithmetic. Generate several FIR filters (lowpass or bandpass) with different numbers of coefficients, test your filters to ensure they are working correctly, and profile your FIR filter code. Experimentally determine how large $M$ can be before your FIR filter no longer runs in real-time for both unoptimized code and optimized code, i.e., with and without compiler optimization. With optimization, you should be able to achieve realtime operation of your FIR filter with $M$ at least equal to 100 coefficients (probably more like 400-500 coefficients). Plot your profiling results as a function of $M$ for several different values of $M$ up to the value of $M$ where the FIR filter no longer runs in real-time for both unoptimized code and optimized code. Do your results agree with the asymptotic complexity analysis?

# 3 Part 2: Frame-by-Frame "Overlap and Save" Fast Convolution FIR Filtering

In this part of the assignment, you will create a new project to perform FIR filtering using frame-by-frame "overlap and save" fast convolution using the technique described in lecture and in the Lyons textbook handout. A possible flow diagram for the assignment is shown in Figure **??**. You will need at least the following buffers:

- Filter coefficients ($N$ COMPLEX)
- Input "filling" buffer ($N$ COMPLEX)
- Input "processing" buffer ($N$ COMPLEX)
- Output buffer ($N$ float)

You may want to consider having two output buffers, analogous to the filling and processing input buffers. Even though the filter coefficients and input samples are real-valued, the FFT code you will use requires the input arrays to be formatted as COMPLEX, which we define as

```
// complex typedef
typedef struct {
  float re,im;
} COMPLEX;
```

Recall that you must select the FFT length $N$, which should be an integer power of two. A good starting choice for $N$ is approximately $4M$, where $M$ is the number of filter coefficients. Recall also that the input buffer length is determined as $K = N - (M - 1)$.

## 3.1 Initialization and Using the TI FFT

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`. The typical usage is

```
digitrev_index(iw,N/RADIX,RADIX); //produces index for bitrev() W
bitrev(w,iw,N/RADIX);      //bit reverse W
cfftr2_dit(x,w,N) ;      //TI floating-pt complex FFT
digitrev_index(ix, N, RADIX);     //produces index for bitrev() X
bitrev(x,ix,N);      //freq scrambled->bit-reverse X
```

Note that the first two lines only need to be run once in order to get the twiddle factors in the bit-reversed order expected by the TI FFT function. It is probably a good idea to take care of that in the initialization part of your code.

You should read the header comments in the provided FFT files to make sure that you know how to use them correctly. The FFT function is called like this:

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

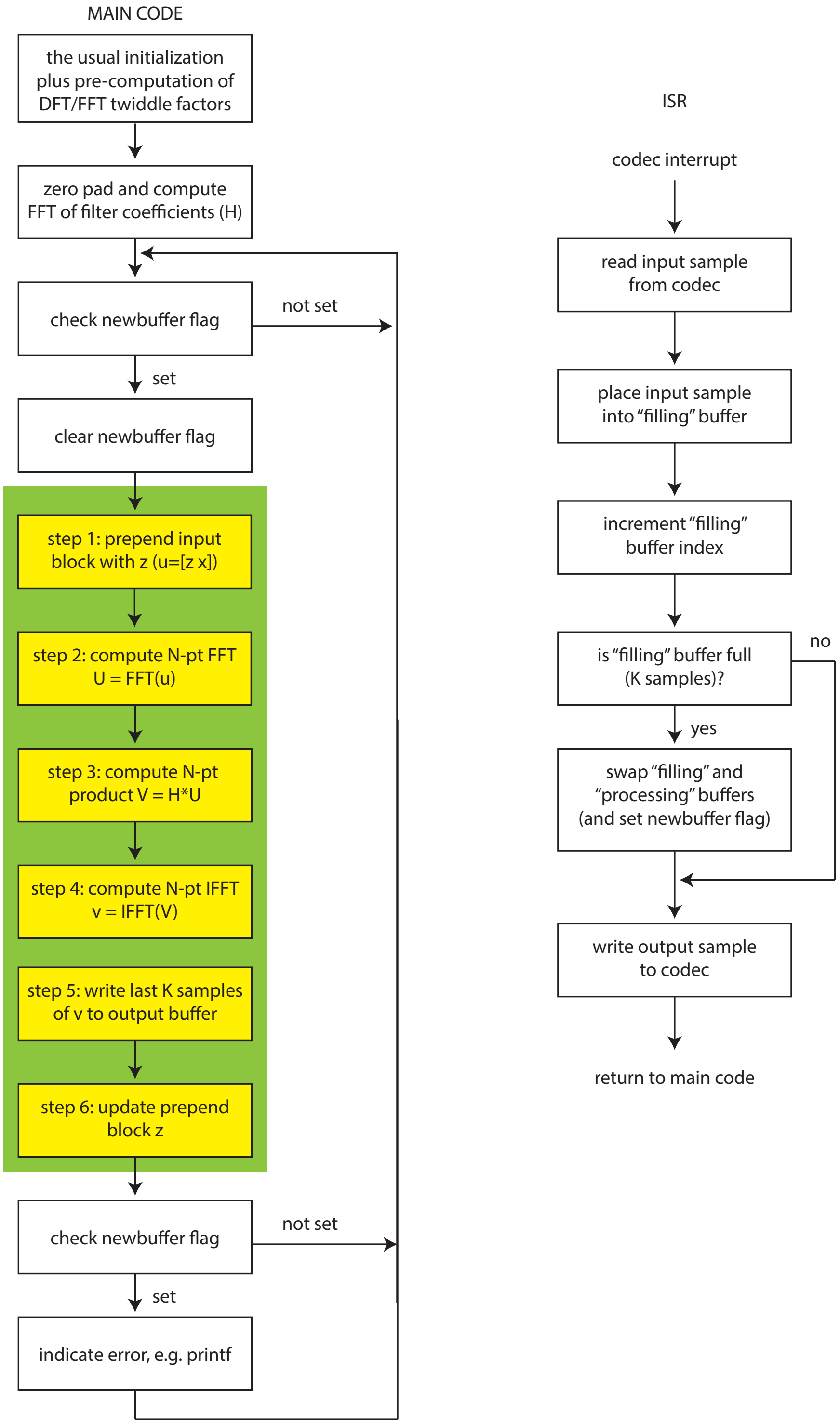
MAIN CODE
the usual initialization plus pre-computation of DFT/FFT twiddle factors
zero pad and compute FFT of filter coefficients (H)
check newbuffer flag
not set
set
clear newbuffer flag
step 1: prepend input block with z (u=[z x])
step 2: compute N-pt FFT U = FFT(u)
step 3: compute N-pt product V = H*U
step 4: compute N-pt IFFT v = IFFT(V)
step 5: write last K samples of v to output buffer
step 6: update prepend block z
check newbuffer flag
not set
set
indicate error, e.g. printf
ISR
codec interrupt
read input sample from codec
place input sample into "filling" buffer
increment "filling" buffer index
is "filling" buffer full (K samples)?
no
yes
swap "filling" and "processing" buffers (and set newbuffer flag)
write output sample to codec
return to main code

Figure 1: Suggested flow of your fast linear convolution code. The green box represents the fast linear convolution function that you should profile for different values of $M$ until the code no longer runs in real-time.

```
// x  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 $x$ and the result of the FFT is returned in the array $x$ (the input is overwritten by the FFT function). 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". You should compute the sin/cos portions of the "twiddle factors" during the initialization part of your code (prior to the first FFT function call) and store them in $W$. All input/output arrays and twiddle factors should be single-precision floating point datatypes.

Prior to processing any input buffers, your code should initialize the length $M-1$ prepending buffer $z$ to be all zeros, zero-pad the FIR filter coefficients $h$ to be a length-$N$ buffer, and compute the length-$N$ buffer of frequency-domain filter coefficients $H$. You should confirm that this part of your code is working correctly before proceeding. One way to do this is to compute the FFT of your (zero padded) filter coefficients in MATLAB and compare them to the coefficients in a CCS watch window.

### 3.2 Step 1

Once an input block is ready to be processed, we must prepend the length-$K$ input buffer $x$ with the length $M-1$ prepend buffer $z$ to generate the length-$N$ buffer $u = [z, x]$. The first time through, $z$ will be all zeros. If you are clever, you can achieve this prepending efficiently by declaring the buffer $z$ to be allocated in memory just before the buffer $x$. In fact, you may want to have two prepending buffers, one for each input buffer. The prepending buffer(s) $z$ should be of type COMPLEX.

### 3.3 Step 2

You can re-use the code you wrote to compute the FFT of the filter coefficients here, substituting $u$ for $h$. If you have tested your FFT code on the filter coefficients and are sure it is working correctly, then this code should also be correct. When this step completes, you should have the result $U = \text{FFT}(u)$, where $U$ is a length-$N$ buffer of type COMPLEX.

### 3.4 Step 3

In this step, you should perform element-by-element complex multiplication with $V_i = H_i \cdot U_i$ for $i = 0, \ldots, N-1$. Recall that the product of two complex numbers

$$(a + jb)(c + jd) = (ac - bd) + j(bc + ad).$$

The result $V$ will be a length-$N$ buffer of type COMPLEX.

### 3.5 Step 4

In this step, you compute $v = \text{IFFT}(V)$. Refer to the comments in TI's FFT function to determine how to perform an IFFT using the same FFT function. There are several options, some of which may be more efficient. After the IFFT has been performed, the imaginary elements of the result $v = \text{IFFT}(V)$ should all be very close to zero. If you are seeing significant imaginary components in $v$, then there is a bug somewhere in your code.

### 3.6 Step 5

Recall that we must discard the first $M-1$ samples of $v$ to generate the desired output. Alternatively, we can copy the last $K$ samples of $v$ to the length-$K$ output buffer $y$.

### 3.7 Step 6

The last step is to update the length $M-1$ prepending buffer $z$. Here we copy the last $M-1$ samples of $x$ to $z$.

### 3.8 Verification and Testing

Confirm your fast convolution code gives the same results as your direct convolution code. You can do this by testing your filtering with white noise as usual or by (for small values of $M$) running both the frame-by-frame and the sample-by-sample code in the same project and comparing the outputs.

Experimentally determine how large $M$ can be before your frame-by-frame code no longer runs in real-time for both unoptimized code and optimized code, i.e. with and without compiler optimization. Profile the CPU cycles of your fast linear convolution function for several different values of $M$ up to the value of $M$ where the function no longer runs in real-time for both unoptimized code and optimized code. Try different values of $N$ to determine any tradeoffs. Do you agree with Lyons' statement that fast convolution can be more efficient than direct FIR filtering when the filter order is larger than 40 to 80? Do your results agree with the asymptotic complexity analysis?

## 4 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.

## 5 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 $M$ 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.

In addition to verifying the correct operation of your code, you discuss and compare your profiling results for direct FIR filtering and fast convolution FIR filtering, with and without optimization. Can you explain your profiling results? You should plot your profiling results to demonstrate the trends clearly and discuss what parts of your fast convolution code dominate the overall complexity at large $M$. Try to fit the asymptotic complexity curves to the actual profiling results to see how closely the actual computational burden tracks the predicted complexity. Recall that, if an algorithm has asymptotic complexity $f(N)$, the number of cycles will look like $cf(N)$ as $N$ becomes large, where $c > 0$ is a constant. You can determine $c$ either by a least-squares fit or by trying different values of $c$ until the fit looks good to your eye. 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.