Real-Time Fast Fourier Transform

Chapter 9

Introduction

The Fourier transform is a standard system analysis tool for viewing the spectral content of a signal or sequence. The Fourier transform of a sequence, commonly referred to as the discrete time Fourier transform or DTFT is not suitable for real-time implementation. The DTFT takes a sequence as input, but produces a continuous function of frequency as output. A close relative to the DTFT is the discrete Fourier transform or DFT. The DFT takes a finite length sequence as input and produces a finite length sequence as output. When the DFT is implemented as an efficient algorithm it is called the Fast Fourier Transform (FFT). The FFT is ultimately the subject of this chapter, as the FFT lends itself to real-time implementation. The most popular FFT algorithms are the radix 2 and radix 4, in either a decimation in time or a decimation in frequency signal flow graph form (transposes of each other)

DTFT, DFT, and FFT Theory Overview

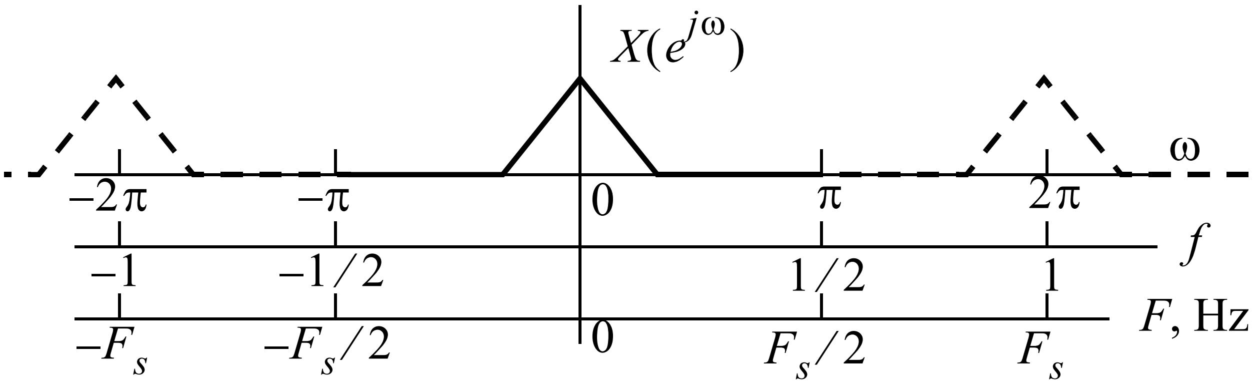
Frequency domain analysis of discrete-time signals begins with the DTFT, but for practicality reasons moves to the DFT and then to the more efficient FFT.

The DTFT

- Recall that the DTFT of a sequence $x[n]$ is simply the sum

$$X(e^{j\omega}) = X(\omega) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n} \tag{9.1}$$

- A characteristic of the spectrum $X(e^{j\omega})$ is that it is a periodic function having period 2π
- The frequency axis in (9.1) is in radians per sample, which is a normalized frequency variable
- If $x[n]$ was acquired by sampling an analog signal every T seconds, e.g., $x[n] = x_a(nT)$, then the ω frequency axes is related to the normalized digital frequency $f = \omega/(2\pi)$ and the analog frequency $F = f \cdot F_s$ as follows:



The DFT and its Relation to the DTFT

- The DFT assumes that the sequence $y[n]$ exists only on a finite interval, say $0 \le n \le N-1$, and zero otherwise
- This may occur as a result of data windowing, e.g.,

$$y[n] = x[n]w[n] \tag{9.2}$$

where $w[n]$ is a window function of duration N samples and zero otherwise

– Unless defined otherwise $w[n]$ is a rectangular window, but it may be a shaped window of the same type as used in windowed FIR filter design

- The DTFT of $y[n]$ is

$$Y(e^{j\omega}) = \sum_{n=0}^{N-1} y[n]e^{-j\omega n}, \forall\omega \tag{9.3}$$

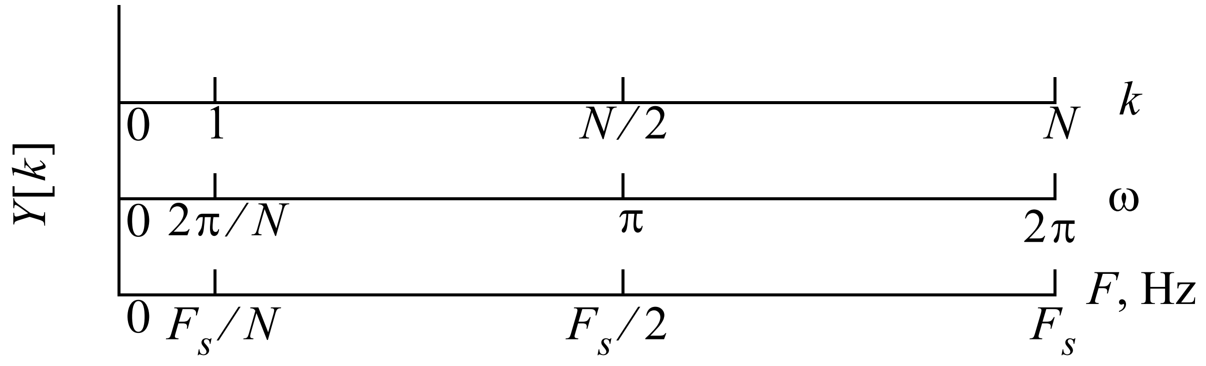
- The DFT of $y[n]$ is defined to be

$$Y[k] = \sum_{n=0}^{N-1} y[n]e^{-\frac{j2\pi kn}{N}}, k = 0, 1, \ldots, N-1 \tag{9.4}$$

- The relation between (9.3) and (9.4) is simply that

$$Y[k] = Y(e^{j\omega})\Big|_{\omega = \frac{2\pi k}{N}} \tag{9.5}$$

- DFT frequency axis relationships are the following:



- A feature of $Y[k]$ is that it is computable; about N^2 complex multiplications and $N(N-1)$ complex additions are required for an N-point DFT, while the radix-2 FFT, discussed shortly, reduces the complex multiplication count down to about $(N/2)\log_2 N$
- The inverse DFT or IDFT is defined to be

$$y[n] = \frac{1}{N}\sum_{k=0}^{N-1} Y[k]e^{\frac{j2\pi kn}{N}}, n = 0, 1, \ldots, N-1 \tag{9.6}$$

- In both the DFT and IDFT expressions it is customary to let

$$W_N \equiv e^{-j\frac{2\pi}{N}} \tag{9.7}$$

then the DFT and IDFT pair can be written as

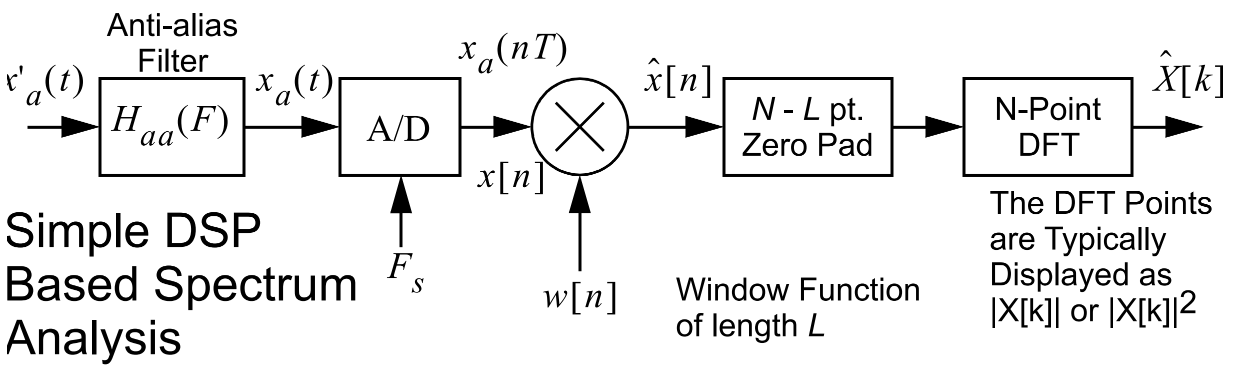
$$\boxed{\begin{aligned} Y[k] &= \sum_{n=0}^{N-1} y[n]W_N^{kn}, k = 0, 1, \ldots, N-1 \\ y[n] &= \frac{1}{N}\sum_{k=0}^{N-1} Y[k]W_N^{-kn}, n = 0, 1, \ldots, N-1 \end{aligned}} \tag{9.8}$$

– The complex weights W_N^k are known as *twiddle constants* or factors

Simple Application Examples

To further motivate study of the FFT for real-time applications, we will briefly consider two spectral domain processing examples

Spectral Analysis



- Suppose that

$$x_a(t) = A_1 \cos(2\pi F_1 t) + A_2 \cos(2\pi F_2 t) \tag{9.9}$$

and $F_1, F_2 \le F_s/2$

- The true spectrum, assuming an infinite observation time, is

$$\begin{aligned} X(\omega) = {} & \frac{A_1}{2}[\delta(\omega - \omega_1) + \delta(\omega + \omega_1)] \\ & + \frac{A_2}{2}[\delta(\omega - \omega_2) + \delta(\omega + \omega_2)] \end{aligned} \tag{9.10}$$

for $-\pi < \omega \le \pi$ and $\omega_1 = 2\pi F_1 / F_s$ and $\omega_2 = 2\pi F_2 / F_s$

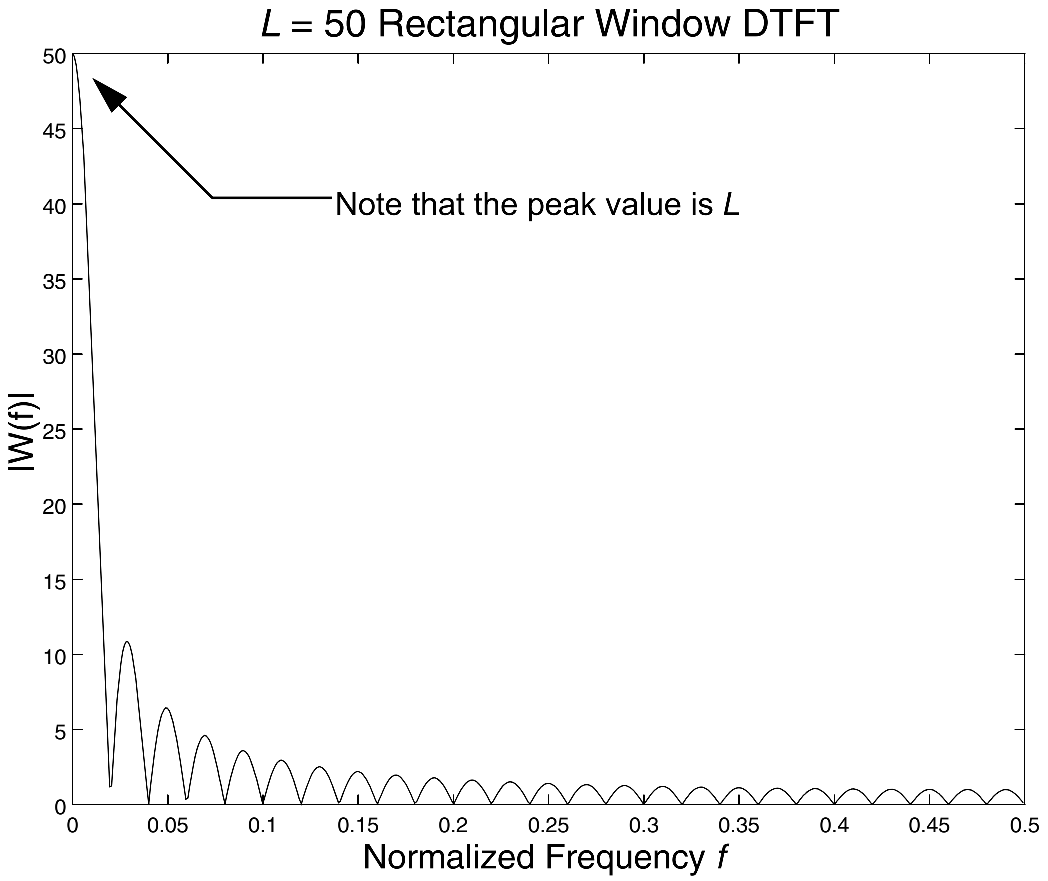
- As a result of windowing the data $X(\omega) \Rightarrow \hat{X}(\omega)$

$$\hat{X}(\omega) = \frac{A_1}{2}[W(\omega - \omega_1) + W(\omega + \omega_1)] + \frac{A_2}{2}[W(\omega - \omega_2) + W(\omega + \omega_2)] \tag{9.11}$$

  for $-\pi < \omega \le \pi$, where $W(\omega) = \text{DTFT}\{w[n]\}$

  – A rectangular window (in MATLAB `boxcar`) of length L samples has DTFT

$$W(\omega) = \frac{\sin(\omega L/2)}{\sin(\omega/2)} e^{-j\omega(L-1)/2} \tag{9.12}$$



- The DFT points, $\hat{X}[k]$, are simply a sampled version of (9.11), that is,

$$\begin{aligned}\hat{X}[k] = {} & \frac{A_1}{2}\left[W\left(\frac{2\pi k}{N} - \omega_1\right) + W\left(\frac{2\pi k}{N} + \omega_1\right)\right] \\ & + \frac{A_2}{2}\left[W\left(\frac{2\pi k}{N} - \omega_2\right) + W\left(\frac{2\pi k}{N} + \omega_2\right)\right]\end{aligned} \tag{9.13}$$

  for $0 \le k \le N$

- The zero-padding operation is optional, but is often used to improve the apparent spectral resolution
- Without zero padding we have only L samples on $(0, 2\pi)$, while with zero padding we effectively make the DFT create additional samples of $X(\omega)$, raising the number from L to N
- The frequency sample spacing changes from

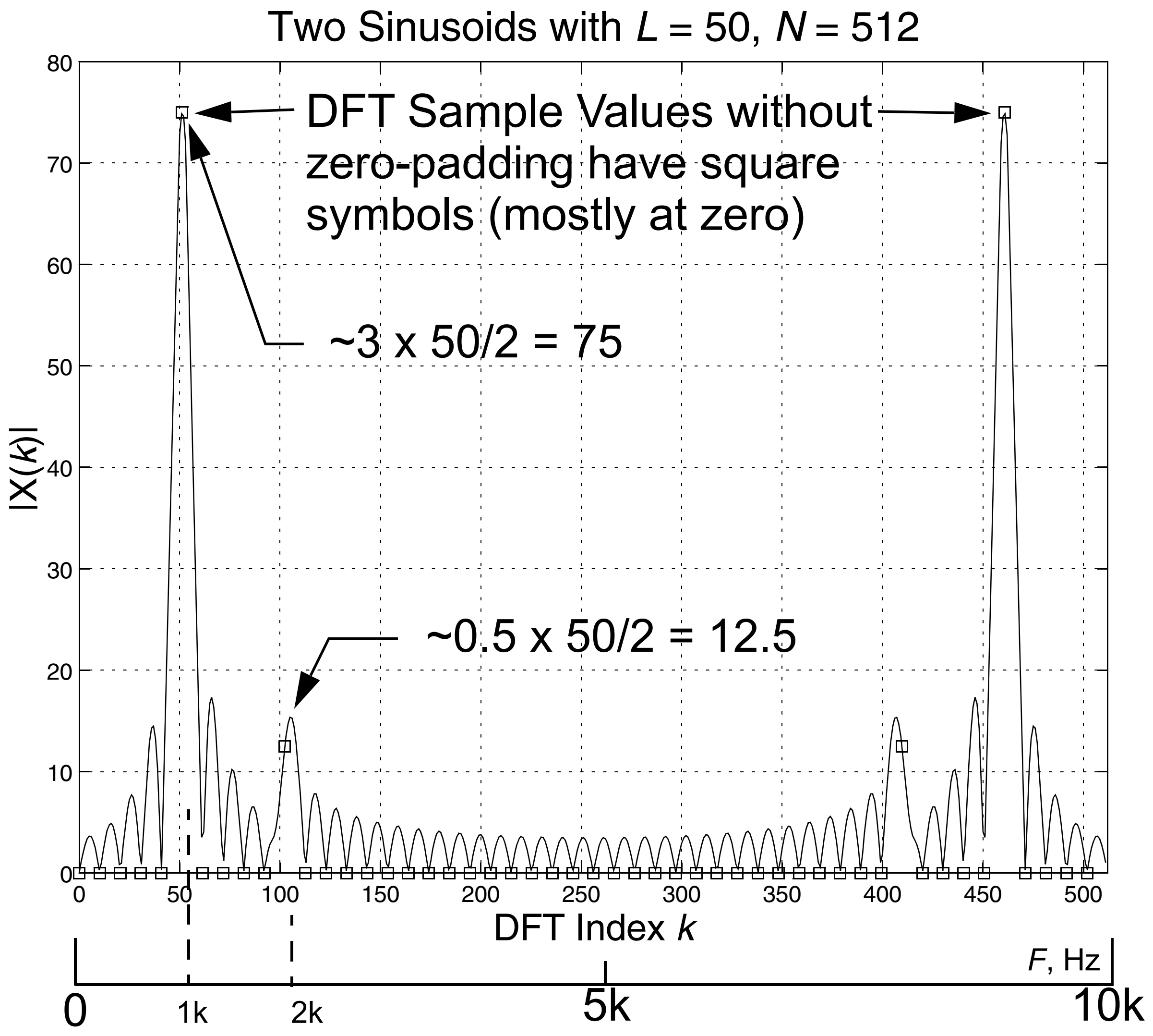
$$\Delta\omega = \frac{2\pi}{L} \text{ to } \Delta\omega = \frac{2\pi}{N} \text{ radians/sample}$$

  or

$$\Delta f = \frac{1}{L} \text{ to } \Delta f = \frac{1}{N} \text{ cycles/sample}$$

Example:

```
» n = 0:49;
» x = 3*cos(2*pi*1000*n/10000) + ...
  0.5*cos(2*pi*2000*n/10000); % Fs = 10000 Hz
» X = fft(x,512);
» plot(0:511,abs(X))
» grid
» title('Two Sinusoids with L = 50','fontsize',18)
» ylabel('|X(k)|','fontsize',16)
» xlabel('DFT Index k','fontsize',16)
» axis([0 512 0 80]);
» printmif('5910_2')
```

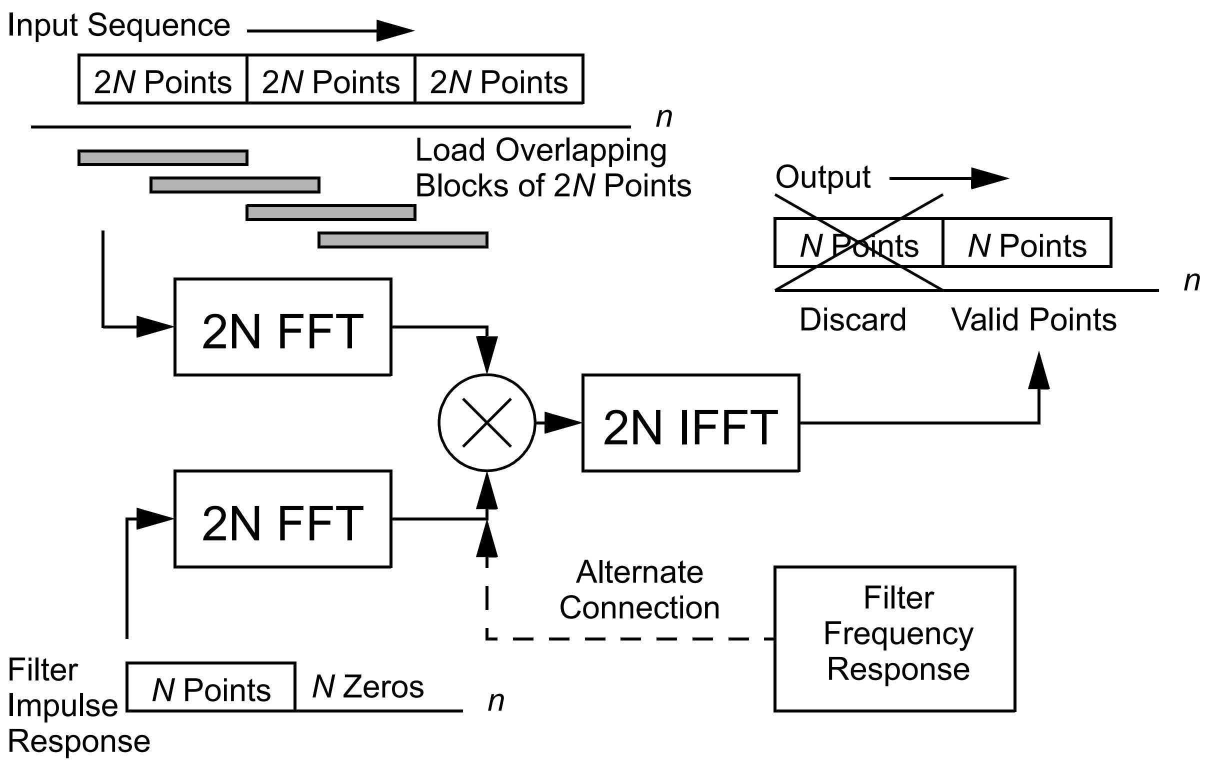


- With spectral analysis the goal may not be to actually display the spectrum, but rather to perform some additional computations on the DFT points

- For example we may try to decide if the frequency domain features of the transformed signal record are close to some template, i.e., target detection, pattern recognition/classification, etc.

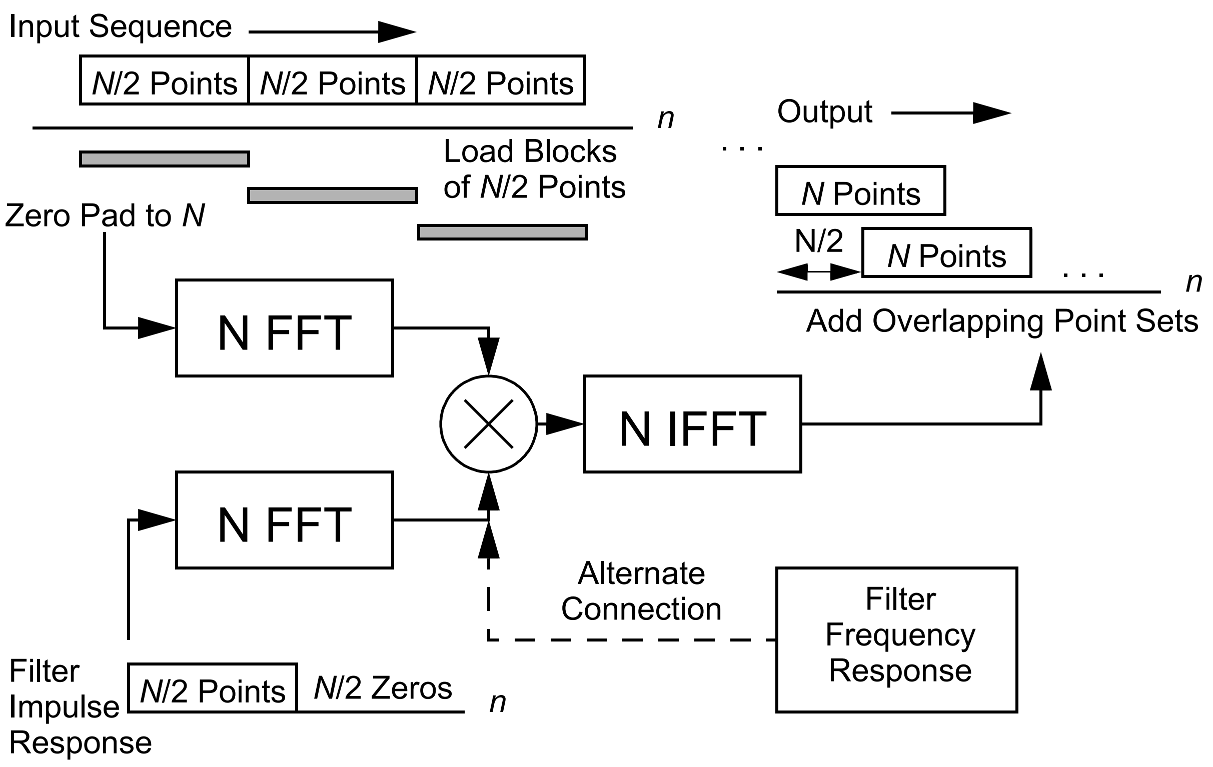
Transform Domain Linear Filtering

- A considerably different application of the DFT is to perform filtering directly in the frequency domain
- The basic assumption here is that the signal sample stream we wish to filter may have infinite duration, but the impulse response of the filter must be finite
- One standard approach is known as the *overlap and save* method of FIR filtering



Overlap and Save Filtering

- As the impulse response length grows the transform domain approach becomes more efficient than the time domain convolution sum, implemented in say a direct form FIR structure
- Another way of performing a transform domain convolution is with the *overlap and add* method



Overlap and Add Filtering

Radix 2 FFT

- A divide-and-conquer approach is taken in which an N-point (N a power of two, $N = 2^{\nu}$) DFT is decomposed into a cascade like connection of 2-point DFTs
  - With the decimation-in-time (DIT) algorithm the decomposition begins by decimating the input sequence $x[n]$

– With the decimation-in-frequency (DIF) algorithm the decomposition begins by decimating the output sequence $X[k]$ and working backwards

Decimation-in-Time

- Begin by separating $x[n]$ into even and odd $N/2$ point sequences

$$X[k] = \sum_{n \text{ even}} x[n]W_N^{kn} + \sum_{n \text{ odd}} x[n]W_N^{kn} \tag{9.14}$$

- In the first sum let $n = 2r$ and in the second let $n = 2r + 1$

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]W_N^{(2r+1)k} \tag{9.15}$$

or

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r](W_N^2)^{rk} + W_N^k\sum_{r=0}^{\frac{N}{2}-1} x[2r+1](W_N^2)^{rk} \tag{9.16}$$

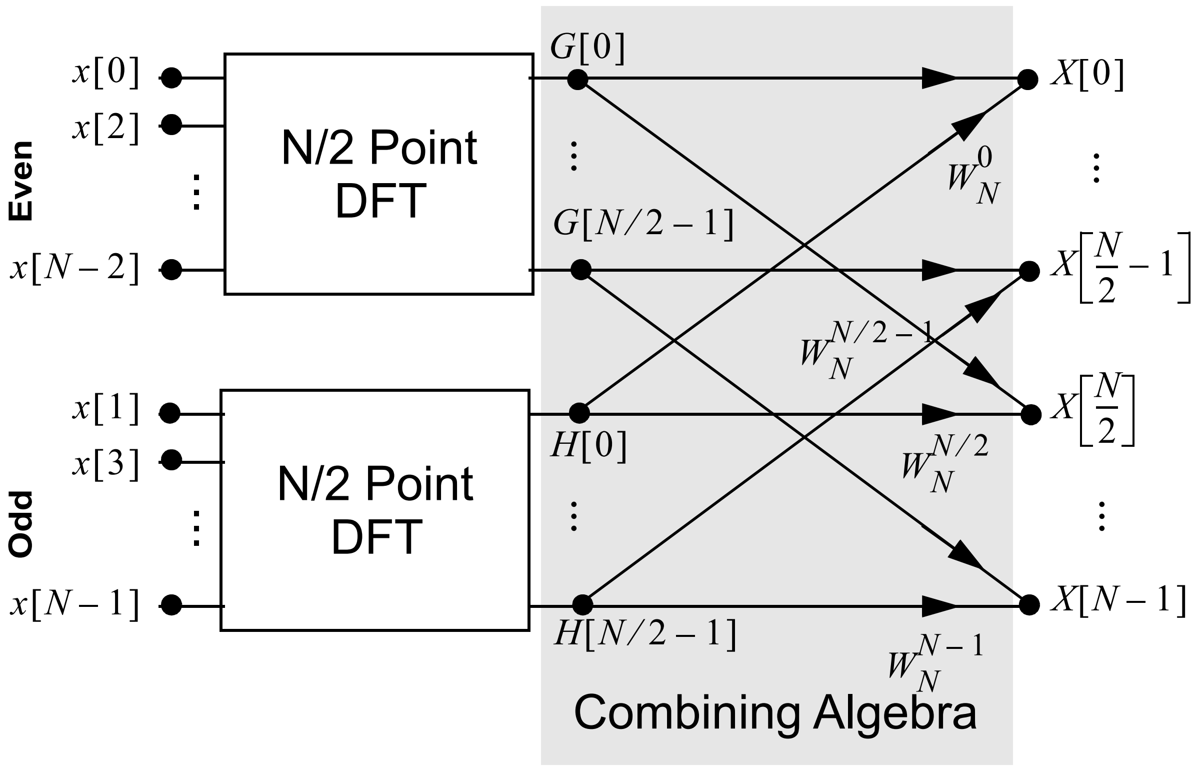
- Note that $W_N^2 = e^{-2j(2\pi/N)} = e^{-j2\pi/(N/2)} = W_{N/2}$, thus

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]W_{N/2}^{rk} + W_N^k\sum_{r=0}^{\frac{N}{2}-1} x[2r+1]W_{N/2}^{rk} \tag{9.17}$$

$$= G[k] + W_N^k H[k]$$

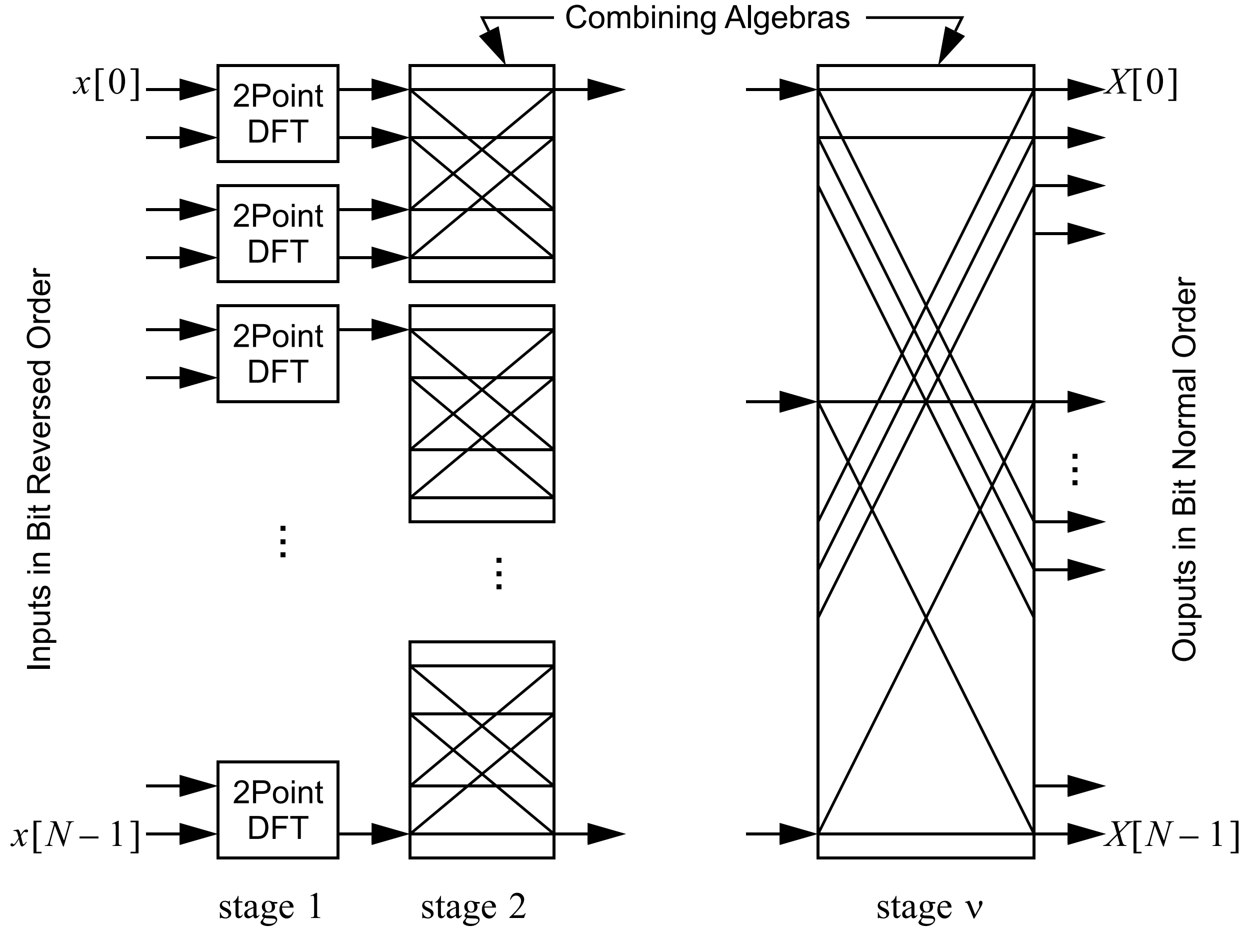
where we now see that $G[k]$ and $H[k]$ are $N/2$ point DFTs of the even and odd points of the original sequence $x[n]$

- A *combining* algebra brings the $N/2$-point DFTs together

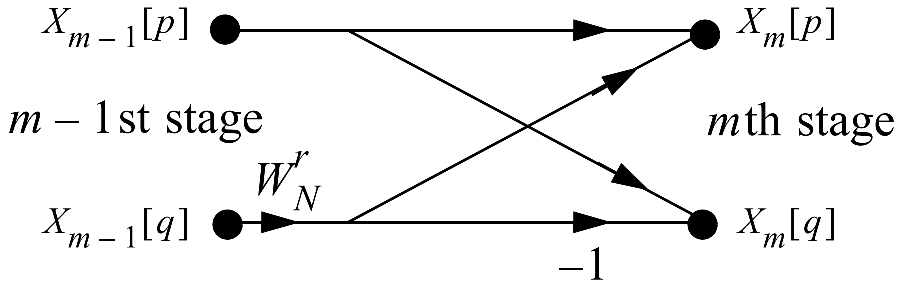


- After this one stage of decimation a computational savings has already been achieved
  - The number of complex multiplies is $2(N/2)^2 + N$ = $N^2/2 + N \le N^2$ for $N \ge 2$
- The procedure can be repeated until there are a total of $\nu = \log_2 N$ stages of decimation

- The general form of the DIT FFT signal flow graph is now



- When the decimation is complete the basic computation in each of the two point DFTs and the combining algebras all look the same
- We have what is known as the *radix-2 butterfly*



Simplified Radix-2 Butterfly

- The number stages is $\log_2 N$ and when using the simplified butterfly one complex multiply is required per stage with $N/2$ butterflies per stage, thus the total number of complex multiplies per FFT is

$$\text{Multiplications} = \frac{N}{2}\log_2 N \tag{9.18}$$

  – Note: Each complex multiply requires four real multiplies and two real additions

- The number of complex additions is

$$\text{Additions} = N\log_2 N \tag{9.19}$$

  – Note: Each complex addition requires two real additions

- Further study of the DIT algorithm reveals that it can be computed *in-place*, that is only one complex array is required to transform $x[n]$ into $X[k]$

- A drawback is that the input array must be initially reordered into *bit reversed* order, that is say for $N = 8$

$$
\begin{aligned}
x[0] &= x[000] \Rightarrow X[000] = X[0] \\
x[4] &= x[100] \Rightarrow X[001] = X[1] \\
x[2] &= x[010] \Rightarrow X[010] = X[2] \\
&\cdots \\
x[3] &= x[011] \Rightarrow X[110] = X[6] \\
x[7] &= x[111] \Rightarrow X[111] = X[7]
\end{aligned}
$$

Decimation-in-Frequency

- With the DIF approach the decimation starts at the output and works towards the input
- The signal flow graph turns out to be the transposed version of the DIT flow graph
  - The in-place calculation property holds again
  - The operations count is identical
  - A significant difference is that the input is in normal order and the output is now in bit reversed order

Computing the IDFT

- To compute the IDFT we do it the same way as the DFT, except we must include a $1/N$ scale factor and the twiddle constants must be conjugated

FFT Implementation

- With the CMSIS-DSP library there is a large collection of FFT algorithms
  - Initial testing of the functions resulted in failure, as the full version of Keil MDK is needed due to memory requirements
  - An example will be included in this chapter when the testing is complete
- As an alternative to CMSIS-DSP, a custom FFT module can be used instead
- A radix-2 C-based in-place FFT is contained in the `fft.c/fft.h`
  - This function uses the data structure `COMPLEX`, defined in the header file, to with complex `float32_t` variables

```
struct cmpx   //complex data structure
{
  float32_t real;
  float32_t imag;
};
typedef struct cmpx COMPLEX;
```

  - The function takes as input a `COMPLEX` array of length `fftlen` and returns the FFT of the data in the same array
  - A twiddle factor array, also of length `fftlen`, must be generated in a `COMPLEX` array beforehand

```
//fft.h
// definition of a complex FFT function written by Rulph Chas-
saing
// A complex data structure, COMPLEX, is also defined.
// Reworked for the Cortex-M4 by Mark Wickert, April 2015

#define ARM_MATH_CM4
#include <stdint.h>
#include "arm_math.h"

struct cmpx   //complex data structure used by FFT
{
  float32_t real;
  float32_t imag;
};
typedef struct cmpx COMPLEX;

void make_twiddle_array(int16_t fftlen,COMPLEX *twiddle);
void fft(COMPLEX *Y, int16_t M, COMPLEX *w);

//fft.c
// Implementation of a complex fft originally written by
// Rulph Chassaing.
// Reworked for the Cortex-M4 by Mark Wickert, April 2015.

#include "fft.h"

void make_twiddle_array(int16_t fftlen,COMPLEX *twiddle)
{
  int16_t n;
  float32_t vn;
  for (n=0 ; n<fftlen; n++)  //set up DFT twiddle factors
  {
    vn = (float32_t) (PI*n/fftlen);
    twiddle[n].real = cosf(vn);
    twiddle[n].imag = -sinf(vn);
  }

}

void fft(COMPLEX *Y, int16_t fftlen, COMPLEX *w)
{
  COMPLEX temp1,temp2;  //temporary storage variables
  int16_t i,j,k;        //loop counter variables
```

```
int16_t upper_leg, lower_leg; //index of upper/lower
                              //butterfly leg
int16_t leg_diff;    //difference between upper/lower leg
int16_t num_stages=0;//number of FFT stages, or iterations
int16_t index, step; //index and step between twiddle factor
i=1;                 //log(base 2) of # of points = # of stages
do
{
  num_stages+=1;
  i=i*2;
} while (i!=fftlen);

leg_diff=fftlen/2;   //starting difference between
                    //upper & lower legs
step=2;         //step between values in twiddle.h
for (i=0;i<num_stages;i++)      //for M-point FFT
{
  index=0;
  for (j=0;j<leg_diff;j++)
  {
    for (upper_leg=j;upper_leg<fftlen;upper_leg+=(2*leg_diff))
    {
      lower_leg=upper_leg+leg_diff;
      temp1.real=(Y[upper_leg]).real + (Y[lower_leg]).real;
      temp1.imag=(Y[upper_leg]).imag + (Y[lower_leg]).imag;
      temp2.real=(Y[upper_leg]).real - (Y[lower_leg]).real;
      temp2.imag=(Y[upper_leg]).imag - (Y[lower_leg]).imag;
      (Y[lower_leg]).real=temp2.real*(w[index]).real-
                          temp2.imag*(w[index]).imag;
      (Y[lower_leg]).imag=temp2.real*(w[index]).imag+
                          temp2.imag*(w[index]).real;
      (Y[upper_leg]).real=temp1.real;
      (Y[upper_leg]).imag=temp1.imag;
    }
    index+=step;
  }
  leg_diff=leg_diff/2;
  step*=2;
}
j=0;
```

```
  for (i=1;i<(fftlen-1);i++)  //bit reversal for
                              //re-sequencing data
  {
    k=fftlen/2;
    while (k<=j)
    {
      j=j-k;
      k=k/2;
    }
    j=j+k;
    if (i<j)
    {
      temp1.real=(Y[j]).real;
      temp1.imag=(Y[j]).imag;
      (Y[j]).real=(Y[i]).real;
      (Y[j]).imag=(Y[i]).imag;
      (Y[i]).real=temp1.real;
      (Y[i]).imag=temp1.imag;
    }
  }
  return;
}              //end of fft()
```

A Simple FFT Test in Main

- As a simple test of the above FFT on the Cortex-M4 consider a 32 point transform for some specific input data sets
- Snippets from the main module:

```
// stm32f4_FFT_dma.c

#include "defines.h"
#include "tm_stm32f4_delay.h"  //needed for USART library
#include "tm_stm32f4_gpio.h"   //include for GPIO library
#include "tm_stm32f4_usart.h"  //USART library to allow serial port control
#include "stm32_wm5102_init.h" //include for LiB codec library
#include "fft.h"
...
//FFT Experiment
//arm_rfft_fast_instance_f32 S_fft;
#define NFFT 32
COMPLEX in_data[NFFT];
```

```
float32_t fft_mag_data[NFFT];
COMPLEX twiddle[NFFT];
float32_t arg;
...
int main(void)
{
  //FFT test debug buffer
  char fft_debug[30];
  ...
  // FFT Test
  //arm_rfft_fast_init_f32(&S_fft,NFFT);
  //arm_rfft_fast_f32(&S_fft,in_data,out_data,0);
...
  //Fill the input array
  for (i=0; i< NFFT; i++)
  {
    arg = (float32_t)(2*PI*i*3/32);
    //in_data[i].real = 1;
    in_data[i].real = cosf(arg);
    //in_data[i].imag = 0;
    in_data[i].imag = sinf(arg);
  }
  make_twiddle_array(NFFT,twiddle);
...
  fft(in_data,NFFT,twiddle);
...
  arm_cmplx_mag_f32((float32_t *) in_data,fft_mag_data,NFFT);
...
  stm32_wm5102_init(FS_48000_HZ, WM5102_LINE_IN, IO_METHOD_DMA);
  while(1)
  {
    ...
  }
}
```

- We load the input array with 32 real 1’s, i.e., `1+j0`
- The results are given below:

```
In[0]: re=1.00, im=0.00, In[1]: re=1.00, im=0.00, In[2]: re=1.00, im=0.00
In[3]: re=1.00, im=0.00, In[4]: re=1.00, im=0.00, In[5]: re=1.00, im=0.00
In[6]: re=1.00, im=0.00, In[7]: re=1.00, im=0.00, In[8]: re=1.00, im=0.00
In[9]: re=1.00, im=0.00, In[10]: re=1.00, im=0.00, In[11]: re=1.00, im=0.00
In[12]: re=1.00, im=0.00, In[13]: re=1.00, im=0.00, In[14]: re=1.00, im=0.00
In[15]: re=1.00, im=0.00, In[16]: re=1.00, im=0.00, In[17]: re=1.00, im=0.00
In[18]: re=1.00, im=0.00, In[19]: re=1.00, im=0.00, In[20]: re=1.00, im=0.00
In[21]: re=1.00, im=0.00, In[22]: re=1.00, im=0.00, In[23]: re=1.00, im=0.00
In[24]: re=1.00, im=0.00, In[25]: re=1.00, im=0.00, In[26]: re=1.00, im=0.00
In[27]: re=1.00, im=0.00, In[28]: re=1.00, im=0.00, In[29]: re=1.00, im=0.00
In[30]: re=1.00, im=0.00, In[31]: re=1.00, im=0.00,
Out[0]: re=32.00, im=0.00, Out[1]: re=0.00, im=0.00, Out[2]: re=0.00, im=0.00
Out[3]: re=0.00, im=0.00, Out[4]: re=0.00, im=0.00, Out[5]: re=0.00, im=0.00
Out[6]: re=0.00, im=0.00, Out[7]: re=0.00, im=0.00, Out[8]: re=0.00, im=0.00
Out[9]: re=0.00, im=0.00, Out[10]: re=0.00, im=0.00, Out[11]: re=0.00, im=0.00
Out[12]: re=0.00, im=0.00, Out[13]: re=0.00, im=0.00, Out[14]: re=0.00, im=0.00
Out[15]: re=0.00, im=0.00, Out[16]: re=0.00, im=0.00, Out[17]: re=0.00, im=0.00
Out[18]: re=0.00, im=0.00, Out[19]: re=0.00, im=0.00, Out[20]: re=0.00, im=0.00
Out[21]: re=0.00, im=0.00, Out[22]: re=0.00, im=0.00, Out[23]: re=0.00, im=0.00
Out[24]: re=0.00, im=0.00, Out[25]: re=0.00, im=0.00, Out[26]: re=0.00, im=0.00
Out[27]: re=0.00, im=0.00, Out[28]: re=0.00, im=0.00, Out[29]: re=0.00, im=0.00
Out[30]: re=0.00, im=0.00, Out[31]: re=0.00, im=0.00,

Mag[0]: 32.00, Mag[1]: 0.00, Mag[2]: 0.00, Mag[3]: 0.00, Mag[4]: 0.00
Mag[5]: 0.00, Mag[6]: 0.00, Mag[7]: 0.00, Mag[8]: 0.00, Mag[9]: 0.00
Mag[10]: 0.00, Mag[11]: 0.00, Mag[12]: 0.00, Mag[13]: 0.00, Mag[14]: 0.00
Mag[15]: 0.00, Mag[16]: 0.00, Mag[17]: 0.00, Mag[18]: 0.00, Mag[19]: 0.00
Mag[20]: 0.00, Mag[21]: 0.00, Mag[22]: 0.00, Mag[23]: 0.00, Mag[24]: 0.00
Mag[25]: 0.00, Mag[26]: 0.00, Mag[27]: 0.00, Mag[28]: 0.00, Mag[29]: 0.00
Mag[30]: 0.00, Mag[31]: 0.00,
```

- As expected a constant input gives a nonzero FFT point at DC only, i.e., $X[0] = 32$ and $X[k] = 0$ otherwise
- As a second test consider a complex sinusoid

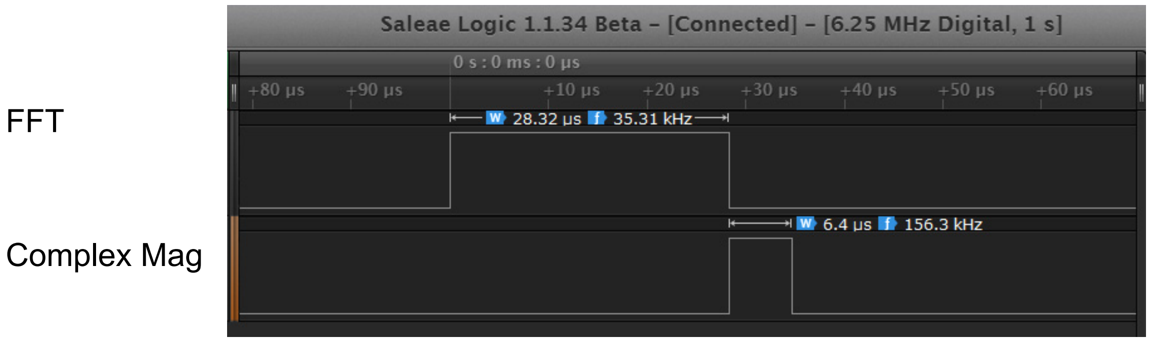
$$x[n] = e^{j2\pi n(3/32)}, n = 0, \ldots, 31 \qquad (9.20)$$

- The output is:

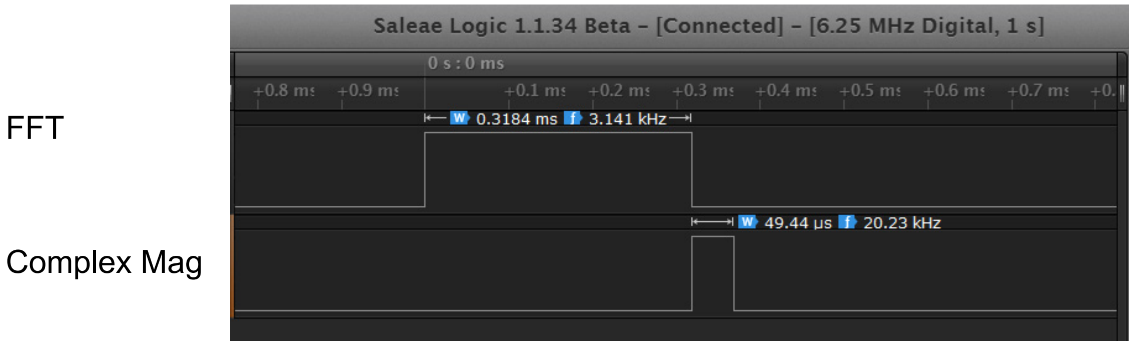
```
In[0]: re=1.00, im=0.00, In[1]: re=0.83, im=0.56, In[2]: re=0.38, im=0.92
In[3]: re=-0.20, im=0.98, In[4]: re=-0.71, im=0.71, In[5]: re=-0.98, im=0.20
In[6]: re=-0.92, im=-0.38, In[7]: re=-0.56, im=-0.83, In[8]: re=0.00, im=-1.00
In[9]: re=0.56, im=-0.83, In[10]: re=0.92, im=-0.38, In[11]: re=0.98, im=0.20
In[12]: re=0.71, im=0.71, In[13]: re=0.20, im=0.98, In[14]: re=-0.38, im=0.92
In[15]: re=-0.83, im=0.56, In[16]: re=-1.00, im=-0.00, In[17]: re=-0.83, im=-0.56
In[18]: re=-0.38, im=-0.92, In[19]: re=0.20, im=-0.98, In[20]: re=0.71, im=-0.71
In[21]: re=0.98, im=-0.20, In[22]: re=0.92, im=0.38, In[23]: re=0.56, im=0.83
In[24]: re=-0.00, im=1.00, In[25]: re=-0.56, im=0.83, In[26]: re=-0.92, im=0.38
In[27]: re=-0.98, im=-0.20, In[28]: re=-0.71, im=-0.71, In[29]: re=-0.20, im=-0.98
In[30]: re=0.38, im=-0.92, In[31]: re=0.83, im=-0.56,

Out[0]: re=0.00, im=-0.00, Out[1]: re=0.00, im=-0.00, Out[2]: re=0.00, im=-0.00
Out[3]: re=32.00, im=0.00, Out[4]: re=-0.00, im=-0.00, Out[5]: re=0.00, im=-0.00
Out[6]: re=-0.00, im=-0.00, Out[7]: re=0.00, im=-0.00, Out[8]: re=0.00, im=0.00
Out[9]: re=-0.00, im=0.00, Out[10]: re=-0.00, im=0.00, Out[11]: re=-0.00, im=0.00
Out[12]: re=-0.00, im=-0.00, Out[13]: re=0.00, im=-0.00, Out[14]: re=-0.00, im=-0.00
Out[15]: re=0.00, im=-0.00, Out[16]: re=0.00, im=0.00, Out[17]: re=0.00, im=-0.00
Out[18]: re=0.00, im=0.00, Out[19]: re=-0.00, im=0.00, Out[20]: re=-0.00, im=0.00
Out[21]: re=-0.00, im=-0.00, Out[22]: re=-0.00, im=0.00, Out[23]: re=-0.00, im=-0.00
Out[24]: re=0.00, im=-0.00, Out[25]: re=-0.00, im=-0.00, Out[26]: re=0.00, im=-0.00
```

```
Out[27]: re=0.00, im=0.00, Out[28]: re=0.00, im=0.00, Out[29]: re=-0.00, im=0.00
Out[30]: re=-0.00, im=0.00, Out[31]: re=0.00, im=-0.00,

Mag[0]: 0.00, Mag[1]: 0.00, Mag[2]: 0.00, Mag[3]: 32.00, Mag[4]: 0.00
Mag[5]: 0.00, Mag[6]: 0.00, Mag[7]: 0.00, Mag[8]: 0.00, Mag[9]: 0.00
Mag[10]: 0.00, Mag[11]: 0.00, Mag[12]: 0.00, Mag[13]: 0.00, Mag[14]: 0.00
Mag[15]: 0.00, Mag[16]: 0.00, Mag[17]: 0.00, Mag[18]: 0.00, Mag[19]: 0.00
Mag[20]: 0.00, Mag[21]: 0.00, Mag[22]: 0.00, Mag[23]: 0.00, Mag[24]: 0.00
Mag[25]: 0.00, Mag[26]: 0.00, Mag[27]: 0.00, Mag[28]: 0.00, Mag[29]: 0.00
Mag[30]: 0.00, Mag[31]: 0.00,
```

- The results are as expected, all of the energy is in the $k = 3$ FFT point as the complex sinusoid located at frequency bin 3
- Timing results for the FFT and the CMSIS_DSP complex magnitude function over 32 points



- See what happens when the length grows to 256 points



- Are the time ratios reasonable?
  - For the radix-2 FFT the ratio should be about

$$R_{\text{FFT}} = \frac{N_2 \log_2(N_2)}{N_1 \log_1(N_1)} = \frac{256 \cdot 8}{32 \cdot 5} = 12.8 \tag{9.21}$$

  - The measured ratio is

$$R_{\text{FFT,measured}} = \frac{318.4}{28.32} = 11.24 \text{ (very close!)} \tag{9.22}$$

- The time ratio of the complex magnitude calculation should be simply N_2/N_1:

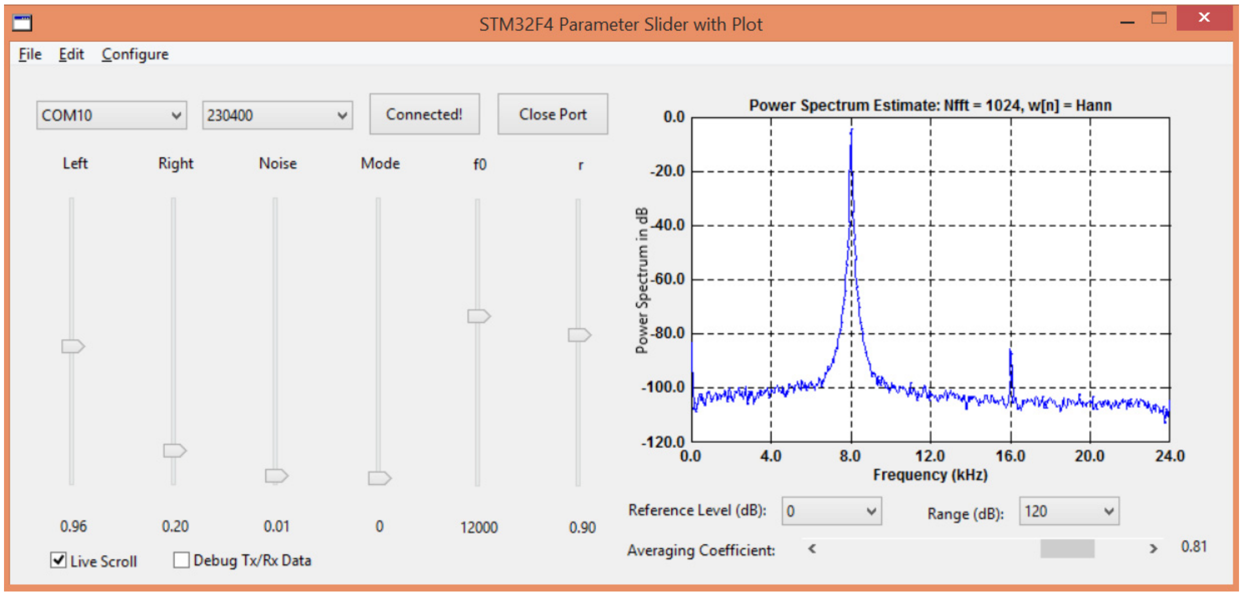
$$\begin{aligned} R_{\text{cmag}} &= \frac{256}{32} = 8 \\ R_{\text{cmag ,measured}} &= \frac{49.44}{6.4} = 7.72 \end{aligned} \tag{9.23}$$

Looks good!

Real-Time Spectrum Analyzer Using DMA

- In this example we consider a spectrum analyzer that is based on the radix-2 FFT introduced in the previous example
- This is an application example as a the GUI parameter slider is enhanced to include a plot window and spectrum analyzer controls
- Here DMA frame buffers are used to:
  - Fill a 1024 point complex array with the sum of the input samples taken from the left and right channels
  - Internally generated white noise samples are also added
- Recall that the default DMA buffer length is 256 samples, with 128 samples per channel
- To fill a 1024 point FFT buffer will require

$$N_{\text{DMA Frames}} = \frac{1024}{128} = 8 \tag{9.24}$$



COM10 | 230400

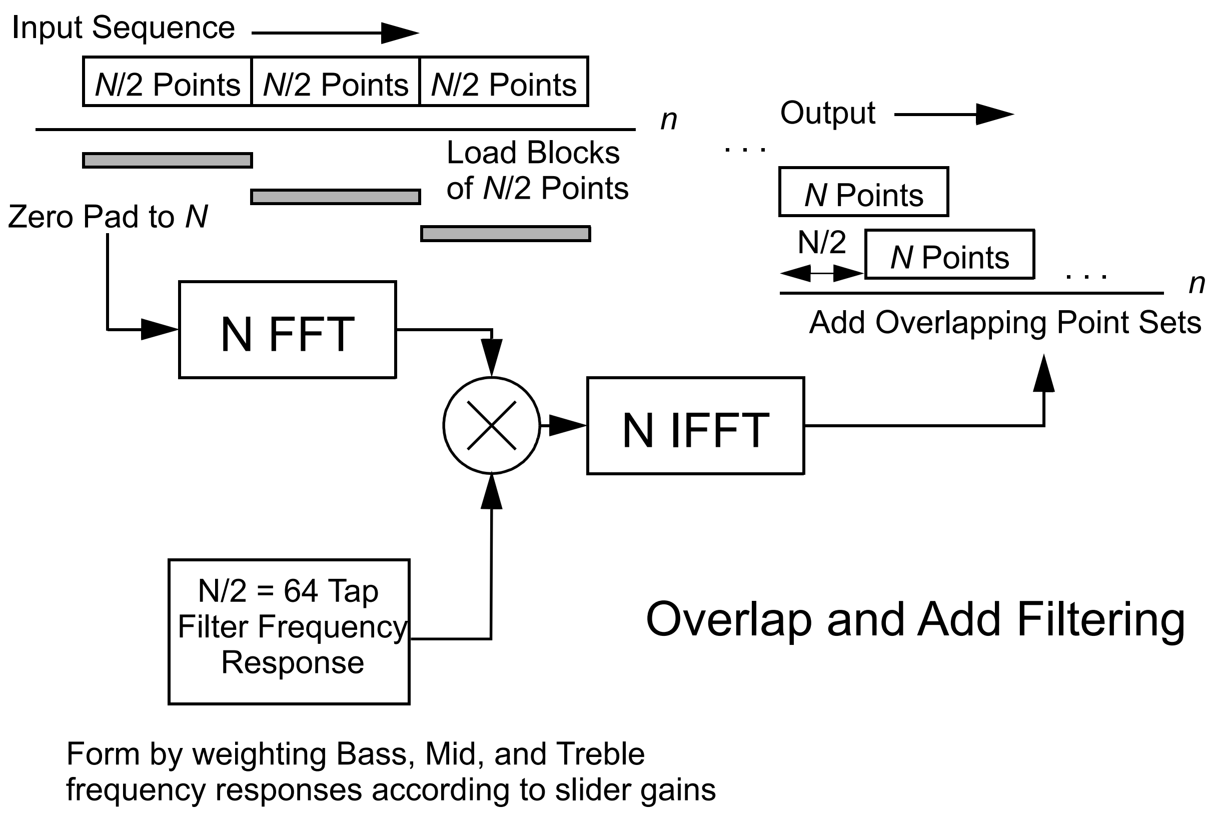
Left 0.96, Right 0.20, Noise 0.01, Mode 0, f0 12000, r 0.90

Reference Level (dB): 0 Range (dB): 120

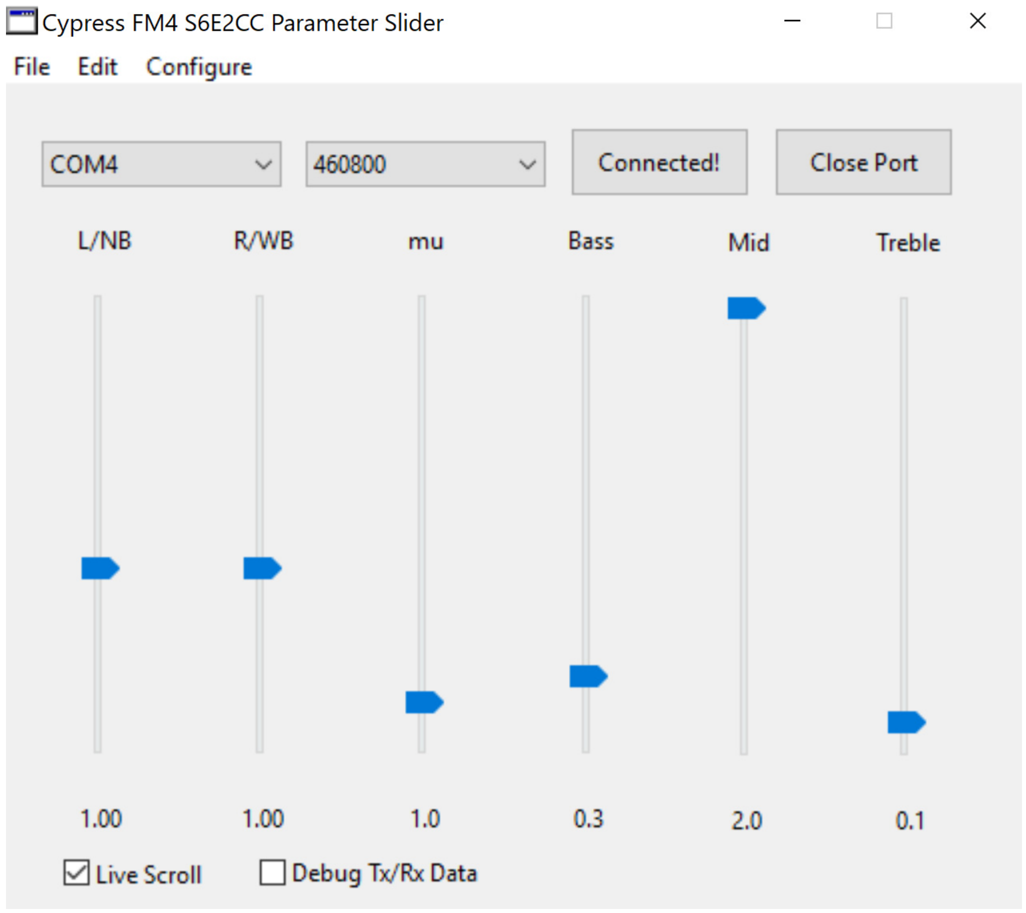
Averaging Coefficient: 0.81

Transform Domain Filtering using Overlap and Save

- Consider the system block diagram


Input Sequence
N/2 Points
N/2 Points
N/2 Points
n
Load Blocks of N/2 Points
Zero Pad to N
Output
N Points
N/2
N Points
n
Add Overlapping Point Sets
N FFT
N IFFT
N/2 = 64 Tap Filter Frequency Response
Overlap and Add Filtering
Form by weighting Bass, Mid, and Treble frequency responses according to slider gains

- Three windowed FIR filter coefficients sets, 64 taps each, are loaded and converted to frequency domain coefficients in `main`
- At the DMA frame rate the frequency domain bass, mid, and treble coefficients are linearly combined according to the slider values
- The FFT size is 128 points to allow linear convolution to properly produce $2 \times 64 - 1 = 127$ valid output points each pass through the overlap and add
- The extra points are retained from pass-to-pass to *add on*



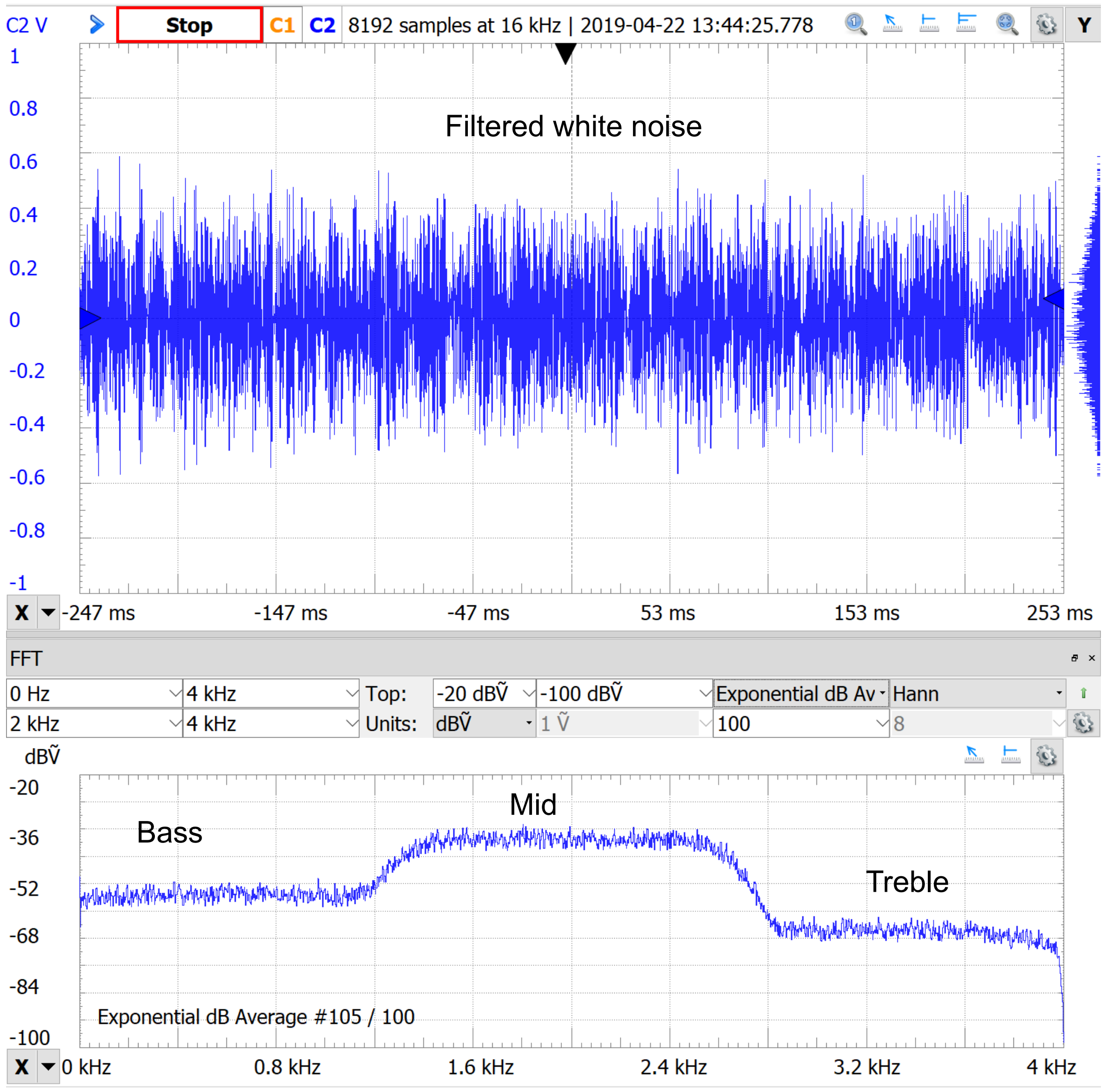
L/NB 1.00, R/WB 1.00, mu 1.0, Bass 0.3, Mid 2.0, Treble 0.1

- Code

```
// fm4_graphicEQ_dma.c

#include "fm4_wm8731_init.h"
#include "GraphicEQcoeff.h"
#include "FM4_slider_interface.h"

#define NN (DMA_BUFFER_SIZE)

typedef struct
{
  float real;
  float imag;
} COMPLEX;

#include "fft_reay.h"

// Create (instantiate) GUI slider data structure
struct FM4_slider_struct FM4_GUI;

COMPLEX procbuf[2*NN],coeffs[2*NN],twiddle[2*NN];
COMPLEX treble[2*NN],bass[2*NN],mid[2*NN];
float32_t overlap[NN];
```

```
float32_t a,b;
float32_t bass_gain = 0.65;
float32_t mid_gain = 0.05;
float32_t treble_gain = 0.25;

int16_t NUMCOEFFS = sizeof(lpcoeff)/sizeof(float32_t);

void process_dma_buffer(void)
{
  int i;
  uint32_t *txbuf, *rxbuf;
  union WM8731_data sample;

  if(tx_proc_buffer == PING) txbuf = dma_tx_buffer_ping;
  else txbuf = dma_tx_buffer_pong;
  if(rx_proc_buffer == PING) rxbuf = dma_rx_buffer_ping;
  else rxbuf = dma_rx_buffer_pong;

  for (i = 0; i < (2*NN) ; i++)
  {
    procbuf[i].real = 0.0;
    procbuf[i].imag = 0.0;
  }
  for (i = 0; i < NN ; i++)
  {
 sample.uint32bit = rxbuf[i];
//    procbuf[i].real = (float32_t)(sample.uint16bit[LEFT]);
    procbuf[i].real = (float32_t)(prbs(8000));
  }
    // Transform to the frequency domain
  fft(procbuf,2*NN,twiddle);
    // Apply filtering in the frequency domain
  for (i=0 ; i<(2*NN) ; i++)
  {
    a = procbuf[i].real;
    b = procbuf[i].imag;
    procbuf[i].real = coeffs[i].real*a
                    - coeffs[i].imag*b;
    procbuf[i].imag = -(coeffs[i].real*b
                      + coeffs[i].imag*a);
  }
    // Transform back to the time domain
  fft(procbuf,2*NN,twiddle);
    // Combine bass, mid, treble
    // frequency domain filter coefficients.
    for (i=0 ; i<NN ; i++)
    {
        coeffs[i].real = bass[i].real*FM4_GUI.P_vals[3] //bass_gain
                       + mid[i].real*FM4_GUI.P_vals[4] //mid_gain
                       + treble[i].real*FM4_GUI.P_vals[5]; //treble_gain;
      coeffs[i].imag = bass[i].imag*FM4_GUI.P_vals[3] //bass_gain
                       + mid[i].imag*FM4_GUI.P_vals[4] //mid_gain
                       + treble[i].imag*FM4_GUI.P_vals[5]; //treble_gain;
    }
  for (i=0 ; i<(2*NN) ; i++)
  {
    procbuf[i].real /= (2*NN);
  }
    // Overlap and Processing for the current DMA frame
  for (i = 0; i < NN ; i++)
  {
    sample.uint16bit[LEFT] = (int16_t)(procbuf[i].real + overlap[i]);
    sample.uint16bit[RIGHT] = 0; // No output on right channel
        // Manage overlap to be applied on next DMA frame
    overlap[i] = procbuf[i+NN].real;
    txbuf[i] = sample.uint32bit;
  }


  tx_buffer_empty = 0;
  rx_buffer_full = 0;
}

int main (void)
{
  int i;

  for (i=0 ; i< (2*NN) ; i++)
  {
    twiddle[i].real = cos(PI*i/(2*NN));
    twiddle[i].imag = -sin(PI*i/(2*NN));
  }

    // Initialize the slider interface by setting the baud rate (460800 or
921600)
    // and initial float values for each of the 6 slider parameters

    init_slider_interface(&FM4_GUI,460800, 1.0, 1.0, 0.0, 0.0, 0.0, 1.0);

    // Send a string to the PC terminal
    //write_uart0("Hello FM4 World!\r\n");

    for(i=0 ; i<(NN*2) ; i++)
  {
    coeffs[i].real = 0.0;
        coeffs[i].imag = 0.0;
    bass[i].real = 0.0;
    mid[i].real = 0.0 ;
    treble[i].real = 0.0;
    bass[i].imag = 0.0;
    mid[i].imag = 0.0 ;
    treble[i].imag = 0.0;
  }
  for(i=0 ; i<NN ; i++)
  {
    overlap[i] = 0.0;
  }
  for(i=0 ; i<NUMCOEFFS ; i++)
  {
    bass[i].real = lpcoeff[i];
    mid[i].real = bpcoeff[i];
    treble[i].real = hpcoeff[i];
  }
  fft(bass,(2*NN),twiddle);
  fft(mid,(2*NN),twiddle);
  fft(treble,(2*NN),twiddle);

 fm4_wm8731_init (FS_8000_HZ,
                    WM8731_LINE_IN,
                    IO_METHOD_DMA,
                    WM8731_HP_OUT_GAIN_0_DB,
                    WM8731_LINE_IN_GAIN_0_DB);
  while(1)
  {
        // wait for next buffer
        while (!(rx_buffer_full && tx_buffer_empty)){
                // Update slider parameters
                update_slider_parameters(&FM4_GUI);
        }
    gpio_set(DIAGNOSTIC_PIN, HIGH);
    process_dma_buffer();
    gpio_set(DIAGNOSTIC_PIN, LOW);
  }
}
```

- Noise testing


C2 V
Stop
C1
C2
8192 samples at 16 kHz | 2019-04-22 13:44:25.778
Y
Filtered white noise
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
X
-247 ms
-147 ms
-47 ms
53 ms
153 ms
253 ms
FFT
0 Hz
4 kHz
Top:
-20 dBṼ
-100 dBṼ
Exponential dB Av
Hann
2 kHz
4 kHz
Units:
dBṼ
1 Ṽ
100
8
dBṼ
-20
-36
-52
-68
-84
-100
Bass
Mid
Treble
Exponential dB Average #105 / 100
X
0 kHz
0.8 kHz
1.6 kHz
2.4 kHz
3.2 kHz
4 kHz