Real-Time Fast Fourier Transform

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

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

- A characteristic of the spectrum $X(e^{j\omega})$ is that it is a periodic function having period $2\pi$

- 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 $\omega$ frequency axes is related to the normalized digital frequency $f = \omega/(2\pi)$ and the analog frequency $F = f \cdot F_s$ as follows:

![Diagram of DTFT and DFT]

The DFT and its Relation to the DTFT

- The DFT assumes that the sequence $y[n]$ exists only on a finite interval, say $0 \leq n \leq N - 1$, and zero otherwise

- This may occur as a result of data windowing, e.g.,

$$y[n] = x[n]w[n] \quad (9.2)$$
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

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

• The DFT of \( y[n] \) is defined to be

\[
Y[k] = \sum_{n=0}^{N-1} y[n]e^{j\frac{2\pi kn}{N}}, \ k = 0, 1, \ldots, N-1
\]  
(9.4)

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

\[
\left. Y[k] = Y(e^{j\omega}) \right|_{\omega = \frac{2\pi k}{N}}
\]  
(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

$$y[n] = \frac{1}{N} \sum_{k=0}^{N-1} Y[k] e^{j\frac{2\pi kn}{N}}, \quad n = 0, 1, \ldots, N-1 \quad (9.6)$$

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

$$W_N \equiv e^{-j\frac{2\pi}{N}} \quad (9.7)$$

then the DFT and IDFT pair can be written as

\[
Y[k] = \sum_{n=0}^{N-1} y[n] W_N^{kn}, \quad k = 0, 1, \ldots, N-1 \quad (9.8)
\]

\[
y[n] = \frac{1}{N} \sum_{k=0}^{N-1} Y[k] W_N^{-kn}, \quad n = 0, 1, \ldots, N-1
\]

– 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
  \[ x_a(t) = A_1 \cos(2\pi F_1 t) + A_2 \cos(2\pi F_2 t) \]  
  and \( F_1, F_2 \leq F_s / 2 \)

- The true spectrum, assuming an infinite observation time, is
  \[
  X(\omega) = \frac{A_1}{2} [\delta(\omega - \omega_1) + \delta(\omega + \omega_1)] 
  + \frac{A_2}{2} [\delta(\omega - \omega_2) + \delta(\omega + \omega_2)]
  \]  
  for \(-\pi < \omega \leq \pi\) 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 $X(\omega) \Rightarrow \hat{X}(\omega)$

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

$$+ \frac{A_2}{2} [W(\omega - \omega_2) + W(\omega + \omega_2)]$$

(9.11)

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

- A rectangular window (in MATLAB boxcar) of length $L$ samples has DTFT

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

(9.12)

$L = 50$ Rectangular Window DTFT

Note that the peak value is $L$
• The DFT points, $X[k]$, are simply a sampled version of (9.11), that is,

$$X[k] = \frac{A_1}{2} \left[ W\left(\frac{2\pi k}{N} - \omega_1\right) + W\left(\frac{2\pi k}{N} + \omega_1\right) \right]$$

$$+ \frac{A_2}{2} \left[ W\left(\frac{2\pi k}{N} - \omega_2\right) + W\left(\frac{2\pi k}{N} + \omega_2\right) \right]$$

(9.13)

for $0 \leq k \leq N$

• 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:

```matlab
» 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
```
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**

[Diagram showing the process of 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.

**Radix 2 FFT**

A divide-and-conquer approach is taken in which an $N$-point ($N$ a power of two, $N = 2^v$) 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 $x[n]$. 

---

**Overlap and Add Filtering**

- The diagram illustrates the process of overlap and add filtering, where an input sequence is zero-padded to length $N$, divided into $N/2$ blocks, and each block is processed by an $N/2$-point FFT, followed by a convolution with the filter impulse response, and finally an $N$-point IFFT to obtain the output. 

---
– 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

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

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

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

or

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

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

$$X[k] = \sum_{r = 0}^{N/2 - 1} x[2r] W_N^{rk} + W_N^k \sum_{r = 0}^{N/2 - 1} x[2r + 1] W_N^{rk} \quad (9.17)$$

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

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 $2(\frac{N}{2})^2 + N = \frac{N^2}{2} + N \leq \frac{N^2}{2}$ for $N \geq 2$

• The procedure can be repeated until there are a total of $v = \log_2 N$ stages of decimation
• The general form of the DIT FFT signal flow graph is now

![Signal Flow Graph]

- 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]

\[
X_{m-1}[p] \quad W_N^r \quad X_m[p] \\
X_{m-1}[q] \quad -1 \quad X_m[q]
\]

**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 \( N = 8 \)
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 work with complex `float32_t` variables
    ```c
    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 Chassaing
// A complex data structure, COMPLEX, is also defined.
// Reworked for the Cortex-M4 by Mark Wickert, April 2015

```c
//complex data structure used by FFT

struct cmpx
{
    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);
```

```c
#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
```
Chapter 9 • Real-Time Fast Fourier Transform

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
FFT Implementation

```c
//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:

```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];
```
Chapter 9 • Real-Time Fast Fourier Transform

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
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

$$x[n] = e^{j\frac{2\pi n}{33}} \quad n = 0, \ldots, 31 \quad (9.20)$$

The output is:

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=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,
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
    \[ R_{\text{FFT}} = \frac{N_2 \log_2(N_2)}{N_1 \log_1(N_1)} = \frac{256 \cdot 8}{32 \cdot 5} = 12.8 \]  
    \( (9.21) \)
  – The measured ratio is
    \[ R_{\text{FFT,measured}} = \frac{318.4}{28.32} = 11.24 \text{ (very close!)} \]  
    \( (9.22) \)
• The time ratio of the complex magnitude calculation should be simply \( N_2/N_1 \):
  \[ R_{\text{cmag}} = \frac{256}{32} = 8 \]  
  \( (9.23) \)
  \[ R_{\text{cmag,measured}} = \frac{49.44}{6.4} = 7.72 \]

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:

\[
N_{\text{DMA Frames}} = \frac{1024}{128} = 8
\]  

(9.24)
Transform domain filtering using overlap and save or overlap and add