Discrete-time Fourier Transform and its implements

Yiming Bi – yb127 – 146004795

Fourier transform is an important method in engineering, both in continuous and discrete types. It helps to analysis signals in frequency domain. Frequency analysis provides a new view of signals besides time domain analysis. In electrical engineering practices, many signals are digital and in discrete domain. So, discrete time Fourier transform (DTFT) is a major topic. A professor in our department has the word "It is very regrettable that in our department, electrical people do not learn much algorithm and software people do not learn much Fourier transform". In this term paper, I will briefly introduce DTFT and its implement algorithms.

Content

Introduction

The importance of DTFT has been stated before. In this term paper, I will start from some basic mathematics of DTFT, discuss how to implement DTFT in algorithm. A famous algorithm has been developed called fast Fourier transform (FFT) is the most important topic. This algorithm improves the time complexity a lot comparing to more direct algorithms. FFT has also been integrated into many libraries of different languages. Some sample codes from python will also be discussed. At the end of this paper, a quantitively analysis will be provided to see FFT improves performance by how much.

Mathematics of DTFT

Analysis equation of DTFT is:

$$X\left(e^{j\omega}\right) = \sum_{n=-\infty}^{\infty} x[n]e^{-j\omega n}, 0 \le \omega \le 2\pi$$

Here, $x[n]$ is the original signal in discrete time domain. $X\left(e^{j\omega}\right)$ is the DTFT of the original signal. By the features of discrete time frequency, $X\left(e^{j\omega}\right)$ is periodic within domain $[0, 2\pi]$. This equation is doing convolution through infinity length. The result of this convolution is the frequency distribution as the real part and the phase shift as the imagine part. However, here the DTFT is a continuous function and it requires infinity length input, which is not practical. As a solution, discrete Fourier transform (DFT) is called. Similarly, analysis equation of DFT is:

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j\frac{2\pi}{N}kn}, 0 \le k \le N-1$$

Here, length of convolution does not have to be infinity. It is limited by the length of input signal, N. But still, DFT $X[k]$ is periodic within domain $[0, 2\pi]$. $X[k]$ is like sampling from $X\left(e^{j\omega}\right)$ when $\omega = \frac{2\pi}{N}k$.

Implement Algorithms

Directly follow the math

The first algorithm is straightforward, we simply follow the equation. For each $x[k]$ value, $N-1$ times additions are needed. For each addition result, N times complex multiplications are needed. Also, each complex multiplication needs 4 times real multiplications with 2 times real additions, total length of input signal N, both real number additions and real number multiplication have complexity growth of order N^2.

Fast Fourier Transform

FFT takes advantage of following properties:

We note $e^{-j\frac{2\pi}{N}kn}$ as W_N^{nk}, then:

1 $\left(W_N^{nk}\right)^* = W_N^{-nk} = W_N^{(N-n)k}$

2 $W_N^{(n+N)k} = W_N^{n(k+N)} = W_N^{nk}$

3 $W_{mN}^{mnk} = W_N^{nk}, W_N^{nk} = W_{\frac{N}{m}}^{\frac{nk}{m}}$

Now, we can set input signal length $N = 2^L$. If the signal is not long enough, we can always do zero padding to meet the length requirement. We can divide our $X[k]$ result by the value of n in sigma function. We now have:

$$X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk} = \sum_{\substack{n=0\\ even}}^{N-1} x[n]\, W_N^{nk} + \sum_{\substack{n=0\\ odd}}^{N-1} x[n]\, W_N^{nk}$$

We divide $X[k]$ into two parts, the first part has all even values of n while the second part has all odd values of n. We use $2r$ and $2r+1$ to represent even and odd numbers, then have:

$$\sum_{\substack{n=0\\ even}}^{N-1} x[n]\, W_N^{nk} + \sum_{\substack{n=0\\ odd}}^{N-1} x[n]\, W_N^{nk} = \sum_{r=0}^{\frac{N}{2}-1} x[2r]\, W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]\, W_N^{(2r+1)k}$$

Also, since $W_N^{(2r+1)k} = W_N^{2rk} * W_N^k$ and property 3:

$$\sum_{r=0}^{\frac{N}{2}-1} x[2r]\, W_N^{2rk} + \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]\, W_N^{(2r+1)k} = \sum_{r=0}^{\frac{N}{2}-1} x_1[r]\, W_{\frac{N}{2}}^{rk} + W_N^k * \sum_{r=0}^{\frac{N}{2}-1} x_2[r]\, W_{\frac{N}{2}}^{rk}$$

Recall the definition of $X[k] = \sum_{n=0}^{N-1} x[n]\, W_N^{nk}$, the original equation becomes:

$$X[k] = \sum_{r=0}^{\frac{N}{2}-1} x_1[r]\, W_{\frac{N}{2}}^{rk} + W_N^k * \sum_{r=0}^{\frac{N}{2}-1} x_2[r]\, W_{\frac{N}{2}}^{rk} = X_1[k] + W_N^k * X_2[k]$$

Here, $k \in [0, \frac{N}{2} - 1]$. Then, for $k \in [\frac{N}{2}, N-1]$ part, because of property 2:

$$X_1\left[\frac{N}{2} + k\right] = X_1[k]$$

$$X_2\left[\frac{N}{2} + k\right] = X_2[k]$$

Consider $W_N^{\frac{N}{2}+k} = W_N^{\frac{N}{2}} * W_N^k = -W_N^k$, the second half is:

$$X\left[k + \frac{N}{2}\right] = X_1[k] - W_N^k * X_2[k]$$

Still, $k \in [0, \frac{N}{2} - 1]$.

Since we now have:

$$X[k] = X_1[k] + W_N^k * X_2[k]$$

$$X\left[k + \frac{N}{2}\right] = X_1[k] - W_N^k * X_2[k]$$

We can compute DFT $X[k]$ by computing $X_1[k]$ and $X_2[k]$, which have half-length of original $x[k]$. We could save around half of workload for one division. I mentioned before that $x[k]$ has the length of 2^n, this divide and conquer can be done multiple times till we have length 1 of $x_1[k]$ and $x_2[k]$. Time complexity of this FFT algorithm is $n \log n$.

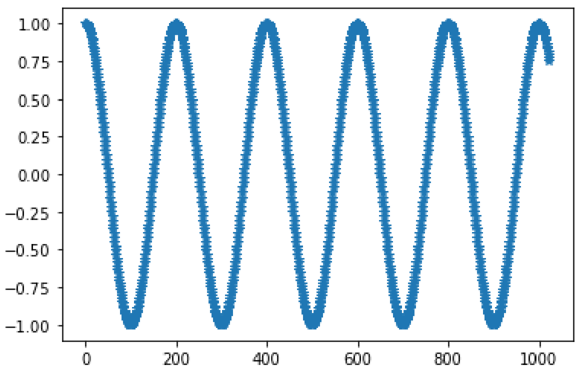
Code sample

This function is used to generate testing signals:

```
def generate_sinusoid(N, A, f0, fs, phi):
    '''''
    N(int) : number of samples
    A(float) : amplitude
    f0(float): frequency in Hz
    fs(float): sample rate
    phi(float): initial phase

    return
    x (numpy array): sinusoid signal which lenght is N
    '''

    T = 1/fs
    n = np.arange(N)    # [0,1,..., N-1]
    x = A * np.cos( 2 * f0 * np.pi * n * T + phi )

    return x
```

This is a simple signal with following parameters:

```
N = 1024
A = 1
f0 = 10
fs = 2000
phi = 0
```



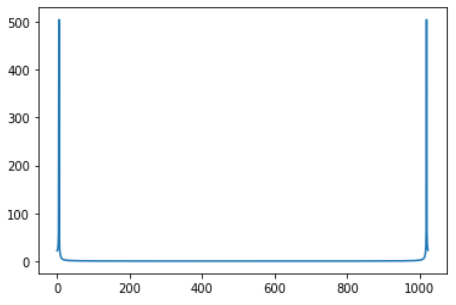
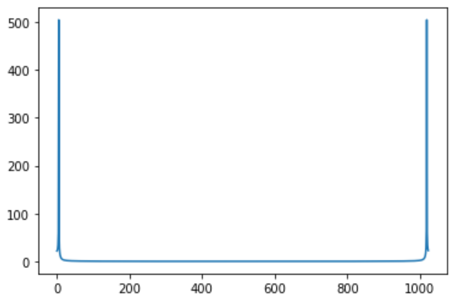
Now, this is DFT function:

```python
def DFT(x):
    """Compute the discrete Fourier Transform of the 1D array x"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)
```

And FFT function:

```python
def FFT(x):
    """A recursive implementation of the 1D Cooley-Tukey FFT"""
    x = np.asarray(x, dtype=float)
    N = x.shape[0]

    if N <= 1:  # this cutoff could be optimized
        return DFT(x)
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = np.exp(-2j * np.pi * np.arange(N) / N)
        return np.concatenate([X_even + factor[:N // 2] * X_odd,
                               X_even + factor[N // 2:] * X_odd])
```

Take Fourier transform by both methods and plot the real part:

Also, we can check the correctness by compare with python build-in FFT function:

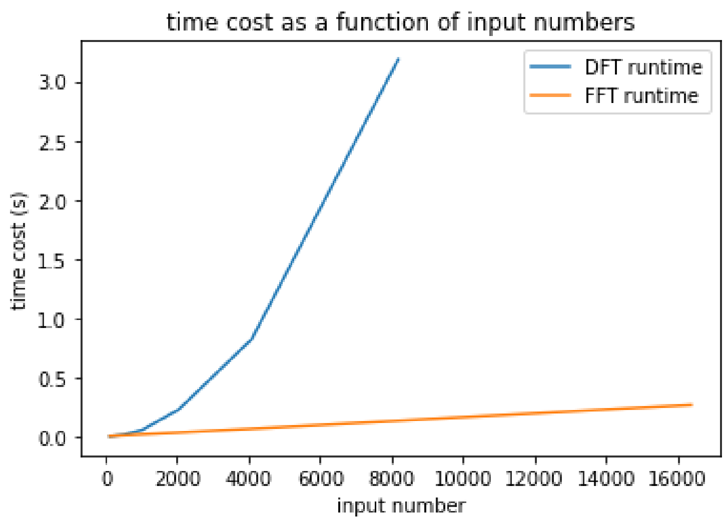
```
np.allclose(DFT(x), np.fft.fft(x))
np.allclose(FFT(x), np.fft.fft(x))
```

Both results are true.

Performance analysis

In this part, I use different length of input signals and remain other parameters same. Run both DFT and FFT functions shown above, time and plot the execution time to compare.

Input samples	128	256	512	1024	2048	4096	8192	16384
DFT runtime(s)	0.0	0.0049	0.0139	0.0528	0.2253	0.8218	3.1835	
FFT runtime(s)	0.0019	0.0050	0.0089	0.0169	0.0319	0.0638	0.1316	0.2652



This performance is basically in line with expectations.

Conclusions

FFT improves the time complexity from order N^2 into $n \log n$ by three properties. It saves time and make real time Fourier transform more realistic. Fourier transform is widely used in image processing, voice processing and other signal processing cases. A typical example of its usage could be music identification. The software may cut a long piece of music into many short pieces and do FFT for each to get a frequency sequence of the long music. In this process, information of this music is greatly reduced but many key information is saved. Then, compare the results of FFT and the database of music information, name of the music can be searched.

This FFT algorithm were popularized in 1965, but some related ideas had been derived earlier. In later years, FFT algorithm has kept improving. More complex ideas will not be introduced in this term paper, but a little phenomenon was seen when I test the performance. Python library Numpy has a build-in FFT function, this function runs rapidly. It takes almost instant for input size around 100k samples. This is interesting. Its implementation may be a further topic of this term paper.

Reference

Wiki page: https://en.wikipedia.org/wiki/Fast_Fourier_transform
Code samples: https://cloud.tencent.com/developer/article/1087336

More codes are available on GitHub: https://github.com/markblitz/RU_573_term_paper