Connor Berson MATH 364

FFT Exploration

December 3, 2020

Introduction

Suppose we have a continuous signal $f(t)$, where t is a unit of time. Then the **Continuous Fourier Transform (CFT)** of $f(t)$ is

$$F(k) = \int_{-\infty}^{\infty} f(t)e^{-kti}dt \tag{1}$$

$F(k)$ is the decomposition of $f(t)$ into a function with respect to frequency k (the reciprocal of the period). While frequency is not an intuitive input for a function, it has useful implications.

Here are a few examples of practical applications of the Fourier Transform:

- The absolute value of $F(k)$ is the amount of frequency k present in $f(t)$
- Differentiation with respect to time corresponds to multiplication by the frequency

Furthermore, the Fourier Transform has an inverse, so if an operation is simpler to do in respect to frequency, but we want a result in respect to time, the operation can be completed in respect to frequency and then inverted back to be in respect to time or vice versa.

However, in many cases, we find that we do not have the whole continuous signal at our disposal. Instead, we only have samples of the signal, but we still want to work with it the same way. In these cases, we utilize the **Discrete Fourier Transform (DFT)**.

Discrete Fourier Transform

Deriving DFT from CFT

Suppose that we only know of N sample values of $f(t)$ that are separated exactly by time T. Therefore, since the integral (1) is only defined at those N samples separated exactly by time T, we can rewrite it as

$$F(k) = \int_{0}^{(N-1)T} f(t)e^{-kti}dt \tag{2}$$

Now, the integral can be rewritten as a sum.

$$F(k) = f[0]e^{-i\cdot 0} + f[1]e^{-iTk} + \cdots + f[N-1]e^{-i(N-1)Tk} = \sum_{n=0}^{N-1} f[n]e^{-inTk} \tag{3}$$

where $f[n]$ is the *impulse* of area $f[n]$. The **Impulse / Dirac Function** has a *sampling* property. That is, if we multiply an impulse function $\delta(t-T)$ delayed by time T by a signal

$\phi(t)$, we sample the signal $\phi(t)$ at $t = T$. Notice that this property means that (3) is summing samples at various T.

Why the anti-derivative of each term is the impulse function $f[n]$ as well as further discussion on the impulse function will be left unexplained as it is outside the scope of this project. For more information regarding both, I found pages 19 - 21 of this lecture on Engineering Analysis by Peter Y K Cheung to be very informative.

We are working with a finite number of points (our N samples) in the context of a signal, so we will assume that the data is periodic with period N. Since we are assuming that our data has period N, we will evaluate it using the *fundamental frequency*, the greatest common divisor of all frequency components of a signal. Therefore, we have 1 cycle per sequence $\frac{1}{NT}$Hz, which is equivalent to $\frac{2\pi}{NT}$ rad / sec.

Finally, we have that the DFT is defined as

$$F(k) = \sum_{n=0}^{N-1} f[n]e^{-\frac{2\pi i}{N}nk} \tag{4}$$

What is the DFT?

First, let's define some notation as follows. I will refer to all N sample values $f[n] = x_n$ where $0 \leq n < N$ as $\{x_n\}$ and all of the outputs of the DFT $F(k) = X_k$ where $0 \leq k < N$ as $\{X_k\}$. That is,

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} \tag{5}$$

Since the DFT is discrete by definition, this notation will allow clearer and more intuitive manipulations of the sum.

The DFT transforms the equally-spaced sequence of complex numbers $x_0, x_1, \ldots, x_{N-1}$ (sampled from some function f) into another same-length equally-spaced sequence of complex numbers $X_0, X_1, \ldots, X_{N-1}$ (values of a function of frequency F). The DFT is of the utmost importance in the realm of *digital signal processing*.

When $\{x_n\}$ is proportional to samples of some continuous function $f(t)$, there exists a periodic summation of the CFT $F(k)$. Therefore, DFTs can be used to partially recover $F(k)$ and $f(t)$ (under certain conditions, they can even be recovered exactly!). This means that when sending a signal, if it were to pass through noise, DFTs allow us to *theoretically* recover the original signal.

In reality, using only the DFT to accomplish signal recovery is impractical because of the sheer number of necessary computations.

Algorithm

The time complexity of the DFT algorithm is $O(N^2)$ because to compute all $\{X_k\}$, (5) must be computed for all $0 \leq k < N$. In the pseudo code below, this is illustrated by the nested *while* loops. The inner loop performs N computations for each of the N iterations of the outer loop, which totals to N^2 total computations.

Algorithm 1 Vanilla DFT

```
function DFT(samples)                                   ▷ samples is $\{x_n\}$
    $N \leftarrow$ size of samples
    $result \leftarrow$ empty list that is size $N$          ▷ result is $\{X_k\}$
    $k \leftarrow 0$
    while $(k < N)$ do
        $total \leftarrow 0 + 0i$                           ▷ total is $X_k$
        $n \leftarrow 0$
        while $(n < N)$ do                              ▷ this computes (5)
            $total \leftarrow total + samples[n] \cdot e^{\frac{-2\pi}{N}nk}$
            $n \leftarrow n + 1$
        end while
        $result[k] \leftarrow total$
        $k \leftarrow k + 1$
    end while
    return result
end function
```

I have also provided an implementation of the DFT algorithm in Python that does not rely on any libraries. Note that words highlighted in red are keywords in Python and words highlighted blue are functions defined by the programmer. CEXPO, CMUL, CADD, and CSUB compute the complex exponential function, multiplies 2 complex numbers, adds 2 complex numbers, and subtracts 2 complex numbers respectively. Each was implemented by representing the complex number z as a tuple (x, y). For the complete implementation of these methods, see **img_utils.py**.

```python
def DFT(samples):
    N = len(samples)
    result = []
    k = 0
    while k < N:
        total = (0, 0)
        n = 0
        while n < N:
            total = CADD(total, CMUL(samples[n], CEXPO(-2 * pi * n * k / N)))
            n += 1
        result.append(total)
        k += 1
    return result
```

Algorithm 1: Vanilla implementation of DFT in Python

Algorithms with time complexity $O(N^2)$ are generally too slow to use for nontrivial inputs as the run-time would be far too long to be practical. For this reason, the vanilla DFT algorithm can only theoretically solve digital signal processing problems. In response, **Fast Fourier Transform (FFT)** algorithms were created to reduce the time complexity.

Fast Fourier Transform

While there are several FFT algorithms, I will be focusing on the most commonly used algorithm: **The Cooley-Tukey FFT Algorithm (CTFFT)**. CTFFT takes advantage of the symmetry of the periodic DFT.

Suppose we are computing some complex number X_k. Let's separate the DFT into 2 sums made of its even and odd indexed elements,

$$X_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} = \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}2mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m+1)k}$$

We can factor $e^{-\frac{2\pi i}{N}k}$ out of the sum of the odd terms,

$$\begin{aligned} X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}2mk} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}2mk} e^{-\frac{2\pi i}{N}k} \\ &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N}2mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N}(2m)k} \end{aligned}$$

Lastly, we can rewrite the exponents within each sum as

$$\begin{aligned} X_k &= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-\frac{2\pi i}{N/2}mk} + e^{-\frac{2\pi i}{N}k} \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-\frac{2\pi i}{N/2}mk} \\ &= E_k + e^{-\frac{2\pi i}{N}k} O_k \end{aligned}$$

Therefore, we have shown that X_k, the DFT made of N samples, can be obtained by computing E_k and O_k, DFTs made of $\frac{N}{2}$ samples.

Now, we can take advantage of the symmetry caused by DFTs being periodic. Since E_k and O_k are periodic in k with period of $\frac{N}{2}$, only $\frac{N}{2}$ values need to be computed.

Notice, that we could repeat this process until we only have DFTs made of 2 samples. We would separate E_k and O_k into DFTs made of $\frac{N}{4}$ samples, then separate the results into DFTs made of $\frac{N}{8}$ samples, and so on.

Altogether, we would repeat this process h times where $N = 2^h$. You may think it is too restrictive for N to be a power of 2, but since N is the number of samples from a continuous signal, we can choose N to be a power of 2. If there is a case where you want to use a N that is not a power of 2, other FFT algorithms can be used instead.

Applying log to both sides gives us $h = \log_2 N$ steps total. Therefore, we have reduced the number of total computations necessary from N^2 down to $N \log_2 N$, which is a much faster time complexity!

Algorithm

The time complexity of the CTFFT algorithm is $O(N \log N)$ as demonstrated above. The implementation provided uses a recursive approach for readability, but iterative approaches can allow further time reduction.

Algorithm 2 Recursive CTFFT

```
 1: function CTFFT(samples)
 2:     $N \leftarrow$ size of samples
 3:     if $N <= 1$ then
 4:         return samples
 5:     end if
 6:     evens $\leftarrow$ Recursively call $FFT$ on even indexed samples
 7:     odds $\leftarrow$ Recursively call $FFT$ on odd indexed samples
 8:     odds $\leftarrow$ Compute odds$[k] \cdot e^{\frac{-2\pi i}{N}k}$ for all $k < N/2$
 9:     left $\leftarrow$ Compute evens$[k]$ + odds$[k]$ for all $k < N/2$
10:     right $\leftarrow$ Compute evens$[k]$ − odds$[k]$ for all $k < N/2$
11:     return concat(left, right)
12: end function
```

I have also provided an implementation of the CTFFT algorithm in Python that does not rely on any libraries and uses the same styling distinctions defined for the DFT implementation.

```python
def CTFFT(samples):
    N = len(samples)
    if N <= 1:
        return x
    else:
        evens = FFT(x[0::2])
        odds  = FFT(x[1::2])

        odds  = [CMUL(CEXPO(-2 * pi * k / N), odds[k])
                                for k in range(N // 2)]
        left  = [CADD(evens[k], odds[k]) for k in range(N / 2)]
        right = [CSUB(evens[k],odds[k]) for k in range(N / 2)]
        return left + right
```

Algorithm 2: Vanilla implementation of CTFFT in Python

Optimization

Determining the minimal number of complex arithmetic steps needed to compute a DFT is an open computer science problem. Some argue that the minimum is the $N \log_2 N$ computations done in CTFFT algorithm, but the claim is unproven. With that being said, as of right now, any further optimizations for computing DFTs come from utilizing efficient programming strategies rather than taking advantage of mathematical properties.

I decided to explore possible optimizations of the vanilla recursive CTFFT algorithm I provided earlier. As a disclaimer, I did not extensively research the previous work done to further improve the CTFFT and other FFT algorithms prior to implementing my own. It is likely that these optimizations were already discovered, but I thought it would be an interesting experiment to wrap up my discussion nonetheless. Additionally, even though I decided to focus on improving the recursive version of CTFFT, the improvements could be implemented in the iterative approach for further speed up.

Algorithm

The time complexity of my optimization of CTFFT is still $O(N \log N)$, but I managed to slightly reduce the amount of work done per computation. I noticed 3 changes that provided some time reduction.

Algorithm 3 Recursive Optimized CTFFT

```
 1: $a \leftarrow -2\pi$                                    ▷ Compute once prior to starting the algorithm
 2: function OPT_CTFFT($samples$)
 3:     $N \leftarrow$ size of $samples$
 4:     if $N <= 1$ then
 5:         return samples
 6:     end if
 7:     evens $\leftarrow$ Recursively call $FFT$ on even indexed $samples$
 8:     odds $\leftarrow$ Recursively call $FFT$ on odd indexed $samples$
 9:     odds $\leftarrow$ Compute odds$[k] \cdot e^{\frac{ai}{N}k}$ for all $k < N/2$
10:     k $\leftarrow$ 0
11:     left, right $\leftarrow$ Initialize empty lists
12:     while ($k < N/2$) do
13:         $(x_2, y_2) \leftarrow$ evens[k]                ▷ Unpack even real and imaginary parts
14:         $(x_1, y_1) \leftarrow$ odds[k]                 ▷ Unpack odd real and imaginary parts
15:         left $\leftarrow$ Append $(x_1 + x_2, y_1 + y_2)$
16:         right $\leftarrow$ Append $(x_1 - x_2, y_1 - y_2)$
17:         k $\leftarrow$ k + 1
18:     end while
19:     return concat(left, right)
20: end function
```

I outlined the 3 modifications that I made below.

- For each of the $N \log_2 N$ computations done during the CTFFT algorithm, we multiply -2π. Instead, I computed the value of -2π once prior to beginning the algorithm.
- When computing the *left* and *right*, we iterate the same loop twice. I combined these 2 loops into one.
- Additionally, the computation of *left* and *right* involves the sum and difference of 2 complex numbers z_1 and z_2. Because the sum and difference were implemented as 2 separate method calls, z_1 and z_2 were unpacked into the real and imaginary parts twice for each computation. Instead, the imaginary numbers were unpacked once and computed in place.

I have also provided the implementation of my optimized CTFFT algorithm in Python that does not rely on any libraries and uses the same styling distinctions defined for the DFT implementation.

```
a = -2 * pi
def OPT_CTFFT(samples):
    N = len(samples)
    if N <= 1:
        return samples
    else:
        evens = OPT_CTFFT(samples[0::2])
        odds  = OPT_CTFFT(samples[1::2])
        odds  = [CMUL(CEXPO(a * k / N), odds[k]) for k in range(N // 2)]

        left  = []
        right = []
        for k in range(N / 2):
            x_1, y_1 = evens[k]
            x_2, y_2 = odds[k]
            left.append((x_1 + x_2, y_1 + y_2))
            right.append((x_1 - x_2, y_1 - y_2))

        return left + right
```

Algorithm 3: Optimized implementation of CTFFT in Python

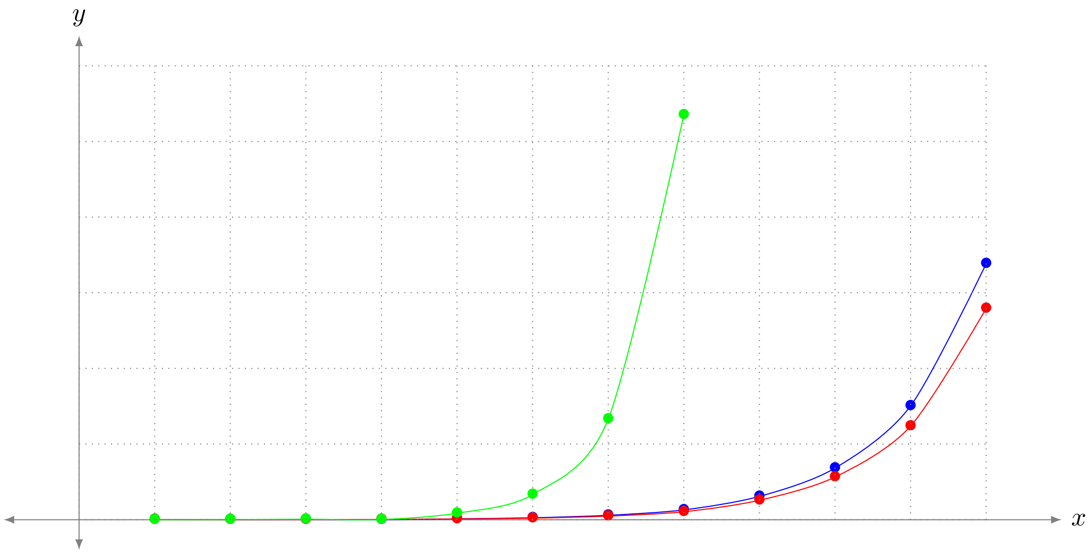
Now, let's consider the performance of each algorithm. To evaluate the 3 algorithms for comparison, I generated 200 random lists of complex numbers, with positive real and imaginary parts less than 10, for each size $N = 2^h$ where $h \in \mathbb{N}, 1 \leq h \leq 12$ and then timed the performance of each algorithm on every list.

To condense the data 1, 2, 3 refer to the CTFFT, OPT_CTFFT, and DFT respectively, and *Tot.* and *Avg.* mean Total Time and Average Time. The unit of all time related data is in seconds and has been rounded to the 5th decimal place where applicable.

N	Tot. 1	Tot. 2	Tot. 3	Avg. 1	Avg. 2	Avg. 3
2	0.006 23	0.005 22	0.001 03	3.11e-06	2.61e-06	5.17e-06
4	0.002 10	0.001 77	0.003 84	1.05e-05	8.85e-06	1.91e-05
8	0.006 32	0.005 13	0.016 44	3.16e-05	2.56e-05	8.22e-05
16	0.012 41	0.010 40	0.053 86	6.21e-05	5.20e-05	0.000 25
32	0.029 20	0.024 08	0.214 29	0.000 15	0.000 12	0.001 07
64	0.065 22	0.052 33	0.833 77	0.000 33	0.000 26	0.004 17
128	0.146 62	0.119 72	3.327 65	0.000 73	0.000 60	0.016 64
256	0.324 43	0.263 49	13.379 20	0.001 622	0.001 32	0.066 90
512	0.770 38	0.642 98	57.229 84	0.003 85	0.003 21	0.286 14
1024	1.704 88	1.409 77	233.636 40	0.008 52	0.007 05	1.168 18
2048	3.761 44	3.093 66	929.643 73	0.018 81	0.015 47	4.648 22
4096	8.460 98	6.980 46	2851.469 86	0.042 30	0.034 90	19.257 35
8192	28.521 33	22.959 90		0.142 61	0.114 80	
16 384	39.998 81	33.848 23		0.199 99	0.169 24	

As N increases, I observed that the gap of performance of all 3 algorithms to grow as expected, which confirms the correctness of their respective implementations. For $N \geq 8192$, I could not use the vanilla DFT algorithm as it took too long to complete, so the total and average times for those N were omitted from the table.

To visualize the improvement of each algorithm, I created the below graph. The x-axis has a step size of 1 and is $\log_2 N$ where N is the sample size. The y-axis has step size of 2 and is the total time in seconds each algorithm took to complete. The green points are the DFT, the blue points are the CTFFT, and the red points are my optimized CTFFT.



In conclusion, time saving the CTFFT algorithm provides over the DFT algorithm is invaluable. The time savings from my optimized CTFFT algorithm, while starting to become relevant for larger N, will only become valuable for extremely large N.

Resources

- Wikipedia
  - Discrete Fourier Transform: https://en.wikipedia.org/wiki/Discrete_Fourier_transform
  - Fourier Analysis: https://en.wikipedia.org/wiki/Fourier_analysis#Discrete-time_Fourier_transform_(DTFT)
  - Fast Fourier Transform: https://en.wikipedia.org/wiki/Fast_Fourier_transform
  - Fourier Transform: https://en.wikipedia.org/wiki/Fourier_transform
- Lectures / Texts
  - Engineering Analysis DE2.3 Electronics 2 by Peter Y K Cheung: http://www.ee.ic.ac.uk/pcheung/teaching/DE2_EE/Lecture%201%20-%20Time%20Domain%20view%20of%20signals%20(x1).pdf
  - Fundamental Period of Continuous Time Signals: https://opencourses.emu.edu.tr/pluginfile.php/43038/mod_resource/content/0/Fundamental%20Period%20of%20Discrete%20Time%20Signals.pdf
  - Lecture 7 - The Discrete Fourier Transform: http://www.robots.ox.ac.uk/ sjrob/Teaching/SP/l7.pdf
- Tutorials
  - Understanding the FFT Algorithm: https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/
  - Fast Fourier Transform - Python: https://rosettacode.org/wiki/Fast_Fourier_transform#Python