Fast Fourier Transform Implementation and Testing

Michael Seaman

CPSC 406, Erik Linstead

May 17 2017

***Abstract*— The Fast Fourier Transform (FFT) is an heuristic improvement upon the Discrete Fourier Transform (DFT), created by Cooley and Tukey in 1965. The decrease in complexity from $\Theta(n^2)$ to $\Theta(n \log n)$ arises from the exploitation of a property of complex roots of unity known as the halving lemma; this allows the problem to be approached with a divide-and-conquer strategy. Ultimately, the resulting algorithm has significant implications to many fields of science and engineering.**

I. Introduction and Applications

The Discrete Fourier Transform is one of the most powerful and widely used mathematical tools in all of engineering. Using it allows the decomposition of a discrete signal into a spectrum of complex frequencies and amplitudes, or in other words, allows a signal to be broken up into the waves that form it. This is extremely useful in the fields of digital signal processing, remote sensing, structural engineering, and even digital music!

This transformation can be derived from the more general Continuous Fourier Transform, which is in the form of an improper integral:

$$X(i\omega) = \int_{-\infty}^{\infty} x(t)e^{-i\omega t}dt$$

Where ω is the angular frequency of the resulting wave [2]. This general form, however, has little use outside of theoretical physics and mathematics. Instead, the DFT replaces the Riemann integral with an infinite sum and calculates the complex coefficients corresponding to a discrete, finite frequency space.

$$X_k = \sum_{j=0}^{n-1} x_j \omega_n{}^{kj}$$

Where $\omega_n \in \mathbf{C}$ is the n-th complex root of unity, satisfying the following equation:

$$(\omega_n)^n = 1$$

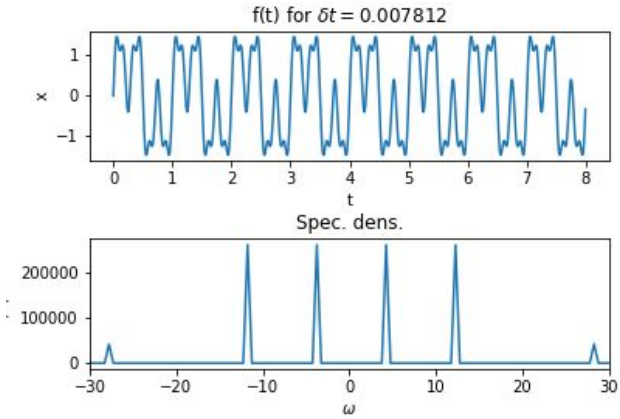
(Which, more visually, also correspond rotations of $\frac{2\pi}{n}$ away from $1 + 0i$ in the complex plane). Ultimately the resulting vector X contains the ordering of complex amplitudes, corresponding to the angular frequencies derived from n. Since these amplitudes are not very useful in their complex form, what we often do is calculate "spectral density" to look at strictly real amplitudes [1].

$$\text{Spectral Density} = |A|^2$$

Where A is the complex amplitude found from the Fourier Transform. Finally, we can plot spectral density against the corresponding frequencies to create a spectroscopy.

Here I have plotted the arbitrary function $f(t)$ from $t \in [0, 8)$ for timesteps of $\delta t = \frac{1}{128}s$. Below that is its spectral decomposition.

$$f(t) = sin(2\pi \cdot t) + sin(2\pi \cdot 3t) + .4sin(2\pi \cdot 7t)$$



The looming problem with calculating the DFT is its bulky $\Theta(n^2)$ runtime. This can be sped up drastically by exploiting the 'halving lemma'- a property of complex numbers - and then using a divide and conquer approach to break up every step into 2 DFTs of the even indices and odd indices. Unfortunately, this leads to the limitation that signals must be of length that is exactly a power of 2 (although ways around this do exist).

II. History

The Fourier transform (in its general form) was first used by French mathematician Jean-Baptiste Joseph Fourier in 1807 in a paper regarding heat transmission in solids. Until the rise of modern computing, it was used in a solely mathematical setting, often times to solve ordinary PDEs [3].

However, with the rise of digital signal processing, the discrete form of the transform became used extremely commonly. Unfortunately, machines running the brute force algorithm were described as "using up hundreds of hours of machine time." It wasn't until in 1965 that computer scientists J.W. Cooley and John Tukey developed the FFT algorithm still utilized today were machines running these computations able to be sped up drastically [3].

III. Pseudo-Code

```
RECURSIVE-FFT(a)
 1  n = a.length
 2  if n == 1
 3      return a
 4  ω_n = e^(2πi/n)
 5  ω = 1
 6  a^[0] = (a_0, a_2, ..., a_(n-2))
 7  a^[1] = (a_1, a_3, ..., a_(n-1))
 8  y^[0] = RECURSIVE-FFT(a^[0])
 9  y^[1] = RECURSIVE-FFT(a^[1])
10  for k = 0 to n/2 - 1
11      y_k = y_k^[0] + ω y_k^[1]
12      y_(k+(n/2)) = y_k^[0] - ω y_k^[1]
13      ω = ω ω_n
14  return y
```

Pseudo-code from [1].

IV. Mathematical Analysis

Because the DFT needs to access all elements in the input for every element for the input (this structure essentially becomes to for-loops), the time complexity of the DFT algorithm very naturally becomes $\Theta(n^2)$. However, in the case of the FFT, we have a more interesting result. Because at every recursion, we split the input into 2 halves, recurse, recombine (a linear time operation), and perform some constant time operations, we get the following recursion relationship:

$$T(n) = 2T(\tfrac{n}{2}) + n$$

Which can luckily be solved using the master method

$$n^{\log_b a} = n \quad f(n) = n$$

Case 2:

$$T(n) = \Theta(f(n) \cdot \log_2 n)$$

$$T(n) = \Theta(n \log n)$$

Ultimately showing that we obtain a logarithmic speed up from the verbose DFT.

V. Empirical Analysis

In order to test the performance of these algorithms (the DFT and FFT) both were implemented in Python (using the numPy library) and C++.

First, in the C++ implementation, both the DFT and FFT are created in complete accordance with source [1]. Because of this, the size of input A must strictly be a power of 2. The implementation uses the `complex<double>` type in the standard C++ library, making the data type size 16 bytes. The table below shows the runtime measurements on 3 different inputs: a small, medium, and large array of real values generated by $f(t)$ as seen on page 1.

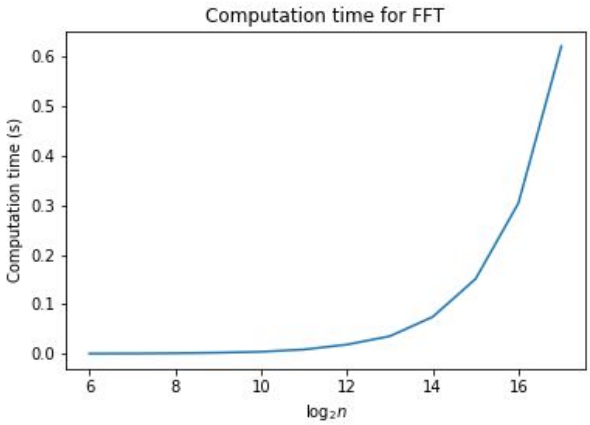
TABLE I
Computation time in C++ for FFT and DFT

Input Size	# of data points	Size of input	Computation Time	
			DFT	FFT
Small	1024	16.4KB	186.710 ms	1.614 ms
Medium	32768	524KB	>>1 minute	59.381 ms
Large	1048576	16.8MB	>>1 minute	2.167246 s

The python implementation is based largely off of the work of source [4]. The DFT is calculated as a matrix multiplication of the input vector and a operator of the complex roots of unity. The FFT, in term, has a more lenient base case than the C++

implementation, which resorts to the DFT for inputs less than n=32. This gives it a bit of leniency in its input size, not being strictly a power of 2, but instead being any multiple of 2 times any number less than 32. $n \in \{k \cdot 2^j \mid k \in [1, 32] \mid \forall j\}$.

The python implementation is, as expected, is significantly slower, to the point that the DFT algorithm stops being feasible for even small n. Below is a plot of computation time in seconds as a function of the size of the input from $n = 2^6$ to $n = 2^{18}$.



VI. Conclusions

The Fourier Transformation is an invaluable tool in countless fields of mathematics, physics, computing and engineering. Being able to do so in $\Theta(n \log n)$ time as opposed to quadratic - in accordance with the Cooley-Tukey FFT algorithm - creates a speed up that enables signals that are very long or have very high sample rate to be analyzed very quickly on modern hardware.

References

[1] Cormen, T. H. (2007). Introduction to algorithms. Cambridge (Massachusetts): The MIT Press.

[2] Wang, R. (n.d.). Continuous Fourier Transform. Retrieved May 17, 2017, from http://fourier.eng.hmc.edu/e101/lectures/Image_Processing/node1.html

[3] Cooley, J., Lewis, P., & Welch, P. (1967). Historical notes on the fast Fourier transform. Proceedings of the IEEE, 55(10), 1675-1677. doi:10.1109/proc.1967.5959

[4] VanderPlas, J. (n.d.). Understanding the FFT Algorithm. Retrieved May 18, 2017, from https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/

All source code used for this report can be found on the repository https://github.com/seama107/FFT