Implementation and Comparison of Different FFT Techniques

Justin Wang, Chandler Watson

December 7, 2019

1 Introduction

The discrete Fourier transform is arguably the most remarkable and practical tool in signal processing and data analysis. The ability of the Fourier transform to compactly capture nuances in time signals or images has contributed greatly to scientific innovation. In biological settings especially, the Fourier transform has helped scientists with edge detection in cellular images, development of hearing aids to transform audio signals, and analysis of echocardiograms to detect heart murmurs, among many other applications. The Fast Fourier transform (FFT) algorithm proposed by James Cooley and John Tukey brought the Fourier transform to the forefront of computing. In this project, we present a handful of FFT implementations in the context of image processing with the hope to explore implementations other than the popular Cooley-Tukey useful for image data.

1.1 Discrete Fourier Transform

The discrete Fourier transform (DFT) is used to convert a series of data, which can be represented a vector or matrix, into the frequency domain. In other words, it models the original series of data as the sum of multiple sinusoidal functions, each with different amplitudes, phases, and frequencies. Doing so often allows for a more compact form of representation of the original data, and can be used to filter out signals from the data as well.

The DFT in image processing is a 2D DFT, since each of the images used in this project can be represented as a 2D matrix, whose values represent pixel intensities. The 2D DFT is calculated as follows where k and l are pixel locations in the transformed image and the image has dimensions M by N.

$$F(k,l) = \sum_{b=0}^{M-1} \sum_{a=0}^{N-1} f(a,b) e^{-i2\pi(\frac{ka}{M} + \frac{lb}{N})}$$

We can express the 2D DFT instead as two 1D DFTs in succession.

$$
\begin{aligned}
F(k,l) &= \sum_{b=0}^{M-1} \sum_{a=0}^{N-1} f(a,b) e^{-i2\pi(\frac{kb}{M} + \frac{la}{N})} \\
&= \sum_{b=0}^{M-1} P(k,a) e^{-i2\pi \frac{kb}{M}}
\end{aligned}
$$

where $P(k,a) = \sum_{a=0}^{N-1} e^{-i2\pi \frac{la}{N}}$.

1.2 Cooley-Tukey FFT

The Cooley-Tukey FFT relies on a divide and conquer approach to produce an $O(n \log n)$ implementation of the DFT. Given that the 2D DFT can be expressed as two series of 1D DFTs in succession, we only need to consider the 1D DFT. Let X be a 1D array of data. The Cooley-Tukey DFT first divides X into vectors of even and odd indexed entries. The 1D DFT becomes

$$
\begin{aligned}
X(k) &= \sum_{a=0}^{N-1} e^{-i2\pi \frac{ka}{N}} \\
&= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-i2\pi \frac{k*2m}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-i2\pi \frac{k*(2m+1)}{N}} \\
&= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-i2\pi \frac{k*2m}{N}} + \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-i2\pi \frac{k*2m}{N}} e^{-i2\pi \frac{k}{N}} \\
&= \sum_{m=0}^{\frac{N}{2}-1} x_{2m} e^{-i2\pi \frac{km}{N/2}} + e^{-i2\pi \frac{k}{N}} * \sum_{m=0}^{\frac{N}{2}-1} x_{2m+1} e^{-i2\pi \frac{km}{N/2}}
\end{aligned}
$$

In other words, for each element in the 1D array, we calculate the 1D DFT on the even numbered indexes and add that to the result from the 1D DFT on the odd numbered indexes multiplied by some fudge factor. But those 1D DFTs can also be calculated using this same technique, hence the "divide and conquer" approach.

1.3 Split-radix

This form of the FFT is a variation of the Cooley-Tukey algorithm. Specifically, it takes the odd indices in the Cooley-Tukey algorithm and further splits them into two parts: one whose indexes are 1 (mod 4) and another whose indexes are 3 (mod 4). Overall, we split the DFT into three sums which we can recursively solve for. This algorithm is designed such that the compute required for managing twiddle factors is minimal.

1.4 Vector-radix

One way to compute a 2D FFT is to apply any 1D FFT algorithm over the rows, then the columns in the *row-column method*. Vector-radix instead takes a different tactic, applying the decomposition that Cooley-Tukey does to both axes at once. In the radix-2 case, this yields a decomposition into four DFTs which can be combined with twiddle factors. In particular,

$$\begin{aligned} X_{k,r} = &\sum_{i=1}^{N/2-1}\sum_{j=0}^{N/2-1} x_{2i,2j} e^{i2\pi\frac{ik+jr}{N/2}} + e^{i2\pi\frac{k}{N}} \sum_{i=1}^{N/2-1}\sum_{j=0}^{N/2-1} x_{2i+1,2j} e^{i2\pi\frac{ik+jr}{N/2}} \\ &+ e^{i2\pi\frac{r}{N/2}} \sum_{i=1}^{N/2-1}\sum_{j=0}^{N/2-1} x_{2i,2j+1} e^{i2\pi\frac{ik+jr}{N/2}} + e^{i2\pi\frac{k+r}{N/2}} \sum_{i=1}^{N/2-1}\sum_{j=0}^{N/2-1} x_{2i+1,2j+1} e^{i2\pi\frac{ik+jr}{N/2}}. \end{aligned}$$

1.5 Prime-factor

Yet another useful property of the Fourier transform is that it lends itself to an FFT without twiddle factors (like in Cooley-Tukey and related implementations) in the special case that the length l and width w can each be factored into $l = l_1 l_2$ and $w = w_1 w_2$ where l_1, l_2 and w_1, w_2 are pairwise relatively prime. As mentioned before, 2D FFTs can be broken down into a series of 1D DFTs, so we will now consider a 1D array X of length N.

This implementation relies on the Chinese Remainder Theorem (CRT) to create a bijective mapping (one-to-one correspondence) between an index n in the input array and two indexes i, j, as well as an index k in the output array and two indexes k_1, k_2. Specifically, these indexes must satisfy:

$$n = i(N_2 n_2) + j(N_1 n_1)$$

$$k_1 = kN_2 \pmod{n_1}$$

$$k_2 = kN_1 \pmod{n_2}$$

$$k = (n_2 k_1 + n_1 k_2) \pmod{N}$$

where $N = n_1 n_2$ and n_1, n_2 are relatively prime.

This way, we get a 2D DFT written as

$$X(k_1, k_2) = \sum_{j=0}^{n_2-1} \left(\sum_{i=0}^{n_1-1} X(i,j) e^{-i2\pi\frac{ik_1}{N_1}} \right) e^{-i2\pi\frac{jk_2}{N_2}}$$

1.6 Rader FFT

Rader's FFT algorithm is another algorithm for computing discrete Fourier transforms, this one specialized for lists of prime numbers. Dually to how the Cooley Tukey FFT exploits symmetries in the sum representation of the DFT for composite N, Rader's FFT uses a number-theoretic approach to exploit the structure of integers modulo prime N.

In particular, the algorithm uses the fact that the integers modulo N, written as $\mathbb{Z}_N$, form a group: a set under some operation (here, addition) that has identity, associativity, and inverses. In particular, the group $\mathbb{Z}_N$ contains the subgroup $\mathbb{Z}_N^\times$, defined as the integers mod N which have a multiplicative inverse mod N. When N is prime, this is every non-zero element of $\mathbb{Z}^N$. The goal of Rader's FFT algorithm is to find a *generator* $g \in \mathbb{Z}^N$: an element where, multiplied by itself mod N over and over, eventually yields every element in $\mathbb{Z}^\times$.

This property of N allows us to rewrite our sum into the form

$$X(0) = \sum_{n=0}^{N} x_n$$

$$X(g^{-p}) = x_0 + \sum_{q=0}^{N-2} x_{g^q} e^{-\frac{2\pi i}{N} g^{-(p-q)}}$$

where the sums in the latter family of equations are precisely the elements of the cyclic convolution of the sequences $\{x_{g^q}\}$ and $\{e^{-\frac{2\pi i}{N} g^{-q}}\}$. Just as in class, such a convolution can be computed by taking a product in the frequency domain: thus, we can use a traditional FFT algorithm (in our case, the built-in `numpy` FFT) to compute this convolution, the Rader FFT having effectively split up one large prime DFT into two smaller cases.

2 Results

We implemented each of the $O((MN)^2)$ DFT, Cooley-Tukey FFT, split-radix FFT, and vector-radix FFTs for 2D images, along with 1D implementations of the prime-factor algorithm FFT and Rader's FFT. To ensure that our implementations were consistent with each other, we computed the Fourier transform using each implementation on an image provided in class. Metrics provided result from running our algorithms on the `sample.gif` image. We also compared our results with the `fft` module provided in Python's `numpy` package. The resultant transforms for the images were the same given different images.

FFT Implementations on `sample.gif`				
Trial	Cooley-Tukey	Split-radix	Vector-radix	`numpy`
1	1.687	1.883	1.462	0.00562
2	1.633	1.944	1.457	0.00360
3	1.609	1.930	1.469	0.00682
4	1.596	1.934	1.463	0.00428
5	1.653	1.967	1.396	0.00403

The slow DFT would take approximately 18 hours to run on the 256 by 256 pixel image provided, as it takes approximately one second to calculate one value in the resultant image.

As a precursor to further endeavors in applying FFTs to images, we also implemented the prime-factor FFT (PFT) and Rader FFT in 1-dimension. For the PFT implementation, we used an array of size 20, while for the Rader implementation we used an array of size 19. The `numpy` implementation was run on both arrays–we average the runtime for the two trials.

1D FFT Implementations			
Trial	PFT	Rader	`numpy`
1	0.00164	0.01438	0.00156
2	0.00163	0.00553	0.00381
3	0.00152	0.00514	0.00238
4	0.00150	0.00527	0.00151
5	0.00173	0.00447	0.00139

3 Discussion

Regarding our implementations only, we see that for images, the vector-radix FFT is the quickest and the split-radix FFT is the slowest of the three. We suspect that this is the case because vector radix intentionally acts on both dimensions at once, rather iterating over rows, then columns, while the 1D split-radix FFT both necessitates row-column iteration and is inherently more complicated than Cooley-Tukey. It is expected, however, that if all algorithms were implemented and aggressively optimized, that split-radix would win out: however, because its main advantage (optimized twiddle factors) has not been capitalized on at the machine level in our implementation, it confers no great benefit.

Our implementations were three orders of magnitude slower than the `numpy` implementation. This holds

for a number of reasons. For one, for pedagogical purposes the methods chosen here are fairly simple: clever tricks based on adapting to image size, primality of image dimension factors, etc. have been foregone for simplicity and readability of our end result. `numpy`'s FFT, on the other hand, has extensive "plan generation" techniques (see `https://github.com/numpy/numpy/tree/master/numpy/fft`).

Additionally, all of our code is implemented in pure Python for simplicity, whereas the Numpy implementation uses C bindings, and beyond that makes very careful optimizations around twiddle factor computation. While our FFT implementations are strong pedagogical tools for understanding how FFTs work, the `numpy` FFT has been aggressively optimized in ways ours hasn't.

For our 1D implementations, we found that Rader's FFT took 2-3 times longer. We were especially alarmed that the PFT performed as well if not better than `numpy`'s implementation by finding a unique indexing to avoid solving for "twiddle factors" needed in Cooley-Tukey-like implementations. We could have saved even more time had we decided to utilize some other FFT algorithm (or the PFT again) on the N_1 and N_2 sub-DFTs rather than running brute DFT.

Major design decisions include using `numpy` for vectorized additions under the hood. Some challenges include sifting through very outdated papers regarding various FFTs, as well as deciding how to best utilize `numpy` to handle various FFT implementations, from the number-theoretical Rader FFT to the classic Cooley Tukey FFT.

Through this project, we gained much more intuitive understanding of the DFT and how to manipulate it mathematically. More specifically, we gained a greater appreciation for the various symmetries of the 2D DFT and how the 2D Cooley-Tukey FFT is a simple extension from the one dimensional case.

Next steps include further optimizing the runtime of the Cooley-Tukey implementation hopefully to the extent of the `numpy` implementation. We also would try other FFT implementations and see which are most optimal for 2D images. Finally, we would move past grayscale images and try images with red, green, and blue channels.

Chandler and Justin contributed equally to the writeup. Justin wrote the pipeline for the image FFTs and wrote the DFT, Cooley-Tukey FFT, and PFT implementations. Chandler wrote the split-radix, vector-radix, and Rader's FFT implementations.

References

[1] Jake VanderPlas. *Python Perambulations: Understanding the FFT Algorithm* `https://jakevdp.github.io/blog/2013/08/28/understanding-the-fft/`. 2013.

[2] Wikipedia. Split-radix FFT algorithm.
https://en.wikipedia.org/wiki/Split-radix_FFT_algorithm. 2019.

[3] Wikipoedia. *Vector-radix FFT algorithm.*
https://en.wikipedia.org/wiki/Vector-radix_FFT_algorithm. 2019.

[4] Rich Radke. *DSP Lecture 12: The Cooley-Tukey and Good-Thomas FFTs*
https://www.youtube.com/watch?v=8cjDKirNIko. 2014.

[5] Shlomo Engelberg. *Elementary Number Theory and Rader's FFT*
https://epubs.siam.org/doi/pdf/10.1137/15M1044990. 2017.

[6] Wikipedia. *Rader's FFT Algorithm*
https://en.wikipedia.org/wiki/Rader%27s_FFT_algorithm. 2019.

[7] Pierre Duhamel and Martin Vetterli. *Fast Fourier Transforms: A Tutorial Review and a State of the Art*. https://www.sciencedirect.com/science/article/pii/016516849090158U. 1999.

[8] Various. *Modular Multiplicative Inverse Function in Python.*
https://stackoverflow.com/questions/4798654 2019.
(Gave the helpful advice of using the $N - 2^{\text{th}}$ power of g as g^{-1}.)