FFT Image Encoding

ACM Reference Format:
. 2018. FFT Image Encoding. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

The Fast Fourier Transform is one of the most important numerical algorithms of our time. First discovered by Gauss in 1805, and later rediscovered by Cooley and Tukey in 1965, the fast fourier transform allows us to compute the discrete fourier transform of a set of points in $O(Nlog(N))$ time instead of $O(N^2)$ time, as the naive algorithm would do. Various different fast fourier transform algorithms have since been discovered, and the fast fourier transform has been used in all kinds of applications, from signal processing to data compression.

2 History of the FFT

3 Discrete Fourier Transform

3.1 Discrete & Discrete Time Fourier Transforms

The discrete time fourier transform (DTFT) takes an infinite number of complex values, x_n, and returns a continuous function of frequency in the form of a fourier series. When the frequency is normalized, then the formula for the function is:

$$X(\omega) = \sum_{n=-\infty}^{\infty} x_n e^{-i\omega n}$$

It can be inverted by:

$$x_n = \frac{1}{2\pi}\int_0^{2\pi} X(\omega)e^{i\omega n}d\omega$$

When the data, x_n, are evenly spaced samples of a continuous function, $s(t)$, then the DTFT results in a periodic summation of its fourier transform, $\mathcal{F}\{s(t)\}$, with period $1/2\pi$. Because the DTFT is continuous and requires an infinite number of points, it isn't very computationally useful. The discrete fourier transform (DFT) takes a finite number of complex values, $x_n, n = 0..N-1$, and returns a different set of complex values $X_n, n = 0..N-1$. If the x_n's are evenly spaced samples of some function, then the X_n's are evenly spaced samples of the DTFT of that function. The DFT can be calculated as follows:

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

Its inverse can be expressed similarly or in terms of the DFT:

$$x_k = \frac{1}{N}\sum_{n=0}^{N-1} X_n e^{\frac{2\pi i}{N}kn}$$

$$\mathcal{F}^{-1}\{x_n\} = \frac{1}{N}\mathcal{F}\{x_{N-n}\}$$

where $x_N = x_0$. There are a few other ways of expressing the inverse transform in terms of the forward transform, but the one written above is the most computationally efficient. The DFT can also be expressed as a matrix vector multiplication. Given $\omega = 2\pi i/N$ and $\vec{x}$, the x_n's expressed as a column vector, the DFT would be $A\vec{x}$, where:

$$A = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2N-2} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega^{N-1} & \omega^{2N-2} & \dots & \omega^{(N-1)^2} \end{bmatrix}$$

3.2 2-D DFT

Given a set of complex values, $x_{n,m}, n = 0..N-1, m = 0..M-1$, the 2D discrete fourier transform returns a set of complex values $X_{n,m}, n = 0..N-1, m = 0..M-1$ according to the following equation:

$$\begin{aligned} X_{j,k} &= \sum_{n=0}^{N-1}\sum_{m=0}^{M-1} x_{n,m} e^{-\frac{2\pi i}{N}jn} e^{-\frac{2\pi i}{M}km} \\ &= \sum_{n=0}^{N-1}\left(e^{-\frac{2\pi i}{N}jn}\sum_{m=0}^{M-1} x_{n,m} e^{-\frac{2\pi i}{M}km}\right) \end{aligned}$$

The inner summation in the second of the equation is a DFT over each row, and the outer summation is a DFT over each column of the result. This means that it is possible to express the 2D DFT as two 1D DFTs. This result holds true for any number of dimensions. Because of this, we only implemented a 1D DFT and used it twice to implement our 2D DFT.

3.3 Different FFT Algorithms

3.3.1 Cooley-Tukey Algorithm

The Cooley-Tukey algorithm is by far the most common fast fourier transform algorithm. It works on any input of composite size $N = N_1N_2$. It does so by doing N_1 fourier

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

transforms of size N_2 followed by multiplications by twiddle factors and then N_2 fourier transforms of size N_1. The implementation and derivation are easiest when $N_1 = 2$, so the following assumes that (as did our implementation). Note that forcing $N_1 = 2$ means that the algorithm will only work when given $2^k | k \in \mathbb{N}$ values. Let E_k be the DFT of the even indexed values, and O_k be the DFT of the odd indexed values:

$$E_k = \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn}$$

$$O_k = \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn}$$

The DFT of the first half is:

$$\begin{aligned} X_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N} kn} \\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N} k*2n} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N} k(2n+1)} \\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn} + e^{-\frac{2\pi i}{N} k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn} \\ &= E_k + e^{-\frac{2\pi i}{N} k} O_k \end{aligned}$$

This is only the first half because k can only go up to $N/2-1$ because of the E_k and O_k. The second half is:

$$\begin{aligned} X_{k+N/2} &= \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}(k+N/2)*n} \\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N}(k+N/2)*2n} + \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N}(k+N/2)(2n+1)} \\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn} e^{-2\pi in} \\ &\quad + e^{-\frac{2\pi i}{N} k} e^{-\pi i} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn} e^{-2\pi in} \\ &= \sum_{n=0}^{N/2-1} x_{2n} e^{-\frac{2\pi i}{N/2} kn} - e^{-\frac{2\pi i}{N} k} \sum_{n=0}^{N/2-1} x_{2n+1} e^{-\frac{2\pi i}{N/2} kn} \\ &= E_k - e^{-\frac{2\pi i}{N} k} O_k \end{aligned}$$

This is how the simplest version of Cooley-Tukey works. It takes the DFT of the even and the odd elements, multiplies the odd elements by a certain term (called the twiddle factor) and then combines the two DFTs together. The fact that it has an addition in the first half and a subtraction in the second is because that is how a DFT of two elements looks. Because the even and odd DFTs are half the size, can also be computed recursively with Cooley-Tukey, and because they can be reused, the complexity of the algorithm is $O(Nlog(N))$ when N is significantly composite. In the event that N is prime or it reaches a prime sized base case quickly, then it must rely on a different algorithm. If it uses naive DFT in these cases, then the complexity is still $O(N^2)$, despite being several times faster. If it uses Rader's or Bluestein's algorithm, then it remains $O(Nlog(N))$.

3.3.2 Rader's Algorithm

Rader's algorithm is significantly more complex than Cooley-Tukey, and is thus used less often. Rader's algorithm only works on prime sized inputs. It does so by calculating the first element of the DFT in $O(N)$ time naively (summing all the elements), and then calculating the rest as a convolution of two different sets of size $N-1$ (gauranteed to be a composite). The convolution theorem states that the fourier transform of the convolution of two sets is the pointwise product of the transform of each set ($\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$). Thanks to this, Cooley-Tukey or another FFT algorithm can be applied to each of the sets, they can be multiplied pointwise, the DFT can be inverted (also using Cooley-Tukey), and then the result can be combined with the first term calculated earlier to get the DFT.

3.3.3 Bluestein's Algorithm

Bluestein's algorithm is even more complicated than Rader's algorithm. It calculates a generalization of the DFT called the Chirp Z-Transform (CZT), using FFT algorithms. It works on any size input (prime and composite), and it also relies on the convolution theorem. Even though it works on any size input, it is slower than Cooley-Tukey by a large constant. In practice, Cooley-Tukey is used, and either Bluestein's or Rader's algorithm is used for prime size base cases.

3.3.4 Other Algorithms

Other less common algorithms that compute the DFT quickly are Winograd's algorithm (any power of a prime), the prime-factor algorithm (works on sizes $N = N_1N_2$ where N_1 and N_2 are relatively prime), and Bruun's Algorithm (even composite sizes).

3.4 Modern Applications of DFT

4 Image Processing

4.1 Python Implementation

All code can be found at:
https://github.com/Anonymous-Stranger/fft-image-encoding

4.1.1 DFT

See 7.1 for our python implementation of the DFT. Our implementation of DFT doesn't use the matrix form of the equation. It uses two nested for loops, one to iterate over all X_k, and another to implement the summation over all x_n,

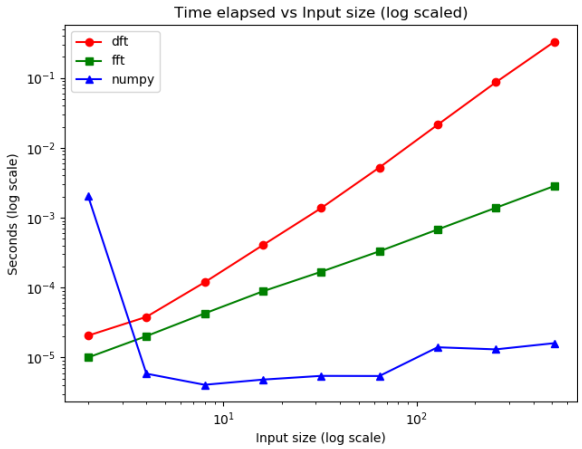
using the standard formula. The inverse DFT is implemented in terms of the DFT, by reversing all but the first element of the input, computing the DFT and dividing all elements by the size of the input. The 2D DFT is implemented as two separate DFTs, one over the rows, and the other over the columns. It does this by taking the DFT over the rows, transposing the matrix, doing the DFT over the rows of the result (stored in a temporary), and then returning the transpose of the result. Both the inverse and 2D discrete fourier transform implementations take a function as input. This allowed us to use one function to implement it for both the naive DFT and Cooley-Tukey.

4.1.2 FFT

See 7.2 for our python implementation of the FFT. Our implementation of Cooley-Tukey uses the simplest version, which only works on sets of data with a power of 2 size. It takes the input as a numpy array of complex values, uses python slicing to split into even and odd parts, recursively calling itself on each half (returning immediately when the size was 1), multiplying the odd values by the twiddle factors, and then concatenating the sum and difference of the even and odd values. The inverse and 2D discrete fourier transforms are implemented as described above.

4.2 Results

4.2.1 Performance



The graph above shows the performance of 3 different implementations of the discrete fourier transform on several different size inputs. The red line shows the time taken by the dft algorithm, the green line shows the time taken by our implementation of Cooley-Tukey, and the blue line shows the time taken by a library implementation. Because the distance between the red and green lines grows (on a log scale), it is easy to see that the naive algorithm is asymptotically slower than Cooley-Tukey. Aside from the anomaly at the first point, the reason the blue line looks as it does is because the library implements the algorithm in C, which is far faster than Python. As a result of this, not enough data points were tested for the algorithm to display its asymptotic behavior.

4.3 Discussion

5 Conclusion

6 References

7 Appendix A: Code

7.1 Naive DFT

```
import numpy as np
def dft(pts, k_max=None):
    k_max = k_max or len(pts)
    return np.array([dftk(pts, k) for k in range(k_max)])
def dftk(pts, k):
    coeff = -2*np.pi*k/len(pts)
    return sum((x * np.exp(complex(0, coeff*n)) for n, x in enumerate(pts)))
```

7.2 Cooley-Tukey

```
import numpy as np
def cooley_tukey(data):
    if(data.size==1): return data
    even = cooley_tukey(data[::2])
    odd = cooley_tukey(data[1::2])
    multiplier = np.exp(np.pi*-2j/data.size)
    k = 1
    for i in range(0,odd.size):
        odd[i] *= k
        k *= multiplier
    return np.concatenate((even+odd,even-odd))
```