The Fourier Transform: Variations and Applications

Sailesh Kaveti

1. The Fourier Transform

1.1. Developing an Intuition for the Fourier Transform.

Imagine we are given we are given a bucket of paint, and asked to replicate the color of that bucket. At first, it seems easy to simply describe it nominally, calling the bucket of paint as red, yellow, blue, or any other color. This is a valid first instinct. However, imagine you were given a purple bucket of paint. Even in this instance, most people are familiar with the color purple and we could just mix equal parts of blue and red. Imagine you were given a bucket of paint that was dark purple. We could still call this bucket "dark purple", but we would immediately run into problems when replicating the paint color. Do we need to put the colors on a 2:1 ratio or a 3:1 ratio? In general, given a color, it can be very useful to know the colors that compose that color, as opposed to just a nominal description. Now imagine that we were given a complex wave, and that we are interested in the simpler waves that compose that wave. The Fourier Transform helps us determine a periodic wave's composite frequencies.

1.2. The Fourier Series.

In order to understand the Fourier Transform, it is very important to gain understand the Fourier Series and its integral and the Fourier Transform is non-discretized extension of the Fourier Series. According to Dym and McKean, the authors of *Fourier Series and Integrals*, the basic idea of the Fourier Series is that any periodic function $f(t)$ can be expressed as a trigonometric sum of sines and cosines. More specifically, for the same period T for both *sine* and *cosine*, we can claim the following:

$$f(t) = \sum_{n=0}^{\infty} [\hat{f}_+(n) \cos\left(\frac{\pi n t}{T}\right) + \hat{f}_-(n) \sin\left(\frac{\pi n t}{T}\right)]$$

From this, we can derive the equations for $\hat{f}_+^a(n)$ and $\hat{f}_-^b(n)$. The equations are as follows

$$\hat{f}_+(n) = \frac{1}{L} \int_{-L}^{L} f(t) cos\Big(\frac{n\pi x}{L}\Big) dx$$

$$\hat{f}_-(n) = \frac{1}{L} \int_{-L}^{L} f(t) sin\Big(\frac{n\pi x}{L}\Big) dx$$

Putting these two together, we can get the Fourier Series, a formula to completely put a periodic function in terms of *sine* and *cosine*. For example, a non-analytic square wave $g(x)$, that has a value of $-h$ from $-\pi$ to 0 and h from 0 to pi with a vertical line from $-h$ to h can be modeled with the following Fourier Series:

$$g(x) = \sum_{n=0}^{\infty} \frac{sin((2n+1)x)}{(2n+1)}$$

The intuition behind a Fourier Series is very similar to a Taylor or MacLaurin Series, where a complex function can be modeled and very closely as an infinite sum of simpler functions.

1.3. The Computation of the Fourier Transform.

Similar to colors, given a complex wave, it is important to know the frequencies that compose that wave. The inverse of this can also provide us with valuable information. A wave $f(x)$ exists in real space, and its equivalent in frequency space is function $F(s)$. In short, $f(x)$ gives us a complex wave in real space, while $F(s)$ gives us a function with peaks at the frequencies that make up $f(x)$.

As described in *The Fourier Transforms and Its Applications*, the following relationship between F(s) and f(x) exists:

The Fourier Transform:

$$F(s) = \int_{-\infty}^{\infty} f(x)e^{-i2\pi xs}dx$$

The Inverse Fourier Transform:

$$f(x) = \int_{-\infty}^{\infty} F(s)e^{i2\pi xs}ds$$

Replacing any instances of $e^{i\theta}$ with $cos(\theta)+isin(\theta)$, we can derive the following equivalent expressions:

The Fourier Transform:

$$F(s) = \int_{-\infty}^{\infty} f(x)(cos(-2\pi xs) + isin(-2\pi xs))dx$$

The Inverse Fourier Transform:

$$f(x) = \int_{-\infty}^{\infty} F(s)(cos(2\pi xs) + isin(2\pi xs))ds$$

As a result, we can see a resemblance of a non-discretized version of the Fourier Series, which we can discretize for more interesting results in the following section.

2. The Discrete Fourier Transform

2.1. The Basics of the Discrete Fourier Transform.

2.2. How the Discrete Fourier Transform varies from the Fourier Transform.

3. The Computation of the Discrete Fourier Transform

3.1. A Rudimentary Algorithm for the Discrete Fourier Transform.

3.2. The Fast Fourier Tranform.

Before starting the Fast Fourier Transform (FFT), it is important to introduce a concept known as complex roots of unity. A complex n^{th} root of unity is a complex number ω such that $\omega^n = 1$. As the name suggests, there are a total of n distinct complex n^{th} roots of unity. In order to find all n complex roots of unity, we say $\omega_k = e^{\frac{2\pi ik}{n}}, \forall k \in \{0, 1, 2, ..., n-1\}$. When $k = 0$, we yield the trivial solution of $\omega = 1$.

Given the Discrete Fourier Transform, it seems in our interest to find a way to compute this value. In *Introduction to Algorithms*, Cormen, Leiserson, Rivest, and Stein introduce the FFT which they claim takes advantage of of the special properties of the complex roots of unity to compute $DFT_n(a)$ in $O(nlog(n))$. For the sake of this implementation, they assume at n is a power of two, mostly because the existing strategies for dealing with the case where n is not a power of two are quite complex. As the runtime of the algorithm suggests, the FFT is a divide and conquer algorithm. Given an $A(x)$, we can represent it as an array a of coefficients. In addition, we must also define the following intermediary functions that will be called in the algorithm:

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + ... + a_{n-2}x^{n/2-1}$$
$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + ... + a_{n-1}x^{n/2-1}$$

such that

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2)$$

The following algorithm to find the FFT of a is defined recursively by Cormen and is as follows:

Algorithm 1 Recursive Implementation of FFT

```
1:  procedure RECURSIVE-FFT(a)                ▷ A Recursive Implementation of FFT
2:      n = a.length
3:      if n == 1 then return a               ▷ Base Case of Recursion
4:      ω_n = e^{2πi/n}
5:      ω = 1
6:      a^[0] = (a_0, a_2, a_4, ..., a_{n-2})   ▷ Define Coefficients
7:      a^[1] = (a_1, a_3, a_5, ..., a_{n-1})
8:      y^[0] = RECURSIVE-FFT(a^[0])          ▷ Recursive Step
9:      y^[1] = RECURSIVE-FFT(a^[1])
10:     for k = 0 to n/2 − 1 do
11:         y_k = y_k^[0] + ω y_k^[1]         ▷ These hold true by definition of y_k
12:         y_{k+(n/2)} = y_k^[0] − ω y_k^[1]
13:         ω = ω ω_n                         ▷ Making sure that ω is updated properly
14:     return y                              ▷ Assumed that y is a column vector
```

From this algorithm, we can arrive at the **Convolution Theorem**, which states that for any two vectors a and b, both of which have a length n that is a power of two, the following holds true:

$$a \otimes b = DFT_{2n}^{-1}(DFT_{2n}(a)DFT_{2n}(b))$$

3.3. Alternatives for the Fast Fourier Tranform.

In *Introduction to Algorithms*, Cormen, Leiserson, Rivest, and Stein present certain improvements to existing implementation of the FFT. One notable improvement is an iterative implementation of the FFT. In Computer Science, iterative and recursive solutions are often considered equivalent. This is proved in the Church-Turing Thesis. However, there certainly runtime tradeoffs that we sometimes have to make when converting from recursion to iteration and vice-versa. That being said, we make no computational trade-offs in this instance, as our iterative solution still has a runtime of $O(nlog(n))$. In this instance, just like the recursive implementation, a must have a length that is a power of 2. The algorithm for an iterative approach to the FFT is as follows:

Algorithm 2 Iterative Implementation of FFT

```
1:  procedure BIT-REVERSE-COPY(a, A)
2:      n = a.length
3:      for k = 0 to n − 1 do
4:          A[rev(k)] = a_k ▷ rev(k) is the reverse of the bit-wise representation of k
5:  procedure ITERATIVE-FFT(a)
6:      BIT-REVERSE-COPY(a, A)
7:      for s = 1 to log_2(n) do
8:          m = 2^s
9:          ω_m = e^{2πi/m}
10:         for k = 0 to n − 1 by m do
11:             w = 1
12:             for j = 0 to m/2 − 1 do
13:                 t = ωA[k + j + m/2]
14:                 u = A[k + j]
15:                 A[k + j] = u + t
16:                 A[k + j + m/2] = u − t
17:                 ω = ωω_m
```

4. Applications of the Fourier Transform

4.1. Multiplication of Polynomials.

According to *Introduction to Algorithms*, the straightforward method of adding two polynomials takes $O(n)$ time. We know the following to hold true:

$$f(x) = \sum_{j=0}^{n} a_j x^j \tag{1}$$

$$g(x) = \sum_{j=0}^{n} b_j x^j \tag{2}$$

$$f(x) + g(x) = \sum_{j=0}^{n} (a_j + b_j) x^j \tag{3}$$

For two polynomials $f(x)$ and $g(x)$, with length k and n, we know that $f(x) + g(x)$ has length $max(k, n)$. Generate two arrays of length n called A_f and A_g where each index i represents the coefficient for term x^i for $f(x)$ and $g(x)$ respectively. Iterate through the arrays and add the value held at each index of $A_f[i]$ and $A_g[i]$ and set that value in $A_{f+g}[i]$. The resulting array represents the coefficients of $f(x) + g(x)$.

With this in mind, we can derive the straightforward algorithm for the multiplication of two polynomials. For the multiplication of the two polynomials, Cormen claims that the following holds true:

$$f(x)g(x) = \sum_{j=0}^{2n} c_j x^j \tag{4}$$

where

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{5}$$

From this information, we can see that equation (4) yields us with the sum of $2n$ polynomials, each of which was computed in $O(n)$ time. As we showed earlier, the sum of any two polynomials can be computed in $O(n)$ time, and since we must add $2n$ total polynomials, we get that the product of any two polynomials can be computed in $O(2n^2)$ time, which can be reduced to $O(n^2)$ time by the definition of big-O notation.

With our understanding of the FFT, Cormen claims that we can find the product of any two polynomials in $O(nlog(n))$ time, a significant asymptotic improvement over $O(n^2)$. Before starting the explanation of the algorithm, it is important to introduce another way to represent a polynomial, known as the point-value representation.

The point-value representation of a polynomial $A(x)$ is a set of pairs in the form $\{(x_0, y_0), (x_1, y_1), ..., (x_n, y_n)\}$ such that $y_i = A(x_i)$. These values can be generated in $O(n^2)$ time using Horner's Metho, but can be reduced to $O(nlog(n))$ time if the values of x are cleverly picked. From this definition, one can conclude that each polynomial has a many different point-value representations. Using this new representation of polynomials, we can indeed determine the product of any two polynomials in $O(n)$ time. Given this information, we can now explore the algorithm to find the product of any two polynomials in $O(nlog(n))$ time. Cormen argues that the following four steps will determine the product of any two polynomials $A(x)$ and $B(x)$ of length n:

(1) *Double degree-bound:* Create the coefficient representations of $A(x)$ and $B(x)$ of length $2n$ by padding n higher-order coefficients to the left of the array representations of $A(x)$ and $B(x)$. This can be done in $O(n)$ time.
(2) *Evaluate:* Compute the point-value representations of $A(x)$ and $B(x)$ of length $2n$ by applying the FFT of order $2n$ on each polynomial. As a result, we can get the representations at the $(2n)^{th}$ roots of unity. This is done in $O(nlog(n))$ time.
(3) *Pointwise Multiply:* Multiply these two point-value representations pairwise, yielding us with the value of $C(x) = A(x)B(x)$ at all roots of unity. As explained earlier, this is done in $O(n)$ time.
(4) *Iterpolate:* We can then create the coefficient representation of polynomial $C(x)$ by applying the FFT on the point-value representation of $C(x)$. This is done in $O(nlog(n))$ time.

This entire process is done in $O(n + nlog(n) + n + nlog(n))$ time, which can reduce to $O(nlog(n))$ time by the definition of big-O notation. Thus, using the above algorithm, one can multiply any two polynomials in $O(nlog(n))$ time.

4.2. Image Processing.