# The Fourier Transform: Variations and Applications

Sailesh Kaveti

Contents

## 1. Complex Analysis Fundamentals

### 1.1. Basic Properties of Complex Numbers.

Saff and Snider, in *Fundamentals of Complex Analysis with Applications to Engineering and Science* introduce a **complex number** as an expression in the form $a + bi$, where $a$ and $b$ must be real numbers and $i = \sqrt{-1}$. Two complex numbers $a + bi$ and $c + di$ can only be equal if and only if $a = c$ and $b = d$. For future reference, complex numbers will be referred to as $z$ where $z = a + bi$. A complex number $a + bi$ exists on the complex plane as a rectangular coordinate $(a, b)$. The following properties of complex numbers will be used over the course of this paper:

(1) $z = a + bi$
(2) $\bar{z} = a - bi$
(3) $|z| = \sqrt{a^2 + b^2}$
(4) $\overline{(z_1 z_2)} = \bar{z_1}\bar{z_2}$
(5) $Re(z) = a = \frac{z+\bar{z}}{2}$
(6) $Im(z) = b = \frac{z-\bar{z}}{2i}$
(7) $z\bar{z} = |z|^2$
(8) $|z_1 + z_2| \leq |z_1| + |z_2|$

A complex number with rectangular coordinates $(x, y)$ can be written in terms of the polar coordinates $(r, \theta)$ such that

$$x = rcos(\theta) \qquad y = rsin(\theta)$$

Conversely, a polar coordinate can be written in terms of rectangular coordinates such that

$$r = \sqrt{x^2 + y^2} = |z| \qquad tan(\theta) = y/x$$

It is important to note that

$$\theta = tan^{-1}\left(\frac{y}{x}\right)$$

does not necessarily hold true because $\theta$ can exist in different quadrants while $tan(\theta)$ still has the same value.

### 1.2. Roots of Unity.

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 $\omega$ 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 $\omega = 1$.

### 1.3. Euler's Formula.

In this subsection, we explore the relationship between the mathematical constant $e$ and the trigonometric functions *cosine* and *sine* as well as the imaginary constant $i$. Let $z = x + iy$.

$$\begin{aligned} z &= x + iy \\ e^z &= e^{x+iy} \\ &= e^x e^{iy} \end{aligned}$$

We can then differentiate the second term on the right hand side to determine that

$$\frac{de^{iy}}{d(iy)} = e^{iy}$$
$$\frac{de^{iy}}{dy} = ie^{iy}$$

Differentiating again, we get:

$$\begin{aligned}\frac{d^2e^{iy}}{dy^2} &= \frac{d}{dy}(ie^{iy}) \\ &= i^2e^{iy} \\ &= -e^{iy}\end{aligned}$$

We are left with the following differential equation, that if

$$g(y) = e^{iy}$$

then,

$$\frac{d^2g}{dy^2} = -g$$

and that any function in the form of

$$g(y) = Acos(y) + Bsin(y)$$

satisfies the differential equation. To evaluage A and B, we can use the following equations:

$$g(0) = e^{i0} = 1 = Acos(0) + Bsin(0)$$
$$\frac{dg}{dy}(0) = ie^{i0} = i = -Asin(0) + Bcos(0)$$

Thus,

$$A = 1 \qquad B = i$$

and

$$e^{iy} = cos(y) + isin(y)$$

Thus, **Euler's Formula** is as follows:

$$e^{i\theta} = cos(\theta) + isin(\theta)$$

## 2. The Fourier Transform

### 2.1. 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.

**2.2. 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 $\hat{f}_{+}(n)$ and $\hat{f}_{-}(n)$. 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.

**2.3. 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.

## 3. The Discrete Fourier Transform

### 3.1. The Basics of the Discrete Fourier Transform.

As the name suggests, the Discrete Fourier Transform (DFT) is a discretized, more applicable version of the regular Fourier Transform. According to Gonzalez and Woods, the authors of *Digital Image Processing*, the DFT is a derivation of the Fourier Transform and the Inverse DFT is a derivation of the Inverse Fourier Transform. The derivation for the DFT and IDFT are not included, but involve sampling a finite number of points from the Fourier Transforms.

The Fourier Transform:

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

The Discrete Fourier Transform:

$$F(s) = \frac{1}{N}\sum_{x=0}^{N-1} f(x)e^{-i2\pi xs/N}$$

The Inverse Fourier Transform:

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

The Inverse Discrete Fourier Transform

$$f(x) = \sum_{s=0}^{N-1} F(s)e^{-i2\pi xs/N}$$

In *The Discrete Fourier Transform*, Sundararajan claims that the DFT is able to transform a $N$-point time-domain sequence into a set of $N$ frequency coefficients that represent the given time-domain sequence in frequency domain. These coefficients represent magnitudes and the amplitudes of the composite *sine* and *cosine* counterparts. When these composite functions are added, the original time-domain sequence is represented. However, we are presented with a strange dilemma. It doesn't seem intuitive to represent a finite length sequence with a sum of infinite sinusoidal waves. However, any finite length sequence that is the sum of sinusoidal waves must be periodic, and thus can be extended periodically. On the other hand, the IDFT can transform the $N$ frequency coefficients back into the original set of $N$ time-domain samples but creating a composition of the sinusoidal waves to generate the original signal. Thus, every signal has two characterizations: its $N$ time-domain samples or its $N$ frequency coefficients.

There are also several interesting properties of the Discrete Fourier Transformation that hold true that allow for more informative results. An example of such a property is known as **Parseval's Theorem**. This theorem implies that the sums of the squared magnitudes of the signal and the DFT sequences are directly related by the constant $N$, which has already been established as the number of samples. Mathematically, we can write that

$$\sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X(k)|^2$$

Sundararajan explains the proof as the following:

Since the square of the magnitude of a complex number can be written in terms of its and its conjugate, the following holds true:

$$\sum_{n=0}^{N-1} |x(n)|^2 = \sum_{n=0}^{N-1} x(n)x^*(n)$$

Substituting the corresponding IDFT expressions for both x(n) and its conjugate, we can get

$$\begin{aligned} &= \sum_{n=0}^{N-1} \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} X(k)X^*(m)W_N^{-n(k-m)} \\ &= \frac{1}{N^2} \sum_{k=0}^{N-1} \sum_{m=0}^{N-1} X(k)X^*(m) \sum_{n=0}^{N-1} W_N^{-n(k-m)} \end{aligned}$$

When k = m, this expression can be simplified to

$$\begin{aligned} &= \frac{1}{N} \sum_{k=0}^{N-1} X(k)X^*(k) \\ &= \frac{1}{N} \sum_{k=0}^{N-1} |X(k)|^2 \end{aligned}$$

When this does not hold true, the equation reduced to zero because of the orthogonal property

## 4. The Computation of the Discrete Fourier Transform

### 4.1. The Fast Fourier Tranform.

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

**procedure** RECURSIVE-FFT(a) ▷ A Recursive Implementation of FFT
  $n = a.length$
  **if** $n == 1$ **then return** a ▷ Base Case of Recursion
  $\omega_n = e^{2\pi i/n}$
  $\omega = 1$
  $a^{[0]} = (a_0, a_2, a_4, ..., a_{n-2})$ ▷ Define Coefficients
  $a^{[1]} = (a_1, a_3, a_5, ..., a_{n-1})$
  $y^{[0]} =$ RECURSIVE-FFT$(a^{[0]})$ ▷ Recursive Step
  $y^{[1]} =$ RECURSIVE-FFT$(a^{[1]})$
  **for** $k = 0$ `to` $n/2 - 1$ **do**
    $y_k = y_k^{[0]} + \omega y_k^{[1]}$ ▷ These hold true by definition of $y_k$
    $y_{k+(n/2)} = y_k^{[0]} - \omega y_k^{[1]}$
    $\omega = \omega\omega_n$ ▷ Making sure that $\omega$ is updated properly
  **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:

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

**4.2. 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

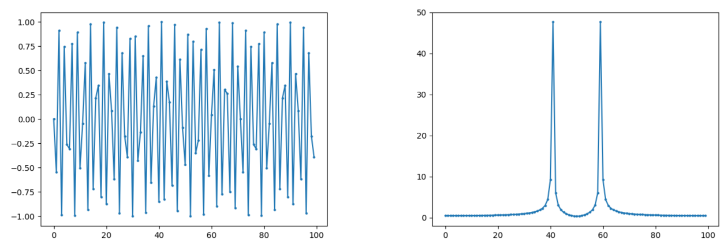
**procedure** BIT-REVERSE-COPY(a, A)<br>
    $n = a.length$<br>
    **for** $k = 0$ `to` $n - 1$ **do**<br>
        $A[rev(k)] = a_k$ ▷ $rev(k)$ is the reverse of the bit-wise representation of k<br>
**procedure** ITERATIVE-FFT(a)<br>
    BIT-REVERSE-COPY(a, A)<br>
    **for** $s = 1$ `to` $log_2(n)$ **do**<br>
        $m = 2^s$<br>
        $\omega_m = e^{2\pi i/m}$<br>
        **for** $k = 0$ `to` $n - 1$ `by` $m$ **do**<br>
            $w = 1$<br>
            **for** $j = 0$ `to` $m/2 - 1$ **do**<br>
                $t = \omega A[k + j + m/2]$<br>
                $u = A[k + j]$<br>
                $A[k + j] = u + t$<br>
                $A[k + j + m/2] = u - t$<br>
                $\omega = \omega\omega_m$

**4.3. A Python Implementation of the Fast Fourier Transform.** In order to better understand the results of the Fast Fourier Transform, it may be useful to write code to explore a Pythonic implementation, and plot the signals in order to visualize the results of the Fast Fourier Transform. There are existing libraries in Python to aid with the visualization of the Fast Fourier Transformation. These include numpy and matplotlib. Numpy is useful for existing libraries that make it possible to manipulate an entire array with a single operation as well as contains the function to perform the Fast Fourier Transformation. Matplotlib is the library that contains the function to fully plot a dataset. Given a signal, the following method plots both the signal in real space as well as in frequency space:

```
def plot_signal(signal):
    signal_fft = np.fft.fft(signal)
    signal_fft_pow = np.abs(signal_fft)
    plt.plot(signal, '-o', markersize = 2)
    plt.savefig('RealSpace.png')
```

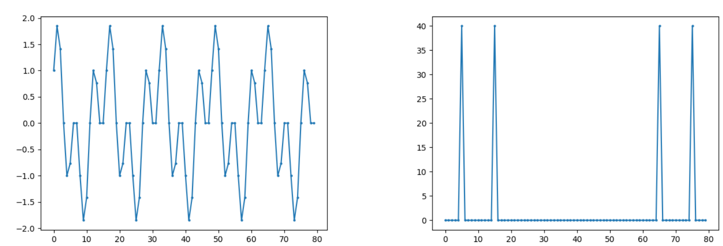
```
plt.clf()
plt.plot(signal_fft_pow, '-o', markersize = 2)
plt.savefig('FrequencySpace.png')
```

With that information, we are able to quickly find the FFT of different waves in real space in frequency space. Here are a few examples generated as a result of the python code included above.



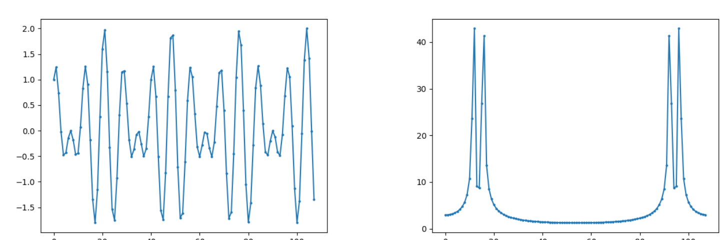
(A) Real Space (B) Frequency Space

FIGURE 1. Wave 1 and its Fast Fourier Transform



(A) Real Space (B) Frequency Space

FIGURE 2. Wave 2 and its Fast Fourier Transform



(A) Real Space (B) Frequency Space

FIGURE 3. Wave 3 and its Fast Fourier Transform

## 5. Applications of the Fourier Transform

### 5.1. 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.

**5.2. Image Processing.**
According to Gonzalez, "an image may be defined as a two-dimensional function, $f(x, y)$ where $x$ and $y$ are *spatial* (plane) coordinates, and the amplitude of $f$ at any pair of coordinates $(x, y)$ is called the *intensity* or *gray level* of the image at that point". In short, an image can be represented by a function that determines the color at a certain pixel. Digital image processing is one of the most widespread fields of study, having applications in virtually every STEM related profession. In addition, digital imaging processing is also not bound to visible light spectrum. We can perform image processing is the Gamma spectrum, X-ray spectrum as well as the Ultraviolet Band. Included are a few examples of imaging outside the visible light spectrum.

(A) Gamma-Ray Imaging (B) X-Ray Imaging (C) Ultraviolet Imaging

FIGURE 4. Imaging across various sources

In every variation of the Fourier Transform above, we were working in one dimension. If we were to follow a similar derivation described above, we can indeed arrive as a two-dimensional DFT and a two-dimensional IDFT.

Two-Dimensional Discrete Fourier Transform

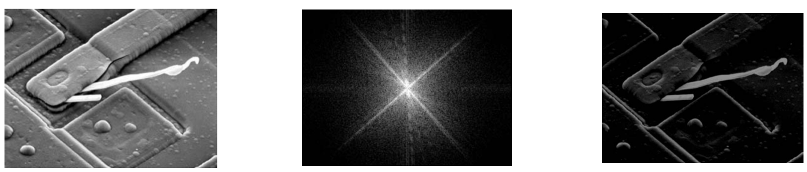
$$F(u,v) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-i2\pi(ux/M + vy/N)}$$

Two-Dimensional Inverse Discrete Fourier Transform

$$f(x,y) = \frac{1}{MN} \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} F(u,v) e^{i2\pi(ux/M + vy/N)}$$

In this case, $f(x,y)$ is a digital image of size $M \times N$, and just like in the 1-D case, must be evaluated for the discrete values $u$ and $v$ in the ranges $u = 0, 1, 2, ..., M-1$ and $v = 0, 1, 2, ..., N-1$

The two-dimensional DFT is an essential part of modern image processing and is essential in various aliasing and filtering techniques. For example, Gonzalez includes following application of the Discrete Fourier Transform:



(A) A Damaged Circuit (B) DFT on the Circuit (C) Modified IDFT (Dark)

Figure 5. Results of DFT and IDFT

In order to better explain the two-dimensional Discrete Fourier Transform, it may help to suggest a Python implementation for the algorithm to automate the conversion between an image and its Fourier Transform. Just like the one-dimensional Fast Fourier Transform, there are multiple existing tools and libraries to aid with the visualization of the two-dimensional Discrete Fourier Transform. Like the implementation of the 1-D FFT, the method to visualize the 2-D DFT involved numpy and matplotlib. In order to generate images, the 2-D DFT method also involves a library known as Pillow/PIL. Given an image, the following method displays the image as well as it's 2-D Discrete Fourier Transform:

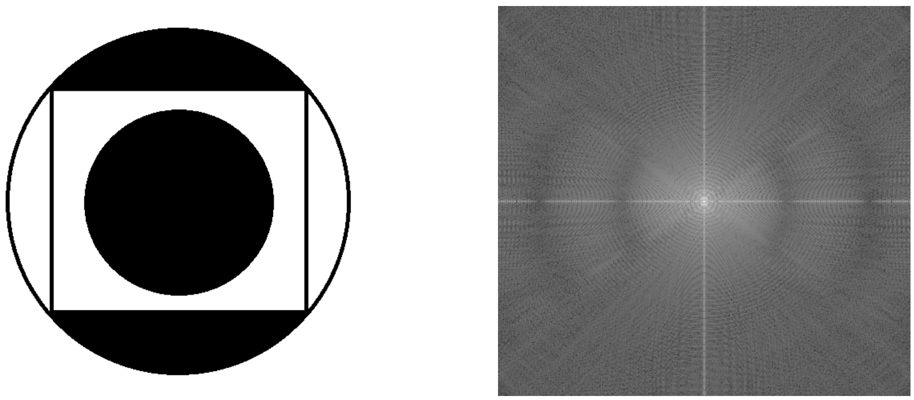
```
def plot_2d_dft(image):
    img = Image.open(image)
    img_arr = np.asarray(img)
    fourier = np.fft.fft2(img_arr)
    fourier = np.fft.fftshift(fourier)

    fourier = np.abs(fourier)
    fourier = fourier + 0.0000001
    fourier = np.log10(fourier)

    lowest = np.nanmin(fourier[np.isfinite(fourier)])
    highest = np.nanmax(fourier[np.isfinite(fourier)])
    original_range = highest - lowest
```

```
norm_fourier = (fourier - lowest) / original_range * 255
fourier_img = Image.fromarray(norm_fourier.astype('uint8'))
fourier_img.save('2ddft1.png')
```

With this method, we are able to generate the two-dimensional DFT for any black and white image. Here is an example of the method being run on the image on the left:



(A) Original Image (B) Fourier Transform

Figure 6. The Two-Dimensional DFT of an Image

### 5.3. Audio Compression.

## 6. An Analysis of Principal Component Analysis and FFT

### 6.1. A Brief Introduction to Principal Component Analysis.

Principal Component Analysis (PCA) is a machine learning algorithm that is a method of dimensionality reduction. Given a set of tuples or vectors that has $n$ attributes, PCA determines $k$ $n$-dimensional orthogonal vectors that can nearly equivalently represent the data. This is a very important algorithm to reduce the number of attributes in a dataset while still retaining the statistical differences between tuples. Han, Kember, and Pei, the authors of *Data Mining: Concepts and Techniques*, describe the algorithm for PCA as the following:

(1) The input data are normalized, so that each attribute falls within the same range. The normalization scale that is chosen is mostly a design decision, but the most common is $z$-score normalization. This step helps ensure that the attributes with large domains will not dominate attributes with smaller domains.
(2) PCA then computes $k$ orthonormal vectors that provide a basis for the normalized input data. These are unit vectors that each point in a direction perpendicular to the others. These are unit vectors that each point in a direction perpendicular. This means that the dot-product of any two vectors must be 0. These vectors are commonly refered to as the *principal components*. The input data are a linear combination of the principal components.

(3) The principal components are then sorted in order of decreasing "significance" or strength. The principal components essentially serve as a new set of axes for the data, providing important information about variance. That is, the sorted axes are such that the first axis shows the most variance among the data, the second axis shows the next highest variance, and so on.
(4) As a result, the data size can be reduced by eliminating the weaker components,

**6.2. A Comparison between PCA and FFT.**

## 7. Bibliography