Student Name: Daniel Alconchel Vázquez
Semester: 1
Academic Year: 2022/2023

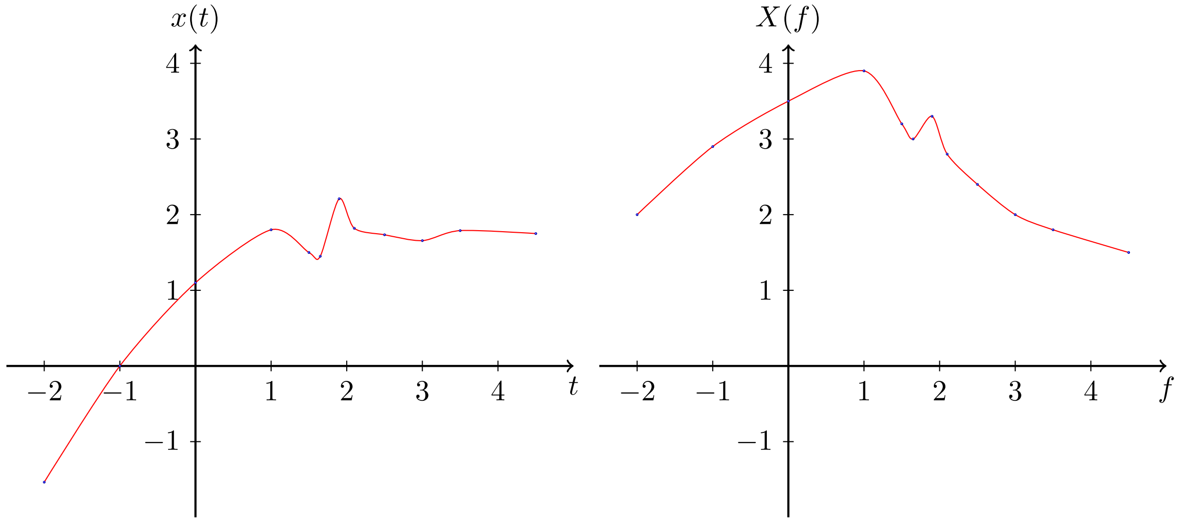
Faculty of Applied Physics and Mathematics
Institute of Physics and Applied Computer Science
Fast Fourier Transform (FFT)

---

1. Theoretical Introduction

Before talking about Fast Fourier Transform we should talk briefly about the Fourier Transform and the Discrete Fourier Transform(DFT).

Suppose we have a mathematical function in the time domain. Lets call it $x(t)$. Usually, it will be a signal.

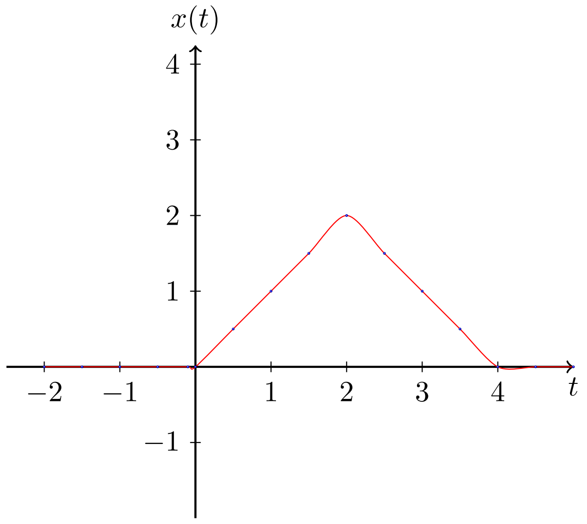


The Fourier Transform is another mathematical function that allows us to express the original signal as a function of frequency. We will call it $X(f)$.

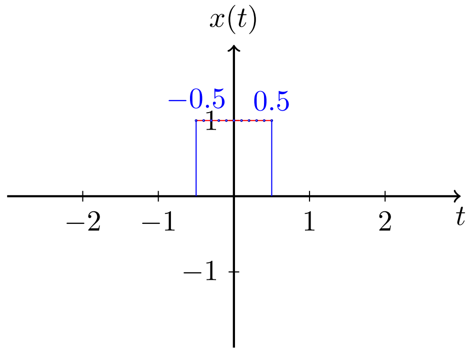
Then we have the following equality:

$$X(f) = \int_{-\infty}^{+\infty} x(t)e^{-j2\pi ft}\,dt$$

But, why do we talk about frequency if we may have signals that are not periodic? Consider for example the following signal:



We can't seem to see that there is frequency. This is an interesting question, so lets consider the easiest example. Lets take the next function:



So we have that:

$$X(f) = \int_{-\infty}^{+\infty} x(t)e^{-j2\pi ft}\,dt = \int_{-1/2}^{1/2} e^{-j2\pi ft}\,dt = \frac{e^{-j2\pi ft}}{e^{-j2\pi f}}\bigg|_{-1/2}^{1/2} = \frac{e^{-j\pi f} - e^{j\pi f}}{-j2\pi f} = \frac{(-1)(e^{j\pi f} - e^{-j\pi f})}{(-1)(2j\pi f)}$$

Remember that the following equalities always hold:

$$\sin(\alpha) = \frac{e^{j\alpha} - e^{-j\alpha}}{2j} \quad \cos(\alpha) = \frac{e^{j\alpha} - e^{-j\alpha}}{2}$$

And using that:

$$sinc(f) = \frac{\sin(\pi \cdot f)}{\pi \cdot f}$$

Using this we have:

$$\frac{(-1)(e^{j\pi f} - e^{-j\pi f})}{(-1)(2j\pi f)} = \frac{e^{j\pi f} - e^{-j\pi f}}{2j\pi f} = \frac{\sin(\pi \cdot f)}{\pi \cdot f} = sinc(f)$$

The idea lies in analyzing the frequency as an idea of variation, so if I have a signal that varies a lot in a short time, that is, that presents sudden jumps, that is associated with sudden changes and "we can see it as cosines or sines high frequency".

Now that we have a base we ask ourselves the following question:

What is the Fast Fourier Transform?

So, FTT is the computation of Discrete Fourier Transforms (DFT) into an algorithmic format. In fact, it is a way to compute Fourier Transforms much faster than using the conventional method.

Lets take a look at DFT to get some idea of how FFT makes computing results faster.

DFT transforms a sequence of N numbers

$$x_n := x_0, x_1, ..., x_{N-1}$$

to a sequence of another set of complex numbers

$$X_k := X_0, X_1, ..., X_{N-1}$$

Without going over the entire theorem, DFT is basically taking any quantity or signal that varies over time and decomposing it into its components (e.g. frequency). This can be the pixels in an image or the pressure of sound in radio waves (used in DSP chips).

DFT can be written using the following formula:

$$X(k) = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}$$

Where $x(n)$ is a discrete signal and N represents the size of the domain.

The problem with DFT is that it takes more calculations to arrive at an answer. This is because you have to do N(multiplications) with N(additions), which in Big Θ Notation is $\Theta(\mathrm{N}^2)$ in terms of time complexity.

With **Fast Fourier Transforms (FFT)** we save some steps by reducing the number of calculations in DFT. We will see later that, as a result of this, we can reduce a domain of size N from $\Theta(N^2)$ to $\Theta(N \log 2N)$.

As the name implies, it is a fast or faster way to compute Fourier Transforms. We will see later that this becomes more noticeable when the size of the domain becomes larger.

How do we do that?

Suppose, we separated the Fourier Transform into even and odd indexed sub-sequences.

$$\begin{cases} n = 2r & if \quad even \\ n = 2r + 1 & if \quad odd \end{cases}$$

where $r = 1, 2, ..., \frac{N}{2} - 1$.

After performing a bit of algebra, we end up with the summation of two terms. The advantage of this approach lies in the fact that the even and odd indexed sub-sequences can be computed concurrently.

$$x[k] = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}$$

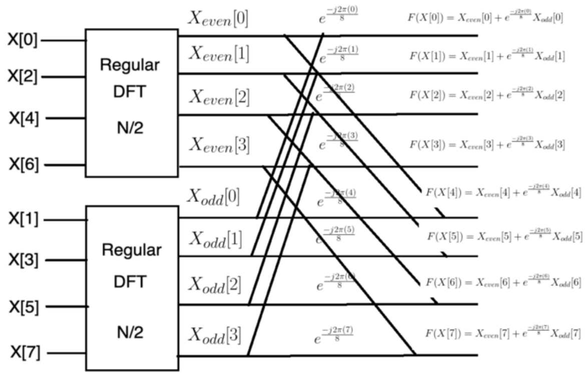
$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]e^{\frac{-j2k\pi(2r)}{N}} + x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]e^{\frac{-j2k\pi(2r+1)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]e^{\frac{-j2k\pi(2r)}{N}} + x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]e^{\frac{-j2k\pi(2r)}{N}}$$

$$x[k] = \sum_{r=0}^{\frac{N}{2}-1} x[2r]e^{\frac{-j2k\pi r}{N/2}} + x[k] = e^{\frac{-j2\pi k}{N}} \sum_{r=0}^{\frac{N}{2}-1} x[2r+1]e^{\frac{-j2k\pi r}{N/2}}$$

$$x[k] = x_{even}[k] + e^{\frac{-j2\pi k}{N}} x_{odd}[k]$$

Suppose, $N = 8$, to visualize the flow of data with time, we can make use of a butterfly diagram. We compute the Discrete Fourier Transform for the even and odd terms simultaneously. Then, we calculate $x[k]$ using the formula from above.


X[0]
X[2]
X[4]
X[6]
X[1]
X[3]
X[5]
X[7]
Regular DFT N/2
Regular DFT N/2
$X_{even}[0]$
$X_{even}[1]$
$X_{even}[2]$
$X_{even}[3]$
$X_{odd}[0]$
$X_{odd}[1]$
$X_{odd}[2]$
$X_{odd}[3]$
$F(X[0]) = X_{even}[0] + e^{\frac{-j2\pi(0)}{8}} X_{odd}[0]$
$F(X[1]) = X_{even}[1] + e^{\frac{-j2\pi(1)}{8}} X_{odd}[1]$
$F(X[2]) = X_{even}[2] + e^{\frac{-j2\pi(2)}{8}} X_{odd}[2]$
$F(X[3]) = X_{even}[3] + e^{\frac{-j2\pi(3)}{8}} X_{odd}[3]$
$F(X[4]) = X_{even}[4] + e^{\frac{-j2\pi(4)}{8}} X_{odd}[4]$
$F(X[5]) = X_{even}[5] + e^{\frac{-j2\pi(5)}{8}} X_{odd}[5]$
$F(X[6]) = X_{even}[6] + e^{\frac{-j2\pi(6)}{8}} X_{odd}[6]$
$F(X[7]) = X_{even}[7] + e^{\frac{-j2\pi(7)}{8}} X_{odd}[7]$

We can express the gains in terms of Big O Notation as follows. The first term comes from the fact that we compute the Discrete Fourier Transform twice. We multiply the latter by the time taken to compute the Discrete Fourier Transform on half the original input. In the final step, it takes N steps to add up the Fourier Transform for a particular k. We account for this by adding N to the final product.

$$\underbrace{2}_{\text{2 DFT}} \times \overbrace{\left(\frac{N}{2}\right)^2}^{\text{DFT on N/2 elements}} + \underbrace{N}_{x[k] = x_{even}[k] + e^{\frac{-j2\pi k}{N}} x_{odd}[k]}$$

$$= 2 \times \frac{N^2}{4} + N$$

$$= \frac{N^2}{2} + N$$

$$O\left(\frac{N^2}{2} + N\right) \sim O(N^2)$$

Notice how we were able to cut the time taken to compute the Fourier Transform by a factor of 2. We can further improve the algorithm by applying the divide-and-conquer approach, halving the computational cost each time. In other words, we can continue to split the problem size until we're left with groups of two and then directly compute the Discrete Fourier Transforms for each of those pairs.

What if we keep splitting?

$$\frac{N}{2} \longrightarrow 2\left(\frac{N}{2}\right)^2 + N = \frac{N^2}{2} + N$$

$$\frac{N}{4} \longrightarrow 2\left(2\left(\frac{N}{4}\right)^2 + \frac{N}{2}\right) + N = \frac{N^2}{4} + 2N$$

$$\frac{N}{8} \longrightarrow 2\left(2\left(2\left(\frac{N}{8}\right)^2 + \frac{N}{4}\right) + \frac{N}{2}\right) + N = \frac{N^2}{8} + 3N$$

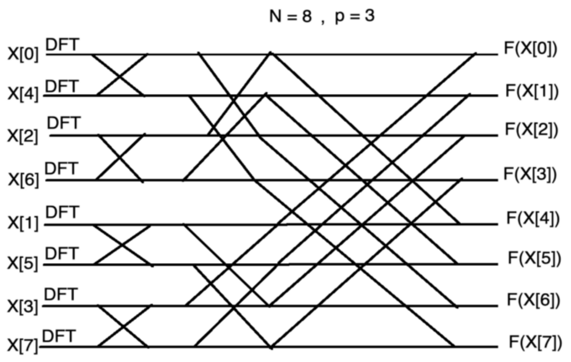
$$\vdots$$

$$\frac{N}{2^P} \longrightarrow \frac{N^2}{2^P} + pN = \frac{N^2}{N} + (\log_2 N)N = N + (\log_2 N)N$$

$$\sim O(N + Nlog_2N) \sim O(Nlog_2N)$$

So as long as N is a power of 2, the maximum number of times you can split into two equal halves is given by $p = log(N)$.

Here's what it would look like if we were to use the Fast Fourier Transform algorithm with a problem size of N = 8. Notice how we have p = log(8) = 3 stages.



What is FFT used for?

FFT are used in digital signal processing and training models used in Convolutional Neural Networks. Also, it has many applications in component analysis.

One application of FFT is in digital signal processing. It can be used as a tool for decomposing signals from the time domain to the frequency domain. This is a good way of separating signals like when quantizing analog waves to discrete signals.

Another application of FFT is in image processing. In convolutional layer overlays a kernel can scan a section of an image. This takes all values from that location to perform operations. The kernel then shifts to another section of the image and repeats the process until it scans the entire image.

2. **Description of implementation**

   (a) **Implementation of the Algorithm**

   Let start with a simple example. For this case I will use a **Jupyter Nootbook**, but we can also use a Python enviroment at the prompt or a py file.

   At first we import the numpy library and matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
```

   Lets implements DFT directly.

```
def dft(x):
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
    n = np.arange(N)
    k = n.reshape((N, 1))
    M = np.exp(-2j * np.pi * k * n / N)
    return np.dot(M, x)
```

   Like we saw before, the Fast Fourier Transform works by computing the Discrete Fourier Transform for small subsets of the overall problem and then combining the results. The latter can easily be done in code using recursion:

```
plt.style.use('seaborn-poster')
%matplotlib inlinedef FFT(x):

    N = len(x)

    if N == 1:
        return x
    else:
        X_even = FFT(x[::2])
        X_odd = FFT(x[1::2])
        factor = \
          np.exp(-2j*np.pi*np.arange(N)/ N)

        X = np.concatenate(\
            [X_even+factor[:int(N/2)]*X_odd,
             X_even+factor[int(N/2):]*X_odd])
        return X
```

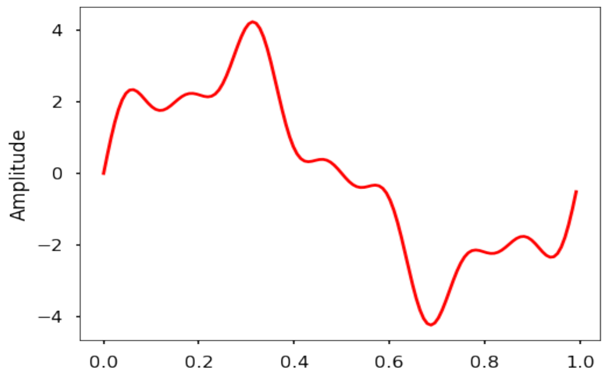
   We will see later that this implementation works and is faster than DFT, but it will be better to use vector operations instead of recursion, so we can proceed as follows:

```
def FFT_V(x):
    x = np.asarray(x, dtype=float)
    N = x.shape[0]
    if np.log2(N) % 1 > 0:
        raise ValueError("must be a power of 2")

    N_min = min(N, 2)

    n = np.arange(N_min)
    k = n[:, None]
    M = np.exp(-2j * np.pi * n * k / N_min)
    X = np.dot(M, x.reshape((N_min, -1)))

    while X.shape[0] < N:
        X_even = X[:, :int(X.shape[1] / 2)]
        X_odd = X[:, int(X.shape[1] / 2):]
        terms = np.exp(-1j * np.pi * np.arange(X.shape[0])
                                / X.shape[0])[:, None]
                X = np.vstack([X_even + terms * X_odd,
                               X_even - terms * X_odd])

            return X.ravel()
```

Now, we can generate the signal:

```
        # sampling rate
        sr = 128
        # sampling interval
        ts = 1.0/sr
        t = np.arange(0,1,ts)

        freq = 1.
        x = 3*np.sin(2*np.pi*freq*t)

        freq = 4
        x += np.sin(2*np.pi*freq*t)

        freq = 7
        x += 0.5* np.sin(2*np.pi*freq*t)

        plt.figure(figsize = (8, 6))
        plt.plot(t, x, 'r')
        plt.ylabel('Amplitude')

        plt.show()
```

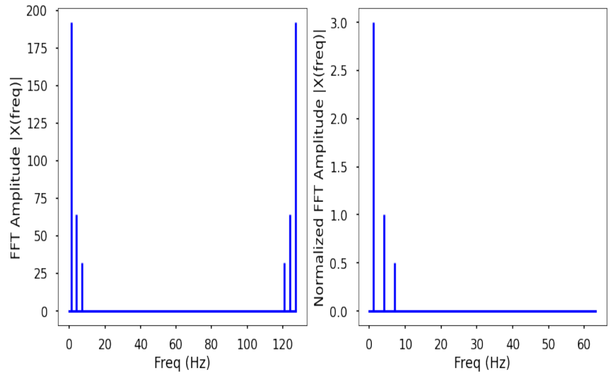
This code produces the graph plot:



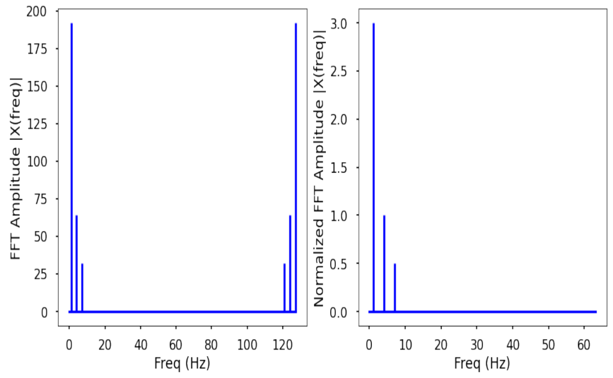
So, finally, we can calculate the Fourier Transform of the signal from the graph:

```
        X=FFT(x)

        # calculate the frequency
        N = len(X)
        n = np.arange(N)
        T = N/sr
        freq = n/T

        plt.figure(figsize = (12, 6))
        plt.subplot(121)
        plt.stem(freq, abs(X), 'b', \
                 markerfmt=" ", basefmt="-b")
        plt.xlabel('Freq (Hz)')
        plt.ylabel('FFT Amplitude |X(freq)|')

        # Get the one-sided specturm
```

```
    n_oneside = N//2
    # get the one side frequency
    f_oneside = freq[:n_oneside]

    # normalize the amplitude
    X_oneside =X[:n_oneside]/n_oneside

    plt.subplot(122)
    plt.stem(f_oneside, abs(X_oneside), 'b', \
             markerfmt=" ", basefmt="-b")
    plt.xlabel('Freq (Hz)')
    plt.ylabel('Normalized FFT Amplitude |X(freq)|')
    plt.tight_layout()
    plt.show()
```



If we do the same but using the function dft instead of FFT we get:



It looks very similar. What's more if we use FFT-V we will get a very similar result.

(b) **Implementation of the unit test**

For doing the unit test we are going to take into account that this algorithm is already implemented in the numpy library, so we can do something like this:

```
    from numpy.fft import fft, ifft
    x = np.random.random(1024)
    X1 = dft(x)
    X2 = fft(x)
    print(np.array_equal(X1,X2))
```

But we will rarely get the arrays to be exactly the same, so it is better to compare them with a certain tolerance, so we can change to this:

```
    from numpy.fft import fft, ifft
    x = np.random.random(1024)
```

```
        X1 = dft(x)
        X2 = fft(x)
        np.allclose(X1,X2,rtol=1e-05,atol=1e-08,equal_nan=False)
```

The code will print true if the result of the algorithm is correct and false otherwise. We can also include in the comparison with all the implementations done before.
Bearing this in mind, we can propose the following unit test (*We could also randomly generate the size of x if we want*):

```
    i = 0
    while i<10:
        x = np.random.random(1024)
        X1 = dft(x)
        X2 = fft(x)      # Numpy implementation
        X3 = FFT(x)
        X4 = FFT_V(x)
        print("Case", i+1, ":")
        print("\t", np.allclose(X1,X2,rtol=1e-05,atol=1e-08,equal_nan=False))
        print("\t", np.allclose(X2,X3,rtol=1e-05,atol=1e-08,equal_nan=False))
        print("\t", np.allclose(X2,X4,rtol=1e-05,atol=1e-08,equal_nan=False))
        i+=1
```

3. **Description of unit tests and computational complexity**

   (a) **Unit tests** We have just seen the idea of the test at (3c)

   (b) **Parameter N of the algorithm**

   We now that the Fourier Transform of an analogue signal $x(t)$ is given by:

$$X(f) = \int_{-\infty}^{+\infty} x(t)e^{-j2\pi ft}\, dt$$

   Further more, the Discrete Fourier Transform (DFT) of a discrete time signal is given by:

$$X(k) = \sum_{n=0}^{N-1} x[n] \cdot e^{-j\frac{2\pi}{N}kn}$$

   Lets call $e^{-j\frac{2\pi}{N}kn} = W_N$, then we will have:

$$X(0) = x[0]W_N^0 + x[1]W_N^{0\cdot 1} + ... + W[N-1]_N^{0\cdot(N-1)}$$
$$X(1) = x[0]W_N^0 + x[1]W_N^{1\cdot 1} + ... + W[N-1]_N^{1\cdot(N-1)}$$
$$...$$
$$X(N-1) = x[0]W_N^0 + x[1]W_N^{(N-1)\cdot 1} + ... + W[N-1]_N^{(N-1)\cdot(N-1)}$$

   To determine the DTF of a discrete signal $x[n]$ (where N is the size of its domain), we multiply each of its value by e raised to some function of n. We then sum the results obtained for a given n. If we used a computer to calculate the Discrete Fourier Transform of a signal, it would need to perform N (multiplications) x N (additions) = $\Theta(N^2)$ operations.

   We have seen at (1) how we can use FFT for reducing the number of computations needed for a problem of size N from $\Theta(N^2)$ to $\Theta(N \log N)$. On the surface, this might not seem like a big deal. However, when N is large enough it can make a world of difference. Have a look at the following table.

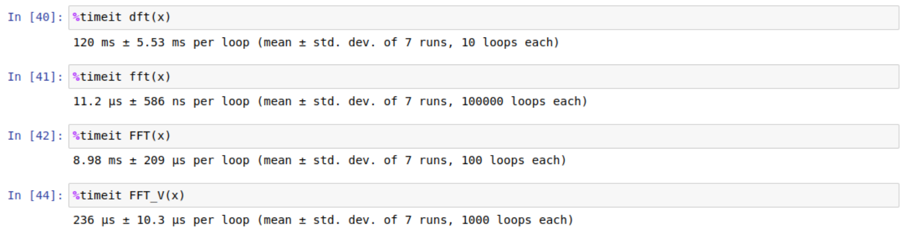
N	1000	10^6	10^9
N^2	10^6	10^{12}	10^{18}
$Nlog_2N$	10^4	$20\,x\,10^6$	$30\,x\,10^9$

Say it took 1 nanosecond to perform one operation. It would take the Fast Fourier Transform algorithm approximately 30 seconds to compute the Discrete Fourier Transform for a problem of size $N = 10^9$. In contrast, the regular algorithm would need several decades.

$$10^{18} ns \rightarrow 31.2\, years$$

$$30\,x\,10^9 ns \rightarrow 30\, seconds$$

Going back to our code, we can see next results ($using\ N = 1024$):



```
In [40]: %timeit dft(x)
         120 ms ± 5.53 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)

In [41]: %timeit fft(x)
         11.2 µs ± 586 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

In [42]: %timeit FFT(x)
         8.98 ms ± 209 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)

In [44]: %timeit FFT_V(x)
         236 µs ± 10.3 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

The FFT algorithm is significantly faster than the direct implementation. However, it still lags behind the numpy implementation by quite a bit. One reason for this is the fact that the numpy implementation uses matrix operations to calculate the Fourier Transforms simultaneously.

Furthermore, we can see, the FFT implementation using vector operations is significantly faster than what we had obtained previously. We still haven't come close to the speed at which the numpy library computes the Fourier Transform. This is because the FFTPACK algorithm behind numpy's fft is a Fortran implementation which has received years of tweaks and optimizations. If you are interested in finding out more, I recommend you have a look at the source code.

(c) **Some examples**

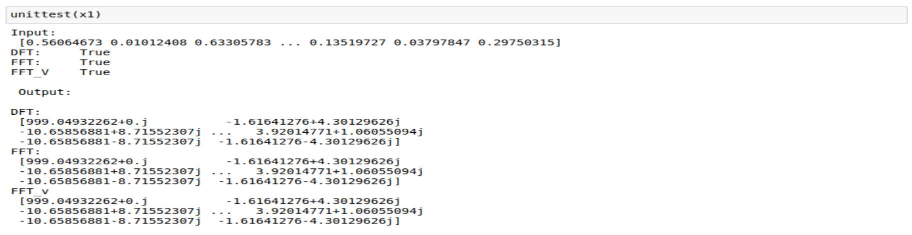
Lets prepare some inputs data. We will generate them randomly:

```
    x1 = np.random.random(2048)          # 2^11
    x2 = np.random.random(1024)          # 2^10
    x3 = np.random.random(8192)          # 2^13
```

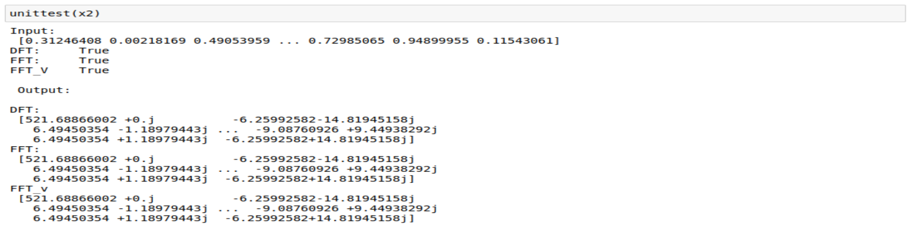
We can prepare the next script for showing the results:

```
def unittest(x):
    X1 = dft(x)
    X2 = fft(x)
    X3 = FFT(x)
    X4 = FFT_V(x)
    print("Input: \n", x)
    print("DFT: \t", np.allclose(X1,X2,rtol=1e-05,atol=1e-08,equal_nan=False))
    print("FFT: \t", np.allclose(X2,X3,rtol=1e-05,atol=1e-08,equal_nan=False))
    print("FFT_V \t", np.allclose(X2,X4,rtol=1e-05,atol=1e-08,equal_nan=False))
    print("\n Output: \n")
    print("DFT: \n", X1)
    print("FFT: \n", X3)
    print("FFT_v \n", X4)
```

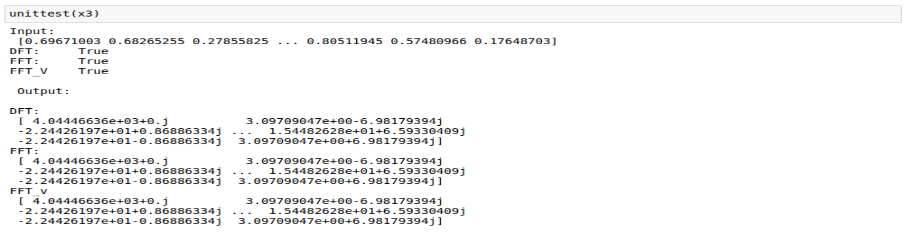
We obtain the next results:



```
unittest(x1)

Input:
 [0.56064673 0.01012408 0.63305783 ... 0.13519727 0.03797847 0.29750315]
DFT:      True
FFT:      True
FFT_V     True

 Output:

DFT:
 [999.04932262+0.j           -1.61641276+4.30129626j
 -10.65856881+8.71552307j ...   3.92014771+1.06055094j
 -10.65856881-8.71552307j  -1.61641276-4.30129626j]
FFT:
 [999.04932262+0.j           -1.61641276+4.30129626j
 -10.65856881+8.71552307j ...   3.92014771+1.06055094j
 -10.65856881-8.71552307j  -1.61641276-4.30129626j]
FFT_v
 [999.04932262+0.j           -1.61641276+4.30129626j
 -10.65856881+8.71552307j ...   3.92014771+1.06055094j
 -10.65856881-8.71552307j  -1.61641276-4.30129626j]
```



```
unittest(x2)

Input:
 [0.31246408 0.00218169 0.49053959 ... 0.72985065 0.94899955 0.11543061]
DFT:      True
FFT:      True
FFT_V     True

 Output:

DFT:
 [521.68866002 +0.j            -6.25992582-14.81945158j
    6.49450354 -1.18979443j ...   -9.08760926 +9.44938292j
    6.49450354 +1.18979443j  -6.25992582+14.81945158j]
FFT:
 [521.68866002 +0.j            -6.25992582-14.81945158j
    6.49450354 -1.18979443j ...   -9.08760926 +9.44938292j
    6.49450354 +1.18979443j  -6.25992582+14.81945158j]
FFT_v
 [521.68866002 +0.j            -6.25992582-14.81945158j
    6.49450354 -1.18979443j ...   -9.08760926 +9.44938292j
    6.49450354 +1.18979443j  -6.25992582+14.81945158j]
```

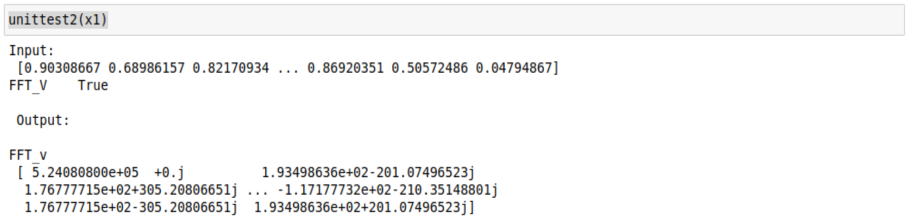


```
unittest(x3)

Input:
 [0.69671003 0.68265255 0.27855825 ... 0.80511945 0.57480966 0.17648703]
DFT:      True
FFT:      True
FFT_V     True

 Output:

DFT:
 [ 4.04446636e+03+0.j           3.09709047e+00-6.98179394j
 -2.24426197e+01+0.86886334j ...   1.54482628e+01+6.59330409j
 -2.24426197e+01-0.86886334j  3.09709047e+00+6.98179394j]
FFT:
 [ 4.04446636e+03+0.j           3.09709047e+00-6.98179394j
 -2.24426197e+01+0.86886334j ...   1.54482628e+01+6.59330409j
 -2.24426197e+01-0.86886334j  3.09709047e+00+6.98179394j]
FFT_v
 [ 4.04446636e+03+0.j           3.09709047e+00-6.98179394j
 -2.24426197e+01+0.86886334j ...   1.54482628e+01+6.59330409j
 -2.24426197e+01-0.86886334j  3.09709047e+00+6.98179394j]
```

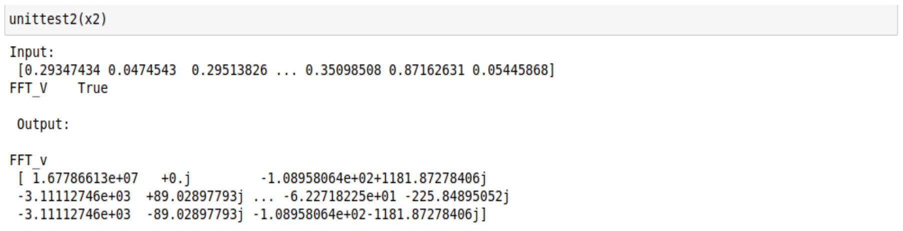
If we want to use a higher value of N we should consider deleting dft and the recursive implementation from the script, because it computational time is too elevated, so lets see what happens using now using a higher value of N:

```
    x1 = np.random.random(1048576)          # 2^20
    x2 = np.random.random(33554432)         # 2^25

    def unittest2(x):
    X2 = fft(x)
    X4 = FFT_V(x)
    print("Input: \n", x)
    print("FFT_V \t", np.allclose(X2,X4,rtol=1e-05,atol=1e-08,equal_nan=False))
    print("\n Output: \n")
    print("FFT_v \n", X4)
```

And we get:



```
unittest2(x1)

Input:
 [0.90308667 0.68986157 0.82170934 ... 0.86920351 0.50572486 0.04794867]
FFT_V    True

 Output:

FFT_v
 [ 5.24080800e+05  +0.j          1.93498636e+02-201.07496523j
  1.76777715e+02+305.20806651j ... -1.17177732e+02-210.35148801j
  1.76777715e+02-305.20806651j  1.93498636e+02+201.07496523j]
```



```
unittest2(x2)

Input:
 [0.29347434 0.0474543  0.29513826 ... 0.35098508 0.87162631 0.05445868]
FFT_V    True

 Output:

FFT_v
 [ 1.67786613e+07   +0.j          -1.08958064e+02+1181.87278406j
 -3.11112746e+03  +89.02897793j ... -6.22718225e+01 -225.84895052j
 -3.11112746e+03  -89.02897793j -1.08958064e+02-1181.87278406j]
```

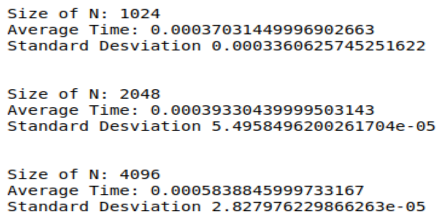
4. **Results**

   From now on we are going to work only with the most optimal algorithm, which is the vector implementation. For this part we are going to take 15 different values of N and we are going to run the algorithm 10 times for each one. Then, we are going to calculate the average time and the standard deviation of those data.

   Lets prepare a script for this:

```python
from timeit import default_timer as timer

N = 1024        # 2^10
i = 0
while i != 15:
    x = np.random.random(N)
    j = 0
    times = []

    while j != 10:
        start = timer()
        X = FFT_V(x)
        end = timer()
        time = end-start
        times.append(time)
        j+=1

    average = np.mean(times)
    desviation = np.std(times)

    print("Size of N:", N)
    print("Average Time:", average)
    print("Standard Desviation", desviation)
    print("\n")
    i+=1
    N*=2
```

   We obtain the next output (Is not all the output):



```
Size of N: 1024
Average Time: 0.00037031449996902663
Standard Desviation 0.0003360625745251622


Size of N: 2048
Average Time: 0.00039330439999503143
Standard Desviation 5.4958496200261704e-05


Size of N: 4096
Average Time: 0.0005838845999733167
Standard Desviation 2.827976229866263e-05
```

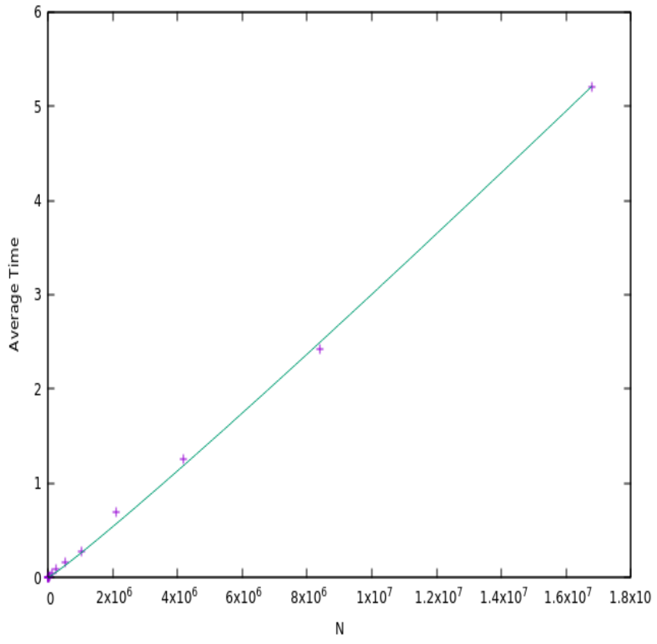
   If we wanted to verify each run we could use the test described in the previous section.

   Lets make now a table with all the data:

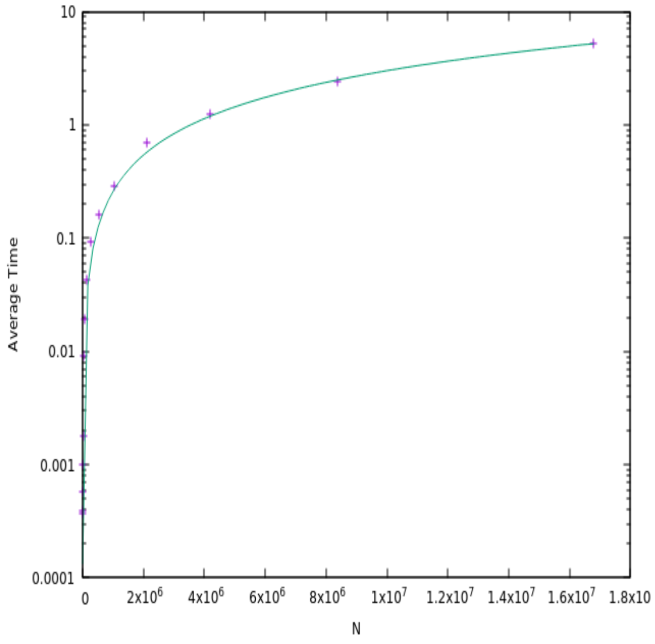
No	N	Average Time	Standard Deviation
1	1024	0.00037031449996902663	0.0003360625745251622
2	2048	0.00039330439999503143	5.4958496200261704e-05
3	4096	0.0005838845999733167	2.827976229866263e-05
4	8192	0.000996049400055199	1.9379925557523726e-05
5	16384	0.0017965203000585462	6.381512801420919e-05
6	32768	0.009251793700013878	0.00010934509003745456
7	65536	0.01931392230003439	0.0006604803157864763
8	131072	0.04263230810006462	0.0019725422730663194
9	262144	0.09390219590000015	0.005217572236558978
10	524288	0.16233227770007944	0.005390355898752742
11	1048576	0.2863475087000552	0.012867395923297893
12	2097152	0.703974573799951	0.026872773602171898
13	4194304	1.2565692351999587	0.05752892342869337
14	8388608	2.43117686150008	0.06658766287357518
15	16777216	5.200203602299962	0.23986044154983005

Fro generating the graphics I will use **gnuplot**. I have a "data.dat" document with the column N, Average Time and Standard Deviation, so we proceed as follows:

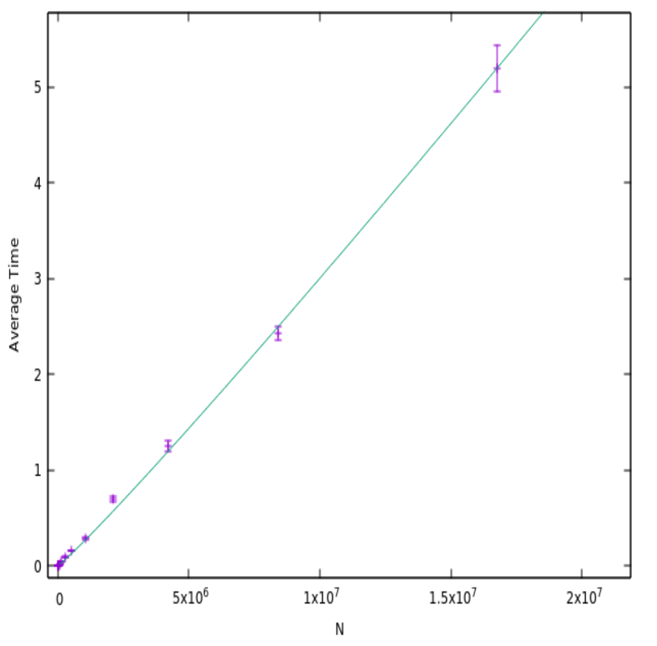
```
    gnuplot> set xlabel "N"
    gnuplot> set ylabel "Average Time"
    gnuplot> unset key
    gnuplot> f(x) = a*x*log(x)
    gnuplot> fit f(x) "data.dat" u 1:2 via a
    iter      chisq       delta/lim  lambda   a
  0 2.9328830347e-02   0.00e+00  1.54e+00    1.870875e-08
  1 2.8849467245e-02  -1.66e+03  1.54e-01    1.864406e-08
  2 2.8847587390e-02  -6.52e+00  1.54e-02    1.863975e-08
  3 2.8847587389e-02  -2.89e-06  1.54e-03    1.863975e-08
    iter      chisq       delta/lim  lambda   a

    After 3 iterations the fit converged.
    final sum of squares of residuals : 0.0288476
    rel. change during last iteration : -2.89255e-11

    degrees of freedom    (FIT_NDF)                        : 14
    rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0453932
    variance of residuals (reduced chisquare) = WSSR/ndf   : 0.00206054

    Final set of parameters            Asymptotic Standard Error
    =======================            ==========================
    a               = 1.86398e-08      +/- 1.428e-10    (0.766%)

    gnuplot> plot "data.dat" u 1:2, f(x)
    gnuplot> set log y
    gnuplot> replot
    gnuplot> plot "data.dat" with yerrorbars, f(x)
    gnuplot> unset log y
    gnuplot> replot
```



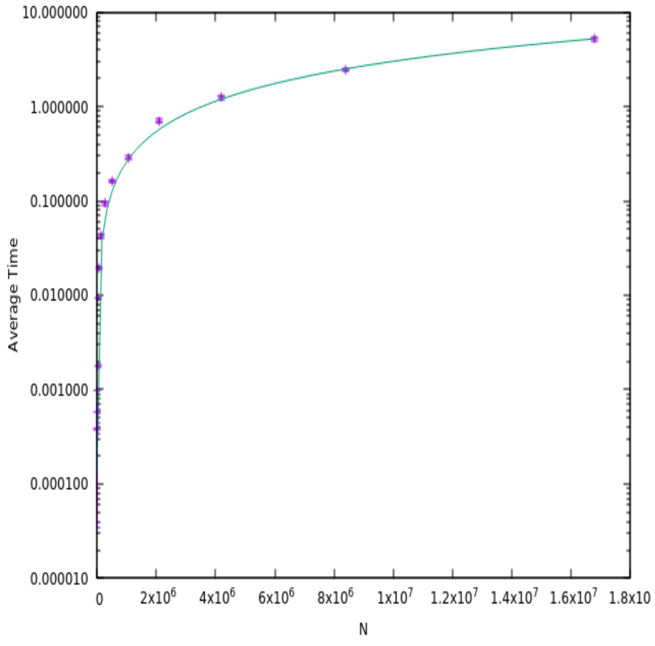
(a) Results Plot



(b) Log Scale Y



(a) Results Plot With Error



(b) Log Scale Y With Error

So, as we can see, the points fit fairly well to the complexity curve calculated above (*By using the **gnuplot** tool the adjustment is quite fast and easy to calculate*). The only drawback of the previous adjustment is that we have very few data to perform the adjustment and these are far apart as N grows as a consequence of the fact that our algorithm requires that the size of the data be a power of two.

5. **Conclusion**

   We have seen throughout this work the importance of optimizing algorithms. We have implemented a DFT and we have seen how it was a complexity $\Theta(N^2)$ algorithm, so when we grew the value of N the algorithm was unusable due to its high execution time. On the other hand, we have done a mathematical development to reduce the complexity of the algorithm to $\Theta(Nlog(N))$, called FFT; in addition to how the execution time varies when using different structures, such as recursion and vectors in our case (*Remember that with recursive implementation the execution time was much higher than using vectors. Also the implementation of numpy was the best because of using FORTRAN.*

   Finally, throughout the work we have seen different tools that have helped us in our goal, such as LaTex, Gnuplot or JupyterNotebooks, which are very useful and used in the scientific and research world.

References

[1] Python Programming and Numerical Methods - A Guide for Engineers and Scientists

[2] FFT Wikipedia

[3] Transformada rápida de Fourier (I)

[4] Transformación rápida de Fourier FFT - Conceptos básicos

[5] TRANSFORMADA DE FOURIER - Parte 1: Interpretación Frecuencial para Ingeniería — El Traductor

[6] ¿Qué es la Transformada de Fourier? Una introducción visual