# COMMUNICATION NETWORKS ASSIGNMENT

**DATE - 10.04.2022**

## TOPIC- Inverse Fast Fourier Transform (IFFT)

**PARTH AGGARWAL**
**ID: 2019AAPS0218H**
**EMAIL: f20190218@hyderabad.bits-pilani.ac.in**

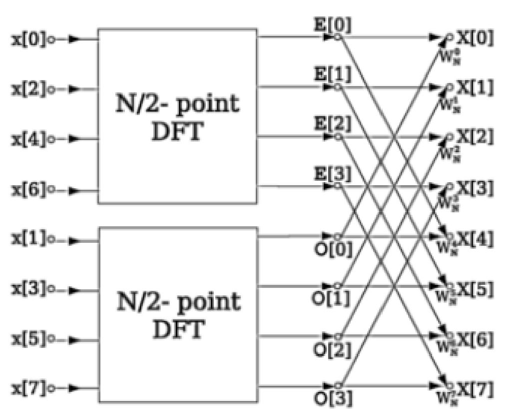
**HARSHIT VARIKOLU**
**ID: 2019AAPS0338H**
**EMAIL: f20190338@hyderabad.bits-pilani.ac.in**

**ANSHUMAN MRIDUL**
**ID: 2019AAPS0330H**
**EMAIL: f20190330@hyderabad.bits-pilani.ac.in**

## Introduction

A fast Fourier transform (FFT) is an algorithm that computes the discrete Fourier transform (DFT) of a sequence, or its inverse (IDFT). DFT is used to convert a signal from time domain to frequency domain while vice versa is true for IDFT.

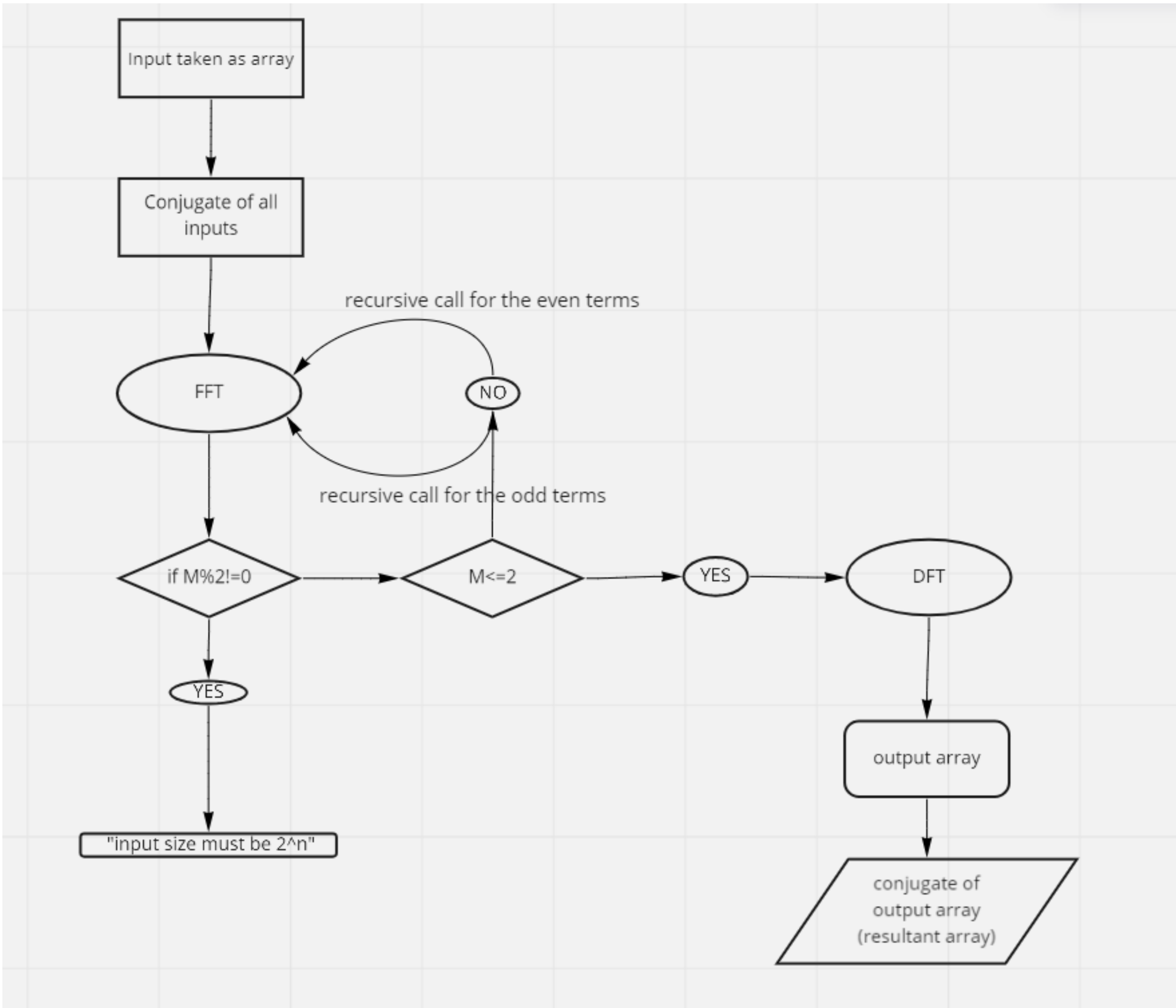
However DFT has proven to be very slow and expensive for a larger number of inputs. Therefore FFT was developed to reduce the time and computing costs for Fourier transform.An FFT rapidly computes all the transformations involved in DFT  by dividing the DFT matrix into a product of sparse (mostly zero) factors. As a result, it manages to reduce the complexity of computing the DFT from O(N^2) to O(N*logN) in FFT.




This is how the FFT algorithm divides the inputs into 2 parts and performs DFT. This process halves the computation time.

In the Python simulation we will notice that this division of inputs goes on until the data size becomes 2 and then DFT is performed. For Inverse FFT we will input the conjugate of the input list into the FFT  algorithm and again take the conjugate of the output.

## Block design




## Inputs and simulations

Inputs for IFFT for both MATLAB and Python code are

[1+1j ,2+2j, 3+3j, 4+4j, 5+5j, 6+6j, 7+7j ,8+8j]

- <u>Python simulation</u>

```
Enter number of elements (must be a power of 2) : 8
1+1j
2+2j
3+3j
4+4j
5+5j
6+6j
7+7j
8+8j
['1+1j', '2+2j', '3+3j', '4+4j', '5+5j', '6+6j', '7+7j', '8+8j']
[ 4.50000000e+00+4.50000000e+00j  7.07106781e-01-1.70710678e+00j
 -4.99600361e-16-1.00000000e+00j -2.92893219e-01-7.07106781e-01j
 -5.00000000e-01-5.00000000e-01j -7.07106781e-01-2.92893219e-01j
 -1.00000000e+00+5.55111512e-16j -1.70710678e+00+7.07106781e-01j]
```

- <u>MATLAB Verification</u>

Editor - C:\Users\91834\Documents\MATLAB\cn.m

cn.m

```
clc;
close all;
clear all;
x=[1+1j 2+2j 3+3j 4+4j 5+5j 6+6j 7+7j 8+8j];

y=ifft(x)

fft(y)
```

Command Window

```
y =

  Columns 1 through 3

   4.5000 + 4.5000i   0.7071 - 1.7071i   0.0000 - 1.0000i

  Columns 4 through 6

  -0.2929 - 0.7071i  -0.5000 - 0.5000i  -0.7071 - 0.2929i

  Columns 7 through 8

  -1.0000 + 0.0000i  -1.7071 + 0.7071i
```

**Assumptions-**

- In the python simulation there are some terms with powers of 10^-16. We assume those terms to be zero
- We are taking number of inputs as power of 2.

**Conclusion**

- Keeping in mind the above assumption we can clearly see that the output of the python simulation is verified by the ‘ifft’ function of MATLAB.

- Therefore, we can conclude that FFT is a very efficient and fast way of calculating DFT as it reduces complexity by factorizing the DFT matrix and bifurcating the even input terms and odd input terms.

**References**

https://en.wikipedia.org/wiki/Fast_Fourier_transform

## Appendix

- Python code-

```
import numpy as np
import math as mt

def DFT_1D(fx):
    fx = np.asarray(fx, dtype=complex)      #storing as an array
    M = fx.shape[0]                          #calculating size
    fu = fx.copy()

    for i in range(M):
        u = i
        sum = 0
        for j in range(M):
            x = j
            tmp = fx[x]*np.exp(-2j*np.pi*x*u*np.divide(1, M,
dtype=complex))  #multiplying with e^(i*2pie*n*k/N)
            sum += tmp
#summation

        fu[u] = sum

    return fu

def FFT_1D(fx):
    fx = np.asarray(fx, dtype=complex)
    M = fx.shape[0]                                       #size of input
    minDivideSize = 2

    if M % 2 != 0:
        raise ValueError("the input size must be 2^n")   #if input size
not power of 2

    if M <= minDivideSize:
        return DFT_1D(fx)                                  #for input of
size <=2
    else:
```

```
        fx_even = FFT_1D(fx[::2])                              # recursive
call for  the even part
        fx_odd = FFT_1D(fx[1::2])                              # recursive
call for  the odd part
        W_ux_2k = np.exp(-2j * np.pi * np.arange(M) / M)
#e^(i*2pie*n*k/N)

        f_u = fx_even + fx_odd * W_ux_2k[:M//2]              #multiplying
with e^(i*2pie*n*k/N)

        f_u_plus_k = fx_even + fx_odd * W_ux_2k[M//2:]

        fu = np.concatenate([f_u, f_u_plus_k])               # summing up
the even and odd parts

    return fu

def inverseFFT_1D(fu):
    fu = np.asarray(fu, dtype=complex) #storing input in an array
    fu_conjugate = np.conjugate(fu)    #taking conjugate of inputs

    fx = FFT_1D(fu_conjugate)

    fx = np.conjugate(fx)              #again taking conjugate to get our
answers in sequence correspond to input sequence
    fx = fx / fu.shape[0]

    return fx





"Input and Initialization"
n = int(input("Enter number of elements (must be a power of 2) : "))

lst = [ ]
# iterating till the range
for i in range(0, n):
    ele = input()
```

```
    lst.append(ele) # adding the element

print(lst)
y=inverseFFT_1D(lst)
print(y)
```

Output

```
Enter number of elements (must be a power of 2) : 8
1+1j
2+2j
3+3j
4+4j
5+5j
6+6j
7+7j
8+8j
['1+1j', '2+2j', '3+3j', '4+4j', '5+5j', '6+6j', '7+7j', '8+8j']
[ 4.50000000e+00+4.50000000e+00j  7.07106781e-01-1.70710678e+00j
 -4.99600361e-16-1.00000000e+00j -2.92893219e-01-7.07106781e-01j
 -5.00000000e-01-5.00000000e-01j -7.07106781e-01-2.92893219e-01j
 -1.00000000e+00+5.55111512e-16j -1.70710678e+00+7.07106781e-01j]
```

**MATLAB verification**

```
clc;
close all;
clear all;
x=[1+1j 2+2j 3+3j 4+4j 5+5j 6+6j 7+7j 8+8j];

y=ifft(x)
```

**Result**

```
y =

  Columns 1 through 3
   4.5000 + 4.5000i   0.7071 - 1.7071i   0.0000 - 1.0000i

  Columns 4 through 6
  -0.2929 - 0.7071i  -0.5000 - 0.5000i  -0.7071 - 0.2929i

  Columns 7 through 8

  -1.0000 + 0.0000i  -1.7071 + 0.7071
```