EE3025 Assignment-1

K. Yasawanth Naidu - EE18BTECH11024

Download all python codes from

https://github.com/yaswanthNaidu99/ASSIGNMENT1/tree/main/codes

and latex-tikz codes from

https://github.com/yaswanthNaidu99/ASSIGNMENT1/blob/main/EE18BTECH11024.tex

1 Problem

1.1. Let

$$x(n) = \left\{\underset{\uparrow}{1}, 2, 3, 4, 2, 1\right\} \tag{1.1.1}$$

$$y(n) + \frac{1}{2}y(n-1) = x(n) + x(n-2) \tag{1.1.2}$$

1.2. Compute

$$X(k) \triangleq \sum_{n=0}^{N-1} x(n)e^{-j2\pi kn/N}, \quad k = 0, 1, \dots, N-1 \tag{1.2.1}$$

and $H(k)$ using h(n).

1.3. Compute $X(k)$, $H(k)$ and $y(n)$ using FFT and IFFT methods.

2 Solution

2.1. Computing h(n) using Z-transform of y(n) as follows :

$$Y(z) + \frac{1}{2}z^{-1}Y(z) = X(z) + z^{-2}X(z) \tag{2.1.1}$$

$$\implies Y(z) = \frac{2(z^2+1)}{z(2z+1)}X(z) \tag{2.1.2}$$

$$H(z) = \frac{Y(z)}{X(z)} = \frac{1+z^{-2}}{1+\frac{1}{2}z^{-1}} \tag{2.1.3}$$

applying inverse Z-transform to compute $h(n)$

$$h(n) = Z^{-1}\left(\frac{1}{1+\frac{1}{2}z^{-1}} + \frac{z^{-2}}{1+\frac{1}{2}z^{-1}}\right) \tag{2.1.4}$$

$$h(n) = \frac{-1}{2}^{n} u(n) + \left(\frac{-1}{2}\right)^{n-2} u(n-2) \tag{2.1.5}$$

2.2. Computing y (for N samples) using FFT and IFFT :

$$X = FFT(x) \tag{2.2.1}$$

$$H = FFT(h) \tag{2.2.2}$$

$$Y = X.H \tag{2.2.3}$$

$$y = IFFT(Y) \tag{2.2.4}$$

2.3. If desired output is real :

$$y = IFFT(Y) = \frac{1}{N} * FFT(Y^*) \tag{2.3.1}$$

where Y^* = complex conjugate(Y)
Thus IFFT can be implemented using the FFT function itself, which can save memory in a hardware setup.

2.4. Implementation and results :
y is computed through above steps (by padding x so that N = 8), while a recursive FFT algorithm is implemented to compute the 8 point FFT.
The code for it is as follows :

https://github.com/yaswanthNaidu99/ASSIGNMENT1/blob/main/codes/fft.py

$$\bar{x} = \begin{bmatrix}1\\2\\3\\4\\2\\1\\0\\0\end{bmatrix} \qquad \bar{h} = \begin{bmatrix}1.\\-0.5\\1.25\\-0.625\\0.3125\\-0.15625\\0.0625\\-0.03125\end{bmatrix} \tag{2.4.1}$$

$$\overline{X} = \begin{bmatrix} 13 \\ -3.121 - 6.536j \\ 1.j \\ 1.121 - 0.536j \\ -1. \\ 1.121 + 0.536j \\ -1.j \\ -3.121 + 6.536j \end{bmatrix} \quad \overline{H} = \begin{bmatrix} 1.312 + 0.j \\ 0.864 - 0.525j \\ 0. \\ 0.511 + 1.85j \\ 3.938 \\ 0.511 - 1.85j \\ 0. \\ 0.864 + 0.525j \end{bmatrix} \tag{2.4.2}$$

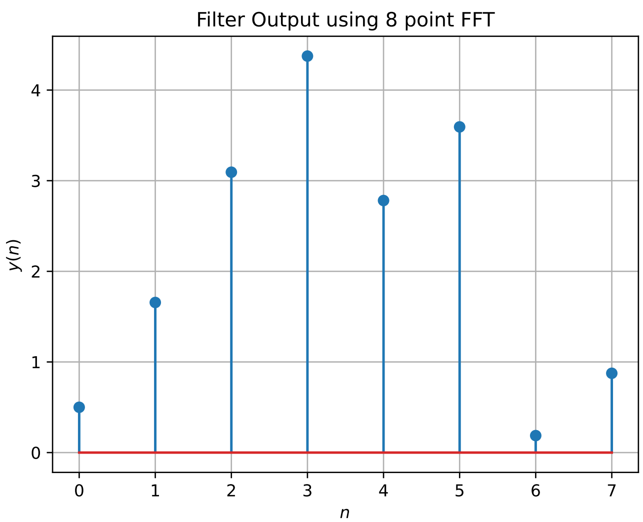

Fig. 2.4: y(n) obtained using 8 point recursive FFT

2.5. Formulating a recursive N-point FFT Algorithm
($N = 2^\gamma; \gamma$ is an integer):
An N-point DFT can be written as :

$$X(k) = \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N}, \quad k = 0, 1, \ldots, N-1 \tag{2.5.1}$$

(2.5.2)

By dividing the inputs into even and odd indices ; where $W_N = e^{\frac{-j2\pi}{N}}$

$$\begin{aligned} X_k &= \sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{j2\pi k2m}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{j2\pi k(2m+1)}{N}} \\ &= \underbrace{\sum_{m=0}^{N/2-1} x_{2m} e^{-\frac{j2\pi km}{N/2}}}_{\text{N/2 DFT with even inputs}} + W_N^k \underbrace{\sum_{m=0}^{N/2-1} x_{2m+1} e^{-\frac{j2\pi k(m)}{N/2}}}_{\text{N/2 DFT with odd inputs}} \end{aligned} \tag{2.5.3}$$

While exploiting symmetry of W_N as :

$$W_N^{k+N/2} = -W_N^k \tag{2.5.4}$$

We can transform the iterative problem to a Divide-Conquer algorithm, where :

$$X_{0 \to \frac{N}{2}-1} = X_{even} + \overline{W}_{N/2} * X_{odd} \tag{2.5.5}$$

$$X_{\frac{N}{2} \to N-1} = X_{even} - \overline{W}_{N/2} X_{odd} \tag{2.5.6}$$

$$\overline{W}_{N/2}(i) = W_N^i$$

; for i = 0,1,2 ... (N/2) -1

Where X_{even} and X_{odd} are again recursively computed using (N/2)-FFT thus halving its computation time; until N = 2(the base case of the recursion) for which a 2-point DFT is computed as follows.

$$X = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} * x \tag{2.5.7}$$

Thus , the time complexity of the algorithm is $O(Nlog_2N)$

2.6. Vector representation of the FFT algorithm
An 8-point DFT can be represented as a Matrix product as follows:

$$\overline{X} = \overline{W} \, \overline{x}$$

$$\overline{x} = \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \\ x(4) \\ x(5) \\ x(6) \\ x(7) \end{bmatrix} \quad \overline{X} = \begin{bmatrix} X(0) \\ X(1) \\ X(2) \\ X(3) \\ X(4) \\ X(5) \\ X(6) \\ X(7) \end{bmatrix} \tag{2.6.1}$$

$$\overline{W} = \begin{bmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^2 & W^4 & W^6 & W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^1 & W^4 & W^7 & W^2 & W^5 \\ W^0 & W^4 & W^0 & W^4 & W^0 & W^4 & W^0 & W^4 \\ W^0 & W^5 & W^2 & W^7 & W^4 & W^1 & W^6 & W^3 \\ W^0 & W^6 & W^4 & W^2 & W^0 & W^6 & W^4 & W^2 \\ W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1 \end{bmatrix} \tag{2.6.2}$$

where $W = W_8 = e^{-j2\pi/8}$

The FFT algorithm exploits the inherent symmetry in $\overline{W}$ matrix by permuting $\overline{x}$ in a bit-reversed fashion.
A 8 point FFT can be represented as :

$$\overline{X} = \overline{W_p}\ \overline{x_p}$$

$$\overline{x}_p = P\ \overline{x}$$

The P matrix rearranges the input x vector in a bit-reversed fashion as in :

$$x_p(i) = x(bit\ reverse(i)) : \tag{2.6.3}$$

For Eg ;

$$x_p(4) = x_p(bin(100)) = x(bin(001)) = x(1) \tag{2.6.4}$$

$$P = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{2.6.5}$$

For such a rearrangement of $\overline{x}$, we can exploit the symmetry in W_p, and thus factorise it into 3 sparse matrices.

$$\overline{W}_p = \overline{W3}\ \overline{W2}\ \overline{W1} \tag{2.6.6}$$

$$\overline{W3} = \begin{bmatrix} 1 & . & . & . & W^0 & . & . & . \\ . & 1 & . & . & . & W^1 & . & . \\ . & . & 1 & . & . & . & W^1 & . \\ . & . & . & 1 & . & . & . & W^1 \\ 1 & . & . & . & -W^0 & . & . & . \\ . & 1 & . & . & . & -W^1 & . & . \\ . & . & 1 & . & . & . & -W^1 & . \\ . & . & . & 1 & . & . & . & -W^1 \end{bmatrix} \tag{2.6.7}$$

$$\overline{W2} = \begin{bmatrix} 1 & . & W^0 & . & . & . & . & . \\ . & 1 & . & W^2 & . & . & . & . \\ 1 & . & -W^0 & . & . & . & . & . \\ . & 1 & . & -W^2 & . & . & . & . \\ . & . & . & . & 1 & . & W^0 & . \\ . & . & . & . & . & 1 & . & W^2 \\ . & . & . & . & 1 & . & -W^0 & . \\ . & . & . & . & . & 1 & . & -W^2 \end{bmatrix} \tag{2.6.8}$$

$$\overline{W1} = \begin{bmatrix} 1 & W^0 & . & . & . & . & . & . \\ 1 & -W^0 & . & . & . & . & . & . \\ . & . & 1 & W^0 & . & . & . & . \\ . & . & 1 & -W^0 & . & . & . & . \\ . & . & . & . & 1 & W^0 & . & . \\ . & . & . & . & 1 & -W^0 & . & . \\ . & . & . & . & . & . & 1 & W^0 \\ . & . & . & . & . & . & 1 & -W^0 \end{bmatrix} \tag{2.6.9}$$

where (.) refers to zero.
Similarly a N-point DFT's W matrix can be factorised into γ sparse matrices , $(N = 2^\gamma)$, with each row containing a 1 and a complex no. These γ sparse matrices represent the γ-stages in the butterfly diagram of an N-point FFT.

2.7. Run time analysis of FFT

Considering no. of multiplications as a metric for time complexity:
1. In N-point DFT, the dense matrix multiplication consist of $2N^2$ real multiplications. Hence time complexity of DFT is $\boldsymbol{O(N^2)}$

2.While in FFT , there are logN(stages) sparse

matrices, each stage requires 4*(N/2) real unique multiplications.
Thus, the total multiplications for N-FFT is 2*N*logN which implies a time complexity of $\boldsymbol{O(NlogN)}$

The below code compares time-complexities of DFT and FFT :

https://github.com/yaswanthNaidu99/ASSIGNMENT1/blob/main/codes/dft_fft.py

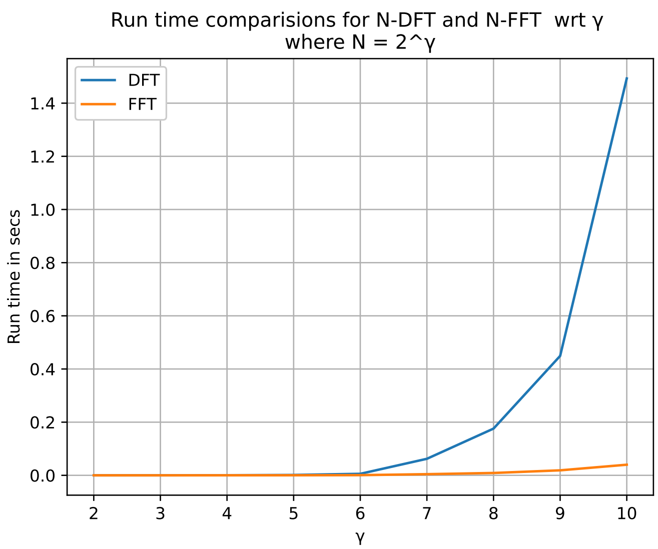

Fig. 2.7: Time complexity comparision

2.8. Convolution vs FFT
A convolution takes N^2 operations $\approx \boldsymbol{O(N^2)}$.
While the same output can be achieved using FFT and IFFT within :

$$\underbrace{O(NlogN)}_{\substack{x \to X \\ h \to H}} + \underbrace{O(N)}_{Y = X * H} + \underbrace{O(NlogN)}_{Y \to y} \approx \boldsymbol{O(NlogN)} \tag{2.8.1}$$