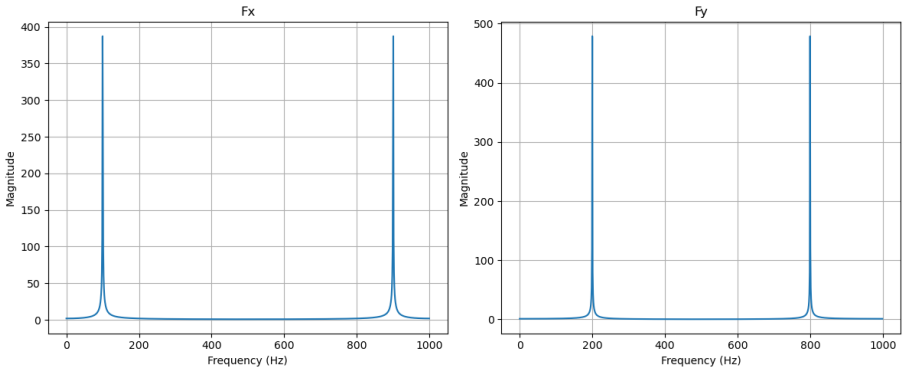
ADSP: HW5

Lo Chun, Chou
R13922136

June 11, 2025

(1)

An example result of the code is shown as the following image:



For the code, please refer to the attached python file on NTUCOOL.

(2)

The two main advantages of the sectioned convolution compared with the original non-sectioned convolutions are:

- We can reduce computation by sectioning, since the complexity of sectioned convolution is linear with N ($\theta(N)$), however, for non-sectioned convolution, the complexity is $\theta(N \log N)$
- Since the restriction $P \geq M+N-1$ needs to be satisfied, when the number of N changes, the number of P should also change.

  Even though this could be easily implemented by software, it is not realistic for hardware implementation, because implementing multiple fourier

transforms on a chip with different Ns, in order to deal with inputs of different length is too costly.

However, if we split the input signal into multiple sections, with each section of length L, for the number of points of the fourier transform (P) to satisfy the inequality $P \geq M + L - 1$, since L is fixed, P would not be affected by the length of the input signal N.

(3)

To compute the convolution operation of $y[n] = x[n] * h[n]$, where:

$$h[n] = [0.09\ 0.36\ 0.55\ 0.55\ 0.36\ 0.09] \quad -2 \leq n \leq 3$$

by an efficient way that the number of multiplciations is minimized, we can

(4)

Given that the length($x[n]$) = 1500, we can find the optimal approach by the cases in lecture note "ADSP_Write6.pdf", p.437, 441. And the following are the formulas showing the amount of multiplications for each case:

- For direct computing, we have:

$$3N \times M, \qquad \text{where } N = \text{length}(x[n]),\ M = \text{length}(h[n])$$

- If we use non-sectioned convolution, which is $y[n] = \text{IFFT}_P(\text{FFT}_P\{x[n]\} \times \text{FFT}_P\{h[n]\})$

$$2 \times \text{MUL}_P + 3P, \qquad \text{where } P \geq N + M - 1$$

- If we use sectioned convolution, we have;

$$S \times (2 \times \text{MUL}_P + 3P), \qquad \text{where } S = \left\lceil \frac{N}{L} \right\rceil$$

(a)

If length$(y[n]) = 250$, then we have $N = 1500,\ M = 250$.

Direct computing:

$$3 \times 1500 \times 250 = 1125000$$

Non-sectioned convolution:

We have $P \geq 1500 + 250 - 1 = 1749$, before applying the formula, we first calculate MUL_P by the code, and we could get the resulting optimal P as 2016.

Thus, the number of multiplications using IFFT and FFT with $P = 2016$ ($\text{MUL}_{2016} = 12728$) is:

$$2 \times \text{MUL}_{2016} + 3 \times 2016 = 2 \times 12728 + 3 \times 2016 = 25456 + 6048 = 31504$$

Sectioned convolution:

(b)

If length$(y[n]) = 50$, then we have $N = 1500,\ M = 50$.

Direct computing:

$$3 \times 1500 \times 50 = 225000$$

Non-sectioned convolution:

(c)

(5)

(a)

From the lecture note "ADSP_Write7.pdf", p.483, we know that Walsh transform will only become multiplication under logical convolution, which is shown as follows:

Walsh transform: Convolution property

Let $\Rightarrow$ denote the Walsh transform, and $\star$ denote the logical convolution, then we have:

$$\text{If } f[n] \Rightarrow F[m],\ g[n] \Rightarrow G[m],\ \text{then } f[n] \star g[n] \Rightarrow F[m] \times G[m]$$

While we do not have this property under linear convolution, thus, Walsh transform is not suitable for calculating the linear convolution.

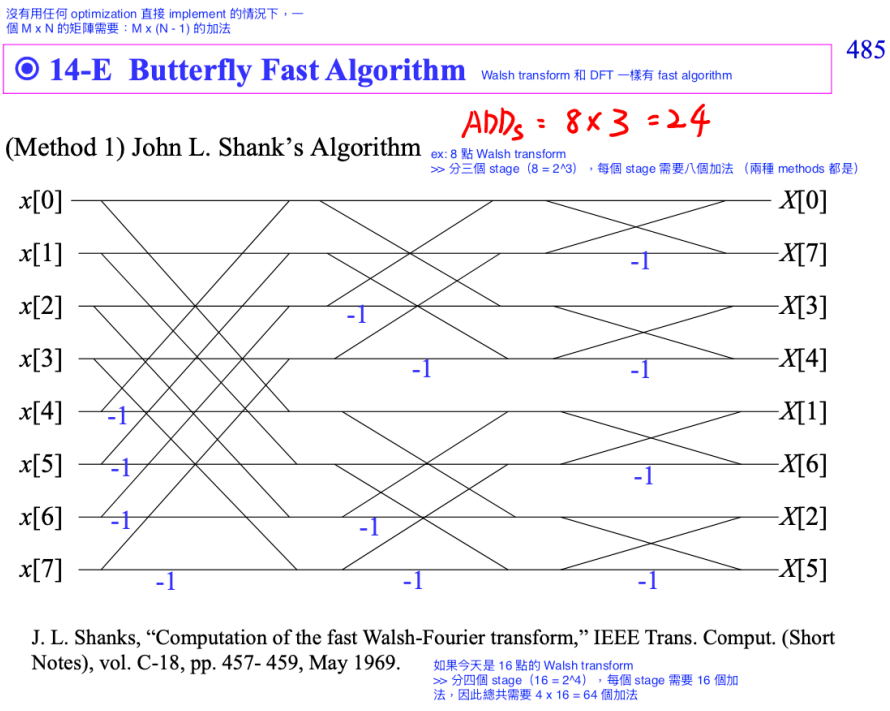
(b)

Stair-like signal anaylsis is suitable for the Walsh transform, since the Walsh transform is a set of orthogonal functions, and the stair-like signal is a combination of step functions.

(6)

(a)

For the 32 point Walsh transform, by using the similar method as in the lecture note "ADSP_Write7.pdf", p.485, which is shown below:

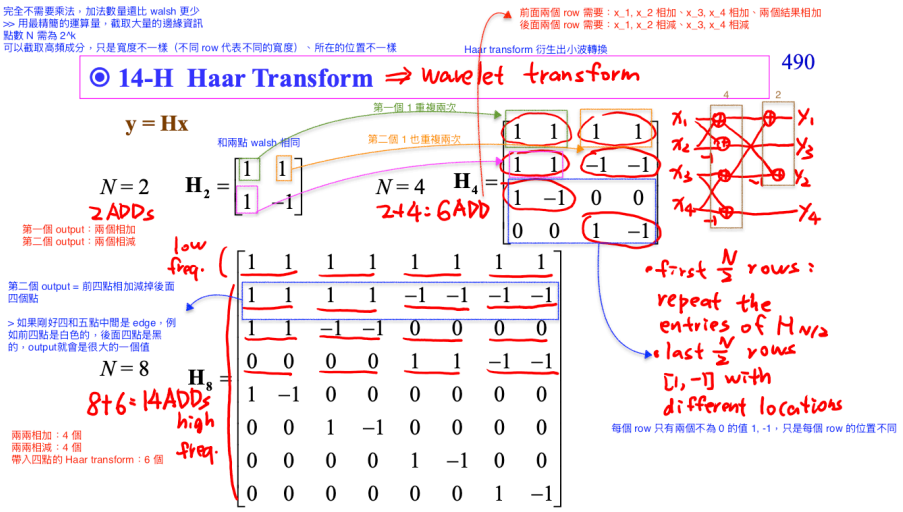


We first need to determine the number of stages needed, then determine the number of additions required for each stage.
Since we have 32 points, which is 2^5, we need 5 stages, and for each stage, we need 32 additions, so the total number of additions needed is:

$$5 \times 32 = 160 \qquad \square$$

(b)

For the 16 point Haar transform, we formulate the matrix as shown in the following lecture note "ADSP_Write7.pdf", p.490:



We need the 8-point Haar transform to realize the 16-point Haar transform, and before applying the 8-point Haar transform, we need every two points to be added and subtracted together, which requires:

1. $16 \div 2 = 8$ (for additions $x_i + x_{i+1},\ i = 0, 2, 4, 6, 8, 10, 12, 14$)
2. $16 \div 2 = 8$ (for subtractions $x_i - x_{i+1},\ i = 1, 3, 5, 7, 9, 11, 13, 15$)

Thus, the total number of additions needed is:

$$\underbrace{8}_{x_i + x_{i+1}} + \underbrace{8}_{x_i - x_{i+1}} + \underbrace{8}_{8 \text{ point Haar}} = 24 \qquad \square$$

7

For the subproblems (a) and (b), I use the code as the attached images to generate the results. Simple explanations are shown in the markdown cells.

(a)

Generate the 16 point Walsh matrix

using scipy

`hadamard(n, dtype=<class 'int'>)`

- `n` is the order of the matrix (must be power of 2)

```
1  import numpy as np
2  from scipy.linalg import hadamard
3  V = hadamard(16)
4  print(V)
```

[44] ✓ 0.0s Python

```
[[ 1  1  1  1  1  1  1  1  1  1  1  1  1  1  1  1]
 [ 1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1  1 -1]
 [ 1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1]
 [ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]
 [ 1  1  1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1]
 [ 1 -1  1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1]
 [ 1  1 -1 -1 -1 -1  1  1  1  1 -1 -1 -1 -1  1  1]
 [ 1 -1 -1  1 -1  1  1 -1  1 -1 -1  1 -1  1  1 -1]
 [ 1  1  1  1  1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1]
 [ 1 -1  1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1]
 [ 1  1 -1 -1  1  1 -1 -1 -1 -1  1  1 -1 -1  1  1]
 [ 1 -1 -1  1  1 -1 -1  1 -1  1  1 -1 -1  1  1 -1]
 [ 1  1  1  1 -1 -1 -1 -1 -1 -1 -1 -1  1  1  1  1]
 [ 1 -1  1 -1 -1  1 -1  1 -1  1 -1  1  1 -1  1 -1]
 [ 1  1 -1 -1 -1 -1  1  1 -1 -1  1  1  1  1 -1 -1]
 [ 1 -1 -1  1 -1  1  1 -1 -1  1 -1  1 -1 -1  1  1]]
```

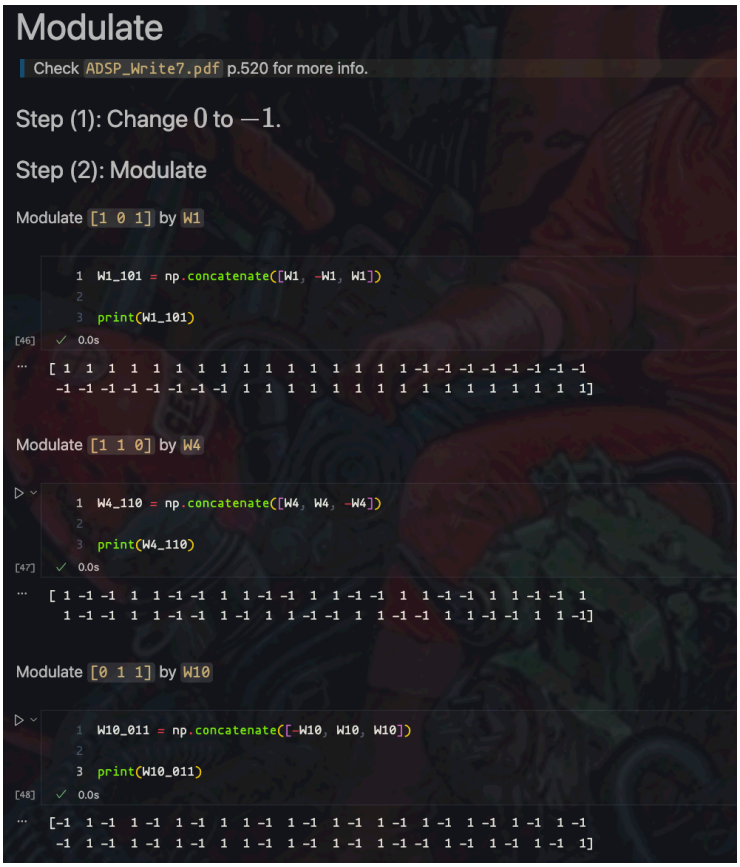
Get the required row values:

row: 1, 4, 10

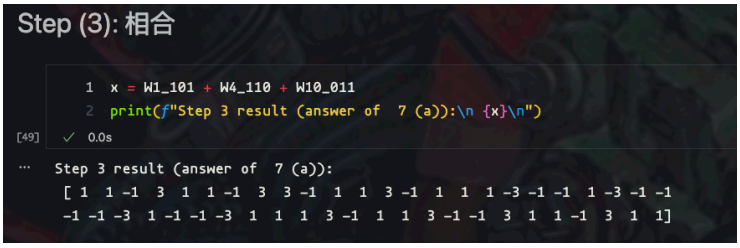
```
1  W1 = V[0]
2  W4 = V[3]
3  W10 = V[9]
4
5  print(f"W1: {W1}\n")
6  print(f"W4: {W4}\n")
7  print(f"W10: {W10}\n")
```

[45] ✓ 0.0s

```
W1: [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1]

W4: [ 1 -1 -1  1  1 -1 -1  1  1 -1 -1  1  1 -1 -1  1]

W10: [ 1 -1  1 -1  1 -1  1 -1 -1  1 -1  1 -1  1 -1  1]
```



And the answer of (a) is as the result of the image below:



Since the image might not be that clear, the answer is:

$$\begin{bmatrix} 1 & 1 & -1 & 3 & 1 & 1 & -1 & 3 & 3 & -1 & 1 & 1 & 3 & -1 & 1 & 1 \\ 1 & -3 & -1 & -1 & 1 & -3 & -1 & -1 & -1 & -1 & -3 & 1 & -1 & -1 & -3 & 1 \\ 1 & 1 & 3 & -1 & 1 & 1 & 3 & -1 & -1 & 3 & 1 & 1 & -1 & 3 & 1 & 1 \end{bmatrix} \quad \square$$

(b)

Demodulation

Check ADSP_Write7.pdf p.521 for more info.

```
x1 = np.split(x, 3)[0]
x2 = np.split(x, 3)[1]
x3 = np.split(x, 3)[2]

print(f"x1: {x1}\n")
print(f"x2: {x2}\n")
print(f"x3: {x3}\n")
```

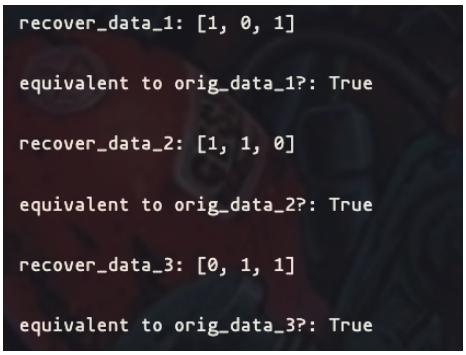
[50] ✓ 0.0s

```
x1: [ 1  1 -1  3  1  1 -1  3  3 -1  1  1  3 -1  1  1]

x2: [ 1 -3 -1 -1  1 -3 -1 -1 -1 -1 -3  1 -1 -1 -3  1]

x3: [ 1  1  3 -1  1  1  3 -1 -1  3  1  1 -1  3  1  1]
```

After getting the values of `x1`, `x2`, `x3`, we try to recover the original data by the following process, note that only the recovery process of `x1` is shown since the rest are the same:

```
orig_data_1 = [1, 0, 1]
orig_data_2 = [1, 1, 0]
orig_data_3 = [0, 1, 1]

recover_data_1 = []
res = np.inner(x1, W1) / 16
if res > 0:
    recover_data_1.append(1)
else:
    recover_data_1.append(0)

res = np.inner(x2, W1) / 16
if res > 0:
    recover_data_1.append(1)
else:
    recover_data_1.append(0)

res = np.inner(x3, W1) / 16
if res > 0:
    recover_data_1.append(1)
else:
    recover_data_1.append(0)

print(f"recover_data_1: {recover_data_1}\n")
print(f"equivalent to orig_data_1?: {recover_data_1 == orig_data_1}\n")
```

The recover result is as follows:



Thus, we can recover the original data by the process shown in the image above.

(c)

Extra problem (ID ends with $1, 6$)

How many real multiplications are needed when the length of the input function is 100 points ($N = \text{length}(x[n]) = 100$), the filter is 19 points ($M = \text{length}(h[n]) = 19$), and we want to implement it by a 120 points Fourier transform ($P = 120$)?

$$\begin{aligned} & 2\text{MUL}_{120} + 120 \times 3 \\ = \; & 2 \times 380 + 120 \times 3 \\ = \; & 760 + 360 \\ = \; & 1120 \end{aligned}$$

Note that we multiplied 120 by 3 since one complex multiplication requires 3 real multiplications, and we got the value of $\text{MUL}_{120} = 380$ by the table in lecture note "ADSP_Write5.pdf", p.378.