NAME: Vishal Shashikant Salvi **UID: 2019230069**
CLASS: TE COMPS **BATCH: C**

EXPERIMENT NO 7

Aim: To implement the Overlap Add Method /overlap-save method for discrete convolution.

Theory:

The overlap-add method is used to break long signals into smaller segments for easier processing. FFT convolution uses the overlap-add method together with the Fast Fourier Transform, allowing signals to be convolved by multiplying their frequency spectra. For filter kernels longer than about 64 points, FFT convolution is faster than standard convolution, while producing exactly the same result.

The Overlap Add Method:

There are many DSP applications where a long signal must be filtered in segments . For instance, high fidelity digital audio requires a data rate of about 5 Mbytes/min, while digital video requires about 500 Mbytes/min. With data rates this high, it is common for computers to have insufficient memory to simultaneously hold the entire signal to be processed. There are also systems that process segment-by-segment because they operate in real time . For example, telephone signals cannot be delayed by more than a few hundred milliseconds, limiting the amount of data that are available for processing at any one instant. In still other applications, the processing may require that the signal be segmented. An example is FFT convolution, the main topic of this article.

The overlap-add method is based on the fundamental technique in DSP: (1) decompose the signal into simple components, (2) process each of the components in some useful way, and (3) recombine the processed components into the final signal. Figure 18-1 shows an example of how this is done for the overlap-add method. Figure (a) is the signal to be filtered, while (b) shows the filter kernel to be used, a windowed-sinc low-pass filter. Jumping to the bottom of the figure, (i) shows the filtered signal, a smoothed version of (a). The key to this method is how the lengths of these signals are affected by the convolution. When an N sample signal is convolved with an M sample filter kernel, the output signal is N + M – 1 samples long. For instance, the input signal, (a), is 300 samples (running from 0 to 299), the filter kernel, (b), is 101 samples (running from 0 to 100), and the output signal, (i), is 400 samples (running from 0 to 399).

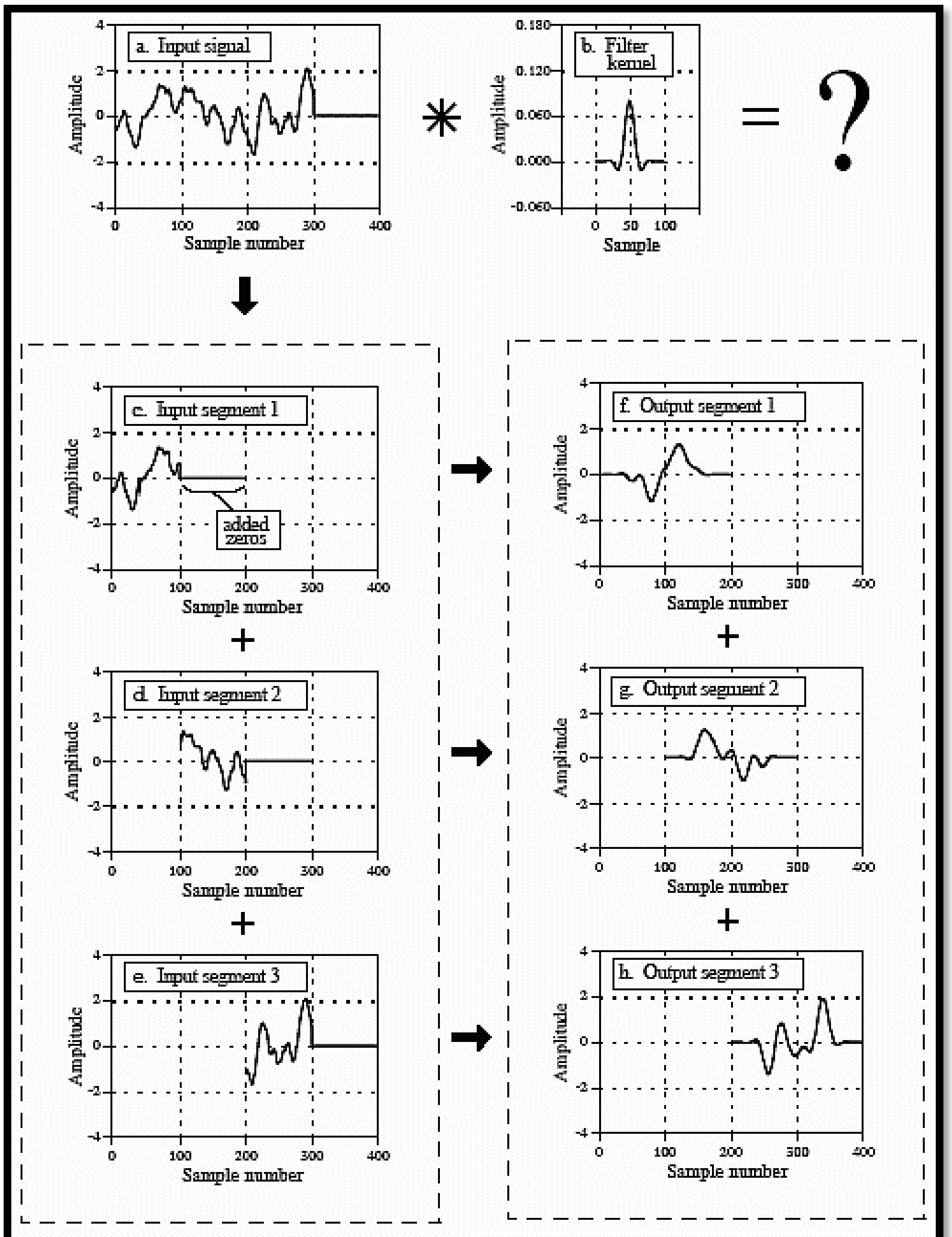
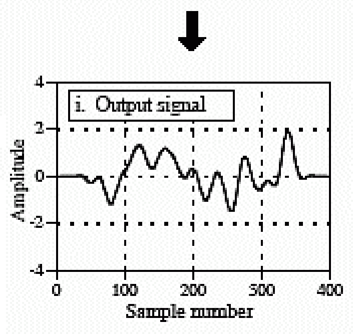


FIGURE 18-1
The overlap-add method. The goal is to convolve the input signal, (a), with the filter kernel, (b). This is done by breaking the input signal into a number of segments, such as (c), (d) and (e), each padded with enough zeros to allow for the expansion during the convolution. Convolving each of the input segments with the filter kernel produces the output segments, (f), (g), and (h). The output signal, (i), is then found by adding the overlapping output segments.



In other words, when an N sample signal is filtered, it will be expanded by M – 1 points to the right . (This is assuming that the filter kernel runs from index 0 to M . If negative indexes are used in the filter kernel, the expansion will also be to the left). In (a), zeros have been added to the signal between sample 300 and 399 to illustrate where this expansion will occur. Don't be confused by the small values at the ends of the output signal, (i). This is simply a result of the windowed-sinc filter kernel having small values near its ends. All 400 samples in (i) are nonzero, even though some of them are too small to be seen in the graph.

Figures (c), (d) and (e) show the decomposition used in the overlap-add method. The signal is broken into segments, with each segment having 100 samples from the original signal. In addition, 100 zeros are added to the right of each segment. In the next step, each segment is individually filtered by convolving it with the filter kernel. This produces the output segments shown in (f), (g), and (h). Since each input segment is 100 samples long, and the filter kernel is 101 samples long, each output segment will be 200 samples long. The important point to understand is that the 100 zeros were added to each input segment to allow for the expansion during the convolution.

Notice that the expansion results in the output segments overlapping each other. These overlapping output segments are added to give the output signal, (i). For instance, samples 200 to 299 in (i) are found by adding the corresponding samples in (g) and (h). The overlap-add method produces exactly the same output signal as direct convolution. The disadvantage is a much greater program complexity to keep track of the overlapping samples.

3.10.1 Overlap - Save Method

As mentioned earlier, we break x(n) into smaller blocks of length N. (x_1 (n), x_2 (n), ...). It is important to note that while the length N can be our choice, it should be longer than the length of h (n).

i.e. $N >$ length h (n).

We use the following notations.

Where, $N =$ Length of each block

$M =$ Length of h (n)

$L =$ Values from current x(n)

$\therefore \quad N = (M - 1) + L$

- Each block x_i (n) of length N is formed such that it contains (M – 1) values from the previous block and the remaining L values from the current x(n).
- Since there is an overlap of (M – 1) values from the previous block, it known as overlap- save method.
- As the 1st block does not have a previous block, we append (M – 1) zeros to it at the beginning. A pictorial representation would help us understand this better.

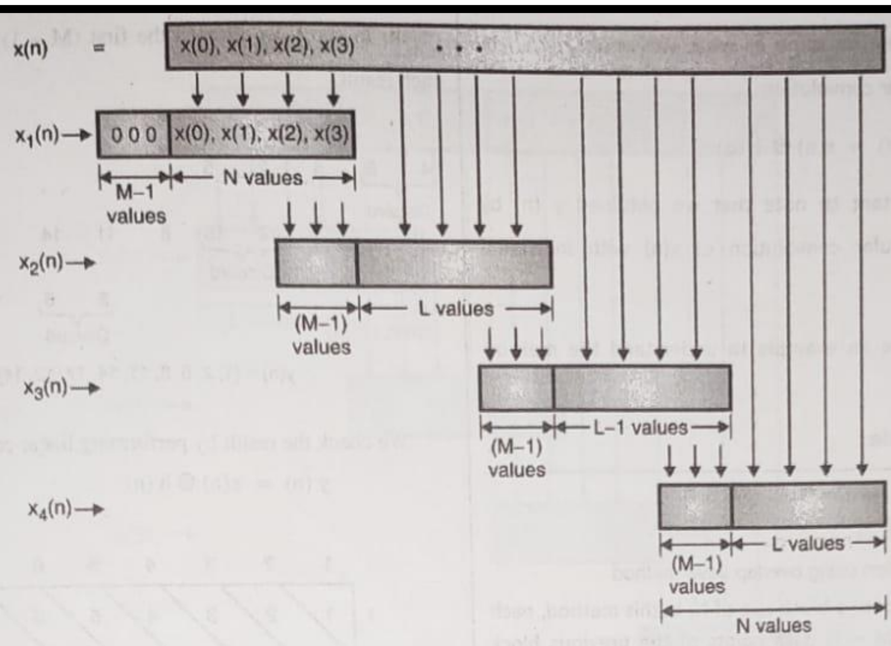


Fig. 3.10.1

Once these block are created, each of these are "circularly convolved" with h (n).

i.e. $y_1(n) = x_1(n) \circledast h(n)$

$y_2(n) = x_2(n) \circledast h(n)$

$\vdots$ $\vdots$ etc.

The final filtered output y (n) is obtained by discarding the first (M − 1) values of each result. $y_1(n)$, $y_2(n)$, ... and appending the remaining values. A pictorial representation is shown in Fig. 3.10.2.

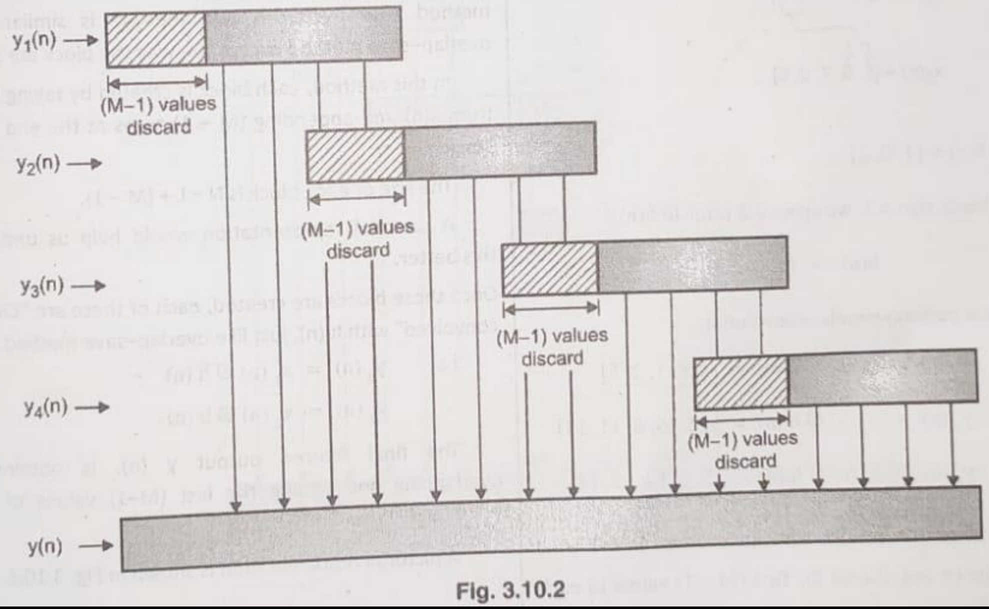
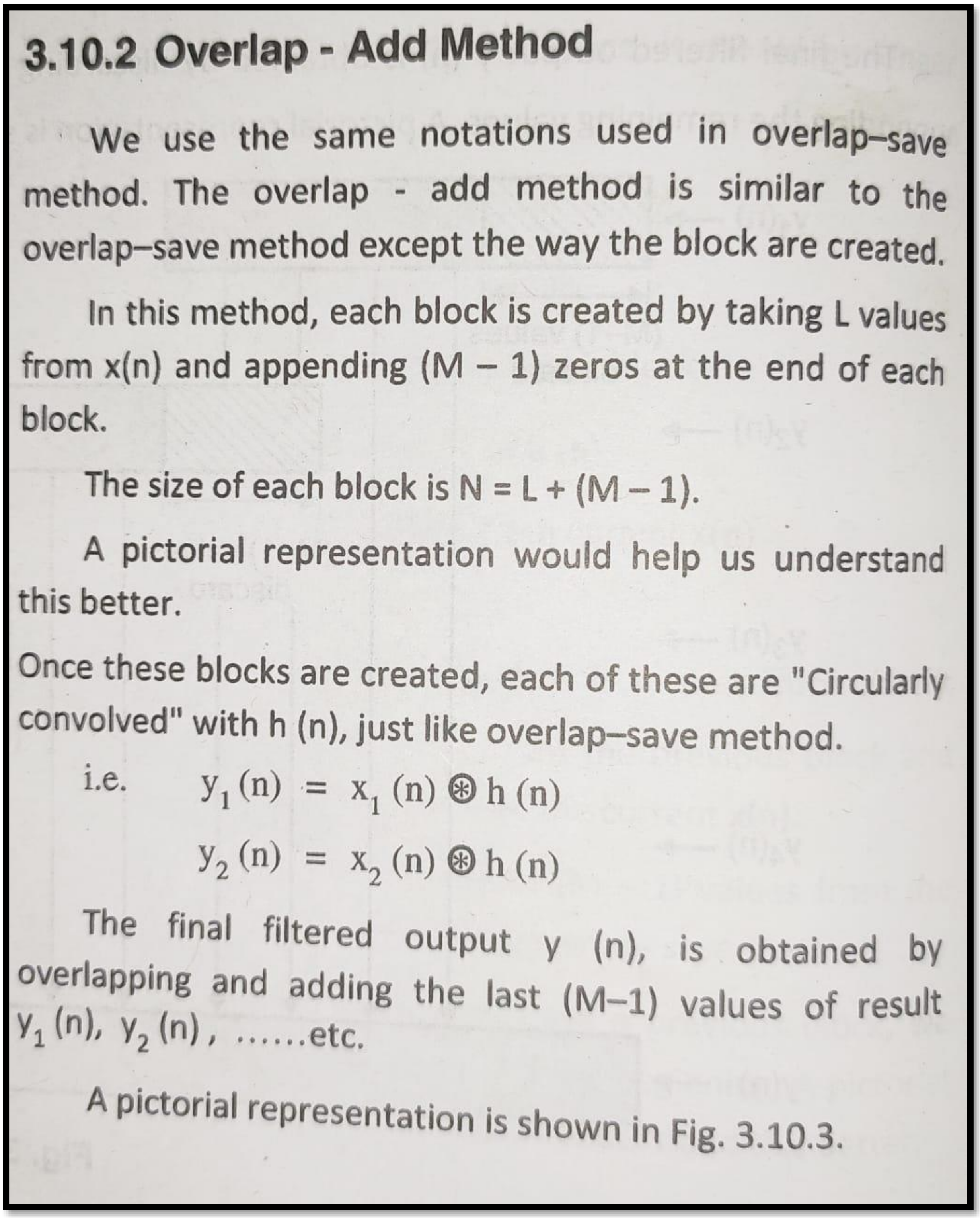


Fig. 3.10.2



3.10.2 Overlap - Add Method

We use the same notations used in overlap–save method. The overlap - add method is similar to the overlap–save method except the way the block are created.

In this method, each block is created by taking L values from x(n) and appending (M – 1) zeros at the end of each block.

The size of each block is $N = L + (M - 1)$.

A pictorial representation would help us understand this better.

Once these blocks are created, each of these are "Circularly convolved" with h (n), just like overlap–save method.

i.e. $$y_1(n) = x_1(n) \circledast h(n)$$

$$y_2(n) = x_2(n) \circledast h(n)$$

The final filtered output y (n), is obtained by overlapping and adding the last (M–1) values of result $y_1(n)$, $y_2(n)$, ...etc.

A pictorial representation is shown in Fig. 3.10.3.

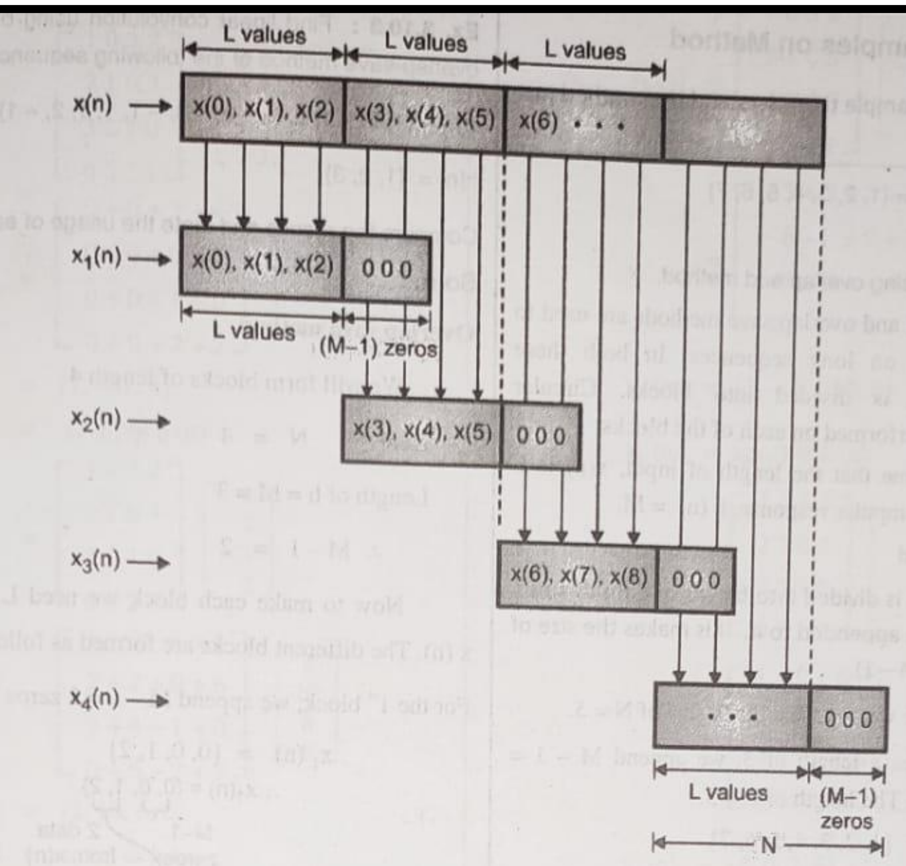


Fig. 3.10.3

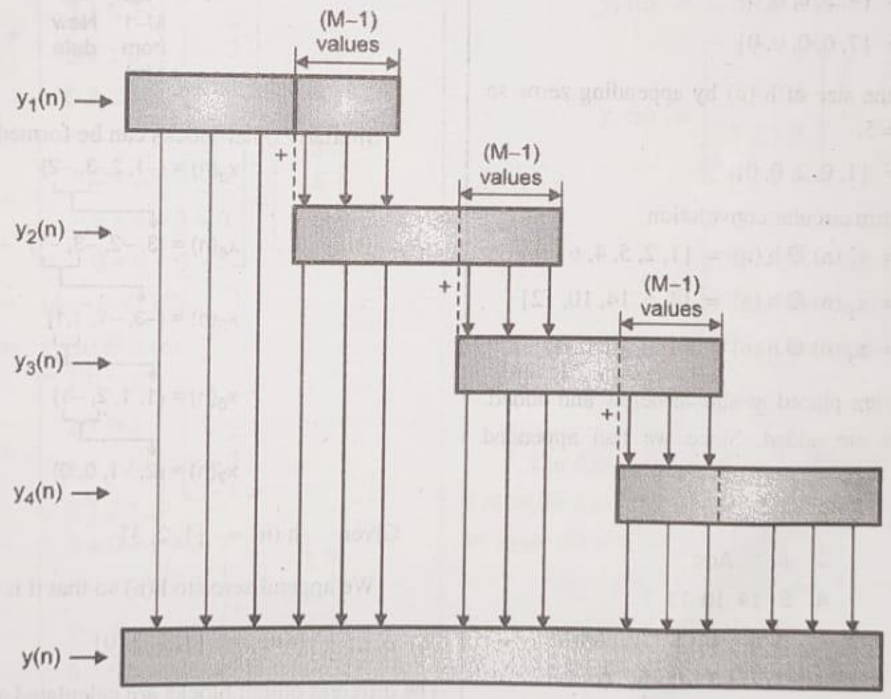


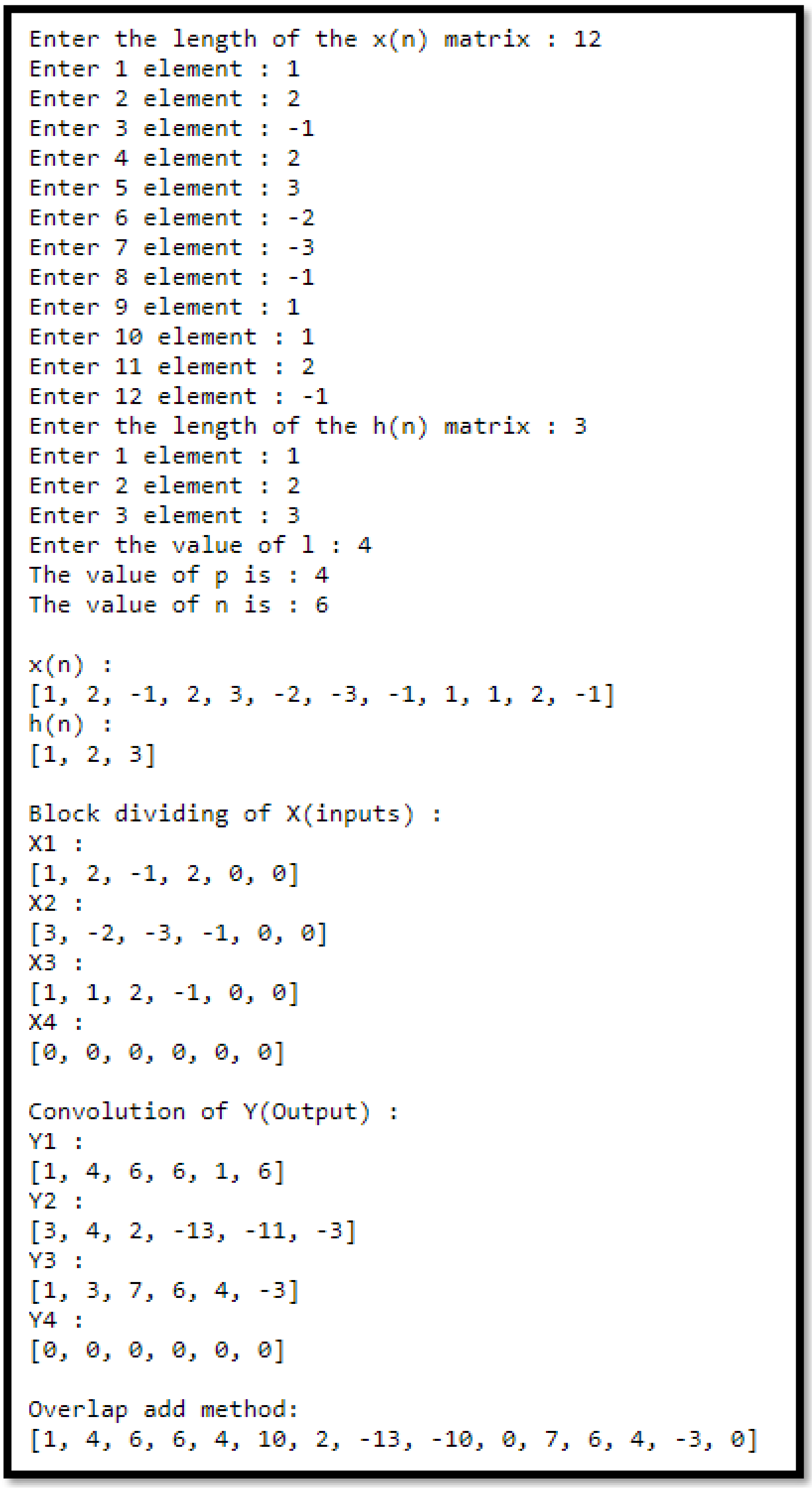
Fig. 3.10.4

Code:(Overlap Add)

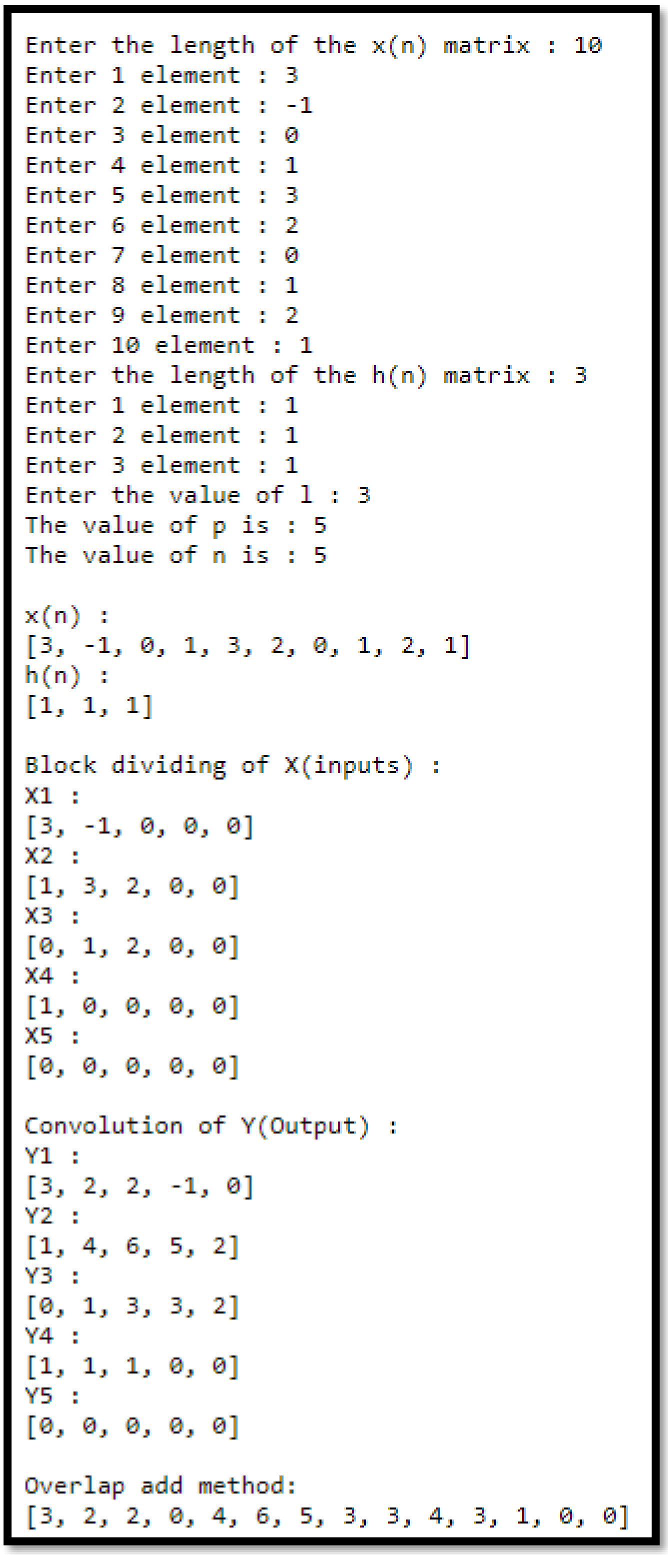
```
def givep(f, l):
    temp = f / l
    if (f % l == 0):
        P = temp + 1
    else:
        P = temp + 2
    return P


def returnx(l, p, n, mat1):
    x = [[0 for row in range(0, int(n))] for col in range(0, int(p))]
    z = int(0)
    lenmat1 = len(mat1)
    for row in range(0, p - 1):
        value = 0
        for col in range(0, n):
            if (value < l):
                if (z < lenmat1):
                    x[row][col] = mat1[z]
                    value = value + 1
                    z = z + 1
                else:
                    x[row][col] = 0
    return x


def findy(p, n, l, x, mat2):
    sfinal = []
    h = len(mat2)
    run = n - h
    for z in range(1, run + 1):
        mat2.append(0)
    y = [[0 for row in range(0, int(n))] for col in range(0, int(p))]
    temp = []
    for i in range(0, p):
        for j in range(0, n):
            temp.append(x[i][j])
        sfinal = convol(temp, mat2)
        for j in range(0, n):
            y[i][j] = sfinal[j]
        del sfinal[:]
        del temp[:]
    return y


def inputmat(z):
    mat = []
    for v in range(1, z + 1):
        q = int(input('Enter %d element : ' % (v)))
        mat.append(q)
    return mat


def check_diff(mat1, mat2):
    x = len(mat1)
    h = len(mat2)
    temp = []
    if (x > h):
        run = x - h
        for z in range(1, run + 1):
            mat2.append(0)
        temp = convol(mat1, mat2)
        return mat1, mat2
    else:
        run = h - x
        for z in range(1, run + 1):
            mat1.append(0)
        temp = convol(mat2, mat1)
        return mat2, mat1


def convol(large, small):
    lenlarge = len(large)
    templist = [0] * lenlarge
    temp = 0
    yn = []
    # print('Last Element is : %d'%(large[lenlarge-1]))
    # convol=[]
    convol = [[0 for row in range(0, lenlarge)] for col in range(0, lenlarge)]
    for r in range(0, lenlarge):
        if (r > 0):
            first = large[0]
            last = large[lenlarge - 1]
            for q in range(0, lenlarge):
                if (q > 0 & q < lenlarge):
                    templist[q] = large[q - 1]
                    convol[r][q] = templist[q]
                else:
                    templist[0] = last
                    convol[r][q] = templist[q]
            for p in range(0, lenlarge):
                large[p] = templist[p]
        else:
            for p in range(0, lenlarge):
                convol[r][p] = large[p]
    yn = matmul(convol, small)
    return yn
```

```
def matmul(convol1, small):
    final = []
    lenlarge = len(convol1)
    for i in range(0, lenlarge):
        total = 0
        for j in range(0, lenlarge):
            total2 = convol1[j][i] * small[j]
            total = total + total2
        final.append(total)
    return final


def giveans(yn, n, p, l):
    ans = []
    var = 0
    temp = []
    for i in range(0, p):
        for j in range(0, n):
            if (j < l):
                if not temp:
                    var = yn[i][j]
                    ans.append(var)
                else:
                    var = yn[i][j]
                    var = var + temp[0]
                    ans.append(var)
                    del temp[0]
            else:
                temp.append(yn[i][j])
    a = len(ans)
    del ans[a - 1]
    return ans


def printvalue(p, n, mat, var):
    printit = []
    values = 1
    for i in range(0, p):
        print('%s%d : ' % (var, values))
        for j in range(0, n):
            printit.append(mat[i][j])
        print(printit)
        del printit[:]
        values += 1


def main():
    print('\n' * 5)
    mat1 = []
    large = []
    small = []
    x = []
    yn = []
    ans = []
    f = int(input('Enter the length of the x(n) matrix : '))
    mat1 = inputmat(f)
    m = int(input('Enter the length of the h(n) matrix : '))
    mat2 = inputmat(m)
    l = int(input('Enter the value of l : '))
    n = l + m - 1
    p = int(givep(f, l))
    print('The value of p is : %d ' % (p))
    print('The value of n is : %d ' % (n))
    print('\nx(n) : ')
    print(mat1)
    print('h(n) : ')
    print(mat2)
    x = returnx(l, p, n, mat1)
    print('\nBlock dividing of X(inputs) : ')
    variable = 'X'
    printvalue(p, n, x, variable)
    variable = 'Y'
    yn = findy(p, n, l, x, mat2)
    print('\nConvolution of Y(Output) : ')
    printvalue(p, n, yn, variable)
    ans = giveans(yn, n, p, l)
    print('\nOverlap add method: ')
    print(ans)


if __name__ == "__main__":
    main()
```

Output:



```
Enter the length of the x(n) matrix : 12
Enter 1 element : 1
Enter 2 element : 2
Enter 3 element : -1
Enter 4 element : 2
Enter 5 element : 3
Enter 6 element : -2
Enter 7 element : -3
Enter 8 element : -1
Enter 9 element : 1
Enter 10 element : 1
Enter 11 element : 2
Enter 12 element : -1
Enter the length of the h(n) matrix : 3
Enter 1 element : 1
Enter 2 element : 2
Enter 3 element : 3
Enter the value of l : 4
The value of p is : 4
The value of n is : 6

x(n) :
[1, 2, -1, 2, 3, -2, -3, -1, 1, 1, 2, -1]
h(n) :
[1, 2, 3]

Block dividing of X(inputs) :
X1 :
[1, 2, -1, 2, 0, 0]
X2 :
[3, -2, -3, -1, 0, 0]
X3 :
[1, 1, 2, -1, 0, 0]
X4 :
[0, 0, 0, 0, 0, 0]

Convolution of Y(Output) :
Y1 :
[1, 4, 6, 6, 1, 6]
Y2 :
[3, 4, 2, -13, -11, -3]
Y3 :
[1, 3, 7, 6, 4, -3]
Y4 :
[0, 0, 0, 0, 0, 0]

Overlap add method:
[1, 4, 6, 6, 4, 10, 2, -13, -10, 0, 7, 6, 4, -3, 0]
```

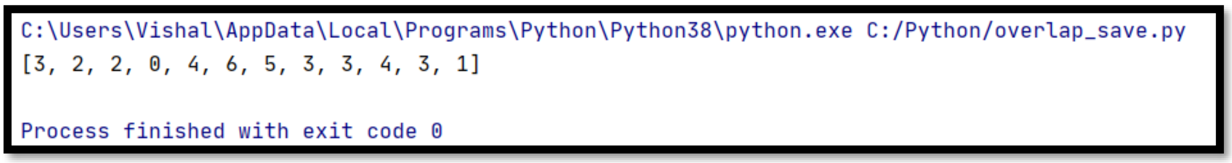


```
Enter the length of the x(n) matrix : 10
Enter 1 element : 3
Enter 2 element : -1
Enter 3 element : 0
Enter 4 element : 1
Enter 5 element : 3
Enter 6 element : 2
Enter 7 element : 0
Enter 8 element : 1
Enter 9 element : 2
Enter 10 element : 1
Enter the length of the h(n) matrix : 3
Enter 1 element : 1
Enter 2 element : 1
Enter 3 element : 1
Enter the value of l : 3
The value of p is : 5
The value of n is : 5

x(n) :
[3, -1, 0, 1, 3, 2, 0, 1, 2, 1]
h(n) :
[1, 1, 1]

Block dividing of X(inputs) :
X1 :
[3, -1, 0, 0, 0]
X2 :
[1, 3, 2, 0, 0]
X3 :
[0, 1, 2, 0, 0]
X4 :
[1, 0, 0, 0, 0]
X5 :
[0, 0, 0, 0, 0]

Convolution of Y(Output) :
Y1 :
[3, 2, 2, -1, 0]
Y2 :
[1, 4, 6, 5, 2]
Y3 :
[0, 1, 3, 3, 2]
Y4 :
[1, 1, 1, 0, 0]
Y5 :
[0, 0, 0, 0, 0]

Overlap add method:
[3, 2, 2, 0, 4, 6, 5, 3, 3, 4, 3, 1, 0, 0]
```

Code:(Overlap Save)

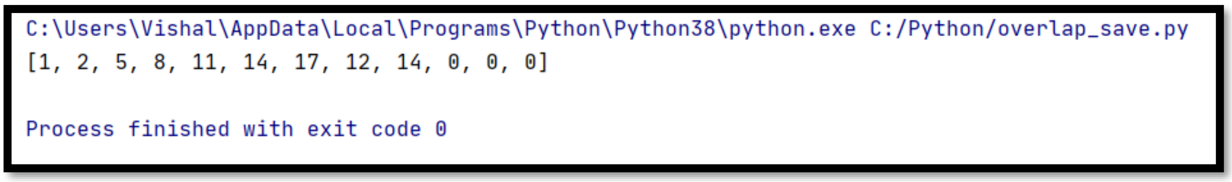
```
import math
def conv(h,g):
   if len(h) > len(g):
      h,g = g,h
   h += [0]*(len(g) - len(h))

   mat = [list(reversed(h[:i+1])) + list(reversed(h[i+1:])) for i in range(len(h))]

   y = [0]*(len(h))

   for i,row in enumerate(mat):
      for j, num in enumerate(row):
         y[i] += num*g[j]
   return y

def overlap_save_method(x,h):
   m = len(h)
   n = 2**m
   l = n - m + 1

   x_start = 0
   y = []
   prev = [0]*(n-l)

   for i in range(math.ceil(len(x)/l)):
      x_i = prev + x[x_start:x_start+l]
      x_i += [0]*(n - len(x_i))
      y += conv(x_i,h)[-l:]
      x_start += l
      prev = x_i[-(n-l):]
   return y

print(overlap_save_method([3,-1,0,1,3,2,0,1,2,1],[1,1,1]))
```

Output:



```
C:\Users\Vishal\AppData\Local\Programs\Python\Python38\python.exe C:/Python/overlap_save.py
[3, 2, 2, 0, 4, 6, 5, 3, 3, 4, 3, 1]

Process finished with exit code 0
```

X(n)= [1,2,3,4,5,6,7],
H(n)= [1,0,2]



```
C:\Users\Vishal\AppData\Local\Programs\Python\Python38\python.exe C:/Python/overlap_save.py
[1, 2, 5, 8, 11, 14, 17, 12, 14, 0, 0, 0]

Process finished with exit code 0
```

Conclusion:

- The overlap-add method allows us to calculate the convolution of very long sequences.
- The overlap-add method breaks a long sequence, x(n), into signals of shorter length and calculates the convolution of each block independently. To arrive at the final result, we need to apply an appropriate time shift to the convolution of the blocks and add them together.
- Analysing the computational complexity of the DFT-based method, we observe that there is an optimum value for the length of the input blocks.

References:

1) https://www.slideshare.net/GourabGhosh4/overlap-add-overlap-savedigital-signal-processing
2) https://www.tutorialspoint.com/digital_signal_processing/dsp_discrete_fourier_transform_sectional_convolution.htm
3) https://www.allaboutcircuits.com/technical-articles/overlap-add-method-linear-filtering-based-on-the-discrete-fourier-transform/