EE 2016

Mid-Semester Examinations

Part – C

EE19B118

SHASHANK NAG

+91 8433647537

Finite Impulse Response Filter

A finite impulse response filter is a one, whose impulse response goes to zero in a finite interval, i.e., it remains non-zero only for a certain range of n.

In DSP, the output sequence of an input sequence given to a FIR filter is given by:

<>

The number N+1 is known as tap of the filter. (“N+1 Tap Filter”)

Convolution

For two sequences, x[n] and h[n], their discrete time convolution is represented as x[n]*h[n], as is expressed as:

< >

Discrete Time Fourier Transform (DTFT)

The DTFT is a transformation of a discrete sequence (w.r.t. time), in terms of variable frequency. Essentially, the sequence is decomposed to be represented in terms of the components of different frequencies.

For a sequence x[n], the DTFT is given by:

Fixed Point Arithmetic Used

The fixed point arithmetic used in this program is the Q1.7 format. A number is represented by 1 byte. Out of the 8 bits, the leading bit represents the integral part (for positive numbers), and the trailing 7 bits denote the fractional part. This representation can be used only for representing decimal values in the range [-1,1].

MSB:7	6	5	4	3	2	1	LSB: 0
2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}

For negative numbers, they are represented by their 2s complement form. We add 2 to the decimal number, before converting them into the Q1.7 fixed point notation.

e.g. 0.75 is represented as 01100000

This is because $0.75=0.5+0.25 = 1x2^{-1} + 1x2^{-2}$

e.g. -0.5 is represented as 11000000

This is because, since -0.5 is negative, we add 2 to it. So 2-0.5=1.5.

$1.5 = 1 + 0.5 = 2^0 + 2^{-1}$

In the assembly program, the **FMULS** instruction has been used for fractional multiplication. When two 8-bit numbers are multiplied, the generated result is a 16 bit number. As the Q1.7 signed representation can be used to represent values in [-1,1] only, when we multiply 2 such numbers, the result also lies in [-1,1]. The **FMULS** instruction, shifts the 16-bit result, such that it is in the Q1.15 format, instead of Q2.14. So, the higher byte of the result is still in the Q1.7 format, and we drop the lower byte in the final result, as it just adds onto the accuracy.

DTFT Details

We have implemented DFT, using FFT functionality of MATLAB, as illustrated below:

<>

Implementation of FIR Filter in AVR : Implementation Details

INPUTS:

As we are using Q1.7 fixed point notation, we need to specify the output accordingly. As all the signals and coefficients are normalized to 1, we can be sure that all these can be represented using the Q1.7 notation.

To feed the input into Atmel Studio, we convert these inputs from decimal form [-1,1] into the decimal equivalent of their corresponding Q1.7 term, after quantisation. The final input sequence, to be fed into the AVR would be a sequence, where each value lies in [0,255](integral values).

e.g. the Q1.7 representation of 0.75 is 01100000. For giving input, we represent it by its equivalent decimal number, i.e., (2^7x0 + 2^6x1 + 2^5x1 + 2^4x0 + 2^3x0 + 2^2x0 + 2^1x0 + 2^0x0)=96.

The samples are given as input into the AVR using the .db directive at the end, as a sequence of .

The three inputs, a sinusoid, dc and white noise are given as separate input series using the .db directive at the end of the program, with two of them being commented out at a given time.

e.g.

```
SINE : .db 0,0,1,2,3,4,4,5,6,7,8,8,9,10,11,12,1 … (sequence of integers in [0,255])
```

Similarly, we give the inputs for dc and white noise, using .db directives. While finding the output for SINE input, we comment out the .db directives of dc and white noise.

This data is stored in the FLASH (program memory), and we access the inputs using the Z pointer. We use the Z pointer to read new data samples, and store them into the buffer (elaborated later).

OUTPUT:

The output is stored in the SRAM. We have allocated 900 bytes in the SRAM to store the output, which is stored using the SRAMRESULT label. We retrieve the address to this section, using the X pointer, and then store this address to the R7:R8 registers. So R7:R8 points to the beginning of the result location in SRAM. After each new output sample is computed, we store it in the location pointed to by R7:R8, and then increment this location.

For the 32 Tap Filter implementation, the output appears from the location 0xA0.

For the 5 Tap Filter implementation, the output appears from the location 0x6A.

The output received from the AVR is in the hexadecimal equivalent of the Q1.7 representation. We convert it into base 10, divide by 128, and if the result is greater than 1, we subtract 2.

Thus the output is also obtained in the range [-1,1]. To ensure this, and to avoid overflow, the coefficients have been scaled appropriately (elaborated later)

CIRCULAR BUFFER:

To implement a N tap filter, we require a buffer of size N, which hold the inputs x[n], x[n-1],…. X[n-(N-1)]. In this program, the circular buffer is implemented in the SRAM. The top of this buffer is pointed to by the BUFFER label, which has been allocated 32 bytes of SRAM memory (for 32 taps), and 5 bytes for 5 tap. The address to the start of the buffer is retrieved using the Y pointer, and is stored in R14:R15 registers. Similarly, R9:R10 points to the end of the buffer.

The pointer Y points to the location to which the new data point has to be loaded from the program memory. The oldest sample in the circular buffer, is replaced by the new sample, in a cyclic manner. While calculating the output sample at a particle instant, we traverse through all the points in the buffer, multiplying them by their respective filter coefficients to get the output sample at that instant (y[n]). After calculating y[n], the pointer Y is modified to point to the next oldest data in the buffer that is to be replaced in the next cycle.

FILTER COEFFICIENTS:

The filter coefficients are stored in the SRAM location corresponding to the label FILTERCOEFFICIENTS. We use the X pointer to point to/ access them, and the location of the first filter coefficient is pointed to by the R24:R25 registers.

The filter coefficients are also normalised and stored in the Q1.7 format.

MULTIPLICATION AND ACCUMULATION DETAILS:

We have used the FMULS AVR Hardware multiplier instruction to multiply numbers in the program. For implementing the FIR Filter, we require to multiply the filter coefficients, with the corresponding input sequence terms from the buffer. As the filter coefficients and the input sequence both lie in the range [-1,1], their product also lies in the range [-1,1].

The filter coefficient of the present term is loaded into the R19 register, and the corresponding term from the input sequence in the R18 register. FMULS instruction multiplies these term and stores the result in R0:R1 (16 bit result of multiplication of two 8 bit numbers). FMULS result shifts the result such that the R1 register stores the result in Q1.7 format, while the R0 register just adds on to the accuracy. The result of the multiplication is added into the accumulator and the next multiplication is performed.

ACCUMULATOR AND OVERFLOW:

The accumulator is used to add the result of the multiplication for the different taps, i.e., for 32 TAP filter, the accumulator should store the sum of 32 products. To prevent overflow, we normalise the filter coefficients such that their absolute sum is 1.

$$\sum |h[i]| = 1$$

When this absolute sum is not 1, we scale down all the filter coefficients by the absolute sum, and then convert them into the equivalent of the Q1.7 format.

The result to be stored in the accumulator is

$$\sum x[n-i]h[i]$$

As $x[n]$ belongs to $[-1,1]$ for all n, and $\sum |h[i]| = 1$, the result to be stored in the accumulator will also lie in the range $[-1,1]$. So, we would require only a 16bit accumulator (R20:R21) to store the result of the multiply accumulate operations, in the Q1.7 format in the higher byte, and the lower byte adds on to the accuracy.

After accumulating all the multiplications, we store the higher byte (in R21) into the SRAM as the 8 bit output (part of the output sequence), in the Q1.7 format.

<>

<>

RESULTS

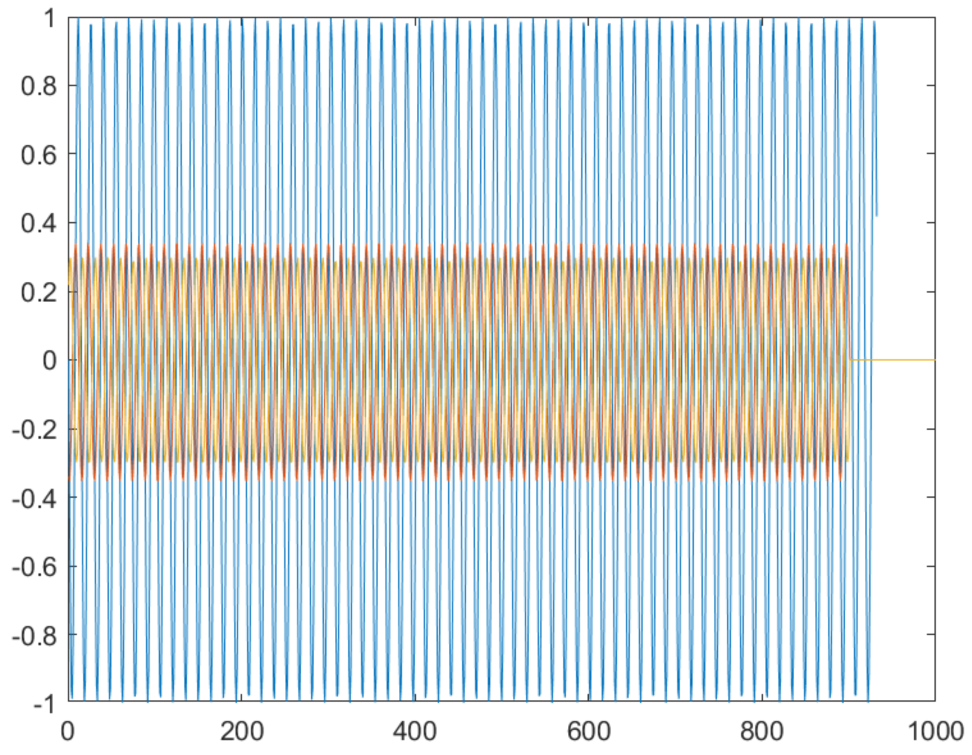
Input and output in the same graph :

BLUE : INPUT

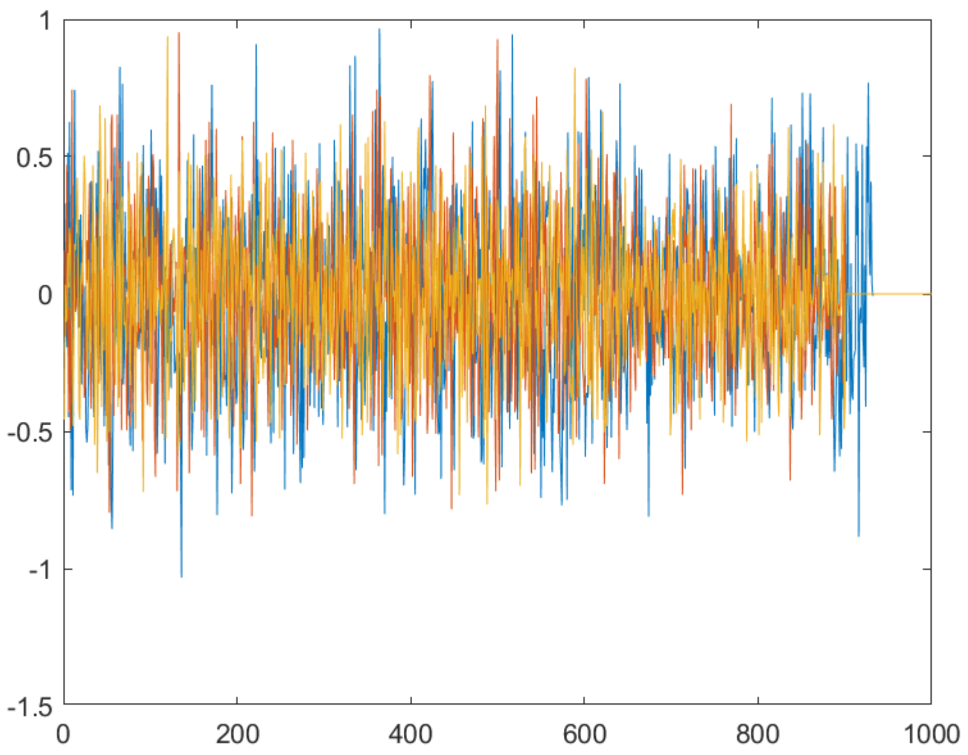
ORANGE : 5 TAP OUTPUT

RED : 32 TAP OUTPUT

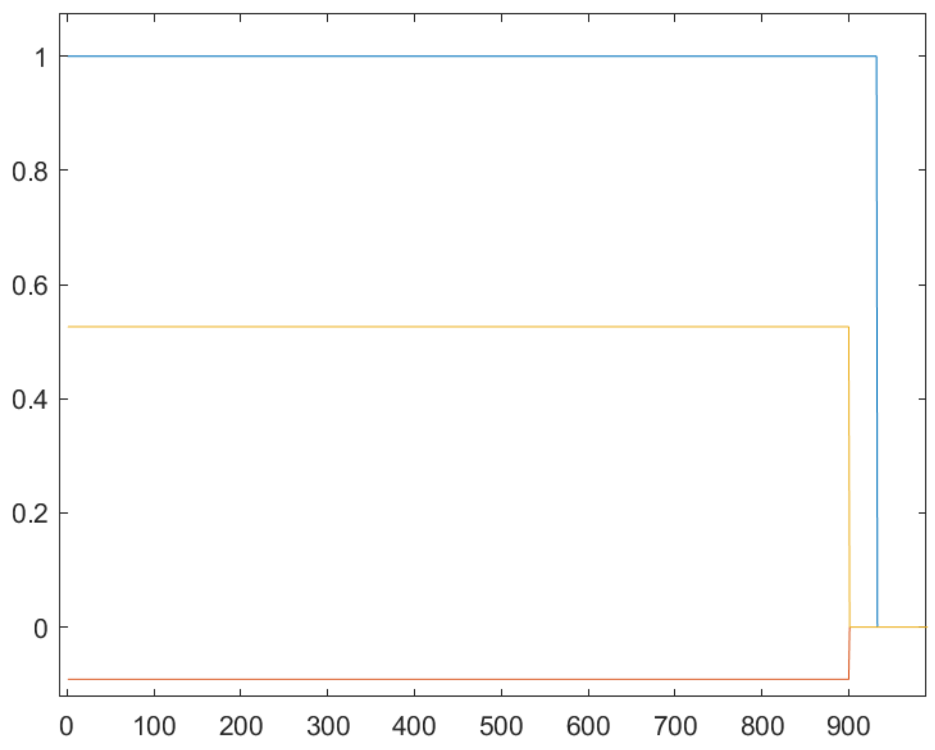
a) Sinusoid of frequency 1800 Hz



b) White Noise



c) DC Input

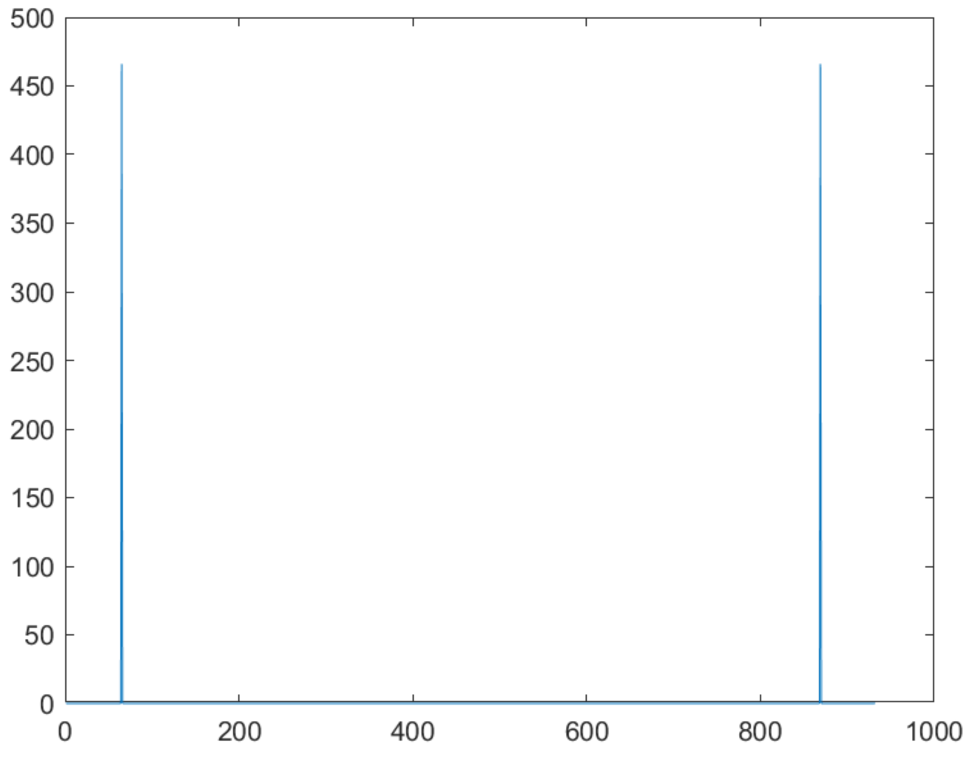


1
0.8
0.6
0.4
0.2
0
0
100
200
300
400
500
600
700
800
900

Frequency Domain Representation:

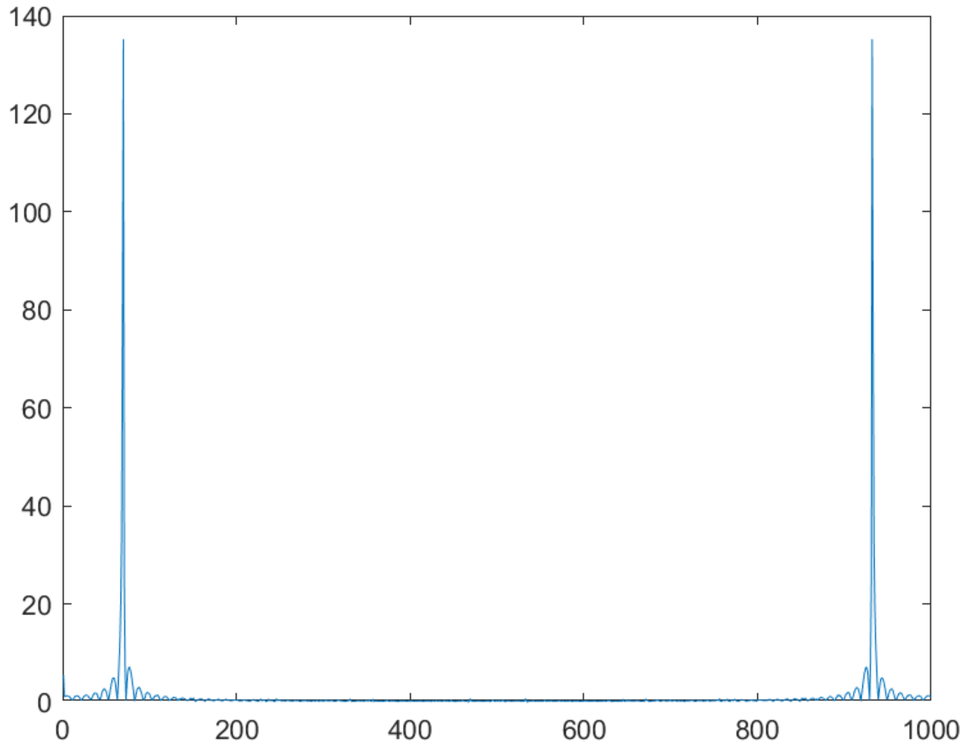
a) Sinusoid

INPUT:

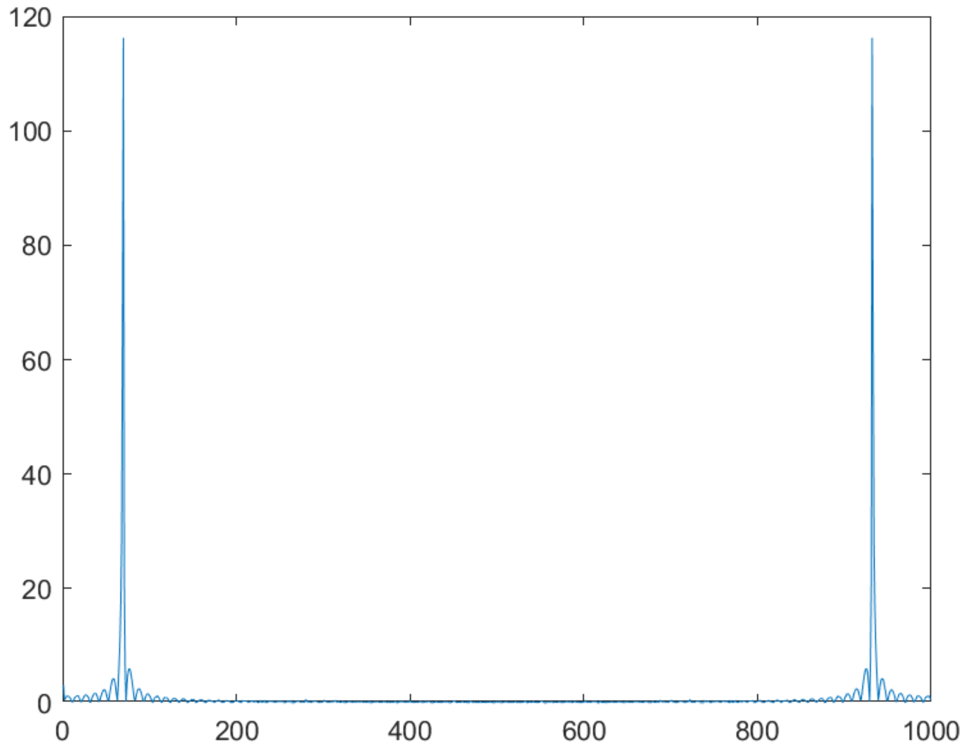


OUTPUT:

5 TAP –

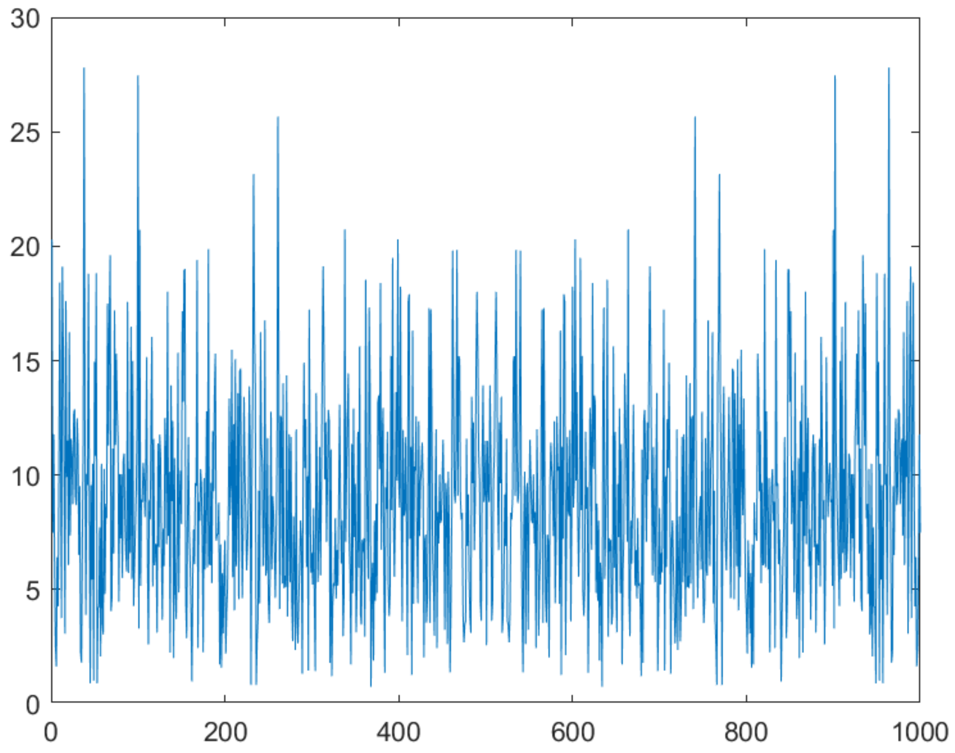

140
120
100
80
60
40
20
0
0
200
400
600
800
1000

32 TAP –


120
100
80
60
40
20
0
0
200
400
600
800
1000

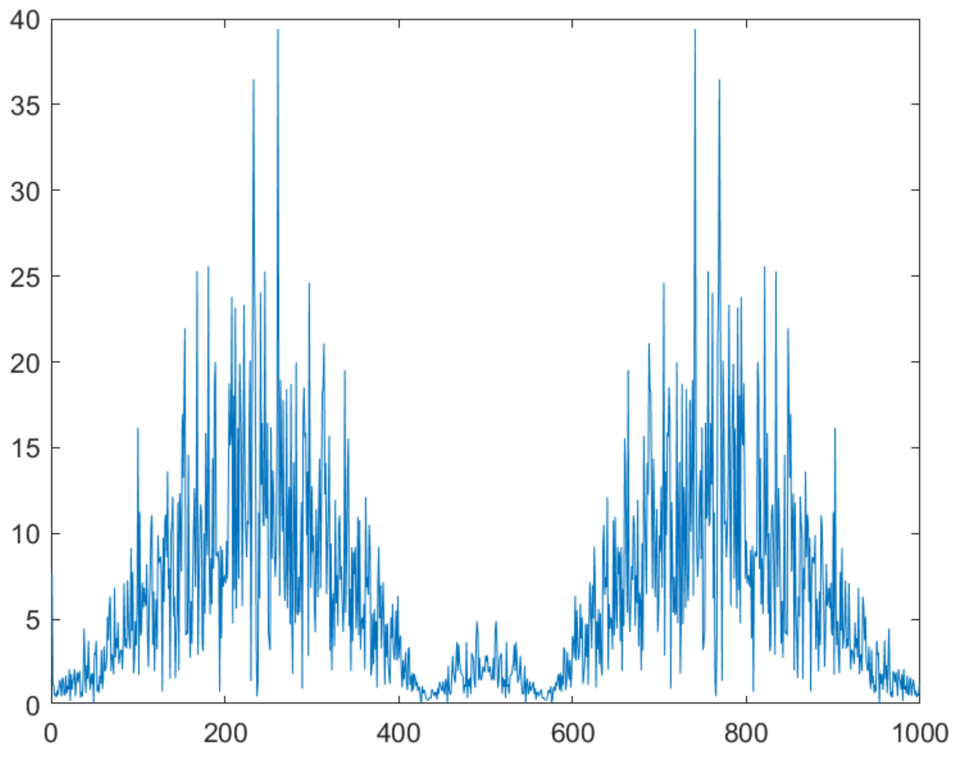
b) White Noise

INPUT:



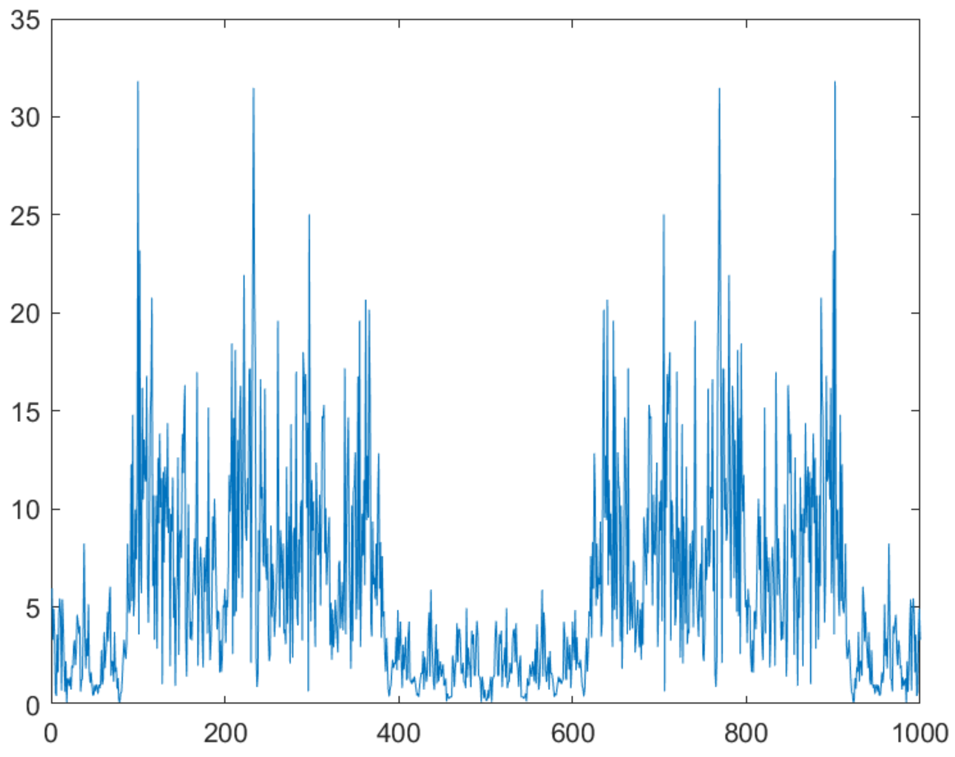
OUTPUT:

5 TAP –



40
35
30
25
20
15
10
5
0
0
200
400
600
800
1000

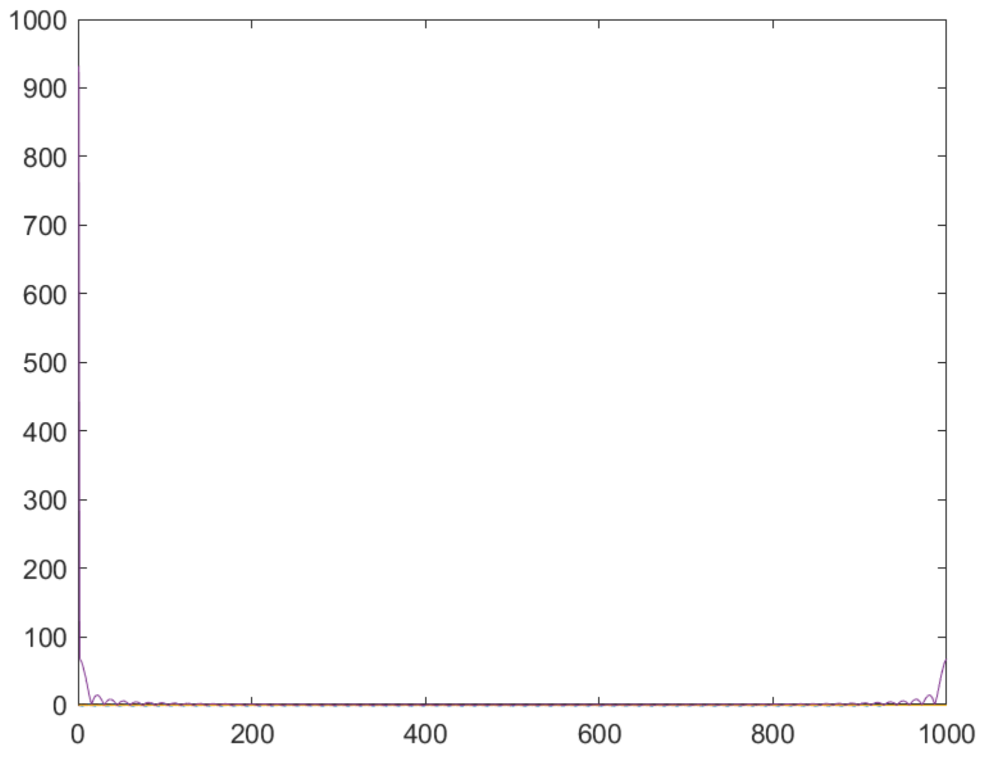
32 TAP –



35
30
25
20
15
10
5
0
0
200
400
600
800
1000

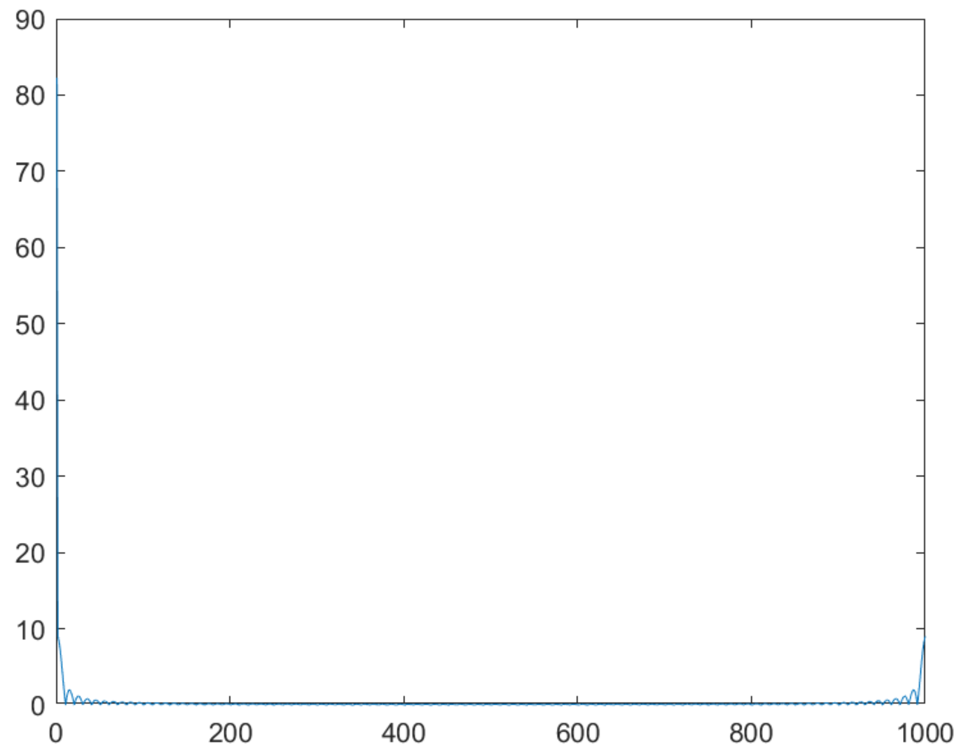
c) DC INPUT

INPUT:



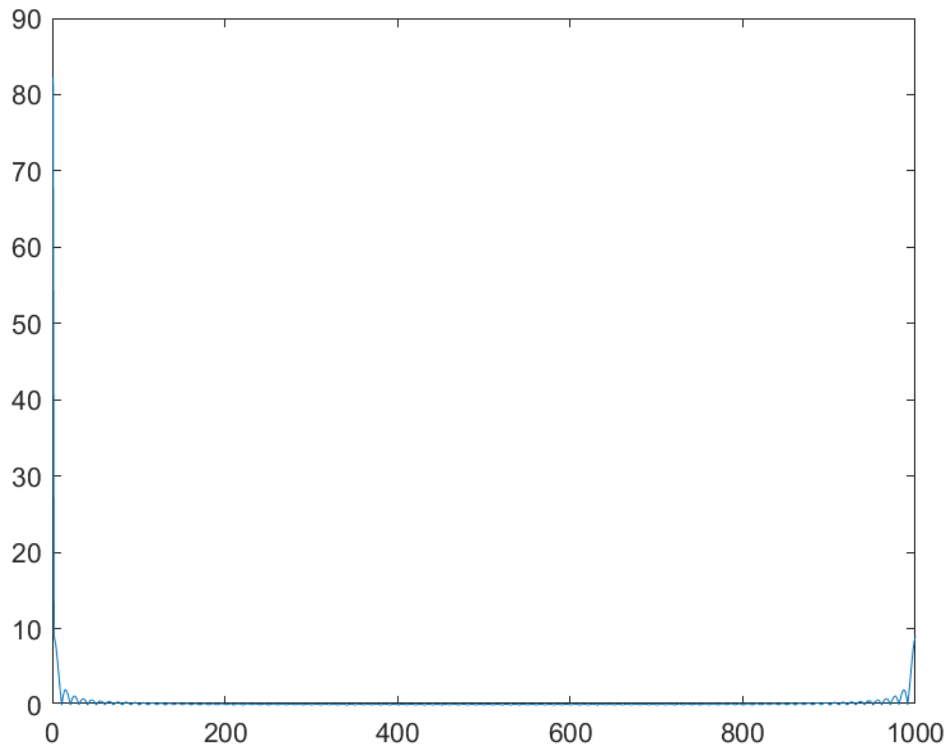
OUTPUT:

5 TAP –



90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000

32 TAP –



90
80
70
60
50
40
30
20
10
0
0
200
400
600
800
1000

xxx