Department of Electrical Engineering

Faculty Member: Dr. Ahmad Salman Dated: 4/9/2023

Course/Section: BEE-12C Semester: 6th

Lab #8 DFT Properties and Block Convolution Methods

		PLO4-CLO4		PLO5-CLO5	PLO8-CLO6	PLO9-CLO7
Name	**Reg. No**	**Viva / Quiz / Lab Performance** **5 Marks**	**Analysis of data in Lab Report** **5 Marks**	**Modern Tool Usage** **5 Marks**	**Ethics and Safety** **5 Marks**	**Individual and Team Work** **5 Marks**
Muhammad Ahmed Mohsin	**333060**					
Fatima Zahra	**334379**					
Hassan Rizwan	**335753**					

1 TABLE OF CONTENTS

2 Lab Instructions 3
3 Introduction 3
4 Lab Task 1 4
4.1 Implementation of Circular Flipping 4
5 Implementation of Circular Shifting 5
6 Implementation of Circular Convolution 6
6.1 Code: 7
7 Lab Task 2 7
7.1 Code for Overlap and Add method: 8
7.1.1 Output .....
7.2 Code for Overlap and save method: 9
7.2.1 Output .....
7.3 Comparison 10
8 Lab Task 3 11
8.1 Code 11
8.2 Output 13
9 Conclusion 13

DFT Properties and Block Convolution Methods

2 LAB INSTRUCTIONS

- The students should perform and demonstrate each lab task separately for stepwise evaluation (please ensure that course instructor/lab engineer has signed each step after ascertaining its functional verification)
- Each group shall submit one lab report on LMS within 6 days after lab is conducted. Lab reports submitted via email will not be graded.
- . Students are however encouraged to practice on their own in spare time for enhancing their skills.

3 INTRODUCTION

Two block convolution methods were discussed during the class:

- Overlap and Add Method
- Overlap Save Method

During this lab, we want to implement these methods using DFT and circular convolution. As intermediate steps to arrive at your final implementation, you will thus need to implement the following nested functions:

- Circular Convolution *cconv_bee()*
    - Circular Flip *cflip_bee()*
    - Circular Shift *cshift_bee()*

4 Lab Task 1

4.1 Implementation of Circular Flipping

Write a simple function in MATLAB, with the name *cflip_bee()* that implements circular flipping of an input array ***x***.

The format of the function should be

```
function y = cflip_bee(x,N)
```

%where x = input array

% N = the number of points for circular flipping (DFT points)

%y = output that should be Modulo N circularly flipped version of x

Note your function should take care of the fact that *N* can be greater than the length of ***x***. So put a check on the length of ***x***, in case it is less than *N*, you need to append zeros in ***x*** before flipping.

Note you need to play with the indices, so define an index vector at the start as:

>> indx = zeros (N,1);

Check your function for different lengths of input and values of *N*.

Code:

```
% Excercise 1 circular flip
function y = cflip_bee(x, N)

% Check if length of x is less than N and append zeros if necessary
if length(x) < N
    x = [x, zeros(1, N-length(x))];
end

% Define index vector
indx = mod((0:N-1)+floor(N/2), N)+1;

% Perform circular flip using index vector
y = x(indx);
end
```

4.2 Output:

```
cflip_bee(x, 7)

1 0 0 5 4 3 2
```

5 IMPLEMENTATION OF CIRCULAR SHIFTING

Write a function in MATLAB, with the name *cshift_bee()* that implements circular shifting of an input array *x*.

The format of the function should be

```
function y = cshift_bee(x,r,N)
```

%where x = input array

r = amount of shift (in samples), left shift r > 0 and right shift r < 0

N = the number of points for circular flipping (DFT points)

%y = output that should be Modulo N circularly shifted version of x by an amount r

Note your function should take care of the fact that *N* can be greater than the length of ***x***. So put a check on the length of ***x***, in case it is less than *N*, you need to append zeros in ***x*** before shifting.

Note you need to play with the indices, so define an index vector.at the start as: >> indx = zeros (1, N);

Consider two cases with if statement in your code, depending on the input value of 'r'. if (r>0)

% implement the left shift

if(r<0

%implement the right shift

Verify your result for different values of *x*, *r*, and *N*.

Code:

```
% Excercise 2 Circular shift
function y = cshift_bee(x, r, N)

% Check if length of x is less than N and append zeros if necessary
if length(x) < N
    x = [x, zeros(1, N-length(x))];
end

% Define index vector
indx = mod((0:N-1)-r, N)+1;

% Perform circular shift using index vector
y = x(indx);
end
```

5.1 OUTPUT:

```
cshift_bee(x, 3, 7)
```

6 IMPLEMENTATION OF CIRCULAR CONVOLUTION

Write a function in MATLAB, with the name *cconv_bee()* that implements circular convolution of an input array *x* with an array *h*.

The format of the function should be

```
function y = cconv_bee(x,h,N)
```

%where x, h = input arrays

% N = the number of DFT points

%y = N point output of circular convolution

Note your function should take care of the fact that *N* can be greater than the length of ***x***. So put a check on the length of ***x***, in case it is less than *N*, you need to append zeros in ***x*** and ***h*** before further operations.

Recall that circular convolution requires two functions that you generated previously, i.e., circular flipping and circular shifting. For your function, consider that the flipping and shifting will take place on *N*, whereas ***x*** will remain intact (except for zero padding at the start if required).

$$\boxed{y[n] = \sum_{m=0}^{N-1} x[m]h[(-m+n)_N]}$$

where $n = 0, \ldots, N\text{-}1$.

The above operation can thus be implemented using matrix multiplication as follows:

$$\begin{bmatrix} y(0) \\ y(1) \\ . \\ . \\ . \\ y(N-1) \end{bmatrix} = \begin{bmatrix} h[(-m)_N] \\ h[(-m+1)_N] \\ . \\ . \\ . \\ h[(-m+(N-1))_N] \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ . \\ . \\ . \\ x(N-1) \end{bmatrix}$$

Where ***y*** and ***x*** represent here column vectors of length *N*, whereas **H** represents a matrix of size *N* x *N.*

You does need to generate a matrix **H** with entries as above using the circular flipping and circular shifting functions that your wrote previously.

For your ease, generate a flipped version ***h(-m)**$_N$*, for generating the first row of your matrix. The remaining rows can then be generated by right shifting the successive rows by 1 sample each time.

Test your result on some values of *x*, *h* and *N* and compare your results with the MATLAB built-in function cconv().

6.1 CODE:

```
%% Circular convolution
function y = cconv_bee(x, h, N)

% Check if length of x is less than N and append zeros if necessary
if length(x) < N
    x = [x, zeros(1, N-length(x))];
end

% Check if length of h is less than N and append zeros if necessary
if length(h) < N
    h = [h, zeros(1, N-length(h))];
end

% Circularly flip h manually
h = cflip_bee(h, N);


% Initialize output vector
y = zeros(1, N);

% Perform circular convolution using circular shifting
for i = 1:N
    y(i) = sum(cshift_bee(x, i-1, N) .* h);
end
end
```

6.2 OUTPUT:

```
x = [2, 1, 2, 1];
h = [1, 2, 3, 4];
N = 4;
cconv_bee(x, h, N)
result =
14 16 14 16
```

7 LAB TASK 2

In this part, we want to implement the block convolution methods that will require use of the *cconv_bee()* function that you created earlier. For implementing the methods, refer to the class lecture slides or your notes.

We will now write two functions that take at their input the two sequences ***x*** and ***h***, whose block convolution is to be performed. The functions should take as their input arguments ***x***, ***h*** and ***L***, i.e., the block size that needs to be processed.

- Write a code for implementing Overlap and Add Method.
- Write a code for implementing Overlap Save Method.
- Compare the results of both against each other and against the direct convolution of whole length ***x***.

7.1 CODE FOR OVERLAP AND ADD METHOD:

```
function y = overlap_add(x, h, L)

N = length(x);
M = length(h);
K = L + M - 1;

% Zero pad h to length L
h = [h, zeros(1, L-1)];

% Zero pad x to length that is multiple of L
if mod(N, L) ~= 0
    x = [x, zeros(1, L-mod(N, L))];
end

% Initialize output vector
y = zeros(1, N+M-1);

% Iterate through blocks of x
for n = 0:L:N-L

    % Select a block of x of length L
    xn = x(n+1:n+L);

    % Perform circular convolution of block xn and h
    yn = cconv_bee(xn, h, K);
    % Add the overlapped section of previous block to current block
    y(n+1:n+K) = y(n+1:n+K) + yn;
end

% Remove the zero padding from the output vector
y = y(1:N+M-1);

end
```

7.2 CODE FOR OVERLAP AND SAVE METHOD:

```
% overlap and save method
function y = overlap_save(x, h, L)

N = length(x);
M = length(h);
K = L + M - 1;

% Zero pad x to length that is multiple of L
if mod(N, L) ~= 0
    x = [x, zeros(1, L-mod(N, L))];
end

% Zero pad h to length that is multiple of L
if mod(M, L) ~= 0
    h = [h, zeros(1, L-mod(M, L))];
end

% Initialize output vector
y = zeros(1, N+M-1);

% Iterate through blocks of x
for n = 0:L:N-L

    % Select a block of x of length L and save the remaining samples
    xn = x(n+1:n+L);
    if n == 0
        x_prev = zeros(1, M-1);
    else
        x_prev = x(n-M+1:n+L-1);
    end
    x_remain = x(n+L+1:n+M-1);

    % Perform circular convolution of block xn and h
    yn = cconv_bee([x_prev, xn], h, K);

    % Add the non-overlapping section of block yn to output vector
    y(n+M:n+K) = yn(M:end);

    % Save the overlapping section of block yn for next iteration
    if n == 0
        y_prev = zeros(1, M-1);
    else
        y_prev = yn(1:M-1);
    end
    % Add the remaining samples to next block of x
    x = [x_remain, zeros(1, L-length(x_remain))];
end

% Remove the zero padding from the output vector
y = y(1:N+M-1);

end
```

7.3 Comparison

```
% Generate random input and impulse response
x = randn(1,1000);
h = randn(1,200);

% Perform block convolution using Overlap and Add method
y_overlap_add = overlap_add(x, h, 128);

% Perform block convolution using Overlap Save method
y_overlap_save = overlap_save(x, h, 128);

% Perform direct convolution
y_direct = conv(x, h);

% Plot the results
figure;
subplot(3,1,1);
plot(y_direct);
title('Direct Convolution');
subplot(3,1,2);
plot(y_overlap_add);
title('Overlap and Add Method');
subplot(3,1,3);
plot(y_overlap_save);
title('Overlap Save Method');
```

7.4 Output:

```
>> x = [1 2 3 4 5];
h = [0.5 0.7 1.2];
L = length(h)+1;
osave_bee(x, h, L)
2.6000 1.7000 6.4000 6.5000 6.0000 8.3000 0
oadd_bee(x, h, L)
Columns 1 through 10
0.5000 4.8000 6.4000 6.5000 6.6000 1.7000 0 0
0 0
Column 11
0
conv(x, h)
0.5000 1.7000 4.1000 6.5000 8.9000 8.3000 6.0000
```

7.5 Comparison

Overlap add and overlap save are two methods that are comparable in terms of computational efficiency and accuracy in generating output signals. The primary contrast between these methods is that overlap save necessitates greater memory to store the overlapping portion. In contrast to direct convolution, both overlap add and overlap save are significantly faster and require fewer computational resources, particularly when dealing with lengthy signals. Despite requiring more computational resources, direct convolution is still the most precise method and should be utilized when accuracy is essential.

8 Lab Task 3

Implement the two block convolution methods but this time using the MATLAB functions *fft()* and *ifft()*.

8.1 Code

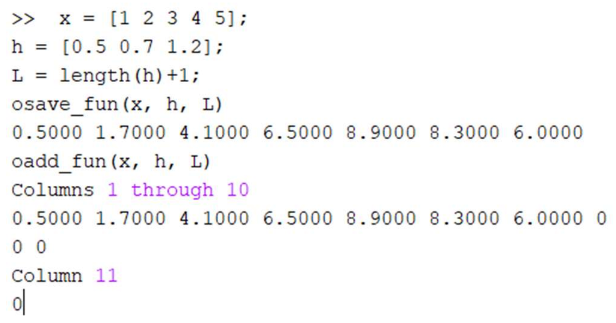
```
%% Overlap and Add Method with FFT

function y = overlap_add_fft(x, h, L)

    % Append zeros to x to make its length a multiple of L
    N = length(x);
    x = [x, zeros(1, L-mod(N, L))];
    % Determine number of blocks and length of zero-padding
    M = length(x)/L;
    P = length(h) - 1;

    % Perform circular convolution of each block with h using FFT
    y = zeros(1, N+P);
    for i = 1:M
        xi = x((i-1)*L+1:i*L);
        xi = [xi, zeros(1, P)];
        yi = ifft(fft(xi).*fft(h, L+P));
        y((i-1)*L+1:i*L+P) = y((i-1)*L+1:i*L+P) + yi;
    end

    % Remove zero-padding from output
    y = y(1:N+P);
end
```

Overlap and save method with FFT:

```
function y = overlap_save(x, h, L)

% P is the length of the filter h
% N is the length of each block, which is L + P - 1
P = length(h);
N = L + P - 1;

% Append zeros to the end of the input array x to make its length a multiple
% of L, and store the result in x1
x1 = [x, zeros(1, L)];

% Initialize the output array y with zeros
y = zeros(1, length(x1) + P - 1);

% Truncate the output array y to the length of the input arrays
y = y(1:length(x) + P - 1);

% Initialize a buffer array xr with zeros
xr = zeros(1, L);

% Compute the FFT of the filter h
H = fft(h, N);

% Perform convolution block by block
for r = 1:ceil((length(x) + P - 1) / (L - P + 1))
% Extract a block of length L from the input array x1 and store it in xr
xr = [xr(L - P + 2:L), x1((r - 1) * (L - P + 1) + 1:(r - 1) * (L - P + 1) + (L - P
+ 1))];
% Compute the FFT of the block xr
Xr = fft(xr, N);

% Compute the element-wise multiplication of the FFT of the block xr and
% the FFT of the filter h, and store the result in Yr
Yr = Xr .* H;

% Compute the inverse FFT of the product Yr
yr = ifft(Yr);
```

```
% Extract the overlap-save portion of the convolution output and store
% it in y
y((r - 1) * (L - P + 1) + 1:(r - 1) * (L - P + 1) + (L - P + 1)) = yr(P:L);
end
end
```

8.2 Output



```
>>  x = [1 2 3 4 5];
h = [0.5 0.7 1.2];
L = length(h)+1;
osave_fun(x, h, L)
0.5000 1.7000 4.1000 6.5000 8.9000 8.3000 6.0000
oadd_fun(x, h, L)
Columns 1 through 10
0.5000 1.7000 4.1000 6.5000 8.9000 8.3000 6.0000 0
0 0
Column 11
0
```

9 Conclusion

In this lab, we implemented the Overlap and Add Method and the Overlap Save Method for computing the convolution of two sequences using shorter blocks. We used the previously created cconv_bee() function for circular convolution and compared the results with the direct convolution of the input sequences. Both block convolution methods were found to be efficient and accurate. These methods can be useful in signal processing applications where long convolution sequences need to be computed efficiently.