Image Processing

Extracting color components

Any color image is considered a 3-dimensional array, the first two dimensions represent the size of the image while the third one states RGB components. To extract the intensity values of each color component from the original image this line of code is used to loop on each pixel getting only the n^{th} value.

```
Color = img ( : , : , n ) ;
```

- n = 1 ➔ extracted color is red
- n = 2 ➔ extracted color is green
- n = 3 ➔ extracted color is blue

now we managed to extract each color component as a 2-dimensional array of the intensity value of this color in each pixel. To display a color component, we must concatenate it with two zero-matrices of the same size.

Conv2 () function

```
Result-image = conv2(original-image, Kernel, 'same') /256 ;
```

It is a function that performs 2D convolution where 'Result-image' equals 'original-image' convolved with 'kernel'. Convolution always produces an array that's bigger than the two input arrays so, the 'same' operation is used to make the result of the same size as the original image. Then each element in the result of the convolution is divided by 256 to normalize its value, normalization means scaling the values after convolution to prevent them from becoming too large.

After achieving the appropriate kernel it is then divided by 256 for the same reason, to normalize the result and prevent it from being too large.

Edge detection kernel: The Laplacian kernel

-1	-1	-1
-1	8	-1
-1	-1	-1

It is designed to detect regions in an image where there is a rapid change in pixel intensity, which typically occurs at edges. The central element (8) in the kernel gives higher weight to the pixel in the center, while the surrounding weights (-1) emphasize the differences between neighboring pixels. Applying this kernel to an image using convolution enhances the contrast at edges. The result is an image where edges are highlighted.

Sharpening kernel

-1	-1	-1
-1	9	-1
-1	-1	-1

It's similar to the Laplacian kernel but with a slight difference. The central element now has a weight of 9, making it more pronounced.

This matrix is good for making pictures look sharper because it boosts the importance of the central pixel, making edges and details stand out more. The increased value in the center (9) makes the image appear crisper by focusing on the differences between the central pixel and its surroundings. This helps enhance small details and improves the overall sharpness of the image.

Blurring kernel

```
blurring_Kernel = ones (10, 10) / 100;
```

This line of code represents a 10x10 matrix filled with ones and then divides each element by 100. The resulting matrix represents a simple average filter. When this kernel is used in convolution with an image, it has a blurring effect. Each pixel in the image gets replaced by the average value of its neighboring pixels, making the overall image blurred.

Motion blurring kernel

```
motion_blurring_Kernel = ones (1, 25) / 25;
```

This kernel is a row vector of length 25 filled with ones and then each element is divided by 25. When convolving this kernel with an image, each pixel in the output is an average of itself and its neighboring pixels to the left and right. This mimics the effect of a moving object or camera in the horizontal direction.

We won't use `'same'` operation in conv2() function because we need the full size after the convolution to restore the image properly. So, the motion-blurred image will be resized before displaying it. When the edges of the image are convolved to the kernel, the kernel extends beyond the image boundary, leading to additional pixels in the motion-blurred image. The Excess pixels will only appear on the left and right edges forming excess columns, this is because the kernel is a row victor. So, these excess columns must be removed to retrieve the original size.

Excess columns due to convolution on each side =
[no. columns of motion-blurred image - no. columns of original image] /2

```
Resized motion-blurred image = motion-blurred image(:,
(excess-columns +1):(original_width + excess-columns), :);
```

This line of code removes the excess columns at the lift and right edge by selecting a submatrix of the `motion-blurred image`.

- (**:** , ... , **:**) ➔This colon indexing is used to select all rows and all color components.
- (excess_columns + 1) : (original_size(2) + excess_columns)➔ This range of columns starts just after the excess columns at the lift edge and goes up to the column before the excess columns at the right edge.

Restoration

From Convolution property of CT Fourier Transform we know that the convolution in time domain is equivalent to multiplication in frequency domain therefore, by dividing the motion-blurred image by the kernel in frequency domain we can obtain the original image in frequency domain. Then ifft2() is used to get the restored image in the space domain.

While performing fft2() to the motion-blurring kernel the dimensions of the motion-blurred image are passed to ensure that the kernel is transformed to the same size. This step is crucial for the next step, where the Fast Fourier Transform (FFT) of the motion-blurred image is divided by the FFT of the motion-blurring kernel.

The operator "**./**" is used to divide two matrices of the same size, each element in the first matrix is divided by the corresponding element in the second matrix.

Due to size mismatch between the motion-blurred image and the original image, the restored image also needs resizing. The restored image will have additional columns of black pixels at the right edge so, these excess columns must be removed to retrieve the original size.

```
Resized restored image= restored (:, 1:original_width , :);
```

This line of code removes the excess black columns at the right edge just like how we resized the motion-blurred image before displaying it.

Audio

Recording

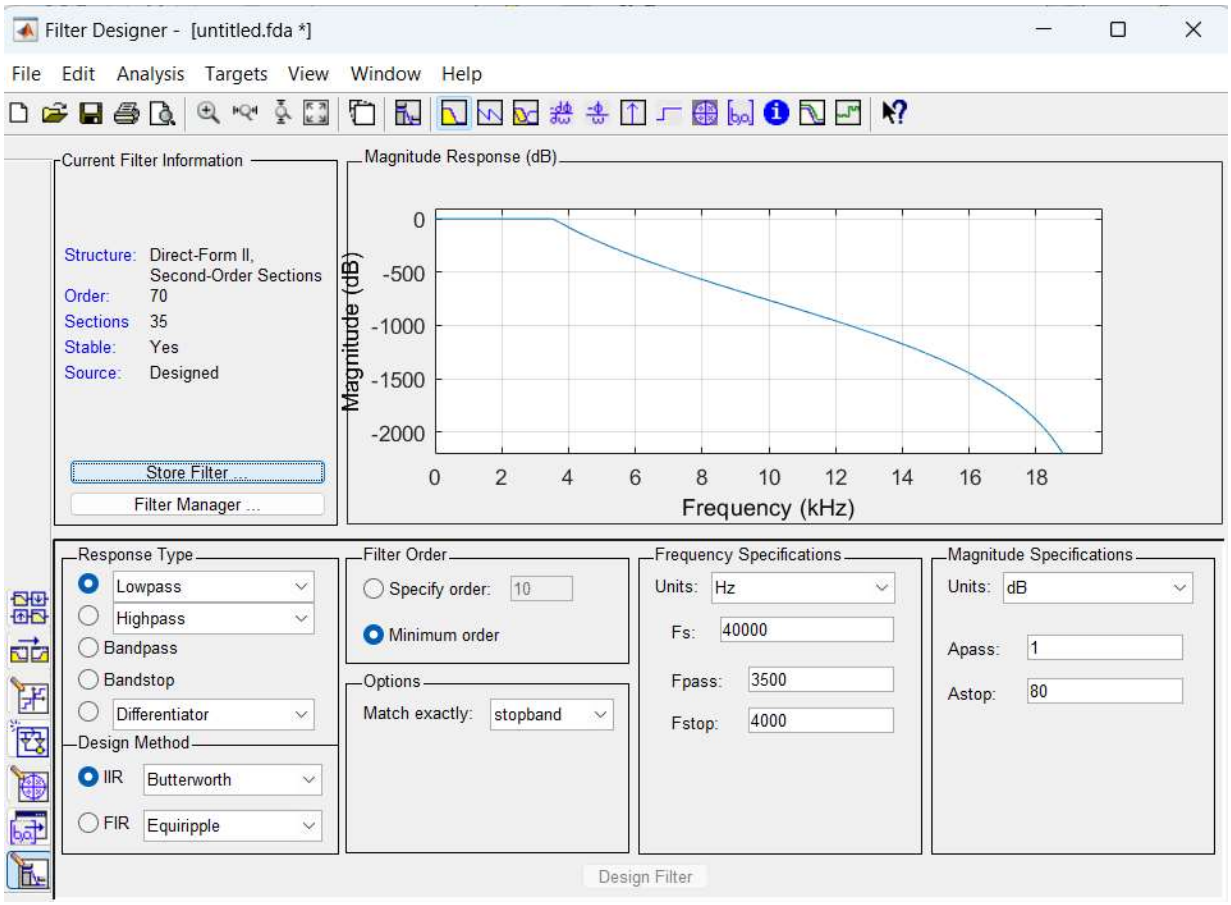
To choose the sampling frequency for recording, different frequencies were tested until reaching a suitable frequency which is 40000 Hz.

The bit depth is set to 16 bit/sample as stated in the lecture to be the suitable value to reach a clear recording.

Using the " audiorecorder( ) " function from the matlab help center [1], two records were recorded as required, then saving them as "input1" and "input2" using " getaudiodata( ) " and " audiowrite( ) " from the matlab help center[2].

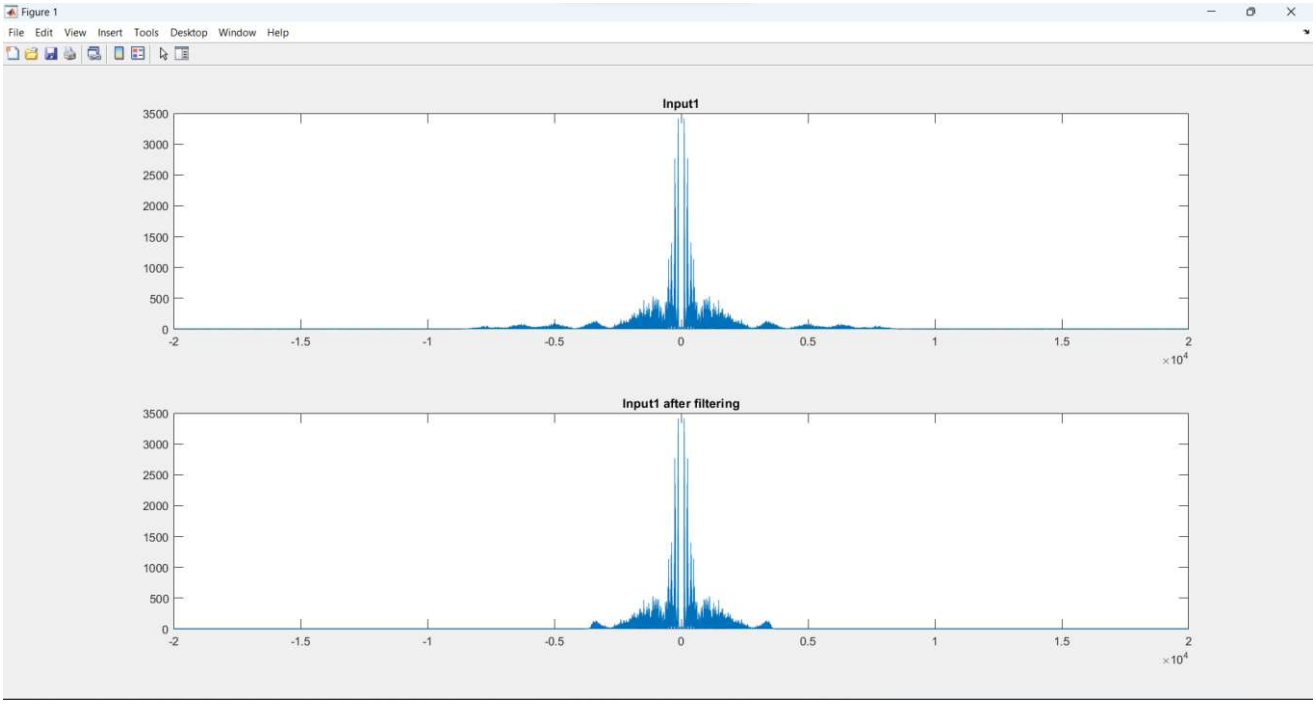
Filtering

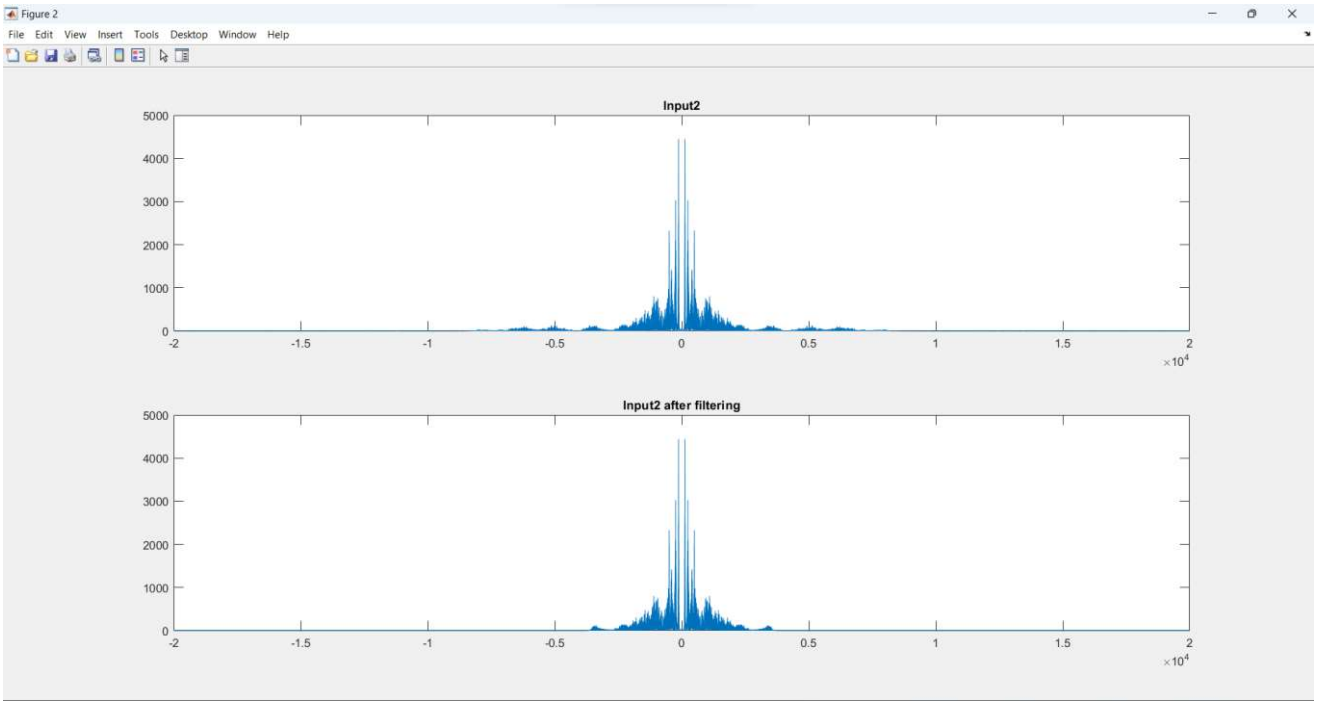
The two inputs were filtered using " filterDesigner " tool



Plotting

Relying on the project video the inputs were plotted using " plot( ) " after transforming it to frequency domain using " fft( ) ", identifying 'n' which is the number of samples and ' f ' as the x-axis giving it a suitable range.





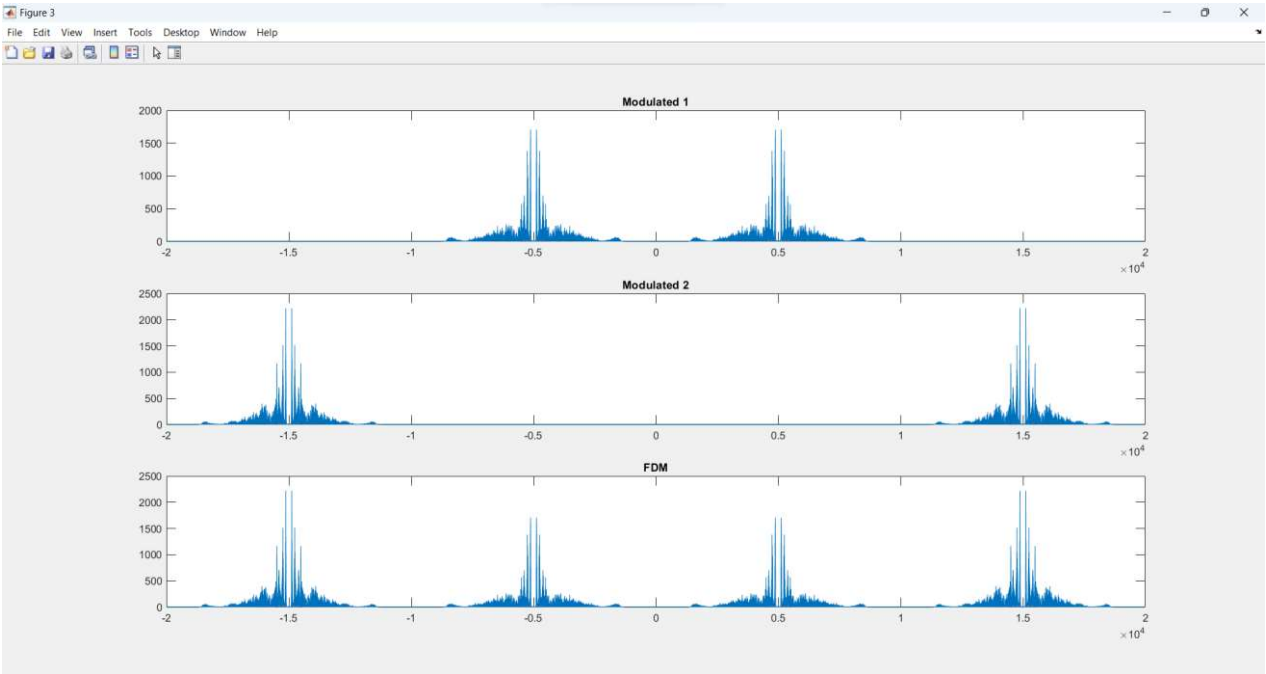
Modulation

First choosing the time ‘t’ for carrier signals is defined from the sample rate and sampling frequency [3], then choosing suitable carrier frequencies where

(f_carrier1 > max. frequency for first filtered input)

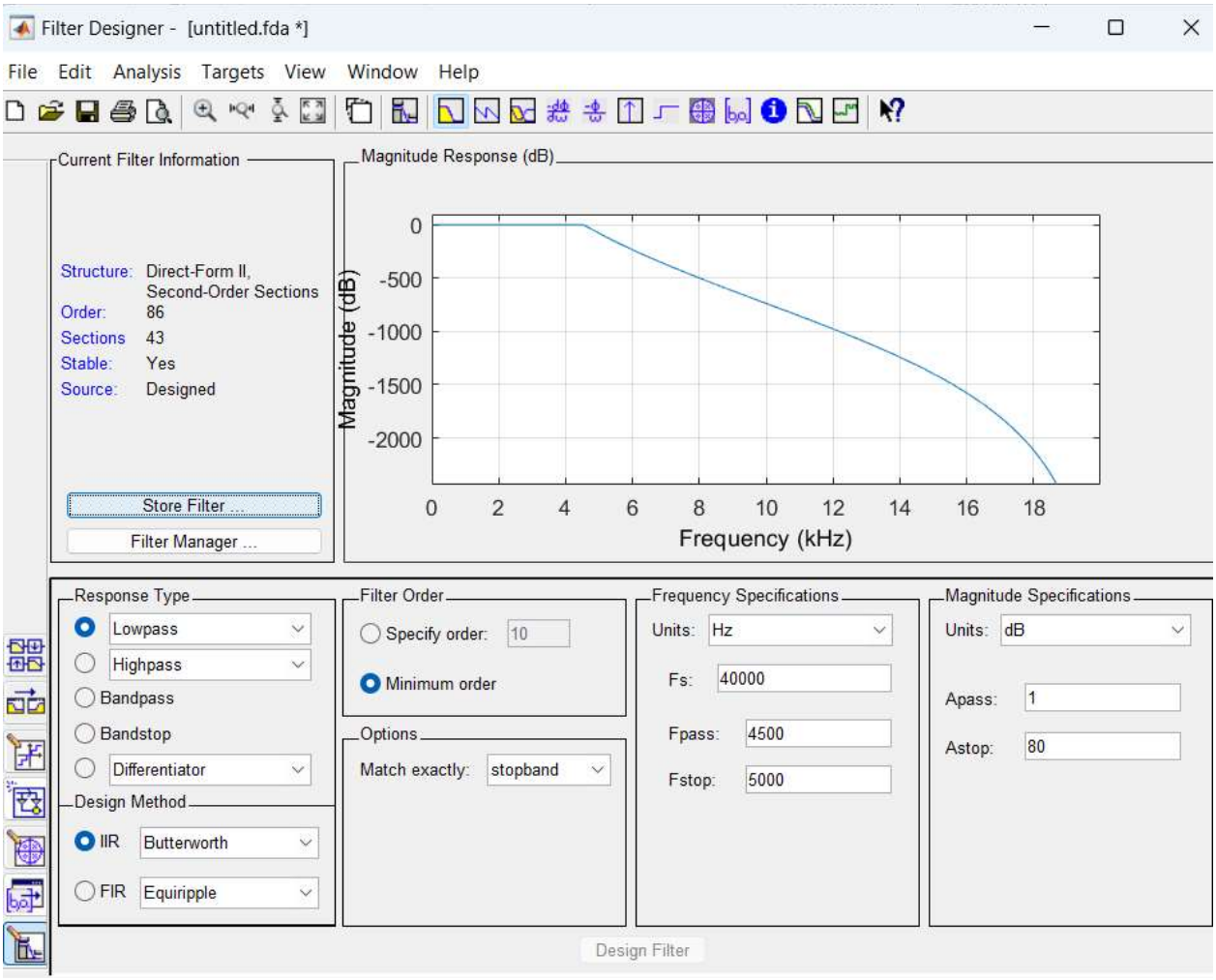
(f_carrier2 > (max. f for 2nd filtered input + max. f for second filtered input + f_carrier1))

After that, the carrier signals were modified by “ transpose( ) ” to perform suitable array multiplication, then adding the 2 modulated signals to get the transmitted signal “ FDM ”

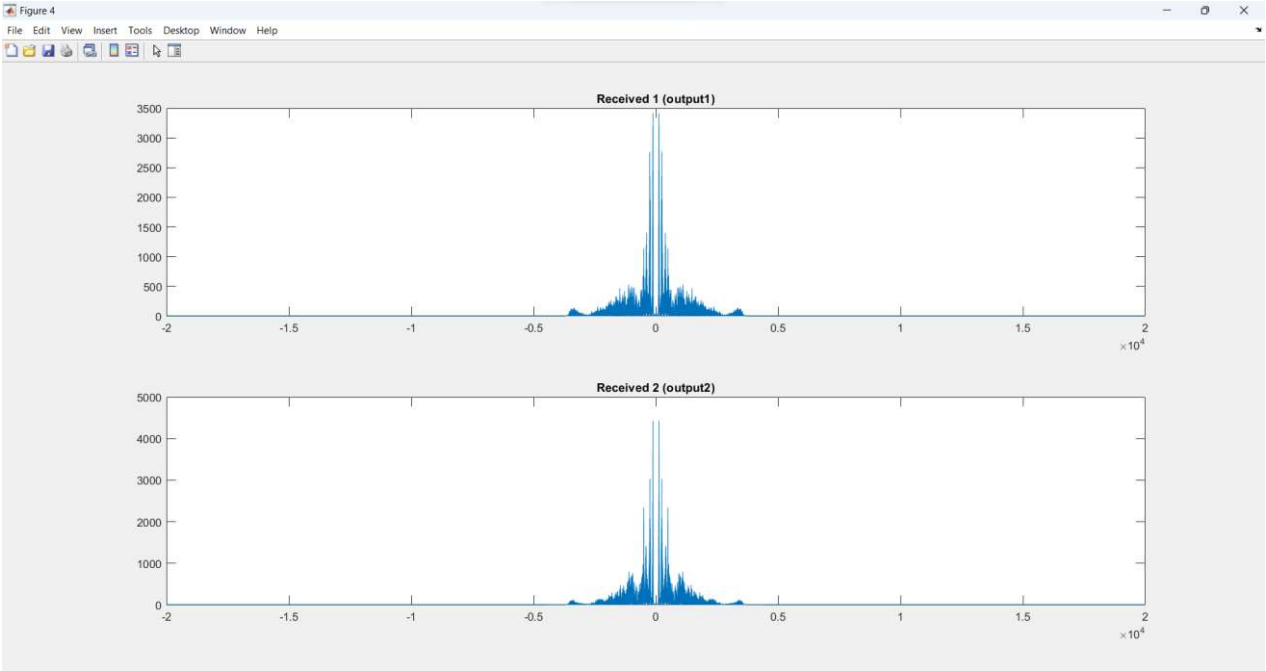


Demodulation

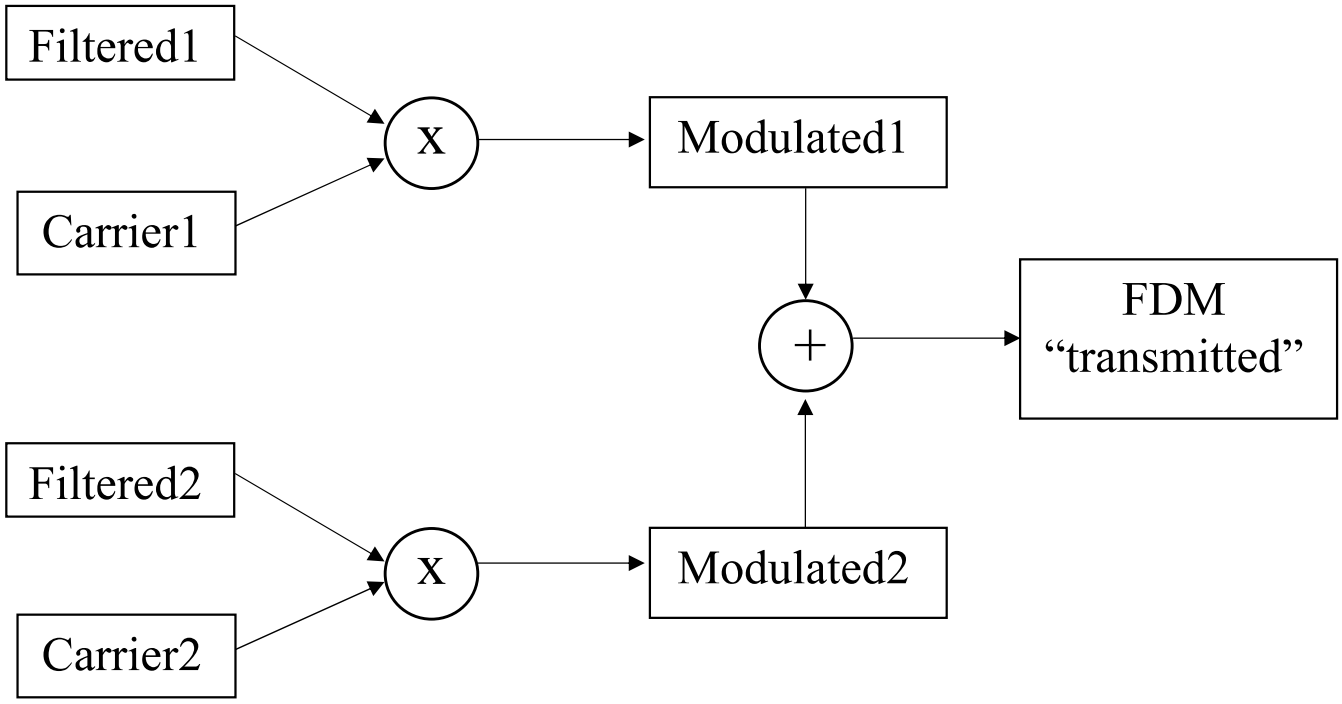
Multiplying the transmitted signal " FDM " again with the carrier signal then filtering the result using a filter with a cutoff frequency w_c where $w_m < w_c < w_a - w_m$ to avoid any accidental removal of data from the signal or having any unwanted signals. Then we multiply the filtered signal by 2 to get the original signal.



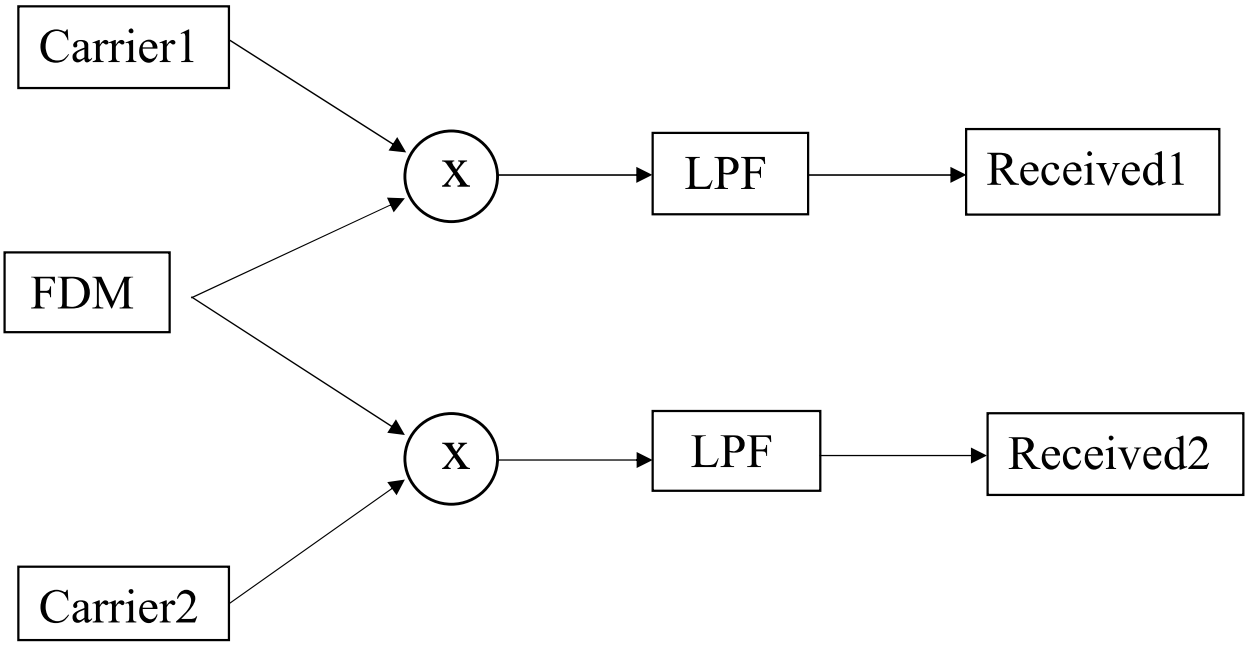
Finally, we save the two outputs as “ output1 ” and “ output2 ” .



Transmitter



Receiver



Receiver explanation

Let:

- x(t) is the received signal with multiple signals at different frequencies.
- Let the first frequency is at (w_a) and carrier signal is cos(w_at)
- w_c is the cutoff frequency, w_m is the max. frequency of the signal at w_a
- z(t) is the original signal

1. x(t).cos(w_at) $\longleftrightarrow$ $(1/2\pi)X(w) * \delta(w)$ $\longrightarrow$ y(t) $\longleftrightarrow$ Y(w)

This operation shifts the received signal, making the signal at w_a and -w_a shift to the center, divide their amplitude by 0.5 and add together so we got the same signal after modulation at the center which is half the original signal.

2. y(t) $\longleftrightarrow$ Y(w) $\rightarrow$ LPF with $w_m < w_c < w_a - w_m$ $\rightarrow$ z(t) $\longleftrightarrow$ Z(w)

In this operation the signal passes a LPF with $w_m < w_c < w_a - w_m$ to ensure that only the signal at the center is extracted and get z(t) which is the original signal.

References

1. https://www.mathworks.com/help/matlab/ref/imread.html
2. https://www.mathworks.com/help/matlab/ref/imshow.html
3. https://www.mathworks.com/help/matlab/ref/cat.html
4. https://youtu.be/KuXjwB4LzSA?si=2Q6cQqcoALYrtMXf
5. https://homepages.inf.ed.ac.uk/rbf/HIPR2/log.htm
6. https://www.globalsino.com/EM/page1371.html
7. https://www.opencv-srf.com/2018/03/homogeneous-blur.html?m=1
8. https://medium.com/@itberrios6/how-to-apply-motion-blur-to-images-75b745e3ef17
9. Record and Play Audio - MATLAB & Simulink (mathworks.com)
10. Record and save audio with your specific file name for the audio. - Help Center - MATLAB & Simulink (mathworks.com)
11. Create Uniform and Nonuniform Time Vectors - MATLAB & Simulink (mathworks.com)

Appendix

Image processing code:

```
% Reading the image file
img = imread("peppers.png");
original_size = size(img);
% original_size(1) = number of rows = the height of the image in pixels
% original_size(2) = number of columns = the width of the image in pixels
% original_size(3) = number of color components = 3 {R, G, B}

% Extracting color components.
red = img(:,:,1);
green = img(:,:,2);
blue = img(:,:,3);

zero_matrix = zeros(size(red));

Red_image= cat(3, red, zero_matrix, zero_matrix);
Green_image = cat(3, zero_matrix, green, zero_matrix);
Blue_image = cat(3, zero_matrix, zero_matrix, blue);

% Display color components
figure;
imshow(img);
title('Original Image');
figure;
subplot(1, 3, 1);
imshow(Red_image);
title('Red component of the Image');
subplot(1, 3, 2);
imshow(Green_image);
title('Green component of the Image');
subplot(1, 3, 3);
imshow(Blue_image);
title('Blue component of the Image');



%%%  Edge detection  %%%
edge_Kernel = [-1 -1 -1; -1 8 -1; -1 -1 -1]/256;  % Laplacian
edges_red = conv2(red, edge_Kernel, 'same') /256;
edges_green = conv2(green, edge_Kernel, 'same') /256;
edges_blue = conv2(blue, edge_Kernel, "same") /256;
final_edge = cat(3, edges_red, edges_green, edges_blue) ;
figure;
imshow(final_edge);
title('Edge-detected Image');
```

```
%%%  Image sharpening  %%%
sharpeningKernel = [-1 -1 -1; -1 9 -1; -1 -1 -1]/256;
sharpened_red = conv2(red, sharpeningKernel, 'same') /256;
sharpened_green = conv2(green, sharpeningKernel, 'same') /256;
sharpened_blue = conv2(blue, sharpeningKernel, 'same') /256;
final_sharpened = cat(3, sharpened_red, sharpened_green, sharpened_blue);
figure;
imshow(final_sharpened);
title('Sharpened Image');



%%%  Blurring  %%%
blurring_Kernel = ones(10, 10) / (100*256);
% a matrix of ones it's order 10*10, divided by the number of elements in the
matrix (average)
blurred_Red = conv2(red, blurring_Kernel, 'same') /256;
blurred_Green = conv2(green, blurring_Kernel, 'same') /256;
blurred_Blue = conv2(blue, blurring_Kernel, 'same') /256;
final_blurred = cat (3, blurred_Red, blurred_Green, blurred_Blue);
figure;
imshow(final_blurred);
title('Blurred Image');


%%%   Motion blurring in the horizontal direction   %%%
motion_blurring_Kernel = ones(1, 25) / (25*256);
motion_blurred_Red = conv2(red, motion_blurring_Kernel) /256;
motion_blurred_Green = conv2(green, motion_blurring_Kernel) /256;
motion_blurred_Blue = conv2(blue, motion_blurring_Kernel) /256;
final_motion_blurred = cat(3, motion_blurred_Red, motion_blurred_Green,
motion_blurred_Blue);
% Resizing the motion-blurred image
excess_columns = ( size(final_motion_blurred, 2) - original_size(2) ) /2 ; %
number of the excess columns on each side.
final_motion_blurred = final_motion_blurred (:, (excess_columns
+1):(original_size(2) + excess_columns), :);
figure;
imshow(final_motion_blurred);
title('Motion-Blurred Image in the horizontal direction');


%%%   Restoring the original image from the motion-blurred image   %%%
FFT_motion_blurred_Red = fft2(motion_blurred_Red) ;
FFT_motion_blurred_Green = fft2(motion_blurred_Green) ;
FFT_motion_blurred_Blue = fft2(motion_blurred_Blue) ;
FFT_motion_bluring_Kernel = fft2( motion_blurring_Kernel,
size(motion_blurred_Red, 1), size(motion_blurred_Red, 2)) *256;
FFT_Red = FFT_motion_blurred_Red ./ FFT_motion_bluring_Kernel;
```

```
FFT_Green = FFT_motion_blurred_Green ./ FFT_motion_bluring_Kernel;
FFT_Blue = FFT_motion_blurred_Blue ./ FFT_motion_bluring_Kernel;
restored_Red = ifft2(FFT_Red);
restored_Green = ifft2(FFT_Green);
restored_Blue = ifft2(FFT_Blue);
final_restored = cat(3, restored_Red, restored_Green, restored_Blue);
% Resizing the restored image
final_restored = final_restored (:, 1:original_size(2) , :);
figure
imshow(final_restored);
title('Restored Image');
```

Audio code:

```
%Recording parameters
bitDepth = 16;
duration = 10;
fs=40000;

%First record
record1 = audiorecorder(fs, bitDepth, 1);
disp('Start record 1');

recordblocking(record1, duration);
disp('End of record 1');

input1 = getaudiodata(record1);
audiowrite('input1.wav', input1, fs);

%Second record
record2 = audiorecorder(fs, bitDepth, 1);
disp('Start record 2');

recordblocking(record2, duration);
disp('End of record 2');

input2 = getaudiodata(record2);
audiowrite('input2.wav', input2, fs);

%Filter input1
filtered1=filter(LowPassFilter_1,input1);
audiowrite('filtered1.wav', filtered1, fs);

%Filter input2
filtered2=filter(LowPassFilter_1,input2);
audiowrite('filtered2.wav', filtered2, fs);
```

```
%Plotting parameters
n=length(input1);         %number of samples
f=(-n/2:n/2-1)*fs/n;      %Define x-axis

%Plot input1
fft_input1=fft(input1);
figure;
subplot(2,1,1);
plot(f, abs(fftshift(fft_input1)));
title('Input1');

%Plot input1 after filtering
fft_filtered1=fft(filtered1);
subplot(2,1,2);
plot(f, abs(fftshift(fft_filtered1)) );
title('Input1 after filtering');

%Plot input2
fft_input2=fft(input2);
figure;
subplot(2,1,1);
plot(f, abs(fftshift(fft_input2)) );
title('Input2');

%Plot input2 after filtering
fft_output2=fft(filtered2);
subplot(2,1,2);
plot(f, abs(fftshift(fft_output2)) );
title('Input2 after filtering');

%Define carrier parameters
t = (0: n-1) / fs;  %Define time for carrier

f_carrier1 = 5000;
f_carrier2 = 15000;

carrier1 = cos(2*pi*f_carrier1*t);
carrier2 = cos(2*pi*f_carrier2*t);

%Modify carrier signals to properly multiply the matrices
carrier1_t = transpose(carrier1);
carrier2_t = transpose(carrier2);

%Amplitude modulation
modulated1 = filtered1 .* carrier1_t;
modulated2 = filtered2 .* carrier2_t;

%Frequency Division Multiplexing
FDM = modulated1 + modulated2;
```

```
%Plotting the modulated and FDM signals
figure;
subplot(3,1,1);
plot(f, abs( fftshift(fft(modulated1)) ) );
title('Modulated 1');

subplot(3,1,2);
plot(f, abs( fftshift(fft(modulated2)) ) );
title('Modulated 2');

subplot(3,1,3);
plot(f, abs( fftshift(fft(FDM)) ) );
title('FDM');

%Demodulation
demodulated1 = FDM .* carrier1_t;
demodulated2 = FDM .* carrier2_t;

output1=2*filter(LowPassFilter_2,demodulated1);
output2=2*filter(LowPassFilter_2,demodulated2);

audiowrite('output1.wav', output1, fs);
audiowrite('output2.wav', output2, fs);

figure;
subplot(2,1,1);
plot(f, abs(fftshift(fft(output1))) );
title("Received 1 (output1)");
subplot(2,1,2);
plot(f, abs(fftshift(fft(output2))) );
title("Received 2 (output2)");
```