HW2 LMS Filter and DWT Filter design

VLSI DSP HW2

Shun-Linag Yeh, NCHU Lab612

3/19/2023

Github Src Code

INDEX

I. Adaptive FIR Low pass filter

Problem

Q1. LMS filter design

For a least mean square (LMS) adaptive filter, assume the filter is of the form finite impulse response (FIR) and 15-tap long (i.e., with 15 coefficients b_0~b_{14} for x(n)~x(n-14)). Given an input signal consisting of 2 frequency components

$$s(n) = \sin(2\pi*n/12)+\cos(2\pi*n/4)$$

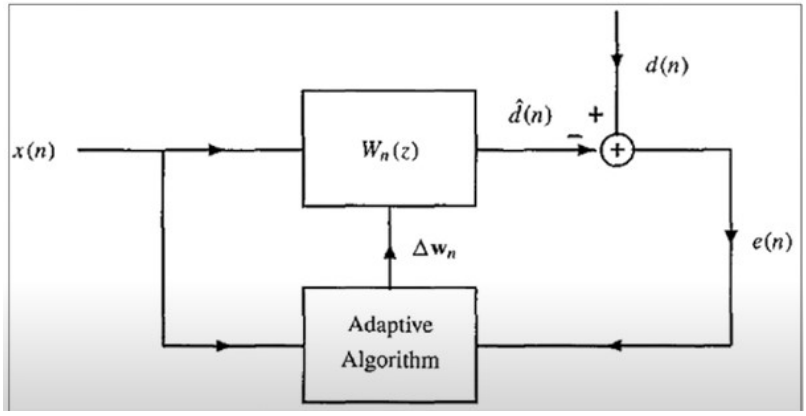
develop an adaptive low pass filter design

Set the target as a low pass filter to remove the high frequency component cos(2 π*n/4) and use sin(2π*n/12) as the desired (or training) signal for LMS adaptation. Assume the step size μ is 10^{-2}.

- write a Matlab code to simulate the LMS based adaptive filtering. Calculate the RMS (root mean square) value of the latest 16 prediction errors
  (i.e., r = sqrt((e^2(n) + e^2(n-1) + ….+ e^2(n-15))/16) and the adaptation is considered being converged if this value is less than 10% of RMS (root mean square) value of the desired signal, which equals 0.1/sqrt(2).
- Show the plot of "r" versus "n" and indicate when the filter converges, i.e. how many training samples are required
- Show the plot of filter coefficients b_i(n), for i = 0~14, versus "n" and see if the values of filter coefficients remain mostly unchanged after convergence
- Apply a 64-point FFT to the impulse response of the converged filter and verify the filter is indeed a low pass one. Note that the input vector to the 64-point FFT is
  (b_0, b_1, ….., b_{14}, 0,0,…..,0) with 49 trailing zeros.
- Change the step size μ to 10^{-4} and see how the behavior of the adaptive filter changes.
- Conduct simulation with a sufficiently large number of samples to see how small the value of "r" can be (the convergence bias)

Derivation steps

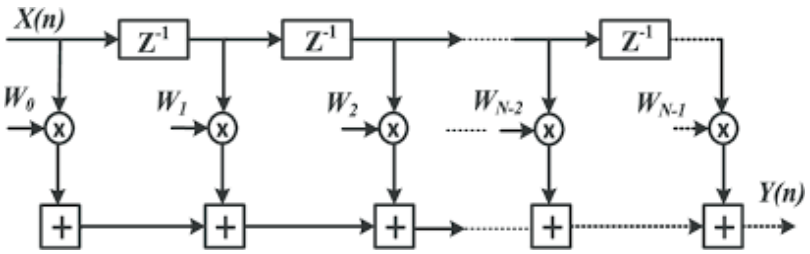
Adaptive Filter specification



1. x(n) is the input signal, wn(z) is the adaptive filter block with coefficients of wn.
2. d_hat(n) is the generated system response and d(n) is the desired signal.
3. e(n) is the error between d_hat(n) and d(n)
4. The adaptive algorithm block determines which kind of policy we should use to find the suitable filter coefficients. In this HW, LMS algorithm is chosen.

The adaptive FIR filter

$$\hat{d}(n) = \sum_{k=0}^{p} \omega_n(k) x(n-k) = \boldsymbol{w}_n^T \boldsymbol{X}(n)$$



- The desired output is genereated through the p-tap FIR filter design, where wn is the coefficients that gets updated on the fly.

Error function

$$e(n) = d(n) - \hat{d}(n)$$
$$= d(n) - w_n^T \boldsymbol{X}(n)$$

$$E\{e(n)x^*(n-k)\} = 0 \;;\; k = 0,1,\dots,p$$

- Error function simply is the difference between the desired signal and the generated system response.
- Ultimate goal is to minimize the autocorrelation between error vector and input signal.

LMS algorithm

$$\boldsymbol{w}_{n+1} = \boldsymbol{w}_n + \mu e(n)\boldsymbol{X}^*(n)$$

$$\omega_{n+1} = \omega_n(k) + \mu e(n)\boldsymbol{X}^*(n-k)$$

- mu is the step sizes for the algorithm, which governs the variability of the coefficients in each iteration.
- e(n)X*(n) is the factor of auto-correlation between the input signal and the error function.

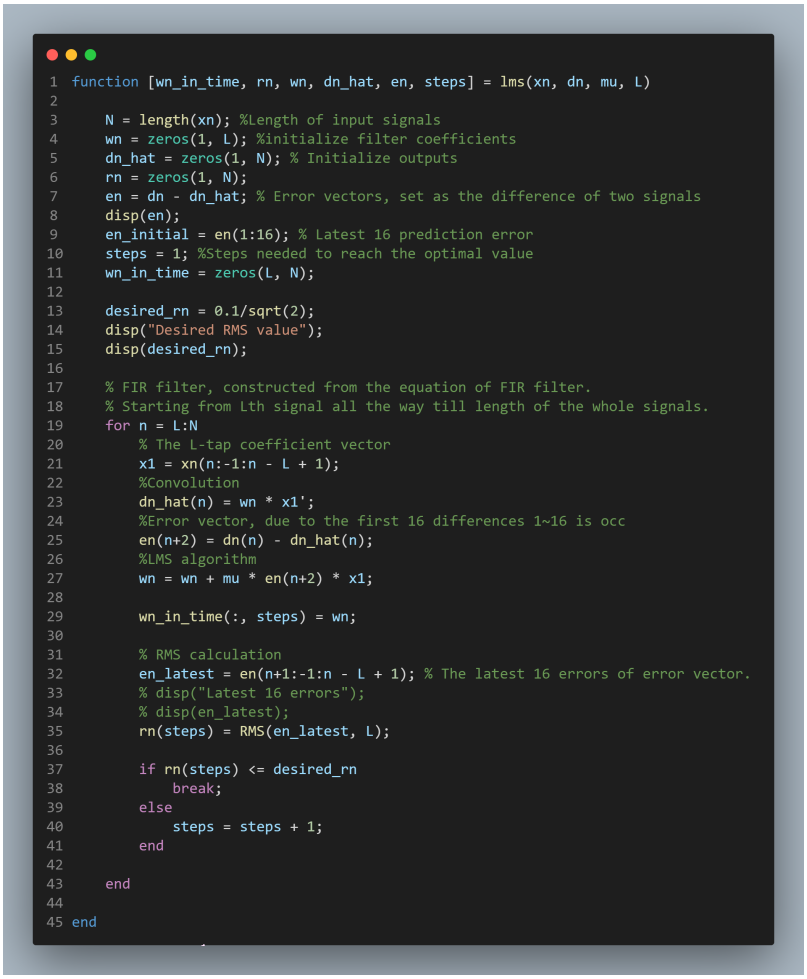
RMS(Root mean square)

$$RMSE = \sqrt{\frac{\sum_{i=1}^{N} (Predicted_i - Actual_i)^2}{N}}$$

- Root mean square used to find the norm of the error vector, we hope that this value be as small as possible s.t. the system is converged.

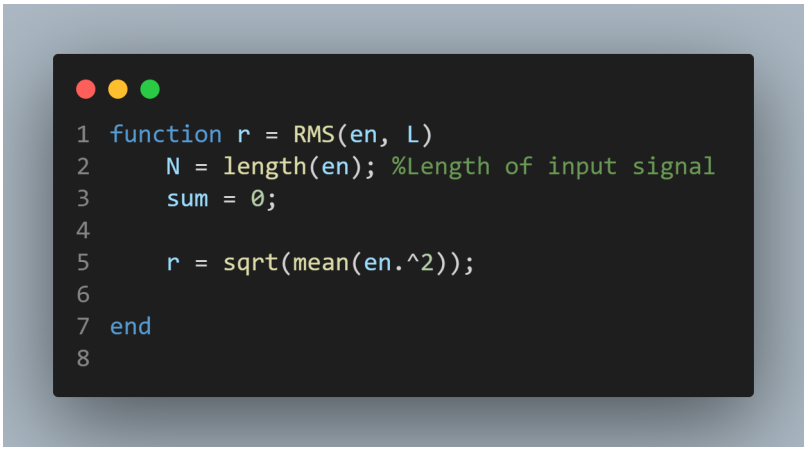
Code

Adaptive Filters



```m
function [wn_in_time, rn, wn, dn_hat, en, steps] = lms(xn, dn, mu, L)

    N = length(xn); %Length of input signals
    wn = zeros(1, L); %initialize filter coefficients
    dn_hat = zeros(1, N); % Initialize outputs
    rn = zeros(1, N);
    en = dn - dn_hat; % Error vectors, set as the difference of two signals
    disp(en);
    en_initial = en(1:16); % Latest 16 prediction error
    steps = 1; %Steps needed to reach the optimal value
    wn_in_time = zeros(L, N);

    desired_rn = 0.1/sqrt(2);
    disp("Desired RMS value");
    disp(desired_rn);

    % FIR filter, constructed from the equation of FIR filter.
    % Starting from Lth signal all the way till length of the whole signals.
    for n = L:N
        % The L-tap coefficient vector
        x1 = xn(n:-1:n - L + 1);
        %Convolution
        dn_hat(n) = wn * x1';
        %Error vector, due to the first 16 differences 1~16 is occ
        en(n+2) = dn(n) - dn_hat(n);
        %LMS algorithm
        wn = wn + mu * en(n+2) * x1;

        wn_in_time(:, steps) = wn;

        % RMS calculation
        en_latest = en(n+1:-1:n - L + 1); % The latest 16 errors of error vector.
        % disp("Latest 16 errors");
        % disp(en_latest);
        rn(steps) = RMS(en_latest, L);

        if rn(steps) <= desired_rn
            break;
        else
            steps = steps + 1;
        end

    end

end
```

RMS



```m
function r = RMS(en, L)
    N = length(en); %Length of input signal
    sum = 0;

    r = sqrt(mean(en.^2));

end

```

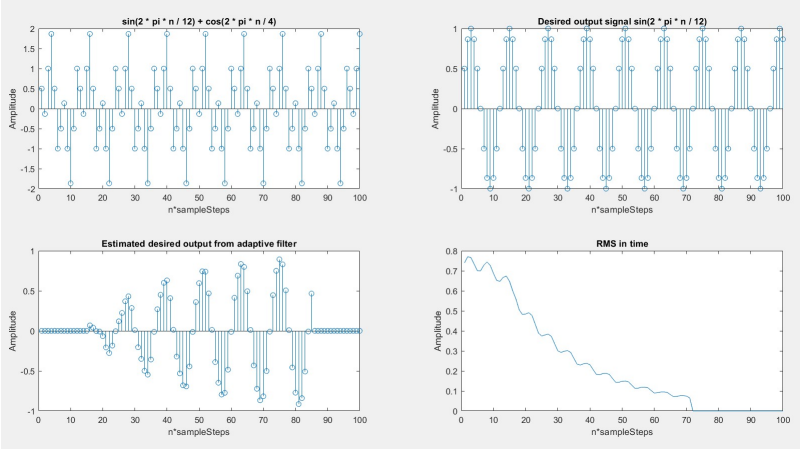
Main drivers

```m
clear all;

% Note sequence starting from 1~60
Sample_size = 100;
n = 1:Sample_size;
L = 15;
mu = 0.01;

%Input signal xn(n)
xn = sin(2 * pi * (n / 12)) + cos(2 * pi * (n / 4));
dn = sin(2 * pi * (n / 12));

%plot of original signal for 50 equally sampled value
figure(1);
sampleSteps = 1;
coefficient_steps = 1;
coefficient_spaced = 1:coefficient_steps:Sample_size;
n_spaced = 1:sampleSteps:Sample_size;
subplot(2, 2, 1);
stem(xn(n_spaced));
title('sin(2 * pi * n / 12) + cos(2 * pi * n / 4)'); xlabel('n*sampleSteps'); ylabel('Amplitude');

subplot(2, 2, 2);
stem(dn(n_spaced));
title('Desired output signal sin(2 * pi * n / 12)'); xlabel('n*sampleSteps'); ylabel('Amplitude');

% Adaptive Filter, Look for the minimum samples to reach 10% of LMS
[wn_in_time, rn, wn, dn_hat, en, steps] = lms(xn, dn, mu, L);

% Plot of dn_hat
subplot(2, 2, 3);
stem(dn_hat(n_spaced));
title('Estimated desired output from adaptive filter'); xlabel('n*sampleSteps'); ylabel('Amplitude');

% Plot of rn
subplot(2, 2, 4);
plot(rn(n_spaced));
title('RMS in time'); xlabel('n*sampleSteps'); ylabel('Amplitude');

disp("RMS final value");
% Since it might not converge, if the value converges, the value is stored in rn(steps), otherwise rn(steps-1)
if Sample_size > 10000
    disp(rn(steps-1));
else
    disp(rn(steps));
end

disp("Total Steps needed to reach 10% of RMS");
disp(steps);
```

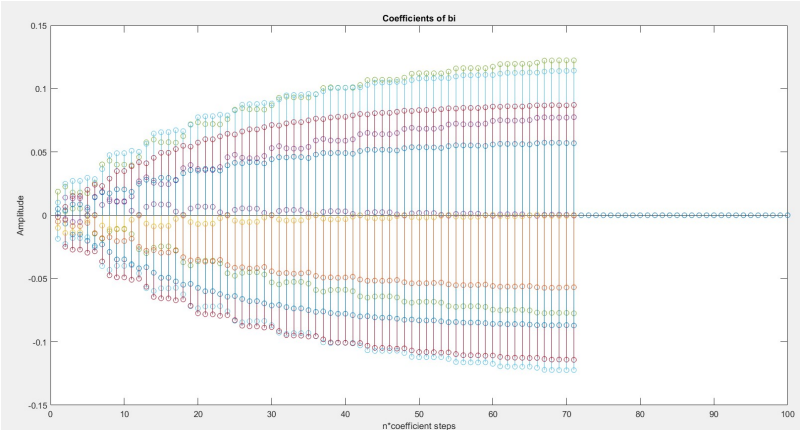
```m
disp("Total Steps needed to reach 10% of RMS");
disp(steps);
% Plot of Coefficients v.s. steps
figure(2);
wn_in_time = wn_in_time';
stem(wn_in_time(coefficient_spaced, :));
title('Coefficients of bi'); xlabel('n*coefficient steps'); ylabel('Amplitude');

% FFT for the impulse response of converged filters.
N = 64;
wn_padded = zeros(1, N);
wn_padded(1:L) = wn;

figure(3);

Y = fft(wn_padded, N);
stem(Y); % Note must use stem.
```

Results

Adaptive Filter Response n=100, mu = 0.01, sampleSteps = 1 and RMS over time



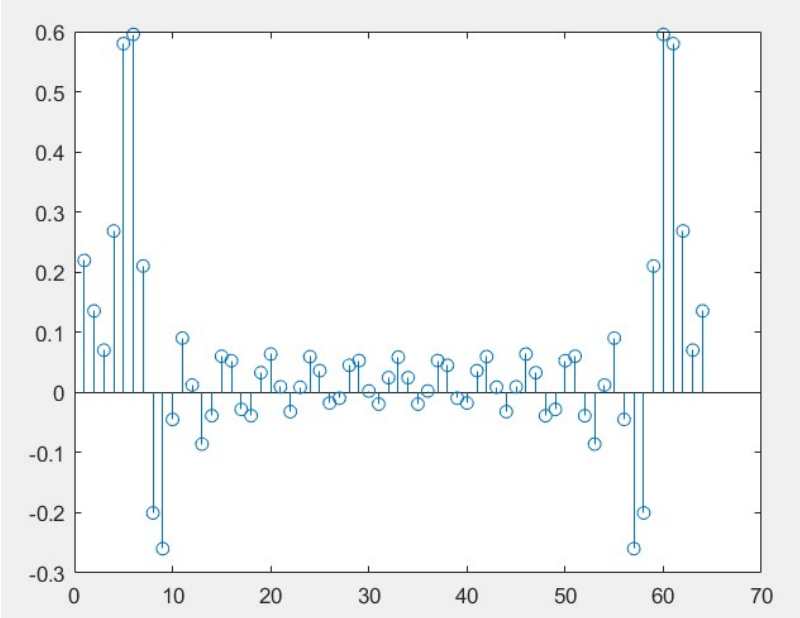
Filter Coefficients over time



Converged steps

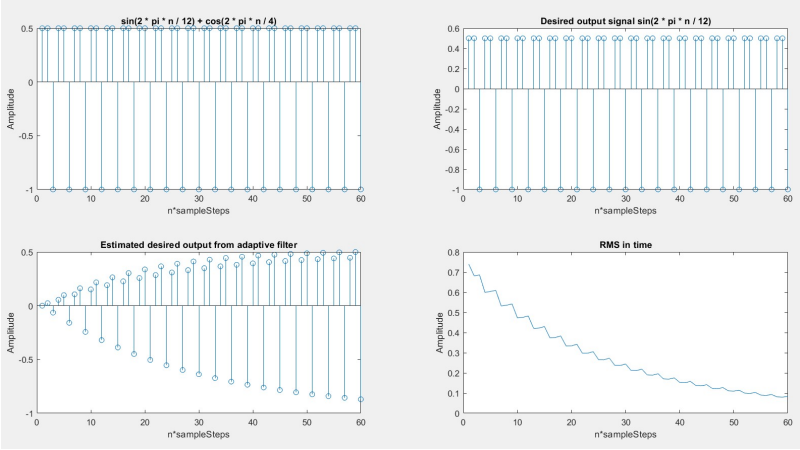
```
Desired RMS value
    0.0707

RMS final value
    0.0653

Total Steps needed to reach 10% of RMS
    71
```

64-point FFT spectrum

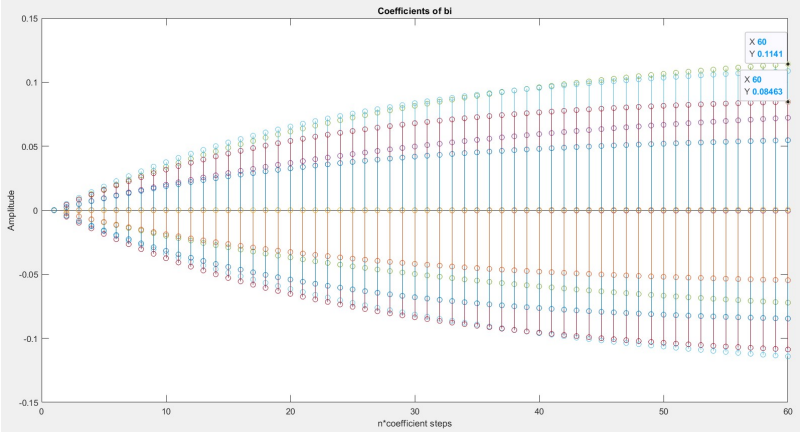


- The response is indeed a Low-Pass filter response.

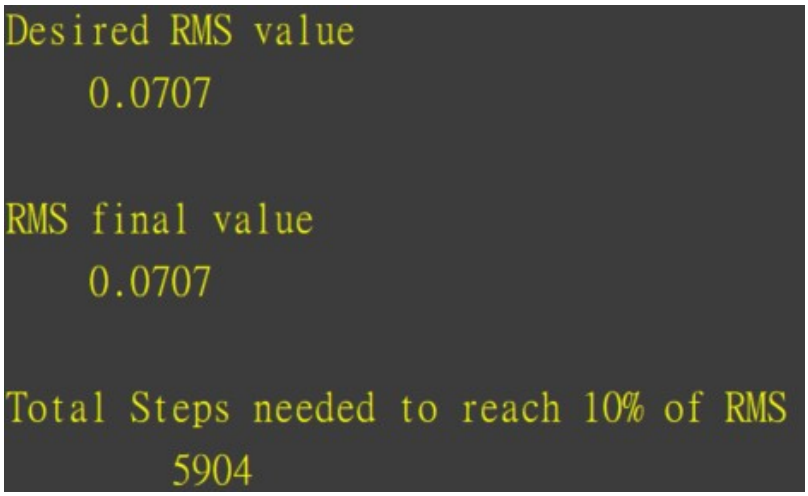
Adaptive Filter Response n=6000, mu = 0.0001, samepleSteps = 100 and RMS over time



Filter Coefficients over time

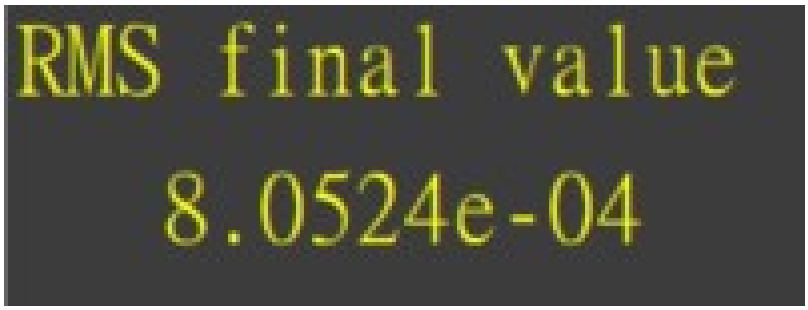


Converged steps



- Notice the changing of the RMS value responses more drmatically due to smaller step difference. Also it takes longer and more sample points for it to converge.

RMS with large sample size n = 20000



- The convergence bias is found, the filter cannot converge any further, it keeps on oscillating between within the convergence bias.

Note

- Due to the fact that sample sizes are large for smaller mu, plotting all of the signals makes analysis hard, thus samples_steps is defined s.t. only a certain multiple of signal sample_steps are selected for plotting.
- The latest 16 prediction errors should be selected for caculation, selecting more than that might yield the wrong results, and the filter would never converge.

II. Discrete Wavelet Filter Design

Problem

For a discrete wavelet transform (DWT) adopting (9/7) filters, i.e. the low pass filter $h(i)$ is 9-taped and the high pass filter $g(i)$ is 7-taped. Both filters are liner phased and have symmetric coefficients. The filter coefficients are given in Table 1. For a corresponding inverse discrete wavelet transform, the low pass filter $q(i)$ is 7-taped and the high pass filter $p(i)$ is 9-taped. The filter coefficients are given in Table 2.

Table 1. Analysis filter coefficients for the floating point 9/7 filter

Analysis Filter Coefficients		
i	Lowpass Filter h_i	Highpass Filter g_i
0	0.852698679009	-0.788485616406
±1	0.377402855613	0.418092273222
±2	-0.110624404418	0.040689417609
±3	-0.023849465020	-0.064538882629
±4	0.037828455507	

Note: the high pass and low pass filter notations here are opposite to those in the lecture note

Table 2. Synthesis filter coefficients for the floating point 9/7 filter

Synthesis Filter Coefficients		
i	Low pass Filter q_i	High pass Filter p_i
0	0.788485616406	-0.852698679009
±1	0.418092273222	0.377402855613
±2	-0.040689417609	0.110624404418
±3	-0.064538882629	-0.023849465020
±4		-0.037828455507

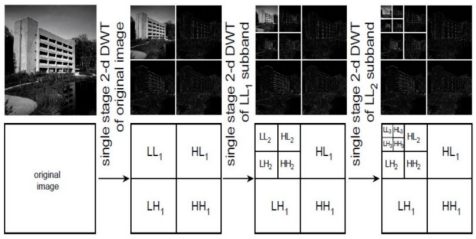
- Goal is to build a DWT filter by adopting the (9/7) filters composed of high pass and low pass components.

Problem a

a) For a 512×512 gray scale image (will be provided along with the homework assignment), please conduct a 2-D 3-level DWT transform (as shown in Figure 1) and show the transformed result. Then conduct a 2-D 3-level IDWT to convert it back. Please compare if the reconstructed image (after IDWT) is same as the original image by calculating its PSNR value.

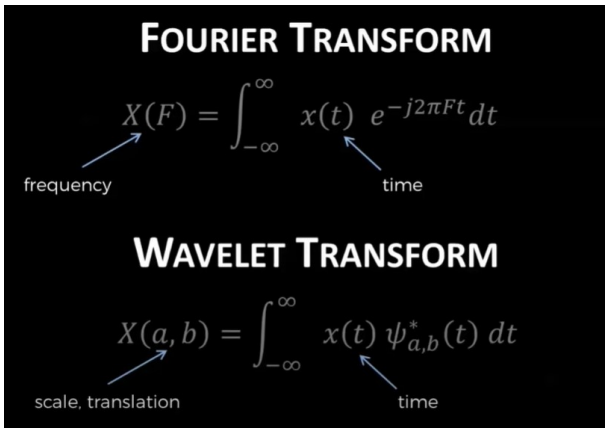
Problem b

b) By setting all three level 1 sub-bands HL1, LH1 and HH1 coefficients to zeros and perform IDWT. See how the reconstructed image is different from the original one and calculate its PSNR value.

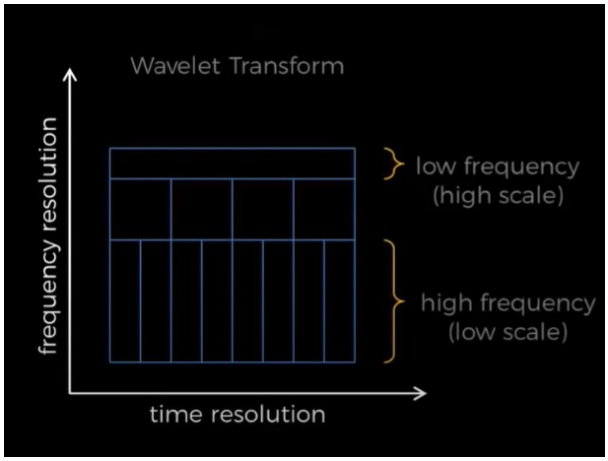


Derivation steps

Wavelet transform



- For a fourier transform, the orthnormal basis is selected as e^j2piFt as analyzing function, however for the wavelet has orthonormal basis of wavelet analyzing fuction phi(t).
- Fourier transform ouputs frequency, yet wavelet outputs a translation and scaled autocorrelation of the input.



- Wavelet would has better time localization for high frequency yet worse time localization for low frequency.
- Wavelet is used to tackle with problem of time and frequency resolution since you cannot have both great time resolution and frequency resolution at the same time. Wavelet is the balance between time and frequency resolution.
- It is extensively used for compressing images.

2D DWT transform octave

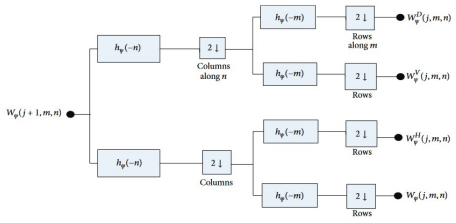


Figure 1: The discrete wavelet transform. Here, $h_\psi(-n)$, $h_\varphi(-n)$, $h_\psi(-m)$, and $h_\varphi(-m)$ are wavelet vectors. 2 ↓ represents downsampling by 2.

- Each octave generates LL,LH,HH,HL subbands. The figure only shows octave1, for three level octave, 3 octaves must be cascaded together.
- The 3 level DWT structure has 3 octaves, where after passing each octaves the image becomes 2 times smaller than the original image due to down-Sampler.
- When doing downsampling, low pass filter sample the odd number output data, while high pass filters keep the even number output data.

2D IDWT transform octave

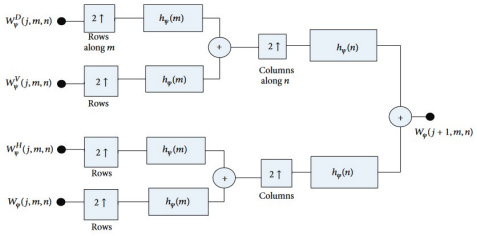


Figure 2: The inverse discrete wavelet transform. Here, $h_\psi(m)$, $h_\varphi(m)$, $h_\psi(n)$, and $h_\varphi(n)$ are wavelet vectors. 2 ↑ signifies upsampling by 2.

- The 3 level IDWT structure has 3 octaves also, where after passing each octaves the image becomes 2 times larger than the original image due to up-Sampler. During the upSampling process, action should be taken to combat the loss of information when doing up Sampling.

Symmetric extension scheme

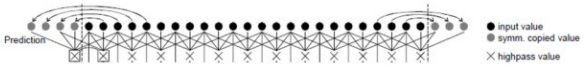


Figure 3. Symmetric extension scheme for boundary pixels

- Due to the boundary condition when doing filtering, one must extend the original signal length s.t. the boundary would not get convolved into 0 values.

PSNR(Peak signal to noise ratio) and MSE(Mean square error)

$$\mathrm{MSE} = \frac{1}{\mathrm{M}\cdot\mathrm{N}}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left(I(i,j)-\hat{I}(i,j)\right)^2$$

$$PSNR = 10\log_{10}\left(\frac{MAXI^2}{\mathrm{MSE}}\right)$$

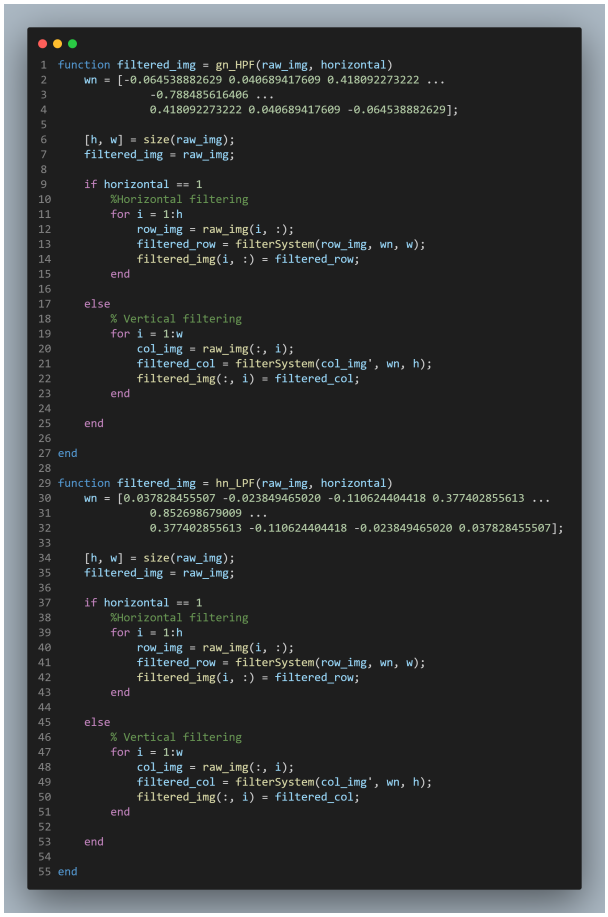
- Used to check whether the reconstructed image is close to the original image or not.

Code

FIR Filter(Symmetric extended)

```
function yn = filterSystem(xn, wn, N)
    N = length(xn);
    M = size(wn, 2);
    L = fix(M / 2); % extend size // 2
    yn = zeros(1, N); % result

    x = [flip(xn(2:L + 1)), xn, flip(xn(N - L:N - 1))]; % Symmetric extension!

    for i = 1:N
        yn(i) = dot(wn, x(1, i:i + M - 1));
    end

end
```

g(n) High pass filter and h(n) low pass filter



```
function filtered_img = gn_HPF(raw_img, horizontal)
    wn = [-0.064538882629 0.040689417609 0.418092273222 ...
             -0.788485616406 ...
             0.418092273222 0.040689417609 -0.064538882629];

    [h, w] = size(raw_img);
    filtered_img = raw_img;

    if horizontal == 1
        %Horizontal filtering
        for i = 1:h
            row_img = raw_img(i, :);
            filtered_row = filterSystem(row_img, wn, w);
            filtered_img(i, :) = filtered_row;
        end

    else
        % Vertical filtering
        for i = 1:w
            col_img = raw_img(:, i);
            filtered_col = filterSystem(col_img', wn, h);
            filtered_img(:, i) = filtered_col;
        end

    end

end

function filtered_img = hn_LPF(raw_img, horizontal)
    wn = [0.037828455507 -0.023849465020 -0.110624404418 0.377402855613 ...
             0.852698679009 ...
             0.377402855613 -0.110624404418 -0.023849465020 0.037828455507];

    [h, w] = size(raw_img);
    filtered_img = raw_img;

    if horizontal == 1
        %Horizontal filtering
        for i = 1:h
            row_img = raw_img(i, :);
            filtered_row = filterSystem(row_img, wn, w);
            filtered_img(i, :) = filtered_row;
        end

    else
        % Vertical filtering
        for i = 1:w
            col_img = raw_img(:, i);
            filtered_col = filterSystem(col_img', wn, h);
            filtered_img(:, i) = filtered_col;
        end

    end

end
```

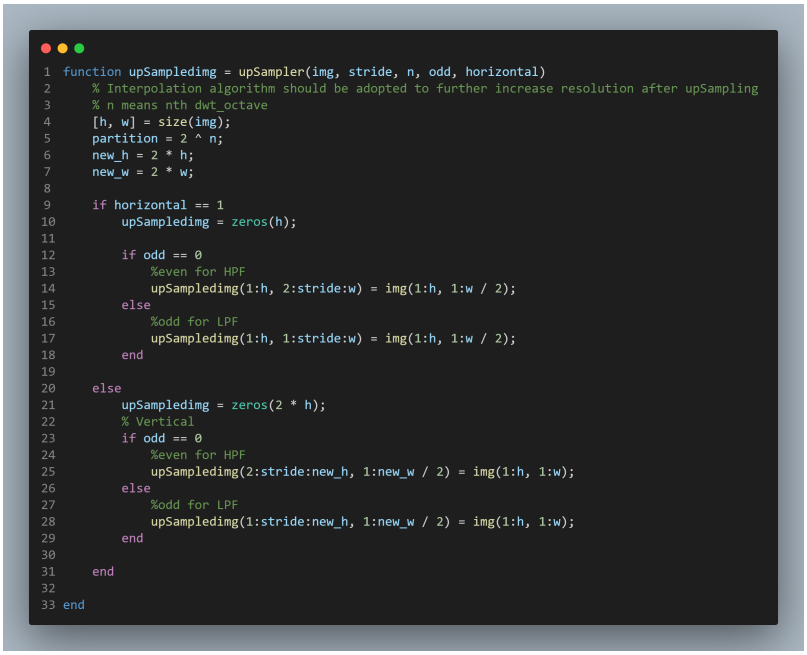
p(n) High pass filter and q(n) low pass filter

```
function filtered_img = qn_LPF(raw_img, horizontal)
    wn = [-0.064538882629 -0.040689417609 0.418092273222 ...
              0.788485616406 ...
              0.418092273222 -0.040689417609 -0.064538882629];

    [h, w] = size(raw_img);
    filtered_img = raw_img;

    if horizontal == 1
        %Horizontal filtering
        for i = 1:h
            row_img = raw_img(i, :);
            filtered_row = filterSystem(row_img, wn, w);
            filtered_img(i, :) = filtered_row;
        end

    else
        % Vertical filtering
        for i = 1:w
            col_img = raw_img(:, i);
            filtered_col = filterSystem(col_img', wn, h);
            filtered_img(:, i) = filtered_col;
        end

    end

end

function filtered_img = pn_HPF(raw_img, horizontal)
    wn = [-0.037828455507 -0.023849465020 0.110624404418 0.377402855613 ...
              -0.852698679009 ...
              0.377402855613 0.110624404418 -0.023849465020 -0.037828455507];

    [h, w] = size(raw_img);
    filtered_img = raw_img;

    if horizontal == 1
        %Horizontal filtering
        for i = 1:h
            row_img = raw_img(i, :);
            filtered_row = filterSystem(row_img, wn, w);
            filtered_img(i, :) = filtered_row;
        end

    else
        % Vertical filtering
        for i = 1:w
            col_img = raw_img(:, i);
            filtered_col = filterSystem(col_img', wn, h);
            filtered_img(:, i) = filtered_col;
        end

    end

end
```

Down Sampler

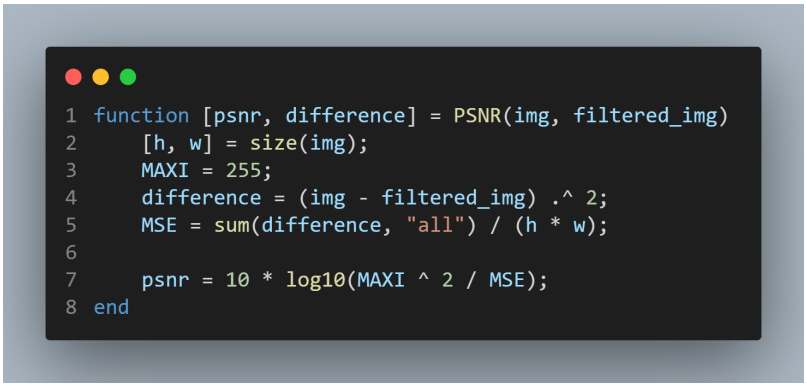
```
function downSampledimg = downSampler(img, stride, odd, n, horizontal)
    [h, w] = size(img);
    partition = 2 ^ n;
    % downSampledimg = img;
    downSampledimg = zeros(h);

    if horizontal == 1

        if odd == 0
            %even for HPF
            downSampledimg(1:h, 1:w / 2) = img(1:h, 2:stride:w);
        else
            %odd for LPF
            downSampledimg(1:h, 1:w / 2) = img(1:h, 1:stride:w);
        end

    else

        if odd == 0
            %even for HPF
            downSampledimg(1:h / 2, 1:w) = img(2:stride:h, 1:w);
        else
            %odd for LPF
            downSampledimg(1:h / 2, 1:w) = img(1:stride:h, 1:w);
        end

    end

end
```

UpSampler



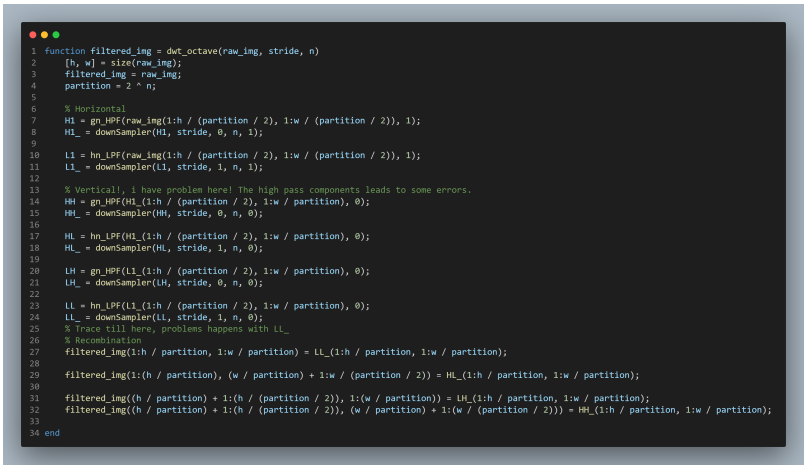
```
function upSampledimg = upSampler(img, stride, n, odd, horizontal)
    % Interpolation algorithm should be adopted to further increase resolution after upSampling
    % n means nth dwt_octave
    [h, w] = size(img);
    partition = 2 ^ n;
    new_h = 2 * h;
    new_w = 2 * w;

    if horizontal == 1
        upSampledimg = zeros(h);

        if odd == 0
            %even for HPF
            upSampledimg(1:h, 2:stride:w) = img(1:h, 1:w / 2);
        else
            %odd for LPF
            upSampledimg(1:h, 1:stride:w) = img(1:h, 1:w / 2);
        end

    else
        upSampledimg = zeros(2 * h);
        % Vertical
        if odd == 0
            %even for HPF
            upSampledimg(2:stride:new_h, 1:new_w / 2) = img(1:h, 1:w);
        else
            %odd for LPF
            upSampledimg(1:stride:new_h, 1:new_w / 2) = img(1:h, 1:w);
        end

    end

end
```

PSNR



```
function [psnr, difference] = PSNR(img, filtered_img)
    [h, w] = size(img);
    MAXI = 255;
    difference = (img - filtered_img) .^ 2;
    MSE = sum(difference, "all") / (h * w);

    psnr = 10 * log10(MAXI ^ 2 / MSE);
end
```

DWT octave



```
function filtered_img = dwt_octave(raw_img, stride, n)
    [h, w] = size(raw_img);
    filtered_img = raw_img;
    partition = 2 ^ n;

    % Horizontal
    H1 = gn_HPF(raw_img(1:h / (partition / 2), 1:w / (partition / 2)), 1);
    H1_ = downSampler(H1, stride, 0, n, 1);

    L1 = hn_LPF(raw_img(1:h / (partition / 2), 1:w / (partition / 2)), 1);
    L1_ = downSampler(L1, stride, 1, n, 1);

    % Vertical!, i have problem here! The high pass components leads to some errors.
    HH = gn_HPF(H1_(1:h / (partition / 2), 1:w / partition), 0);
    HH_ = downSampler(HH, stride, 0, n, 0);

    HL = hn_LPF(H1_(1:h / (partition / 2), 1:w / partition), 0);
    HL_ = downSampler(HL, stride, 1, n, 0);

    LH = gn_HPF(L1_(1:h / (partition / 2), 1:w / partition), 0);
    LH_ = downSampler(LH, stride, 0, n, 0);

    LL = hn_LPF(L1_(1:h / (partition / 2), 1:w / partition), 0);
    LL_ = downSampler(LL, stride, 1, n, 0);
    % Trace till here, problems happens with LL_
    % Recombination
    filtered_img(1:h / partition, 1:w / partition) = LL_(1:h / partition, 1:w / partition);

    filtered_img(1:(h / partition), (w / partition) + 1:w / (partition / 2)) = HL_(1:h / partition, 1:w / partition);

    filtered_img((h / partition) + 1:(h / (partition / 2)), 1:(w / partition)) = LH_(1:h / partition, 1:w / partition);
    filtered_img((h / partition) + 1:(h / (partition / 2)), (w / partition) + 1:(w / (partition / 2))) = HH_(1:h / partition, 1:w / partition);

end
```

IDWT octave

```
function reconstructed_img = idwt_octave(compressed_img, stride, n)

    [h, w] = size(compressed_img);
    partition = 2 ^ n;
    reconstructed_img = compressed_img;

    % Extraction
    LL_ = compressed_img(1:h / partition, 1:w / partition);
    LH_ = compressed_img((h / partition) + 1:(h / (partition / 2)), 1:(w / partition));

    HL_ = compressed_img(1:(h / partition), (w / partition) + 1:w / (partition / 2));
    HH_ = compressed_img((h / partition) + 1:(h / (partition / 2)), (w / partition) + 1:(w / (partition / 2)));

    % Vertical!
    LL = upSampler(LL_, stride, n, 1, 0);
    LL_test = LL(1:h / (partition / 2), 1:w / partition);
    LL(1:h / (partition / 2), 1:w / (partition)) = qn_LPF(LL_test, 0);

    % LH is wrong~ for the last 3 columns.
    LH = upSampler(LH_, stride, n, 0, 0);
    LH(1:h / (partition / 2), 1:w / (partition)) = pn_HPF(LH(1:h / (partition / 2), 1:w / partition), 0);

    s0 = LL + LH;

    HL = upSampler(HL_, stride, n, 1, 0);
    HL(1:h / (partition / 2), 1:w / (partition)) = qn_LPF(HL(1:h / (partition / 2), 1:w / partition), 0);

    HH = upSampler(HH_, stride, n, 0, 0);
    HH(1:h / (partition / 2), 1:w / (partition)) = pn_HPF(HH(1:h / (partition / 2), 1:w / partition), 0);

    % Minor difference in boundary of 160 columns and rows.
    s1 = HL + HH;

    % disp("Size of s0");
    % disp(size(s0));
    % disp("Size of s1");
    % disp(size(s1));

    % Horizontal!
    s0_ = upSampler(s0, stride, n, 1, 1);
    disp("Size of s0_");
    disp(size(s0_));
    s0_ = qn_LPF(s0_, 1);

    s1_ = upSampler(s1, stride, n, 0, 1);
    disp("Size of s1_");
    disp(size(s1_));
    s1_ = pn_HPF(s1_, 1);

    s_temp = s0_ + s1_;

    reconstructed_img(1:h / (partition / 2), 1:w / (partition / 2)) = s_temp;

end
```

Main driver

```
%================================================================
%  RD image
%================================================================
img = imread('image.bmp');
img = double(img);
[h, w] = size(img);
n = 1;
stride = 2;
```

```
%================================================================
%  DWT
%================================================================
% Octave 1
octave1 = dwt_octave(img, stride, 1);

figure(2);
imshow(octave1, []);
title('Octave1');

% Octave 2
octave2 = octave1;
octave2 = dwt_octave(octave1, stride, 2);

figure(3);
imshow(octave2, []);
title('Octave2');

% Octave 3

octave3 = dwt_octave(octave2, stride, 3);
```

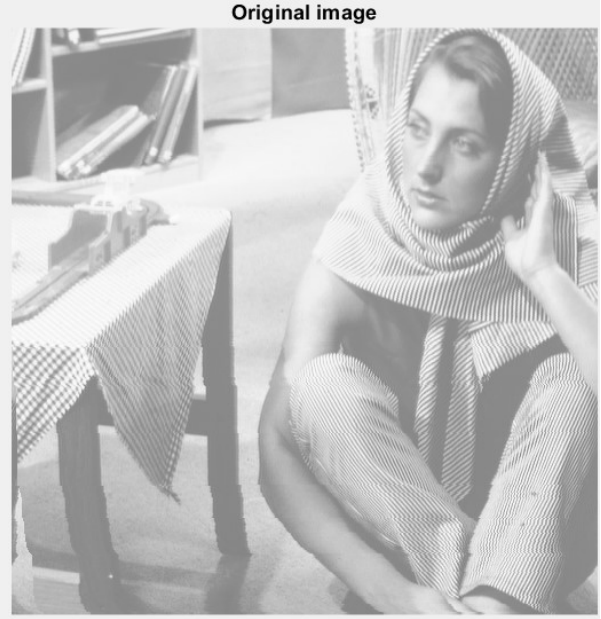
```
%==========================================================
%  IDWT
%==========================================================
zero_padded = 1;
octave3_ = zeros(h);

if zero_padded == 1
    octave3_(1:h / 2, 1:w / 2) = octave3(1:h / 2, 1:w / 2);
else
    octave3_ = octave3;
end

ioctave3 = idwt_octave(octave3_, stride, 3);

ioctave2 = idwt_octave(ioctave3, stride, 2);

ioctave1 = idwt_octave(ioctave2, stride, 1);

restored_image = ioctave1;

figure(7);
imshow(restored_image, []);
title('Restored Image');
```

```
%==========
%  PLOTS
%==========
figure(4);
imshow(octave3, []);
title('DWT result');

%# make sure the image doesn't disappear if we plot something else
hold on

%# define points (in matrix coordinates)
p1 = [1, 256];
p2 = [512, 256];

p3 = [256, 1];
p4 = [256, 512];

p5 = [1, 128];
p6 = [256, 128];

p7 = [128, 1];
p8 = [128, 256];

p9 = [1, 64];
p10 = [128, 64];

p11 = [64, 1];
p12 = [64, 128];

%# plot the points.
%# Note that depending on the definition of the points,
%# you may have to swap x and y
plot([p1(2), p2(2)], [p1(1), p2(1)], [p3(2), p4(2)], [p3(1), p4(1)], 'Color', 'r', 'LineWidth', 2);
plot([p5(2), p6(2)], [p5(1), p6(1)], [p7(2), p8(2)], [p7(1), p8(1)], 'Color', 'r', 'LineWidth', 2);
plot([p9(2), p10(2)], [p9(1), p10(1)], [p11(2), p12(2)], [p11(1), p12(1)], 'Color', 'r', 'LineWidth', 2)
```

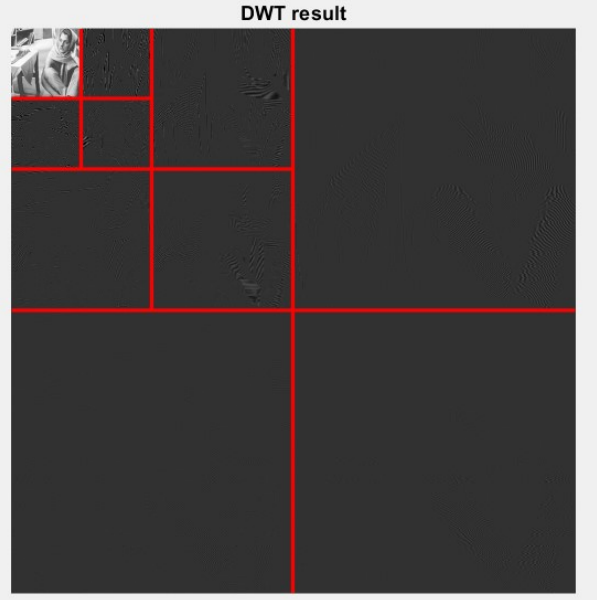
```
%==========================================================
%  PSNR
%==========================================================
disp("PSNR:");
[psnr, difference] = PSNR(img, ioctave1);
fprintf('%.2f db\n', psnr);
```

Result

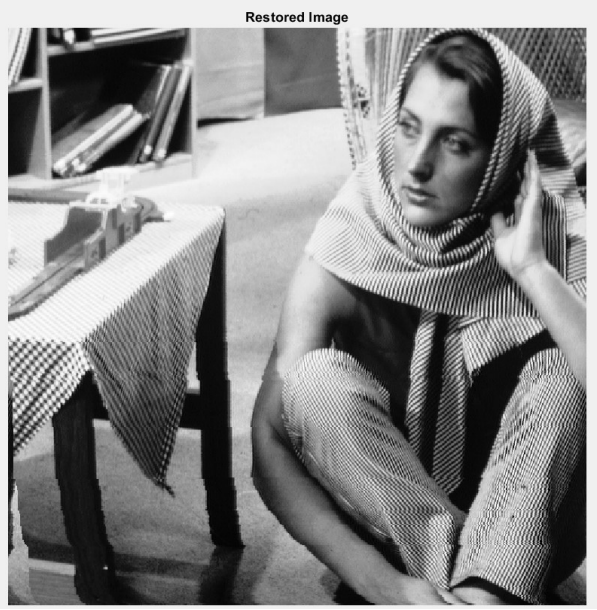
Original image



a) DWT result



Holds LH,HH,HL Restored image

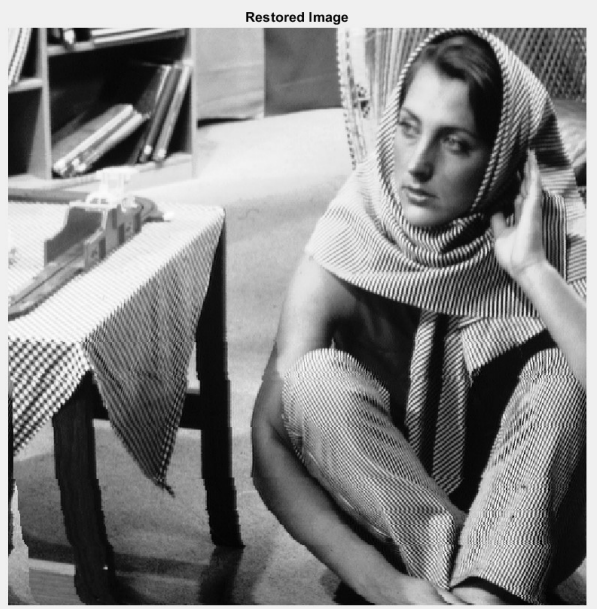


PSNR

```
PSNR:
234.20 db
```

b) Sets LH,HH,HL all to 0

Restored image



PSNR

```
PSNR:
23.29 db
```

- Note the image after setting LL,LH,HL,HH to zero is as expected lower than using the high frequency data of the LL,LH,HL,HH subbands.

Note

1. The DWT with the subbands attached can yields amazing restoration results.
2. High frequency components are important for image reconstruction.

III. References

[1] Advanced Digital Signal Processing, Adaptive Filters by Prof.Vaibhav Pandit

[2] Advanced Digital Signal Processing, LMS Algorithm by Prof.Vaibhav Pandit

[3] MIT RES.6-008 Digital Signal Processing,Lec 17, 1975 by Alan Oppenheim

[4] EE123 Digital Signal Processing, SP'16 L12 - Discrete Wavelet Transform

[5] Easy Introduction to Wavelets, by Simon Xu

[6] VLSI Digital Signal processing systems Design and Implementation, p25~28 by Parhi

[7] Image Denoising Based on Improved Wavelet Threshold Function for Wireless Camera Networks and Transmissions,Sep 2015, Reserach Gate,Xiaoyu Wang Xiaoxu Ou Bo-Wei Chen Mucheol Kim