Use IIR Wiener filter for Audio Denoising

I use IIR Wiener Filter for Audio denoising as below - first we need to compute the PSD(Power Spectral Density) of signals:

- Clean signal: $P_d \ or \ P_d(e^{j\omega})$
- Noise: $P_v \ or \ P_v(e^{j\omega})$
- Observed Signal: $P_x \ or \ P_x(e^{j\omega})$

For finding PSD we use FFT (Fast Fourier Transform) to find it. And then we compute squared absolute value of the FFT to estimate PSD:

$$FFT: \ X(k) = \sum_{n=0}^{N-1} x(n) W_n^{kn} = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi kn}{N}} \qquad k = 0,1,\dots,N-1$$

For PSD we have:

$$PSD[k] = |X[k]|^2$$

Finally, we use this formula to find the filter coefficient *W* and implement it:

$$W(e^{j\omega}) = \frac{P_{dx}(e^{j\omega})}{P_x(e^{j\omega})} = \frac{P_d(e^{j\omega})}{P_d(e^{j\omega}) + P_v(e^{j\omega})}$$

- The **IFFT** of the $W(e^{j\omega})$ gives the **time domain FIR Wiener filter** *h[n]* at later stage.

Here we apply the *Wiener Filter* in frequency domain:

$$Y_{freq} = W * X \ \ or \ Y_{freq} = \sum_{n=0}^{N-1} W(n) * X(n)$$

- Here we used element-wise multiplication (.* in MATLAB).

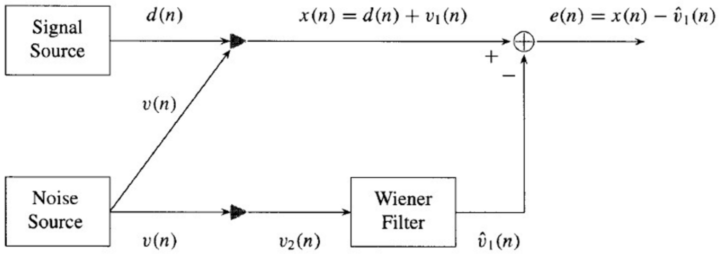
After calculating the wiener filtered signal in frequency domain, we have to go to time domain. We use *IFFT* (Inverse FFT) algorithm for it:

$$FFT: X(k) = \sum_{n=0}^{N-1} x(n) e^{-\frac{j2\pi kn}{N}} \quad \Leftrightarrow \quad IFFT: x(n) = \frac{1}{N} \sum_{k=0}^{N-1} X(k) e^{+\frac{j2\pi kn}{N}}$$

- MATLAB code: *IFFT(x)*

After computing IFFT of the filter, we apply it for finding: $\hat{d}(n)$

we have 2 different ways to determine PSD of the input noise P_v in real world applications:



1- Use a second antenna and a wiener filter to extract the noise *v(n)* from the received signal and then continue to do computations
2- Use subtraction of Observed signal – Desired signal ($v(n) = x(n) - d(n)$)

The first approach is useful for practical applications where we have not access to d(n). Here we try to use the both approaches and find out the differences.

- Othe approaches:
  - Estimate P_v & P_d from noise-only signal: a part of signal like beginning or end of it be noise-only → calculate P_v from it and then compute $P_d = P_x - P_v$ ⸙the $P_d \geq 0$ must be satisfied
  - Use a trained-model or template model

LMS(Least Mean Squared):

$$w(n+1) = w(n) + \mu . e(n) x^*(n-l)$$

$x^*(n)$: reference noise from microphone
$w(n)$: filter coefficients → $w(n) = [w_0(n), w_1(n), w_2(n), \dots, w_{L-1}(n)]^T$
μ: step size
$e(n)$: instantaneous error between the desired and estimated output

We have primary input: $x(n) = d(n) + v1(n)$ → clean voice + noise
Reference noise from microphone (near the noise source): $v2(n)$

- The LMS estimated the noise and **subtracts** it from the **primary signal.**

Real world signal:

$$x(n) = d(n) + v_1(n)$$

Noise picked by microphone:

$$v_2(n) = v_1(n) + small\ variation$$

Simulate a scenario when clean or noisy signals are emphasized differently:

$$x_{weighted}(n) = k_1 d(n) + k_2 v_2(n)$$

Estimate the noise component y(n):

$$\boldsymbol{y(n) = w^T(n) v_2(n)}$$

Error signal:

$$\boldsymbol{e(n) = x(n) - v_1(n)}$$

Weights of the filter coefficients should adaptively update to minimize the error:

$$w(n+1) = w(n) + \mu . e(n) x^*(n-l)$$

- In the code I replaced $x^*(n-1)$ with $v_2(n)$

Finally we have:

$$d(n) = x(n) - v(n) \quad \rightarrow \quad \hat{d}(n) = e(n)$$

MATLAB Code:

```
clc;
clear;
close all;
```

- Insert an audio file by *audiored* function. It returns vector of data(audio samples) & sampling frequency(Fs)

```
%% === Load a real clean signal ===
[clean_audio, Fs] = audioread('Voice matlab.wav');
clean_audio = clean_audio(:,1);  % Use mono if stereo
d = clean_audio';                % Row vector
```

- Add random noise to the original audio.

```
%% === Simulate Noise and Composite Signal ===
noise = 0.05 * randn(size(d));   % Additive noise
v1 = noise;          % Noise added to the signal
x = d + v1;          % Primary input: noisy signal d(n)+v1(n)
v2 = noise + 0.005 * randn(size(d));  % Reference mic noise
(slightly different, to simulate real-world)
```

- Insert LMS parameters and k1 & k2 coefficient parameters:

```
mu = 0.01;  % Step size-> small:stable-slow, large:fastunsatbel
L = 64;     % Filter length v(n-L) & w(L-1) -> past samples
N = length(d);
w = zeros(1, L); % Adaptive filter weights // weight vector of
adaptive filter
```

```
%% === k1 and k2 Coefficients ===
k1 = 1.0;           % Weight for desired signal
k2 = 1.0;           % Weight for noise signal
% Composite weighted signal
x = k1*d + k2*v1;    % Use v1 as added noise -> apply K1 & k2

%% === Buffers ===
x_buffer = zeros(1, L);
d_hat = zeros(1, N); % final denoised signal
error = zeros(1, N); % learning signal for LMS
```

- LMS adaptive filtering algorythm

```
%% === LMS Adaptive Filtering ===
for n = L:N
v2_vec = v2(n:-1:n-L+1);    % Reference noise vector
y = w * v2_vec';    % Filter output

error(n) = x(n) - y; % Error signal (desired - estimated noise)

w = w + mu * error(n) * v2_vec;  % Update filter weights

d_hat(n) = error(n); % Store estimate
  end
```

- Plot the signals:

```
%% === Plot results ===
t = (0:N-1)/Fs;

figure;
subplot(3,1,1); plot(t, d); title('Original Clean Signal');
xlabel('Time (s)'); grid on;
subplot(3,1,2); plot(t, x); title('Noisy Signal'); xlabel('Time
(s)'); grid on;
subplot(3,1,3); plot(t, d_hat); title('Denoised Output (Freq-
Domain Wiener)'); xlabel('Time (s)'); grid on;
```

- Apply *soundsc* function to play the audio signal

```
%% === Playback ===
disp('Playing original signal...'); soundsc(d, Fs);
pause(length(d)/Fs + 1);
```

```
disp('Playing noisy signal...'); soundsc(x, Fs);
pause(length(x)/Fs + 1);
disp('Playing denoised signal (Freq-Domain)...'); soundsc(d_hat,
Fs);
```