ELE8059 Assignment 2

QUEEN'S UNIVERSITY BELFAST

Student ID : 40178580
Ben Russell
Date : 20/02/2023

1 Overview

This report aims to estimate the inter-peak interval, and amplitudes of the primary and secondary peak in an ECG waveform. The report will follow 4 main stages of analysis, these are:

1) Time-Frequency analysis will be complete on the provided noise-corrupt ECG signal, and the frequencies at which any spurious components fall will be identified.

2) A suitable approach to eliminate the identified noise will be proposed. This will use digital filters and follow approaches currently used in relevant literature.

3) The digital filters will be designed on MATLAB, and applied in the time domain to the measurement signal, with the aim to produce a clean ECG signal with easily identifiable peaks/features.

4) The inter-peak interval will be estimated by visual inspection.

5) The height of both the primary-peak and secondary-peak will be estimated by visual inspection.

2 Time-Frequency Analysis

The first step to solve this problem is to identify the frequency bands that we need to filter from the ECG signal. We expect the ECG signal to be a periodic (Roughly) signal and so looking in the time-frequency domain, we should be able to easily identify the frequencies at which noise is being introduced to the signal. Provided this noise falls outside of the frequency range of the primary and secondary peaks, or follows a fixed, continuous frequency (Ie mains noise at 50Hz)

To begin, we can look at the signal in the time-domain. For this particular signal, this can allow us to appreciate why we can't just rely on the time domain and need to further our analysis to the frequency domain:

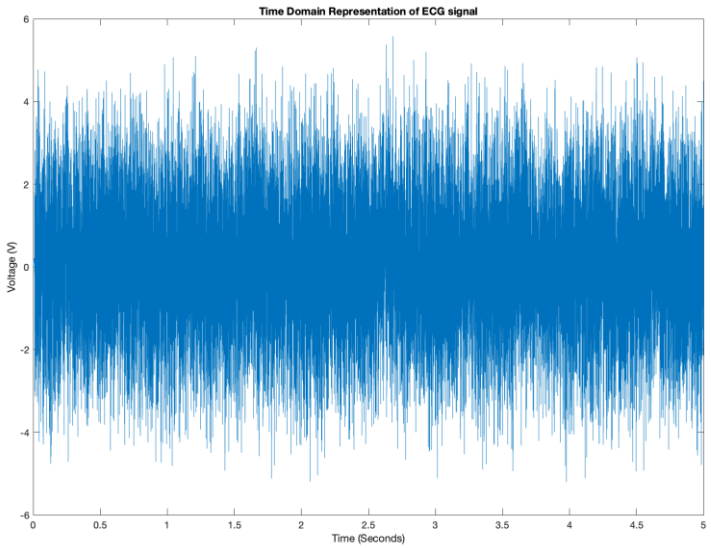

Figure 1 Noise Corrupt ECG signal in the time domain.

Figure 1 clearly shows how this signal would be very challenging to interpret, especially when most of the automated analysis around ECG's is done in the time domain (R-Peak detection, HR, HRV ect)

We can plot a spectrogram of the above signal, to investigate how the frequency components change with respect to time. This has been done in MATLAB using the spectrogram.m function:

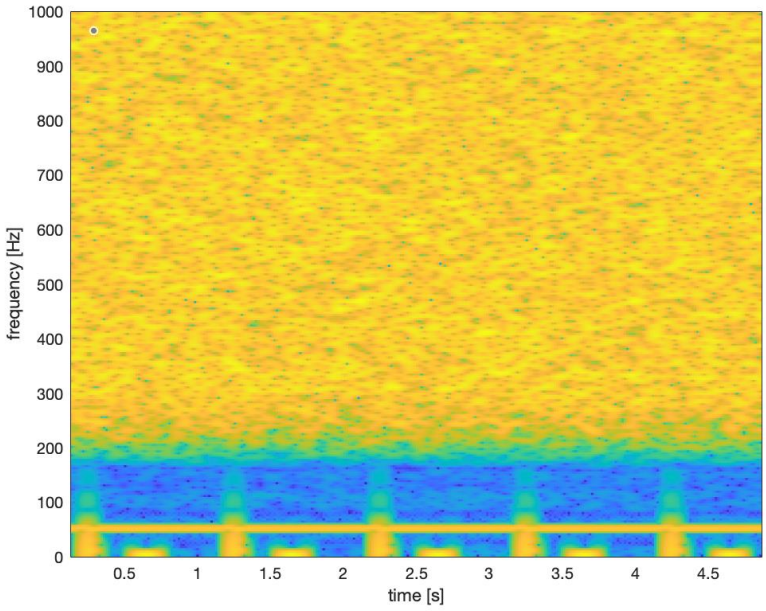

Figure 2 Spectrogram of the ECG signal. Showing how the frequency components vary over time,

From figure 2, we can easily identify the spurious components that have been introduced into the signal. The periodic peaks we can see are the primary and secondary waves that we are looking to investigate, and so we need to maintain these as much as possible.

Now, we can see the signal is clearly corrupt with a high power of high frequency noise. This noise falls outside the frequency range of our primary and secondary waves and so should not present much of a challenge to remove, a low-pass filter with a specified cut-off frequency should be satisfactory in removing this type of noise.

We can also see a continuous line around the 50Hz mark, this not a periodic signal and so is likely not part of our desired ECG signal. The fact that is fixed at 50Hz suggests that this is powerline interference noise, a very common type of noise caused by mains interference. This noise type overlaps with the frequency band of our desired ECG waveforms, and so may present a higher level of difficulty to remove. However there have been a range of algorithms that have been shown to remove mains noise without significantly effecting the original ECG signal, notable an adaptive notch filter.

3 Q2 - Identify and motivate a suitable approach to remove the noise.

We have identified 2 primary noise components contained within this signal. As such, we can remove all the noise using 2 digital filters. We need to develop an algorithm that has the following two objectives:

1) Applies a suitable filter that does not add distortion to the original ECG waveform and removes the 50Hz noise.
2) Applies a suitable low-pass filter that does not distort the original ECG and removes the noise above ~190Hz.

Since 50Hz noise overlaps with the frequency range of our primary and secondary waves under investigation, this will likely prove to be tricky, however there have been methods to reduce this type of noise done in previous literature that will be used as part of our algorithm.

The high frequency noise in our signal falls outside our primary and secondary wave frequency range so this should not be difficult to complete.

The following flow chart outlines the premise of our proposed algorithm:

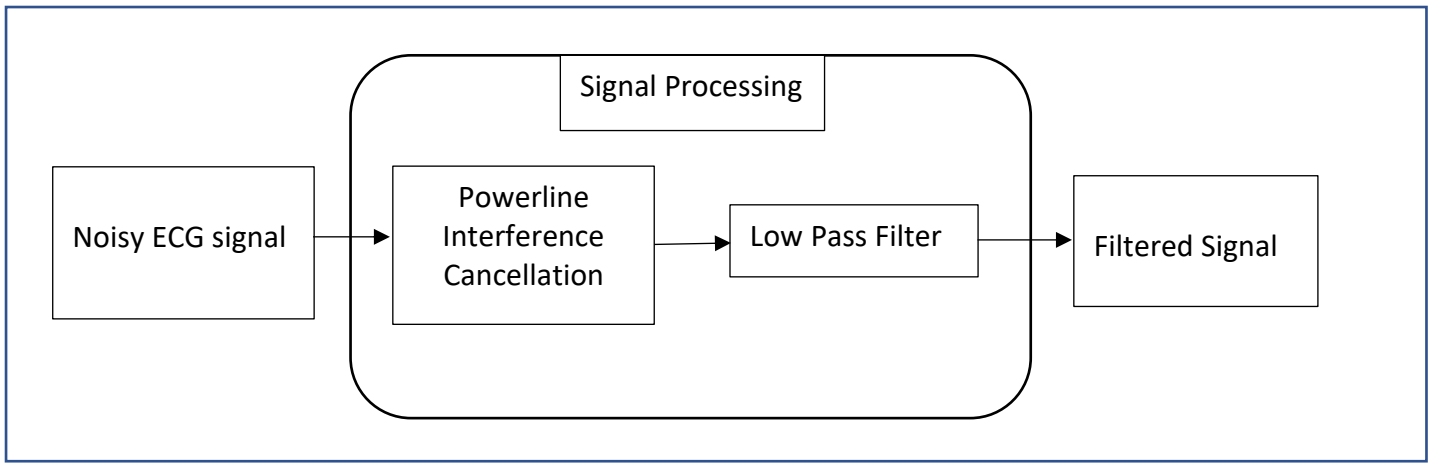

Figure 3 Block Diagram of our proposed algorithm.

3.1 Powerline Removal

The 50Hz noise will be removed based on an algorithm proposed in [1]. This algorithm follows 4 primary steps:

1. Adaptive Notch filter is used to estimate the fundamental frequency of the noise.
2. Harmonics are generated by using discrete-time oscillators.
3. Amplitude and Phase of each harmonic are estimated through a recursive least squares algorithm.
4. Estimated interference is subtracted from the data.

Some MATLAB code was made available for this algorithm in the reference paper, however in this report, we will make modifications to better filter the 50Hz noise from ECG recordings.

One benefit of this algorithm is that it will locate the frequency band of the powerline interference automatically, and so will minimise any error associated with assuming a fixed 50Hz from the spectrogram in figure 2.

Notch filtering has been widely used to reject the pre-determined frequency components (50Hz) to avoid adding any distortion into the ECG signal, however can prove to be ineffective when the frequency varies. A solution to this is to use an adaptive filter which can track the variations in the frequency, phase and amplitude of the powerline interference.

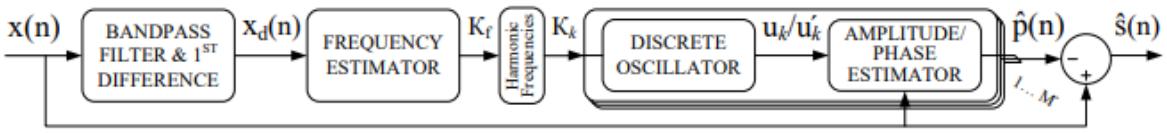

Figure 4 Block Diagram of 50Hz cancellation algorithm [1]

Where x(n) is our original noise contaminated signal, p(n) is our estimated noise content in the time domain, and s(n) is our reconstructed signal with powerline noise removed.

A high level overview on how our MATLAB function will work is as follows:

1. Parameter initialisation.
2. Perform a 4th order Bandpass IIR filter to enhance the fundamental harmonic of the interference. The pass-band has been changed in this report as the we know the powerline frequency is 50Hz from figure 2. The pass-band is set to 45-55Hz to deal with worst-case variations in the interference.
3. Apply a lattice adaptive notch filter (ANF) based frequency estimator to estimate the desired frequencies. [2].
4. Harmonic sinusoids are generated through the use of discrete-time oscillators and subsequently, the amplitudes and phases of the harmonics are estimated to match the corresponding interference.
5. Amplitudes and phases are subtracted from the original signal.

4 Low Pass Filter

After the powerline interference has been removed following section 3, the signal will be passed through a low-pass filter to remove the high frequency noise content we have seen in figure 2. We can see that this noise is present at frequencies greater than 190Hz, and so a low-pass filter with a cutoff at 120Hz should remove this noise source. The original ECG signal should not be affected as this noise lies outside the range of the periodic ECG signal. We will develop a low-pass FIR filter.

5 Design of the Filters.

This section explores the design of the 2 filters.

5.1 Adaptive Powerline Removal

The first filter we design is a bandpass filter in order to enhance the fundamental frequencies of the interference. This was done using the freqz.m MATLAB function after obtaining the filter coefficients:

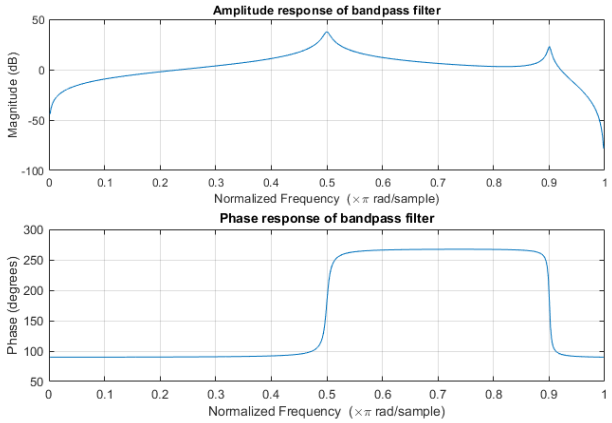

Figure 5 Bandpass responses (Magnitude and Phase)

There is no need to plot a time-domain signal here as the intention of this filter is to enhance the fundamental interference frequencies, rather than remove noise.

After applying the adaptive powerline cancellation mentioned in the previous section on our noisy signal, we can plot a new spectrogram to validate that the 50Hz noise has been removed effectively.

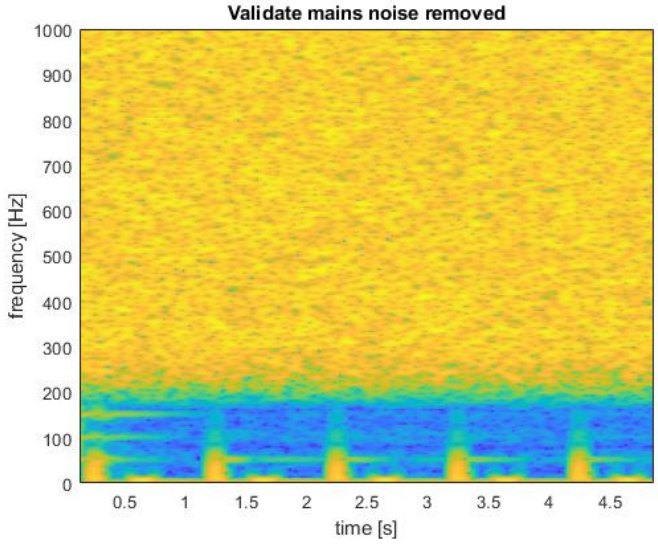

Figure 6 Validation that our powerline removal algorithm has worked correctly.

We can see in figure 6 that the 50Hz noise has almost entirely been removed from the signal, validating that the method we followed is correct. Plotting the time-domain signal we get:

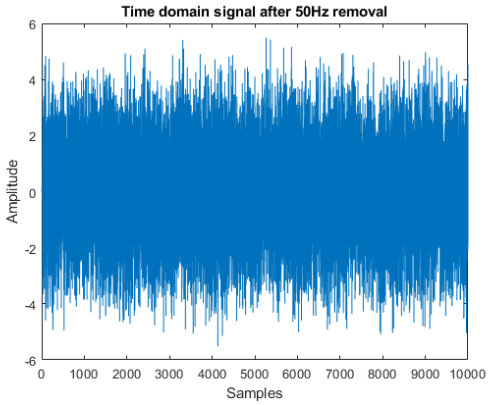

Figure 7 Time domain signal after 50Hz removal

We can see that the signal is still corrupt with a significant amount of noise, making the signal impossible to interpret in the time-domain. As such, we need another filter.

Now we are just left with High frequency noise above ~190Hz, we will remove this with an FIR lowpass filter with a cut-off set at 120Hz (To account for stopband attenuation). Applying this filter, we get the following amplitude response and spectrogram:

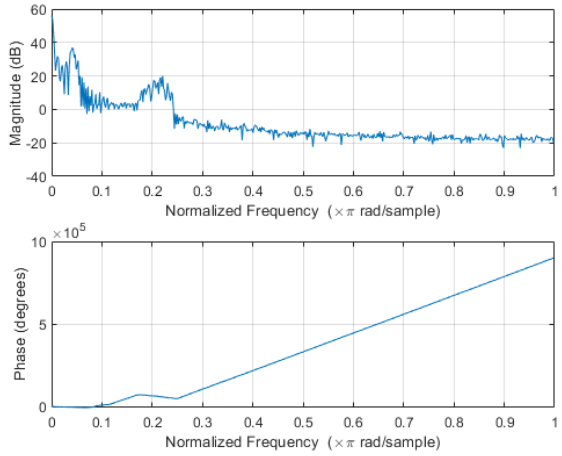

Figure 8 Filter response from lowpass.

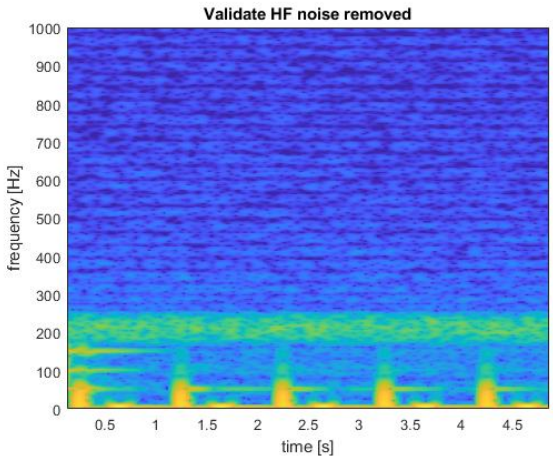

Figure 9 Final Spectrogram after 2 filtering stages

We can see from figure 7, that most of the noise has now been removed from the noisy signal. There is still some HF noise present but this is insignificant and does not affect our ability to interpret the interpeak distance and amplitude as required.

The final time domain representation can be seen below:

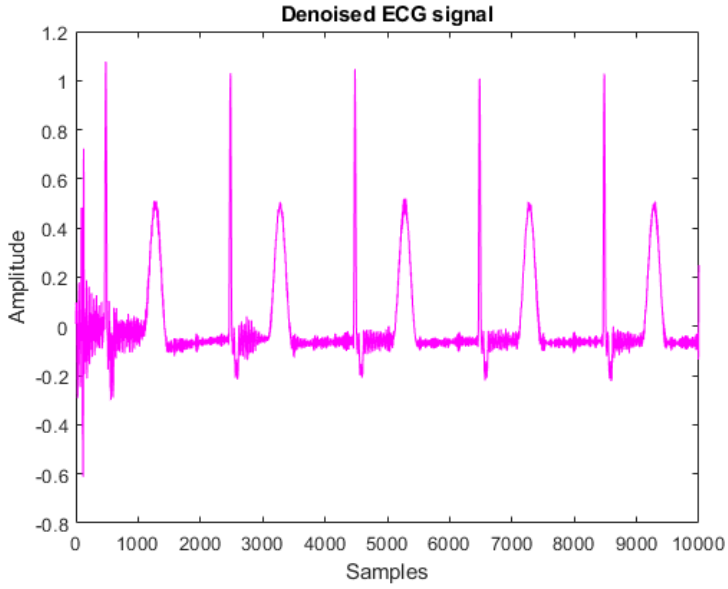

Figure 8 Denoised ECG signal.

6 Q4 Estimations of interpeak interval and amplitude

From visual inspection of the ECG signal in figure 9, we can state the required information:

Interpeak Interval	Primary Wave Height	Secondary Wave Height
800 Samples = 400ms	1.07V	0.55V

Note, samples was converted to seconds through time(s) = numOfSamples / Fs.

7 References

[1] M. R. Keshtkaran and Z. Yang, "A fast, robust algorithm for power line interference cancellation in neural recording," J. Neural Eng., vol. 11, no. 2, p. 026017, Apr. 2014 arXiv:1402.6862

[2] Cho, Nam Ik and Sang Uk Lee. “On the adaptive lattice notch filter for the detection of sinusoids.” *IEEE Transactions on Circuits and Systems Ii: Analog and Digital Signal Processing* 40 (1993): 405-416.

8 Appendix

Spectrogram Plotting

```
% Load Test Signal
[testSignal, Fs] = audioread('C:\b-secur\Digital Signal Processing
Assignment\Digital Signal Processing Assignment\signals/signal23.wav');
timeSeries = 1/Fs : 1 / Fs : numel(testSignal)/Fs;

% Plot time domain signal
figure(1);

plot(timeSeries, testSignal);
xlabel("Time (Seconds)");
```

```
ylabel("Voltage (V)");
title('Time Domain Representation of ECG signal');


AZ = 0;                 % azimuth
EL = 90;                % elevation
TH = -100;              % amplitude threshold (dB)
Ns = length(testSignal);
dur = Ns/Fs;

% Perform the short-time Fourier Transform.
[S, F, T, ~] = spectrogram(testSignal, blackman(512), 480, 512, Fs);

% Set the absolute values of the spectrogram.
A = (abs(S));

% Find the max value
Amax = max(max(A));

% Devide A by AMax
A = A/Amax;

% Set AL as the log of A
AL = 20*log10(A);


% Find values of AL that are less that our threshold
I = find(AL < TH);

% Set these values as being our threshold
AL(I) = TH;

% PLot
figure(2);
xlabel('time [s]'); ylabel('signal');
surf(T,0.001*F*1000,AL,'edgecolor','none');
grid;
axis tight;
xlabel('time [s]'); ylabel('frequency [Hz]');
box on;
set(gca,'BoxStyle','full');
set(gca,'XAxisLocation','origin');
view(AZ,EL);
```

Adaptive Powerline Filter

```
function processedSignal = adaptiveNotchFilter(signal, fs)
% This function will perform adaptive notch filtering as detailed in
% [1].
%
% The four stages are:
% 1. An adaptive notch filter is used to estimate the fundamental frequency
%    of the noise. If this is known, this can also be input.
% 2. Based on the estimated frequency, harmonics are generated using
%    discrete-time oscillators.
% 3. The amplitude and phase of each harmonic are then estimated using a
%    modified recursive least squares algorithm.
% 4. The estimated interference is subtracted from the signal.
```

```
DEFAULT_BANDPASS = [45, 55]; % [Hz]
FILTER_ORDER = 4;
NUM_OF_HARMONICS_TO_REMOVE = 3;

% Define in-line functions for converting inputs. [Eq 15]
convertToForgettingFactor = @(x) exp(log(0.05) / (x * fs + 1));
convertToPoleRadii = @(x) (1 - tan(pi * x / fs)) / (1 + tan(pi * x / fs));

% Pre-allocate.
nSamples = numel(signal); % Ref: N
processedSignal = zeros(size(signal)); % Ref: S^(n)

% Zero mean the signal.
signal = signal(:);

% Accumulate mean
runningSum = 0;

for iSample = 1 : nSamples

    runningSum = runningSum + signal(iSample);
    signal(iSample) = signal(iSample) - (runningSum / iSample);

end

% Convert inputs to pole radii.
alphaF = convertToPoleRadii(30); % Ref: a_f
alphaInf = convertToPoleRadii(0.01); % Ref: a_inf

% Convert inputs to forgetting factors.
alphaSt = convertToForgettingFactor(3); % Ref: a_st
lambdaF = convertToForgettingFactor(0.01); % Ref: ^_f
lambdaInf = convertToForgettingFactor(4); % Ref: ^_inf
lambdaSt = convertToForgettingFactor(1); % Ref: ^_st

% Set the smoothing parameter (which has a fixed cut-off of 90 Hz).
gammaSmooth = convertToPoleRadii(0.5 * min(90, fs / 2)); % Ref: y

% Set up the phase / amplitude forgetting factor (might be multiple).
lambdaA = convertToForgettingFactor(0.5);
lambdaA = lambdaA * ones(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: ^_a

% Initialise lattice variables.
kappaF = 0; % Ref: k_f
kappaK = zeros(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: k_k
latticeC = 5; % Ref: C
latticeD = 10; % Ref: D
fn1 = 0;
fn2 = 0;

% Initialise the oscillator.
ukp = ones(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: u_k
uk = ones(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: u'_k

% Initialise RLS parameters.
rlsR1 = 10 * ones(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: r1_k
rlsR4 = 10 * ones(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: r4_k
rlsA = zeros(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: b^_k
```

```
rlsB = zeros(1, NUM_OF_HARMONICS_TO_REMOVE); % Ref: c^_k

% IIR bandpass.
% Phase distortion doesn't matter here as we are only using this signal for
% frequency estimation.
filterCutOffs = DEFAULT_BANDPASS;

% Design the filter and convert to SOS format. Note that filter order is
% divided by 2 because butter designs bandpass filters of order 2n.
[z, p, k] = butter(FILTER_ORDER / 2, filterCutOffs / (fs / 2), 'bandpass');
sosParams = real(zp2sos(z, p, k));

% Plot amplitude response
figure();
freqz(z, p);
title("Bandpass Response");

% Filter and differentiate the signal. [Eq 4]
filteredSignal = sosfilt(sosParams, signal); % Ref: x_f
filteredSignal = [0; diff(filteredSignal)]; % Ref: x_d

% Loop over each sample.
for iSample = 1 : nSamples

    % Compute the output of the lattice filter. [Eq 6]
    fn = filteredSignal(iSample) + kappaF * (1 + alphaF) * fn1 - alphaF * fn2;

    % Compute the updated frequency estimation.
    latticeC = lambdaF * latticeC + (1 - lambdaF) * fn1 * (fn + fn2);
    latticeD = lambdaF * latticeD + (1 - lambdaF) * 2 * fn1 ^ 2;
    kappaT = latticeC / latticeD; % Ref: k_t

    % Limit kappaT.
    if kappaT > 1

        kappaT = 1;

    elseif kappaT < -1

        % This shouldn't be possible but is included here to ensure filter
        % stability.
        kappaT = -1;

    end

    % Update kappaF.
    kappaF = gammaSmooth * kappaF + (1 - gammaSmooth) * kappaT;

    % Update the previous lattice values.
    fn2 = fn1;
    fn1 = fn;

    % Update bandwidths and forgetting factors. [Eq 7]
    alphaF = alphaSt * alphaF + (1 - alphaSt) * alphaInf;
    lambdaF = lambdaSt * lambdaF + (1 - lambdaSt) * lambdaInf;

    % Remove harmonics.
    instantaneousError = signal(iSample); % Ref: e
```

```
    % Loop over each harmonic.
    for jHarmonic = 1 : NUM_OF_HARMONICS_TO_REMOVE

        % Compute harmonic. [Eq 9]
        if jHarmonic == 1

            kappaK(jHarmonic) = kappaF;

        elseif jHarmonic == 2

            kappaK(jHarmonic) = 2 * kappaF ^ 2 - 1;

        else

            kappaK(jHarmonic) = 2 * kappaF * kappaK(jHarmonic - 1) ...
                - kappaK(jHarmonic - 2);

        end

        % Discrete oscillator.
        s1 = kappaK(jHarmonic) * (uk(jHarmonic) + ukp(jHarmonic));
        s2 = ukp(jHarmonic);
        ukp(jHarmonic) = s1 - uk(jHarmonic);
        uk(jHarmonic) = s1 + s2;

        % Compute gain control.
        gainControl = 1.5 - (ukp(jHarmonic) ^ 2 - uk(jHarmonic) ^ 2 * ...
            (kappaK(jHarmonic) - 1) / (kappaK(jHarmonic) + 1)); % Ref: G

        % Limit gain control.
        if gainControl < 0

            gainControl = 1;

        end

        % Scale by gain control.
        ukp(jHarmonic) = gainControl * ukp(jHarmonic);
        uk(jHarmonic) = gainControl * uk(jHarmonic);

        % Amplitude and Phase estimation via recursive least squares. [Eq 13]
        hk = rlsA(jHarmonic) * ukp(jHarmonic) + rlsB(jHarmonic) * uk(jHarmonic);
        instantaneousError = instantaneousError - hk;
        rlsR1(jHarmonic) = lambdaA(jHarmonic) * rlsR1(jHarmonic) + ukp(jHarmonic)
^ 2;
        rlsR4(jHarmonic) = lambdaA(jHarmonic) * rlsR4(jHarmonic) + uk(jHarmonic) ^
2;
        rlsA(jHarmonic) = rlsA(jHarmonic) + instantaneousError * ukp(jHarmonic) /
rlsR1(jHarmonic);
        rlsB(jHarmonic) = rlsB(jHarmonic) + instantaneousError * uk(jHarmonic) /
rlsR4(jHarmonic);

    end

    % The filter error is the remaining true signal. [Eq 14]
    processedSignal(iSample) = instantaneousError;

end
```

Low Pass Filter

```
function filteredSignal = removeHFnoise(signal, fs)
% This function will remove the HF noise seen in the signal.

%% CONSTANTS
CUT_OFF = 120;

y = lowpass(signal, CUT_OFF, fs);

filteredSignal = filter(1, 1, y);

freqz(y)

end
```