Signal and System Computer homework

Dr.Hamid Behroozi

department : Electrical Engineering

Amirreza Velae 400102222

Computer homework
Report

July 2, 2023

Signal and System Computer homework

Computer homework

Amirreza Velae 400102222

ECG Signal

IIR and FIR Filter

IIR and FIR filters are two types of digital filters commonly used in signal processing.

Infinite Impulse Response

filters are digital filters that use feedback, meaning that the output of the filter is fed back into the input. This creates a recursive relationship between the input and output, which allows *IIR* filters to have very steep roll-off characteristics and to be efficient in terms of computation. However, *IIR* filters can be unstable if not designed properly, and they can introduce phase distortion.

Finite Impulse Response

filters, on the other hand, do not use feedback. They operate by convolving the input signal with a finite impulse response, which is a sequence of coefficients that define the filter's frequency response. *FIR* filters are typically more stable than *IIR* filters, and they introduce less phase distortion. However, they require more computation to implement, especially for filters with long impulse responses.

Choice of IIR and FIR

In general, the choice between *IIR* and *FIR* filters depends on the specific requirements of the application. *IIR* filters are often used in applications where steep roll-off characteristics are necessary, such as in audio equalizers and notch filters. *FIR* filters are often used in applications where phase distortion must be minimized, such as in digital communications and image processing.

Best choice for ECG and why?

First and Second Signal Graph

I used the following function to plot signal itself and generate fourier transform of it and plot it too:

```
function plot_ecg(name,ecg,sampling_rate)
    n = length(ecg);
    t = n/sampling_rate;
    x = 0:1/sampling_rate:t-1/sampling_rate;
    figure('units','normalized','outerposition',[0 0 1 1])
    plot(x,ecg);
    xlim([0 t]);
    xlabel('S',Interpreter='latex',Color=[0.25, 0.25, 0.25],FontSize=17);
    ylabel('mV',Interpreter='latex',Color=[0.25, 0.25, 0.25],FontSize=17)
    title(append(name,' Time Domain'),Interpreter='latex',Color=[0.15, 0.15,
    0.15],FontSize=28)
    grid on
    max_y = max(abs(min(ecg)),max(ecg));
    ylim([-1.5*max_y 1.5*max_y])
```

```
14
15      [f,func_fft] = plot_furier_transform(ecg);
16
17      figure('units','normalized','outerposition',[0 0 1 1])
18      plot(f ,func_fft)
19      xlabel('f',Interpreter='latex',Color=[0.25, 0.25, 0.25],FontSize=17);
20      ylabel('abs',Interpreter='latex',Color=[0.25, 0.25, 0.25],FontSize=17)
21      title(append(name,' Frequency Domain'),Interpreter='latex',Color=[0.15,
        0.15, 0.15],FontSize=28)
22      grid on
23      ylim([1.5*min(func_fft) 1.5*max(func_fft)])
24  end
```

Source Code 1: ECG plot function

Also I used the following function to generate fourier transform of signal:

```
1  function [f,func_fft] = plot_furier_transform(func)
2      func_fft = fft(func);
3      f_m = length(func_fft)/2;
4      f = -f_m:f_m-1;
5      func_fft = abs(fftshift(func_fft));
6  end
```

Source Code 2: Fourier transform function

Result of this function for first and second signal for both time and frequency domain is shown in figures 1, 2, 3 and 4.

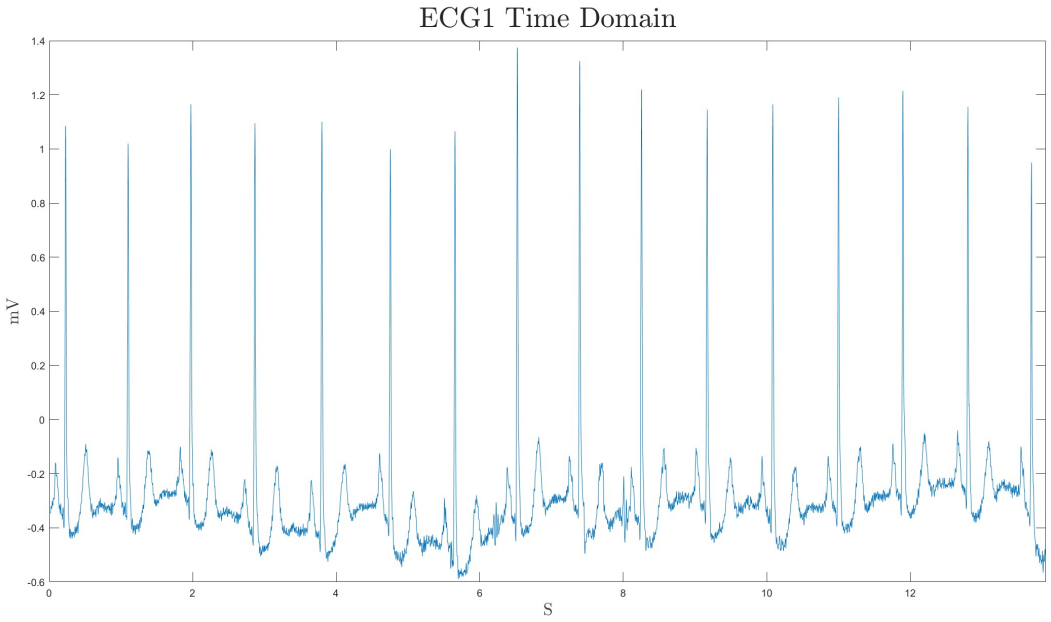

Figure 1: First signal in time domain

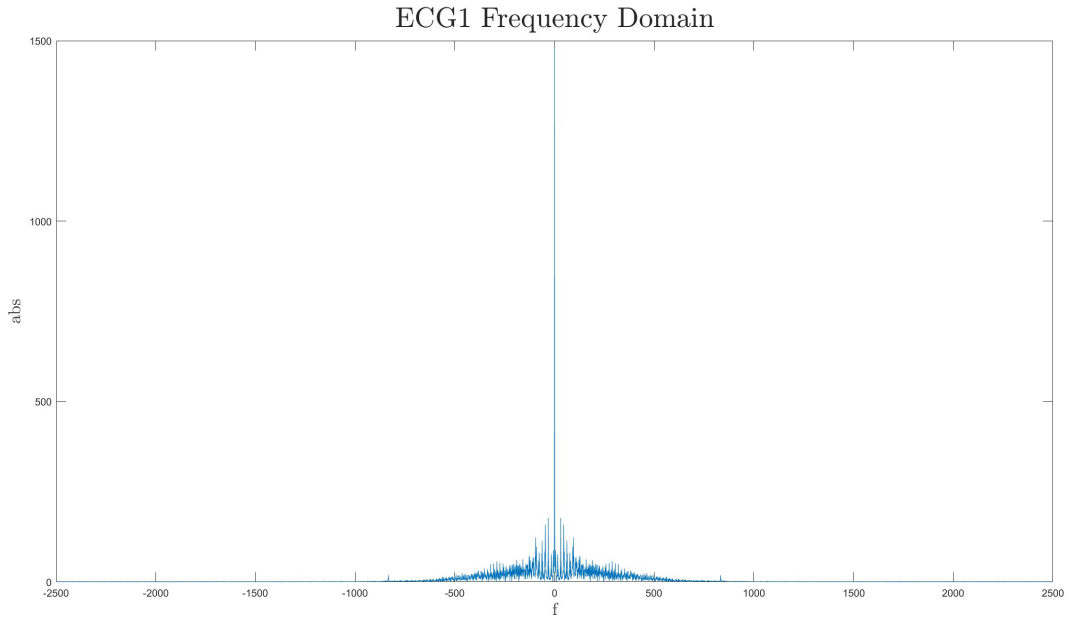

Figure 2: First signal in frequency domain

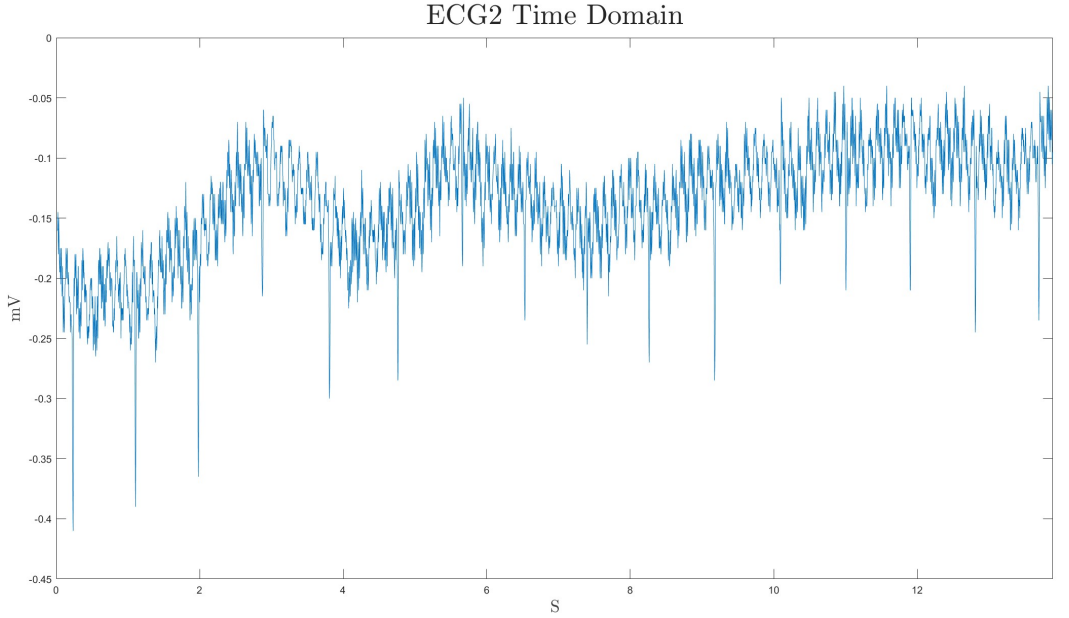

Figure 3: Second signal in time domain

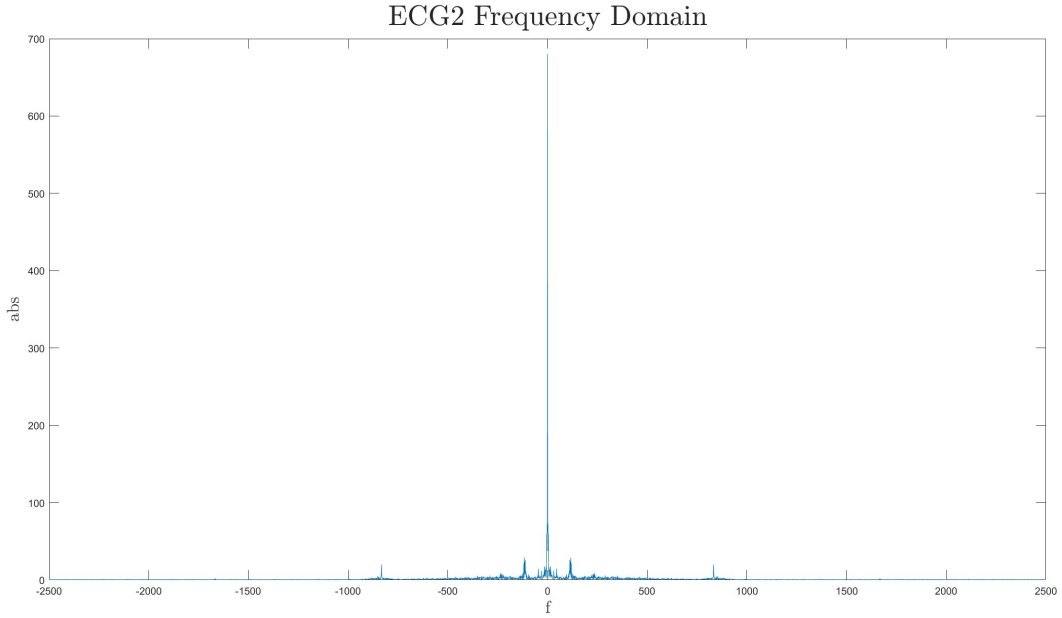

Figure 4: Second signal in frequency domain

Filter urban noise

I used the following function to build filter to remove urban noise:

```
function Hd = Build_BandPass_Filter(fstop1,fstop2,Fs)
    Fpass1 = fstop1-7;          % First Passband Frequency
    Fstop1 = fstop1;            % First Stopband Frequency
    Fstop2 = fstop2;            % Second Stopband Frequency
    Fpass2 = fstop2+7;          % Second Passband Frequency
    Dpass1 = 0.025;             % First Passband Ripple
    Dstop  = 0.001;             % Stopband Attenuation
    Dpass2 = 0.055;             % Second Passband Ripple
    dens   = 20;                % Density Factor

    % Calculate the order from the parameters using FIRPMORD.
    [N, Fo, Ao, W] = firpmord([Fpass1 Fstop1 Fstop2 Fpass2]/(Fs/2), [1 0 ...
                            1], [Dpass1 Dstop Dpass2]);

    % Calculate the coefficients using the FIRPM function.
    b  = firpm(N, Fo, Ao, W, {dens});
    Hd = dfilt.dffir(b);
end
```

Source Code 3: Filter urban noise function

I used the following code to build and plot filter:

```
Fs = 360;
Hd1 = Build_BandPass_Filter(55,65,Fs);
fvtool(Hd1);
```

Source Code 4: Build and plot filter

Phase and magnitude response of this filter is shown in figure 5.

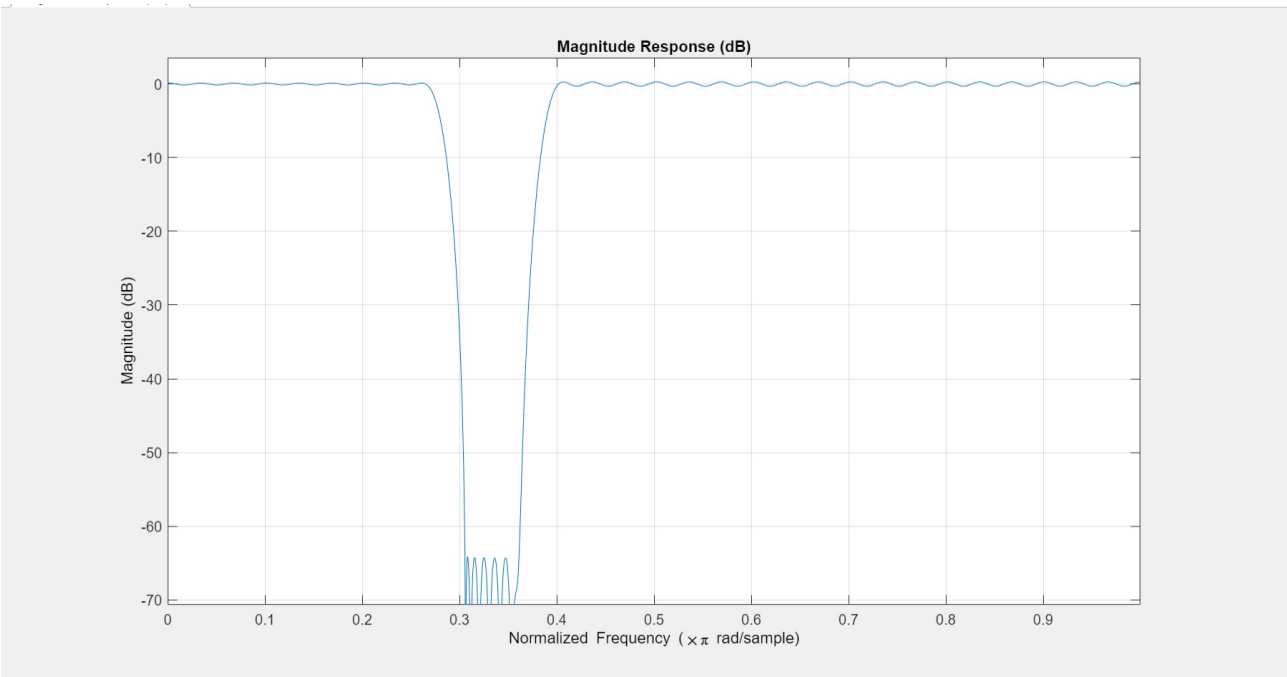

Figure 5: Filter for urban noise

Result of applying this filter to first and second signal is shown in figures 6 and 7.
Also Ecg's in time domain after applying urban noise filter is shown in figures 8 and 9.

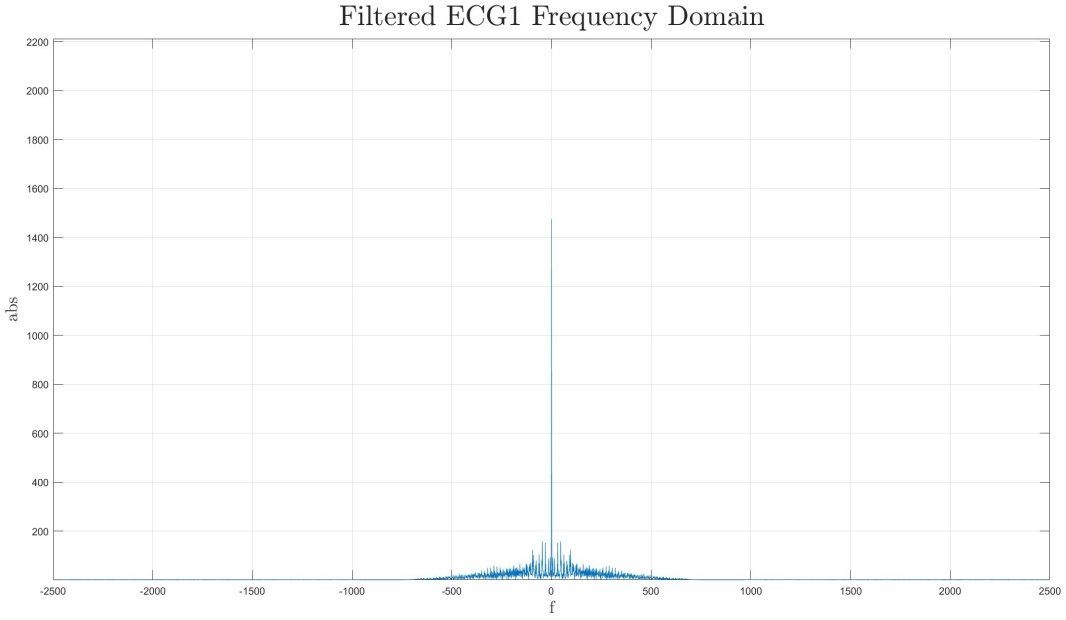

Figure 6: First signal after applying urban noise filter

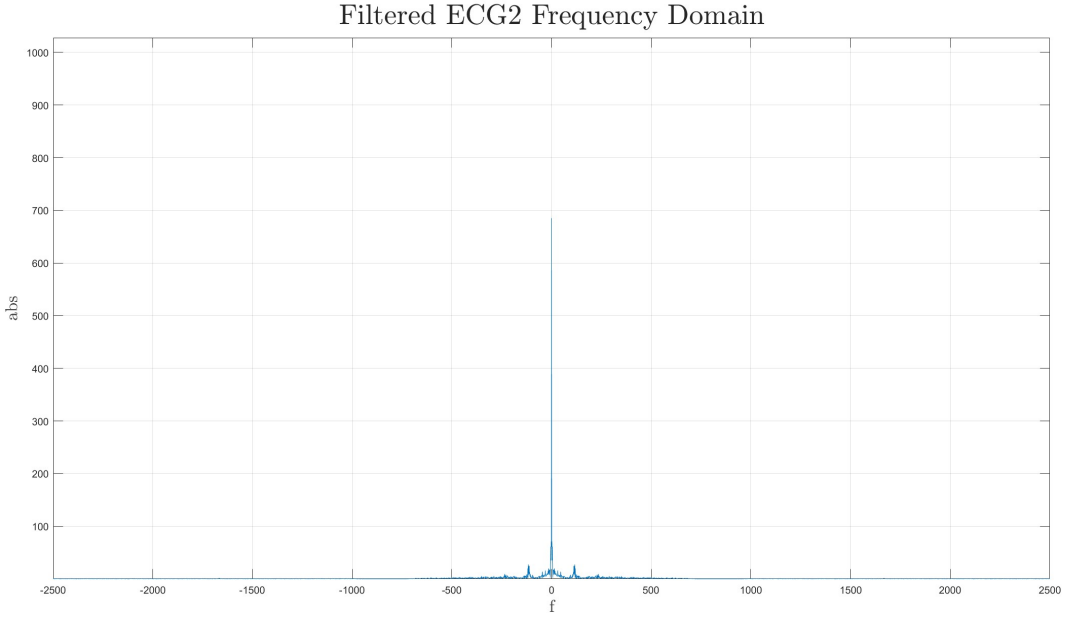

Figure 7: Second signal after applying urban noise filter

Figure 8: First signal after applying urban noise filter in time domain

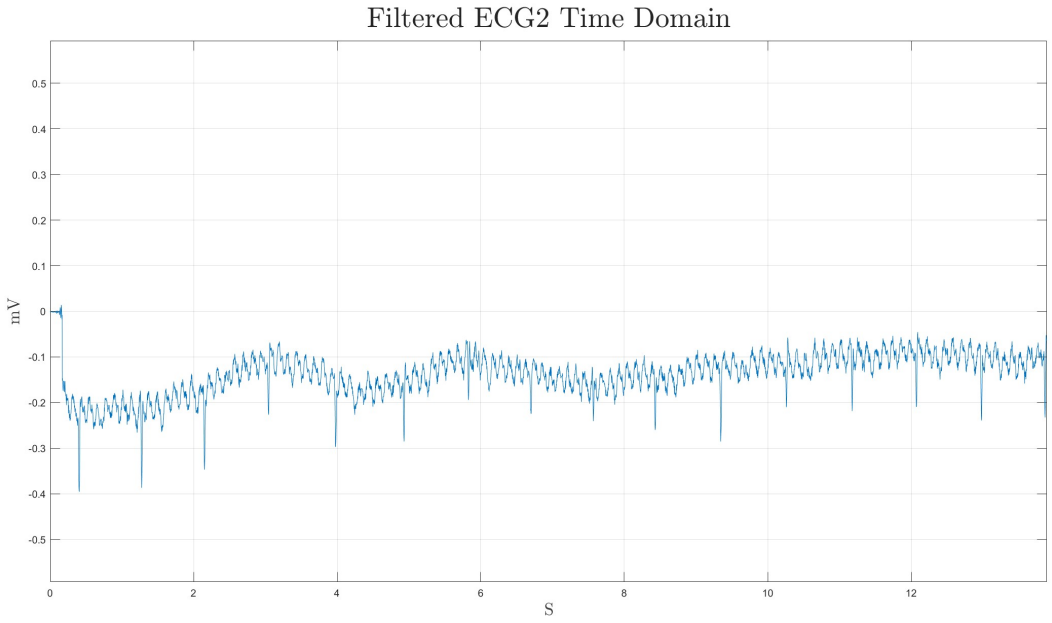

Figure 9: Second signal after applying urban noise filter in time domain

Filter Baseline Wander

Since baseline wander is a low frequency noise, I used a high pass filter to remove it.
I used the following function to build high pass filter to remove baseline wander:

```
function Hd = HighPassFilter(Fstop , Fpass,Fs)
    Dstop = 0.0001;          % Stopband Attenuation
    Dpass = 0.055;           % Passband Ripple
    dens  = 20;              % Density Factor

    % Calculate the order from the parameters using FIRPMORD.
    [N, Fo, Ao, W] = firpmord([Fstop, Fpass]/(Fs/2), [0 1], [Dstop, Dpass]);

    % Calculate the coefficients using the FIRPM function.
    b  = firpm(N, Fo, Ao, W, {dens});
    Hd = dfilt.dffir(b);

end
```

Source Code 5: Filter baseline wander function

I used the following code to build and plot filter:

```
Hd2 = HighPassFilter(1 , 3,Fs);
fvtool(Hd2);
```

Source Code 6: Build and plot filter

Phase and magnitude response of this filter is shown in figure 10.

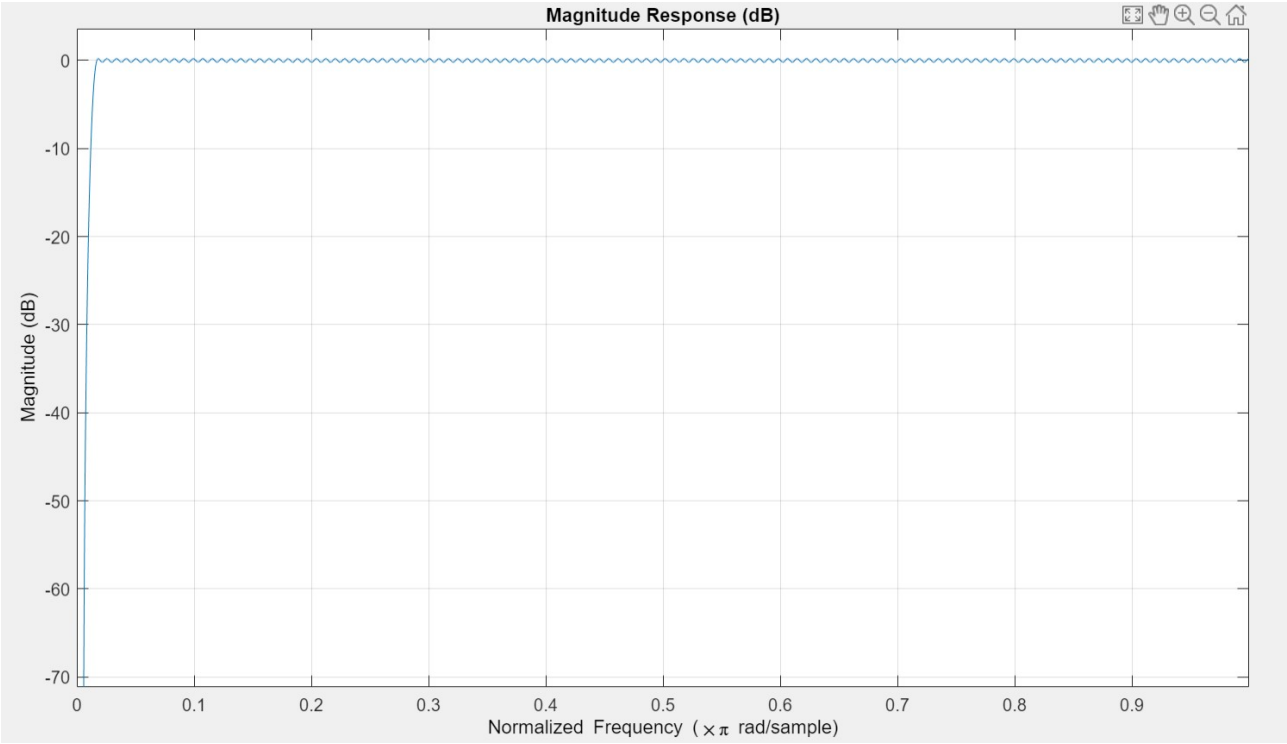

Figure 10: Filter for baseline wander

Result of applying this filter to first and second signal is shown in figures 11 and 12.
Also Ecg's in time domain after applying baseline wander filter is shown in figures 13 and 14.

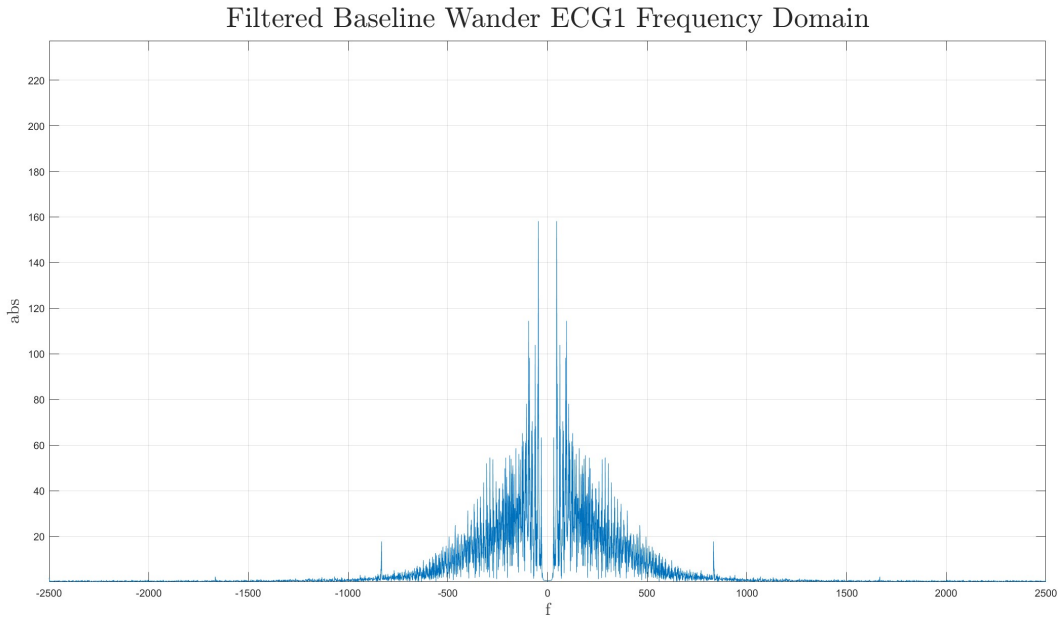

Figure 11: First signal after applying baseline wander filter

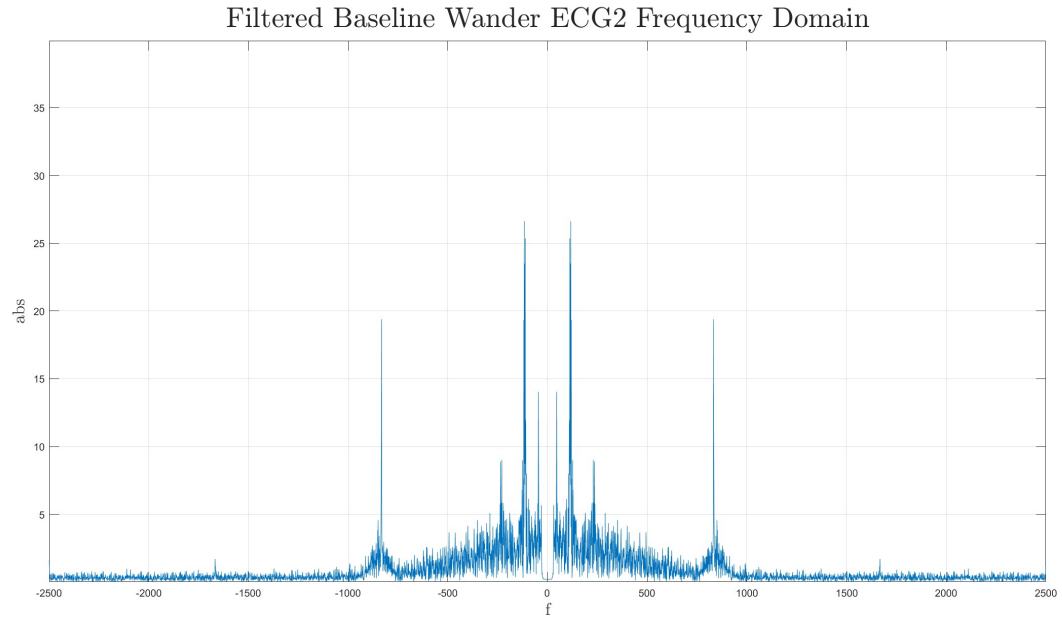

Figure 12: Second signal after applying baseline wander filter

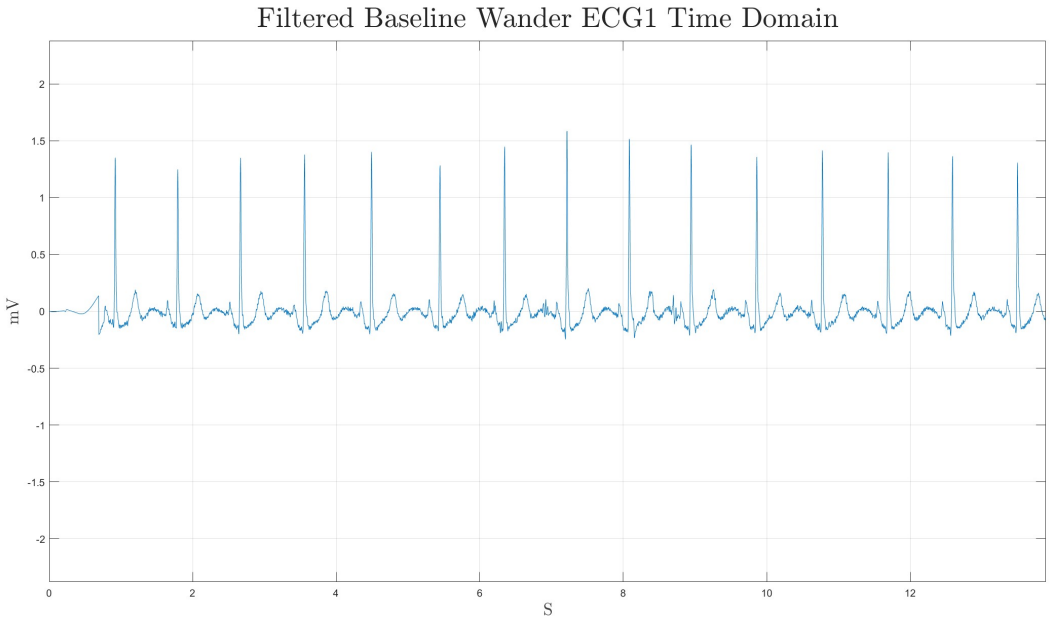

Figure 13: First signal after applying baseline wander filter in time domain

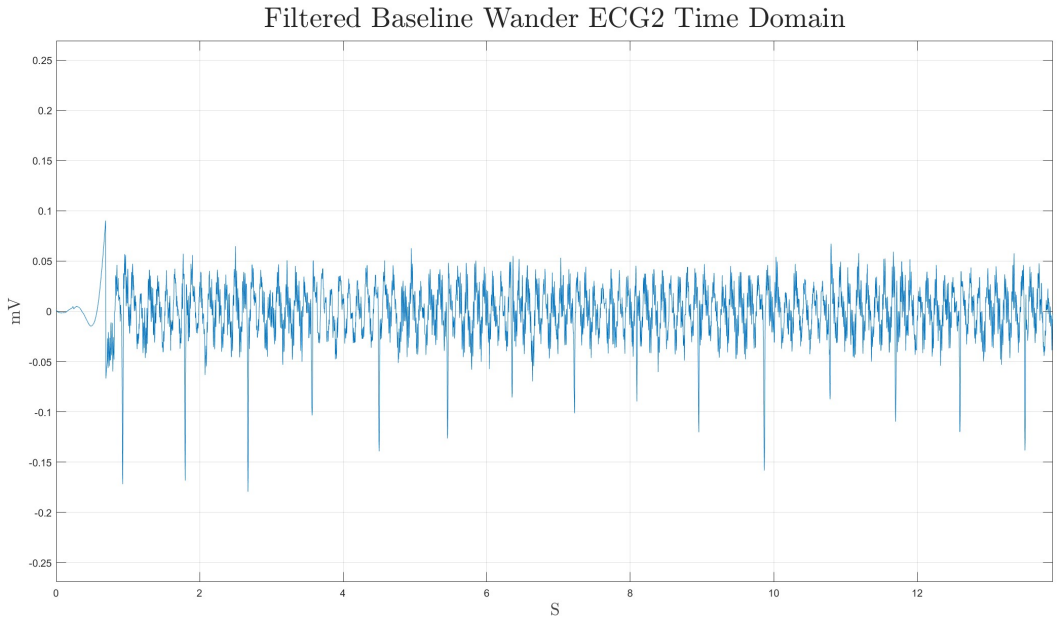

Figure 14: Second signal after applying baseline wander filter in time domain

Filter Both Noise

Since we want to remove both noise, I combined two filters and used them to remove both noise.
I used the following code to combine two filters:

```
Hd3 = dfilt.cascade(Hd1, Hd2);
fvtool(Hd3);
```

Source Code 7: Combine two filters

Phase and magnitude response of this filter is shown in figure 15. Rsult of applying this filter to

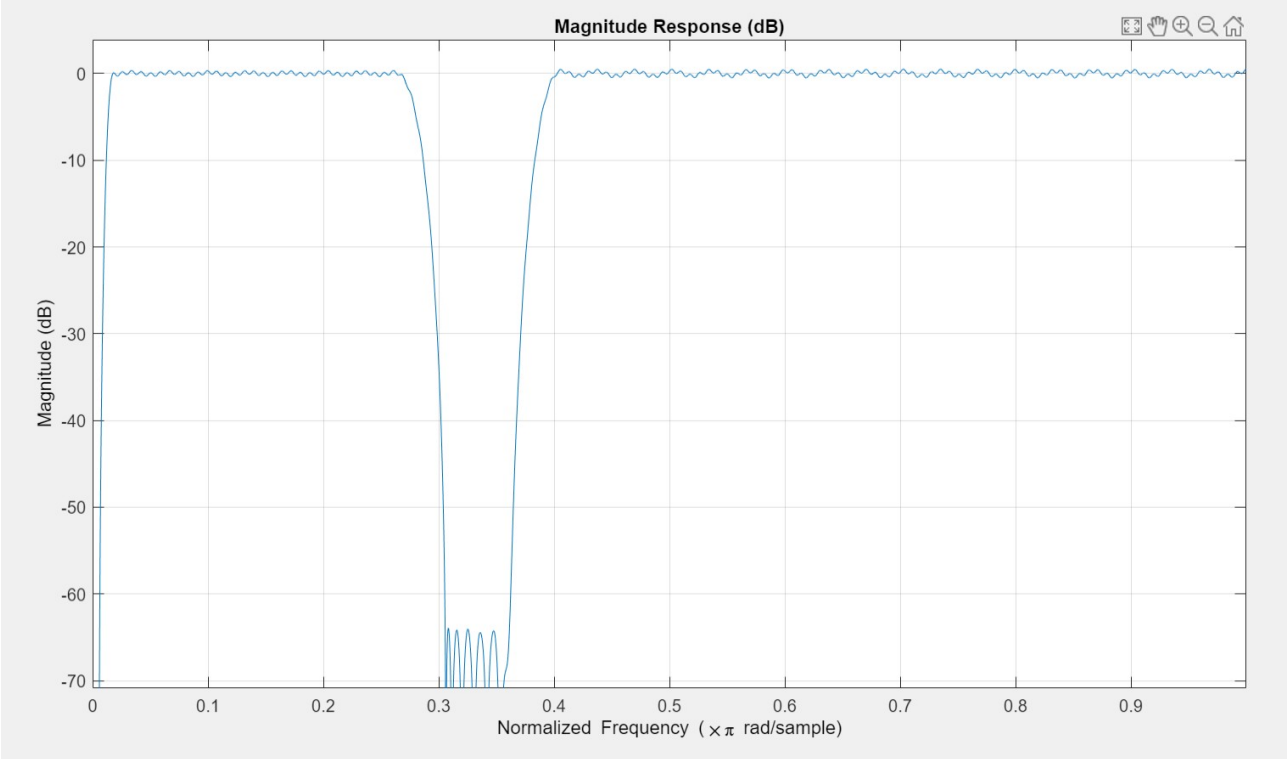

Figure 15: Filter for both noise

first and second signal is shown in figures 16 and 17. Also Ecg's in time domain after applying both noise filter is shown in figures 18 and 19.

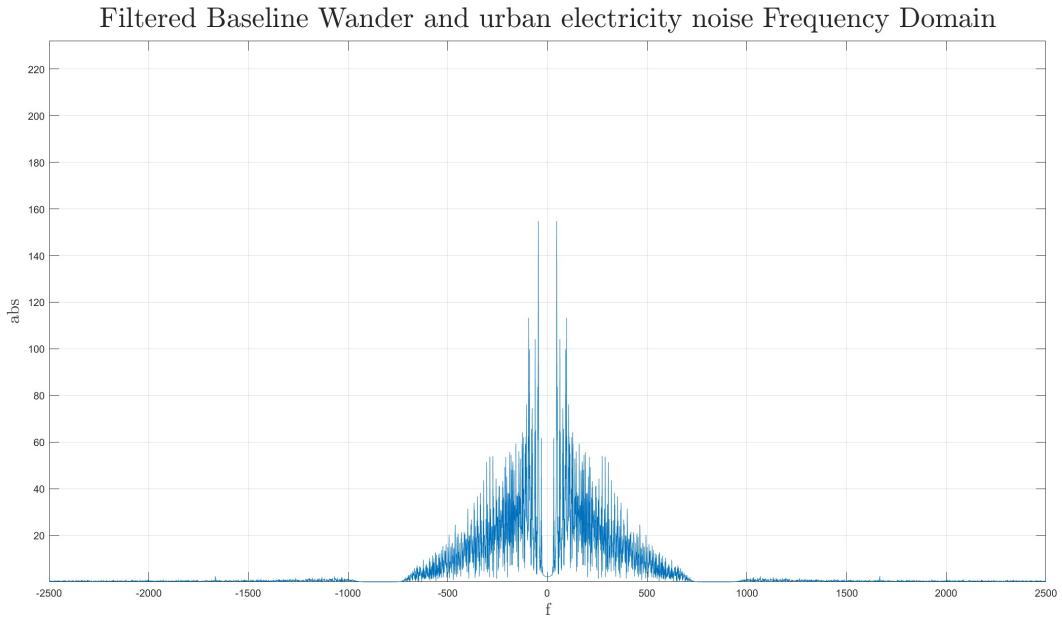

Figure 16: First signal after applying both noise filter

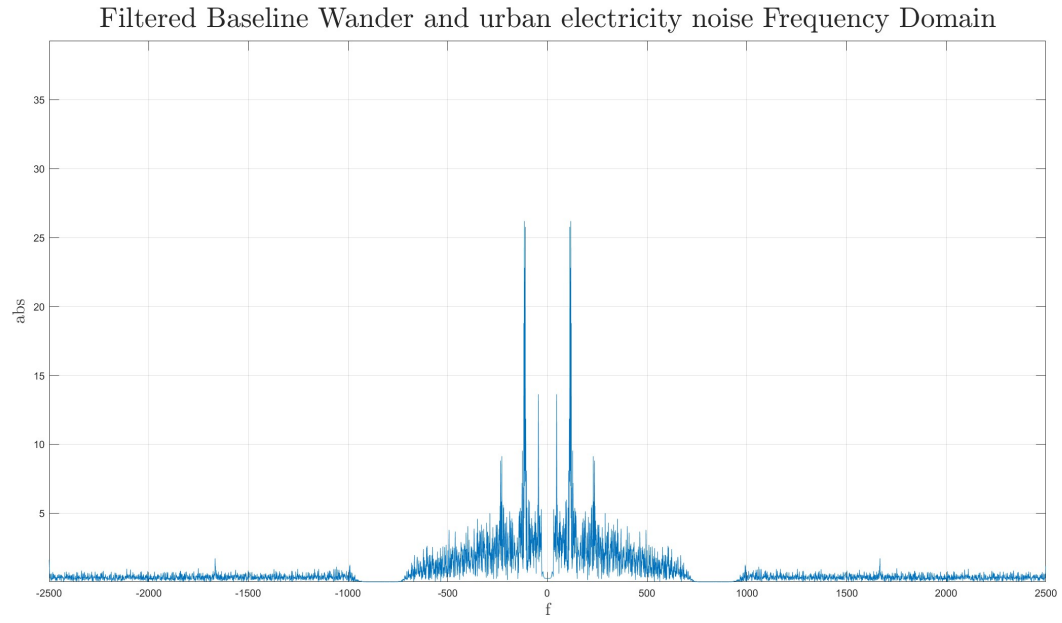

Figure 17: Second signal after applying both noise filter

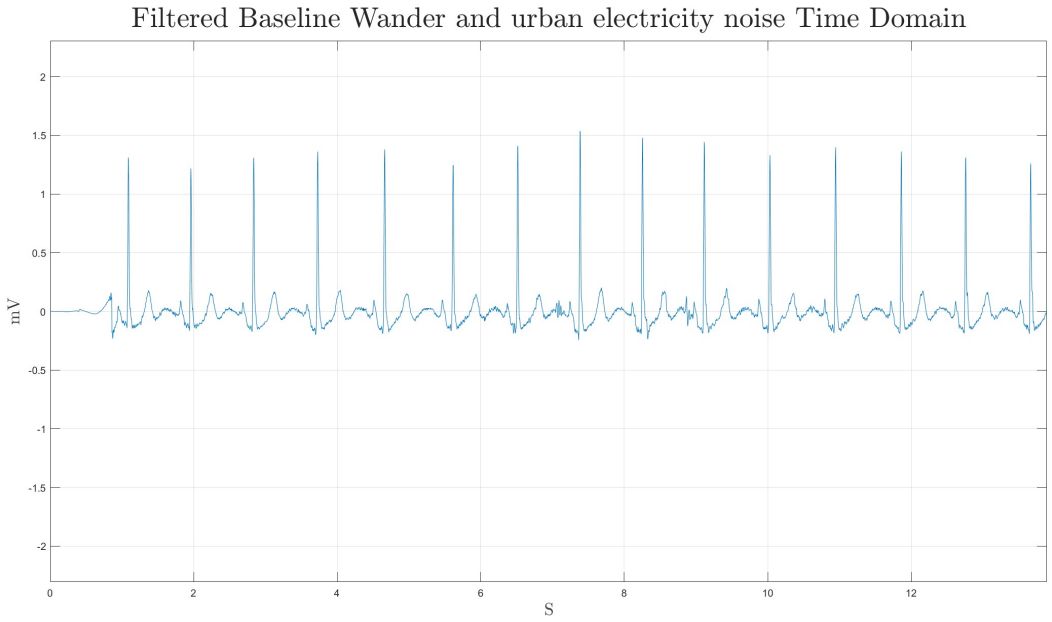

Figure 18: First signal after applying both noise filter in time domain

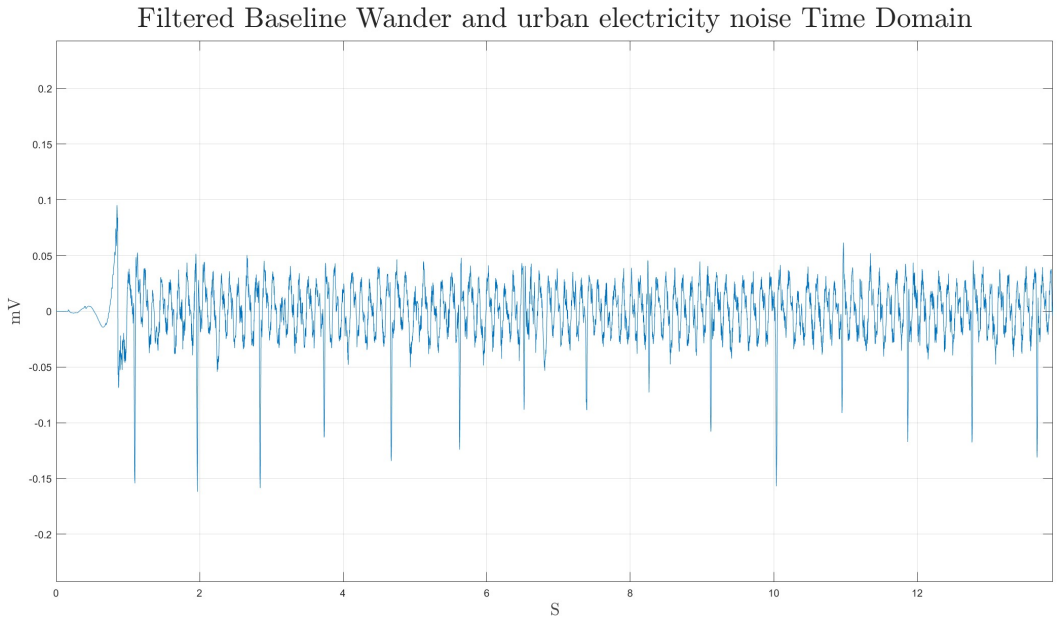

Figure 19: Second signal after applying both noise filter in time domain

JPG Compression

Describe how jpg compression works

Describe bmp format

BMP is a standard format used by Windows to store device-independent and bitmap images. The number of bits used to represent each pixel determines the number of colors that can be displayed.
The BMP file format is capable of storing two-dimensional digital images both monochrome and color, in various color depths, and optionally with data compression, alpha channels, and color profiles.
The BMP file format was designed for compatibility with the IBM PC display standard, EGA, and the VGA display system.

If we take picture with a 108 Megapixel camera and save it in BMP format, how much space does it take?

Assuming the image has a resolution of 12000×9000 pixels (which is a common resolution for 108 MP cameras), and is saved with a color depth of 24 bits per pixel (which is typical for BMP images), the uncompressed size of the image would be:

$$12000 \times 9000 \times 24 = 2,592,000,000 \text{ bits}$$
$$\frac{2,592,000,000}{8} = 324,000,000 \text{ bytes}$$
$$\frac{324,000,000}{1,048,576} = 308.58 \text{ megabytes}$$

So, a 108 MP BMP image with these settings would take approximately 308.58 megabytes to store.

What is DCT(Direct Cosine Transform)?

The discrete cosine transform (DCT) is a technique for converting a signal into elementary frequency components. It is widely used in image compression.
The DCT, first proposed by Nasir Ahmed in 1972, is a widely used transformation technique in signal processing and data compression. It is used in most digital media, including digital images (such as JPEG and HEIF, where small high-frequency components can be discarded), digital video (such as MPEG, H.26x and VCEG-MCM), digital audio (such as Dolby Digital, MP3 and AAC), digital television (such as SDTV, HDTV and VOD), and digital radio (such as AAC+ and DAB+).
The DCT is a Fourier-related transform similar to the discrete Fourier transform (DFT), but using only real numbers. It is equivalent to the DFT of roughly twice the length, operating on real data with even symmetry (since the Fourier transform of a real and even function is real and even), where in some variants the input and/or output data are shifted by half a sample. There are eight standard DCT variants, of which four are common. The most common variant of discrete cosine transform is the type-II DCT, which is often called simply ”the DCT”. Its inverse, the type-III DCT, is correspondingly often called simply ”the inverse DCT” or ”the IDCT”. Two related transforms are the discrete sine transform (DST), which is equivalent to a DFT of real and odd functions, and the modified discrete cosine transform (MDCT), which is based on a DCT of overlapping data.

I.E. if we write it in math language we have:

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right]$$
$$x_n = \frac{2}{N}\sum_{k=0}^{N-1} X_k \cos\left[\frac{\pi}{N}\left(n+\frac{1}{2}\right)k\right]$$
$$k = 0, 1, \ldots, N-1$$
$$n = 0, 1, \ldots, N-1$$

Load image and find YCbCr components

To load image and find YCbCr components I used the following code:

```
    img = imread("sample_19201280.bmp");
    imshow(img)
    img = rgb2ycbcr(img);
    Y = img(:,:,1);
    Cb = img(:,:,2);
    Cr = img(:,:,3);
```

Source Code 8: Load image and find YCbCr components

To compress it a little bit via saving the first component of every 2x2 block I used the following code:

```
y_downsampled = imresize(Y, 0.5, 'nearest');
cb_downsampled = imresize(Cb, 0.5, 'nearest');
cr_downsampled = imresize(Cr, 0.5, 'nearest');

% replace the original Cb component with the downsampled version
img_ycc(:,:,1) = y_downsampled;
img_ycc(:,:,2) = cb_downsampled;
img_ycc(:,:,3) = cr_downsampled;

% convert back to RGB color space
img_downsampled = ycbcr2rgb(img_ycc);
```

Source Code 9: Compress image

Result of this compression is shown in figure 20.

Use 2D DCT to compress image

To use 2D DCT to compress image I used the following code:

```
    double_Cb = double(cb_downsampled) - 128;
    double_Cr = double(cr_downsampled) - 128;
    double_y = double(y_downsampled) - 128;



    dct2_Cb = block_dct(double_Cb);
    dct2_Cr = block_dct(double_Cr);
    dct2_Y = block_dct(double_y);
```

Source Code 10: Use 2D DCT to compress image

Figure 20: Compressed image

Where function **block_dct** is defined as follows:

```
function arr = block_dct(double_arr)
    [h, w] = size(double_arr);
    num_blocks_h = floor(h / 8);
    num_blocks_w = floor(w / 8);
    dct_blocks = zeros(8, 8, num_blocks_h, num_blocks_w);
    for i = 1:num_blocks_h
        for j = 1:num_blocks_w
            block = double_arr((i-1)*8+1:i*8, (j-1)*8+1:j*8);
            dct_block = dct2(block);
            dct_blocks(:, :, i, j) = dct_block;
        end
    end
    arr = dct_blocks;
end
```

Source Code 11: block_dct function

To remove high frequency components via quantization with different quality factors given in the assignment I used the following code:

```
K = [1 5 10 20 50 80];
Quantisized_Cr = Quantisize(false,dct2_Cr,K);
Quantisized_Cb = Quantisize(false,dct2_Cb,K);
Quantisized_Y = Quantisize(true,dct2_Y,K);
```

Source Code 12: Remove high frequency components

Where function **Quantisize** is defined as follows:

```
function Quantisized = Quantisize(lum,dct2_arr,K)
    [a, b, h, w] = size(dct2_arr);
    Quantisized = zeros(length(K),a,b,h,w);
    alpha = 1;
    for quality = K
        if(lum)
        Quantisize_mat = [17 18 24 47 99 99 99 99 ;
                          18 21 26 66 99 99 99 99 ;
                          24 26 56 99 99 99 99 99 ;
                          47 66 99 99 99 99 99 99 ;
                          99 99 99 99 99 99 99 99 ;
                          99 99 99 99 99 99 99 99 ;
                          99 99 99 99 99 99 99 99 ;
                          99 99 99 99 99 99 99 99 ];
        else
        Quantisize_mat = [16 11 10 16 24 40 51 61 ;
                          12 12 14 19 26 58 60 55 ;
                          14 13 16 24 40 57 69 56 ;
                          14 17 22 29 51 87 80 62 ;
                          18 22 37 56 68 109 103 77 ;
                          24 35 55 64 81 104 113 92 ;
                          49 64 78 87 103 121 120 101 ;
                          72 92 95 98 112 100 103 99];
        end

        if(quality > 50)
                Quantisize_mat = Quantisize_mat * (100 - quality) / 50;
        else
                Quantisize_mat = Quantisize_mat * 50 / quality;
        end

         for i = 1:h
            for j = 1:w
                dct2_arr(:,:,i,j)= dct2_arr(:,:,i,j) ./ Quantisize_mat;
            end
         end
         Quantisized(alpha,:,:,:,:)=dct2_arr;
         alpha=alpha+1;
    end

end
```

Source Code 13: Quantisize function

To reconstruct the image for different qualities I used the following code:

```
new_y = 256*dct2_inverse(Quantisized_Y)+128;
new_Cr = 256*dct2_inverse(Quantisized_Cr)+128;
new_Cb = 256*dct2_inverse(Quantisized_Cb)+128;
figure
for i =1:6
    subplot(2,3,i)
    img_ycc(:,:,1) = new_y(i,:,:);
    img_ycc(:,:,2) = new_Cb(i,:,:);
    img_ycc(:,:,3) = new_Cr(i,:,:);
    img_new = ycbcr2rgb(img_ycc);
    imshow(img_downsampled)
    title(strcat("Quality =" , int2str(K(i))));
    sgtitle("JPG Image With Diffrent Qualities")
```

```
14 end
```

Source Code 14: Reconstruct image

Where new_X is 5D array of quantisized 8x8 blocks for each quality factor.
Function **dct2_inverse** is defined as follows:

```
1  function temp = dct2_inverse(Quantisized)
2      [a,b,c,d,e] = size(Quantisized);
3      temp=zeros(a,b*d,c*e);
4      for i=1:a
5          for j =1:d
6              for l=1:e
7                  temp(i,8*(j-1)+1:8*j,8*(l-1)+1:8*l)=reshape(idct2(Quantisized(
       i,:,:,j,l)),[8 8]);
8              end
9          end
10     end
11
12 end
```

Source Code 15: dct2_inverse function

Results are shown in Figure 21.

Figure 21: JPG Image With Diffrent Qualities

which shows that as the quality factor decreases the image quality decreases and the image becomes more blurry.Thats because we are removing high frequency components from the image, so the image takes less space but the quality doesn't get effected too much.

End of Computer homework