CM2208 Scientific Computing Report

c1911627

March 25, 2021

Basic Requirements

Displaying audio

```
11 function plotReg(app,y)
12     Fs = getappdata(app.UIAxes, 'sampRate');
13     len = length(y);
14     xAxis = (0:len-1)/Fs;
15     plot(app.UIAxes, xAxis, y);
16 end
```

This takes in a signal and plots it in the time domain. Line 15 creates an x axis by taking the values from 0 to the length of the signal (-1 to prevent indexing errors) and divides it by the sampling frequency. The division is to put it into discrete steps of T where $T = 1/Fs$

```
24 function plotSpectra(app, y)
25     Fs = getappdata(app.UIAxes, 'sampRate');
26 
27     N = 16000;
28     steps = (0:N-1)*(Fs/N);
29     fourier = abs(fft(y,N))/N;
30 
31     base = mean(y);
32     decibels = 20 * log10(fourier/base);
33     plot(app.UIAxesSpectrum, steps, decibels);
34 end
```

This was adapted from the Scientific Computing Slides

This also takes in a signal, however it plots it in the frequency domain instead. I've set N to 16000 as my equaliser only goes up to that point however the signal may have frequencies beyond that point. The *steps* on line 28 creates equidistant frequency steps of $\frac{f_s}{N}$ and *fourier* on line 29 gets the magnitude of the frequencies (need to do this because it has a real and imaginary part).

Shelving filters

```
42 function y = lowshelving (~, x, Wc, G)
43     % y = lowshelving (x, Wc, G)
44     % Author: M. Holters
45     % Applies a low-frequency shelving filter to the input signal x.
46     % Wc is the normalized cut-off frequency 0<Wc<1, i.e. 2*fc/fS.
47     % G is the gain in dB
48 
49     V0 = 10^(G/20); H0 = V0 - 1;
50     if G >= 0
51         c = (tan(pi*Wc/2)-1) / (tan(pi*Wc/2)+1);     % boost
52     else
```

```
53          c = (tan(pi*Wc/2)-V0) / (tan(pi*Wc/2)+V0);    % cut
54      end
55      xh = 0;
56
57      for n = 1:length(x)
58          xh_new = x(n) - c*xh;
59          ap_y = c * xh_new + xh;
60          xh = xh_new;
61          y(n) = 0.5 * H0 * (x(n) - ap_y) + x(n);  % change to minus for HS
62      end
63  end
```

This was taken from DAFX chapter 2.3

This is a low shelving filter. It's a first order lowpass filter. You can modify the parameters to make it a highpass filter which I've done in the code. It takes in a signal and boosts/cuts frequencies below a certain cuttoff point.

It has the difference equation:

$$x_h(n) = x(n) - c_{B/C}x_h(n-1)$$
$$y_l(n) = c_{B/C}x_h(n) + x_h(n-1)$$
$$y(n) = \frac{H_0}{2}[x(n) + yl(n)] + x(n)$$
$$H_0 = V_0 - 1$$
$$V_0 = 10^{G/20}$$
$$C_B = \frac{tan(\pi\frac{f_c}{f_S}) - 1}{tan(\pi\frac{f_c}{f_S}) + 1}$$
$$C_C = \frac{tan(\pi\frac{f_c}{f_S}) - V_0}{tan(\pi\frac{f_c}{f_S}) + V_0}$$

C_B is for boosting the amplitude for that frequency and C_C is for cutting.

The H_0 and V_0 calculations are done on line 49.

On lines 50 - 55 the program determines the correct parameters to cut or boost the frequency based on the gain. Then it applies the difference equation from lines 57 - 62.

The high shelving filter is very similar to this but $y(n) = \frac{H_0}{2}[x(n) - yl(n)] + x(n)$ is used instead.

Peak filters

```
function y = peakfilt(~, x, Wc, Wb, G)
    % y = peakfilt (x, Wc, Wb, G)
    % Author: M. Holters
    % Applies a peak filter to the input signal x.
    % Wc is the normalized center frequency 0<Wc<1, i.e. 2*fc/fS.
    % Wb is the normalized bandwidth 0<Wb<1, i.e. 2*fb/fS.
    % G is the gain in dB.

    V0 = 10^(G/20); H0 = V0 - 1;
    if G >= 0
      c = (tan(pi*Wb/2)-1) / (tan(pi*Wb/2)+1);     % boost
    else
      c = (tan(pi*Wb/2)-V0) / (tan(pi*Wb/2)+V0);   % cut
    end
    d = -cos(pi*Wc);
```

```
75      xh = [0, 0];
76      for n = 1:length(x)
77        xh_new = x(n) - d*(1-c)*xh(1) + c*xh(2);
78        ap_y = -c * xh_new + d*(1-c)*xh(1) + xh(2);
79        xh = [xh_new, xh(1)];
80        y(n) = 0.5 * H0 * (x(n) - ap_y) + x(n);
81      end
82  end
```

This is also taken from DAFX chapter 2.3

This is a peaking filter. It's a second order allpass filter. It takes in a center frequency and a bandwidth and boosts or cuts the frequencies within that bandwidth.

$$x_h(n) = x(n) - d(1 - c_{B/C})x_h(n-1) + c_{B/C}x_h(n-2)$$

$$yl(n) - c_{B/C}x_h(n) + d(1 - c_{B/C})x_h(n-1) + x_h(n-2)$$

$$y(n) = \frac{H_0}{2}[x(n) - y_1(n)] + x(n)$$

$$d = -cos(2\pi f_c/f_s)$$

$$V_0 = 10^{\frac{G}{20}}$$

$$H_0 = V_0 - 1$$

C_B and C_C are the same as the shelving filters.

Lines 67 to 73 are setting the parameters for the difference equations and 75 to 80 are applying the difference equations to the signal.

Putting it together

We can now use these functions to build our 10 band equaliser.

```
        function EqualisePeakButtonPushed(app, event)
            y = getappdata(app.UIAxes, 'sound');
            Fs = getappdata(app.UIAxes, 'sampRate');

            %low shelving
            y1 = lowshelving(app, y, (2*32)/Fs, app.HzSlider.Value);

            %peaking
            y2 = peakfilt(app, y1, 2*(64)/Fs, (2*64)/Fs, app.HzSlider_2.Value);
            y3 = peakfilt(app, y2, 2*(128)/Fs, (2*128)/Fs, app.HzSlider_3.Value);
            y4 = peakfilt(app, y3, 2*(256)/Fs, (2*256)/Fs, app.HzSlider_5.Value);
            y5 = peakfilt(app, y4, 2*(512)/Fs, (2*512)/Fs, app.HzSlider_4.Value);
            y6 = peakfilt(app, y5, 2*(1024)/Fs, (2*1024)/Fs, app.KHzSlider.Value);
            y7 = peakfilt(app, y6, 2*(2048)/Fs, (2*2048)/Fs, app.KHzSlider_2.Value);
            y8 = peakfilt(app, y7, 2*(4096)/Fs, (2*4096)/Fs, app.KHzSlider_3.Value);
            y9 = peakfilt(app, y8, 2*(8192)/Fs, (2*8192)/Fs, app.KHzSlider_4.Value);

            %high shelving
            y10 = highshelving(app, y9, (2*16348)/Fs, app.KHzSlider_5.Value);

            %amp
            y11 = y10*(app.ampSlider.Value/100);

            setappdata(app.UIAxes, 'Eqsound', y11);
            plotReg(app,y11);
```

```
125             plotSpectra(app, y11);
126         end
```

This piece of code runs when a button is pushed, it first takes in the gain of the first slider on line 109. The output of that filter is original signal with a cut/boost at 32Hz. The modified signal is then passed through the peak filters one by one (lines 112 - 119) and then the high shelving filter on line 125. Finally, the signal is multiplied by some amplitude defined by the amplitude slider on line 153.

This is the basis of the equaliser. By boosting and cutting specific frequencies it allows the user to equalise their sound file.

Setting and loading presets

```
function LoadData(app)
    [fileName, pathName] = uigetfile(['*.','txt'],'Load Slider Levels');
    if (fileName ~= 0)
        data = load([pathName, fileName], '-mat');
        app.ampSlider.Value = data.sliderLevels(1);
        app.HzSlider.Value = data.sliderLevels(2);
        app.HzSlider_2.Value = data.sliderLevels(3);
        app.HzSlider_3.Value = data.sliderLevels(4);
        app.HzSlider_4.Value = data.sliderLevels(5);
        app.HzSlider_5.Value = data.sliderLevels(6);
        app.KHzSlider.Value = data.sliderLevels(7);
        app.KHzSlider_2.Value = data.sliderLevels(8);
        app.KHzSlider_3.Value = data.sliderLevels(9);
        app.KHzSlider_4.Value = data.sliderLevels(10);
        app.KHzSlider_5.Value = data.sliderLevels(11);
    end
end

function saveData(app)
    var1 = app.ampSlider.Value;
    var2 = app.HzSlider.Value;
    var3 = app.HzSlider_2.Value;
    var4 = app.HzSlider_3.Value;
    var5 = app.HzSlider_4.Value;
    var6 = app.HzSlider_5.Value;
    var7 = app.KHzSlider.Value;
    var8 = app.KHzSlider_2.Value;
    var9 = app.KHzSlider_3.Value;
    var10 = app.KHzSlider_4.Value;
    var11 = app.KHzSlider_5.Value;

    sliderLevels = [var1, var2, var3, var4, var5, var6, var7, var8, var9, var10, var11];
    [fileName,pathName] = uiputfile(['sliders.','txt'],'Save slider levels');
    if (fileName ~= 0)
        save([pathName,fileName],'sliderLevels');
    end
end
```

Here I've created 2 functions to save and load presets. The code saves the sliders to a text file which the program can load later.

Novel features

Wah Wah Filter

```
function y = wahwah(app,x)
    %Taken from the CM0268 MATLAB DSP Graphics Slides
    Fs = getappdata(app.UIAxes, 'sampRate');

    %damping factor
    %Lower the damping factor the smaller the pass band
    damp = 0.05;

    % min and max centre cutoff frequency of variable bandpass
    % filter
    minf=500;
    maxf=3000;

    %wah frequency, how many Hz per second are cycled through
    Fw = 3000;

    % change in centre frequency per sample (Hz)
    delta = Fw/Fs;

    % create triangle wave of centre frequency values
    Fc=minf:delta:maxf;
    while(length(Fc) < length(x))
        Fc = [Fc (maxf:-delta:minf)];
        Fc = [Fc (minf:delta:maxf)];
    end

    % trim tri wave to size of input
    Fc = Fc(1:length(x));

    %difference equation coefficients

    F1 = 2*sin(pi*(Fc(1)/Fs)); %must be recalculated each time Fc changes
    Q1 = 2*damp; % dictates the size of the pass bands

    yh=zeros(size(x)); %preallocating vectors
    yb=zeros(size(x));
    yl=zeros(size(x));

    % first sample, to avoid referencing of negative signals
    yh(1) = x(1);
    yb(1) = F1*yh(1);
    yl(1) = F1*yb(1);

    % apply difference equation to the sample
    for n=2:length(x)
        yh(n) = x(n) - yl(n-1) - Q1*yb(n-1);
        yb(n) = F1*yh(n) + yb(n-1);
        yl(n) = F1*yb(n) + yl(n-1);
        F1 = 2*sin((pi*Fc(n))/Fs);
    end

    %normalise
    maxyb = max(abs(yb));
```

```
    y = yb/maxyb;
end
```

Taken from the CM0268 MATLAB DSP Graphics Slides

1. Select the cutoff frequencies for the triangle. In this case it's 500Hz - 3000Hz
2. Select the wah frequency. This determines how many Hz/s are cycled through.
3. Create the wave.
   - This involves creating an array going from your minimum cutoff frequency to your maximum cutoff frequency.
   - Then you append it with an array doing the opposite i.e maximum cutoff frequency → minimum cutoff frequency.
   - Repeat until your triangle wave is at least the length of the sound.
4. Use this triangle wave to obtain frequencies for a sine wave. These frequencies will go up and down giving the wah wah effect.
5. Set the band pass size.
6. The sine wave is used as a parameter in the state variable filter.

The state variable filter combines a lowpass, bandpass and highpass filter and is defined by:

$$y_l(n) = F_1 y_b(b) + y_l(n-1)$$
$$y_b(n) = F_1 y_h(n) + y_b(n-1)$$
$$y_h(x) = x(n) - y_l(n-1) - Q_1 y_b(n-1)$$
$$F_1 = 2sin(\pi f_c/f_S), Q_1 = 1/Q$$

Flanger and Chorus

```
function y = flanger(app,x)
    %y(n) = x(n) + gx[x - M(n)]
    %M(n) = M0 * [1 + A*sin(2*pi*n*(f/Fs))]
    %f = flang freq
    %A = sweep/excursion
    %M0 = average delay length
    %g = depth control (should be set to 1 for maximum effect)
    %(also -1 for inverted mode)

    Fs = getappdata(app.UIAxes, 'sampRate');

    f = 0.3; %can be from 0.1 to 1hz
    g = 0.8;
    M0 = 0.008*Fs;
    A = 0.5;


    y = zeros(1,length(x));
    for i =1:M0
        y(i) = x(i);
    end

    for n = (round(M0)+1):length(x)
        M = abs(M0 *(A*sin(2*pi*n*(f/Fs))));
        M = round(M);
        y(n) = x(n) + g*x(n - M);
    end
end
```

The flanger filter is a basic delay based filter. It can be describe by $y(n) = x(n) + gx[x - M(n)]$. Where $M(n)$ is a sine wave. You can see this in figure 2 of the supporting material. This causes the delay $n - M(n)$ to oscillate giving the flanger effect. Various parameters are set to adjust the effect.E.g

- f is the flanger frequency which determines the speed of the oscillation.
- A determines the amplitude of the occilation.
- M0 is the average length of the delay. We need to make it a small number for it to be flanger effect.
- g determines how big of an effect the flanger filter has on the signal. We can set this to be negative to do an inverted flanger.

I also implemented a chorus which is just a flanger with a longer delay.

Attempt at a Fourier equaliser

```
function newY = fourierFilter(app, y, G, sF, eF)

    %Take fourier transform -> Filter it in freq domain -> Take
    %inverse fourier
    Fs = getappdata(app.UIAxes, 'sampRate');
    G = 10^(G/20);
    N = 16348;
    fourier = fft(y);
    freqs = (0:N-1)*(Fs/N);

    newY=fourier;

    mask=find((freqs>=sF*(Fs/N))&(freqs<=eF*(Fs/N)));
    newY(mask) = fourier(mask)*G;
    newY = ifft(newY);
    newY = real(newY);
end
```

This was my attempt at a Fourier filter. It doesn't do exactly what I want, however the idea was to take the Fourier transform, select the frequencies I wanted to boost/cut and multiply those frequencies by some amplitude. However, this doesn't have the effect I want.

Supporting material

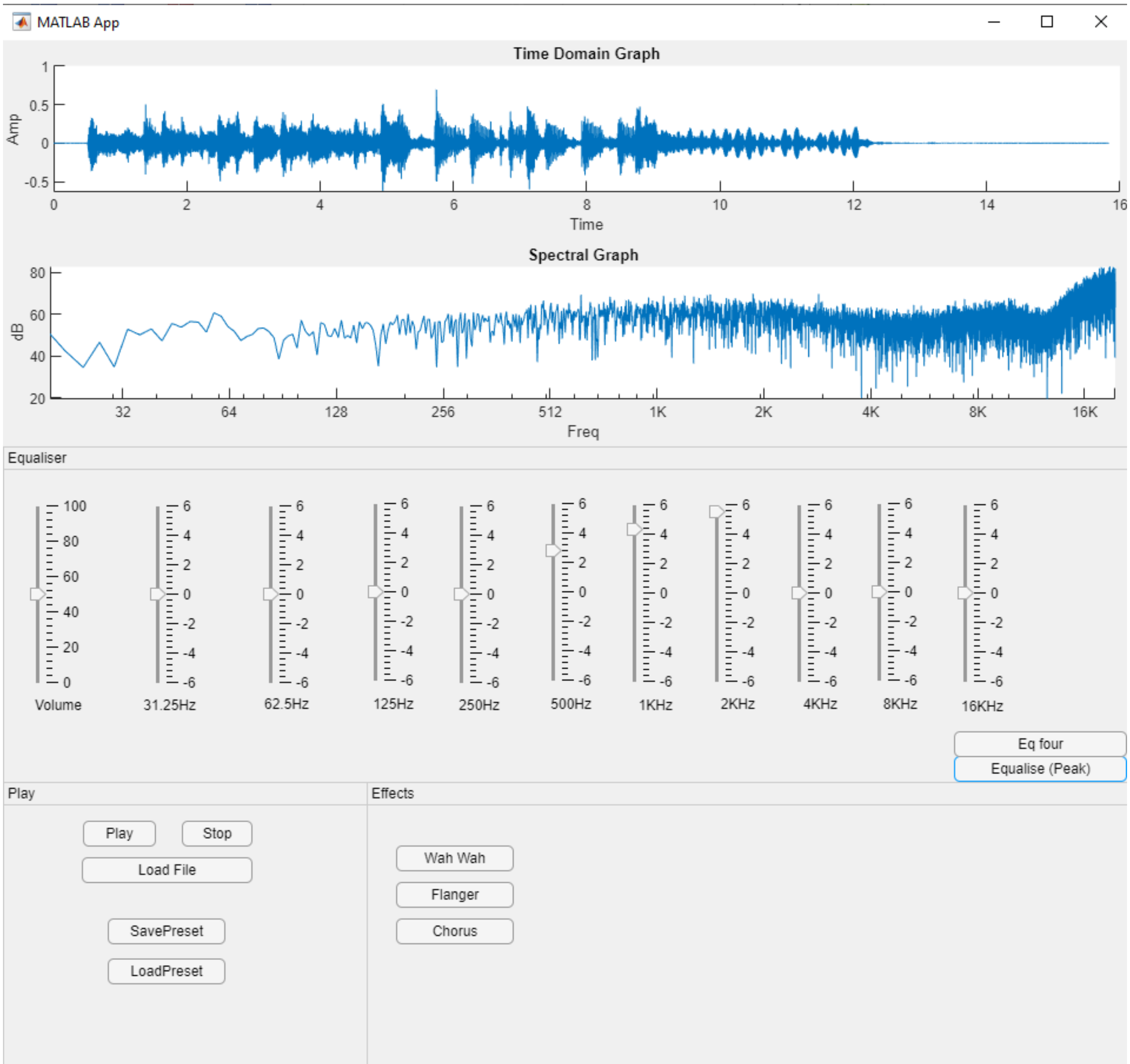
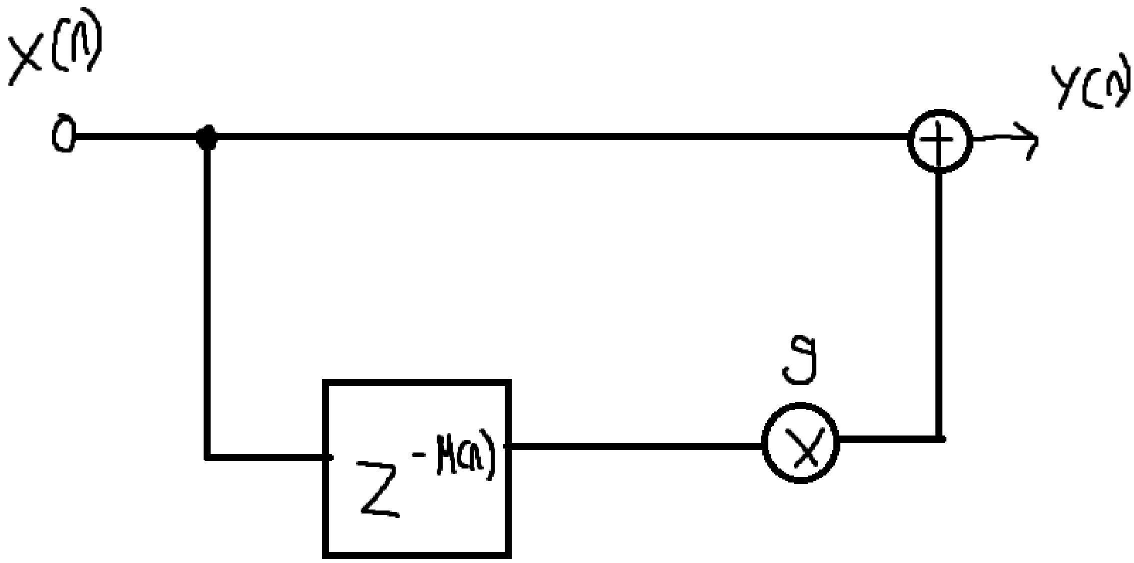


Figure 1: 10 Band Equaliser



$$M(n) = M_0 + A \cdot \sin(2 \cdot \pi \cdot n \cdot f)$$

Figure 2: Flanger Diagram