Problem Set 9: 10-Band Graphic Equalizer

Please send back via NYU Classes

- A zip archive named as
  PS09_<your name as Last_First>.zip
  containing the C code files that implements all aspects of all problems.

You have two weeks to complete this assignment.

Total points: 200
Points are awarded as follows:
main.cpp

- **25 Points** – main.cpp command line parsing, usage() printout, set filter gains, open WAV files
- **25 Points** – Loop over all buffers in input file and filter the buffers
- **10 Points** – Perform test of main.cpp using default equalizer.cpp, `#ifdef (1)`

equalizer.cpp

- **25 Points** – Class constructor and misc. member functions
- **25 Points** – filter() member function,
- **40 Points** – filter_band() member function
- **25 Points** – Perform 2 tests of full main.cpp and equalizer.cpp

General

- **25 Points** - clear code, good comments, sensible formatting

Task Overview

In this problem set you will create a 10-band graphic equalizer. In your program you create a **main.cpp** that reads frames of an input file, filters them by the 10-band graphic equalizer, and writes the result to an output file. In order to keep the **main.cpp** program simple, there is no interactive control and there is no real-time audio output.

The audio signal processing part of the problem set is creating the 10-band graphic equalizer filter program. This will be written as a C++ class.

The instructor has supplied the complete code for these files

`main.h`
`equalizer.h`
`coef.h`

You will write **all** of

main.cpp

You will **complete** the code in

equalizer.cpp

The elements of **main.cpp** and **equalizer.cpp** are shown below.

You can use

```
build.sh
```

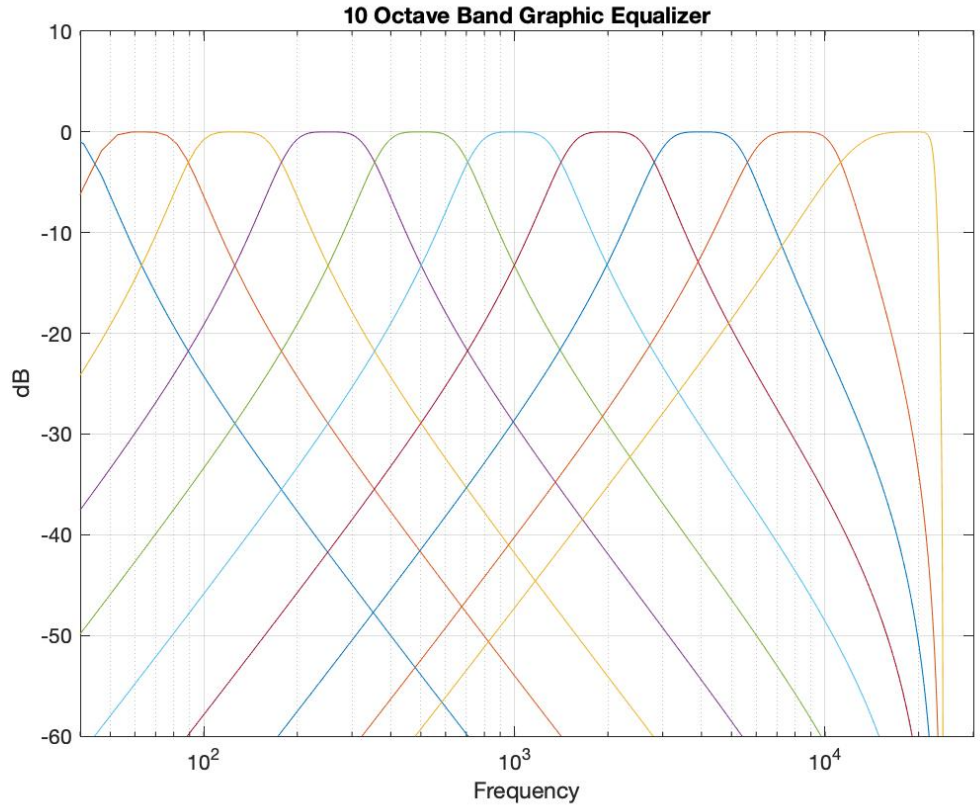
to compile the equalizer program, and you can test it with the supplied WAV files

```
trilogy.wav
chirp.wav
```

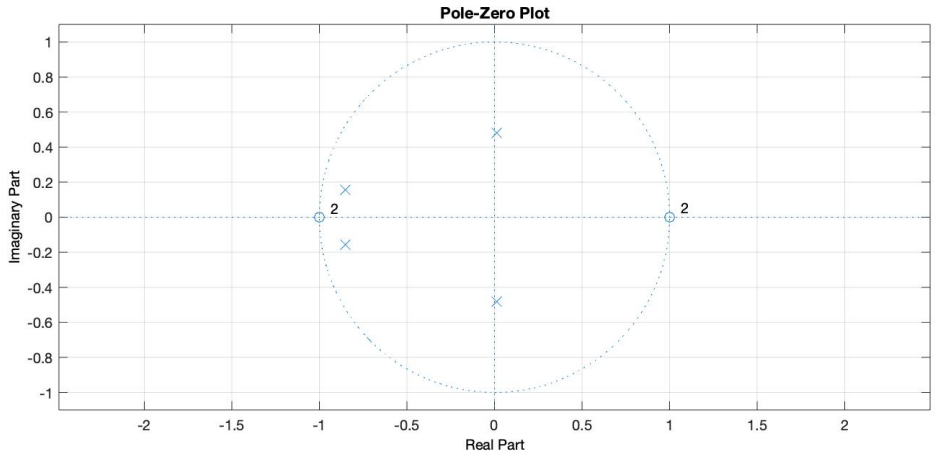
Background Information

The 10 octave-band filters that are specified in coef.h are auto-regressive filters, i.e. they have both FIR feedforward taps and IIR feedback taps. The center frequencies of the 10 bands conform to the ANSI S1.11-2004 standard.

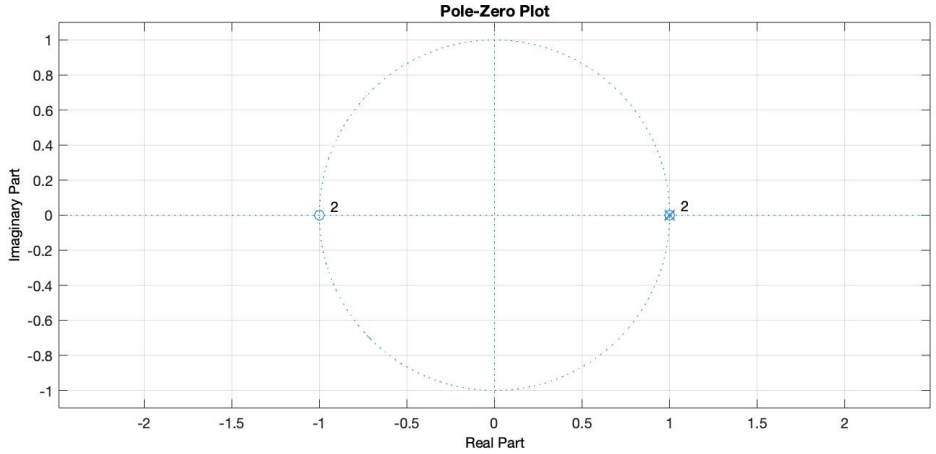
The filters were designed using the Matlab graphicEQ() utility. A plot of the filter response is shown here (on a log-log scale, where the x-axis is dB).



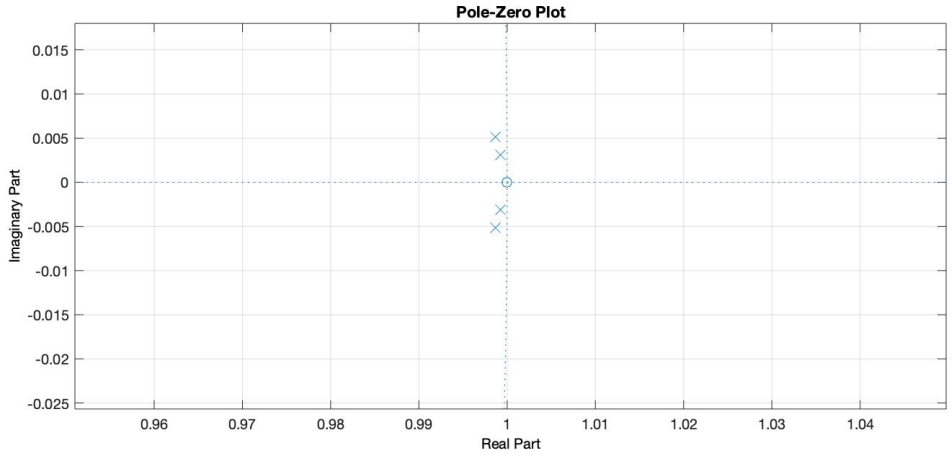
The pole (x) and zero (o) plot for the highest frequency filter (band 10) is shown here:



The pole (x) and zero (o) plot for the lowest frequency filter (band 1) is shown here:



For band 1, all of the poles are very close to together. Zooming in we get this view:



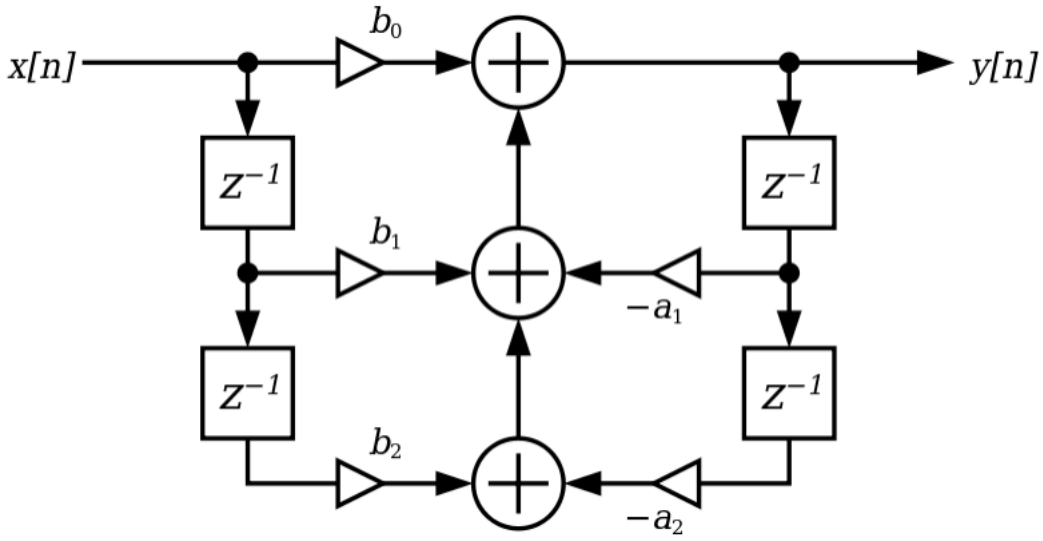
Note that the unit circle in the previous plot now appears to be a nearly straight vertical line. These poles are very close to the unit circle because the lowest band is the narrowest. Remember that if the poles are actually *outside* the unit circle, the filter becomes unstable, and its output typically increases exponentially until the filter arithmetic saturates. Because of this, the coefficients and the y_i values involved

in the calculation of our 10-band harmonic equalizer filters must be **type double**, so that there is sufficient precision to keep all filters stable.

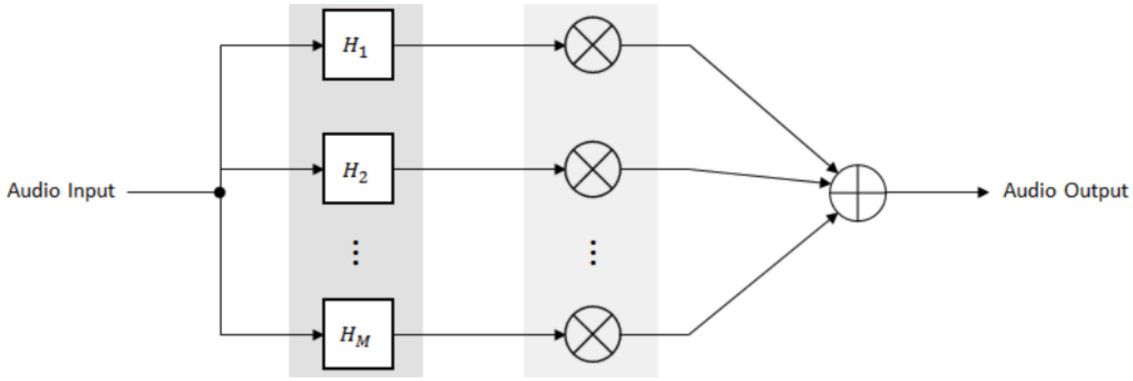
Each filter is computed as shown in this equation, where the subscript *i* indicates the filter band number:

$$y_i(n) = \sum_{k=0}^{Mb-1} x(n-k)b_i(k) - \sum_{k=1}^{Ma-1} y_i(n-k)a_i(k)$$

This equation is realized in the following "Direct Form I" digital filter signal flow graph, where each z^{-1} box is a delay line element, or an ibuf[] or obuf []storage element.



The complete graphic equalizer is the sum of the individual filter outputs, with some gain applied to each filter output, as shown here:



Create main.cpp

Use the following #includes:

```
#include <stdio.h>
#include <stdlib.h>      /* malloc() */
#include <string.h>      /* memset() */
```

```
#include <sndfile.h>     /* libsndfile */
#include "main.h"        /* frames, channels */
#include "equalizer.h"   /* defines equalizer class */
```

Declare the input and output buffers, for example:

```
float input[FRAMES_PER_BUFFER*MAX_CHAN];
float output[FRAMES_PER_BUFFER*MAX_CHAN];
```

Declare an instance of the Equalizer class and a pointer to the class object. For example:

```
Equalizer eq, *pe = &eq;
```

Declare libsndfile data structures (SNDFILE and SF_INFO) for ifile and ofile, for example:

```
SNDFILE *isndfile, *osndfile;
SF_INFO isfinfo, osfinfo;
```

Parse command line and print usage if appropriate. Command line should be as shown in this example usage:

```
Usage: main F|L|B|H input_file.wav output_file.wav
Where F is flat, L is low-pass, B is band-pass and H is high-pass
```

Let the letter in `argv[1]` be the `mode`:

```
mode = argv[1][0];
```

Set filter gains based on `mode`, where
`F` is no action (use default settings for all bands)
`L` is bands 7, 8, 9 are set to -80 dB, all others are default values (low pass filter)
`B` is bands 4,5,6 are set to -80 dB, all others are default values (band-reject filter)
`H` is bands 0, 1, 2 are set to -80 dB, all others are default values (high pass filter)

Use `memset()` to zero libsndfile structures.

Open input WAV file and print parameters, for example:

```
Input audio file trilogy.wav:
   Frames: 449721 Channels: 2 Samplerate: 48000
```

Set parameters for output WAV file to be the same as input WAV file and open output WAV file

Declare single-channel buffers

```
float icbuf[FRAMES_PER_BUFFER];
float ocbuf[FRAMES_PER_BUFFER];
```

Using `while()`, loop until end of input file, for example:

```
while ( (count = sf_readf_float (isndfile, input,
FRAMES_PER_BUFFER)) == FRAMES_PER_BUFFER) {
```

On each iteration of the loop:

Read `FRAMES_PER_BUFFER` frames from input file (done in arg of `while()` loop in example statement above. As shown in example code, if there are less than `FRAMES_PER_BUFFER` frames left in file the `while()` terminates.

Within the body of the loop:
De-interleave from input buffer to single channel buffers
Filter the single-channel buffers
Re-interleave from single channel buffers to output buffer
Write `FRAMES_PER_BUFFER` frames to output file
Close output file

Test main.cpp

Test your finished main.cpp using the instructor-supplied equalizer.cpp by setting the arg of `#if (0)` to `#if (1)`. Now it just copies the input buffer to the output buffer. Hence, the output file from main.cpp should be an exact copy of the input file.

After this test is successful, set the #ifdef arg to 0 in member function `Equalizer::filter()`. Then add your own code to complete the equalizer class.

Create equalizer.cpp

The class Equalizer is declared in **equalizer.h**, which is supplied by the instructor.
You must complete code in **equalizer.cpp** that defines the class.

The file begins with #includes.

Next, it defines the 10-band octave center frequencies:

```
static  float octave_bands[] = {31.5, 63, 125, 250,
500, 1000, 2000, 4000, 8000, 16000};
```

Write the code to for the body of the
Constructor (discussed below), which will
Initialize equalizer parameters
Initialize the band center frequencies
Initialize pointer to filter coefficients. Optionally, print the filter coefficients
Zero filter state
Destructor
This can be an empty function. It is complete as supplied.
Public methods (discussed below)

```
int get_num_bands(void)
void get_band_cf_str(int band, char *s)
float get_gain_dB(int band)
void set_gain_dB(int band, float gain)
void filter(const float *ibuf, float *obuf, int num_frames,
      int chan)
```

Private methods (discussed below)

```
void filter_band(const float *ibuf, float *ibuf_state,
      double *obuf_band, double *obuf_state, FiltCoef *pfc,
      int num_frames, int num_coef);
```

Class private data are shown in `equalizer.h`, and their values are initialized in constructor.

Private:

```
int num_bands;
FiltGain filt_gain[MAX_BAND];
float band_center_freq[MAX_BAND];
FiltCoef *pfc; // access as pfc[band].b[i]
float ibuf_state[MAX_CHAN][MAX_COEF];
double obuf_state[MAX_CHAN][MAX_BAND][MAX_COEF];
double obuf_band[MAX_BAND][FRAMES_PER_BUFFER];
```

Class Constructor

Initialize equalizer parameters:

Use #define `MAX_BAND` and `INIT_GAIN_DB` values from **equalizer.h**.

Set

```
num_bands = MAX_BAND;
```

For each band, set the values in struct FiltGain to

```
filt_gain[].gain_dB = INIT_GAIN_DB;
```

Then set the `gain_lin` value based on the dB value using the formula

$$\text{Linear\_gain} = 10^{(\text{gain\_dB}/20)}$$

Which is realized by the expression

```
pow(10.0, gain_dB/20.0);
```

Initialize the band center frequencies

Copy the global value `octave_bands[]` in equalizer.cpp to the private data `band_center_freq[]`.

Initialize pointer to filter coefficients and optionally print the coefficients as a check that they are initialized correctly.

The filter coefficients are defined in `coef.h` as a global struct `FiltCoef filt_coef[10]`. Note that this is an array of structures, and so the initialization requires use of braces and commas just as shown. Set the private data pointer `pfc` equal to the address of the base of the `filt_coef[]` array, that is:

```
pfc = &filt_coef[0];
```

Zero filter state

Since the filter has both FIR and IIR components (i.e. the convolution involves both input (x) values and output (y) values, there is state for both input and output. Zero the input state ibuf_state[][], the output state obuf_state[][][] and obuf_band[][]. The range of the array indices is obvious from the declaration in **equalizer.h**.

Public methods

`int get_num_bands(void)`

This function returns the private data variable num_bands.

`void get_band_cf_str(int band, char *s)`
This is complete as supplied, and is only used in the next problem set, creating a VST plugin.

`float get_gain_dB(int band)`
This function returns the private data value `filt_gain[band].gain_dB`

`void set_gain_dB(int band, float gain_dB)`
Assume that the function argument `gain_dB` is a value in dB. Copy it to the private data value `filt_gain[band].gain_dB`. Also convert it to linear gain (see discussion on constructor, above) and use this to set the private data value `filt_gain[band].gain_lin`.

`void filter(const float *ibuf, float *obuf, int num_frames, int chan)`
Get the number of bands from the member function `get_num_bands()`

For each filter band, set the value of num_coef to `pfc[band].num_coef, that is:`

```
int num_coef = pfc[band].num_coef;
```

Next call

```
filter_band()
```

Which is defined below, under private methods.

After all bands are filtered, compute the output values, that is the values in `obuf[]` which are equal to the sum of the associated values in each band output. In addition, multiply the band output by the band gain. That is, use two loops:
For each frame index i

```
For each band index band
     obuf[i] += filt_gain[band].gain_lin *
(float)obuf_band[band][i];
```

Finally, update the input state by saving the last MAX_COEF values from `ibuf[]` to `ibuf_state[chan][].`

Private methods

`void Equalizer::filter_band(const float *ibuf, double *obuf_band,`
`FiltCoef *pfc, float *ibuf_state, double *obuf_state,`
`int num_frames, int num_coef)`
This is the member function that actually does the filtering, but does the filtering for just one band. See the comment block above the function name in **equalizer.cpp** for an exact description of the filtering process.

Remember that the block of frames are de-interleaved into single-channel buffers in your main.cpp, so at this point the convolution is on a single-channel buffer, which consistes of a current input buffer and a state input buffer (i.e. saved from the last block).

Use an outer `for()` loop over all frames in the input buffer, where the number of frames is the is the function argument `num_frame`.

Declare a filter output value and set it to zero:

```
double y = 0;
```

Create two inner `for()` loops: a first over the number of `b` coefficients, and a second over the number of `a` coefficients. For each loop, the number of coefficients is the same, and is the function argument `num_coef`.

As discussed in lecture, the convolution is performed as:

$$y(n) = \sum_{k=0}^{Nb-1} x(n-k)b(k) - \sum_{k=1}^{Na-1} y(n-k)a(k)$$

Note that the elements in the second convolution sum are *subtracted* and also that the sum begins at index 1 (since a(0) is 1.0 and can be neglected). As stated above, the number of coefficients *Nb* and *Na* are the same, and equal to `num_coef`.

In the first convolution sum, the values of x are the values in ibuf. However, if the index (`n-k`) is negative, you must use the values in ibuf_state. Offset this by MAX_COEF so that when (`n-k`) is -1, `ibuf_state[MAX_COEF+(n-k)]` is the top value (newest value) in `ibuf_state[]`

Similarly, in the convolution sum, the values of y are the values in `obuf_band[]`. If the index (`n-k`) is negative, you must use the values in `obuf_state[].` As above, offset this by MAX_COEF so that when (`n-k`) is -1, `obuf_state[MAX_COEF+(n-k)]` is the top value (newest value) in `obuf_state[]`

After both loops (for the first and second convolution sums), save the filter output value:

```
obuf_band[n] = y;
```

Since the second convolution sum starts at k=1, the value of `obuf_band[n]` is not part of the sum.

Finally, update the vaues in `obuf_state[]` by copying the upper (newest) MAX_COEF values from `obuf_band[]` to `obuf_state[].`

Test Equalizer

Run your program using the following command lines and observe the output.

Command Line	**Result (in foo.wav)**
`./main F trilogy.wav foo.wav`	Sounds the same as input
`./main L trilogy.wav foo.wav`	Highs are gone
`./main B trilogy.wav foo.wav`	Mid-range is gone
`./main H trilogy.wav foo.wav`	Lows are gone

Run your program using the following command line and, for each command, observe the resulting foo.wav in a DAW. The signal `chirp.wav` is a tone signal whose frequency increases on a log scale. Specifically, the frequency increases one octave every second. Hence, the signal "occupies" each band of your 10-band equalizer for exactly 1 second.

Command Line	**Result (in foo.wav)**
`./main F chirp.wav fooF.wav`	Essentially the same as input
`./main L chirp.wav fooL.wav`	Last 3 seconds of signal are attenuated
`./main B chirp.wav fooB.wav`	The middle 3 seconds of signal are attenuated
`./main H chirp.wav fooH.wav`	First 3 seconds of signal are attenuated

Note that the graphic equalizer filterbank is not perfect, in that it has perhaps 3 dB of ripple across the audio band. In the figure below, the top waveform is the input, chirp.wav and the second is the output with all band gains set to 0 dB. The ripple, which occurs at the band edges is quite visible.