Programming Assignment #5

CS 246, Fall 2018

Due Wednesday, October 24

I will provide you with the files `BandPass2.h` and `BandPass2.cpp` that implements a 12 dB/oct band–pass filter. The interface for the filter is exactly the same as the 6 dB/oct band–pass filter that you constructed in the previous assignment. For this assignment, you will use this 12 dB/oct filter to implement a complex filter that combines multiple band–pass filters. You may choose either a multi–band graphic equalizer, or a channel vocoder. Each of the filters is described below. Choose one!

Option #1: Multi–band equalizer

Implement a class for a multi–band graphic equalizer (with constant Q–factors) for the frequency range 20 Hz—20 kHz. You may choose to hard–code the number of bands, although you there must be at least 5 bands. If you do this, you will receive a *maximum* of 92% for this assignment. To receive full credit, you must allow for a variable number of frequency bands. The *public* interface to the class must be:

```
class Equalizer : public Filter {
  public:
    Equalizer(int n=1, float R=44100);
    ~Equalizer(void);
    int bandCount(void) const;
    void setGain(int n, float g);
    float operator()(float x);
};
```

`Equalizer(N,R)` — (constructor) creates a constant Q–factor equalizer with sampling rate R. If you hard–code the number of frequency bands, you may ignore the value of N. Otherwise, the value of N determines the number of bands. All frequency bands should have an initial gain of 0 dB; i.e., a linear gain factor of 1.

`~Equalizer()` — (destructor) destroys an equalizer object. Depending on your implementation, this may be trivial.

`bandCount()` — returns the actual number of frequency bands of the equalizer (if you do not hard–code the number of bands, the function should should return the value N specified in the constructor).

`setGain(n,g)` — sets the gain of frequency band n to the (linear) gain factor g. You may assume that n is between 0 and $(N-1)$. No range checking needs to be performed.

`operator()(x)` — (filter function) returns the output of the equalizer, given the input sample x. The n–th call to this function returns the n–th output sample.

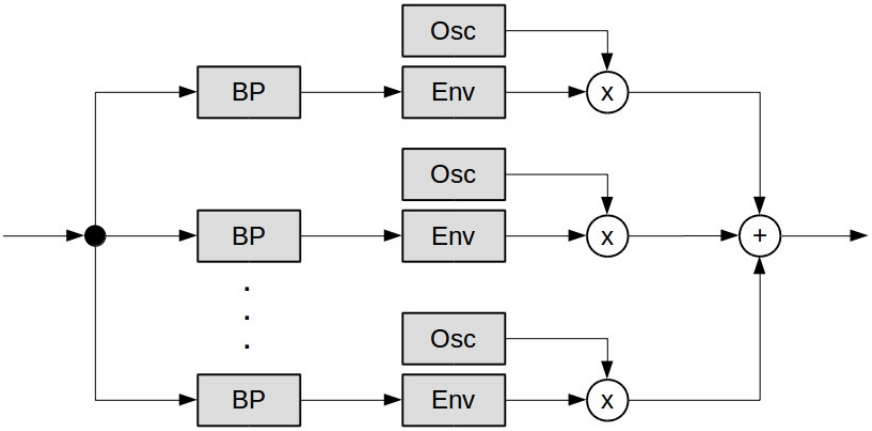
You may not add or remove functionality from the public section of the class. However, you are free to declare the *private* portion of the class as you see fit.

If you choose this for your assignment, you should submit the two files: (1) the header file `Echo.h`, and (2) the implementation file `Equalizer.cpp`. You may include the `Filter.h`, `Equalizer.h`, and `BandPass2.h` header files, as well as any standard `C++` header file.

Option #2: Channel vocoder

Vocoder filter

To construct a channel vocoder, we first divide the input signal — the **modulator signal** — into several frequency bands using band–pass filters, just like with an equalizer. However, the second step is very different. For each frequency band, we use a filter called an *envelope follower* (or *envelope detector*), to extract the envelope from the filtered signal. This is then used as an envelope for a different signal, the **carrier signal**. For each frequency band, we use a separate carrier signal, usually with a frequency related to the central frequency of the frequency band.



For the purposes of this assignment, the carrier signal should be a waveform of your choice; e.g., a sine wave, or a sawtooth wave. However for each frequency band, the carrier signal should have a frequency that is a *fixed multiple* μ of the central frequency of the band–pass filter used to construct the frequency band. The multiple has the same value for all frequency bands (this will be a vocoder parameter). If you are feeling adventurous, you can use filtered noise as the carrier signal of a frequency band, with the noise filtered by a band–pass filter that has a central frequency that is a fixed multiple of the central frequency of the frequency band (you can make the Q–factor different).

Envelope follower

We may construct an envelope follower by modifying a low–pass filter. Recall that our (first order improved) low–pass filter from assignment #4 is given by the recurrence relation

$$y_n = a(x_n + x_{n-1}) + by_{n-1}, \quad \text{where} \quad a \doteq \tfrac{\theta}{1+\theta},\ b \doteq \tfrac{1-\theta}{1+\theta},\ \theta \doteq \tan\left(\tfrac{\pi f_L}{R}\right)$$

with f_L the cutoff frequency. In our case, we desire an envelope of the *rectified* signal; that is, the absolute value of the input signal $|x|$. Moreover, we will allow the coefficients a and b to be different in the cases when the signal amplitude is increasing or decreasing. The envelope–follower is then cast in the form

$$y_n = \begin{cases} a_u(|x_n| + |x_{n-1}|) + b_u y_{n-1} & \text{if } |x_n| > y_{n-1} \\ a_d(|x_n| + |x_{n-1}|) + b_d y_{n-1} & \text{if } |x_n| \leq y_{n-1} \end{cases}$$

where

$$a_u \doteq \tfrac{\theta_u}{1+\theta_u},\ b_u \doteq \tfrac{1-\theta_u}{1+\theta_u},\ \theta_u \doteq \tan\left(\tfrac{\pi f_u}{R}\right) \quad \text{and} \quad a_d \doteq \tfrac{\theta_d}{1+\theta_d},\ b_d \doteq \tfrac{1-\theta_d}{1+\theta_d},\ \theta_d \doteq \tan\left(\tfrac{\pi f_d}{R}\right)$$

For audio signals in the range of human hearing, the one finds that the values

$$f_u = 20000\,\text{Hz} \quad \text{and} \quad f_d = 20\,\text{Hz}$$

produce a reasonable envelope.

Vocoder interface

The public interface for the vocoder class should be declared as

```
class Vocoder : public Filter {
  public:
    Vocoder(int N=1, float R=44100);
    ~Vocoder(void);
    int bandCount(void) const;
    void setOffset(float p);
    float operator()(float x);
};
```

`Vocoder(N,R)` — (constructor) creates a channel vocoder object with N frequency bands with equal Q–factors. R is the sampling rate. You are allowed to ignore the passed–in value of N, and choose a fixed number of frequency bands instead, although you must use at least 8 frequency bands. The frequency bands should span a range of about 80 Hz to 4000 Hz (you can use different values if you wish).

`~Vocoder()` — (destructor) destroys a vocoder object. Depending on your implementation, this may be trivial.

`bandCount()` — returns the actual number of frequency bands used by the vocoder.

`setOffset(p)` — sets the frequency multiplier of the carrier signals to $\mu = 2^{p/1200}$, where p is given in cents.

`operator()(x)` — (filter function) returns the output of the channel vocoder, given the input value x. The n–th call to this function returns the n–th output sample.

You are free to declare the *private* portion of the class as you see fit, but the public portion must be exactly as delineated above: you may not remove, add, or modify any of the functionality.

If you choose to implement the channel vocoder, you should submit two files: (1) the header file `Vocoder.h`, and (2) the source file `Vocoder.cpp`. You may include the `Filter.h`, `Vocoder.h`, and `BandPass2.h` header files, as well as any standard `C++` header file.