Chapter 11. Equalizer

An *equalizer* changes the magnitude of frequencies in selected bands, while leaving other frequency bands unchanged. It can be implemented with a single frequency filter, similarly to the high pass filter employed in the bass chorus. Calculating the coefficients of the equalizer filter is only slightly more complex.

A simple equalizer is presented in chapter 11 in volume 1. The equalizer frequency filter is the sum of several separate filters. Each of these filters passes the frequencies in a specific frequency band and stops the frequencies outside of that band. Applying gain to these individual filters means applying gain to the frequencies in the corresponding frequency band, but not to any other frequencies. The output of the equalizer filter is equivalent to the sum of the output of individual filters. With the additional gain provided by the user, the equalizer increases or decreases the gain of frequencies in specific frequency bands.

11.1. Low pass, high pass, and band pass filters

The following are filter computations used in the equalizer. We reexamine the computations for low pass and high pass filters and present computations for band pass and band stop filters.

Code 60. Low pass filter

```
public static void computeLowPassFilter(float [] a, float fc, float fs)
{
   // This function creates a finite impulse response low pass filter.  The
   // filter coefficients are placed in the floating-point array a.  The length
   // of the filter is the length of a.  fc is the cutoff frequency of the
   // filter.  fs is the sampling frequency

   // Calculate the coefficients of the low pass filter
   for(int i = 0; i < a.length; i++)
   {
      if(i == (a.length - 1) / 2)
         a[i] = 2 * fc / fs;
      else
         a[i] = (float) (Math.sin(2D * Math.PI * fc * (i - (a.length - 1) / 2)
            / fs) / (Math.PI * (i - (a.length - 1) / 2)));
   }

   // Apply a Blackman window to the low pass filter to reduce its Gibbs
   // phenomenon ripples
   blackmanWindow(a);
}
```

The difference between this computation and the computation used in creating the bass chorus high pass filter is the Blackman window. We window the low pass filter to reduce the Gibbs phenomenon ripples characteristic of finite impulse response filters. Although adding a window is not required, it makes the magnitude response of the equalizer smoother.

Code 61. Blackman window

```
public static void blackmanWindow(float [] a)
{
   // This function applies a Blackman window to the filter a.  The length of the
   // window is the same as the length of a.  The coefficients of the filter a
   // must be computed beforehand

   for(int i = 0; i < a.length; i++)
      a[i] *= (0.42 - 0.5D * Math.cos((2D * Math.PI * i)
         / (double) (a.length - 1))) + 0.08 * Math.cos((4D * Math.PI * i)
         / (double) (a.length - 1));
}
```

There are many possible windows and the choice of window depends on the task at hand. For example, if we want a very smooth magnitude response of the equalizer, we might want to add a Bartlett-Hann window. We can do so with the understanding that the magnitude response of a filter with the Bartlett-Hann window may have large transition bands and we therefore cannot create an equalizer that produces sharp changes from the gain in one frequency band to the gain in a neighboring frequency band. The Bartlett-Hann window can be computed as follows.

Code 62. Bartlett-Hann window

```
public static void bartlettHannWindow(float [] a)
{
   // This function applies a Blackman window to the filter a.  The length of the
   // window is the same as the length of a.  The coefficients of the filter a
   // must be computed beforehand

   for(int i = 0; i < a.length; i++)
      a[i] *= 0.62 - 0.48 * Math.abs((double) i
         / (a.length - 1)) - 0.38 * Math.cos((2D * Math.PI * i)
         / (double) (a.length - 1));
}
```

A high pass filter can be computed from the low pass filter as before.

Code 63. High pass filter

```
public static void computeHighPassFilter(float [] a, float fc, float fs)
{
   // This function creates a finite impulse response high pass filter.  The
   // filter coefficients are placed in the floating-point array a.  The length
   // of the filter is the length of a.  fc is the cutoff frequency of the
   // filter.  fs is the sampling frequency
   // Create a low pass filter.  Note that we are inverting a low pass filter
   // with a window and so the high pass filter computed here is also windowed
   computeLowPassFilter(a, fc, fs);

   // Invert the low pass filter into a high pass filter
   for(int i = 0; i < a.length; i++)
      a[i] *= -1F;
   a[(a.length - 1) / 2]++;
}
```

A band stop filter is the sum of a low pass filter and a high pass filter. Here, of course, we expect that if **fclow** is the lower cutoff frequency of the band stop filter and **fchigh** is the higher cutoff frequency, then **fchigh** > **fclow**.

Code 64. Band stop filter

```
public static void computeBandStopFilter(float [] a, float fclow, float fchigh,
   float fs)
{
   // This function creates a finite impulse response band stop filter.  The
   // filter coefficients are placed in the floating-point array a.  The length
   // of the filter is the length of a.  fclow is the bottom cutoff frequency of
   // the filter.  fchigh is the top cutoff frequency of the filter. fs is the
   // sampling frequency

   // Compute a low pass filter at the lower cutoff frequency
   computeLowPassFilter(a, fclow, fs);

   // Compute a high pass filter at the higher cutoff frequency
   float [] b = new float [a.length];
   computeHighPassFilter(b, fchigh, fs);

   // Sum the two filters
   for (int i = 0; i < a.length; i++)
      a[i] = a[i] + b[i];
}
```

A band pass filter is an inverted band stop filter.

Code 65. Band pass filter

```
public static void computeBandPassFilter(float [] a, float fclow, float fchigh,
   float fs)
{
   // This function creates a finite impulse response band pass filter.  The
   // filter coefficients are placed in the floating-point array a.  The length
   // of the filter is the length of a.  fclow is the bottom cutoff frequency of
   // the filter.  fchigh is the top cutoff frequency of the filter. fs is the
   // sampling frequency
   // Create a band stop filter
   computeBandStopFilter(a, fclow, fchigh, fs);

   // Invert the band stop filter into a band pass filter
   for(int i = 0; i < a.length; i++)
      a[i] *= -1F;
   a[(a.length - 1) / 2]++;
}
```

11.2. The equalizing filter

Conceptually, the equalizer presented here is a collection of low pass, high pass, and band pass filters. These filters split the incoming signal into several signals, each fitting into a separate frequency interval. Thus, some filters split out the low frequencies, some filters split out the mid frequencies, and some filters split the high frequencies. We can use as many such frequency bands as we wish. Once the signals are separated, the equalizer applies some gain to each of them (e.g., to lower mids and boost highs) and puts them back together.

In practice, all equalizer computations are linear, since we are using finite impulse response filters. We can change the order of tasks in the pseudo algorithm in the previous paragraph. Instead of splitting the signal, applying gain, and putting the signal back together, we take the set of equalizer filters, apply gain to them, and combine them into one single equalization filter. We then simply apply this one filter to the signal.

The equalizer presented below is simple. Frequency bands are fixed and cannot be adjusted by the user. The filter length, which determines the precision of the equalizer, is also fixed and cannot be changed by the user. The user can only change the gains applied to each of the frequency bands in the equalizer.

In practice, an equalizer may allow the user to adjust not only the gains on frequency bands, but also the bands themselves and the length of the filter. The structure of the equalizer below can be adjusted easily to allow for such changes. For example, a fixed band and fixed precision equalizer can compute the individual frequency band filters once, in its constructor. Below, these computations are performed during the call to the **`apply`** function. The equalizer can be changed easily to allow changes to bands and precision. Where such changes should occur is noted below.

The following are the member data of the equalizer.

`int NUMBANDS` – This is the number of bands in the equalizer. In a fixed band equalizer, this variable is fixed. If the user can change the number of bands, the user should also be allowed to change the actual bands **`BANDS`** below.

`int FILTERLENGTH` – This is the length of the filter. The code below works whether this is a final constant or a variable that the user can change. Modifying this variable changes the precision of the equalizer.

`float [] BANDS` – These are the bands of the equalizer. Suppose, for example, that we have a 10-band equalizer (i.e., **`NUMBANDS`** = 10). We wish to split the frequency spectrum between 20 Hz and 22000 Hz – the lower and upper limits of the human hearing – into 10 bands. We want to do so evenly, but exponentially, as this is how the human ear perceives frequencies. We set the top of the first band at $(22000 / 20)^{1/10} = 40$ Hz, the top of the second band at $(22000 / 20)^{2/10} = 81$ Hz and so on. The length of **`BANDS`** is **`NUMBANDS`**. Both variables can be final or subject to user changes.

`boolean m_filtersSet` – This variable shows whether the filter for each of the frequency bands has been computed (**`true`**) or needs to be computed (**`false`**). In a fixed band equalizer, the filters for each frequency band can be computed only once, in the equalizer constructor. If the user changes the equalizer bands, this variable should be set to **`false`** so that the equalizer can recompute the individual band filters accordingly.

`boolean m_totalFilterSet` – This variable shows whether the equalizer filter has been computed (**`true`**) or needs to be recomputed (**`false`**). Even if the individual band filters stay the same, the total equalizer filter must be recomputed any time the user changes the gains applied to the frequency bands.

`float [] m_h` – This array contains the equalizer filter coefficients. This array can be allocated once in a fixed band equalizer, but must be allocated again, if the user changes **`FILTERLENGTH`**.

`Band [] m_bands` – These are the bands of the equalizer. Band is the following simple class.

Code 66. An equalizer band

```
public class Band
{
   public float m_lowCutoff;     // the bottom cutoff frequency
   public float m_highCutoff;    // the top cutoff frequency
   public float m_gain;          // the gain applied to the band
   public float m_h[];           // the band pass filter
}
```

The members of **`Band`** are as follows.

`float m_lowCutoff` – This is the bottom end of the frequency band and the bottom cutoff frequency for the corresponding filter.

`float m_highCutoff` – This is the top end of the frequency band and the top cutoff frequency of the corresponding filter.

`float m_gain` – This is the gain that the user applies to the signal in the frequency band. The gain is measured in decibels.

`float [] m_h` – This is the low pass, high pass, or band pass filter that is used to "extract" the frequencies in the frequency band.

The equalizer constructor sets up the frequency bands as follows.

Code 67. Equalizer bands

```
m_bands = new Band[NUMBANDS];
for(int i = 0; i < NUMBANDS; i++)
{
   m_bands[i].m_h = null;
   m_bands[i].m_gain = 0;
}
m_bands[0].m_lowCutoff = 0;
m_bands[0].m_highCutoff = BANDS[0];
for(int i = 1; i < NUMBANDS; i++)
{
   m_bands[i].m_lowCutoff = BANDS[i - 1];
   m_bands[i].m_highCutoff = BANDS[i];
}
```

The individual band filters can be computed as follows.

Code 68. Equalizer band filters

```
public void computeFilter(AudioFormat format)
{
   // The audio format of incoming data is needed, as the filter calculations
   // use the sampling rate

   // Allocate the filter arrays
   for(int i = 0; i < NUMBANDS; i++)
   {
      if (m_bands[i].m_h == null)
         m_bands[i].m_h = new float[FILTERLENGTH];
      if (m_bands[i].m_h.length != FILTERLENGTH)
         m_bands[i].m_h = new float[FILTERLENGTH];
   }

   // The first (lowest) band uses a low pass filter
   computeLowPassFilter(m_bands[0].m_h, m_bands[0].m_highCutoff,
      format.getSampleRate());

   // The last (highest) band uses a high pass filter
   computeHighPassFilter(m_bands[NUMBANDS - 1].m_h,
      m_bands[NUMBANDS - 1].m_lowCutoff, format.getSampleRate());

   // The remaining bands use band pass filters
   for(int i = 1; i < NUMBANDS - 1; i++)
   {
      computeBandPassFilter(m_bands[i].m_h, m_bands[i].m_lowCutoff,
         m_bands[i].m_highCutoff, format.getSampleRate());
   }

   m_filtersSet = true;
}
```

The total equalizer filter is simply the sum of the individual band filters, with the corresponding gain applied.

Code 69. Total equalizer filter

```
public void computeTotalFilter()
{
   // allocate and initialize the total filter array
   if(m_h == null)
      m_h = new float[FILTERLENGTH];
   else if(m_h.length != FILTERLENGTH)
      m_h = new float[FILTERLENGTH];
   for(int i = 0; i < FILTERLENGTH; i++)
      m_h[i] = 0.0F;

   // Simply sum the individual band filters with the appropriate gain
   // the gain is assumed to be in decibels and must be converted to a multiple
   for(int i = 0; i < NUMBANDS; i++)
   {
       float gain = (float) Math.pow(10D, m_bands[i].m_gain / 20F);
       for(int j = 0; j < FILTERLENGTH; j++)
          m_h[j] += gain * m_bands[i].m_h[j];
   }

   m_totalFilterSet = true;
}
```

11.3. Implementation of the equalizer

The following is the constructor of the equalizer.

Code 70. Equalizer constructor

```
public Equalizer()
{
   // Create the equalizer bands based on the information in BANDS
   m_bands = new Band[NUMBANDS];
   for(int i = 0; i < NUMBANDS; i++)
   {
      m_bands[i].m_h = null;
      m_bands[i].m_gain = 0;
   }
   m_bands[0].m_lowCutoff = 0;
   m_bands[0].m_highCutoff = BANDS[0];
   for(int i = 1; i < NUMBANDS; i++)
   {
      m_bands[i].m_lowCutoff = BANDS[i - 1];
      m_bands[i].m_highCutoff = BANDS[i];
   }

   // The total equalizer filter is computed during playback, in case the
   // user makes changes to the gains of individual bands (or to the bands
   // themselves, if this is a parametric equalizer)
   m_h = null;
   m_filtersSet = false;
   m_totalFilterSet = false;

   m_storeBuffers = new Vector<byte []>(0);
}
```

The following code is executed at the beginning of playback.

Code 71. Equalizer at the beginning of playback

```
public void startPlay()
{
   m_filtersSet = false;
   m_totalFilterSet = false;
   m_storeBuffers.setSize(0);
}
```

Once the filter is computed, the implementation of the equalizer is the same as the application of the high pass filter in the bass chorus.

Code 72. Equalizer

```
public void apply(byte [] dry, byte [] wet, byte [] control, AudioFormat format,
   double time)
{
   // Calculate the individual filters, if needed.  This must happen if the user
   // changes the filter length (precision) or the position of the frequency
   // bands
   if(!m_filtersSet)
      computeFilter(format);

   // Calculate the total equalizer filter, if needed.  This must happen if the
   // user changes the filter length, the frequency bands, or the gain applied to
   // any of the bands
   if(!m_totalFilterSet)
      computeTotalFilter();
```

```
// Initialize variables
int blockAlign = (format.getChannels() * format.getSampleSizeInBits()) / 8;
float sFrom = 0.0F;
float sTo = 0.0F;

// Store incoming audio data
byte [] storeBuffer = new byte [dry.length];
System.arraycopy(dry, 0, storeBuffer, 0, dry.length);
m_storeBuffers.add(storeBuffer);

// Calculate how far back the equalizer should look to apply the filter.  At
// each sample, the filter uses the previous FILTERLENGTH samples
int curBufferStart = m_storeBuffers.size() - 1;
int curByteStart = -FILTERLENGTH * blockAlign;
while (curByteStart < 0)
{
   curByteStart += dry.length;
   curBufferStart--;
}
byte bufferFromStart[] = null;
if(curBufferStart >= 0)
   bufferFromStart = m_storeBuffers.get(curBufferStart);

// Remove unused past audio buffers
while (curBufferStart > 0)
{
   m_storeBuffers.remove(0);
   curBufferStart--;
}

// Apply the filter at each sample
for(int i = 0; i < dry.length; i += blockAlign, curByteStart += blockAlign)
{
   if(curByteStart >= dry.length)
   {
      curBufferStart++;
      bufferFromStart = m_storeBuffers.get(curBufferStart);
      curByteStart = 0;
   }

   int curBuffer = curBufferStart;
   int curByte = curByteStart;
   byte bufferFrom[] = bufferFromStart;
   sTo = 0.0F;

   // Apply the filter at a sample
```

```
      for(int j = 0; j < FILTERLENGTH; j++, curByte += blockAlign)
      {
         if(curByte >= dry.length)
         {
            curBuffer++;
            bufferFrom = m_storeBuffers.get(curBuffer);
            curByte = 0;
         }
         if(curBufferStart >= 0)
         {
            sFrom = (short)(((bufferFrom[curByte + 1] & 0xff) << 8)
               + (bufferFrom[curByte] & 0xff));
            sTo += sFrom * m_h[j];
         }
      }

      // Store the output
      sTo = Math.max(Math.min(sTo, Short.MAX_VALUE), Short.MIN_VALUE);
      wet[i] = (byte)((int)sTo & 0xff);
      wet[i + 1] = (byte)((int)sTo >>> 8 & 0xff);
   }
}
```

Note that the equalizer uses filters of equal length and, for neighboring filters, with the same cutoff frequencies. This ensures that a smooth transition from one frequency band to another. If, for example, the user specifies 0 dB gain in each of the frequency bands, this equalizer should produce a flat magnitude response (i.e., without any peaks, dips, or ripples).

11.4. Magnitude response of the equalizer

Suppose that we want to calculate how the filter affects frequencies in the frequency spectrum. The actual magnitude response of the equalizer is different than the desired one. Perhaps we want to draw this actual response for the user.

The following function calculates the magnitude response of the filter for a frequency.

Code 73. Magnitude response of the equalizer

```
public float magnitudeResponse(float [] a, float f, float fs)
{
   // a is the filter, the magnitude response of which we want to compute
   // f is the frequency at which the magnitude response is computed
   // fs is the sampling frequency

   // Compute the angular frequency corresponding to the frequency f
   double w = 2 * Math.PI * f / fs;

   // Since our filters are symmetric around their middles, calculate the
   // the middle to reduce the computations that follow
   int M = (a.length - 1) / 2;

   // Compute the magnitude response at f
   float output = a[M];
   for(int i = 0; i < M; i++)
      output += 2 * a[i] * Math.cos(w * (i - M));

   // The result is the magnitude response of the filter, as a multiple
   // (i.e., a response of 1 means there were no changes to the magnitude
   // of f
   return Math.abs(output);
}
```

11.5. Phase response of the equalizer

Note that the equalizer does not allow dry and wet mix. There should be no reason to mix the original and the equalized signal and hence there is no reason to compute the phase.

Code 74. Dry and wet mix in the equalizer

```
public boolean allowsDryWetMix()
{
   return false;
}
```

If we wanted to allow dry and wet mix, we must recognize that the equalized wet signal is delayed by **`(FILTERLENGTH – 1) / 2`**, assuming **`FILTERLENGTH`** is an odd number. This means that we also must delay the original signal by the same number of samples.