Chapter 7  PROGRAMMING EXERCISE

Creating FIR Filters in C++

FIR (finite impulse response) filters are time-domain audio filters that work by applying convolution to audio data represented in the time domain. Their operation is sketched in Figure 1.

Convolve x with h to get y.
\[ y(n) = h(n) \otimes x(n) \]

\[ h = [a_0, a_1, a_2, a_3, a_4] \]

\[ y(5) = a_4 x_0 + a_3 x_1 + a_2 x_2 + a_1 x_3 + a_0 x_4 \]

Filter coefficients

<table>
<thead>
<tr>
<th>a_4</th>
<th>a_3</th>
<th>a_2</th>
<th>a_1</th>
<th>a_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>x_0</td>
<td>x_1</td>
<td>x_2</td>
<td>x_3</td>
<td>x_4</td>
</tr>
</tbody>
</table>

Audio samples

x_5 x_6 x_7 x_8 ...

To compute y(6), slide x to the left.

Figure 1 Convolution with an FIR filter

The filter is essentially an array of multipliers that is run as a “mask” across the audio samples. The multipliers are sometimes referred to as taps. The number of taps affects the precision of the filter. In general, the more taps there are, the more precise the filter is in filtering out unwanted frequencies. However, more taps require more memory and computation time. Filtering is accomplished by giving each sample a new value based on multiplying the filter values by neighboring samples and summing them, as shown in the figure. The operation is defined mathematically as follows:

Given a digital audio signal \( x(t) \) of size \( L \) in the time domain and an FIR filter \( h(k) \) of size \( N \), the FIR filtering operation is defined by

\[ y(t) = h(k) \otimes x(t) = \sum_{k=0}^{N-1} h(k)x(t-k) \]

where \( x(t-k) = 0 \) if \( t-k < 0 \).

\( y(t) \) is the filtered audio signal.
A graph of the values in an FIR filter takes the shape shown in Figure 2. This is the shape of a sinc function. A sinc function is the product of a sine function and a monotonically decreasing function. Its basic form is

$$sinc(x) = \begin{cases} \frac{\sin(x)}{x} & \text{for } x \neq 0 \\ 1 & \text{for } x = 0 \end{cases}$$

Figure 2 Graph of a sinc function, the shape of an FIR filter

In this exercise, we explore how you create a filter of the proper shape in order to filter out the frequencies you don’t want and keep the ones you do want.

There are four commonly-used types of FIR filters: low-pass, high-pass, bandpass, and bandstop. A low-pass filter leaves in only frequencies below a cutoff frequency $f_c$. A high-pass filter leaves in only frequencies above a cutoff frequency $f_c$. A bandpass filter leaves in only frequencies between $f_1$ and $f_2$. A bandstop filter leaves in only frequencies below $f_1$ or above $f_2$.

The algorithms for low-pass, high-pass, bandpass, and bandstop filters are given below. These algorithms create the filters. After the filters are created, you have to apply them to the audio data with convolution, so you’ll also have to write a convolution function.
algorithm FIR_low_pass filter
/*
Input:
  f_c, the cutoff frequency for the low-pass filter, in Hz
  f_samp, sampling frequency of the audio signal to be filtered, in Hz
  N, the order of the filter; assume N is odd
Output:
  a low-pass FIR filter in the form of an N-element array */
{
  //Normalize f_c and ω_c so that π is equal to the Nyquist angular frequency
  f_c = f_c/f_samp
  ω_c = 2*π*f_c
  middle = N/2 /*Integer division, dropping remainder*/
  for i = −N/2 to N/2
    if (i == 0) fltr(middle) = fltr(middle) = 2*f_c
      else fltr(i + middle) = sin(ω_c*i)/(π*i)
  //Now apply a windowing function to taper the edges of the filter, e.g.
  //Hamming, Hanning, or Blackman
}

Algorithm 1 Low-pass filter

algorithm FIR_high_pass filter
/*
Input:
  f_c, the cutoff frequency for the high pass filter, in Hz
  f_samp, sampling frequency of the audio signal to be filtered, in Hz
  N, the order of the filter; assume N is odd
Output:
  a high-pass FIR filter in the form of an N-element array */
{
  //Normalize f_c and ω_c so that π is equal to the Nyquist angular frequency
  f_c = f_c/f_samp
  ω_c = 2*π*f_c
  middle = N/2 /*Integer division, dropping remainder*/
  for i = −N/2 to N/2
    if (i == 0) fltr(middle) = fltr(middle) = 1 - 2*f_c
      else fltr(i + middle) = -sin(ω_c*i)/(π*i)
  //Now apply a windowing function to taper the edges of the filter, e.g.
  //Hamming, Hanning, or Blackman
}

Algorithm 2 High-pass filter
algorithm FIR_bandpass filter
/*
 Input:
 f1, the lowest frequency to be included, in Hz
 f2, the highest frequency to be included, in Hz
 f_samp, sampling frequency of the audio signal to be filtered, in Hz
 N, the order of the filter; assume N is odd
 Output:
 a bandpass FIR filter in the form of an N-element array */
{
 //Normalize f_c and ω _c so that π is equal to the Nyquist angular frequency
 f1_c = f1/f_samp
 f2_c = f2/f_samp
 ω1_c = 2*π*f1_c
 ω2_c = 2*π*f2_c
 middle = N/2 /*Integer division, dropping remainder*/
 for i = −N/2 to N/2
   if (i == 0) fltr(middle) = 2*f2_c - 2*f1_c
   else
     fltr(i + middle) = sin(ω1_c*i)/(π*i) -
                        sin(ω2_c*i)/(π*i)
//Now apply a windowing function to taper the edges of the filter, e.g.
//Hamming, Hanning, or Blackman
}

Algorithm 3 Bandpass filter

algorithm FIR_bandstop filter
/*
 Input:
 f1, the highest frequency to be included in the bottom band, in Hz
 f2, the lowest frequency to be included in the top band, in Hz
 Everything from f1 to f2 will be filtered out
 f_samp, sampling frequency of the audio signal to be filtered, in Hz
 N, the order of the filter; assume N is odd
 Output:
 a bandstop FIR filter in the form of an N-element array */
{
 //Normalize f_c and ω _c so that π is equal to the Nyquist angular frequency
 f1_c = f1/f_samp
 f2_c = f2/f_samp
 ω1_c = 2*π*f1_c
 ω2_c = 2*π*f2_c
 middle = N/2 /*Integer division, dropping remainder*/
 for i = −N/2 to N/2
   if (i == 0) fltr(middle) = 1 - 2*(f2_c - f1_c)
   else
     fltr(i + middle) = sin(ω1_c*i)/(π*i) -
                        sin(ω2_c*i)/(π*i)
//Now apply a windowing function to taper the edges of the filter, e.g.
//Hamming, Hanning, or Blackman
}

Algorithm 4 Bandstop filter
Your assignment is to do the following for each type of filter given in the algorithms. Filters of size $N=1025$ should be fine, but you can experiment with other sizes as well.

- Implement the filter in C++.
- Write a convolution function.
- Try the filter on an audio file by convolving the audio file with the filter using your convolution function.
- Listen to the audio file before and after to see if you notice a difference.
- If you have MATLAB, Octave, or some equivalent program, you may want to export your filter and your audio files and graph them to verify your results. Your filters should have the shape of a sinc function. In MATLAB, you can take the Fourier transform of the audio data before and after filtering to verify that the correct frequencies have been filtered out.