Appendix B

Additional Hands-On Experiments and Applications

This appendix introduces several hands-on experiments for specific applications related to the FIR filtering discussed in Chapter 6, IIR filtering discussed in Chapter 7, FFT discussed in Chapter 8, and adaptive filtering discussed in Chapter 9.

B.1 3-D AUDIO-SOUND LOCALIZER USING FINITE-IMPULSE RESPONSE FILTERS

This experiment implements a 3-D audio-sound localizer, which plays two sound files at specific locations around the listener's head using a headset. This experiment is designed to allow the user to achieve the following objectives:

1. Understand the principle of 3-D audio-signal processing and its application in headset listening. A detailed description of 3-D audio processing can be found in [1, 2].
2. Evaluate the effects of a fixed-point implementation of 3-D audio using the fixed-point tools in MATLAB.
3. Implement 3-D audio on C5000 processors using a mixed C-and-assembly program, where the C5000 assembly functions are available in the DSPLIB. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

Note that if a headset with gain control is used for the experiment, the gain should be set to a comfortable listening level, and the same gain should be used during testing.

B.1.1 Introduction

Unlike stereo sound, which allows only lateral listening, 3-D sound allows the listener to listen to the sound around the listener's head in every direction using only two loudspeakers. This effect is achieved by using a set of FIR filters, commonly known as head-related transfer functions (HRTF). A monosound source at the fixed location can be perceived at any position in 3-D space by convolving the original signal with the impulse response of two HRTF filters (left and right) corresponding to that position.

In this experiment, two independent sound sources are convolved with a set of HRTF filters at 0° elevation, 150° clockwise and anticlockwise from the front of the listener. The HRTF filters are measured in [1] and are available for download from the Web. The outputs from the left HRTF filters are summed to form the left channel signal for playback over the headset. Similarly, the right HRTF filters' outputs are added to form the right channel signal. In most 3-D sound applications, the HRTF is realized by an FIR filter; thus, the impulse response of an HRTF is represented by the coefficients of the filter.

B.1.2 Lab Experiment Using MATLAB/Simulink

Open the Simulink file `ex6_i.mdl` (or the MATLAB file `ex6_i.m`) and examine the function of each block. Use the following data files for this experiment [1]: `female_footsteps2.wav` as sound #1, `door-open2.wav` as sound #2, and the HRTF file `hrtf.mat`, which contains HRTF data for `left_150`, `right_150`, `left_210`, `right_210`. Complete the following exercises, answer the related questions, and benchmark the results:

1. Plot the magnitude responses of four HRTF filters using MATLAB, and comment on the differences.
2. Play back the output files (`out_left.mat` and `out_right.mat`) using MATLAB and listen to the sound. Can you perceive the two signal sources at two different locations? Make sure to use the correct sampling frequency for playback.
3. Compare the spectra of the input and output signals, and comment on the differences.
4. Adjust the gains of the HRTF filters, and observe the differences in performance.
5. Convert the double-precision, floating-point Simulink blocks to fixed-point Q.15 format using Fixed-Point Blockset. Examine the effects of using a fixed-point implementation in comparison with a double-precision, floating-point simulation.
6. Finish converting to the fixed-point implementation, and then listen to the processed signals, and observe the perceptual differences in comparison with

the floating-point results. In addition, truncate the length of the HRTF filters to 64, and observe the perception differences.

7. Reduce the wordlength from 16 bits to 8 bits using Q.7 format, and evaluate the performance.

B.1.3 Lab Experiment Using the C5000 Code Composer Studio

Use the C5000 DSPLIB to write a C program that calls the library functions to implement HRTF filters. Create, build, and run the project in CCS. Remember to add the relevant files into the project before building the executable code. Complete the following exercises, and benchmark the results:

1. Access the input signals via FILE I/O. Implement the C5000 version of the 3-D audio localizer using FIR filters. Save the results in data files, and listen to the results. Since there is a memory limit on C5000 processors, we may store only partial output samples in the file when processing a long data file. Observe the differences between the MATLAB and CCS results.
2. Profile the code of the 3-D audio localizer, and benchmark the cycle count, MIPS used, and data- and program-memory requirements.
3. Use different optimization levels in CCS, and observe the differences in the benchmark.

B.1.4 Additional Exercises

1. Download other HRTF files from the website [1], and use more channels for 3-D audio processing.
2. Insert a simple reverberation algorithm [2] after the HRTF filter to enhance out-of-head listening.
3. Determine if loudspeaker listening is required. If it is, the user needs a cross-talk canceller that cancels crosstalk signals from the loudspeakers to the ears. Refer to [2] for details on cross-talk cancellers. Comment on the differences between 3-D audio listening using headphones and loudspeakers.

B.2 GRAPHIC EQUALIZER USING FINITE-IMPULSE RESPONSE FILTERS

An L-band graphic equalizer can be implemented using L FIR filters connected in parallel [3]. The most important objectives of this experiment are as follows:

1. Understand the principle of the graphic equalizer. An important consideration in designing a graphic equalizer is the frequency specification for each filter, which includes passband and stopband frequencies, ripples, and the filter order N.
2. Study the gain control at the output of each filter. The gain allows the user to amplify or attenuate the signal at each band in order to enhance the quality of the signal.

3. Ensure that no overflow occurs when adding the outputs from L filters in a fixed-point implementation. Signal scaling is required to ensure proper operation of the fixed-point graphic equalizer.
4. Implement the designed graphic equalizer on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.2.1 Introduction

A graphic equalizer amplifies or attenuates the specific frequency contents of an audio signal to compensate for signal components that are distorted by recording devices, boosts some frequency contents of the signal to make it sound better, or removes undesired bandlimited noise. The block diagram of an L-band graphic equalizer is shown in Fig. B.1. The input signal, $x(n)$, may be added directly to the output in applications that allow the original signal to pass through when all of the gains at the outputs of bandpass filters are set to 0. In other applications, such as the removal of bandlimited noise, only bandpass filter outputs with attenuation are added to form the overall output, $y(n)$.

In this experiment, we investigate the design and implementation of an octave-band graphic equalizer that covers the frequency range from 22 Hz to 22,500 Hz. The sampling rate is 44,100 Hz. The frequency specifications for the 10-band, octave graphic equalizer are summarized in Table B.1.

It is important to note that when designing a bandpass filter at a low frequency with a high sampling rate, a very high order FIR filter is required. One way to overcome this problem is to use the multirate filter-design technique to obtain a lower-order filter. Another way is to trade the frequency selection for a reduced computational load. For example, in this experiment we group four low-frequency bands (bands #1 to #4) into a single lowpass filter with a cutoff frequency of 353 Hz.

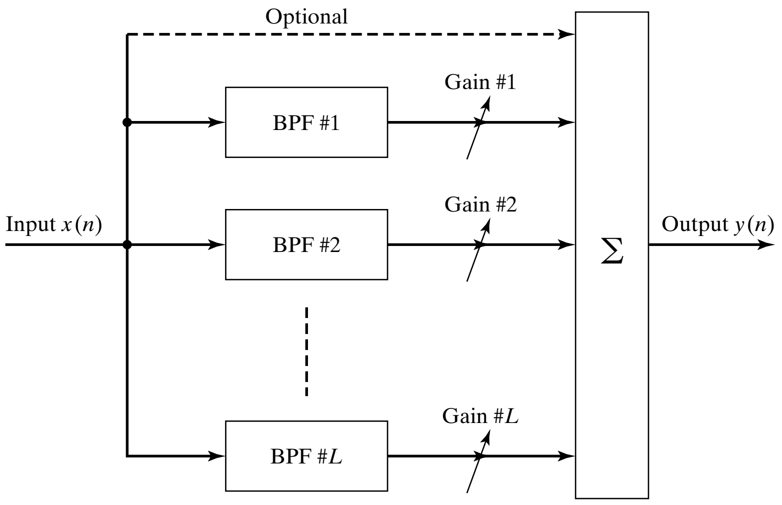

Figure B.1 Block diagram of an L-band graphic equalizer

TABLE B.1 Frequency Specifications for a 10-Band, Octave Graphic Equalizer

Band number	Center frequency	Passband (Hz)
#1	31.5	22–44
#2	63	44–88
#3	125	88–176
#4	250	176–353
#5	500	353–707
#6	1,000	707–1,414
#7	2,000	1,414–2,825
#8	4,000	2,825–5,650
#9	8,000	5,650–11,300
#10	16,000	11,300–22,500

Other filters remain unchanged (and they remain as listed in Table B.1). The passband and stopband ripples are chosen as 1 dB and 60 dB, respectively. The gain of the i-th bandpass filter, Gain(i), is set to +10 dB. A default gain, $\text{Gain}_{\text{default}}(i) = 1/\text{Gain}(i)$, is applied to the output of the i-th filter so that the default gain is referenced at 0 dB. Any increase or decrease is considered an amplification or attenuation from this reference. In this experiment, we precompute the gain that corresponds to ±3 dB, ±6 dB, and ±9 dB, and the user can simply apply these gains to the output of each band.

B.2.2 Lab Experiment Using MATLAB/Simulink

Complete the following exercises, answer the related questions, and benchmark the results:

1. Open the Simulink file `ex6_ii.mdl` and use FDATool to design seven octave filters (the lower four bands are combined into a single filter). Use the equiripple and window methods to design these FIR filters, and write down the required order for each band. Which design method results in a lower filter order? Refer to the M-file `geq.m` for the MATLAB version of the program.
2. Examine the magnitude responses of the designed bandpass filters to verify the frequency specifications given in Table B.1.
3. Amplify the low-frequency bands and attenuate the high-frequency bands using the graphic equalizer. Apply the audio signal in the file `drums.wav` [18] to the equalizer, and save the equalized signal as `drums_eq.wav`. Examine the differences between the input and output signals in terms of their spectra and perception.
4. Modify the Simulink file to implement the fixed-point (Q.15 format) version of the seven-band graphic equalizer using the Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.2.3 Lab Experiment Using the C5000 Code Composer Studio

Use the C5000 DSPLIB to write a C program that calls the library functions to implement the designed seven-band graphic equalizer. Create, build, and run the project in CCS. Use the same gain setting as the one used in the Simulink code. Complete the following exercises, and benchmark the results:

1. Access the input signal `drums.dat` via FILE I/O. Save the output in a data file, and listen to the result. Observe the differences between the MATLAB and CCS results.
2. Profile the seven-band octave graphic-equalizer code, and benchmark the cycle count, MIPS used, and data- and program-memory requirements.
3. Use different optimization levels in CCS, and observe the differences in the benchmark.

B.2.4 Additional Exercises

1. Extend the octave graphic equalizer to a 1/3-octave graphic equalizer. The 1/3-octave band is commonly used in applications that require the filter bandwidth to be matched to the human auditory system. The lowest and highest frequencies are 22 Hz and 22,500 Hz, respectively. Starting from $f_{p1} = 22$ Hz, the passband frequencies from f_{p1} to f_{p2} can be computed as

$$f_{p2} = 2^{1/3} f_{p1}. \tag{B.2.1}$$

   How many bands can fit into the frequency range from 22 to 22,500 Hz?
2. Use the 1/3-octave graphic equalizer to reduce the bandlimited noise in the corrupted audio signal `noisy_drums.wav`. (Hint: The user can determine the contents of the bandlimited noise during the silent periods of the signal and use this information to adjust the gains of the graphic equalizer to attenuate the undesired bandlimited noise.)
3. Combine L FIR filters into a single FIR filter if the gain of each filter is fixed and is included in the filter coefficients. This task can be done by expressing the impulse response of each FIR filter in polynomial and adding the corresponding terms to form a single impulse response. By predetermining the gain at each band, we can implement the graphic equalizer using a single FIR filter. This method reduces computational complexity at the expense of gain-control flexibility.

B.3 PERFECT-RECONSTRUCTION FILTERBANK

A time-domain signal can be decomposed into separate frequency components using a transform such as the FFT introduced in Chapter 8 or using a filterbank such as the quadrature-mirror filterbank. Frequency decomposition using a filterbank allows time-domain signal-processing techniques to be used at different frequency channels.

In this experiment, we use a simple two-channel quadrature-mirror filterbank that consists of a lowpass filter (0 to $\pi/2$) and a highpass filter ($\pi/2$ to π) to split the signal into two bands. The analysis filter outputs are downsampled by a factor of two. Assuming there is no distortion or aliasing, we are able to reconstruct the signal by upsampling with a factor of two and then performing synthesis filtering. The most important objectives of this experiment are as follows:

1. Understand the principle of a perfect-reconstruction filterbank. An important consideration in designing analysis and synthesis filters is to ensure that the reconstructed signal is same as the original signal. Detailed information is available in [5, 6].
2. Evaluate the fixed-point implementation of a perfect-reconstruction filter using fixed-point tools in MATLAB and Simulink.
3. Implement the filterbank on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.3.1 Introduction

The block diagram of a perfect-reconstruction filterbank is illustrated in Fig. B.2, where $A_{lp}(z)$ and $A_{hp}(z)$ are the analysis lowpass and highpass filters, and $S_{lp}(z)$ and $S_{hp}(z)$ are the corresponding synthesis lowpass and highpass filters. The analysis filter outputs are downsampled by a factor of two, which can be easily implemented by ignoring every other sample of the filter outputs. In the synthesis end, a factor-of-two upsample is achieved by inserting a zero sample between two consecutive samples and then performing synthesis filtering. The outputs from the synthesis filters are added to form the reconstructed signal, $y(n)$. Efficient techniques for using polyphase filters in implementing analysis and synthesis filters with associated downsampling and upsampling are introduced in Section 6.5.

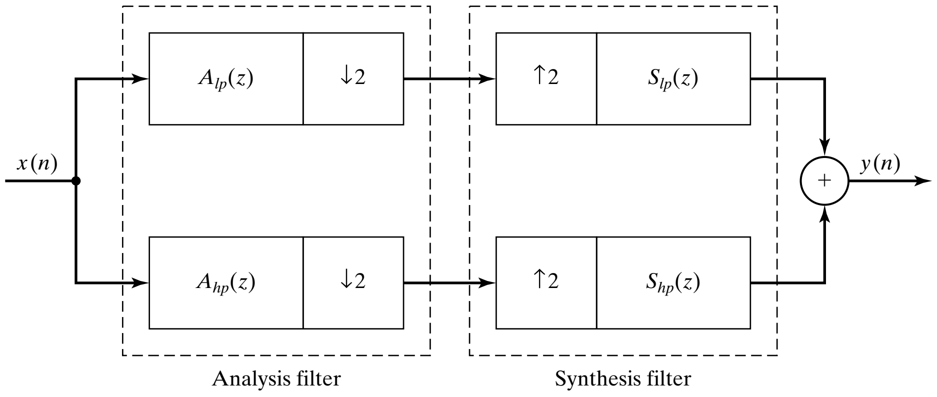

Figure B.2 Block diagram of a perfect-reconstruction filterbank

The z-transform of the input and output signals shown in Fig. B.2 can be expressed as

$$\begin{aligned} Y(z) &= 0.5X(z)[A_{lp}(z)S_{lp}(z) + A_{hp}(z)S_{hp}(z)] \\ &\quad + 0.5X(-z)[A_{lp}(-z)S_{lp}(z) + A_{hp}(-z)S_{hp}(z)], \end{aligned} \tag{B.3.1}$$

where the first term on the right-hand side represents the desired output, and the second term is due to aliasing from downsampling. Ideally, aliasing can be canceled completely if the second term equals zero. That is,

$$A_{lp}(-z)S_{lp}(z) + A_{hp}(-z)S_{hp}(z) = 0. \tag{B.3.2}$$

This aliasing cancellation requires that

$$\frac{A_{lp}(-z)}{A_{hp}(-z)} = -\frac{S_{hp}(z)}{S_{lp}(z)}, \tag{B.3.3}$$

where

$$S_{lp}(z) = -C(z)A_{lp}(-z), S_{hp}(z) = C(z)A_{hp}(-z) \tag{B.3.4}$$

and where $C(z)$ is an arbitrary rational function.

If there is no distortion, the first term in Eq. (B.3.1) can be expressed as

$$A_{lp}(z)S_{lp}(z) + A_{hp}(z)S_{hp}(z) = 2z^{-N}, \tag{B.3.5}$$

where N is the overall delay of the FIR filterbank, and the output signal $y(n) = x(n - N)$ is the delayed version of the input. Therefore, a perfect-reconstruction filterbank can be derived by satisfying the conditions given in Eqs. (B.3.2) and (B.3.5).

The analysis filter $A_{lp}(z)$ is first designed with a cutoff frequency of $\pi/2$. This filter satisfies the power-symmetric condition

$$A_{lp}(z)A_{lp}(z^{-1}) + A_{lp}(-z)A_{lp}(-z^{-1}) = 1. \tag{B.3.6}$$

To achieve perfect reconstruction, the remaining filters, $A_{hp}(z)$, $S_{lp}(z)$, and $S_{hp}(z)$, can be derived from the prototype filter $A_{lp}(z)$ as follows:

$$A_{hp}(z) = z^{-N}A_{lp}(-z^{-1}), \tag{B.3.7a}$$

$$S_{lp}(z) = 2z^{-N}A_{lp}(z^{-1}), \tag{B.3.7b}$$

and

$$S_{hp}(z) = 2z^{-N}A_{hp}(z^{-1}). \tag{B.3.7c}$$

Perfect reconstruction can be verified by substituting Eq. (B.3.7) into Eqs. (B.3.2) and (B.3.5). Therefore, by designing the prototype lowpass filter $A_{lp}(z)$, we can obtain the rest of the filters, $A_{hp}(z)$, $S_{lp}(z)$, and $S_{hp}(z)$, using Eq. (B.3.7). Note that $A_{lp}(z)$ and $A_{hp}(z)$ are mirror images about $\pi/2$, and $S_{lp}(z)$ and $S_{hp}(z)$ are also mirror images. Therefore, these filter pairs are called quadrature-mirror filters (QMFs).

B.3.2 Lab Experiment Using MATLAB/Simulink

We show the design of a two-channel QMF step by step in the MATLAB file `design_qmf.m`. The analysis lowpass filter is designed with the following specifications:

- Sampling frequency $f_s = 8{,}000$ Hz
- Passband frequency fpass = 1,000 Hz (or 0.25π)
- Stopband frequency fstop = 3,000 Hz (or 0.75π)
- Passband ripple $\delta_p = 0.01$
- Stopband ripple $\delta_s = 0.01$

We first design the lowpass filter using the MATLAB function `remez`, as shown in `design_qmf.m`. We then derive the rest of the filters based on Eq. (B.3.7) using MATLAB. Note that highpass analysis and synthesis filters have the same specifications as lowpass analysis filters.

Complete the following exercises, and benchmark the results:

1. Compute the magnitude responses of these designed filters, and determine the aliasing error in dB.
2. Open the Simulink file `ex6_iii.mdl` and use the designed analysis and synthesis filters. Use the speech file `timit1.wav` to examine whether the reconstructed signal `reconstruct.wav` is similar to the original signal.
3. Modify the Simulink file to implement two-channel QMF in fixed-point Q.15 format using Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.3.3 Lab Experiment Using the C5000 Code Composer Studio

Write a C program that calls the library functions in the C5000 DSPLIB to implement the two-channel QMF. Create, build, and run the project in CCS. Complete the following exercises, and benchmark the results:

Step 1. Access the input signal from `timit1.asc` via FILE I/O. Save the output in a data file, and listen to the result. Compare the differences between the MATLAB and CCS outputs.

Step 2. Profile the code that implements the two-channel QMF, and record the cycle count, MIPS usage, and data- and program-memory requirements.

Step 3. Use different optimization levels in CCS, and observe the differences in the benchmark.

B.3.4 Additional Exercises

1. Note that the two-channel QMF can be extended to a three-channel QMF [5] by further dividing the low-frequency band signal into second-level lowpass (0 to $\pi/4$) and highpass ($\pi/4$ to $\pi/2$) filters. The outputs from these filters are downsampled by a factor of two. There is no need to redesign the second-level lowpass and highpass filters since the previous lowpass and highpass filters are designed with symmetrical characteristics and thus can be applied to different sampling frequencies. Similarly, the synthesis portion of the three-channel QMF consists of upsampling and lowpass filtering. This structure is commonly called a tree-structured filterbank, or octave-band QMF. Plot the magnitude responses of the three-channel QMF. Examine the increase in computation and memory usage over the two-channel QMF.
2. Repeat the preceding exercise for the four-channel tree-structured QMF. Plot the magnitude responses of the four-channel QMF. Observe the increase in computation and memory usage over the three-channel QMF.
3. Note that the *N*-channel QMF is commonly used to split a signal into different frequency bands in speech and audio-coding applications. We then encode only the most important frequency bands and use fewer bits to encode other bands, thus achieving an overall bit-rate reduction. We normally ignore high-frequency bands since the signal energy concentrates in lower-frequency bands.
4. Note that other applications that use a filterbank include acoustic echo cancellation and noise reduction in speech signals that have been corrupted by noise. The user can set the threshold to limit the amount of the signal that passes through at each frequency band. The proper threshold can be set based on the signal-to-noise ratio at that band. Derive a set of thresholds to reduce noise in the noisy signal `timit_noise.wav`.

B.4 DIGITAL-SIGNAL GENERATORS USING INFINITE-IMPULSE RESPONSE FILTERS

As discussed in Section 7.6, we can generate a sinusoidal signal by driving a second-order IIR filter into oscillation. This simple oscillator can be extended to generate any periodic waveform using Fourier series analysis. A detailed explanation of signal generation and its implementation is given in [7]. The most important objectives of this experiment are as follows:

1. Understand the concept of signal generation using IIR filters, and introduce some important design considerations in generating digital signals.
2. Ensure that the poles of the filter are inside the unit circle when implementing the IIR filter on fixed-point processors. Signal scaling is required to ensure proper operation of the fixed-point implementation.
3. Implement the digital-signal generator on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.4.1 Introduction

A square wave with an amplitude of ±1 and a 50% duty cycle can be expressed as

$$x(n) = \frac{4}{\pi}\sum_{k=0}^{k1}\frac{\sin[2\pi n f_0(2k+1)/f_s]}{(2k+1)}, \tag{B.4.1}$$

where f_0 is the frequency of the square wave, and the upper limit $k1 < \text{int}[(f_s - 2f_0)/4f_0]$. Similarly, the triangular wave can be expressed as

$$x(n) = \frac{4}{\pi}\sum_{k=0}^{k1}\frac{\cos[2\pi n f_0(2k+1)/f_s]}{(2k+1)^2}. \tag{B.4.2}$$

The sawtooth wave can be expressed as

$$x(n) = 2\sum_{k=1}^{k2}\frac{(-1)^{k-1}\sin(2\pi n f_0 k/f_s)}{k}, \tag{B.4.3}$$

where $k2 < \text{int}(f_s/2f_0)$. Finally, the full-wave sine rectifier with an amplitude between 0 and 1 can be expressed as

$$x(n) = \frac{2}{\pi} - \frac{4}{\pi}\sum_{k=0}^{k3}\frac{\cos[2\pi n f_0(2k)/f_s]}{(2k+1)(2k+3)}, \tag{B.4.4}$$

where $k3 < \text{int}(f_s/4f_0)$.

Similar to the DTMF signal generator described in Section 7.6.1, the transfer functions of IIR filters for implementing both sine and cosine waves are expressed as

$$H_{\sin}(z) = \frac{\sin(2\pi f_0/f_s)z^{-1}}{1 - 2\cos(2\pi f_0/f_s)z^{-1} + z^{-2}} \tag{B.4.5}$$

and

$$H_{\cos}(z) = \frac{1 - \cos(2\pi f_0/f_s)z^{-1}}{1 - 2\cos(2\pi f_0/f_s)z^{-1} + z^{-2}}. \tag{B.4.6}$$

These IIR filters are connected in parallel for implementing these periodic waveforms.

B.4.2 Lab Experiments Using MATLAB/Simulink

Complete the following exercises, and benchmark the results:

1. Use a frequency $f_0 = 100$ Hz and a sampling frequency $f_s = 10{,}000$ Hz to generate the square, triangular, sawtooth, and sine-rectifier waveforms in MATLAB. Verify the generated signals in terms of amplitude and period by

plotting the signals in MATLAB. Determine the number of sine generators required for each case to give a good approximation of the desired waveform. An example of the triangular-wave generator is given in `gen_triangular.m`.

2. Use a five-tap smoothing filter to smooth the square and sawtooth waves. Observe the improvement over the original waveform.
3. Examine the pole-zero plot for each sine generator using FDATool. Determine whether the poles lie exactly on or inside the unit circle for both the double-precision, floating-point and Q.15 fixed-point formats.
4. Set the initial state of the IIR filter such that it starts the oscillation without injecting an impulse signal as input. Modify the MATLAB program to implement the signal generator without injecting the impulse signal.
5. Generate the signals in Simulink using both the double-precision, floating-point and fixed-point Q.15 formats. Examine the effects of a fixed-point implementation in comparison with a double-precision simulation.

B.4.3 Lab Experiment Using the C5000 Code Composer Studio

Use the C5000 DSPLIB to write a C program that calls the library functions to implement the signal generators. Create, build, and run the project in CCS. Complete the following exercises, and benchmark the results:

1. Save the generated signal in memory, and display it using the graphic window. Compare the differences between the MATLAB and CCS results.
2. Profile the code and benchmark the cycle count, MIPS used, and data- and program-memory requirements.
3. Use different optimization levels in CCS, and compare the differences in the benchmark.
4. Perform the benchmark for both the sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.4.4 Additional Exercises

1. Extend the preceding exercises to generate a chirp signal with frequency swept linearly from 100 Hz to 1,000 Hz in 1 second. The sampling frequency is 10,000 Hz.
2. Note that another method of generating periodic signals is to use an IIR filter, in which the feedforward coefficients b_i contain all samples within one period of the signal. For example, to generate a triangular wave with a period of 10 samples, the feedforward coefficients are 0.2, 0.4, 0.6, 0.8, 1, 0.8, 0.6, 0.4, 0.2, and 0. The feedback coefficients are 0, except the last coefficient $a_9 = 1$. The input to the IIR filter is an impulse signal, which generates one period of triangular waveform from the feedforward coefficients. The impulse signal rotates back

from the feedback section to the feedforward section at the end of every period in generating the next period of signal samples. Implement this type of digital signal generator in MATLAB and CCS.

B.5 DIGITAL REVERBERATION USING INFINITE-IMPULSE RESPONSE FILTERS

Room reverberation can be simulated using a set of IIR filters. The reverberation algorithm simulates the sound reflections in an enclosed environment such as rooms, concert halls, etc. A detailed explanation of the reverberation algorithm and its implementation is given in [8, 9]. The most important objectives of this experiment are as follows:

1. Understand the concept of generating artificial reverberation. The most important design consideration involves tuning the parameters of IIR filters to simulate the reverberation in different environments.
2. Ensure that the poles of the filter are inside the unit circle when implementing the IIR filter. Signal scaling is required to ensure proper operation for the fixed-point implementation.
3. Implement the digital reverberation algorithm on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.5.1 Introduction

Reverberation consists of three components: direct sound, early reflections, and late reflections (or reverberation). Direct sound takes a direct (shortest) path from the sound source to the receiver (or listener). Early reflections, which arrive within 10 to 100 msec after the direct sound, are caused by sound waves reflected once before reaching the listener. A digital reverberation algorithm can be developed based on the characteristics of the room. As shown in Fig. B.3, this algorithm simulates room reverberation by using four comb filters, $C_1(z)$, $C_2(z)$, $C_3(z)$, and $C_4(z)$, connected in parallel, followed by two cascaded allpass filters, $A_5(z)$ and $A_6(z)$.

The transfer function of the comb filter is expressed as

$$C_i(z) = \frac{1}{1 - a_i z^{-Di}}, \quad i = 1, 2, 3, 4, \tag{B.5.1}$$

which is the IIR filter with Di poles equally spaced $(2\pi/Di)$ on the unit circle. The coefficient a_i determines the decay rate of the impulse response for each comb filter. The comb filter increases the echo density and gives the impression of the acoustics environment and room size. However, it also causes distinct coloration to the incoming signal. Allpass filters prevent such coloration and emulate more natural sound characteristics in a real room. The transfer function of the allpass filter is

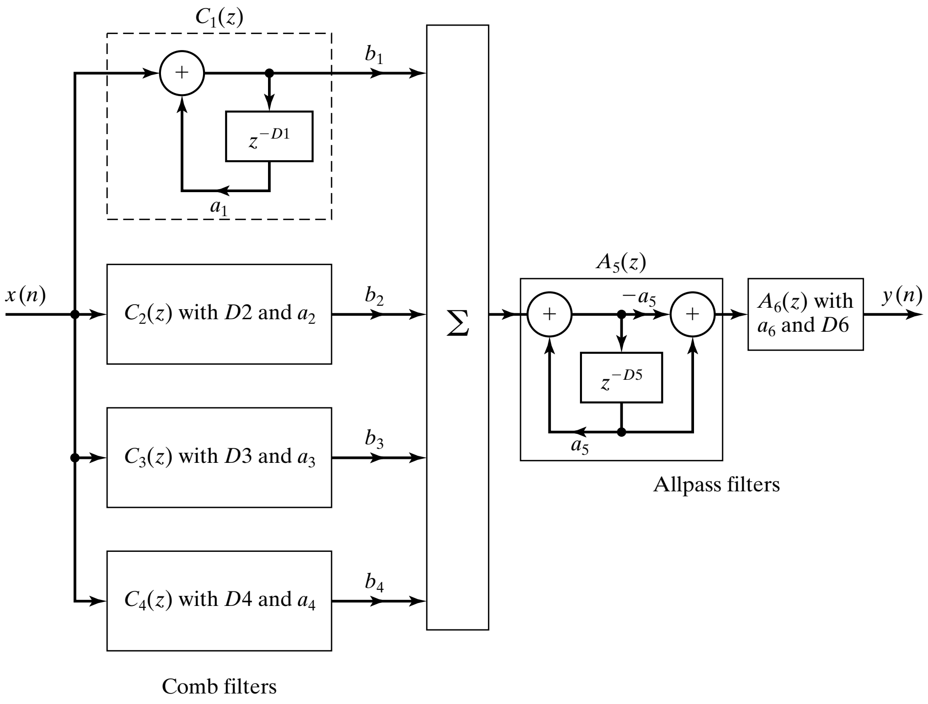

Figure B.3 Block diagram of a digital reverberation algorithm

expressed as

$$A_i(z) = \frac{-a_i + z^{-Di}}{1 - a_i z^{-Di}}, \quad i = 5, 6. \tag{B.5.2}$$

By cascading the allpass filters with comb filters, the impulse response of the overall system becomes more diffuse.

B.5.2 Lab Experiment Using MATLAB /Simulink

Complete the following exercises, answer the related questions, and benchmark the results:

1. Open the Simulink file `ex7_ii.mdl`, and use the DSP Blockset to realize the reverberation algorithm. Enter the delays, Di, for the comb filters as 29, 37, 44, and 50, and enter the delays for the allpass filters as 27 and 31. Set all feedback coefficients, a_i, $i = 1, \ldots, 4$, of the comb filters as 0.75. What are these delays in msec if the sampling frequency is 8,000 Hz? Run the simulation using the wave file `drums.wav` as input, and play back the processed signal `reverb_out.wav`. Do you perceive any reverberation effect?
2. Examine the impulse and magnitude responses of the comb filters and allpass filters. Compare the differences of using a longer delay, Di, and a larger coefficient.

3. Use an impulse signal as input to the reverberation algorithm shown in Fig. B.3, and save the impulse response. Examine the impulse and magnitude responses. The reverberation time t_{60} is defined as the time needed for the impulse response to decay by 60 dB. Compute the reverberation time, t_{60}, based on the given parameters in the Simulink file.
4. Modify the Simulink file to implement the fixed-point reverberation generator in Q.15 format using Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.5.3 Lab Experiment Using the C5000 Code Composer Studio

Use the C5000 DSPLIB to write a C program that calls the library functions to implement the reverberation generator. Create, build, and run the project in CCS. Complete the following exercises, and benchmark the results:

1. Create an impulse signal and access it via FILE I/O. Save the output impulse response in a data file and plot it. Compare the differences between the MATLAB and CCS results. In addition, use a speech file as input for the experiment.
2. Profile the code of the reverberation generator, and record the results of the cycle count, MIPS used, and data- and program-memory requirements.
3. Use different optimization levels in CCS to compare the differences in the benchmarks.
4. Perform the benchmark for both the sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.5.4 Additional Exercises

1. Note that a variation of the preceding Schroeder's method is to include a lowpass filter in place of the feedback coefficient a_i of the comb filter. The function of the lowpass filter is to spread out the echoes, resulting in a more diffuse reverberation. The lowpass filter also reduces the metallic sound effects of the comb filter and creates a more realistic room reverberation. Design and implement a suitable low-order IIR filter to replace the comb-filter coefficient. Examine the impulse and magnitude responses of this modified reverberation algorithm, and compare it with the original one.
2. Extend the reverberation algorithm using longer delays of 40, 44, 48, and 52 msec for the comb filters and delays of 7 and 8 msec for the allpass filters. Determine the delay-buffer lengths for implementing these delays when sampling at 8,000 Hz. Can these delay buffers be stored in the internal memory of C5000 processors?
3. Implement the modified reverberation algorithm on C5000 processors.

B.6 PARAMETRIC EQUALIZER USING INFINITE-IMPULSE RESPONSE FILTERS

A group of second-order IIR filters can be designed to implement the parametric equalizer. The parametric equalizer allows the user to specify its center frequency, bandwidth, filter type, and gain, while the graphic equalizer only allows the user to control its gain. A more detailed explanation is given in [10, 11]. The most important objectives of this experiment are as follows:

1. Understand the principle of designing a parametric equalizer. An important design consideration involves mapping an analog IIR filter to its digital equivalent using bilinear transformation.
2. Ensure that the poles of the IIR filter are inside the unit circle. Prevent overflow of the parametric equalizer by scaling the signal when the IIR filter is implemented on fixed-point processors.
3. Implement the parametric equalizer on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.6.1 Introduction

A direct-form I, second-order IIR filter (biquad) with five coefficients is used in this application to adjust the bandwidth, center frequency, and gain of the parametric equalizer. An analog prototype filter is used in specifying the frequency response, and it is converted to a digital filter using the bilinear transform. The coefficients of the analog filter are first determined by satisfying the set of frequency-response equations given in [11]. As explained in Section 7.2.1, the desired digital frequencies (center and bandedge frequencies) need to be prewarped using Eq. (7.2.16).

The digital IIR filter and its coefficients [11] are expressed as

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}, \tag{B.6.1}$$

where

$$b_0 = \frac{1 + \gamma\sqrt{K}}{1 + \gamma/\sqrt{K}}, \tag{B.6.2a}$$

$$b_1 = \frac{-2\cos\Omega_c}{1 + \gamma/\sqrt{K}}, \tag{B.6.2b}$$

$$b_2 = \frac{1 - \gamma\sqrt{K}}{1 + \gamma/\sqrt{K}}, \tag{B.6.2c}$$

$$a_2 = \frac{1 - \gamma/\sqrt{K}}{1 + \gamma/\sqrt{K}}, \tag{B.6.2d}$$

$a_0 = 1$, and $a_1 = b_1$. Note that the constant

$$K = 10^{G/20} \tag{B.6.3a}$$

is the gain of the filter at a digital-center frequency of ω_c (or an analog frequency of Ω_c), and G is the gain expressed in dB. The variable γ is given as

$$\gamma = \sqrt{K}\tan\left(\frac{BW}{2}\right), \tag{B.6.3b}$$

where BW is the bandwidth (3 dB) of the filter.

The Q (quality) factor defines the bandwidth (or sharpness) of the filter. The relationship between the Q factor, the center frequency, f_c, and the bandwidth, BW, of the filter is expressed as

$$Q = \frac{f_c}{BW} = \frac{\sqrt{f_1 f_2}}{f_2 - f_1}, \tag{B.6.4}$$

where f_1 and f_2 are the bandedge frequencies. The filter type can be low- and high-frequency shelving filters, peak filters, and lowpass and highpass filters.

In this experiment, we investigate the design and implementation of an octave-band parametric equalizer that covers the frequency range from 22 Hz to 22,500 Hz at a sampling frequency of 44,100 Hz. The frequency specifications for the 10-band octave equalizer are given in Table B.1. In octave spacing, $f_2 = 2f_1$; thus, the Q factor is $\sqrt{2}$, as defined in Eq. (B.6.4). The filter type selected for this application is the peak filter, which allows the user to boost or attenuate the signal. The center frequencies are stated in Table B.1. The parametric equalizer provides more flexibility in controlling its gain and frequency parameters (e.g., center frequency, bandwidth, and rolloff) as compared with the graphic equalizer. In order to implement the 10-band parametric equalizer with octave spacing, we can simply cascade 10 second-order IIR peak filters with the coefficients calculated based on Eq. (B.6.2). The coefficients need to be recomputed on the fly with a change of gain at each band. A better alternative is to use a look-up table that stores the coefficients for different gains, center frequencies, and bandwidth settings.

B.6.2 Lab Experiment Using MATLAB/Simulink

Complete the following exercises, and benchmark the results:

1. Open the Simulink file `ex7_iii.mdl`, and use FDATool to design 10 octave filters using the cascade-biquad structure. Enter the coefficients (derived from the MATLAB file `equalizer.m` from [11] and `octave.m`) for each filter block. Examine the impulse response, pole-zero plot, stability, magnitude and phase responses, and group delay of all of the bandpass filters.
2. Apply the audio-signal file `drums.wav` to the equalizer, and examine the output signal `drums_peq.wav` in terms of its magnitude response and its perceived differences from the original signal.

3. Modify the Simulink file to implement the fixed-point (Q.15 format) version of the 10-band octave parametric equalizer using the Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.6.3 Lab Experiment Using the C5000 Code Composer Studio

Use the C5000 DSPLIB to write a C program that calls the library functions to implement the 10-band octave equalizer using the cascade-biquad structure. Create, build, and run the project in CCS. Complete the following exercises, and benchmark the results:

1. Access the input signal `drums.dat` via FILE I/O. Save the output in a data file, and listen to the result. Compare the differences between the MATLAB and CCS results.
2. Profile the code of the parametric equalizer, and benchmark the cycle count, MIPS used, and data- and program-memory requirements. Also, list the additional time required to alter the gain of the filters. Compare this data with the

TABLE B.2 Twenty-Five Critical Bands in the Human Hearing Range

Critical band	Lower cutoff (Hz)	Center frequency (Hz)	Upper cutoff (Hz)	Q-factor
1	0	50	100	0.50
2	100	150	200	1.50
3	200	250	300	2.50
4	300	350	400	3.50
5	400	450	510	4.50
6	510	570	630	4.75
7	630	700	770	5.00
8	770	840	920	5.60
9	920	1,000	1,080	6.25
10	1,080	1,170	1,270	6.15
11	1,270	1,370	1,480	6.52
12	1,480	1,600	1,720	6.66
13	1,720	1,850	2,000	6.60
14	2,000	2,150	2,320	6.72
15	2,320	2,500	2,700	6.58
16	2,700	2,900	3,150	6.44
17	3,150	3,400	3,700	6.18
18	3,700	4,000	4,400	5.71
19	4,400	4,800	5,300	5.33
20	5,300	5,800	6,400	5.27
21	6,400	7,000	7,700	5.38
22	7,700	8,500	9,500	4.72
23	9,500	10,500	12,000	4.20
24	12,000	13,500	15,500	3.86
25	15,500	19,500	22,050	2.98

benchmark obtained for the FIR-filter-based graphic equalizer in the previous experiment.

3. Use different optimization levels in CCS to compare the differences in the benchmarks.
4. Perform the benchmark for both the sample processing and block processing, and determine a good frame size for the block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.6.4 Additional Exercises

1. Extend the octave-band parametric equalizer to implement a system that mimics the critical band of the human auditory system. The critical band (listed in Table B.2) is commonly used in perceptual audio coding that relates how the ear discriminates between energy inside and outside the band.
2. Use the preceding critical-band parametric equalizer to reduce the bandlimited noise in the audio-signal file `noisy_drums.wav`. What are the advantages of using the parametric equalizer over the graphic equalizer when controlling the noise level?

B.7 FAST-CONVOLUTION METHODS

The overlap-save and overlap-add techniques were introduced in Section 8.7.1. These signal-segmentation and signal-combination techniques allow a long signal sequence to be convolved continuously with a vector such as a filter coefficient vector. The most important objectives of this experiment are as follows:

1. Understand the differences in the signal-segmentation and signal-combination techniques used in the overlap-save and overlap-add methods.
2. Examine the effects of different segment lengths used in fast convolution.
3. Implement the fast-convolution techniques (overlap-add and overlap-save) on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

The reader can refer to Section 8.7.1 for a detailed introduction to the overlap-add and overlap-save techniques.

B.7.1 Lab Experiment Using MATLAB/Simulink

Complete the following exercises, and benchmark the results:

1. Open the MATLAB file `overlap_save.m`, and examine the steps used to complete the overlap-save method. Replace the fast-convolution code with the segmented time-domain convolution program. Compare the time needed to execute the fast convolution and the time-domain convolution using the `etime.m` function in MATLAB.

2. Repeat the preceding exercise for the MATLAB file `overlap_add.m`.
3. Compare the differences in terms of memory usage and computational load of the overlap-add and over-save techniques.
4. Examine the Simulink (DSP Blockset) blocks **Overlap-Add FFT Filter** and **Overlap-Save FFT Filter**. Compare the differences in these processing blocks, and examine the results obtained from these blocks.
5. Modify the preceding Simulink blocks to implement the fixed-point (Q.15 format) version of fast convolution using the Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.7.2 Lab Experiment Using the C5000 Code Composer Studio

Write a C program that calls the library functions in the C5000 DSPLIB to implement the fast-convolution algorithm. Create, build, and run the project in CCS. Complete the following exercises, and benchmark the results:

1. Save the results in data files, and observe the magnitude responses. Compare the differences between the MATLAB and CCS results.
2. Profile the code of the fast-convolution algorithm, and record the cycle count, MIPS used, and data- and program-memory requirements. Compare the benchmark results obtained from the time-domain convolution and the fast convolution.
3. Use different optimization levels in CCS to compare the differences in the benchmarks.
4. Perform the benchmark for both the sample processing and block processing, and determine a suitable frame size for the block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.7.3 Additional Exercises

1. Examine the C5000 DSPLIB function `fir` and check whether the overlap-add or overlap-save method is being used in block-processing mode.
2. Note that crosscorrelation between two vectors can be computed by simply time-reversing elements in one of the vectors before computing the crosscorrelation using convolution operations. Verify the results using the MATLAB crosscorrelation function `xcorr.m`.

B.8 IMPLEMENTATION OF A SLIDING FAST FOURIER TRANSFORM

As discussed in Chapter 8, the DFT or FFT is a block-processing algorithm that processes a block of input samples $\{x(n), n = 0, 1, \ldots, N - 1\}$ into its frequency-domain outputs $\{X(k), k = 0, 1, \ldots, N - 1\}$. Due to the superposition and time-shifting properties of the FFT, a series of FFT computations can be processed in a sample-processing mode. This computational technique is called the sliding FFT. A more detailed explanation of this algorithm is given in [13]. The most important objectives of this experiment are as follows:

1. Understand the principle of sliding FFT and its differences from conventional block FFT processing.
2. Ensure that the processing is stable in a fixed-point implementation. Use proper signal scaling to ensure the correct operation in the fixed-point implementation of the sliding FFT algorithm.
3. Implement the sliding FFT on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.8.1 Introduction

The linear property of the FFT implies that it is possible to compute an N-point FFT by summation of the results of N separate transforms, where each transform has only one sample in its original signal vector. This concept can be expressed as

$$\begin{aligned} FFT\begin{bmatrix} x(n) \\ x(n-1) \\ \vdots \\ x(n-N+1) \end{bmatrix} &= FFT\begin{bmatrix} x(n) \\ 0 \\ \vdots \\ 0 \end{bmatrix} + FFT\begin{bmatrix} 0 \\ x(n-1) \\ \vdots \\ 0 \end{bmatrix} \\ &+ \cdots FFT\begin{bmatrix} 0 \\ 0 \\ \vdots \\ x(n-N+1) \end{bmatrix} \\ &= x(n) + x(n-1)e^{-j2\pi n/N} \\ &+ \cdots + x(n-N+1)e^{-j2\pi n(N-1)/N}. \end{aligned} \quad \text{(B.8.1)}$$

Another property used in the sliding FFT algorithm is the circular-shifting property introduced in Section 8.3.2, which states that the transform of a time-delayed sequence is a phase-rotated version of the transform of the original sequence, expressed as

$$FFT[x(n-m)] = X(k)e^{-j2\pi km/N}, \quad \text{(B.8.2)}$$

where $x(n-m)$ is the input vector that is delayed by m samples.

The sliding FFT algorithm computes the first N-point FFT at time n to obtain $X_n(k)$ and uses a sliding window (shift one sample) to compute the FFT for the next signal vector (within the sliding window) at time $n+1$ to obtain $X_{n+1}(k)$. To prevent the repeat computation of the FFT for every sample shift, we use the properties of shifting given in Eq. (B.8.2) and the linearity given in Eq. (B.8.1) to update the new FFT recursively as

$$X_{n+1}(k) = X_n(k)e^{-j2\pi k/N} + x(n+1) - x(n-N), \qquad \text{(B.8.3)}$$

where $x(n+1)$ is the new sample in the new window at time $n+1$, and $x(n-N)$ is the oldest sample of the previous window at time n.

Therefore, the new FFT is computed from the previous FFT that is phase-shifted by $e^{-j2\pi k/N}$. The updating samples $x(n+1)$ and $x(n-N)$ are added to and subtracted from, respectively, at all frequency bins. This results in N complex multiplications and $2N$ complex additions to obtain $X_{n+1}(k)$, as opposed to $(N\log_2 N)/2$ complex multiplications and $(N\log_2 N)$ complex additions in a conventional FFT algorithm. In addition, the sliding FFT can compute only the frequency bins of interest, since each frequency bin is calculated separately as in the DFT computation.

B.8.2 Lab Experiment Using MATLAB/Simulink

Complete the following exercises, and benchmark the results:

1. Write a MATLAB program to verify that the sliding FFT algorithm produces the same results as the block FFT function `fft.m`. Use an 8-point FFT for the signal sequence [1 2 3 4 5 6 7 8 7 6 5 4 3 2 1] as an example. Compare the time taken (use the MATLAB function `etime.m`) to complete the FFT of the whole sequence using these two different methods. Refer to the MATLAB file `slidingfft.m`.
2. Implement the sliding FFT algorithm using Simulink.
3. Modify the Simulink file to implement the fixed-point (Q.15 format) version of the sliding FFT algorithm using the Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

It is important to note that multiplication by the twiddle factor in finite precision may result in instability. The phase-rotating coefficients $e^{-j2\pi k/N}$ must be less than unity (e.g., 0.9999). However, scaling these twiddle factors by 0.9999 causes the value at each frequency bin to be multiplied by 0.9999^N. Therefore, the N-th oldest sample $x(n-N)$ needs to be scaled by 0.9999^N before subtraction occurs.

B.8.3 Lab Experiment Using C5000 CCS

Complete the following exercises, and benchmark the results:

1. Write a C program that calls the library functions in the C5000 DSPLIB to implement the sliding FFT algorithm. Create, build, and run the project in CCS.
2. Save the results in data files and observe the magnitude-spectrum plots. Compare the differences between the MATLAB and CCS results.
3. Profile the program for the sliding FFT algorithm, and record the cycle count, MIPS used, data- and program-memory requirements. Compare the benchmarks obtained using the conventional FFT and the sliding FFT.

4. Use different optimization levels in CCS to compare the differences in benchmarks.
5. Perform the benchmark for both sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.8.4 Additional Exercises

1. Compute the spectrogram of the speech signal in the file `timit1.asc` using the sliding FFT technique. Use the MATLAB functions `imagesc.m` and `colorbar.m` to plot the spectrogram of the speech signal computed by the sliding FFT method.
2. Plot the spectrogram using the `specgram.m` function, and compare the result with the sliding FFT implementation.

B.9 IMPLEMENTATION OF THE ZOOM FAST FOURIER TRANSFORM

A zoom FFT can be used to reduce the computational load of a large-order FFT in achieving a very fine frequency resolution at a specific frequency range without computing the entire spectrum. The zoom-FFT technique is particularly useful in applications such as radar, sonar, radio frequency (RF) communications, and vibration analysis, where the desired signal components may occupy only a very narrow frequency band. This technique can extract the narrowband spectrum on an expanded (zoom-in) frequency scale efficiently. The zoom-FFT technique is explained in [14]. The most important objectives of this experiment are as follows:

1. Understand the principles of the zoom-FFT technique and its differences from conventional FFT for spectral analysis.
2. Ensure proper fixed-point implementation of the zoom-FFT algorithm by using signal scaling.
3. Implement the zoom-FFT on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.9.1 Introduction

When a signal is sampled at a high sampling frequency, f_s, a large number of samples are needed to span a given time to achieve a fine resolution. This large-size FFT is especially wasteful and is not even practical when only a narrow frequency band is of interest. The zoom-FFT uses downsampling to cover a longer time with fewer samples, thus reducing the computational load of the FFT used in computing a very fine resolution. Downsampling zooms in the spectrum and allows the user to magnify the spectrum of the narrowband signal. For example, two closely spaced signals at 2,000 Hz and 2,005 Hz (with a sampling rate of 48 kHz) can be viewed clearly using a zoom-FFT with a reasonable order. However, the size of the FFT in the conventional spectral analysis must be greater than 9,600 (48 kHz/5) in order to achieve the resolution of 5 Hz.

In order to implement the zoom-FFT, the signal is first sampled at a high sampling frequency, f_s, and filtered by a bandpass filter to focus on the frequency band of interest. In the preceding example, a bandpass filter with a passband of 1,800 Hz to 2,200 Hz can be used. The filtered signal is then downsampled to f_s/M, where M is a downsampling integer. The frequency f_s/M must be greater than twice the bandwidth of the signal of interest in order to avoid aliasing. The process of downsampling was explained in Section 6.5.2. These decimated samples are transformed using a lower-order N-point FFT to compute the spectrum. The resulting frequency resolution is equal to $f_s/(MN)$, which is equivalent to using a higher-order $L\ (=MN)$-point FFT without downsampling.

Using the same example, if we select a downsample factor of $M = 10$, the order of the FFT can be reduced to $N = 1{,}024\ (>960)$. This size is manageable in terms of computation and memory requirements when computing the FFT. However, it must be noted that both the zoom-FFT and conventional FFT techniques must acquire the same amount of samples (0.2 second) in order to achieve a frequency resolution of 5 Hz. Note that the use of zero-padding alone cannot improve the frequency resolution, as explained in Chapter 8.

B.9.2 Lab Experiment Using MATLAB/Simulink

Complete the following exercises, answer the related questions, and benchmark the results:

1. Use the zoom-FFT algorithm to extract the spectral information of the signal in the data file `input.dat`, which is sampled at 48 kHz. It is known that the signal of interest lies in the frequency range from 1,000 to 2,200 Hz. Use a suitable downsampling factor to zoom in on the spectrum, and select a proper order of FFT for achieving a frequency resolution of less than 5 Hz. Plot the spectrum obtained by the zoom-FFT algorithm. Compare the result with the conventional FFT-based spectrum. What is the minimum order of the conventional FFT required to achieve a frequency resolution of 5 Hz? Refer to `zoomfft.m`.
2. Determine what happens if the input sequence has only 5,000 samples. Can we still analyze the spectrum accurately?
3. Implement the zoom-FFT algorithm using Simulink to process the same signal in the file `input.dat`.
4. Modify the Simulink file to implement the fixed-point (Q.15 format) version of the zoom-FFT algorithm using the Fixed-Point Blockset. Examine the effects of a fixed-point implementation in comparison with a double-precision, floating-point simulation.

B.9.3 Lab Experiment Using the C5000 Code Composer Studio

Complete the following exercises, and benchmark the results:

1. Write a C program that calls the library functions in the C5000 DSPLIB to implement the zoom-FFT algorithm. Create, build, and run the project in CCS.

2. Save the results in data files and observe the magnitude-spectrum plots. Compare the differences between the MATLAB and CCS results.
3. Profile the program of a zoom-FFT algorithm, and record the cycle count, MIPS used, and data- and program-memory requirements. Compare the benchmarks obtained by the conventional FFT and the zoom-FFT techniques.
4. Use different optimization levels in CCS to compare the differences in the benchmark.
5. Perform the benchmark for both sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.9.4 Additional Exercises

1. Note that another method of extracting useful information is to demodulate the signal instead of the bandpass filter. This demodulation can be done by multiplying the signal with a complex exponential to translate the frequency band of interest to the baseband, and extract the baseband information using a lowpass filter. The filtered baseband signal is then decimated by a factor of M and transformed using a suitable order N of the FFT. For example, an ultrasonic signal centered at 40 kHz is used to amplitude-modulate two audio signals at 2,000 Hz and 2,010 Hz, and the amplitude-modulated signal is saved in the file `commsignal.dat`. In this case, the audio signal is confined within a bandwidth of 3,000 Hz. Perform proper demodulation to extract the audible signal, and then perform the FFT algorithm.
2. Note that, instead of demodulating the signal to its baseband and using a sampling frequency $f_s > 2 \times 40$ kHz, we can use the technique of undersampling to recover the audible signal with a bandwidth of 3,000 Hz. In this method, a proper undersampling frequency must be selected to recover the baseband version of the original signal. Apply the downsampling technique on `commsignal.dat`, and use a suitable length FFT to observe its zoom-in spectrum. There is no need to perform further downsampling, as the process of undersampling has taken into account the downsampling.

B.10 ACOUSTIC ECHO CANCELLATION

The hands-free phone (or speakerphone) provides the convenience of a phone conversation and thus is used widely in cars and conference rooms. As illustrated in Fig. B.4, the far-end speech is broadcast through one or more loudspeakers in the room. The acoustic echo consists of direct coupling and reflections that are picked up by the microphone and transmitted back to the far end. The most effective technique for eliminating the acoustic echo is adaptive echo cancellation. A detailed introduction of adaptive echo cancellation can be founded in [16]. The objectives of this experiment are as follows:

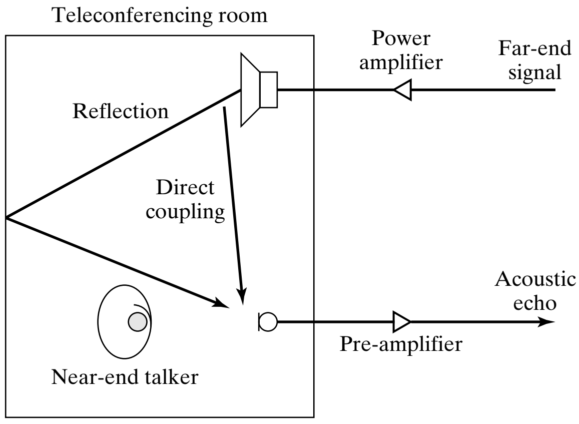

Figure B.4 Acoustic echo in a teleconferencing room

1. Understand the principle of adaptive echo cancellation and its application in reducing acoustic echoes.
2. Implement the acoustic echo canceller on C5000 processors using a mixed C-and-assembly program. Benchmark the cycle count, MIPS used, and data- and program-memory requirements.

B.10.1 Introduction

A block diagram of an acoustic echo canceller is illustrated in Fig. B.5. The acoustic echo canceller consists of a loudspeaker, a microphone, and an adaptive filter updated by the LMS algorithm. The generation of acoustic echoes between the loudspeaker and the microphone can be modeled as an acoustic echo path, $P(z)$, which includes the DAC, the smoothing lowpass filter, the power amplifier, the loudspeaker, the microphone, the preamplifier, the anti-aliasing lowpass filter, the ADC, and the acoustic-transfer function of the room from the loudspeaker to the microphone. The adaptive filter, $W(z)$, uses the far-end speech, $x(n)$, to model the acoustic echo path,

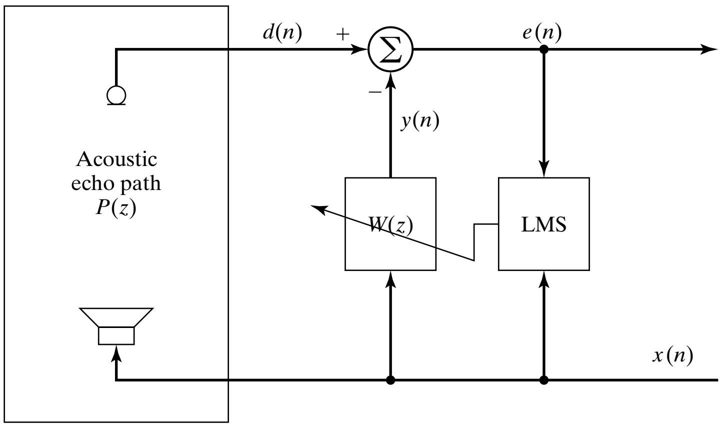

Figure B.5 Block diagram of an acoustic echo canceller

$P(z)$, and generate the echo replica, $y(n)$, which is used to cancel undesired acoustic echoes in the microphone signal, $d(n)$.

The acoustic echo canceller removes acoustic echoes by generating the replica of an echo expressed as

$$y(n) = \sum_{l=0}^{L-1} w_l(n)x(n-l), \tag{B.10.1}$$

where $w_l(n)$ are the coefficients of the $W(z)$ at time n. This echo replica is subtracted from the microphone signal, $d(n)$, expressed as

$$e(n) = d(n) - y(n). \tag{B.10.2}$$

The coefficients of $W(z)$ are updated by the most commonly used LMS algorithm expressed as

$$w_l(n+1) = w_l(n) + \mu e(n)x(n-l), \quad l = 0, 1, \ldots, L-1, \tag{B.10.3}$$

where μ is the step size. This adaptation must be stopped if the near-end talker is speaking.

B.10.2 Lab Experiment Using C Programs

Open the C program `aec.c` and study the functionality of the program. This program reads $x(n)$ and $d(n)$ from the data files. Use the data file `gmos.dat` as the far-end speech and the file `bmjs.dat` as the near-end speech. Complete the following exercises, and benchmark the results:

1. Generate the microphone signal, $d(n)$, using the C program `echogen.c`, and save the signal in the file `microphoneIn.dat`. This microphone signal consists of the near-end speech and the acoustic echo.
2. Plot the waveforms of `gmos.dat`, `bmjs.dat`, and `microphoneIn.dat`. Identify four different cases: (1) receive mode with only the far-end speech, (2) transmit mode with only the near-end speech, (3) double-talk mode with both the far-end and near-end speeches, and (4) idle mode without speech.
3. Compile the C program using a C compiler (e.g., Visual C/C++ 6.0 on a personal computer), execute the program, and save the output, $e(n)$, in the file `lineOut.dat`.
4. Play back the output file `lineOut.dat`. Compare it with other data files, and evaluate the performance of the acoustic echo cancellation.
5. Adjust the parameters (e.g., filter length and step size) in the C program `aec.c`, and observe the differences in performance.
6. Convert the floating-point C program to a fixed-point C program that can be executed on a personal computer using the techniques introduced in Chapters 6 and 9. Examine the effects of using a fixed-point implementation in comparison with a floating-point simulation.
7. Plot and listen to the floating-point and fixed-point results.

B.10.3 Lab Experiment Using the C5000 Code Composer Studio

Complete the following exercises, and benchmark the results:

1. Write a C program that calls the library functions (e.g., FIR filtering and the LMS update) in the C5000 DSPLIB to implement the acoustic echo canceller. Create, build, and run the project in CCS.
2. Save the results in data files and evaluate the waveforms. Compare the differences between the computer and CCS results.
3. Profile the acoustic echo-cancellation program, and record the cycle count, MIPS used, and data- and program-memory requirements. Compare the benchmark obtained using an adaptive filter with different lengths.
4. Use different optimization levels in CCS to compare the differences in the benchmarks.
5. Perform the benchmark for both the sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.10.4 Additional Exercises

1. Use the data file `gmos.dat` as the near-end speech and the file `bmjs.dat` as the far-end speech. Compare the performance differences with the preceding exercises that use different files.
2. Change the characteristics of the acoustic echo path by modifying the parameters used in `echogen.c`. Evaluate the performance differences of the echo cancellation. Identify the factors that affect the performance of acoustic echo cancellation.
3. Evaluate the performance of the double-talk detector implemented in `aec.c`. Change its parameters, and observe the degradation of the echo canceller performance. Improve the performance of the double-talk detector.
4. Use the different adaptive algorithms introduced in Section 9.2.2 to modify `aec.c`, and compare the performance and computational complexity.
5. Replace the FIR filter used in `aec.c` by the filterbank introduced in Section B.3. In addition, use the decimation and interpolation techniques discussed in Section 6.5 such that the adaptive filtering and coefficient update are performed at a lower sampling rate to save computation requirements.

B.11 ACTIVE NOISE CONTROL

Active noise control (ANC) involves an electroacoustic or electromechanical system that cancels the primary (unwanted) noise based on the principle of superposition. Specifically, an antinoise of equal amplitude and opposite phase is generated and combined with the primary noise, thus resulting in the cancellation of both noises. The ANC system efficiently attenuates low-frequency noise where passive methods are either ineffective or tend to be very expensive or bulky. ANC is developing rapidly because it permits improvements in noise control, often with potential benefits in size, weight, volume, and cost. Detailed information on ANC is given in [16, 17].

B.11.1 Introduction

A single-channel acoustic ANC system in a duct is illustrated in Fig. B.6. A reference microphone close to the noise source senses the undesired noise produced by the noise source before it passes a loudspeaker downstream in the duct. The ANC system uses this reference signal, $x(n)$, to generate the output signal, $y(n)$, that drives the secondary (or canceling) loudspeaker in the duct. The secondary noise produced by the loudspeaker has the same amplitude, but it is 180° out of phase and thus is able to cancel the undesired noise. The error microphone measures the residual noise, $e(n)$, which is minimized by adapting the coefficients of the adaptive filter in the ANC system.

The block diagram of the single-channel broadband feedforward ANC system is illustrated in Fig. B.7. The signal $x(n)$ is the input signal sensed by the reference microphone, $e(n)$ is the residual-error signal measured by the error microphone, and

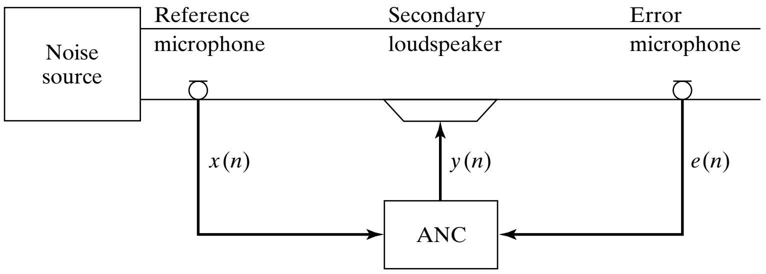

Figure B.6 Single-channel broadband-acoustic ANC system in a duct

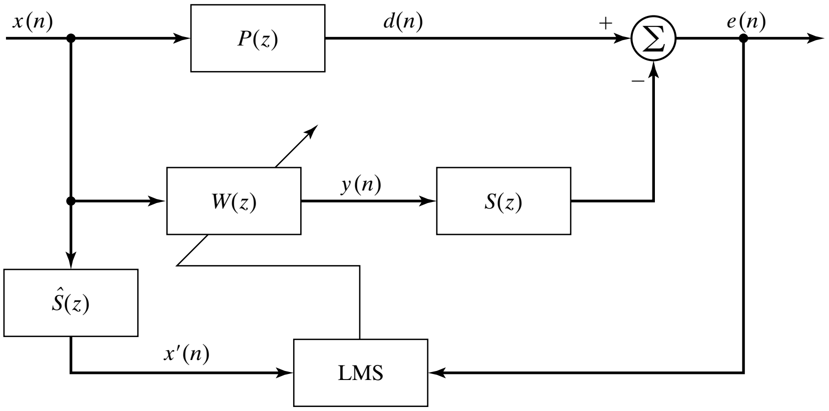

Figure B.7 Block diagram of an ANC system using the filtered-X LMS algorithm

$d(n)$ is the primary noise to be reduced. The transfer function, $P(z)$, represents the primary path from the reference microphone to the error microphone, and $S(z)$ is the secondary-path transfer function between the output of the adaptive filter, $W(z)$, and the output of the error microphone. The secondary path includes the DAC, reconstruction filter, power amplifier, loudspeaker, acoustic path from the loudspeaker to the error microphone, error microphone, preamplifier, anti-aliasing filter, and ADC.

As illustrated in Fig. B.7, the secondary signal, $y(n)$, is generated by filtering the signal $x(n)$ by the adaptive FIR filter, $W(z)$, expressed as

$$y(n) = \mathbf{w}^T(n)\mathbf{x}(n), \tag{B.11.1}$$

where T denotes the transpose of the vector, $\mathbf{w}(n) = [w_0(n)\ w_1(n) \ldots w_{L-1}(n)]^T$ and $\mathbf{x}(n) = [x(n)\ x(n-1) \ldots x(n-L+1)]^T$ are the coefficient and signal vectors of $W(z)$, respectively, and L is the filter order.

To ensure the convergence of the LMS algorithm, it is necessary to compensate $S(z)$ from $y(n)$ to $e(n)$. There are many methods for compensating the effect of $S(z)$. The most effective technique is to place an estimation of the secondary path in the reference-signal path to the LMS algorithm, which is called the filtered-X LMS algorithm. In this algorithm, the reference signal is filtered by the secondary-path estimation filter, $\hat{S}(z)$, expressed as

$$\mathbf{x}'(n) = \hat{s}(n)*\mathbf{x}(n), \tag{B.11.2}$$

where $\hat{s}(n)$ is the impulse response of $\hat{S}(z)$, and * denotes linear convolution. The adaptive filter minimizes the instantaneous squared error using the filtered-X LMS algorithm expressed as

$$\mathbf{w}(n+1) = \mathbf{w}(n) + \mu\mathbf{x}'(n)e(n), \tag{B.11.3}$$

where $\mu > 0$ is the step size (or convergence factor).

B.11.2 Lab Experiment Using MATLAB

Complete the following exercises, and benchmark the results:

1. Open the MATLAB program `fxlms.m`, and study the functionality of the program. This program reads $x(n)$ from the data file `xn.dat`, the coefficients of $P(z)$ (modeled as an IIR function) from the file `p_z.asc` (numerator) and `p_p.asc` (denominator), and the coefficients of $S(z)$ from the file `s_z.asc` (numerator) and `s_p.asc` (denominator). The models of $P(z)$ and $S(z)$ are obtained from [16].
2. Modify the MATLAB program to save the error signal, $e(n)$, in the data file `en.dat`. Plot both the primary noise `xn.dat` and the residual noise `en.dat`, and evaluate the performance of the ANC system. Compare the spectra of these two signals, and identify the noise reduction in dB.

3. Write a C program that implements the filtered-X LMS algorithm based on `fxlms.m`. Verify the C program by comparing its error output with `en.dat` generated by `fxlms.m`.
4. Convert the floating-point C program to a fixed-point C program that can be executed on a personal computer using the techniques introduced in Chapter 9. Examine the effects of using a fixed-point implementation in comparison with a floating-point simulation.

B.11.3 Lab Experiment Using the C5000 Code Composer Studio

Complete the following exercises and benchmark the results:

1. Write a C program that calls the library functions (e.g., FIR filtering and the LMS update) in the C5000 DSPLIB to implement the ANC. Create, build, and run the project in CCS.
2. Save the results in data files, and observe the waveforms. Compare the differences between the results obtained in the previous section and the CCS results.
3. Profile the ANC program, and record the cycle count, MIPS used, and data- and program-memory requirements. Compare the benchmark obtained using an adaptive filter with different lengths.
4. Use different optimization levels in CCS to compare the differences in the benchmarks.
5. Perform the benchmark for both sample processing and block processing, and determine a good frame size for block processing. State the advantages and disadvantages of using block processing in terms of the MIPS usage, system setup, system latency, and memory requirement for buffering.

B.11.4 Additional Exercises

1. Use MATLAB to generate multiple sinewaves corrupted by white noise, and save the result in a data file. Perform the preceding ANC exercise, and compare the results.
2. Change the parameters such as the order of $W(z)$ and the step size, and repeat the preceding exercises. Compare the performance differences with different parameters.
3. The ANC algorithm shown in Fig. B.7 is called the broadband ANC system, which uses the reference sensor. Assuming that the primary noise consists of multiple sinewaves and that the frequencies of these narrowband components are known, it is possible to synthesize the reference signal $x(n)$ that consists of multiple sinewaves with the same frequencies. The detailed narrowband ANC algorithm is given in [16]. Implement the narrowband ANC algorithm in MATLAB and C for experiments.
4. The ANC algorithm shown in Fig. B.7 is called the feedforward ANC system, which uses both the reference and error sensors. It is possible to estimate the reference signal, $x(n)$, using the antinoise, $y(n)$, and the residual error, $e(n)$.

The detailed algorithm is given in [16]. Implement this adaptive feedback ANC algorithm in both MATLAB and CCS for experiments, and evaluate its performance.

5. Use Simulink to simulate the ANC implemented in the MATLAB script `fxlms.m`.

SUGGESTED READINGS

1 Gardner, W. G. and K. D. Martin. "HRTF Measurements of a KEMAR." *The Journal of The Acoustical Society of America,* 97, no. 6 (1995): 3907–3908. HRTF measurements and other data files can be downloaded from MIT Media Lab, *http://sound.media.mit.edu/KEMAR.html*.

2 Begault, D. R. *3-D Sound for Virtual Reality and Multimedia*. Boston, MA: Academic Press, 1994.

3 Embree, P. M. *C Algorithms for Real-Time DSP*. Upper Saddle River, NJ: Prentice Hall, 1995.

4 Proakis, J. G. and D. Manalokis. *Digital Signal Processing: Principles, Algorithms, and Applications.* 3rd Edition. Upper Saddle River, NJ: Prentice Hall, 1996.

5 Mitra, S. K. *Digital Signal Processing–A Computer-Based Approach*. 2nd Ed. New York NY: McGraw-Hill, 2001.

6 Vaidyanathan, P. P. *Multirate Systems and Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1993.

7 Spiegel, M. R. and J. Liu. *Mathematical Handbook of Formulas and Tables*. 2nd Ed. Singapore: McGraw-Hill International Editions, Schaum's Outline Series, 1999.

8 Schroeder, M. R. "Digital Simulation of Sound Transmission in Reverberant Spaces." *The Journal of The Acoustical Society of America,* 47 (1970): 424.

9 Moorer, J. A. "About the Reverberation Business." *Computer Music Journal*, 3, no. 2 (1979): 13–28.

10 Orfanidis, S. J. *Introduction to Signal Processing*. Upper Saddle River, NJ: Prentice Hall, 1996.

11 Bristow-Johnson, R. "The Equivalence of Various Methods of Computing Biquad Coefficients for Audio Parametric Equalizers." Electronic document available at *www.Harmony-central.com/Effects/Articles/EQ_Coefficients/EQ-Coefficients.pdf*.

12 Higgins, R. J. *Digital Signal Processing in VLSI*. Upper Saddle River, NJ: Prentice Hall, 1990.

13 Brigham, E. O. *The Fast Fourier Transform and Its Applications*. Upper Saddle River, NJ: Prentice Hall, 1988.

14 Ifeachor, E. C. and B. W. Jervis. *Digital Signal Processing: A Practical Approach*. 2nd Ed. Prentice Hall, 2002.

15 Kuo, S. M. and B. H. Lee. *Real-Time Digital Signal Processing*. New York, NY: Wiley, 2001.

16 Kuo, S. M. and D. R. Morgan. *Active Noise Control Systems*. New York, NY: Wiley, 1996.

17 Kuo, S. M. and D. R. Morgan. "Active Noise Control: A Tutorial Review." *Procedures of The IEEE*, 87, no. 6 (1999): 943–973.

18 This wave file is extracted from WaveWarp, the audio software from Sounds Logical, *http://www.soundslogical.com*.