

Musical Signal Processing with LabVIEW (All Modules)

Collection Editor:

Sam Shearman

Musical Signal Processing with LabVIEW (All Modules)

Collection Editor:

Sam Shearman

Author:

Ed Doering

Online:

< <http://cnx.org/content/col10507/1.3/> >

C O N N E X I O N S

Rice University, Houston, Texas

This selection and arrangement of content as a collection is copyrighted by Sam Shearman. It is licensed under the Creative Commons Attribution 2.0 license (<http://creativecommons.org/licenses/by/2.0/>).

Collection structure revised: January 5, 2010

PDF generated: January 13, 2010

For copyright and attribution information for the modules contained in this collection, see p. 154.

Table of Contents

Musical Signal Processing with LabVIEW	1
1 LabVIEW Programming Techniques for Audio Signal Processing	
1.1 Getting Started with LabVIEW	9
1.2 Editing Tips for LabVIEW	12
1.3 Essential Programming Structures in LabVIEW	12
1.4 Create a SubVI in LabVIEW	18
1.5 Arrays in LabVIEW	18
1.6 Audio Output Using LabVIEW's "Play Waveform" Express VI	20
1.7 Real-Time Audio Output in LabVIEW	20
1.8 Audio Sources in LabVIEW	20
1.9 Reading and Writing Audio Files in LabVIEW	21
Solutions	??
2 Introduction to Audio and Musical Signals	
2.1 Perception of Sound	23
2.2 [mini-project] Musical intervals and the equal-tempered scale	27
Solutions	31
3 Analog Synthesis and Modular Synthesizers	
3.1 Analog Synthesis Modules	33
3.2 [mini-project] Compose a piece of music using analog synthesizer techniques	36
Solutions	40
4 MIDI for Synthesis and Algorithm Control	
4.1 MIDI Messages	41
4.2 Standard MIDI Files	48
4.3 Useful MIDI Software Utilities	51
4.4 [mini-project] Parse and analyze a standard MIDI file	52
4.5 [mini-project] Create standard MIDI files with LabVIEW	55
4.6 [LabVIEW application] MIDI_JamSession	64
Solutions	67
5 Tremolo and Vibrato Effects (Low-Frequency Modulation)	
5.1 Tremolo Effect	69
5.2 [mini-project] Vibraphone virtual musical instrument (VMI) in LabVIEW	72
5.3 Vibrato Effect	75
5.4 [mini-project] "The Whistler" virtual musical instrument (VMI) in LabVIEW	78
Solutions	82
6 Modulation Synthesis	
6.1 Amplitude Modulation (AM) Mathematics	83
6.2 Pitch Shifter with Single-Sideband AM	87
6.3 [mini-project] Ring Modulation and Pitch Shifting	89
6.4 Frequency Modulation (FM) Mathematics	91
6.5 Frequency Modulation (FM) Techniques in LabVIEW	95
6.6 Chowning FM Synthesis Instruments in LabVIEW	96
6.7 [mini-project] Chowning FM Synthesis Instruments	98
Solutions	101
7 Additive Synthesis	
7.1 Additive Synthesis Techniques	105

7.2	Additive Synthesis Concepts	107
7.3	[mini-project] Risset Bell Synthesis	109
7.4	[mini-project] Spectrogram Art	112
	Solutions	??
8	Subtractive Synthesis	
8.1	Subtractive Synthesis Concepts	115
8.2	Interactive Time-Varying Digital Filter in LabVIEW	119
8.3	Band-Limited Pulse Generator	120
8.4	Formant (Vowel) Synthesis	122
8.5	Linear Prediction and Cross Synthesis	124
8.6	[mini-project] Linear Prediction and Cross Synthesis	126
8.7	Karplus-Strong Plucked String Algorithm	128
8.8	Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy	130
	Solutions	136
9	Sound Spatialization and Reverberation	
9.1	Reverberation	137
9.2	Schroeder Reverberator	141
9.3	Localization Cues	144
	Solutions	147
	Index	148
	Attributions	154

Musical Signal Processing with LabVIEW¹

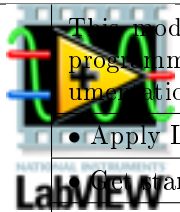
	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide² module for tutorials and documentation that will help you:</p> <ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 1

Introduction

Music synthesis and audio signal processing apply digital signal processing (DSP) concepts to create and control sound and to apply interesting special effects to musical signals. As you implement and experiment with synthesis and filtering algorithms, you develop a deeper understanding of the inter-relationships between a physical sound, its visual representations such as time-domain waveform and spectrogram, and its mathematical model.

Richard Hamming once said "the purpose of computing is insight, not numbers," and you gain manifold insights when you interact with a signal processing system that you created yourself. The LabVIEW Development Environment by National Instruments Corporation is an excellent tool you can use to convert mathematical algorithms into real time interactive music synthesizers and audio signal processors. The unique graphical dataflow programming environment of LabVIEW allows DSP elements such as signal generators, filters, and other mathematical operators to be placed and interconnected as a block diagram. Placing user controls and indicators on the block diagram automatically generates an interactive graphical user interface (GUI) front-panel display, dramatically reducing the development effort needed to make an interactive application.

Musical Signal Processing with LabVIEW, a multimedia educational resource for students and faculty, augments traditional DSP courses and supports dedicated courses in music synthesis and audio signal processing. Each of the learning modules blends video, text, sound clips, and LabVIEW **virtual instruments (VIs)** into explanation of theory and concepts, demonstration of LabVIEW implementation techniques to transform theory into working systems, and hands-on guided project activities. **Screencasts** – videos captured directly from the computer screen with audio narration and a hallmark of this resource – use a mixture of hand-drawn text, animations, and video of the LabVIEW tool in operation to provide a visually rich learning environment.

¹This content is available online at <<http://cnx.org/content/m15510/1.3/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

Learning Module Collections

- LabVIEW Programming Techniques for Audio Signal Processing³ – After completing this course you will be well-equipped to start creating your own audio and signal processing applications within the LabVIEW development environment. The course includes a "Getting Started" tutorial, editing tips, essential programming structures, subVIs, arrays, audio sources, audio output to the soundcard, reading and writing audio files, and real-time audio output with interactive parameter control.
- Introduction to Audio and Musical Signals⁴ – Learn about human perception of sound, including pitch and frequency, intensity and amplitude, harmonics, and tuning systems. The treatment of these concepts is oriented to the creation of music synthesis algorithms. A hands-on project investigates the specific choice of frequencies for the tuning system called "equal temperament," the most common tuning system for Western music.
- Analog Synthesis and Modular Synthesizers⁵ – Analog modular synthesizers popular in the 1960s and 1970s produce sound with electronic devices such as oscillators, amplifiers, filters, and envelope generators linked together by cables. A specific cable configuration (or "patch") produces a distinct sound controlled by a keyboard or sequencer. While digital synthesis has largely replaced analog synthesizers, the concepts and techniques of analog synthesis still serve as the basis for many types of synthesis algorithms. Learn about modular synthesizers and use LabVIEW to compose a piece of music by emulating an analog synthesizer.
- MIDI for Synthesis and Algorithm Control⁶ – The Musical Instrument Digital Interface (MIDI) standard specifies how to convey performance control information between synthesizer equipment and computers. Standard MIDI files (.mid extension) include timing information with MIDI messages to embody a complete musical performance. Learn about the MIDI standard, discover useful MIDI-related software utilities, and learn how to use LabVIEW to create a standard MIDI file according to your own design that can be played by any media appliance. Also learn about "MIDI JamSession," a LabVIEW application VI that renders standard MIDI files to audio using "virtual musical instruments" of your own design.
- Tremolo and Vibrato Effects (Low-Frequency Modulation)⁷ – Tremolo and vibrato add interest to the sound of musical instruments and the singing voice. Tremolo is a low-frequency variation in amplitude, while vibrato is a low-frequency variation in frequency. Learn how to model each of these effects mathematically, and discover how to implement these effects in LabVIEW.
- Modulation Synthesis⁸ – Amplitude modulation (AM) and frequency modulation (FM) are familiar types of communications systems. When the modulating frequency is in the audio range, AM (also called ring modulation) produces interesting special effects by shifting the source signal spectrum, and can be used to raise or lower the pitch of an instrument or voice. FM creates rich, time-varying spectra that can be designed to emulate the sound of many different musical instruments. Learn about the mathematics of AM and FM, and learn how to implement these modulation schemes as audio signal processors and music synthesizers in LabVIEW.
- Additive Synthesis⁹ – Additive synthesis creates complex sounds by adding together individual sinusoidal signals called partials. A partial's frequency and amplitude are each time-varying functions, so a partial is a more flexible version of the harmonic associated with a Fourier series decomposi-

³*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
<http://cnx.org/content/col10440/latest/>

⁴*Musical Signal Processing with LabVIEW – Introduction to Audio and Musical Signals*
<http://cnx.org/content/col10481/latest/>

⁵*Musical Signal Processing with LabVIEW – Analog Synthesis and Modular Synthesizers*
<http://cnx.org/content/col10480/latest/>

⁶*Musical Signal Processing with LabVIEW – MIDI for Synthesis and Algorithm Control*
<http://cnx.org/content/col10487/latest/>

⁷*Musical Signal Processing with LabVIEW – Tremolo and Vibrato Effects (Low-Frequency Modulation)*
<http://cnx.org/content/col10482/latest/>

⁸*Musical Signal Processing with LabVIEW – Modulation Synthesis* <http://cnx.org/content/col10483/latest/>

⁹*Musical Signal Processing with LabVIEW – Additive Synthesis* <http://cnx.org/content/col10479/latest/>

tion of a periodic waveform. Learn about partials, how to model the timbre of natural instruments, various sources of control information for partials, and how to make a sinusoidal oscillator with an instantaneous frequency that varies with time.

- Subtractive Synthesis¹⁰ – Most musical instruments as well as the human voice create sound by exciting a resonant structure or cavity by a wideband pulsed source. The resonant structure amplifies select frequency bands (called formants) and suppresses (or "subtracts") others. Subtractive synthesis algorithms use time-varying sources and time-varying digital filters to model physical instruments. Learn how to use the DSP capabilities of LabVIEW to implement an interactive time-varying filter, a band-limited wideband source, a vowel synthesizer for speech, a "cross synthesizer" in which a speech signal's spectral envelope is superimposed on a musical signal, and a remarkably life-like plucked string sound.
- Sound Spatialization and Reverberation¹¹ – Reverberation is a property of concert halls that greatly adds to the enjoyment of a musical performance. Sound waves propagate directly from the stage to the listener, and also reflect from the floor, walls, ceiling, and back wall of the stage to create myriad copies of the direct sound that are time-delayed and reduced in intensity. Learn how to model a reverberant environment using comb filters and all-pass filters, and how to implement these digital filters in LabVIEW to create an audio signal processor that can add reverberation an audio signal. In addition, learn how to place a virtual sound source in a stereo sound field using interaural intensity difference (IID) and interaural timing difference (ITD) localization cues.

Learning Module Descriptions

LabVIEW Programming Techniques for Audio Signal Processing

- Getting Started with LabVIEW (Section 1.1) – Learn about the LabVIEW programming environment, create your first virtual instrument (VI), learn about LabVIEW's graphical dataflow programming paradigm, become acquainted with some of LabVIEW's data types, and review some useful debugging techniques.
- Editing Tips for LabVIEW (Section 1.2) – Learn how to efficiently create and edit LabVIEW block diagrams and front panels.
- Essential Programming Structures in LabVIEW (Section 1.3) – Learn how to work with LabVIEW's essential programming structures such as for-loops, while-loops, case structure, MathScript node, and diagram disable.
- Create a SubVI in LabVIEW (Section 1.4) – A **subVI** is equivalent to a function, subroutine, or method in other programming languages, and useful for encapsulating code that will be reused multiple time. A subVI is also used to develop hierarchical programs.
- Arrays in LabVIEW (Section 1.5) – Learn how to create and manipulate arrays, perform mathematical operations on them, and use spreadsheets to read and write arrays to the file system.
- Audio Output Using LabVIEW's "Play Waveform" Express VI (Section 1.6) – Learn how to play an audio signal (1-D array) using your computer's soundcard.
- Audio Sources in LabVIEW (Section 1.8) – Learn how to use the 'Sine Wave' subVI from the Signal Processing palette as an audio source.
- Reading and Writing Audio Files in LabVIEW (Section 1.9) – Learn how to use LabVIEW to retrieve an audio signal from a WAV-format file, and how to save an audio signal that you have created to a WAV-format file.
- Real-Time Audio Output in LabVIEW (Section 1.7) – Learn how to set up the framework for your own LabVIEW application that can generate continuous audio output and respond to changes on the front panel in real time.

¹⁰ *Musical Signal Processing with LabVIEW – Subtractive Synthesis* <<http://cnx.org/content/col10484/latest/>>

¹¹ *Musical Signal Processing with LabVIEW – Sound Spatialization and Reverberation*
<<http://cnx.org/content/col10485/latest/>>

Introduction to Audio and Musical Signals

- Perception of Sound (Section 2.1) – A basic understanding of human perception of sound is vital if you wish to design music synthesis algorithms to achieve your goals. In this module, learn about pitch and frequency, intensity and amplitude, harmonics, and tuning systems. The treatment of these concepts is oriented to the creation of music synthesis algorithms.
- Mini-Project: Musical Intervals and the Equal-Tempered Scale (Section 2.2) – Learn about musical intervals, and discover the reason behind the choice of frequencies for the tuning system called "equal temperament."

Analog Synthesis and Modular Synthesizers

- Analog Synthesis Modules (Section 3.1) – Learn about analog synthesizer modules, the foundation for synthesizers based on analog electronics technology. While analog synthesis has largely been replaced by digital techniques, the concepts associated with analog modular synthesis (oscillators, amplifiers, envelope generators, and patches) still form the basis for many digital synthesis algorithms.
- Mini-Project: Compose a Piece of Music Using Analog Synthesizer Techniques (Section 3.2) – Design sounds in LabVIEW using analog synthesis techniques. You will create two subVIs: one to implement an ADSR-style envelope generator and the other to create a multi-voice sound source. You will then create a top-level application VI to render a simple musical composition as an audio file.

MIDI for Synthesis and Algorithm Control

- MIDI Messages (Section 4.1) – Basic MIDI messages include those that produce sound, select voices, and vary a sound in progress, such as pitch bending. In this module, learn about the most common types of MIDI messages at the byte level, including: Note-On, Note-Off, Program Change, Control Change, Bank Select, Pitch Wheel, and Sys-Exclusive. The General MIDI (GM) standard sound set is also introduced.
- Standard MIDI Files (Section 4.2) – A complete musical performance can be recorded by sequencing software, which saves individual MIDI messages generated by a synthesizer and measures the time interval between them. The messages and timing information is stored in a standard MIDI file, a binary-format file designed to maximize flexibility and minimize file size. In this module, learn how to understand the structure of a standard MIDI file at the byte level.
- Useful MIDI Software Utilities (Section 4.3) – Freeware MIDI-related software utilities abound on the Internet; especially useful utilities are introduced here. Each section includes a screencast video to illustrate how to use the utility.
- Mini-Project: Parse and Analyze a Standard MIDI File (Section 4.4) – This mini-project develops your ability to interpret the binary file listing of a standard MIDI file. First parse the file into its component elements (headers, MIDI messages, meta-events, and delta-times), then analyze your results.
- Mini-Project: Create Standard MIDI Files with LabVIEW (Section 4.5) – In this project, create your own LabVIEW application that can produce a standard MIDI file. First develop a library of utility subVIs that produce the various components of the file (header chunk, track chunks, MIDI messages, meta-events, and delta times), as well as a subVI to write the finished binary file. Next, combine these into a top-level VI (application) that creates a complete MIDI file based on an algorithm of your choosing.
- LabVIEW Application: MIDI JamSession (Section 4.6) – MIDI_JamSession is a LabVIEW application VI that reads a standard MIDI file (.mid format) and renders it to audio using subVIs called **virtual musical instruments** (VMIs) that you design.

Tremolo and Vibrato Effects (Low-Frequency Modulation)

- Tremolo Effect (Section 5.1) – Tremolo is a type of low-frequency amplitude modulation. Learn about the vibraphone, a mallet-type percussion instrument that can create tremolo, experiment with the tremolo effect using an interactive LabVIEW VI, and learn how to model the tremolo effect mathematically.
- Mini-Project: Vibraphone Virtual Musical Instrument (VMI) in LabVIEW (Section 5.2) – The vibraphone percussion instrument can be well-modeled by a sinusoidal oscillator, an attack-decay envelope with a short attack and a long decay, and a low-frequency sinusoidal amplitude modulation. In this mini-project, develop code to model the vibraphone as a LabVIEW "virtual musical instrument" (VMI) that can be "played" by a MIDI music file.
- Vibrato Effect (Section 5.3) – Vibrato is a type of low-frequency frequency modulation. Learn about vibrato produced by the singing voice and musical instruments, experiment with the vibrato effect using an interactive LabVIEW VI, and learn how to model the vibrato effect mathematically.
- Mini-Project: "The Whistler" virtual musical instrument (VMI) in LabVIEW (Section 5.4) – An individual who can whistle with vibrato can be well-modeled by a sinusoidal oscillator, an attack-sustain-release envelope with a moderate attack and release time, and a low-frequency sinusoidal frequency modulation. In this mini-project, develop code to model the whistler as a LabVIEW "virtual musical instrument" (VMI) to be "played" by a MIDI file.

Modulation Synthesis

- Amplitude Modulation (AM) Mathematics (Section 6.1) – Amplitude modulation (AM) creates interesting special effects when applied to music and speech signals. The mathematics of the modulation property of the Fourier transform are presented as the basis for understanding the AM effect, and several audio demonstrations illustrate the AM effect when applied to simple signals (sinusoids) and speech signals. The audio demonstration is implemented by a LabVIEW VI using an event structure as the basis for real-time interactive parameter control.
- Pitch Shifter with Single-Sideband AM (Section 6.2) – Pitch shifting makes an interesting special effect, especially when applied to a speech signal. Single-sideband amplitude modulation (SSB-AM) is presented as a method to shift the spectrum of a source signal in the same way as basic AM, but with cancellation of one sideband to eliminate the "dual voice" sound of conventional AM. Pre-filtering of the source signal to avoid aliasing is also discussed.
- Mini-Project: Ring Modulation and Pitch Shifting (Section 6.3) – Create a LabVIEW VI to experiment with ring modulation (also called amplitude modulation, or AM), and develop a LabVIEW VI to shift the pitch of a speech signal using the single-sideband modulation technique.
- Frequency Modulation (FM) Mathematics (Section 6.4) – Frequency modulation (FM) in the audio frequency range can create very rich spectra from only two sinusoidal oscillators, and the spectra can easily be made to evolve with time. The mathematics of FM synthesis is developed, and the spectral characteristics of the FM equation are discussed. Audio demonstrations as implemented by LabVIEW VIs illustrate the relationships between the three fundamental FM synthesis parameters (carrier frequency, modulation frequency, modulation index) and the synthesized spectra.
- Frequency Modulation (FM) Techniques in LabVIEW (Section 6.5) – Frequency modulation synthesis (FM synthesis) creates a rich spectrum using only two sinusoidal oscillators. Implementing the basic FM synthesis equation in LabVIEW requires a special technique in order to make one oscillator vary the phase function of the other oscillator. In this module, learn how to implement the basic FM equation, and also hear an audio demonstration of the equation in action.
- Chowning FM Synthesis Instruments in LabVIEW (Section 6.6) – John Chowning pioneered frequency modulation (FM) synthesis in the 1970s, and demonstrated how the technique could simulate a diversity of instruments such as brass, woodwinds, and percussion. FM synthesis produces rich spectra from only two sinusoidal oscillators, and more interesting sounds can be produced by using a time-varying

modulation index to alter the effective bandwidth and sideband amplitudes over time. A LabVIEW VI is developed to implement the sound of a clarinet, and the VI can be easily modified to simulate the sounds of many other instruments.

- Mini-Project: Chowning FM Synthesis Instruments (Section 6.7) – Implement several different Chowning FM instruments (bell, wood drum, brass, clarinet, and bassoon) and compare them to the sounds of physical instruments. Develop code to model the Chowning algorithms as LabVIEW "virtual musical instruments" (VMIs) to be "played" by a MIDI file within MIDI JamSession (Section 4.6).

Additive Synthesis

- Additive Synthesis Concepts (Section 7.2) – Additive synthesis creates complex sounds by adding together individual sinusoidal signals called partials. A partial's frequency and amplitude are each time-varying functions, so a partial is a more flexible version of the harmonic associated with a Fourier series decomposition of a periodic waveform. Learn about partials, how to model the timbre of natural instruments, various sources of control information for partials, and how to make a sinusoidal oscillator with an instantaneous frequency that varies with time.
- Additive Synthesis Techniques (Section 7.1) – Learn how to synthesize audio waveforms by designing the frequency and amplitude trajectories of partials. LabVIEW programming techniques for additive synthesis will also be introduced in two examples.
- Mini-Project: Risset Bell Synthesis (Section 7.3) – Use additive synthesis to emulate the sound of a bell using a technique described by Jean-Claude Risset, an early pioneer in computer music.
- Mini-Project: Spectrogram Art (Section 7.4) – Create an oscillator whose output tracks a specified amplitude and frequency trajectory, and then define multiple frequency/amplitude trajectories that can be combined to create complex sounds. Learn how to design the sound so that its spectrogram makes a recognizable picture.

Subtractive Synthesis

- Subtractive Synthesis Concepts (Section 8.1) – Subtractive synthesis describes a wide range of synthesis techniques that apply a filter (usually time-varying) to a wideband excitation source such as noise or a pulse train. The filter shapes the wideband spectrum into the desired spectrum. This excitation/filter technique well-models many types of physical instruments and the human voice. Excitation sources and time-varying digital filters are introduced in this module.
- Interactive Time-Varying Digital Filter in LabVIEW (Section 8.2) – A time-varying digital filter can easily be implemented in LabVIEW, and this module demonstrates the complete process necessary to develop a digital filter that operates in real-time and responds to parameter changes from the front panel controls. An audio demonstration of the finished result includes discussion of practical issues such as eliminating click noise in the output signal.
- Band-Limited Pulse Generator (Section 8.3) – Subtractive synthesis techniques often require a wideband excitation source such as a pulse train to drive a time-varying digital filter. Traditional rectangular pulses have theoretically infinite bandwidth, and therefore always introduce aliasing noise into the input signal. A band-limited pulse (BLP) source is free of aliasing problems, and is more suitable for subtractive synthesis algorithms. The mathematics of the band-limited pulse is presented, and a LabVIEW VI is developed to implement the BLP source. An audio demonstration is included.
- Formant (Vowel) Synthesis (Section 8.4) – Speech and singing contain a mixture of voiced and unvoiced sounds (sibilants like "s"). The spectrum of a voiced sound contains characteristic resonant peaks called formants caused by frequency shaping of the vocal tract. In this module, a formant synthesizer is developed and implemented in LabVIEW. The filter is implemented as a set of parallel two-pole resonators (bandpass filters) that filter a band-limited pulse source.

- Linear Prediction and Cross Synthesis (Section 8.5) – Linear prediction coding (LPC) models a speech signal as a time-varying filter driven by an excitation signal. The time-varying filter coefficients model the vocal tract spectral envelope. "Cross synthesis" is an interesting special effect in which a musical instrument signal drives the digital filter (or vocal tract model), producing the sound of a "singing instrument." The theory and implementation of linear prediction are presented in this module.
- Mini-Project: Linear Prediction and Cross Synthesis (Section 8.6) – Linear prediction is a method used to estimate a time-varying filter, often as a model of a vocal tract. Musical applications of linear prediction substitute various signals as excitation sources for the time-varying filter. This mini-project guides you to develop the basic technique for computing and applying a time-varying filter in LabVIEW. After experimenting with different excitation sources and linear prediction model parameters, you will develop a VI to cross-synthesize a speech signal and a musical signal.
- Karplus-Strong Plucked String Algorithm (Section 8.7) – The Karplus-Strong algorithm plucked string algorithm produces remarkably realistic tones with modest computational effort. The algorithm requires a delay line and lowpass filter arranged in a closed loop, which can be implemented as a single digital filter. The filter is driven by a burst of white noise to initiate the sound of the plucked string. Learn about the Karplus-Strong algorithm and how to implement it as a LabVIEW "virtual musical instrument" (VMI) to be played from a MIDI file using "MIDI JamSession."
- Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy (Section 8.8) – The basic Karplus-Strong plucked string algorithm must be modified with a continuously adjustable loop delay to produce an arbitrary pitch with high accuracy. An all-pass filter provides a continuously-adjustable fractional delay, and is an ideal device to insert into the closed loop. The delay characteristics of both the lowpass and all-pass filters are explored, and the modified digital filter coefficients are derived. The filter is then implemented as a LabVIEW "virtual musical instrument" (VMI) to be played from a MIDI file using "MIDI JamSession."

Sound Spatialization

- Reverberation (Section 9.1) – Reverberation is a property of concert halls that greatly adds to the enjoyment of a musical performance. Sound waves propagate directly from the stage to the listener, and also reflect from the floor, walls, ceiling, and back wall of the stage to create myriad copies of the direct sound that are time-delayed and reduced in intensity. In this module, learn about the concept of reverberation in more detail and ways to emulate reverberation using a digital filter structure known as a comb filter.
- Schroeder Reverberator (Section 9.2) – The Schroeder reverberator uses parallel comb filters followed by cascaded all-pass filters to produce an impulse response that closely resembles a physical reverberant environment. Learn how to implement the Schroeder reverberator block diagram as a digital filter in LabVIEW, and apply the filter to an audio .wav file.
- Localization Cues (Section 9.3) – Learn about two localization cues called interaural intensity difference (IID) and interaural timing difference (ITD), and learn how to create a LabVIEW implementation that places a virtual sound source in a stereo sound field.

Chapter 1

LabVIEW Programming Techniques for Audio Signal Processing

1.1 Getting Started with LabVIEW¹

1.1.1 Overview

Welcome to the exciting world of **LabVIEW** for audio and signal processing applications! This module contains five **screencast** videos, meaning that the videos were captured directly from my computer screen. As I operate the LabVIEW software, I explain each step and discuss what is going on. After you watch each of the videos, you will possess a good idea of some fundamental LabVIEW concepts, including:

- Front panel and block diagram programming paradigm
- Dataflow paradigm
- Data types
- Broken wires
- Debugging techniques

1.1.2 A Bit of History

In 1986 National Instruments Corporation released the first version of LabVIEW (**L**aboratory **V**irtual **I**nstrument **E**ngineering **W**orkbench), which was designed to help engineers use a computer (the Apple Macintosh) to control and gather data from electronic instrumentation (voltmeters, oscilloscopes, and the like) all interconnected by the standard General Purpose Instrumentation Bus, or GP-IB. From its inception, LabVIEW programming was **graphical** in nature. Instead of writing a text file and compiling it to an executable, you connect various elements such as **controls**, **indicators**, **nodes** and **subVIs** together with **wires**, and in this way create a **blockdiagram**. The controls and indicators reside on the **front panel**, which looks just a traditional electronic instrument, i.e., it can have knobs, sliders, buttons, and display panels. The complete application is called a **virtual instrument**, or **VI** for short.

LabVIEW has since evolved into a complete programming environment; anything that you can imagine can probably be implemented in LabVIEW. Recent versions of LabVIEW have added a full suite of tools for doing signal processing, and since soundcard operations are provided, it becomes natural to develop audio signal processing applications in LabVIEW.

LabVIEW's interactive front panel offers a unique opportunity to explore signal processing concepts in real time. As you work your way through other modules in this series, you will learn how to implement

¹This content is available online at <<http://cnx.org/content/m14764/1.4/>>.

your own applications whose user interface consists of knobs, sliders and switches that can adjust processing parameters **while you listen** to the results.

1.1.3 Your First Virtual Instrument (VI)

Watch the following screencast video to learn how to connect front panel **controls** and **indicators** together. You will also learn how to use the **While Loop** structure to make your VI operate continuously until you press a “STOP” button on the front panel.

Image not finished

Figure 1.1: [video] Creating your first "VI" (Virtual Instrument)

1.1.4 The Dataflow Concept

The notion of LabVIEW’s **dataflow** programming paradigm must be grasped **immediately** in order for you to make forward progress learning about LabVIEW to create your own applications. Dataflow programming means that valid data must be present at **all** of the input **terminals** on a node (or subVI) before that node (or subVI) will produce valid data on its output terminals. Moreover, the node (or subVI) does **not** continually process its inputs data unless it is embedded in some sort of looping structure.

Click on the following animation of the dataflow concept to watch a screencast video that explains and further explores the dataflow programming concept.

Image not finished

Figure 1.2: [video] Understanding the LabVIEW "dataflow" paradigm

1.1.5 Data Types

LabVIEW supports a broad range of **data types**, including **numeric**, **Boolean**, and **string**. The following screencast video will acquaint you with the floating point and integer styles of numeric data type, as well as the Boolean data type. The screencast explains the significance of the **coercion indicator** – the red dot that flags a mismatch on data types applied to the input of a node or subVI – as well as **data type conversion** nodes that you can use to intentionally convert a value from one data type to another.

Image not finished

Figure 1.3: [video] Datatypes: Numeric and Boolean

1.1.6 Broken Wires

Broken wires indicate an error that must be corrected before your VI will run. Broken wires result from a number of causes, and it is important to understand why the wire is broken and how to correct the situation. The following screencast describes broken wires in detail.

Image not finished

Figure 1.4: [video] Understanding and correcting broken wires

1.1.7 Debugging Techniques

As you begin learning LabVIEW so that you can develop your own VIs, you will find the debugging techniques described by the next screencast video helpful. Topics include adding additional indicators, using the **Highlight Execution** feature, using the **Retain Wire Values** feature, **single-stepping**, viewing wire values using **probes**, and creating **breakpoints** to pause execution when new data is available on a wire.

Image not finished

Figure 1.5: [video] Basic debugging techniques

1.1.8 For Further Study

If you are new to LabVIEW, I recommend the excellent text by Robert H. Bishop, *Learning with LabVIEW 8* (Pearson Prentice Hall, 2007, ISBN 0-13-239025-6). With this text you can learn basic LabVIEW programming techniques and get a better idea of everything that LabVIEW has to offer.

Once you have developed some skill with LabVIEW, consider Peter A. Blume's text, *The LabVIEW Style Book* (Prentice Hall 2007, ISBN 0-13-145835-3). This text covers a wide variety of techniques that will help you to develop robust and well-designed LabVIEW applications.

Don't forget to check out the on-line documentation that is part of your LabVIEW product installation.

Visit the National Instruments² website, including their Academic³ page and the NI Developer Zone⁴ .

1.2 Editing Tips for LabVIEW⁵

1.2.1 Overview

LabVIEW offers an extensive range of techniques to create and edit block diagrams and front panels. This screencast video reviews some of the commonly-used editing techniques, including:

- adding comments and documentation with **text labels**
- placing front panel terminals as **icons**
- **replacing** front panel controls and indicators with alternate forms
- creating **controls**, **constants**, and **indicators** directly at the **terminal** (using context menu)
- cleaning up wire routing (**Clean Up Wire** context menu)
- **moving** objects using **arrow keys** (fine positioning) and Shift+arrow keys (larger steps)
- **aligning** multiple objects
- **replacing** nodes and subVIs with similar forms
- **removing** all **broken wires** (Ctrl+B)
- **distributing** objects
- converting front panel **controls** into **constants**
- obtaining help: **Context Help** and detailed **help page**
- adding **decorations** (arrows, lines, text labels)
- viewing the **Navigation Window**
- **replicating** existing structures

Image not finished

Figure 1.6: [video] Editing tips for front panels and block diagrams

1.3 Essential Programming Structures in LabVIEW⁶

1.3.1 Overview

Signal processing applications developed in LabVIEW make frequent use of basic constructs that are common to all high-level programming languages: for-loops, while-loops, and case structures. A **For Loop** repeats a block of code a fixed number of times, a **While Loop** repeats a block of code as long as a particular condition is true, and a **Case Structure** executes one of several blocks of code depending on some selection criterion. After completing this module you will be able to use these three essential structures in your own LabVIEW VIs.

You will also learn about two additional structures. The **MathScript Node** provides a way for you to develop a customized node whose behavior is defined using the MathScript text-based programming

²<http://www.ni.com/>

³<http://www.ni.com/academic/>

⁴<http://zone.ni.com/devzone/cda/main>

⁵This content is available online at <<http://cnx.org/content/m14765/1.3/>>.

⁶This content is available online at <<http://cnx.org/content/m14766/1.6/>>.

language. MathScript syntax and functions are quite similar to MATLAB, so the MathScript node can help you to leverage any MATLAB programming experience you may have. Lastly, the **Diagram Disable** structure is useful when you need to temporarily “comment out” a portion of your LabVIEW code.

The following diagram highlights the structures on the “Programming | Structures” palette about which you will learn in this module:

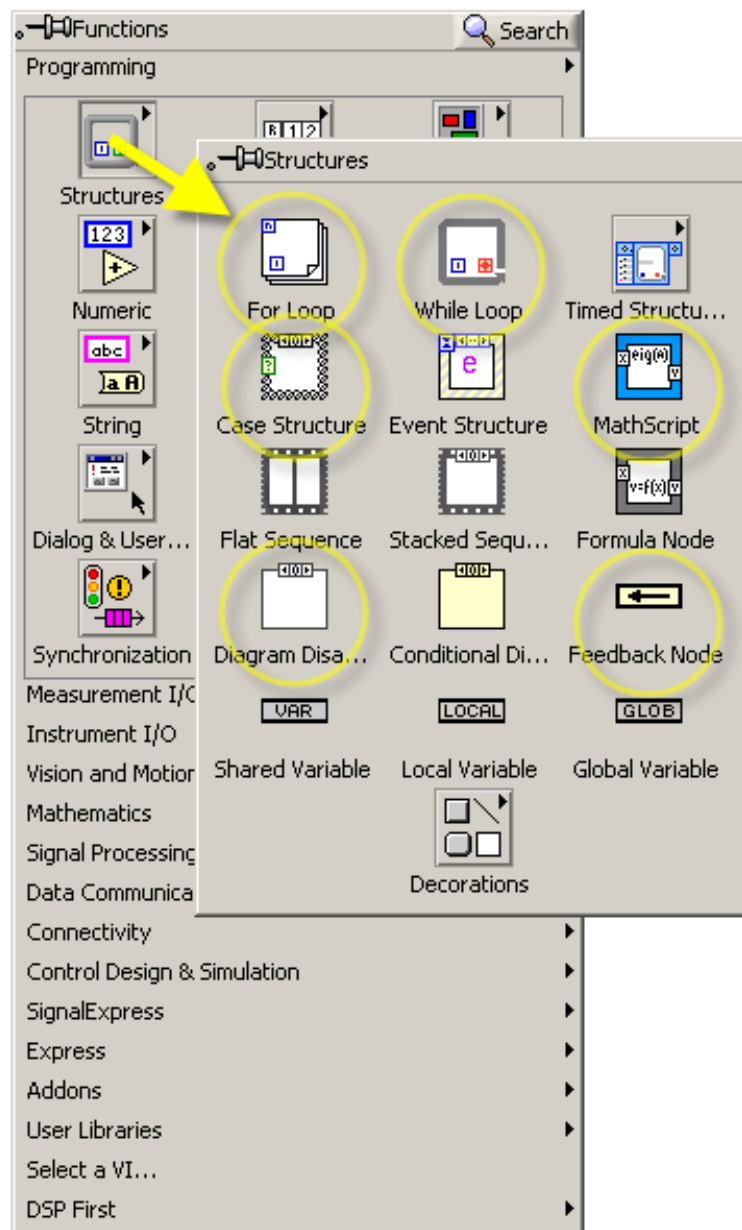


Figure 1.7: Structures on the "Programming | Structures" palette described in this module

1.3.2 For-Loop Structure

1.3.2.1 Basic concepts

The **For Loop** structure provides a means to repeatedly run a block of code (or **subdiagram**) for a fixed number of times. The following screencast video introduces the **For Loop** structure, including concepts such as **loop count terminal**, **iterator**, **loop tunnel**, **indexing**, **shift register**, and the **feedback node**.

Image not finished

Figure 1.8: [video] LabVIEW Techniques: For-Loop structure

1.3.2.2 Working with arrays as inputs

The **For Loop** structure works efficiently with arrays. The next screencast video describes how to work with arrays that serve as inputs to the for-loop structure. Array elements can be used individually or collectively within the for-loop, depending on whether you have enabled **indexing** on the **loop tunnel**. Indexing on the output side of the for-loop can also be used to store a value (either a scalar or an array) on each iteration of the for-loop, thereby producing an array as output. Arrays can also be **concatenated**, a standard technique for constructing an audio signal from individual segments.

Image not finished

Figure 1.9: [video] LabVIEW Techniques: For-Loop structure with arrays as input

1.3.3 While-Loop Structure

The **While Loop** structure is similar to the **For Loop** structure with its ability to repeatedly run a **subdiagram**, but the number of times is not fixed in advance. Instead, the while-loop structure will execute its subdiagram as long as a particular condition is true. The following screencast will show you how to use the **While Loop** structure.

Image not finished

Figure 1.10: [video] LabVIEW Techniques: While-Loop structure

1.3.4 Case Structure

The **Case Structure** provides a mechanism by which exactly one of several possible subdiagrams will be executed, depending on the value connected to the **selector terminal**. When the selector terminal is a Boolean type (either True or False), the case structure implements the “if-else” construct of text-based languages. When the selector terminal is an integer type, the case structure implements the “case” or “switch” construct of text-based languages.

The following screencast introduces you to the **Case Structure**. You will learn how to add and delete subdiagrams, how to choose the default subdiagram, and how to ensure that valid outputs are generated for all possible cases. The **Boolean** and **integer** data types are covered in this screencast; the next screencast describes how to work with the **string** and **enumerated** data types, which provide a user-friendly way to select cases from the front panel.

Image not finished

Figure 1.11: [video] LabVIEW Techniques: Case Structure with "Boolean" and "integer" data types at the selector terminal

Image not finished

Figure 1.12: [video] LabVIEW Techniques: Case Structure with "string" and "enumerated" data types at the selector terminal

1.3.5 MathScript Node

The **MathScript Node** offers a convenient way to implement a programming concept that may be otherwise difficult to implement using standard G code (i.e., creating LabVIEW block diagrams by wiring available structures and nodes). **MathScript** is a text-based programming language that uses syntax very similar to **MATLAB**. If you have prior experience with MATLAB, you can easily develop and debug a MathScript-based script in the **MathScript interactive window**, then copy the text into a MathScript node on your block diagram. After you create input and output terminals on the MathScript node, it can be connected to the rest of the block diagram as you would any other structure.

The following screencast video introduces you to the **MathScript Node**. The example walks you through the process to create a specialized node that accepts a scalar N as input and produces a specialized array as output.

Image not finished

Figure 1.13: [video] LabVIEW Techniques: MathScript node and MathScript Interactive Window

```
% Create a triangle array
```

```
x = [linspace(0,1,N/2) linspace(1,0,N/2)]
```

The screencast in this section (just above) walks through creation of a LabVIEW VI that includes a MathScript node.

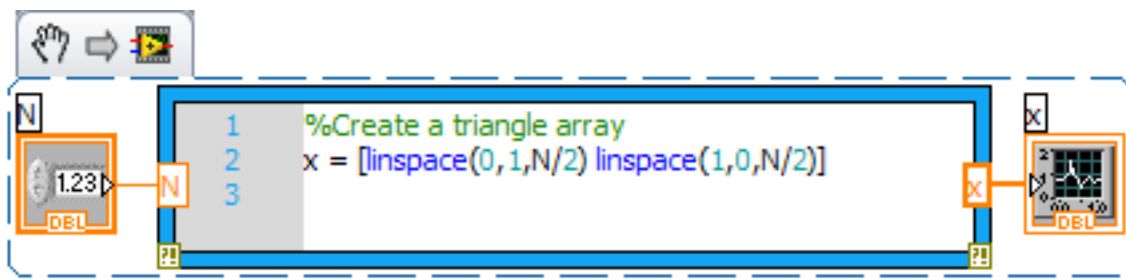


Figure 1.14: The screencast in this section (just above) describes how to build a VI that includes a MathScript node.

1.3.6 Diagram-Disable Structure

You probably have used the software debugging technique known as “commenting out” a block of code; you do this so that you can effectively remove a portion of code without actually deleting it from the file. Watch the next screencast video to learn how to use the **Diagram Disable** structure to “comment out” a portion of your LabVIEW block diagram.

Image not finished

Figure 1.15: [video] LabVIEW Techniques: Use Diagram-Disable structure to "comment out" a section of code

1.4 Create a SubVI in LabVIEW⁷

A **subVI** is the LabVIEW equivalent to “functions,” “subroutines,” and “methods” in other programming languages. You can easily create a subVI that can be used reused many times in other VIs; editing the subVI one time updates its behavior for all VIs that use the subVI, which simplifies program design and maintenance. Moreover, the subVI forms the basic method by which you can create a hierarchical program.

The screencast video below details the necessary steps to convert a conventional block diagram into a subVI by adding a **connector pane** and an **icon**. These two elements enable the block diagram to appear as an element on another block diagram with a connection pattern specific to the requirements of the task performed by the subVI. In addition, you can set the **VI Description** (part of the **VI Properties**) of your subVI so that hovering your cursor over your subVI reveals a helpful description of its behavior.

Image not finished

Figure 1.16: [video] LabVIEW Techniques: Create a subVI from a block diagram

1.5 Arrays in LabVIEW⁸

1.5.1 Overview

Arrays are a fundamental data type for signal processing. LabVIEW offers a complete set of techniques to create and manipulate arrays, and to perform mathematical and signal processing operations on arrays. This module will acquaint you with the basic techniques for working with arrays.

1.5.2 Creating Arrays

The following video screencast describes how to create arrays on the front panel and on the block diagram. The elements of an array can be edited and modified directly, a suitable technique when the array is relatively small.

Image not finished

Figure 1.17: [video] Creating arrays

1.5.3 Manipulating Arrays

The next video screencast explains essential array manipulations for signal processing tasks, including: determining the **dimensions** of an array, retrieving individual elements, rows, columns, or other **subarrays**,

⁷This content is available online at <<http://cnx.org/content/m14767/1.4/>>.

⁸This content is available online at <<http://cnx.org/content/m14768/1.4/>>.

appending (concatenating) arrays, and **reshaping** arrays.

Image not finished

Figure 1.18: [video] Manipulating arrays

1.5.4 Mathematical Operations with Arrays

Signal processing operations commonly operate on all of the array elements at once. For example, adding a scalar constant to an array implies a loop operation in which the constant is added to each element of the array.

The following screencast video describes several techniques for performing mathematical operations on arrays. Important side effects (such as what happens when two arrays of different lengths are added together) are discussed. The **Ramp Pattern** subVI is also described as a method to create a time basis for mathematical functions such as the **exponential**.

Image not finished

Figure 1.19: [video] Performing mathematical operations with arrays

1.5.5 Arrays and Spreadsheets

Spreadsheets are often used as a data storage mechanism by other applications. The next screencast video shows you how to retrieve an array from a comma-separated-values (CSV-format) spreadsheet, and how to create your own CSV-format spreadsheet from an existing array within LabVIEW.

Image not finished

Figure 1.20: [video] Retrieving arrays from a spreadsheet; saving arrays to a spreadsheet

1.6 Audio Output Using LabVIEW's "Play Waveform" Express VI⁹

1.6.1 Overview

A 1-D array representing an audio signal can be played directly by the computer's soundcard using the **Play Waveform** Express VI. The screencast video shows how to listen to an audio signal source (a sinusoid in this example). The 1-D array must first be converted to the **waveform** data type using the **Build Waveform** node.

Image not finished

Figure 1.21: [video] Listen to an audio signal (1-D array) on the soundcard

1.7 Real-Time Audio Output in LabVIEW¹⁰

1.7.1 Overview

The interactive front panel is a hallmark of the LabVIEW programming paradigm. In this screencast video you will learn how to use the low-level **Sound Output** subVIs called **Configure**, **Write**, and **Clear** to build a general-purpose framework that can continually produce audio output, all the while responding to parameter changes on the front panel in real time. The example uses a sinusoidal source, but you can easily adapt the VI for your own signal sources; the finished VI from this example is available: `lvt_audio_realtime-out.vi`¹¹ (right-click and choose "Save As")

Image not finished

Figure 1.22: [video] Real-time audio output with interactive controls

1.8 Audio Sources in LabVIEW¹²

1.8.1 Overview

The **Sine Wave** subVI in the Signal Processing palette is a useful way to produce a sinusoidal audio signal. The screencast video will acquaint you with the Sine Wave subVI, especially its requirement for **normalized frequency**. This subVI is one of several **re-entrant** signal generators, meaning that they have the ability to "pick up where they left off;" that is, they can maintain continuity in their output across execution runs and loop iterations.

⁹This content is available online at <http://cnx.org/content/m14769/1.3/>.

¹⁰This content is available online at <http://cnx.org/content/m14772/1.5/>.

¹¹http://cnx.org/content/m14772/latest/lvt_audio_realtime-out.vi

¹²This content is available online at <http://cnx.org/content/m14770/1.3/>.

Image not finished

Figure 1.23: [video] Sinewave source for audio applications

1.9 Reading and Writing Audio Files in LabVIEW¹³

1.9.1 Overview

LabVIEW offers a variety of ways to read and write audio files in WAV format. After completing this module you will be able to use the **Simple Read** and **Simple Write** subVIs located in the “Programming | Graphics & Sound | Sound | Files” palette to retrieve an audio signal as a 1-D array from a .wav file, and also to save a 1-D array that represents an audio signal to a .wav file. Additional points covered include scaling your audio signal to have a maximum absolute value of one before saving as an audio file, and how to create a two-channel (stereo) audio file.

1.9.2 Retrieve an Audio Signal from a .wav File

The **Sound File Simple Read** subVI accepts a filename for an audio file in .wav format and returns a waveform data type. You can read mono or stereo files, and also determine information such as the audio signal’s sampling frequency and its total number of samples.

The following video screencast shows how to use **Simple Read** to retrieve the audio signal as an array data type that can be used as a signal input for your own VIs. You may want to start LabVIEW now, then follow along to create your own version of the VI pictured below. If so, you will need the two audio files referenced in the screencast: tone-noise.wav¹⁴ and left-right.wav¹⁵.

Image not finished

Figure 1.24: [video] LabVIEW Techniques: Retrieve an audio signal from a .wav file

1.9.3 Save an Audio Signal to a .wav File

The **Sound File Simple Write** subVI accepts a signal in waveform data type and a filename and stores the signal as a .wav file. You can set the number of bits per sample (16 is recommended for best fidelity). You must ensure that your signal values lie in the range -1 to +1, otherwise other applications may not be able to read your .wav file properly.

The next video screencast shows how to use **Simple Write** to save an existing 1-D array that represents an audio signal to a .wav file. The screencast also covers some not-so-obvious data-type transformations

¹³This content is available online at <<http://cnx.org/content/m14771/1.6/>>.

¹⁴See the file at <<http://cnx.org/content/m14771/latest/tone-noise.wav>>

¹⁵See the file at <<http://cnx.org/content/m14771/latest/left-right.wav>>

required to successfully create your .wav file. You may want to start LabVIEW now, then follow along to create your own version of the VI pictured below. If so, you will need the audio file referenced in the screencast: tone-noise.wav¹⁶.

Image not finished

Figure 1.25: [video] LabVIEW Techniques: Save an audio signal to a .wav file

1.9.4 Scale Your Audio Before Saving to a .wav File

As mentioned earlier, you must ensure that all of the samples in your signal lie in the range -1 to +1. It is easy to create signals that exceed this range, especially when you add multiple signals together.

The next screencast illustrates how **Quick Scale** is an easy way to scale your 1-D array such that the maximum absolute value is always one; **Quick Scale** is located in the “Signal Processing | Sig Operation” palette, and should be used as the last step before converting to the waveform data type.

Image not finished

Figure 1.26: [video] LabVIEW Techniques: Scale audio signal to +/- 1 range before saving to a .wav file

1.9.5 Create a Two-Channel (Stereo) .wav File

Creating a stereo .wav file requires that you assemble an array of waveforms, one for the left channel and the other for the right channel. Watch the next screencast video to learn how.

Image not finished

Figure 1.27: [video] LabVIEW Techniques: Save a stereo (two-channel) audio signal to a .wav file

¹⁶See the file at <<http://cnx.org/content/m14771/latest/tone-noise.wav>>

Chapter 2

Introduction to Audio and Musical Signals

2.1 Perception of Sound¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Table 2.1

2.1.1 Introduction

A basic understanding of human perception of sound is vital if you wish to design music synthesis algorithms to achieve your goals. Human hearing and other senses operate quite well in a relative sense. That is, people perceive properties of sound such as pitch and intensity and make relative comparisons. Moreover, people make these comparisons over an enormous dynamic range: they can listen to two people whispering in a quiet auditorium and determine which person is whispering the loudest. During a rock concert in the same auditorium, attendees can determine which vocalist is singing the loudest. However, once the rock concert is in progress, they can no longer hear someone whispering! Senses can adapt to a wide range of conditions, but can make relative comparisons only over a fairly narrow range.

In this module you will learn about **pitch** and **frequency**, **intensity** and **amplitude**, **harmonics** and **overtones**, and **tuning systems**. The treatment of these concepts is oriented to creating music synthesis algorithms. **Connexions** offers many excellent modules authored by Catherine Schmidt-Jones that treat these concepts in a music theory context, and some of these documents are referenced in the discussion below. For example, Acoustics for Music Theory³ describes acoustics in a musical setting, and is a good refresher on audio signals.

¹This content is available online at <<http://cnx.org/content/m15439/1.2/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

³"Acoustics for Music Theory" <<http://cnx.org/content/m13246/latest/>>

2.1.2 Pitch and Frequency

Pitch is the human perception of **frequency**. Often the terms are used interchangeably, but they are actually distinct concepts. Musicians normally refer to the pitch of a signal rather than its frequency; see Pitch: Sharp, Flat, and Natural Notes⁴ and The Circle of Fifths⁵.

Perception of frequency is **logarithmic** in nature. For example, a change in frequency from 400 Hz to 600 Hz will **not** sound the same as a change from 200 Hz to 400 Hz, even though the difference between each of these frequency pairs is 200 Hz. Instead, you perceive changes in pitch based on the **ratio** of the two frequencies; in the previous example, the ratios are 1.5 and 2.0, respectively, and the latter pitch pair would sound like a greater change in frequency. Musical Intervals, Frequency, and Ratio⁶ offers additional insights.

Often it is desirable to synthesize an audio signal so that its perceived pitch follows a specific **trajectory**. For example, suppose that the pitch should begin at a low frequency, gradually increase to a high frequency, and then gradually decrease back to the original. Furthermore, suppose that you should perceive a uniform rate of change in the frequency.

The screencast video of Figure 2.1 illustrates two different approaches to this problem, and demonstrates the perceptual effects that result from treating pitch perception as linear instead of logarithmic.

Image not finished

Figure 2.1: [video] Two approaches to the design of a frequency trajectory: one linear, and the other logarithmic

2.1.3 Intensity and Amplitude

Perception of sound **intensity** also logarithmic. When you judge one sound to be twice as loud as another, you actually perceive the **ratio** of the two sound intensities. For example, consider the case of two people talking with one another. You may decide that one person talks twice as loud as the other, and then measure the acoustic power emanating from each person; call these two measurements T_1 and T_2 . Next, suppose that you are near an airport runway, and decide that the engine noise of one aircraft is twice the intensity of another aircraft (you also measure these intensities as A_1 and A_2). In terms of acoustic intensity, the difference between the talkers $T_2 - T_1$ is negligible compared to the enormous difference in acoustic intensity $A_2 - A_1$. However, the **ratios** T_2/T_1 and A_2/A_1 would be identical.

The **decibel** (abbreviated **dB**) is normally used to describe ratios of acoustic intensity. The decibel is defined in (2.1):

$$R_{\text{dB}} = 10 \log_{10} \left(\frac{I_2}{I_1} \right) \quad (2.1)$$

where I_1 and I_2 represent two acoustic intensities to be compared, and R_{dB} denotes the ratio of the two intensities.

Acoustic intensity measures power per unit area, with a unit of watts per square meter. The operative word here is **power**. When designing or manipulating audio signals, you normally think in terms of **amplitude**, however. The power of a signal is proportional to the **square** of its amplitude. Therefore, when

⁴"Pitch: Sharp, Flat, and Natural Notes" <<http://cnx.org/content/m10943/latest/>>

⁵"The Circle of Fifths" <<http://cnx.org/content/m10865/latest/>>

⁶"Musical Intervals, Frequency, and Ratio" <<http://cnx.org/content/m11808/latest/>>

considering the ratios of two amplitudes A_1 and A_2 , the ratio in decibels is defined as in (2.2):

$$R_{\text{dB}} = 20 \log_{10} \left(\frac{A_2}{A_1} \right) \quad (2.2)$$

Can you explain why "10" becomes "20"? Recall that $\log(a^b) = b \log(a)$.

Often it is desirable to synthesize an audio signal so that its perceived intensity will follow a specific **trajectory**. For example, suppose that the intensity should begin at silence, gradually increase to a maximum value, and then gradually decrease back to silence. Furthermore, suppose that you should perceive a uniform rate of change in intensity.

The screencast video of Figure 2.2 illustrates two different approaches to this problem, and demonstrates the perceptual effects that result from treating intensity perception as linear instead of logarithmic.

Image not finished

Figure 2.2: [video] Two approaches to the design of an intensity trajectory: one linear, and the other logarithmic

2.1.4 Harmonics and Overtones

Musical instruments produce sound composed of a **fundamental** frequency and **harmonics** or **overtones**. The relative strength and number of harmonics produced by an instrument is called **timbre**, a property that allows the listener to distinguish between a violin, an oboe, and a trumpet that all sound the same pitch. See *Timbre: The Color of Music*⁷ for further discussion.

You perhaps have studied the concept of Fourier series, which states that any periodic signal can be expressed as a sum of sinusoids, where each sinusoid is an exact integer multiple of the fundamental frequency; refer to (2.3):

$$f(t) = a_0 + \sum_{n=1}^{\infty} a_n \cos(n2\pi f_0 t) + b_n \sin(n2\pi f_0 t) \quad (2.3)$$

where f_0 is the fundamental frequency (in Hz), n denotes the harmonic number, and a_0 is the DC (constant) offset.

When an instrument produces overtones whose frequencies are essentially integer multiples of the fundamental, you do not perceive all of the overtones as distinct frequencies. Instead, you perceive a single tone; the harmonics **fuse** together into a single sound. When the overtones follow some other arrangement, you perceive multiple tones. Consider the screencast video in Figure 2.3 which explains why physical instruments tend to produce overtones at approximately integer multiples of a fundamental frequency.

Image not finished

Figure 2.3: [video] Why musical instruments produce overtones at approximately integer multiples of a fundamental frequency

⁷"Timbre: The Color of Music" <<http://cnx.org/content/m11059/latest/>>

Musicians broadly categorize combinations of tones as either **harmonious** (also called **consonant**) or **inharmonious** (also called **dissonant**). Harmonious combinations seem to "fit well" together, while inharmonious combinations can sound "rough" and exhibit **beating**. The screencast video in Figure 2.4 demonstrates these concepts using sinusoidal tones played by a synthesizer.

Image not finished

Figure 2.4: [video] Illustration of harmonious and inharmonious sounds using sinusoidal tones

Please refer to the documents Consonance and Dissonance⁸ and Harmony⁹ for more information.

2.1.5 Tuning Systems

A **tuning system** defines a relatively small number of pitches that can be combined into a wide variety of harmonic combinations; see Tuning Systems¹⁰ for an excellent treatment of this subject.

The vast majority of Western music is based on the tuning system called **equal temperament** in which the **octave interval** (a 2:1 ratio in frequency) is equally subdivided into 12 subintervals called **semitones**.

Consider the 88-key piano keyboard below. Each adjacent pair of keys is one semitone apart (you perhaps are more familiar with the equivalent term **half step**). Select some pitches and octave numbers and view the corresponding frequency. In particular, try pitches that are an octave apart (e.g., A3, A4, and A5) and note how the frequency doubles as you go towards the higher-frequency side of the keyboard. Also try some single semitone intervals like A0 and A#0, and A7 and A#7.

This is an unsupported media type. To view, please see
http://cnx.org/content/m15439/latest/keyboard_pitches.llb

The frequency values themselves may seem rather mysterious. For example, "middle C" (C4) is 261.6 Hz. Why "261.6" exactly? Would "262" work just as well? Humans can actually perceive differences in the sub-Hz range, so 0.6 Hz is actually noticeable. Fortunately an elegantly simple equation exists to calculate any frequency you like. The screencast video of Figure 2.5 explains how to derive this equation that you can use in your own music synthesis algorithms. Watch the video, then try the exercises to confirm that you understand how to use the equation.

Image not finished

Figure 2.5: [video] Derivation of equation to convert semitones to frequencies

Exercise 2.1

What is the frequency seven semitones above concert A (440 Hz)?

(Solution on p. 31.)

⁸"Consonance and Dissonance" <<http://cnx.org/content/m11953/latest/>>

⁹"Harmony" <<http://cnx.org/content/m11654/latest/>>

¹⁰"Tuning Systems" <<http://cnx.org/content/m11639/latest/>>

Exercise 2.2*(Solution on p. 31.)*

What is the frequency six semitones below concert A (440 Hz)?

Exercise 2.3*(Solution on p. 31.)*

1 kHz is approximately how many semitones away from concert A (440 Hz)? Hint: $\log_2(x) = \frac{\log_a(x)}{\log_a(2)}$. In other words, the base-2 log of a value can be calculated using another base (your calculator has log base 10 and natural log).

2.2 [mini-project] Musical intervals and the equal-tempered scale¹¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ¹² module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 2.2

2.2.1 Overview

In this mini-project you will learn about musical intervals, and also discover the reason behind the choice of frequencies for the **equal-tempered** musical scale.

Spend some time familiarizing yourself with the piano keyboard below. Enter the pitch (letter) and its octave number to display the corresponding frequency. For example, **middle C** is C4, and C2 is two octaves below middle C. The frequency 440 Hz is an international standard frequency called **concert A**, and is denoted A4. Concert A is **exactly** 440 cycles per second, by definition.

The black keys are called **sharps** and are signified by the hash symbol #. For example, G#1 indicates the sharp of G1, and is located to the right of G1.

This is an unsupported media type. To view, please see
http://cnx.org/content/m15440/latest/keyboard_pitches.llb

Try the following exercises to make sure that you can properly interpret the keyboard:

Exercise 2.4*(Solution on p. 31.)*

What is the frequency of the leftmost black key?

Exercise 2.5*(Solution on p. 31.)*

What is the name and frequency of the white key to the immediate left of C7?

Exercise 2.6*(Solution on p. 31.)*

What is the name of the key that has a frequency of 370.0 Hz?

¹¹This content is available online at <<http://cnx.org/content/m15440/1.1/>>.

¹²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

2.2.2 Deliverables

1. Completed mini-project worksheet¹³
2. Hardcopy of your LabVIEW VI from Part 4 (block diagram **and** front panel)

2.2.3 Part 1

One aspect of the design of any scale is to allow a melody to be transposed to different keys (e.g., made higher or lower in pitch) while still sounding "the same." For example, you can sing the song "Twinkle, Twinkle Little Star" using many different starting pitches (or frequencies), but everyone can recognize that the melody is the same.



Download and run `tone_player.vi`¹⁴, a VI that accepts a vector of frequencies (in Hz) and plays them as a sequence of notes, each with a duration that you can adjust. Listen to the five-note sequence given by the frequencies 400, 450, 500, 533, and 600 Hz (it should sound like the first five notes of "Do-Re-Mi").

Now, transpose this melody to a lower initial pitch by subtracting a constant 200 Hz from each pitch; write the frequencies on your mini-project worksheet¹⁵:



Modify `tone_player.vi`¹⁶ by inserting an additional front-panel control so that you can add a constant offset to the array of frequencies. Be sure that you keep the "Actual Frequencies" indicator so that you always know to which frequencies you are listening.

Set the offset to -200Hz, and listen to the transposed melody. How does the transposed version compare to the original? Does it sound like the same melody? Enter your response on the worksheet:

Transpose the original melody to a higher initial pitch by adding 200 Hz to each pitch; write the frequencies on your worksheet:

Set the offset to 200Hz, and listen to the transposed melody. How does the transposed version compare to the original? How does it compare to the version that was transposed to a lower frequency? Enter your response on the worksheet:

Draw a conclusion: Is a **constant** frequency offset a good way to transpose a melody?

2.2.4 Part 2

In music theory, an **interval** is a standard distance between two pitches. For example, if you play middle C, and then the G above that, you have played a **perfect fifth**. If you start with an F#, then a perfect fifth above that is a C#. The first note you play is called the **fundamental**.

Refer back to the piano keyboard diagram at the top of this page. Each step to an adjacent key is called a **half step** (also known as a **semitone**).

If you play middle C (C4 on the diagram), how many half steps up do you need to go in order to play a perfect fifth interval? Enter answer on your worksheet:

If you begin on A4, which note is a perfect fifth above? Enter answer on your worksheet:

More intervals are listed below; the musical mnemonic may be helpful to hear the interval in your mind:

- Minor 2nd - one half step above fundamental (shark theme from "Jaws" movie)
- Major 2nd - two half steps above fundamental ("Do-Re-Mi," first two notes)
- Major 3rd - four half steps ("Kumbaya", first two notes of phrase)
- Perfect 4th - five half steps ("Here Comes the Bride")

¹³http://cnx.org/content/m15440/latest/ams_MP-intervals-worksheet.pdf

¹⁴http://cnx.org/content/m15440/latest/tone_player.vi

¹⁵http://cnx.org/content/m15440/latest/ams_MP-intervals-worksheet.pdf

¹⁶http://cnx.org/content/m15440/latest/tone_player.vi

- Perfect 5th - seven half steps ("Twinkle, twinkle, little star", first two notes)
- Major 6th - nine half steps ("My Bonnie Lies Over the Ocean," first two notes)
- Major 7th - eleven half steps ("There's a Place for Us" from West Side Story, first two notes)
- Octave - twelve half steps ("Somewhere Over the Rainbow," first two notes)

Listen to each of these intervals by entering the frequencies from the keyboard diagram. Remember to set your offset to zero. Also, you can silence a note by entering zero frequency. For example, if you want to hear a perfect 6th interval beginning at B3, you should use the frequencies 246.9 Hz and 415.3 Hz (G#4).

2.2.5 Part 3

Use C4 as the fundamental. Enter its frequency on your worksheet:

What is the frequency of a major 3rd above the fundamental? Enter its frequency on your worksheet:

What is the frequency **ratio** of the interval? Express your result in the form "a : 1", where "a" corresponds to the higher of the two frequencies. Enter the ratio on your worksheet:

Repeat the previous three questions using C5 as the fundamental (remember, C5 is one octave above C4). Enter the three values on your worksheet:

Try this again using A#2 as the fundamental; enter the three values on your worksheet:

Try this again using several different fundamental pitches for another type of interval.

Now, draw a conclusion: Based on what you have experienced about musical intervals so far, can you develop at least part of an explanation for why the frequencies have been selected as they have? Enter your comments on the worksheet:

2.2.6 Part 4

A variety of scales or tuning systems have been devised for musical instruments, some dating back several millennia. Scales include **Pythagorean tuning**, **just-tempered**, **mean-tempered**, **well-tempered**, (have you heard of Bach's "Well-Tempered Clavichord"?), and **equal-tempered**. For example, a **just-tempered** scale uses the following ratios of whole numbers for the intervals:

- Major 2nd, $9:8 = 1.125:1$
- Major 3rd, $5:4 = ______ : 1$
- Perfect 4th, $4:3 = ______ : 1$
- Perfect 5th, $3:2 = ______ : 1$
- Major 6th, $5:3 = ______ : 1$
- Major 7th, $15:8 = ______ : 1$
- Octave, $2:1 = ______ : 1$

Complete the table above to show each interval as a ratio of the form "a : 1"; enter these ratios on your worksheet:

Modify your VI so that you can enter a single fundamental frequency (in Hz) and an array of interval ratios to play. Be sure to keep the "Actual Frequencies" indicator so that you always know to what frequencies you are listening!

Listen to the scale formed by the following sequence of ratios, and use A4 (440 Hz) as the fundamental: $1, 9/8, 5/4, 4/3, 3/2, 5/3, 15/8, 2$. Comment on how well this scale sounds to you (enter your comments on your worksheet):

Transpose the same scale to G4 as the fundamental, and then F4 as the fundamental. Comment on well this scale transposes to different keys (the differences may be rather subtle); enter your comments on the worksheet:

2.2.7 Part 5

The frequencies on the keyboard diagram above show the piano tuned using the **equal-tempered** scale. An equal-tempered scale sacrifices the pure whole number ratios scheme for intervals, but offers the advantage that a melody transposed to any other key will sound the same. Thus, an equal-tempered scale is a "global compromise" – a given melody will be the same level of out of tune no matter which key is used for the fundamental. The other scales mentioned above will cause a given melody to sound quite nice in some keys, and quite out of tune in other keys.

Derive a mathematical function to calculate the frequencies used by the equal-tempered scale, i.e., given a fundamental frequency and a semitone offset, calculate the frequency. For example, when your formula is presented with the frequency 440 Hz and an offset of 2 (i.e., two semitones above concert A), it should return 493.9 Hz. Be sure to show your complete derivation **process** on your worksheet, and not simply the end result.

Hints:

- Your function should include a fundamental frequency "f" in Hz.
- Your function should include a way to calculate the interval selected by the number of semitones (or half steps) above or below the fundamental frequency.
- Your function should double the frequency when you enter 12 semitones above the fundamental (what should it do when you enter 12 semitones below the fundamental?).

Solutions to Exercises in Chapter 2

Solution to Exercise 2.1 (p. 26)

659.3 Hz ($n=7$)

Solution to Exercise 2.2 (p. 27)

311.1 Hz ($n=6$)

Solution to Exercise 2.3 (p. 27)

14

Solution to Exercise 2.4 (p. 27)

29.14 Hz

Solution to Exercise 2.5 (p. 27)

B6, 1976 Hz

Solution to Exercise 2.6 (p. 27)

F#4

Chapter 3

Analog Synthesis and Modular Synthesizers

3.1 Analog Synthesis Modules¹

3.1.1 Introduction

Analog synthesizers dominated music synthesis technology throughout all but the last 15 years of the 20th century. Early synthesizers were based on vacuum tubes or other electro-mechanical devices, and transistor technology entered the music scene during the early 1960s. Analog synthesizers produce sound waveforms as **continuous** voltages. Oscillators produce basic waveforms such as sinusoids, square waves, and triangle waves, much like a function generator in the electronics laboratory. These waveforms are shaped by time-varying amplifiers to emulate the characteristics of physical instruments, e.g., loud at the beginning transient of a note, softer during the sustained portion of the note.

You probably know that synthesizers defined many of the pop music styles of the 1970s. Watch (and listen!) to the screencast video in Figure 3.1 to learn more about some of the common synthesizer techniques. A real-time graphical signal analyzer is used to visualize the sounds so that you can better understand what you hear.

Image not finished

Figure 3.1: [video] Examples of analog synthesizer sounds, including visualization of waveform and frequency spectra

The history of electronic synthesizers is really fascinating. In particular, the following sites form an excellent starting point:

- SynthMuseum.com²
- EMF Institute³ – Follow the link for "The Big Timeline"
- 120 Years of Electronic Music⁴ – Look for the entries under 1960

¹This content is available online at <<http://cnx.org/content/m15442/1.2/>>.

²<http://www.synthmuseum.com>

³<http://emfinstitute.emf.org/>

⁴http://www.obsolete.com/120_years

3.1.2 Analog Synthesizer Modules

Everything about an analog synthesizer is analog! For example, a keyboard-based synthesizer uses a **control voltage (CV)** to change the frequency of the oscillator; the oscillator is therefore called a **voltage-controlled oscillator** or **VCO**. The time-varying gain of an amplifier is also controlled by a CV, so the amplifier is called a **voltage-controlled amplifier** or **VCA**. The VCO and VCA are two of many types of synthesizer **modules** that can be interconnected (or **patched** together) in many different ways.

Take a look at the video in Figure 3.2 to find out why interconnected modules are called **patches**, and to learn how to put together a simple patch involving a VCO, VCA, **envelope generator**, and keyboard controller.

Image not finished

Figure 3.2: [video] Origins of the term "patch", and simple example of an analog synthesizer patch using a VCO, VCA, envelope generator, and keyboard

Analog synthesizer modules can be grouped into four categories: **sources**, **processors**, **envelope generators**, and **controllers**; each of these is discussed in detail in the following sections.

3.1.2.1 Sources

Signal sources include the **VCO** and the noise generator. View the video in Figure 3.3 to learn more, then quiz yourself to check your understanding.

Image not finished

Figure 3.3: [video] VCO and noise generator signal sources

Exercise 3.1

The amplitude of a VCO's waveform can be adjusted (true or false).

(Solution on p. 40.)

Exercise 3.2

How does a VCO interpret its control voltage to produce a desired frequency?

(Solution on p. 40.)

Exercise 3.3

Which VCO has the richest harmonic content?

(Solution on p. 40.)

3.1.2.2 Processors

Signal processors include the (**VCA**) and the **voltage-controlled filter (VCF)**. View the video in Figure 3.4 to learn more, then quiz yourself to check your understanding.

Image not finished

Figure 3.4: [video] VCA and VCF signal processors

Exercise 3.4

How does a VCA interpret its control voltage to produce a desired gain?

(Solution on p. 40.)

Exercise 3.5

What types of filter functions can be implemented by a VCF?

(Solution on p. 40.)

Exercise 3.6

What VCF filter parameters can be adjusted by control voltages?

(Solution on p. 40.)

3.1.2.3 Envelope Generators

An **envelope generator** creates a **CV** to operate other voltage-controlled modules such as the VCA and VCF. View the video in Figure 3.5 to learn more about envelope generators, in particular why they are usually called an **ADSR**.

Image not finished

Figure 3.5: [video] Envelope generators, especially the ADSR-style envelope generator

Exercise 3.7

What is the normal (un-triggered) output of an envelope generator?

(Solution on p. 40.)

Exercise 3.8

What does the acronym **ADSR** mean?

(Solution on p. 40.)

Exercise 3.9

Why is the exponential shape used for envelope generators?

(Solution on p. 40.)

3.1.2.4 Controllers

A **controller** creates a control voltage (CV) to operate other voltage-controlled modules such as the VCA and VCF. An **interactive controller** offers the musician direct and immediate control of the sound, such as a keyboard, knob, or slider. A **programmed controller** generates a control voltage in some pre-defined way, such as a **low-frequency oscillator (LFO)** to produce vibrato, and a sequencer to produce a repeating pattern of control voltages for the VCO. View the video in Figure 3.6 to learn more, and then quiz yourself to check your understanding.

Image not finished

Figure 3.6: [video] Controllers including keyboard, knobs and sliders, low-frequency oscillator (LFO), and sequencer

Exercise 3.10 *(Solution on p. 40.)*

Which types of output voltages does a keyboard controller produce?

Exercise 3.11 *(Solution on p. 40.)*

What is the **portamento** effect, and how is it produced?

Exercise 3.12 *(Solution on p. 40.)*

Which device would a keyboardist use to automatically play a repeating pattern of notes?

3.2 [mini-project] Compose a piece of music using analog synthesizer techniques⁵


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ⁶ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none"> • Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 3.1

3.2.1 Objective

This project will give you the opportunity to begin designing sounds in LabVIEW. You will create two subVIs: one to implement an ADSR-style envelope generator and the other to create a multi-voice sound source. You will then create a top-level application VI to render a simple musical composition as an audio file.

3.2.2 Prerequisite Modules

If you have not done so already, please study the prerequisite module Analog Synthesis Modules (Section 3.1). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing⁷, which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

⁵This content is available online at <http://cnx.org/content/m15443/1.2/>.

⁶"NI LabVIEW Getting Started FAQ" <http://cnx.org/content/m15428/latest/>

⁷*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
<http://cnx.org/content/col10440/latest/>

3.2.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

3.2.4 Part 1: ADSR Envelope

Create the subVI **ADSR.vi** to generate the ADSR-style envelope specified in Figure 3.7; recall that ADSR stands for "Attack Decay Sustain Release."

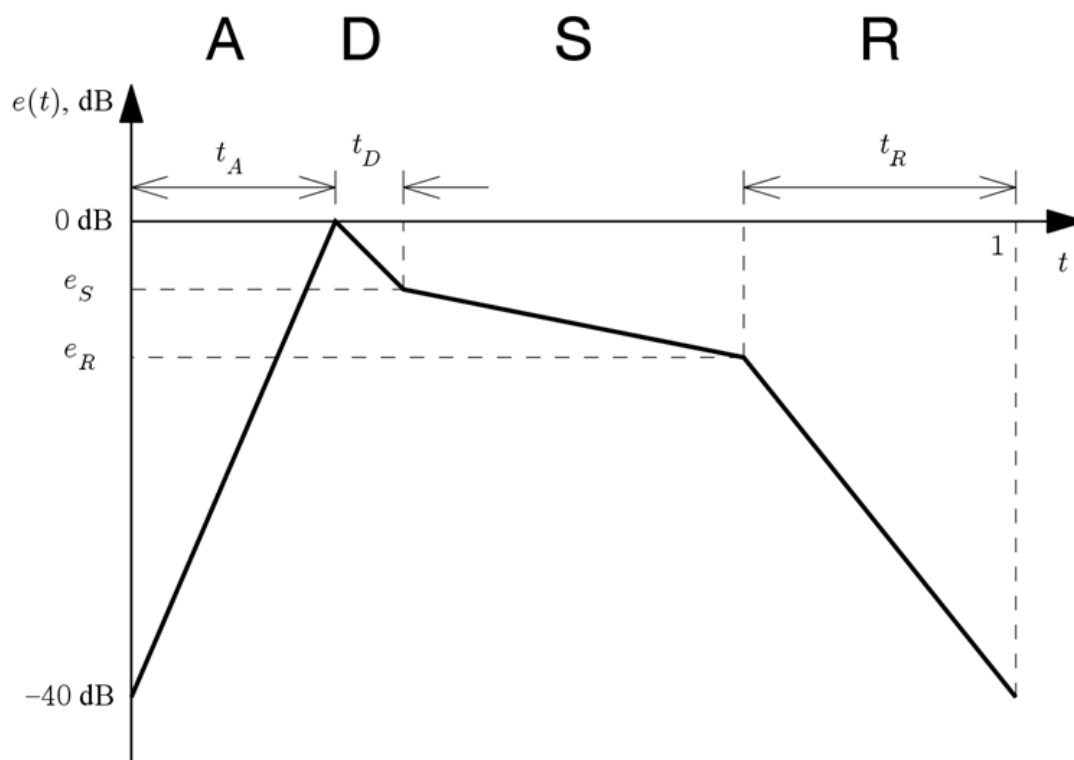


Figure 3.7: Specifications for the ADSR-style envelope

The envelope shape is defined over a normalized time interval of 0 to 1; the envelope stretches or compresses to match the actual duration requested. The three time intervals are likewise expressed in normalized units, so an attack duration (t_A) of 0.2 indicates that the attack time is 20% of the entire envelope duration. The envelope values range from 0 dB (full sound intensity) to -40 dB (close to silence). Once you have defined the envelope shape in terms of straight-line segments, you can create the envelope amplitude waveform by undoing the decibels operation; recall that an amplitude "A" expressed in dB is $20 \log_{10}(A)$.

The subVI requires the following controls as input parameters (units specified in brackets):

1. **duration [s]** - total duration of envelope

2. **fs [Hz]** - system-wide sampling frequency
3. **t values** - three-value array containing attack (A) interval, decay (D) interval, and release (R) interval. All three values are in the range 0 to 1, so the intervals are to be treated as fractions of the total duration of the envelope. For example, A=0.3 indicates the attack interval is 30% of the envelope's duration.
4. **e values** - two-value array containing the envelope value at the beginning of the sustain interval (eS in the diagram above) and the envelope value at the beginning of the release interval (eR).

The subVI requires the single indicator (output) **envelope**, an array with values in the range 0 to 1.

Show that your ADSR VI functions properly by plotting the envelope for at least two distinct cases. Plot both the dB form of the envelope as well as its non-dB form.

The following screencast video provides coding tips and other suggestions to help you develop your subVI.

Image not finished

Figure 3.8: [video] Coding tips and suggestions for **ADSR.vi**

3.2.5 Part 2: Sound Generator

Make a subVI called **SoundGen.vi** that combines a signal source with the ADSR envelope generator. Use a **case structure** with an **enumerated data type** control that allows you to choose one of several different "voices" or "instruments." The case structure then defines the ADSR input parameters to choose the type of signal source (or noise source for a percussive instrument).

The subVI requires the following controls as input parameters (units specified in brackets):

1. **duration [s]** - total duration of note
2. **fs [Hz]** - system-wide sampling frequency
3. **fref [Hz]** - reference frequency
4. **note** - note frequency as described by an integer value that denotes the number of semitone intervals from the reference frequency; **note** can be a negative, zero, or positive value
5. **instrument** - enumerated data type with values of your choice; your SoundGen.vi should have at least three different tone-type sounds and at least two different percussive sounds. Use a variety of envelope shapes.

If the discussion in Item 4 for the **note** control seems mysterious, review the module [INSERT NAME AND MODULE LINK] to learn how to calculate the frequency of a semitone interval using **equal temperament**.

Show that your sound generator VI functions properly by developing a top-level VI that plays a chromatic scale. Vary the duration of the notes. Plot the waveform to ensure that the envelope shape makes sense. Save your generated sound to a .wav file to be included with your deliverables.

The screencast video of Figure 3.9 offers coding tips and other suggestions to help you develop your subVI.

Image not finished

Figure 3.9: [video] Coding tips and suggestions for **SoundGen.vi**

3.2.6 Part 3: Multi-Voice Composition

Compose a simple piece of music using analog synthesis techniques of your choice. Better compositions will include variety, e.g., different envelope parameters at different times, multiple channels (chords), stereo, percussive sounds, and so on. Write a paragraph or two that describes your compositional technique. Better compositions are click-free, i.e., joining the notes together does not produce any pops or clicks.

Hints:

- You may wish to use a familiar melody (remember, you've got an equation that converts a note from the equal-tempered scale into frequency), or you may want to use a musical "experiment" based on an algorithm as the basis for your composition.
- Operate multiple instances of **SoundGen.vi** in parallel (each with a different instrument setting) and add their outputs together.
- Import multiple spreadsheets to define your note lists (use "File IO | Read Spreadsheet"). Spreadsheets are the easiest way to hand-edit the contents of arrays.
- Remember to normalize your finished output before saving it to a .wav file. Use **QuickScale** ("Signal Processing | Signal Operation | Quick Scale") for this purpose.
- View your finished composition as a waveform plot. Confirm that the envelope shapes make sense, that the waveform is symmetrical about the zero axis (it should not contain any constant offset), and that the waveform fits within the ± 1 range.

3.2.7 Alternative Part 3: Multi-Voice Composition Using MIDI File

The overall objectives of this alternative Part 3 are the same as described above, but the technical approach is different.



Go to MIDI JamSession (Section 4.6) to learn all about a LabVIEW application VI of the same name that reads a standard MIDI file (.mid format) and renders it to an audio file (.wav format) using "instrument" subVIs of your own design. You have already done all of the work necessary to create your own instruments, and it is a simple matter to place them inside a standard subVI wrapper (or "Virtual Musical Instrument," VMI) that MIDI JamSession can use.

Solutions to Exercises in Chapter 3

Solution to Exercise 3.1 (p. 34)

False

Solution to Exercise 3.2 (p. 34)

One octave per volt

Solution to Exercise 3.3 (p. 34)

Square wave has highest amplitude harmonics, but contains odd harmonics only; triangle wave has the most harmonics (even and odd)

Solution to Exercise 3.4 (p. 35)

Zero volts mean zero gain, one volt mean unit gain

Solution to Exercise 3.5 (p. 35)

Lowpass, highpass, bandpass, etc.

Solution to Exercise 3.6 (p. 35)

Corner (cutoff) frequency, bandwidth, resonance frequency

Solution to Exercise 3.7 (p. 35)

Zero

Solution to Exercise 3.8 (p. 35)

Attack - Decay - Sustain - Release

Solution to Exercise 3.9 (p. 35)

Easy to produce with RC-networks; matches behavior of real instruments

Solution to Exercise 3.10 (p. 36)

A control voltage to control the frequency of a VCO, and a gate voltage to control an envelope generator

Solution to Exercise 3.11 (p. 36)

A **portamento** is a continuous frequency transition from one note to the next; instead of producing a step-change in the control voltage connected to the VCO, the keyboard produces a continuously-varying voltage from the starting note to the ending note

Solution to Exercise 3.12 (p. 36)

Sequencer

Chapter 4

MIDI for Synthesis and Algorithm Control

4.1 MIDI Messages¹

4.1.1 Introduction

A **MIDI message** conveys information between MIDI-capable equipment. For example, a message could indicate that a note should begin sounding, or that a specific type of sound be selected, or that the position of a pitch-bender control has just changed. MIDI messages are typically three bytes long: a **status byte** followed by two **data bytes**. Status and data bytes are distinguished by the value of the MSB (most-significant bit); the MSB is set to a “1” for status bytes, and is cleared to a “0” for data bytes.

When the MIDI standard was developed in 1983, MIDI messages were designed for compatibility with serial communications through **UARTs** (universal asynchronous receiver-transmitters), the digital device “hiding” between the COM port on a desktop computer. In this way standard computer equipment could be interconnected to musical equipment including synthesizers, keyboards, sound modules, and drum machines.

The original MIDI electrical interconnection was designed for compatibility with standard audio cables terminated with **DIN-5** connectors. Look at the back of a typical synthesizer, and you will see three connectors that look like this:

¹This content is available online at <<http://cnx.org/content/m15049/1.2/>>.



Figure 4.1: Rear-panel MIDI connectors for IN, OUT, and THRU

The electrical connection is unidirectional. The **MIDI IN** connector accepts a signal transmitted from the **MIDI OUT** connector of another device. Many different devices can be cabled together in a **daisy chain**, like this:

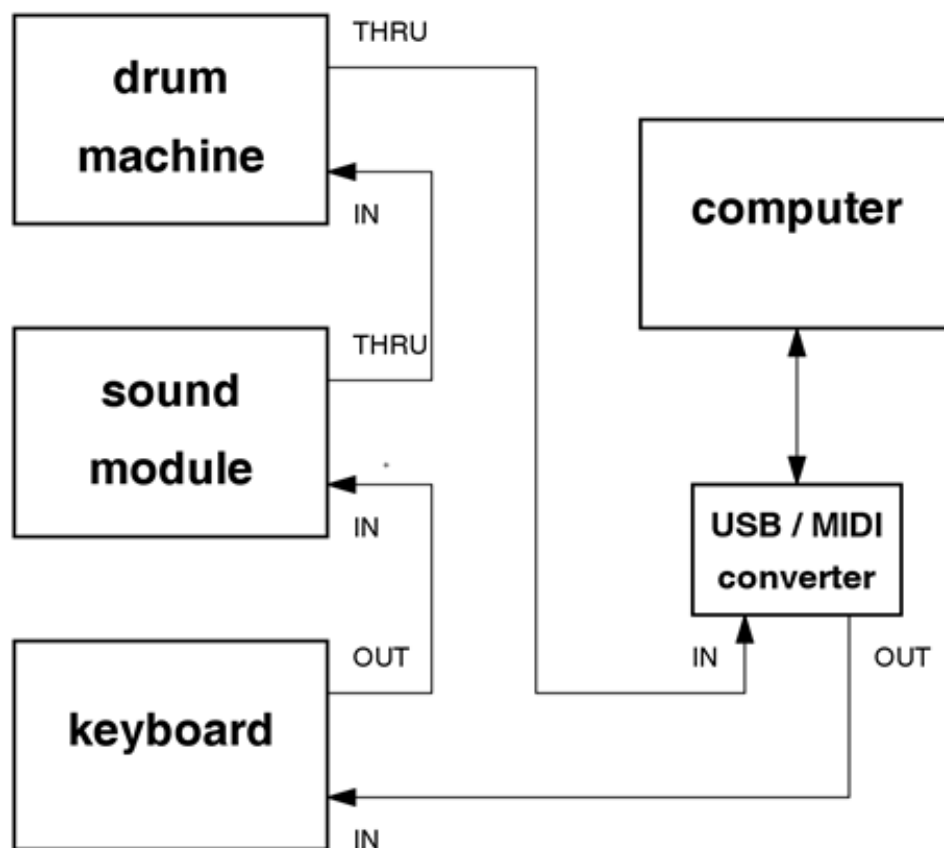


Figure 4.2: Typical daisy-chain connection scheme for several MIDI devices

Since it is not always desirable to have every single device process and re-transmit received MIDI messages, the **MIDI THRU** connector offers an electrically-identical version of the signal received on the **MIDI IN** connector. For example, the sound module and drum machine do not generate MIDI messages, so they can simply pass through the signals. However, the keyboard generates messages, so the **MIDI OUT** connector must be used.

MIDI messages associated with the actual musical performance – note on, note off, voice selection, and controller status – use the concept of a channel. For example, suppose that one synthesizer has been configured to associate “Channel 3” with the sound of a cello, and another synthesizer has been configured to associate “Channel 7” with the sound of a flute. When both of these interconnected synthesizers receive a “note on” message for Channel 3, only the first synthesizer will begin to sound a cello; the second synthesizer will ignore the message. In this way, many different devices can be interconnected, and configured to respond individually according to the channel number.

The MIDI standard for electrical interconnection specifies a fixed bit rate of 31.25 kbits/second. In the days of 8-MHz personal computers, 31.25 kbaud was considered quite fast. The rate is adequate to communicate performance information between several interconnected devices without noticeable delay, or **latency**. Today, however, MIDI messages are more often conveyed through **USB**. MIDI-to-USB converter boxes are available for older synthesizers that do not support USB directly.

4.1.2 Making a Sound: Note-On and Note-Off Events

As mentioned earlier, most MIDI messages are three bytes in length: a status byte followed by two data bytes. The status byte for the **Note-On event** is **1001nnnn** (0x9n in hexadecimal), where nnnn indicates the **channel number**. Since a four-bit value selects channel, there are a total of 16 channels available. The channels are called “Channel 1” (nnnn=0000) to “Channel 16” (nnnn=1111).

The first data byte indicates the **note number** to begin sounding, and the second data byte indicates the **velocity** at which the key was pressed. Since the MSB (most-significant bit) of a data byte is zero by definition, note number and velocity are each 7-bit values in the range 0 to 127.

Hexadecimal notation is commonly used to describe MIDI messages. For example, the three-byte message 0x93 0x5C 0x42 indicates a Note On event on Channel 4 for note number 92 with velocity 66.

The **Note-Off event** is similar; its status byte is **1000nnnn** (0x8n). For example, the three-byte message 0x83 0x5C 0x35 indicates that note number 92 on Channel 4 should cease sounding, and that the key release velocity was 53. Synthesizers and sound modules generally equate keypress velocity with amplitude; however, they are less likely to use the release velocity. A “Note On” event with zero velocity is equivalent to a “Note Off” event.

The following screencast video shows MIDI Note On and Note Off messages produced by the Roland XP-10 synthesizer and visualized using the MIDI OX software application². MIDI OX is a free MIDI utility that serves as a MIDI protocol analyzer. The video also shows how MIDI OX can play standard MIDI files such as instruments.mid³, a short multi-instrument composition that illustrates the concept of MIDI channels:

Image not finished

Figure 4.3: [video] Visualize MIDI note-on and note-off messages generated by the Roland XP-10 synthesizer

If you plan to start using MIDI OX right away, take a look at the following screencast that explains how to set up MIDI OX to work with a MIDI device such as a synthesizer, and how to set up your soundcard so that you can visualize the MIDI messages produced by a MIDI player such as Winamp or Windows Media Player. You will also need to install the support application called MIDI Yoke⁴, which serves as a virtual MIDI patch bay to interconnect all of the MIDI-related devices on your computer.

Image not finished

Figure 4.4: [video] Setting up MIDI-OX to view MIDI messages

²<http://www.midiox.com/>

³<http://instruments.mid/>

⁴<http://www.midiox.com/>

Image not finished

Figure 4.5: [video] Setting up your soundcard to work properly with MIDI-OX and a media player

4.1.3 Selecting a Voice: Program Change

The **Program Change** message selects which of 128 possible voices (also called sounds, tones, or patches) to associate with a particular channel. The status byte for the Program Change message is **1100nnnn** (0xCn), where nnnn indicates the channel number. Only one data byte called the **program number** follows the status byte for this message type. For example, the two-byte message 0xC7 0x5F directs the synthesizer to use program number 95 for all subsequent Note-On and Note-Off messages directed to Channel 8.

The original MIDI standard did not constrain equipment manufacturers in what sounds or voices to associate with program numbers. However, as it soon became apparent that defining standard voices for the 128 program numbers would make it easier for composers to distribute multi-voice (or **multitimbral**) compositions with the expectation of the listener hearing what the composer had intended. The **General MIDI (GM) standard** defines 128 basic voices to be associated with the 128 program numbers. Moreover, Channel 10 is always defined as a percussion instrument under the General MIDI standard. See the GM Level 1 Sound Set⁵ for a table of voices and percussion key map. Note that the seven-bit program number ranges from 0 to 127; the associated GM voice will range from 1 to 128. The following screencast video gives you a quick tour of the 128 General MIDI voices as they sound on my Roland XP-10 synthesizer.

Image not finished

Figure 4.6: [video] Tour of the General MIDI sound set as played by the Roland XP-10 synthesizer

The seven-bit program number can select from among a palette of 128 voices, yet musicians and composers find this value too limiting. Equipment manufacturers can easily provide thousands of voices thanks to the falling cost of solid-state memory.

With these facts in mind, it will be easier to understand why selecting a new voice for a channel can produce up to three distinct MIDI messages! The first two messages are Control Change messages, and the third message is a Program Change message. The status byte of a **Control Change** message is **1011nnnn** (0xBn), where nnnn indicates the channel number. The first data byte following the status byte indicates the **controller number**, and the second data byte indicates the **controller value**. The Control Change message will be described more fully in the following section. When the Control Change message is used select a voice, the controller number can be either 0x00 or 0x20 to indicate that the second data byte is either the most significant byte or the least significant byte, respectively, of a combined 14-bit **bank select value**. An example will clarify these concepts.

Suppose that the following three MIDI messages are received by a synthesizer (each message is destined for Channel 1):

```
0xB0 0x00 0x2F
0xB0 0x20 0x38
```

⁵<http://www.midi.org/about-midi/gm/gm1sound.shtml>

0xC0 0x0E

The first message is a Control Change with control number 0x00, so the upper seven bits of the bank select value are 010_1111. The second message is also a Control Change, but the 0x20 control number signifies that the lower seven bits of the bank select value are 011_1000. Thus the first two messages communicate a single 14-bit bank select value of 01_0111_1011_1000. Lastly, the Program Change message indicates that program number 16 is to be used. With all three messages, it is possible to select a unique voice from among 2 raised to the 21st power possible voices (2,097,152 possible voices).

This clever scheme permits newer equipment with more than 128 voices to properly respond to the rich palette of possible voices, while older equipment that only supports the basic 128 voices defined by the General MIDI standard will simply ignore the two bank select messages and respond to the Program Change message. Equipment manufacturers therefore will often define tone variations on a theme defined by the program number. For example, since program number 1 defines “Acoustic Grand Piano,” the bank select technique can be used to define many different types of acoustic grand pianos.

Image not finished

Figure 4.7: [video] Visualize “Bank Select” and “Program Change” MIDI messages that select different voices

4.1.4 Modifying a Sound: Control Change and Pitch Wheel Change

Musicians need to do more than simply turn a note on and off. To be expressive, a musician needs to be able to modify a sound already in progress. Synthesizers offer various knobs and sliders that can introduce pitch bends and vibrato, for example.

The **Control Change** message indicates that a control has just been changed. The status byte of a Control Change message is **1011nnnn** (0xBn), where nnnn indicates the channel number. The first data byte following the status byte indicates the **controller number**, and the second data byte indicates the **controller value**. Since the controller number is seven bits, 128 controller types are possible. Typical controllers include modulation wheel (0x01), main volume (0x07), sustain pedal (0x40), and general-purpose controllers (0x50 to 0x53).

Musicians use a synthesizer’s **pitch wheel** introduces to bend (temporarily raise or lower) the pitch of the entire keyboard. In order to provide more frequency resolution, the pitch wheel has a dedicated MIDI message; the two data bytes afford fourteen bits of resolution rather than the seven bits of resolution that are possible with a control change message.

The **Pitch Wheel Change** message indicates that the pitch wheel has just moved. The status byte of a Pitch Wheel Change message is **1110nnnn** (0xEn), where nnnn indicates the channel number. The first data byte following the status byte is the lower seven bits of the overall 14-bit position value, and the second data byte contains the upper seven bits of the position value. The nominal value is 0x2000 when the pitch wheel is centered (data byte #1 = 0x00, data byte #2 = 0x40). Moving the pitch wheel to its lower (leftmost limit) produces a value of 0x0000, while moving it to its rightmost limit produces a value of 0x3FFF (data byte #1 = 0x7F, data byte #2 = 0x7F). Note that the Pitch Wheel Change message simply indicates the **position** of the pitch wheel; the synthesizer may interpret this position in various ways depending on other settings. For example, my synthesizer defaults to a whole step down when the pitch wheel is moved to its left limit, however, it can be adjusted to shift by an entire octave for the same amount of pitch wheel movement.

The following screencast illustrates the MIDI messages generated by the slider controls and pitch wheel.

Image not finished

Figure 4.8: [video] Visualize "Control Change" MIDI messages produced by a general-purpose slider, and the "Pitch Wheel Change" messages produced by the pitch-bender

4.1.5 Transferring Data: System Exclusive Messages

The **System Exclusive (SysEx)** message provides a mechanism to transfer arbitrary blocks of information between devices. For example, the complete configuration of a synthesizer can be uploaded to a computer to be retrieved at a later time. The term “exclusive” indicates that that information pertains only to a particular vendor’s piece of equipment, and that the organization of the information is vendor-specific.

The transfer process begins with the **System Exclusive Start** message with status byte **11110000** (0xF0). The following data byte is the manufacturer ID. All following bytes must be data bytes in the sense that their most-significant bit is always zero. An arbitrary number of bytes may follow the manufacturer ID. The transfer process ends with the **System Exclusive End** message with status byte **11110111** (0xF7); no data bytes follow this status byte.

4.1.6 Other Messages

The messages discussed in this module are representative of what you will encounter when working with synthesizers and standard MIDI files, but are by no means a complete listing of all available messages. Refer to the MIDI Manufacturers Association⁶ for full details on the MIDI standard; see the specific tables noted in the last section of this module.

4.1.7 For Further Study

- MIDI Manufacturers Association (MMA)⁷
- MMA: Tutorial on MIDI and Music Synthesis⁸
- MMA: Table 1: Summary of MIDI messages⁹
- MMA: Table 2: Expanded Messages List (Status Bytes)¹⁰
- MMA: Table 3: Summary of Control Change Messages (Data Bytes)¹¹
- MMA: About General MIDI¹²
- MMA: General MIDI Level 1 Sound Set¹³
- Textbook: Francis Rumsey, *MIDI Systems and Control*, 2nd ed., Focal Press, 1994. See Tables 3.1 (MIDI messages summarized), 2.2 (MIDI note numbers related to the musical scale), and 2.4 (MIDI controller functions).

⁶<http://www.midi.org/>

⁷<http://www.midi.org/>

⁸<http://www.midi.org/about-midi/tutorial/tutorial.shtml>

⁹<http://www.midi.org/about-midi/table1.shtml>

¹⁰<http://www.midi.org/about-midi/table2.shtml>

¹¹<http://www.midi.org/about-midi/table3.shtml>

¹²<http://www.midi.org/about-midi/gm/gminfo.shtml>

¹³<http://www.midi.org/about-midi/gm/gmlsound.shtml>

4.2 Standard MIDI Files¹⁴

4.2.1 Introduction

In the 1970s, analog synthesizers were either played live by a keyboard musician or by a hardware device called a **sequencer**. The sequencer could be programmed with a pattern of notes and applied continuously as a loop to the keyboard. In 1983 the **MIDI (Musical Instrument Digital Interface)** standard was released, and MIDI-capable synthesizers could be connected to personal computers; the IBM PC and Apple Macintosh were among the first PCs to be connected to synthesizers. Software applications were developed to emulate the behavior of hardware sequencers, so naturally these programs were called “sequencers.”

A **sequencer application** records MIDI messages and measures the time intervals between each received message. When the song is played back, the sequencer transmits the MIDI messages to the synthesizer, with suitable delays to match the original measurements. Sequencers store related MIDI messages in **tracks**. For example, the musician can record a drum track, and then record a bass track while listening to the drum track, and continue in this fashion until a complete song is recorded with full instrumentation. The finished result is stored in a **standard MIDI file (SMF)**, normally named with the **.mid** suffix. In recent years sequencing software has merged with audio recording software to create a **digital audio workstation**, or **DAW**. Audio and MIDI tracks may be easily recorded, edited, and produced in one seamless environment.

Standard MIDI files can be played by a number of non-sequencer applications such as Winamp and Windows Media Player. In the absence of a synthesizer, these applications send the MIDI messages directly to the MIDI synthesizer included on the computer’s soundcard. With a sufficiently high-quality soundcard, a software synthesizer application can be used instead of conventional synthesizer hardware to produce high-quality music; a USB piano-style keyboard is the only additional piece of gear needed.

After completing this module you will understand the structure of a standard MIDI file, including fundamental concepts such as **files chunks**, **delta-times** in **variable-length format**, **running status**, and **meta-events**. See MIDI Messages (Section 4.1) to learn more about the types of messages that can be contained in a standard MIDI file.

4.2.2 High-Level Structure

A **standard MIDI file** contains two high-level types of information called **chunks**: a single **header chunk**, and one or more **track chunks**. The header chunk defines which of three possible file types is in force, the number of track chunks in the file, and information related to timing. The track chunk contains pairs of **delta times** and **events**; the delta time indicates the elapsed time between two Note-On messages, for example.

A chunk begins with the **chunk type**, a four-byte string whose value is either **MThd** for a header chunk or **MTrk** for a track chunk. The **chunk length** follows this string, and indicates the number of bytes remaining in the chunk. The chunk length is a 32-bit unsigned integer in **big-endian** format, i.e., the most-significant byte is first.

4.2.3 Header Chunk

The **header chunk** begins with the string **MThd**, and is followed by a fixed **chunk length** value of 6; as a 32-bit unsigned integer this is **0x00_00_00_06**.

The **file type** follows, and is a 16-bit unsigned integer that takes on one of three possible values: 0 (**0x00_00**) indicates single track data, 1 (**0x00_01**) indicates multi-track data which is vertically synchronous (i.e., they are parts of the same song), and 2 (**0x00_02**) indicates multi-track data with no implied timing relationship between tracks. Type 1 is by far the most common, where each track contains MIDI messages on a single channel. In this way tracks can be associated with a single instrument and recorded individually.

Next is the **number of tracks**, a 16-bit unsigned integer. The number of tracks can range from one (**0x00_01**) to 65,535 (**0xFF_FF**). In practice the number of tracks is typically about 20 or so.

¹⁴This content is available online at <<http://cnx.org/content/m15051/1.3/>>.

The last value in the header chunk is the **division**, a 16-bit unsigned integer that indicates the number of **ticks per quarter note**. The **tick** is a dimensionless unit of time, and is the smallest grain of time used to indicate the interval between events. A typical value of division is 120 (0x00_78).

4.2.4 Track Chunk

The **track chunk** begins with the string **MTrk**, and is followed by the **track length**, a 32-bit unsigned integer that indicates the number of bytes remaining in the track. In theory a track could be as long as 4 Gbytes!

The remainder of the track is composed of pairs of delta-times and events. A **delta-time** is in units of ticks, and indicates the time interval between events. An **event** is either a MIDI message or a **meta-event**. Meta-events are unique to the standard MIDI file, and indicate information such as track name, tempo, copyright notice, lyric text, and so on.

A delta time uses **variable-length format (VLF)**, and can be anywhere from one to four bytes in length. Short delta times require only one byte, and long delta times can require up to four bytes. Since short delta times tend to dominate the standard MIDI file (think of a chord hit that generates a burst of Note-On messages with very little time between events), most of the delta times are only one byte long and the overall file size is thus minimized. However, since very long delta times must be accommodated as well, the variable-length format can use up to four bytes to represent very long time intervals.

4.2.5 Variable-Length Format

Variable-length format (VLF) is used to represent delta times and the length of meta-events (to be described below). A numerical value represented in VLF requires from one to four bytes, depending on the size of the numerical value. Since the standard MIDI file is parsed one byte at a time, some type of scheme is required to let the parser know the length of the VLF number. A naïve approach would be to include an additional byte at the beginning of the VLF to indicate the number of bytes remaining in the value. However, this approach would mean that a delta time always requires a minimum of two bytes. Instead, the VLF uses the most-significant bit (MSB) as a flag to indicate whether more bytes follow.

For example, consider the unsigned integer $0x42 = 0b0100_0010$ (66 decimal), which can fit within seven bits. Since the MSB is clear (is zero), the parser will not seek any further bytes, and will conclude that the numerical value is $0x42$.

Now consider the slightly larger value $0x80 = 0b1000_0000$ (128 in decimal), which requires all eight bits of the byte. Since a VLF byte only has seven usable bits, this value must be encoded in two bytes, like this: $0x81\ 0x00$. Why? If we group the original value into two 7-bit groups, we have the two binary values $0b000_0001$ and $0b000_0000$. The MSB of the first group must be set to 1 to indicate that more bytes follow, so it becomes $0b1000_0001 = 0x81$. The MSB of the second group must be set to 0 to indicate that no bytes follow, so it becomes $0b0000_0000 = 0x00$.

Exercise 4.1

(Solution on p. 67.)

A delta time in VLF appears as two bytes, $0x820C$. What is the delta time expressed as a decimal value?

Exercise 4.2

(Solution on p. 67.)

A delta time in VLF appears as four bytes, $0xF3E8A975$. What is the delta time expressed as a hexadecimal value?

4.2.6 Running Status

Running status is another technique developed to minimize the size of a standard MIDI file. Consider a long sequence of Note-On and Note-Off messages (“long” meaning perhaps thousands of events). Eventually you would notice that storing the status byte for each and every note message would seem redundant and

wasteful. Instead, it would be more economical to store one complete Note-On message (a status byte followed by the note number byte and the velocity byte), and then from that point onwards store **only** the note number byte and velocity byte.

Clearly this is a simple solution for the application that creates the MIDI file, but what about the application that must read and parse the file? How can the parser know whether or not the status byte has been omitted from a message? Fortunately, the status byte can easily be distinguished from the data bytes by examination of the MSB (most-significant bit). When the parser expects a new message to start and finds that the byte has its MSB cleared (set to zero), the parser recovers the status information by using the information from the most-recent complete message. That is, the status continues to be the same as whatever was indicated by the most recent status byte received. In this way, a series of Note On messages can be conveyed by only two bytes per message instead of three.

But how do notes get turned off without storing a Note-Off message? Fortunately a Note-On message with zero velocity is equivalent to a Note-Off event! Thus, once the running status is established as “note on,” it is possible to turn notes on and off for as long as you like, with each event requiring only two bytes.

Any time another message type needs to be inserted (such as a Program Change), the running status changes to match the new type of message. Alternatively, when a different channel is required for a note message, a fresh Note-On status byte must be sent. For these reasons, most standard MIDI files are organized as Type 1 (multi-track, vertically synchronous), where each track corresponds to a different voice on its own channel. The Program Change message occurs at the beginning of the track to select the desired voice, a complete Note-On message is sent, and running status is used for the duration of the track to send Note-On and Note-Off messages as two-byte values.

4.2.7 Meta-Events

Meta-events provide a mechanism for file-related information to be represented, such as track name, tempo, copyright notice, lyric text, and so on. A meta-event begins with an 8-bit unsigned integer of value 0xFF. Note that the MSB (most-significant bit) is set, so a meta-event begins in the same way as a MIDI message status byte, whose MSB is also set. Next, the meta-event type is indicated by an 8-bit unsigned integer. After this, the number of bytes remaining in the meta-event is indicated by a numerical value in variable-length format (VLF); see an earlier section for full details). Lastly, the remainder of the meta-event information follows.

Some common meta-event types include 0x01 (**text event**), 0x02 (**copyright notice**), 0x03 (**track name**), 0x04 (**instrument name**), 0x05 (**lyric text**), and 0x7F (**sequencer-specific**). All of these meta-events can have an arbitrary length. Sequencer-specific is analogous to the **System-Exclusive** MIDI message, in that the data contained in the meta-event is arbitrary, and can normally be interpreted only by the sequencer application that created the standard MIDI file.

Tracks normally conclude with the **end-of-track** meta-event, whose type is 0x2F and whose length is zero. The end-of-track meta-event appears as the byte sequence 0xFF 0x2F 0x00 (meta-event, end-of-track type, zero length, with no data following).

The **set-tempo** meta-event (type 0x51) provides the value **microseconds per quarter note**, a 24-bit (3-byte) unsigned integer. This value in conjunction with the division value in the header chunk (division = ticks per quarter note) determines how to translate a delta time in ticks to a time in seconds. If the set-tempo meta-event is not included in the standard MIDI file, the value defaults to 500,000 microseconds per quarter note (or 0.5 seconds per quarter note).

Exercise: Given T (the value of a set-tempo meta-event) and D (the division value in the header chunk), determine an equation that can convert a delta time in ticks to a delta time in seconds.

4.2.8 For Further Study

- Description of MIDI Standard File Format¹⁵

¹⁵<http://www.4front-tech.com/pguide/midi/midi7.html>

- Standard MIDI Files 1.0¹⁶
- The USENET MIDI Primer¹⁷ by Bob McQueer
- Outline of the Standard MIDI File Structure¹⁸
- Textbook: Francis Rumsey, MIDI Systems and Control, 2nd ed., Focal Press, 1994. See Table 5.2 (A selection of common meta-event type identifiers)

4.3 Useful MIDI Software Utilities¹⁹

4.3.1 mf2t / t2mf

Standard MIDI files are binary files, and therefore cannot be read using a standard text editor. Piet van Oostrum has developed a companion pair of console applications called **mf2t** (MIDI File to Text) and **t2mf** (Text to MIDI File) that translate back and forth between the standard MIDI file and a human-readable version. You can more easily study the text-version MIDI file to see the messages, meta-events, and timing information. Also, you can edit the text-version, and then convert it back to the standard binary format.

Available at <http://www.midiox.com>²⁰ (scroll towards bottom of the page).

Image not finished

Figure 4.9: [video] Tour of the mf2t / t2mf MIDI-to-text conversion utilities

4.3.2 XVI32

Created by Christian Maas, **XVI32** is an excellent binary file editor (or “hex editor”) when you need to view a standard MIDI file directly. XVI32 can also modify the file by tweaking individual byte values, or by inserting and deleting ranges of values. The editor also includes tools to interpret data values, i.e., select and range of 8 bytes and interpret as an IEEE double-precision floating point value.

Available at <http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm>²¹

Image not finished

Figure 4.10: [video] Tour of the XVI32 hex editor for viewing standard MIDI files

¹⁶<http://jedi.ks.uiuc.edu/~johns/links/music/midifile.html>

¹⁷<http://www.sfu.ca/sca/Manuals/247/midi/UseNet-MidiPrimer.html>

¹⁸<http://www.ccarh.org/courses/253/handout/smf/>

¹⁹This content is available online at <http://cnx.org/content/m14879/1.2/>.

²⁰<http://www.midiox.com/>

²¹<http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm>

4.3.3 MIDI-OX

MIDI-OX is a wonderful utility developed by Jamie O'Connell and Jerry Jorgenrud. MIDI-OX serves as a MIDI protocol analyzer by displaying MIDI data streams in real-time. MIDI-OX can also filter MIDI streams (remove selected message types) and map MIDI streams (change selected message types according to some rule). MIDI-OX includes other useful features: you can use your computer's keyboard to generate note events (you can even hold down multiple keys to play chords!), you can play standard MIDI files (.mid files), and you can capture a MIDI data stream and save it to a file.

In order to make full use of MIDI-OX, you will also want to install the **MIDI-Yoke** driver. MIDI-Yoke works like a virtual **MIDI patch bay**, a device that connects MIDI inputs and outputs together. For example, you can connect the output of a MIDI sequencer application to MIDI-OX to view the MIDI messages, and then send the message to the MIDI synthesizer on your soundcard. When you have many MIDI-capable devices connected to your computer, MIDI-OX and MIDI-Yoke make it easy to quickly reconfigure the virtual connections without changing any external cables.

Available at <http://www.midiox.com>²²

Image not finished

Figure 4.11: [video] Tour of the MIDI-OX utility

4.3.4 JAZZ++

JAZZ++ is a **MIDI sequencer** created by Andreas Voss and Per Sigmond. JAZZ++ serves as a multi-track recorder and editor for MIDI-capable instruments, and also supports audio tracks. A musician can create a composition with full instrumentation by recording tracks one at a time. Performances such as a piano solo recorded by the sequencer can easily be edited to correct any mistakes. JAZZ++ is also a great way to graphically visualize the contents of a standard MIDI file.

Available at <http://www.jazzware.com/zope>²³

Image not finished

Figure 4.12: [video] Tour of the JAZZ++ MIDI sequencer application

4.4 [mini-project] Parse and analyze a standard MIDI file²⁴

4.4.1 Objective

MIDI files consist of three types of data: (1) MIDI events, (2) timing information, and (3) file structure information (file header and track headers). Your task in this mini-project is to parse a MIDI file by hand by

²²<http://www.midiox.com/>

²³<http://www.jazzware.com/zope>

²⁴This content is available online at <http://cnx.org/content/m15052/1.2/>.

examining the individual bytes of the file. Once you can successfully parse a file, you will be well-positioned to write your own software applications that can create standard MIDI files directly, or even to read the file directly (a challenge!).

If you have not done so already, please study the pre-requisite modules, MIDI Messages (Section 4.1) and Standard MIDI Files (Section 4.2). You will need to refer to both of these modules in order to complete this activity.

4.4.2 Part 1: Parse the File

The file to be parsed is available here: `parse_me.mid`²⁵ (MIDI files are binary files, so right-click and choose “Save As” to download the file). If you wish, you can use a hexadecimal editor such as Xvi32²⁶ to display the bytes, otherwise you can refer to the Xvi32 screenshot below:

²⁵http://cnx.org/content/m15052/latest/parse_me.mid

²⁶<http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm>

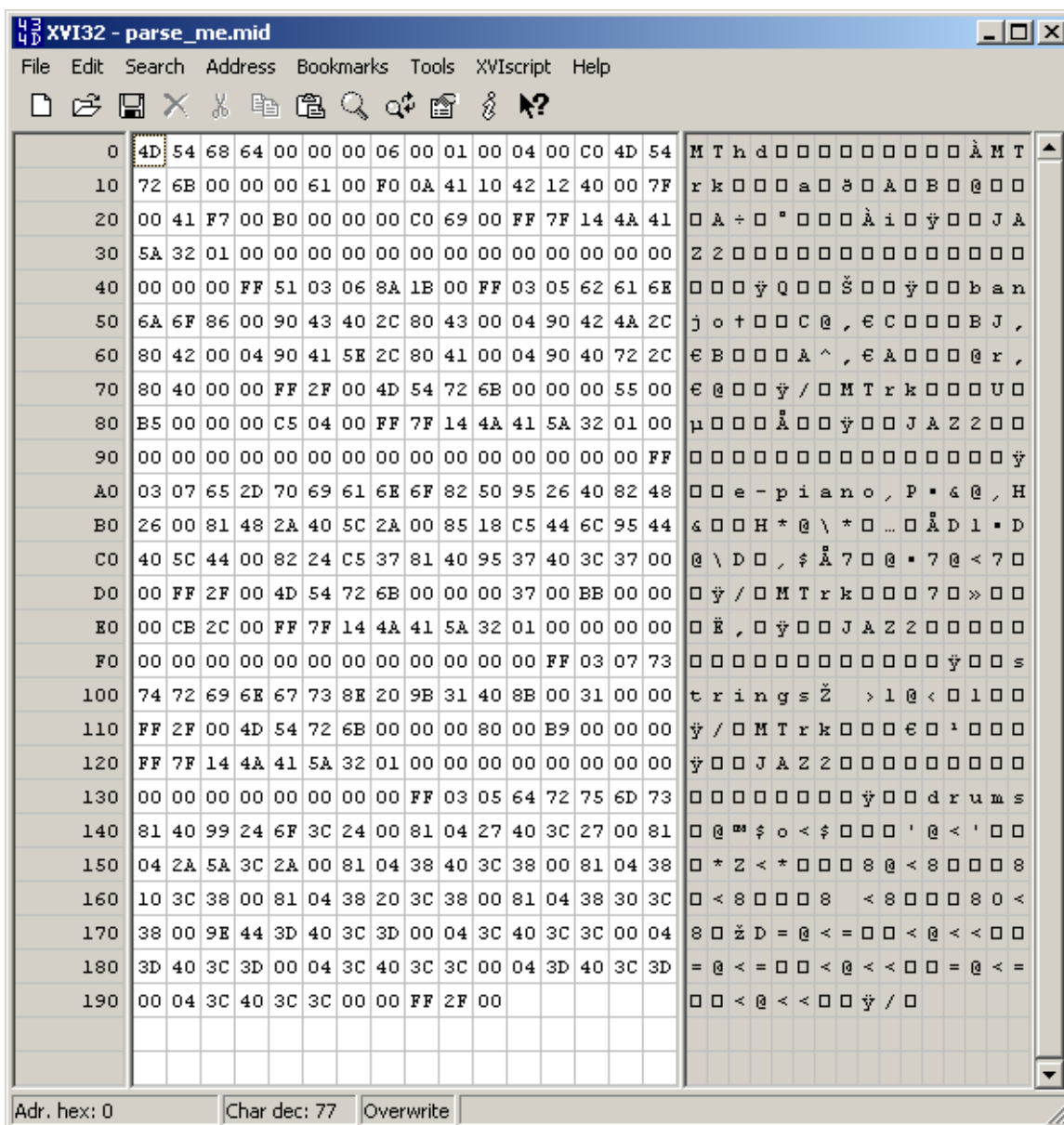


Figure 4.13: Hexadecimal listing of the 'parse_me.mid' file

The byte position (address) is shown on the far left panel, the hexadecimal representation of each byte in the file is shown in the central panel, and the ASCII representation of each byte is shown in the right panel. Most of the ASCII characters look like gibberish, but important landmarks such as the track chunk ID (MTrk) are easily visible.

Interpret each part of the file, and list your results in tabular format, like this (use hexadecimal notation for the “Starting Address” column):

Starting Address	Description	Value
0	Header chunk ID	
4	Header chunk length	6
8	File format type	1
A	Number of tracks	4
etc.		

Remember, a byte value of 0xFF signifies the beginning of a meta-event. Also, remember the concept of **running status**, so be on the lookout for MIDI messages whose first byte is NOT a status byte, i.e., whose MSB is clear. Take a look at the following video if you need some assistance:

Image not finished

Figure 4.14: [video] Walk through the MIDI file "parse_me.mid" at a high level, then show some parsing examples at the beginning of the file.

4.4.3 Part 2: Analyze Your Results

Determine the following information for the MIDI file:

- File format (Type 0, 1, or 2)
- Number of tracks
- Number of distinct channels used
- Number of distinct voices (instruments) used (look for Program Change messages)
- Number of ticks per quarter note
- Shortest two non-zero delta times, reported both in ticks and in microseconds (hint: the timing information you need here is derived from information in two distinct places)
- Top two longest delta times, both in ticks and in microseconds
- Minimum non-zero note-on velocity, and its channel and note number
- Maximum note-on velocity, and its channel and note number
- Total number of meta-events

If you have not done so already, listen to parse_me.mid²⁷ (click the link to launch your default music player application). How do your parsing and analysis results compare to what you hear?

4.5 [mini-project] Create standard MIDI files with LabVIEW²⁸

4.5.1 Required Background

If you have not done so already, please study the pre-requisite modules, MIDI Messages (Section 4.1) and Standard MIDI Files (Section 4.2). You will need to refer to both of these modules in order to complete this activity. Also, you will find it helpful to have already worked through the mini-project MIDI File Parsing (Section 4.4).

²⁷http://cnx.org/content/m15052/latest/parse_me.mid

²⁸This content is available online at <<http://cnx.org/content/m15054/1.2/>>.

4.5.2 Introduction

In this project you will create your own LabVIEW application that can produce a standard MIDI file. You will first develop a library of six subVIs that can be combined into a top-level VI that operates just like **MIDI_UpDown.vi** below (click the “Run” button (right-pointing arrow) to create the MIDI file, then double-click on the MIDI file to hear it played by your soundcard):

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/MIDI_UpDown.llb

MIDI_UpDown.vi produces a two-track MIDI file, with one track an ascending chromatic scale and the other a descending chromatic scale. You can select the voice for each track by choosing a tone number in the range 1 to 128. You can also select the duration of each note (“on time”) and space between the notes (“off time”).

Remember, **MIDI_UpDown.vi** is simply a demonstration of a top-level VI constructed from the subVIs that you will make. Once you have constructed your library of subVIs, you will be able to use them in a wide array of projects; here are some ideas:

- simulated wind chime: with an appropriate voice selection (see the General MIDI Level 1 Sound Set²⁹ to choose a bell-like sound) and random number generator for delta times
- bouncing ping-pong ball: write a mathematical formula to model the time between bounces and the intensity of bounces
- custom ring-tone generator for a cell phone

These are just a few ideas – be creative! Remember to take advantage of your ability to control the sound type, note-on velocity, pitch bend, etc.

4.5.3 Tour of the Top-Level Block Diagram

Click the image below to take a quick tour of the top-level block diagram of **MIDI_UpDown.vi**. The role of each subVI will be discussed in some detail, and you will have a better idea of the design requirements for each of the subVIs you will create.

Image not finished

Figure 4.15: [video] Tour of the top-level block diagram of **MIDI_UpDown.vi**

4.5.4 SubVI Library

You will create six subVIs in this part of the project. **Develop them in the exact order presented!** Also, make sure you **test and debug each subVI** before moving on to the next. Many of the concepts and techniques you learn at the beginning carry forward to the more sophisticated subVIs you develop toward the end.

The requirements for the subVIs are detailed in the following sections. **Input Requirements** specify the name of the front panel control, its data type, and default value, if needed). **Output Requirements**

²⁹<http://www.midi.org/about-midi/gm/gm1sound.shtml>

are similar, but refer to the front panel indicators. **Behavior Requirements** describe in broad terms the nature of the block diagram you need to design and build.

An interactive front panel is provided for each of the subVIs as an aid to your development and testing. By running the subVI with test values, you can better understand the behavioral requirements. Also, you can compare your finished result with the “gold standard,” so to speak.

Screencast videos offer coding tips relevant to each subVI. The videos assume that you are developing the modules in the order presented.

All right, time to get to work!

NOTE: The file name convention adopted for this project will help you to better organize your work. Use the prefix “midi_” for the subVIs, and “MIDI_” for top-level applications that use the subVIs. This way all related subVIs will be grouped together when you display the files in the folder.

4.5.5 midi_PutBytes.vi

midi_PutBytes.vi accepts a string and writes it to a file. If the file already exists, the user should be prompted before overwriting the file.

4.5.5.1 Input Requirements

- file path (file path type)
- string (string type)

4.5.5.2 Output Requirements

- error out (error cluster)

4.5.5.3 Behavior Requirements

- Create a new file or replace an existing file
- If replacing a file, prompt the user beforehand to confirm
- Write the string to the file, then close the file
- Connect the file-related subVIs to **error out**

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_PutBytes.llb

4.5.5.4 Coding Tips

Watch the screencast video to learn how to use the built-in subVIs **Open/Create/Replace File**, **Write to Binary File**, and **Close File**. Refer to the module Creating a subVI in LabVIEW (Section 1.4) to learn how to create a **subVI**.

Image not finished

Figure 4.16: [video] Learn how to write a string to a binary file

4.5.6 midi_AttachHeader.vi

Once all of the track strings have been created, **midi_AttachHeader.vi** will attach a header chunk to the beginning of the string to make a complete string prior to writing to a file. The header chunk requires the MIDI file type, number of tracks, and division (ticks per quarter note).

4.5.6.1 Input Requirements

- string in (string type)
- type (16-bit unsigned integer type; defaults to 1)
- number of tracks (16-bit unsigned integer type; defaults to 1)
- ticks per quote (16-bit unsigned integer type; defaults to 120)

4.5.6.2 Output Requirements

- string out (string type)

4.5.6.3 Behavior Requirements

- Create a header chunk ID sub-string (MThd)
- Create a sub-string for chunk length (always 0x00_00_00_06)
- Create sub-strings for the three unsigned integers applied as inputs
- Assemble the sub-strings into a string in order as chunk ID, chunk length, type, number of tracks, and division
- Append the inbound string to the header and output this result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_AttachHeader.llb

4.5.6.4 Coding Tips

Watch the screencast video to learn how to use the **Concatenate Strings** node to join substrings together into a single string. You will also learn how use the nodes **To Variant** and **Variant to Flattened String** to convert a numerical value into its representation as a sequence of bytes in a string.

Image not finished

Figure 4.17: [video] Learn how to concatenate strings and convert numerical values to a sequence of bytes

4.5.7 midi_FinishTrack.vi

Once all of the delta-time/event pairs have been assembled into a string, **midi_FinishTrack.vi** will attach a track chunk header to the beginning of the string and append an end-of-track meta-event at the end of the string. The resulting string will represent a complete track chunk.

4.5.7.1 Input Requirements

- string in (string type)
- delta-time / event pairs (string type)

4.5.7.2 Output Requirements

- string out (string type)

4.5.7.3 Behavior Requirements

- Create a track chunk ID sub-string (MTrk)
- Create a sub-string for a zero delta-time followed by an end-of-track meta-event
- Determine the total number of bytes in the track, and create a four-byte substring that represents this value
- Assemble the sub-strings into a string in order as chunk ID, chunk length, inbound string, and zero delta-time, and end-of-track meta-event, and output this result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_FinishTrack.llb

4.5.7.4 Coding Tips

Watch the screencast video to learn how to use the nodes **String Length** and **To Unsigned Long Integer** to determine the number of bytes in the track.

Image not finished

Figure 4.18: [video] Learn how to determine the length of string

4.5.8 midi_ToVLF.vi

midi_ToVLF.vi accepts a 32-bit unsigned integer and produces an output string that is anywhere from one to four bytes in length (recall that VLF = variable length format). You may find this subVI to be one of the more challenging to implement! Review the module Standard MIDI Files (Section 4.2) to learn about variable-length format.

4.5.8.1 Input Requirements

- x (32-bit unsigned integer)
- string in (string type)

4.5.8.2 Output Requirements

- string out (string type)

4.5.8.3 Behavior Requirements

- Accept a numerical value to be converted into a sub-string one to four bytes in length in variable-length format
- Append the sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_ToVLF.llb

4.5.8.4 Coding Tips

Watch the screencast video to learn how to convert a numerical value to and from the **Boolean Array** data type, an easy way to work with values at the bit level.

Image not finished

Figure 4.19: [video] Learn how to convert a numerical value to a Boolean array in order to work at the individual bit level

4.5.9 midi_MakeDtEvent.vi

midi_MakeDtEvent.vi creates a delta-time / event pair. The subVI accepts a delta-time in ticks, a MIDI message selector, the channel number, and two data values for the MIDI message. The delta-time is converted into variable-length format, and the (typically) three-byte MIDI message is created. Both of these values are appended to the inbound string to produce the output string. Review the module MIDI Messages (Section 4.1) to learn more.

4.5.9.1 Input Requirements

- string in (string type)
- delta time (32-bit unsigned integer; defaults to 0)
- event (enumerated data type with values Note Off, Note On, Control Change, Program Change, Pitch Wheel; defaults to Note Off)
- channel (8-bit unsigned integer; defaults to 1)
- data 1 (8-bit unsigned integer; defaults to 0)
- data 2 (8-bit unsigned integer; defaults to 0)

4.5.9.2 Output Requirements

- string out (string type)

4.5.9.3 Behavior Requirements

- Convert the delta time value to VLF format using the **midi_ToVLF.vi** subVI you created in a previous step
- Subtract 1 from the inbound channel number (this way you can refer to channel numbers by their standard numbers (in the range 1 to 16) outside the subVI).
- Create a MIDI message status byte using the channel number and event selector
- Finish the MIDI message by appending the appropriate byte values; depending on the MIDI message you need to create, you may use both ‘data 1’ and ‘data 2’, or just ‘data 1’ (Program Change message), or you may need to modify the incoming data value slightly (for example, outside the subVI it is more convenient to refer to the tone number for a Program Change message as a value between 1 and 128)
- Append the delta-time sub-string and the MIDI message sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_MakeDtEvent.llb

4.5.9.4 Coding Tips

Watch the screencast video to learn how to assemble a byte at the bit level, and also how to set up the enumerated data type for a case structure.

Image not finished

Figure 4.20: [video] Learn how to assemble a byte at the bit level, and learn how to set up an enumerated data type for a case structure

4.5.10 midi_MakeDtMetaEvent.vi

midi_MakeDtMeta.vi creates a delta-time / meta-event pair. The subVI accepts a delta-time in ticks, a meta-event selector, text string, and tempo value (only certain meta-events require the last two inputs). The delta-time is converted into variable-length format, and the meta-event is created. Both of these values are appended to the inbound string to produce the output string. Review the module Standard MIDI Files (Section 4.2) to learn about meta-events.

4.5.10.1 Input Requirements

- string in (string type)
- delta time (32-bit unsigned integer; defaults to 0)
- event (enumerated data type with values Text, Copyright Notice Text, Track Name, Instrument Name, Lyric Text, Marker Text, Cue Point Text, Sequencer-Specific, End of Track, and Set Tempo; defaults to Track Name)
- text (string type)
- tempo (32-bit unsigned integer; defaults to 500,000)

4.5.10.2 Output Requirements

- string out (string type)

4.5.10.3 Behavior Requirements

- Convert the delta time value to VLF format using the **midi_ToVLF.vi** subVI you created in a previous step
- Create a meta-event sub-string using the sequence 0xFF (indicates meta-event), meta-event type (refer to a table of meta-event type numbers), meta-event length (use **midi_ToVLF.vi** for this purpose), and meta-event data.
- Append the delta-time sub-string and the MIDI message sub-string to the inbound string, and output the result

Your finished subVI should behave like this one:

This is an unsupported media type. To view, please see
http://cnx.org/content/m15054/latest/midi_MakeDtMetaEvent.llb

4.5.10.4 Coding Tips

At this point you should have enough experience to proceed without assistance!

4.5.11 Top-Level VI Application

Congratulations! You now have assembled and tested six subVIs that can form the basis of many other projects. Now use your subVIs to build an **application VI** (top-level VI) to demonstrate that your subVIs work properly together. Two applications that you can easily build are described next.

4.5.11.1 Sweep Through Notes and Tones

The application block diagram pictured below produces a single-track MIDI file containing an ascending chromatic sweep over the entire range of note numbers (0 to 127). Before sounding the note, the Program Number (tone or voice selection) is set to the same value as the note number. Thus, the voice changes for each note, adding additional interest to the sound. The note duration is specified by a control whose unit is milliseconds. As an exercise, you will need to complete the grayed-out area to convert the units of “duration” from milliseconds to ticks.

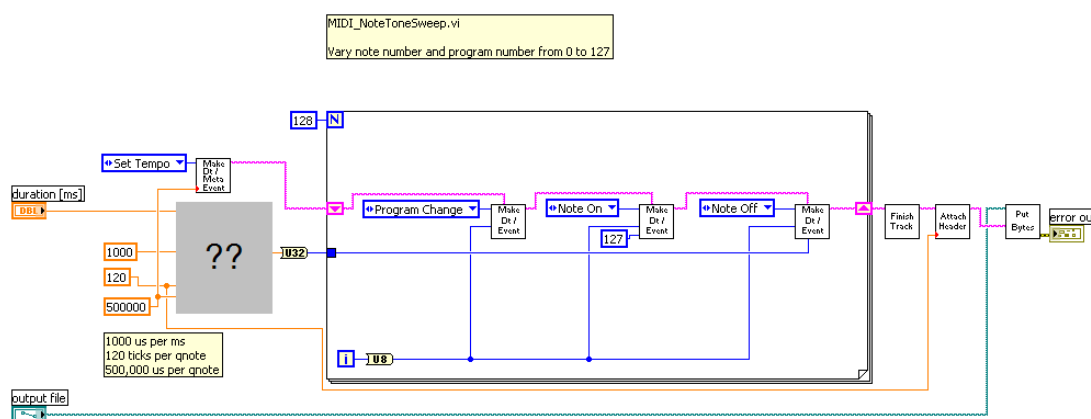


Figure 4.21: Block diagram to produce a single-track MIDI file containing an ascending chromatic sweep over the entire range of note numbers (0 to 127)

4.5.11.2 Measure the Velocity Profile of Your Soundcard

The application block diagram pictured below creates a MIDI file in which the same note is played repeatedly, but the velocity varies from the maximum to the minimum value in unit steps. When you play the MIDI file, you can record the soundcard’s audio output and measure its **velocity profile**, i.e., the mapping between the note’s velocity value and its waveform amplitude. The Audacity³⁰ sound editing application works well for this purpose; choose “Wave Out Mix” as the input device to record the soundcard’s output.

³⁰<http://audacity.sourceforge.net/>

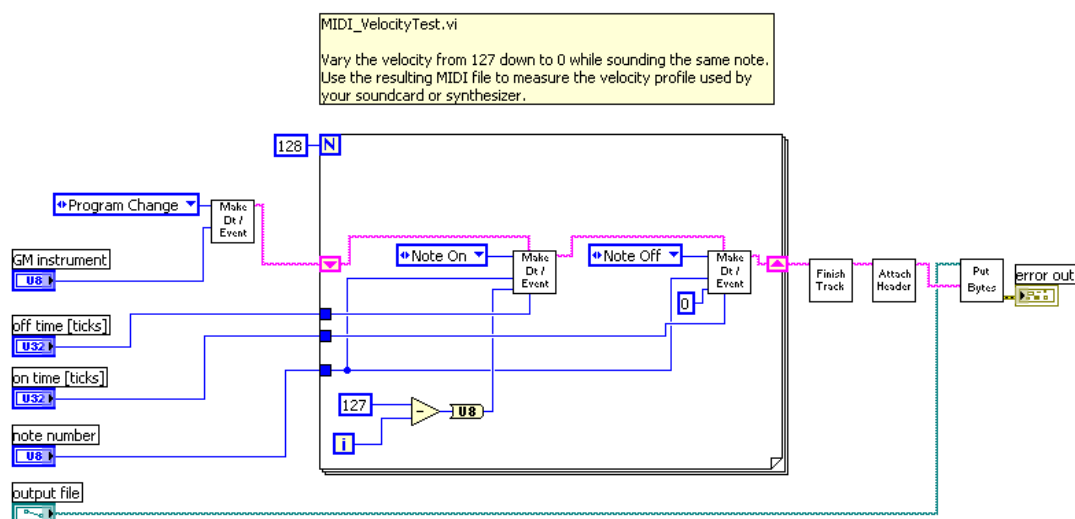


Figure 4.22: Block diagram to play a single note with velocity varied from 127 down to 0

4.6 [LabVIEW application] MIDI_JamSession³¹

4.6.1 Introduction

MIDI_JamSession is a LabVIEW application VI that reads a standard MIDI file (.mid format) and renders it to a audio using "instrument" subVIs of your own design. Following are the key features of **MIDI_JamSession**:

- Reads standard MIDI files (.mid)
- Renders note events to stereo audio using user-defined subVIs called "virtual musical instruments" (VMIs) or built-in preview instruments
- Displays relevant MIDI file information to help determine how to assign instruments to MIDI channels
- Includes basic "mixing board" with controls for instrument type, mute, and stereo pan
- Creates files for rendered audio (.wav format) and note events (.csv spreadsheet format)

A MIDI file contains note and timing information (see MIDI Messages (Section 4.1) and Standard MIDI Files (Section 4.2) for full details). Notes are associated with **channels** (up to 16 channels possible). A single channel is almost always associated with a single instrument sound. **MIDI_JamSession.vi** uses all of this information to repeatedly call your **virtual musical instrument (VMI)** which creates a single note (an audio fragment) according to the requested duration, frequency, and amplitude; the audio fragment is then superimposed on the output audio stream at the correct time.

The following screencast video demonstrates how to use **MIDI_JamSession** to render MIDI files using the default preview instruments, and how to get started creating subVIs to render audio according to your own algorithms.

³¹This content is available online at <<http://cnx.org/content/m15053/1.2/>>.

Image not finished

Figure 4.23: [video] Demonstration of the MIDI_JamSession application

4.6.2 Source Distribution

MIDI_JamSession.vi is available in this .zip archive: MIDI_JamSession_v0.92.zip³². Right-click and choose "Save As" to download the .zip file, unpack the archive into its own folder, and double-click "MIDI_JamSession_run-me.vi" to start the application.

4.6.3 Instructions

- Start "MIDI_JamSession.vi" and choose a source MIDI file (.mid format); several MIDI files are included in the .zip distribution archive (see 'readme_midi-files.txt' for details). Click the folder icon to the right of the text entry browse to browse for a file. Once you select a file, "MIDI_JamSession" immediately reads the file and updates the MIDI information display panels. If you enter a filename in the "note events output file" field, a spreadsheet (in comma-separated values format) will be created that contains all of the note events extracted from the MIDI file. The columns are: channel number (1 to 16), start time (in seconds), duration (in seconds), note number (0 to 127), and velocity (0 to 127). The .csv file will be updated each time you load a new MIDI file.
- Leave all of the audio rendering controls at their default settings at first in order to use the built-in preview instruments, and to render only the first 10 percent of the song to audio. The relatively low sampling frequency and the simple algorithm for the preview instruments ensure quick rendering when you are exploring different MIDI files. Click "Render Audio" to listen to your MIDI file.
- If you have not done so already, double-click on your MIDI file to hear it played by your default media player using the built-in synthesizer on your computer's soundcard. "MIDI_JamSession" may not work properly for some types of MIDI files, so please compare the rendered audio to your media player's rendition before you continue.
- Look at the information text panels on the lower left, especially the track listing. Each channel number (inside square brackets) is typically associated with a unique instrument, and will often be labeled as such. The text entry boxes labeled "The Band" are where you assign your "virtual musical instrument" (VMI) to render notes for a given channel. Note that Channel 10 is reserved for percussion. The preview drum instrument renders all note events on Channel 10 the same way, regardless of note number or note velocity (it sounds a bit like a snare drum).
- Experiment with the pan controls to position each instrument in the stereo sound field; click "random pan" to make a random assignment for each instrument. You can also mute selected channels in order to isolate certain instruments, or to create a solo. Click the "Lock to 1" button to cause all controls to track those of Channel 1; this is an easy way to mute or unmute all channels, for example. Adjust the two sliders on the "time range to render" control to pick the start and stop times to render. You can quickly preview sections in the middle or end of the song this way. Set the controls to 0 and 100 percent to render the entire song.
- You will eventually find it more convenient to turn off the "Listen to audio" option and enter a filename in the "audio output file (.wav)" field. Each time you click "Render Audio" the .wav file will update, and you can use your own media player to listen to the .wav file. There is presently no way to interrupt

³²http://cnx.org/content/m15053/latest/MIDI_JamSession_v0.92.zip

the built-in audio player, and this can be a nuisance when you render long pieces. The yellow LED indicator at the upper right corner indicates when the built-in audio player is active.

- Once you are ready to create your own instrument sounds, open "vmi_Prototype.vi" and carefully follow the instructions inside. Eventually you will create a number of different VMIs, with each having the ability to generate an audio fragment that renders a single note.
- De-select the "Preview only" button, and select the VMI you wish to use for each channel in the vertical array of folders called "The Band." Blank entries will render as silence. Remember to adjust your sampling frequency as needed, bearing in mind that CD-quality (44.1 kHz) will increase the rendering time and increase the size of the .wav file.
- Render your new audio file.
- Enjoy listening!

IMPORTANT: Once you have invested a lot of effort to adjust the front panel settings, exit the application (click "Exit" just under the "MIDI Jam Session" logo), select "Edit | Make Current Values Default," and press Ctrl+S to save "MIDI_JamSession.vi" with your own settings.

Solutions to Exercises in Chapter 4

Solution to Exercise 4.1 (p. 49)

268 [watch a video of the solution process³³]

Solution to Exercise 4.2 (p. 49)

E7A14F5 [watch a video of the solution process³⁴]

³³http://cnx.org/content/m15051/latest/midi_VLF-ex1.html

³⁴http://cnx.org/content/m15051/latest/midi_VLF-ex2.html

Chapter 5

Tremolo and Vibrato Effects (Low-Frequency Modulation)

5.1 Tremolo Effect¹

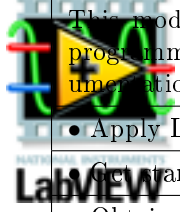
	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	<ul style="list-style-type: none">• Apply LabVIEW to Audio Signal Processing
	Get started with LabVIEW
	<ul style="list-style-type: none">• Obtain a fully-functional evaluation edition of LabVIEW

Table 5.1

5.1.1 Overview

Tremolo is a type of low-frequency amplitude modulation. After learning about the vibraphone, a mallet-type percussion instrument that creates tremolo, experiment with the tremolo effect using an interactive LabVIEW VI and learn how to model the tremolo effect mathematically.

5.1.2 Physical Tremolo: Vibraphone

The **vibraphone** is a mallet-type percussion instrument similar to the xylophone and marimba. The percussionist in the right foreground of Figure 5.1 is playing a vibraphone.

¹This content is available online at <<http://cnx.org/content/m15497/1.2/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>



Figure 5.1: Vibraphone (right foreground); click for larger version. Photographer: Charles Dietlein (<http://www.flickr.com/photos/dietlein/232084492/>³). Copyright holder has granted permission to display this image under the Creative Commons Attribution-NonCommercial-NoDerivs license⁴.

³<http://www.flickr.com/photos/dietlein/232084492/>

⁴<http://creativecommons.org/licenses/by-nc-nd/2.0>

Following are the vibraphone's key characteristics:

- Range of three octaves, beginning on F3 (the F below middle C)
- Playing surface is covered by metal bars; the pitch of each bar increases as the length decreases; bars are typically struck by soft cord- or yarn-covered mallets
- Sound intensity is increased by placing a series of resonating tubes (resonators) directly under each bar
- Sustain pedal controls whether or not a damper is applied to the metal bars, giving the vibraphonist similar expressive control as a piano
- Motor-driven disks rotate between the metal bars and resonators cause sound intensity to fluctuate (**tremolo** effect)

The name "vibraphone" was originally derived from the term "vibrato," since the undulating sound of a vibraphone resembles that of a vocalist singing a long note with vibrato. However, **vibrato** refers to a low-frequency fluctuation in **frequency**, an altogether different effect (see Vibrato Effect (Section 5.3) for details).

5.1.3 Tremolo Demonstration



Download and run the LabVIEW VI tremolo_demo.vi⁵, which demonstrates the tremolo effect applied to a sinusoidal oscillator. Tremolo normally requires two controls: **rate** determines how quickly the amplitude should fluctuate, and **depth** establishes the amount of amplitude fluctuation. The third control adjusts the pitch of the sinusoidal oscillator.

5.1.4 Modeling the Tremolo Effect

Tremolo is a type of low-frequency **amplitude modulation**. The screencast video of Figure 5.2 develops the mathematical equations needed to model the tremolo effect. After watching the video, try the exercises below to ensure that you understand the main concepts.

Image not finished

Figure 5.2: [video] Develop the mathematical equations needed to model the tremolo effect

Exercise 5.1

What is the name of the term f_R ?

(Solution on p. 82.)

Exercise 5.2

Which ratio is the basis of depth when expressed in decibels (dB)?

(Solution on p. 82.)

Exercise 5.3

True/False: Tremolo rate is typically above 20 Hz.

(Solution on p. 82.)

Exercise 5.4

Which modification to the basic envelope equation is required to avoid clipping?

(Solution on p. 82.)

Now that you have been introduced to the main concepts of the tremolo effect, return to the interactive VI of the previous section. Experiment with the depth and rate controls, and confirm that the typical values mentioned in the screencast video in Figure 5.2 seem reasonable.

⁵http://cnx.org/content/m15497/latest/tremolo_demo.vi

5.2 [mini-project] Vibraphone virtual musical instrument (VMI) in LabVIEW⁶

⁶This content is available online at <<http://cnx.org/content/m15498/1.2/>>.


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide⁷ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 5.2

5.2.1 Objective

The **vibraphone** percussion instrument can be well-modeled by a sinusoidal oscillator, an attack-decay envelope with a short attack and a long decay, and a low-frequency sinusoidal amplitude modulation. In this mini-project you will develop code to model the vibraphone as a LabVIEW **virtual musical instrument (VMI)** to be "played" by a MIDI file within **MIDI JamSession**.

5.2.2 Prerequisite Modules

If you have not done so already, please study the pre-requisite module Tremolo Effect (Section 5.1). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing⁸ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

5.2.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

5.2.4 Part 1: Tremolo Envelope Generator

Create LabVIEW code to create a time-varying intensity envelope for the tremolo effect. Your code will require tremolo rate (in Hz), depth (in dB), and total number of samples and will produce a tremolo envelope with a sinusoidal shape as follows:

This is an unsupported media type. To view, please see
<http://cnx.org/content/m15498/latest/behavior-tremolo-envelope.llb>

The maximum intensity will be fixed at 0 dB, and the sinusoid's amplitude will be "depth." Once you develop your code, compare its behavior with that of the interactive front panel above. Note that the time range of the interactive front panel is fixed at 1 second, but your code should produce any number of required samples.

⁷"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

⁸*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
 <<http://cnx.org/content/col10440/latest/>>

5.2.5 Part 2: Attack/Decay Envelope Generator

Create LabVIEW code to create a time-varying intensity envelope for the overall attack and decay of the note. Your code will require attack time and decay time (both in seconds), and will produce an envelope composed of two straight-line segments as follows:

This is an unsupported media type. To view, please see
<http://cnx.org/content/m15498/latest/behavior-AD-envelope.llb>

The maximum intensity will be fixed at 0 dB, and the minimum intensity will be -40 dB. Once you develop your code, compare its behavior with that of the interactive front panel above.

5.2.6 Part 3: Attenuator

Create LabVIEW code that will accept an "amplitude" parameter in the range 0 to 1, and will convert this parameter to an attenuation in the range -40 dB to 0 dB. The amplitude parameter will ultimately be supplied by MIDI_JamSession and represents the MIDI "note-on" velocity. Your code will map linear velocity onto a logarithmic intensity.

5.2.7 Part 4: Overall Amplitude Envelope

Combine the code fragments you developed in Parts 1 to 3 to create an overall intensity envelope. Remember that when you work with intensity values in decibels, you simply need to **add** them together. Next, "undo" the equation for decibels to convert the intensity envelope into an amplitude envelope (hint: you need a value of "20" someplace). Choose a representative set of parameter values and plot your overall intensity envelope and your overall amplitude envelope.

5.2.8 Part 5: Vibraphone VMI

In this part you will design a vibraphone **virtual musical instrument** (VMI for short) that can be played by "MIDI_JamSession." If necessary, visit MIDI_JamSession (Section 4.6), download the application VI .zip file, and view the screencast video in that module to learn more about the application and how to create your own virtual musical instrument. Your VMI will accept parameters that specify frequency and amplitude of a single note, and will produce an array of audio samples corresponding to a single strike on the metal bar of a vibraphone instrument. Use a sinusoidal signal as the oscillator (tone generator), and apply the amplitude envelope you generated in Part 4. You may wish to keep your parameters as front-panel controls and add the "Play Waveform" ExpressVI to listen to your VMI during development. Adjust the parameters to obtain pleasing and realistic settings, then convert the front-panel controls to constants and remove "Play Waveform." Your finished VMI must not contain any front panel controls or indicators beyond what is provided in the prototype instrument.

NOTE: The prototype VMI includes the "length" parameter to set the number of samples to be produced by your own design. The length is derived from the amount of elapsed time between "note on" and "note off" MIDI messages for a given note. To make a more realistic sound for the vibraphone, ignore this length value and create a fixed number of samples determined by your attack and decay times.

Finally, choose a suitable MIDI file and use MIDI_JamSession to play your vibraphone VMI. MIDI files that contain a solo instrument, slower tempo, and distinct notes will likely produce better results. Create a .wav file of your finished work. The Figure 5.3 screencast video provides some coding tips.

Image not finished

Figure 5.3: [video] Coding tips for Part 5

5.2.9 Optional: Modifications to Basic Vibraphone VMI

Following are some suggested modifications you could try for your basic vibraphone VMI:

- Make the decay time vary according to the "length" parameter provided by the prototype VMI. While a variable decay time may not necessarily be physically realistic, it may sound interesting.
- Use a fixed decay time, but use the "length" parameter to determine when to cut off (or damp) the tone. You will need to include a short envelope segment to taper the amplitude back to zero, because an abrupt cutoff will cause click noise.
- Make the tremolo depth (or rate, or both) vary according to the "amplitude" parameter provided by the prototype VMI. For example, a higher amplitude could be mapped to a faster rate or more depth.
- Remove the tremolo envelope from the vibraphone VMI, and use it as a single envelope for the entire piece (you would need to read the .wav file produced by MIDI JamSession and apply the tremolo envelope). On the real vibraphone, the rotating disks turn at the same rate for all of the resonators, so placing the tremolo on each individual note is not the best way to model the physical instrument.

5.3 Vibrato Effect⁹


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹⁰ module for tutorials and documentation that will help you:</p> <ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing <p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW
---	---

Table 5.3

5.3.1 Overview

Vibrato is a type of low-frequency frequency modulation. After learning about vibrato produced by the singing voice and musical instruments, you will experiment with the vibrato effect using an interactive LabVIEW VI and learn how to model the vibrato effect mathematically.

⁹This content is available online at <<http://cnx.org/content/m15496/1.2/>>.

¹⁰"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

5.3.2 Physical Vibrato: Singing Voice and Instruments

Vocalists and instrumentalists will introduce **vibrato** – a low-frequency variation in pitch – into long sustained notes primarily to add musical interest. Listeners are drawn to sounds with dynamic (changing) spectral characteristics, and vibrato makes a sustained note sound much more interesting than a constant frequency. Moreover, sustaining a long note at a constant frequency with sufficient accuracy to avoid drifting "out of tune" is challenging for vocalists and wind-based instruments. Vibrato is produced in a variety of ways, depending on the instrument. Trombonists wiggle the slide slightly to change the overall tube length that sets pitch. A violinist will rock his or her left hand that presses the string to slightly alter the effective string length.

5.3.3 Vibrato Demonstration



Download and run the LabVIEW VI `vibrato.vi`¹¹ to demonstrate the vibrato effect applied to a sinusoidal oscillator. This VI requires the TripleDisplay¹² front-panel indicator. Vibrato normally requires two controls: **rate** determines how quickly the frequency should fluctuate, and **depth** establishes the amount of frequency fluctuation. The third control adjusts the pitch of the sinusoidal oscillator.

5.3.4 Modeling the Vibrato Effect

Vibrato is a type of low-frequency **frequency modulation**. In this section the mathematical equations necessary to model the vibrato effect will be developed. In addition, two important effects associated with the singing voice will be discussed to produce a more realistic model.

5.3.4.1 Naive Approach

The Figure 5.4 screencast video develops the mathematical equation needed to model the vibrato effect in perhaps an intuitively-obvious (but unfortunately incorrect) way. After watching the video, try the interactive front panel VI below that is part of the demonstration, then respond to the exercise questions to ensure that you understand the main concepts.

Image not finished

Figure 5.4: [video] Perhaps "intuitively-obvious" (but incorrect) way to model vibrato



Download and run the LabVIEW VI `vibrato_naive.vi`¹³.

Exercise 5.5

(Solution on p. 82.)

What is the main auditory effect produced by the intuitively-obvious approach to modeling vibrato?

Exercise 5.6

(Solution on p. 82.)

When modifying the basic sinusoidal oscillator equation, which part – frequency or phase - requires the most attention?

¹¹<http://cnx.org/content/m15496/latest/vibrato.vi>

¹²"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

¹³http://cnx.org/content/m15496/latest/vibrato_naive.vi

Exercise 5.7*(Solution on p. 82.)*

How should the phase function $\varphi(t)$ be designed to achieve vibrato?

5.3.4.2 Correct Approach

The Figure 5.5 screencast video develops the mathematical equation needed to model the vibrato effect for a constant low-frequency variation.



Refer again to the LabVIEW VI vibrato.vi¹⁴ you downloaded earlier.

Image not finished

Figure 5.5: [video] Correct way to model vibrato

5.3.4.3 Improved Realism for Singing Voice

Several effects become immediately apparent when listening to an opera singer:

- Vibrato rate begins slowly then increases to a faster rate; for example, listen to this short clip: sing.wav¹⁵
- Vibrato depth increases as the note progresses (listen to the clip again: sing.wav¹⁶)
- Loudness (intensity) is initially low then gradually increases (listen to the same clip one more time: sing.wav¹⁷)
- The "brightness" (amount of overtones or harmonics) is proportional to intensity (please listen to the same clip one last time: sing.wav¹⁸)

These effects are also evident when listening to expressive instrumentalists from the strings, brass, and woodwind sections of the orchestra. The mathematical model for vibrato can therefore be improved by (1) making the vibrato depth **track** (or be proportional to) the intensity envelope of the sound, and by (2) making the vibrato rate track the intensity envelope. Modeling the "brightness" effect would require adding overtones or harmonics to the sound.

¹⁴<http://cnx.org/content/m15496/latest/vibrato.vi>


¹⁵<http://cnx.org/content/m15496/latest/sing.wav>

¹⁶<http://cnx.org/content/m15496/latest/sing.wav>

¹⁷<http://cnx.org/content/m15496/latest/sing.wav>

¹⁸<http://cnx.org/content/m15496/latest/sing.wav>

5.4 [mini-project] "The Whistler" virtual musical instrument (VMI) in LabVIEW¹⁹

 The LabVIEW logo features a stylized 'E' with a red and blue waveform on the left and a green waveform on the right, all within a black square frame. Below the frame is the text 'NATIONAL INSTRUMENTS' and 'LabVIEW'.	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide²⁰ module for tutorials and documentation that will help you:</p>
	<p><i>continued on next page</i></p>

¹⁹This content is available online at <<http://cnx.org/content/m15500/1.1/>>.

• Apply LabVIEW to Audio Signal Processing
• Get started with LabVIEW
• Obtain a fully-functional evaluation edition of LabVIEW

Table 5.4

5.4.1 Objective

An individual who can whistle with vibrato can be well-modeled by a sinusoidal oscillator, an attack-sustain-release envelope with a moderate attack and release time, and a low-frequency sinusoidal frequency modulation. In this mini-project you will develop code to model the whistler as a LabVIEW **virtual musical instrument (VMI)** to be "played" by a MIDI file.

5.4.2 Prerequisite Modules

If you have not done so already, please study the pre-requisite module Vibrato Effect (Section 5.3). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing²¹ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

5.4.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

5.4.4 Part 1: Tone Generator with Vibrato

In this part you will create a basic tone generator with vibrato. The tone generator will be a sinusoid of the form $y(t) = \sin(\varphi(t))$, where the phase function $\varphi(t)$ has the following form ((5.1)):

$$\varphi(t) = 2\pi f_0 t + \Delta f \sin(2\pi f_R t) \quad (5.1)$$

where f_0 is the tone frequency, Δf is the frequency deviation (vibrato depth), and f_R is the vibrato rate in Hz. Use the "Play Waveform" Express VI to listen to your end result $y(t)$, and experiment with the parameters to find suitable values for rate and depth to simulate the sound of a whistler. Refer to the screencast video in the module Frequency Modulation (FM) Techniques in LabVIEW (Section 6.5) for coding tips for this part.

5.4.5 Part 2: Attack-Sustain-Release Envelope Generator

Create LabVIEW code to generate a time-varying intensity envelope for the overall attack, sustain, and decay of the note. Your code will require attack time and decay time (both in seconds), as well as the total number of required samples, and will produce an envelope composed of three straight-line segments as plotted in Figure 5.6.

²⁰"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

²¹*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
<<http://cnx.org/content/col10440/latest/>>

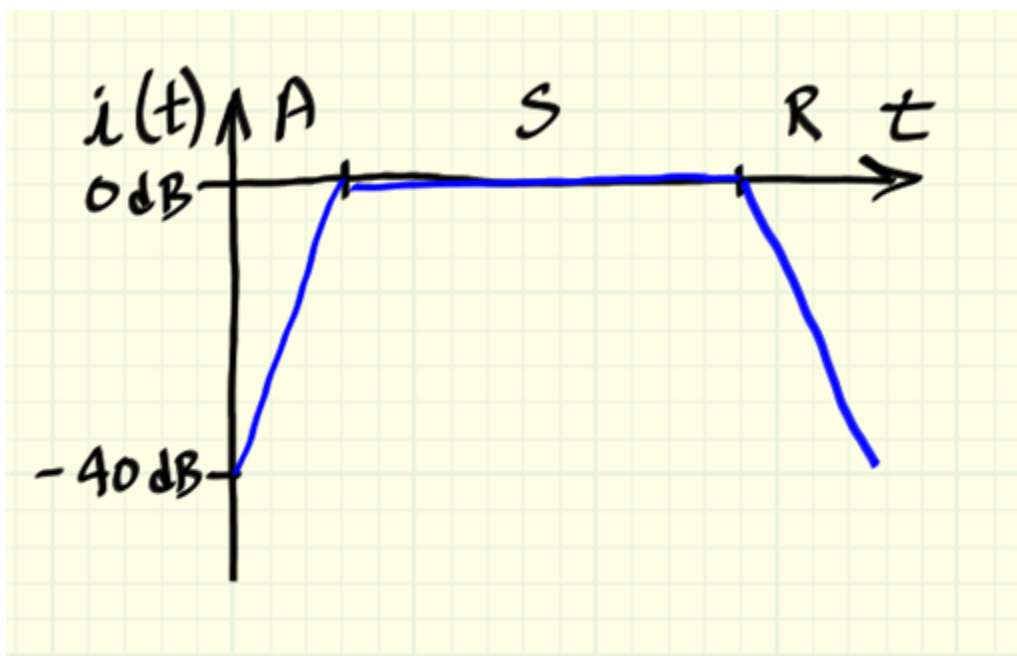


Figure 5.6: Attack-Sustain-Release envelope

The maximum intensity is fixed at 0 dB, and the minimum intensity is -40 dB. The attack and release times are fixed parameters that you adjust, and the sustain time is "stretchable" depending on the total number of required samples. If you have the inclination, make your envelope generator more robust so that it can handle the situation where the requested number of samples is less than the number of samples required for your attack and release intervals.

5.4.6 Part 3: Attenuator

Create LabVIEW code that accepts an "amplitude" parameter in the range 0 to 1 and converts this parameter to attenuation in the range -40 dB to 0 dB. The amplitude parameter will ultimately be supplied by MIDI_JamSession and represents the MIDI "note-on" velocity. Your code will map linear velocity onto a logarithmic intensity.

5.4.7 Part 4: Overall Amplitude Envelope

Combine the code fragments you developed in Parts 2 and 3 to create an overall intensity envelope. Remember that when you use intensity values in decibels, you simply **add** them together. Next, "undo" the equation for decibels to convert the intensity envelope into an amplitude envelope (hint: you need a value of "20" someplace). Choose a representative set of parameter values and plot your overall intensity envelope and your overall amplitude envelope.

5.4.8 Part 5: Whistler VMI

Design a **virtual musical instrument** (VMI for short) that sounds like someone whistling with vibrato. Your VMI will be played by "MIDI Jam Session." If necessary, visit MIDI Jam Session (Section 4.6), download

the application VI .zip file, and view the screencast video in that module to learn more about the application and how to create an instrument subVI, or VMI. Your VMI will accept parameters that specify frequency, amplitude, and length of a single note, and will produce an array of audio samples corresponding to a single note. Use the tone generator you developed in Part 1, and apply the amplitude envelope you generated in Part 4. You may wish to keep all of your parameters as front-panel controls and add the "Play Waveform" Express VI to listen to your VMI during development. Adjust the parameters to obtain pleasing and realistic settings, then convert the front-panel controls to constants and remove "Play Waveform." Your finished VMI must not contain any front panel controls or indicators beyond those provided in the prototype instrument. Choose a suitable MIDI file and use MIDI_JamSession to play your whistler VMI. MIDI files that contain a solo instrument, slow tempo, and long sustained notes likely produce better results, for example, Johann Pachelbel's "Canon in D." Try Pachelbel_Canon_in_D.mid²² at the Classical Guitar MIDI Archives²³. You can also find a more extensive collection at ClassicalArchives.com²⁴, specifically Pachelbel MIDI files²⁵. Create a .wav file of your finished work.

5.4.9 Optional: Modifications to Basic Whistler VMI

Following are some suggested modifications you could try for your basic whistler VMI:

- Make the vibrato rate proportional to the intensity envelope. This characteristic is common for vocalists and many types of instrumentalists.
- Make the vibrato depth proportional to the intensity envelope. This is another characteristic common for vocalists and many types of instrumentalists.
- Vary either the vibrato rate or depth (or possibly both) according to the "amplitude" parameter provided by the prototype VMI. For example, higher amplitudes could be mapped to a faster rate or more depth.
- Duplicate the tone generator two more times with frequencies of $2f_0$ and $3f_0$ and intensities of -10 dB and -20 dB, respectively, to create some overtones. Each of the tone generators should have the same vibrato rate and depth. The overtones make the whistler sound a bit more like a flute or a singing voice.

²²http://www.classicalguitarmidi.com/subivic/Pachelbel_Canon_in_D.mid

²³<http://www.classicalguitarmidi.com>

²⁴<http://www.classicalarchives.com>

²⁵<http://www.classicalarchives.com/midi/p.html>

Solutions to Exercises in Chapter 5

Solution to Exercise 5.1 (p. 71)

Rate

Solution to Exercise 5.2 (p. 71)

Ratio of maximum to minimum loudness

Solution to Exercise 5.3 (p. 71)

False; tremolo rate is typically between 3 and 10 Hz

Solution to Exercise 5.4 (p. 71)

Subtract the depth value D

Solution to Exercise 5.5 (p. 76)

The amount of frequency fluctuation (deviation) increases with time rather than remaining constant.

Solution to Exercise 5.6 (p. 76)

Phase; the entire argument to the sine function must be considered as a time-varying phase function $\varphi(t)$

Solution to Exercise 5.7 (p. 77)

A ramp function with a superimposed sinusoidal variation.

Chapter 6

Modulation Synthesis

6.1 Amplitude Modulation (AM) Mathematics¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	<ul style="list-style-type: none">• Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none">• Get started with LabVIEW• Obtain a fully-functional evaluation edition of LabVIEW

Table 6.1

6.1.1 Overview

Amplitude modulation (AM) is normally associated with communications systems; for example, you can find all sorts of "talk radio" stations on the AM band. In communication systems, the **baseband** signal has a bandwidth similar to that of speech or music (anywhere from 8 kHz to 20 kHz), and the modulating frequency is several orders of magnitude higher; the AM radio band is 540 kHz to 1600 kHz.

When applied to audio signals for music synthesis purposes, the modulating frequency is of the same order as the audio signals to be modulated. As described below, AM (also known as **ring modulation**) splits a given signal spectrum in two, and shifts one version to a higher frequency and the other version to a lower frequency. The modulated signal is the sum of the frequency-shifted spectra, and can provide interesting special effects when applied to speech and music signals.

6.1.2 Modulation Property of the Fourier Transform

The **modulation property** of the Fourier transform forms the basis of understanding how AM modifies the spectrum of a source signal. The screencast video of Figure 6.1 explains the modulation property concept and derives the equation for the modulation property.

¹This content is available online at <<http://cnx.org/content/m15447/1.2/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

Image not finished

Figure 6.1: [video] Modulation property concepts and derivation

Suppose the source signal to be modulated contains only one spectral component, i.e., the source is a sinusoid. The screencast video of Figure 6.2 shows how to apply the modulation property to predict the spectrum of the modulated signal. Once you have studied the video, try the exercises below to ensure that you understand how to apply the property for a variety of different modulating frequencies.

Image not finished

Figure 6.2: [video] Determine the spectrum of a modulated sinusoid

The time-domain signal $x(t)$ is a sinusoid of amplitude $2A$ with corresponding frequency-domain spectrum as shown in Figure 6.3.

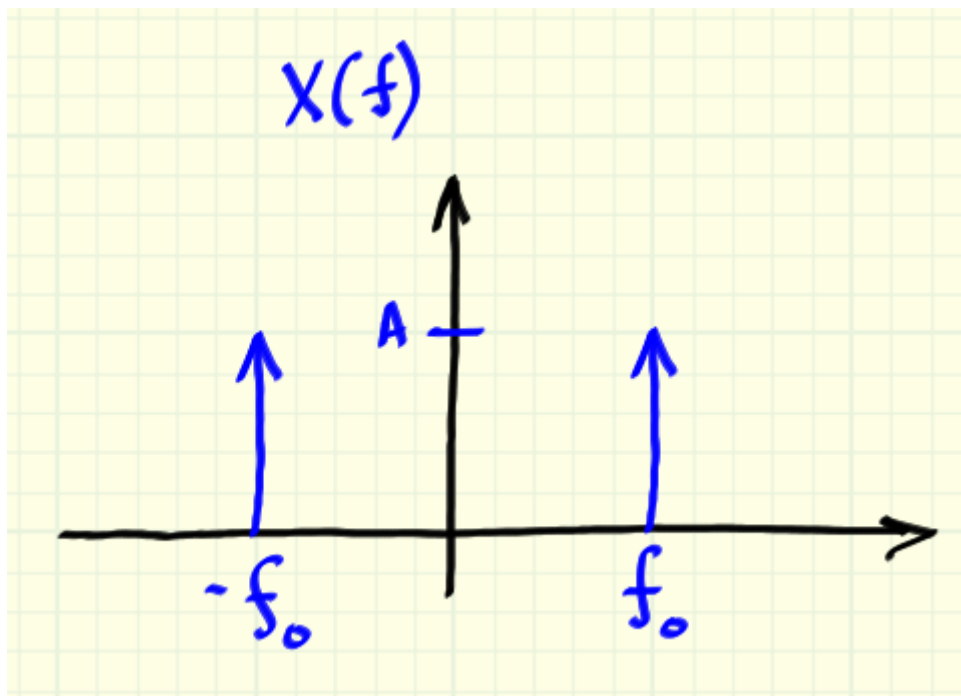


Figure 6.3: Spectrum of the signal $x(t)$

Suppose $x(t)$ is modulated by a sinusoid of frequency f_m . For each of the exercises below, draw the spectrum of the modulated signal $y(t) = \cos(2\pi f_m t) \times x(t)$, where the exercise problem statement indicates the modulation frequency.

Exercise 6.1

$$f_m = f_0/5$$

(Solution on p. 101.)

Exercise 6.2

$$f_m = f_0/2$$

(Solution on p. 101.)

Exercise 6.3

$$f_m = f_0$$

(Solution on p. 101.)

Exercise 6.4

$$f_m = 1.5f_0$$

(Solution on p. 102.)

Exercise 6.5

$$f_m = 2f_0$$

(Solution on p. 102.)

Did you notice something interesting when f_m becomes larger than f_0 ? The right-most negative frequency component shifts into the positive half of the spectrum, and the left-most positive frequency component shifts into the negative half of the spectrum. This effect is similar to the idea of **aliasing**, in which a sinusoid whose frequency exceeds half the sampling frequency is said to be "folded back" into the principal alias. In the case of AM, modulating a sinusoid by a frequency greater than its own frequency folds the left-most component back into positive frequency.

6.1.3 Audio Demonstrations

The screencast video of Figure 6.4 demonstrates the aural effects of modulating a single spectral component, i.e., a sinusoid. The LabVIEW code for the demo is also described in detail, especially the use of an **event structure** contained in a **while-loop structure** (see video in Figure 6.5). The event structure provides an efficient way to run an algorithm with real-time interactive parameter control without polling the front panel controls. The event structure provides an alternative to the polled method described in Real-Time Audio Output in LabVIEW (Section 1.7).



The LabVIEW VI demonstrated within the video is available here: [am_demo1.vi](http://cnx.org/content/m15447/latest/am_demo1.vi)³. Refer to TripleDisplay⁴ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 6.4: [video] Modulating a single sinusoid

³http://cnx.org/content/m15447/latest/am_demo1.vi

⁴"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

Image not finished

Figure 6.5: [video] LabVIEW implementation of AM demo using **event structure**

The next screencast video (see Figure 6.6) demonstrates the aural effects of modulating two spectral components created by summing together a sinusoid at frequency f_0 and another sinusoid at frequency $2f_0$. You can obtain interesting effects depending on whether the spectral components end up in a harmonic relationship; if so, the components fuse together and you perceive a single pitch. If not, you perceive two distinct pitches.



The LabVIEW VI demonstrated within the video is available here: [am_demo2.vi](#)⁵. Refer to TripleDisplay⁶ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 6.6: [video] Modulating a pair of sinusoids

The third demonstration (see Figure 6.7) illustrates the effect of modulating a music clip and a speech signal. You can obtain interesting special effects because the original source spectrum simultaneously shifts to a higher and lower frequency.



The LabVIEW VI demonstrated within the video is available here: [am_demo3.vi](#)⁷. Refer to TripleDisplay⁸ to install the front-panel indicator used to view the signal spectrum.

The two audio clips used in the example are available here: [flute.wav](#)⁹ and [speech.wav](#)¹⁰ (speech clip courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech¹¹; the sentences are two of the many phonetically-balanced **Harvard Sentences**, an important standard for the speech processing community).

Image not finished

Figure 6.7: [video] Modulating a music clip and a speech signal

⁵http://cnx.org/content/m15447/latest/am_demo2.vi

⁶"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

⁷http://cnx.org/content/m15447/latest/am_demo3.vi

⁸"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

⁹<http://cnx.org/content/m15447/latest/flute.wav>

¹⁰<http://cnx.org/content/m15447/latest/speech.wav>

¹¹http://www.voiptroubleshooter.com/open_speech

6.2 Pitch Shifter with Single-Sideband AM¹²

¹²This content is available online at <<http://cnx.org/content/m15467/1.2/>>.

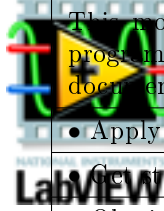
	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹³ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p>
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 6.2

6.2.1 Overview

Amplitude modulation (AM) of a source signal divides the signal's spectrum into two copies, with one copy shifted towards higher frequency and the other copy shifted towards lower frequency; refer to AM Mathematics (Section 6.1) for a complete treatment of basic AM. The shifted and dual spectrum makes an interesting special effect when applied to a musical instrument or the human voice, creating the sensation of two different people speaking the identical phrase, for example.

If one of these spectral images could somehow be cancelled out, AM seems to be a feasible way to implement a **pitch shifter**, a device or algorithm that shifts the source spectrum higher or lower in frequency. When this special effect is applied in real time, you can speak into a microphone and sound just like one of "Alvin and the Chipmunks."

As an example of what you will be able accomplish by applying the techniques presented in this module, listen to this original speech clip `speech.wav`¹⁴ and its pitch-shifted version `speech_shifted.wav`¹⁵ (speech clip courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech¹⁶; the sentences are two of the many phonetically balanced **Harvard Sentences**, an important standard for the speech processing community).

6.2.2 Single-Sideband AM (SSB-AM)

The screencast video of Figure 6.8 develops the basic theory of **single-sideband (SSB) modulation**, a technique borrowed from communications systems that provides a way to apply amplitude modulation with spectral image cancellation.

Image not finished

Figure 6.8: [video] Single-sideband modulation for pitch shifting

As an exercise to ensure that you followed each step, draw a block diagram or flow diagram to show how the original signal is modified to produce the final shifted signal. Your diagram should include directed lines (arrows) to show signal flow, and should use symbols (blocks) for the multipliers, cosine and sine oscillators, Hilbert transformer, sign changer, and adder.

¹³"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

¹⁴<http://cnx.org/content/m15467/latest/speech.wav>

¹⁵http://cnx.org/content/m15467/latest/speech_shifted.wav

¹⁶http://www.voiptroubleshooter.com/open_speech

6.2.3 Pre-Filtering to Avoid Aliasing

Pre-filtering the source signal ensures the shifted spectrum does not alias, since the source signal typically fills the available bandwidth. The screencast video of Figure 6.9 discusses the aliasing problem as well as the techniques you can use to design a suitable pre-filter.

Image not finished

Figure 6.9: [video] Pre-filtering to avoid aliasing

6.3 [mini-project] Ring Modulation and Pitch Shifting¹⁷


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ¹⁸ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 6.3

6.3.1 Overview

Ring modulation (AM) is an audio special effect that produces two frequency-shifted replicas of the spectrum of a source signal, with one replica shifted to higher frequency and the other replica to a lower frequency. **Single-sideband AM (SSB-AM)** provides a way to shift the source signal's spectrum higher or lower but without the additional replica. SSB-AM provides one way to implement a **pitch shifter**, an audio special effect that shifts the frequency of speech, singing, or a musical instrument to a higher or lower frequency.

In this project, use LabVIEW to implement several types of ring modulators and a pitch shifter.

6.3.2 Prerequisite Modules

If you have not done so already, please study the prerequisite modules AM Mathematics (Section 6.1) and Pitch Shifting (Section 6.2). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing¹⁹ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

¹⁷This content is available online at <<http://cnx.org/content/m15468/1.1/>>.

¹⁸"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

¹⁹Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing
<<http://cnx.org/content/col10440/latest/>>

6.3.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

6.3.4 Part 1: Multiple Modulators

Consider an original signal $x(t)$, which is a sinusoid of frequency f_0 . The original signal is modulated by a cosine function of frequency $f_0/2$ to produce $x_1(t)$, which is in turn modulated by a cosine function of frequency $f_0/5$ to produce $x_2(t)$, which is in turn modulated by a cosine function of frequency $f_0/9$ to produce $x_3(t)$. Sketch the frequency-domain version of the four signals, i.e., sketch $X(f)$, $X_1(f)$, $X_2(f)$, and $X_3(f)$.

Create a LabVIEW implementation of the above arrangement and plot the spectrum of each of the four signals. Compare your LabVIEW results to your prediction.

6.3.5 Part 2: Multiple Modulators with Soundfile Input

Create a LabVIEW implementation of the multiple modulation scheme of Part 1 that can process a .wav audio file as the input signal. Use controls for the three modulators that will allow you to easily change their modulation frequencies. Experiment with various choices of modulation frequencies to make an interesting effect. Create two .wav files using different parameter choices.

6.3.6 Part 3: Pitch Shifter



Implement the pitch shifting algorithm based on the single-sideband AM technique discussed in Pitch Shifter with Single-Sideband AM (Section 6.2). Use a design similar to that of "am_demo3.vi" provided at the bottom of the page of AM Mathematics (Section 6.1) which accepts a .wav file as input and plays the sound. The sound clip should be relatively short (on the order of several seconds). For this part of the project, do **not** implement the pre-filter; you will do this in Part 4.

Evaluate the quality of your pitch shifter by presenting some written discussion and suitable spectrogram plots. Especially indicate whether you can find audible and visual evidence of aliasing.

The **fast Hilbert transform** built-in subVI is available in the "Signal Processing | Transforms" pallet.

6.3.7 Part 4: Pitch Shifter with Anti-Aliasing Filter

Modify your pitch shifter to include a bandpass filter. State how you will compute the bandpass filter's upper and lower corner frequencies, given that you want to preserve as much of the original signal's bandwidth as possible.

Evaluate the quality of your modified pitch shifter by presenting some written discussion and suitable spectral plots. Compare your results with those you obtained in Part 3.

A variety of digital filters are available in the "Signal Processing | Filters" pallet.

6.3.8 Optional Part 5: Real-Time Processor

Choose one of the previous LabVIEW implementations and make it work in real time with a signal input (microphone) and interactive front-panel controls.



Evaluate the interrupt-driven approach using an event structure (see "am_demo1.vi" described in AM Mathematics (Section 6.1), as well as the polled approach used by mic_in_speaker_out.vi²⁰). Use whichever technique you prefer.

Submit your finished LabVIEW implementation as a distinct .zip file.

6.4 Frequency Modulation (FM) Mathematics²¹

	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide²² module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p>
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 6.4

6.4.1 Overview

Frequency modulation (FM) is most often associated with communications systems; for example, you can find all sorts of music stations on the FM band of your radio. In communications systems the **baseband** signal has a bandwidth similar to that of speech or music (anywhere from 8 kHz to 20 kHz), and the modulating frequency is several orders of magnitude higher; the FM radio band is 88 MHz to 108 MHz.

When applied to audio signals for music synthesis purposes, the modulating frequency is of the same order as the audio signals to be modulated. FM can create very rich spectra, and the spectra can easily be made to evolve with time. The ability of FM to produce a wide variety of interesting spectra from only two sinusoidal oscillators makes FM a fascinating synthesis technique.

6.4.2 Brief History of FM Synthesis

John Chowning was the first to systematically evaluate FM in the audio spectrum, and along with Stanford University, filed for a patent on the technique in 1975 (see U.S. Patent 4,018,121 at U.S. Patent and Trademark Office²³ or at Google Patent Search²⁴). The patent was issued in 1977, and Stanford University licensed the technology to Yamaha Corporation. Six years later in 1983, Yamaha introduced the revolutionary DX7 synthesizer (Figure 6.10), the first commercially successful instrument based on FM synthesis. The DX7 was also a milestone by introducing two other new technologies: digital synthesis and MIDI (Musical Instrument Digital Interface). The "FM sound" defines much of the pop music styles of the 1980s.

²⁰http://cnx.org/content/m15468/latest/mic_in_speaker_out.vi

²¹This content is available online at <<http://cnx.org/content/m15482/1.2/>>.

²²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

²³<http://www.uspto.gov>

²⁴<http://www.google.com/patents?vid=USPAT4018121>



Figure 6.10: Yamaha DX7 synthesizer, the first commercially successful instrument to offer FM synthesis, digital synthesis, and MIDI compatibility. The instrument pictured here is packaged in a road case. Photographer: schoschie (<http://www.flickr.com/photos/schoschie/51653026/>²⁵). Copyright holder has granted permission to display this image under the Creative Commons Attribution-ShareAlike license²⁶.

6.4.3 FM Equation

The basic FM equation is presented in (6.1):

$$y(t) = A \sin(2\pi f_c t + I \sin(2\pi f_m t)), \quad (6.1)$$

where the parameters are defined as follows:

- f_c = carrier frequency (Hz)
- f_m = modulation frequency (Hz)
- I = modulation index

²⁵<http://www.flickr.com/photos/schoschie/51653026/>

²⁶<http://creativecommons.org/licenses/by-sa/2.0/deed>

The Figure 6.11 screencast video continues the discussion by explaining the significance of each part of (6.1), and demonstrates in a qualitative fashion how the different parameters of the equation influence the spectrum of the audio signal.



Download the LabVIEW VI demonstrated in the video: fm_demo1.vi²⁷. Refer to TripleDisplay²⁸ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 6.11: [video] Significance of each part of the basic FM equation, and audio demonstration

6.4.4 FM Spectrum

The following trigonometric identity facilitates quantitative understanding of the spectrum produced by the basic FM equation of (6.1):

$$\sin(\theta + a \sin \beta) = J_0(a) \sin \theta + \sum_{k=1}^{\infty} J_k(a) \left[\sin(\theta + k\beta) + (-1)^k \sin(\theta - k\beta) \right] \quad (6.2)$$

The term $J_k(a)$ defines a Bessel function of the first kind of order k evaluated at the value a .

Note how the left-hand side of the identity possesses exactly the same form as the basic FM equation of (6.1). Therefore, the right-hand side of the identity explains where the spectral components called **sidebands** are located, and indicates the amplitude of each spectral component. The Figure 6.12 screencast video continues the discussion by explaining the significance of each part of (6.2), especially the location of the sideband spectral components.

Image not finished

Figure 6.12: [video] Trig identity of (6.2) and location of sideband spectral components

As discussed Figure 6.12 video, the basic FM equation produces an infinite number of sideband components; this is also evident by noting that the summation of (6.2) runs from $k=1$ to infinity. However, the amplitude of each sideband is controlled by the Bessel function, and non-zero amplitudes tend to cluster around the central carrier frequency. The Figure 6.13 screencast video continues the discussion by examining the behavior of the Bessel function $J_k(a)$ when its two parameters are varied, and shows how these parameters link to the modulation index and sideband number.

²⁷http://cnx.org/content/m15482/latest/fm_demo1.vi

²⁸"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

Image not finished

Figure 6.13: [video] Discussion of the Bessel function $J_k(a)$ and its relationship to modulation index and sideband number

Now that you have developed a better quantitative understanding of the spectrum produced by the basic FM equation, the Figure 6.14 screencast video revisits the earlier audio demonstration of the FM equation to relate the spectrum to its quantitative explanation.

Image not finished

Figure 6.14: [video] FM audio demonstration revisited

6.4.5 Harmonicity Ratio

The basic FM equation generates a cluster of spectral components centered about the carrier frequency f_c with cluster density controlled by the modulation frequency

f_m . Recall that we perceive multiple spectral components to be a single tone when the components are located at integer multiples of a fundamental frequency, otherwise we perceive multiple tones with different pitches. The **harmonicity ratio** H provides a convenient way to choose the modulation frequency to produce either harmonic or inharmonic tones. Harmonicity ratio is defined as:

$$H = \frac{f_m}{f_c} \quad (6.3)$$

Harmonicity ratios that involve an integer, i.e., $H = N$ or $H = 1/N$ for $N \geq 1$, result in sideband spacing that follows a harmonic relationship. On the other hand, non-integer-based harmonicity ratios, especially using irrational numbers such as π and $\sqrt{2}$, produce interesting inharmonic sounds.



Try experimenting with the basic FM equation yourself. The LabVIEW VI `fm_demo2.vi`²⁹ provides front-panel controls for carrier frequency, modulation index, and harmonicity ratio. You can create an amazingly wide variety of sound effects by strategically choosing specific values for these three parameters. The Figure 6.15 screencast video illustrates how to use the VI and provides some ideas about how to choose the parameters. Refer to TripleDisplay³⁰ to install the front-panel indicator used to view the signal spectrum.

²⁹http://cnx.org/content/m15482/latest/fm_demo2.vi

³⁰"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

Image not finished

Figure 6.15: [video] Demonstration of fm_demo2.vi

6.4.6 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Dodge, C., and T.A. Jerse, "Computer Music: Synthesis, Composition, and Performance," 2nd ed., Schirmer Books, 1997, ISBN 0-02-864682-7.

6.5 Frequency Modulation (FM) Techniques in LabVIEW³¹


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide³² module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 6.5

6.5.1 Overview

Frequency modulation synthesis (FM synthesis) creates a rich spectrum using only two sinusoidal oscillators; refer to FM Mathematics (Section 6.4) for a complete treatment of the mathematics of FM synthesis. Implementing the basic FM synthesis equation in LabVIEW requires a special technique to make one sinusoidal oscillator vary the **phase function** of the other sinusoidal oscillator. The following screencast video walks you through the implementation process of the basic FM equation and includes an audio demonstration of the equation in action.

Refer to TripleDisplay³³ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 6.16: [video] Implementing the basic FM synthesis equation in LabVIEW

³¹This content is available online at <<http://cnx.org/content/m15493/1.2/>>.

³²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

³³"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

6.6 Chowning FM Synthesis Instruments in LabVIEW³⁴

³⁴This content is available online at <http://cnx.org/content/m15494/1.2/>.


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide³⁵ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 6.6

6.6.1 Overview

Frequency modulation synthesis (FM synthesis) produces incredibly rich spectra from only two sinusoidal oscillators; refer to FM Mathematics (Section 6.4) for a complete description of the spectral characteristics of FM synthesis. You can produce even more interesting sounds with a time-varying modulation index to alter the effective bandwidth and sideband amplitudes over time. This relatively simple modification to the basic FM equation creates tones with time-varying spectra to emulate many types of physical musical instruments.

John Chowning pioneered FM synthesis in the 1970s and demonstrated how the technique could simulate instruments such as brass, woodwinds, and percussion. These techniques are the subject of this module.

6.6.2 Significance of Time-Varying Spectra

Physical musical instruments produce audio spectra that evolve with time. A typical sound begins with some type of dynamic transient, for example, as pressure builds up within a brass instrument or when a percussion instrument is first struck. The sound continues with some type of quasi steady-state behavior when mechanical energy is continually applied, i.e., blowing on a flute, bowing a violin string, and repeatedly striking a gong. Once the mechanical energy input ceases, the sound concludes by decaying in some fashion to silence.

Clearly the amplitude of the instrument's audio signal changes during the course of the tone, following the typical attack-decay-sustain-release (ADSR) envelope described in Analog Synthesis Modules (Section 3.1). Even more important, the **intensity** of the higher-frequency spectral components changes as well. The high-frequency components are often more evident during the initial transient. In fact, the dynamic nature of the spectra during the instrument's transient plays an important role in timbre perception.

6.6.3 FM Equation with Time-Varying Modulation Index

The basic FM equation with time-varying amplitude and modulation index is presented in (6.4):

$$y(t) = a(t) \sin(2\pi f_c t + i(t) \sin(2\pi f_m t)) \quad (6.4)$$

You can easily model a physical instrument with this equation by causing the modulation index $i(t)$ to **track** the time-varying amplitude $a(t)$. In this way, a louder portion of the note also has more sidebands, because the modulation index effectively controls the bandwidth of the FM spectra.

6.6.4 Chowning Instruments

John Chowning's publication, *The Synthesis of Complex Audio Spectra by Means of Frequency Modulation* (*Journal of the Audio Engineering Society*, 21(7), 1973), describes a basic structure to implement (6.4) with the following parameters:

³⁵"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

- A - peak amplitude
- I_{\max} - maximum value of modulation index $i(t)$
- I_{\min} - minimum value of modulation index $i(t)$
- f_c [Hz] - carrier frequency
- H - harmonicity ratio (f_m/f_c)
- duration [s] - duration of generated audio
- $w_1(t)$ - prototype waveform for time-varying amplitude
- $w_2(t)$ - prototype waveform for time-varying modulation index

The prototype waveforms are normalized in both dimensions, i.e., the range and domain are both zero to one. The prototype waveform $w_1(t)$ is converted to the time-varying amplitude as $a(t) = Aw_1(t)$. The prototype waveform $w_2(t)$ is converted to the time-varying modulation index as $i(t) = (I_{\max} - I_{\min})w_2(t) + I_{\min}$. Representative Chowning FM instrument specifications are described in the PDF document [chowning_instruments.pdf](#)³⁶. The Figure 6.17 screencast video walks through the complete process to implement the Chowning clarinet instrument in LabVIEW.



Download the finished VI: [chowning_clarinet.vi](#)³⁷. Refer to [TripleDisplay](#)³⁸ to install the front-panel indicator used to view the signal spectrum.

You can easily adapt the VI to create the remaining Chowning instruments once you understand the general implementation procedure.

Image not finished

Figure 6.17: [video] Implement a Chowning FM instrument in LabVIEW

6.7 [mini-project] Chowning FM Synthesis Instruments³⁹


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide⁴⁰ module for tutorials and documentation that will help you:</p> <ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing • Get started with LabVIEW • Obtain a fully-functional evaluation edition of LabVIEW
---	--

Table 6.7

³⁶http://cnx.org/content/m15494/latest/chowning_instruments.pdf

³⁷http://cnx.org/content/m15494/latest/chowning_clarinet.vi

³⁸"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

³⁹This content is available online at <<http://cnx.org/content/m15495/1.1/>>.

⁴⁰"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

6.7.1 Objective

FM synthesis creates rich spectra from only two sinusoidal oscillators, and is able to emulate the sound of many physical musical instruments. John Chowning's seminal publication on audio-range frequency modulation (FM) in 1973 describes a number of different orchestral instruments such as woodwinds, brass, and percussion that can be created merely by adjusting a few basic parameters of the basic FM equation.

In this mini-project, implement several different Chowning FM instruments and compare them to the sounds of physical instruments. Also develop code to model the Chowning algorithms as LabVIEW **virtual musical instruments (VMIs)** to be "played" by a MIDI file within **MIDI JamSession**.

6.7.2 Prerequisite Modules

If you have not done so already, please study the prerequisite modules FM Mathematics (Section 6.4) and Chowning FM Synthesis Instruments in LabVIEW (Section 6.6). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing⁴¹ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

6.7.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

6.7.4 Part 1: Chowning FM Instruments

Chowning FM Synthesis Instruments in LabVIEW (Section 6.6) provides the specifications for a generic FM synthesis instrument, parameters for a number of different instruments, and a screencast video that walks through the complete process to implement the Chowning clarinet in LabVIEW. Refer to the PDF document in that module that contains the parameters for the remaining instruments: bell, wood-drum, brass, and bassoon. Create your own LabVIEW implementation of each of these four instruments (the clarinet VI is available in that module, as well).

Save a representative sound from each of the five Chowning instruments to a .wav file.

NOTE: Consider using an audio editor such as Audacity⁴² to merge the individual .wav files into a single .wav file that you submit as your deliverable. You can also add your own voice annotation to explain your work.

6.7.5 Part 2: Comparison with Physical Instruments

Visit the **Musical Instrument Samples** database created by the Electronic Music Studios of the University of Iowa at <http://theremin.music.uiowa.edu/MIS.html>⁴³. These recordings of actual instruments were made inside an anechoic chamber to eliminate reflections and other artifacts, so the spectra of the instruments are as accurate as possible. The files are stored in AIFF format; use an audio editor such as Audacity⁴⁴ to import the AIFF format. Audacity also includes a tool to view the spectra of the soundfile.

Compare and contrast the FM sounds you created for brass, clarinet, and bassoon to those of the real instruments. Consider time-domain envelope shape and spectrogram patterns.

⁴¹ *Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
<<http://cnx.org/content/col10440/latest/>>

⁴² <http://audacity.sourceforge.net>

⁴³ <http://theremin.music.uiowa.edu/MIS.html>

⁴⁴ <http://audacity.sourceforge.net>

6.7.6 Part 3: Chowning VMIs for MIDI JamSession

In this part, convert each of the five Chowning instruments you implemented in Part 1 into its own **virtual musical instrument** (VMI for short) that can be played by "MIDI JamSession." If necessary, visit MIDI JamSession (Section 4.6), download the application VI.zip file, and view the screencast video in that module to learn more about the application and how to create your own VMI. Your VMI will accept parameters that specify frequency, amplitude, and duration of a single note, and will produce a corresponding array of audio samples.

You may wish to keep all of your existing front-panel controls available so that you can listen to your VMI during development. Adjust the parameters to obtain pleasing and realistic settings, and convert the front-panel controls to constants and remove all indicators. Your finished VMI must not contain any front-panel controls or indicators beyond those provided in the prototype instrument.

Finally, choose a suitable MIDI file and use MIDI JamSession to play your FM VMIs. MIDI files that contain multiple channels are ideal, because you can individually assign each of your five VMIs to a different instrument.

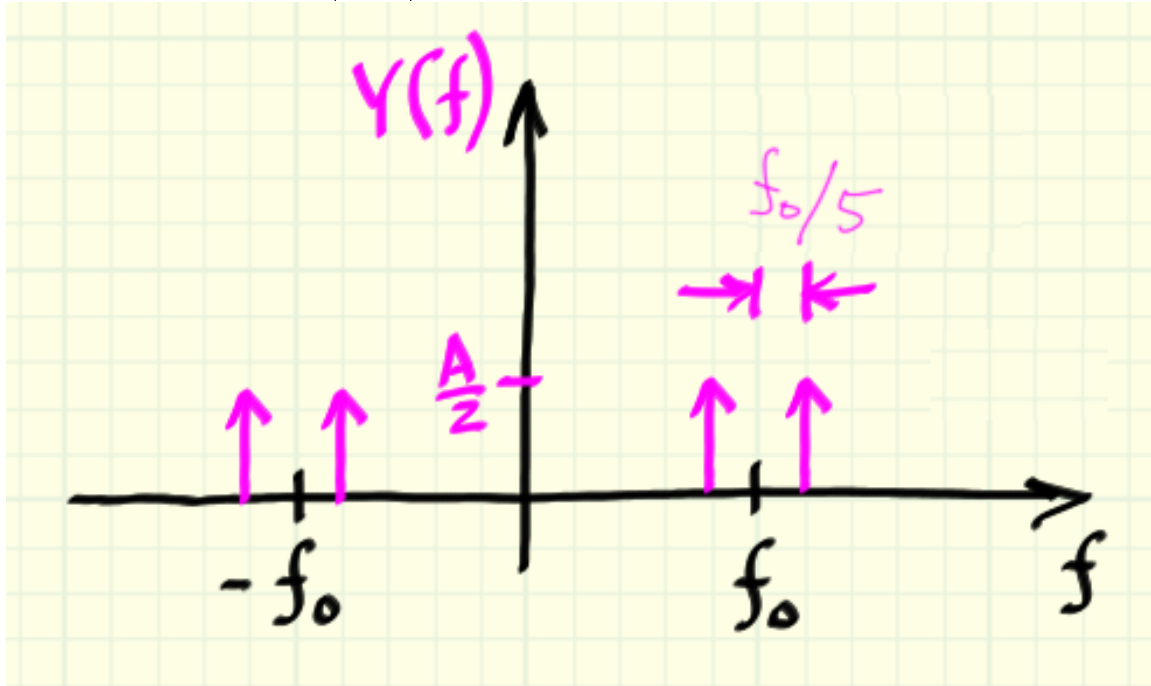
Create a .wav file of your finished work.

IMPORTANT: MIDI percussion events are found on Channel 10, a reasonable place to use your wood-drum instrument. Be aware that the "frequency" value produced by the prototype VMI derives directly from the "note number" value of the MIDI "Note On" messages in the file. On Channel 10, the note number selects from a palette of different percussion instruments as defined in the **General MIDI Sound Set** (<http://www.midi.org/about-midi/gm/gm1sound.shtml>⁴⁵), so interpreting the value as frequency is meaningless. You can either set up your wood-drum to produce the same waveform independent of the frequency parameter, or you can devise a scheme to translate the note number into useful parameter change for your instruments.

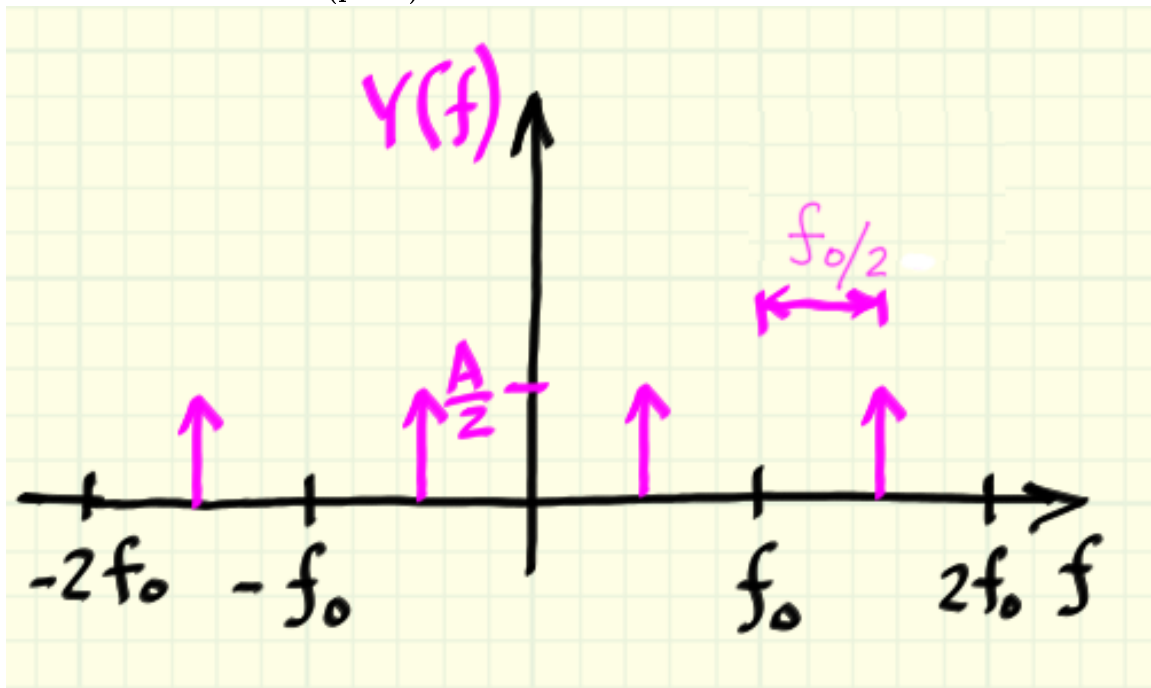
⁴⁵<http://www.midi.org/about-midi/gm/gm1sound.shtml>

Solutions to Exercises in Chapter 6

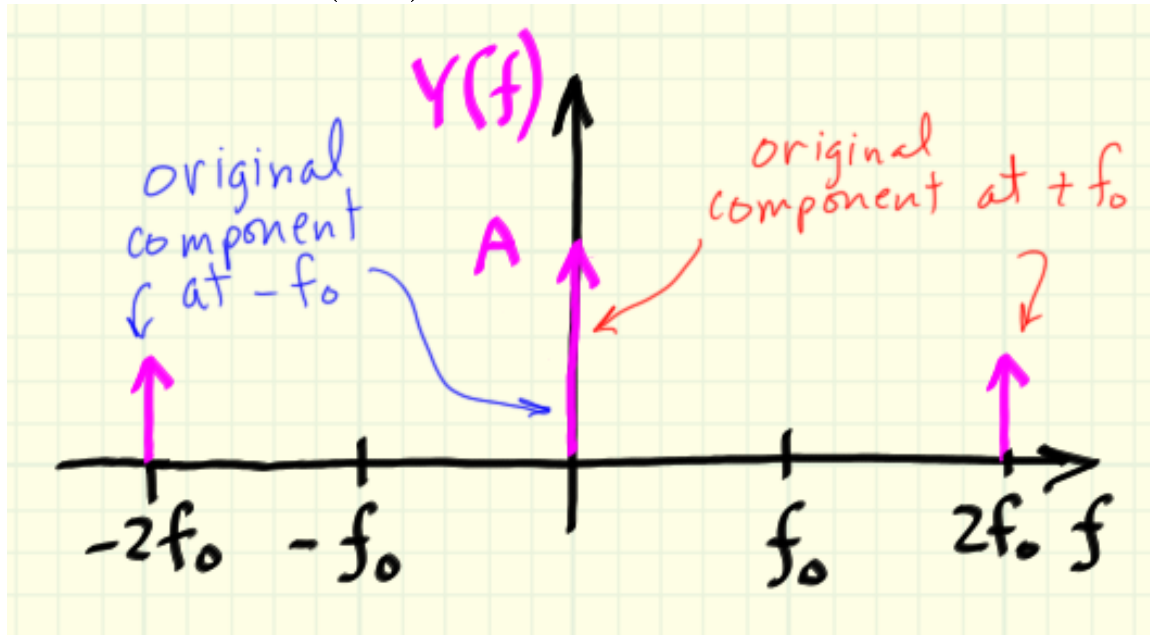
Solution to Exercise 6.1 (p. 85)



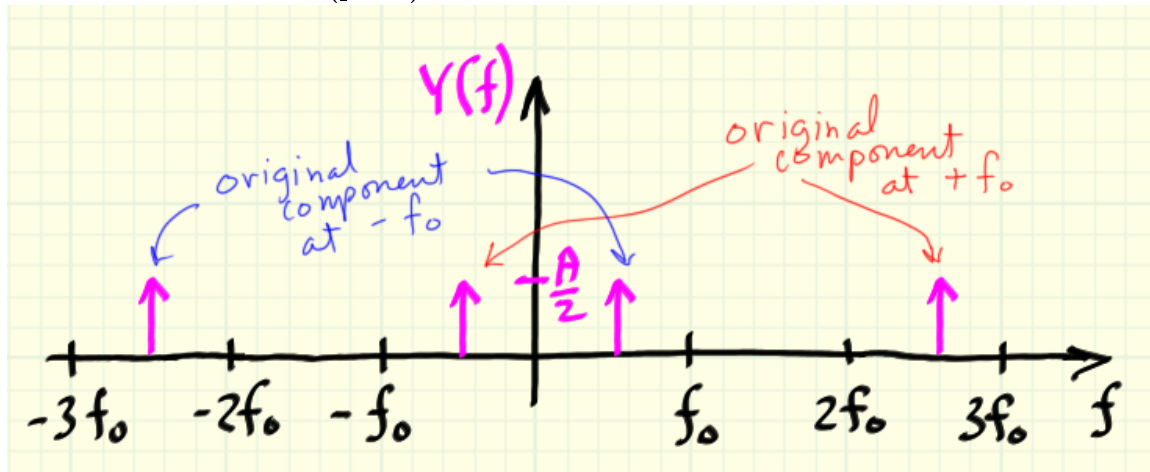
Solution to Exercise 6.2 (p. 85)



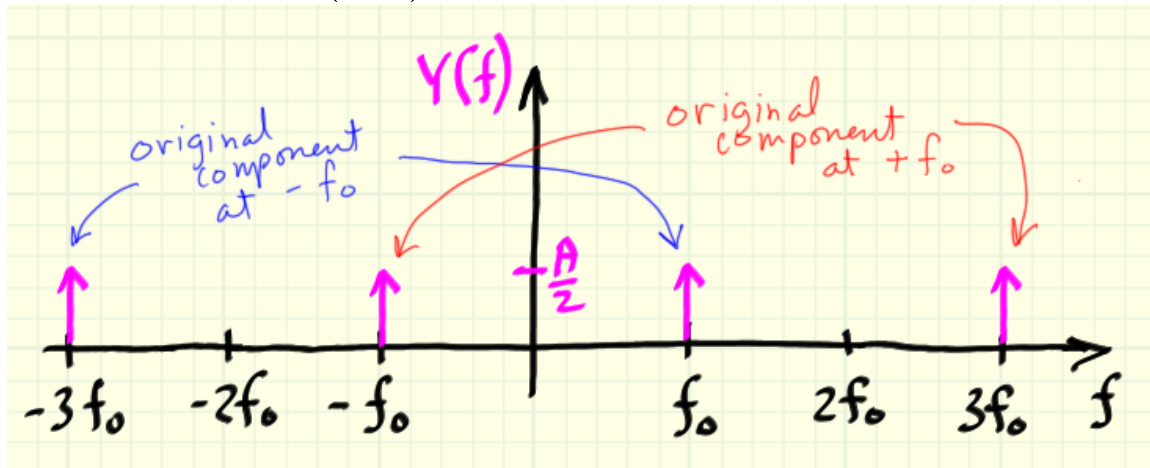
Solution to Exercise 6.3 (p. 85)



Solution to Exercise 6.4 (p. 85)



Solution to Exercise 6.5 (p. 85)



Chapter 7

Additive Synthesis

7.1 Additive Synthesis Techniques¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	<ul style="list-style-type: none">• Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none">• Get started with LabVIEW• Obtain a fully-functional evaluation edition of LabVIEW

Table 7.1

7.1.1 Overview

Additive synthesis creates complex sounds by adding together individual sinusoidal signals called **partials**. The prerequisite module Additive Synthesis Concepts (Section 7.2) reviews the main concepts of additive synthesis. In this module you will learn how to synthesize audio waveforms by designing the frequency and amplitude trajectories of the partials. Also, LabVIEW programming techniques for additive synthesis will be introduced in two examples.

7.1.2 Frequency and Amplitude Trajectory Design

A **partial** is the fundamental building block of additive synthesis. A partial is a single sinusoidal component whose amplitude and frequency are each time-varying. The time-varying amplitude denoted $a(t)$ is called the **amplitude trajectory** and the time-varying frequency denoted $f(t)$ is called the **frequency trajectory**. Additive synthesis requires the design of both trajectories for each partial; the partials are then summed together to create the sound.

The screencast video of Figure 7.1 shows how to begin the design of a sound as a spectrogram plot, how to design the amplitude trajectory first as an intensity (loudness) trajectory in "log space" using decibels, and how to design the frequency trajectory in "log space" using octaves. Designing the partials in log space accounts for hearing perception which is logarithmic in both intensity and in frequency; refer to Perception of Sound (Section 2.1) for a detailed treatment of this subject.

¹This content is available online at <<http://cnx.org/content/m15445/1.3/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

Image not finished

Figure 7.1: [video] Design of frequency and amplitude trajectories

7.1.3 Example 1: Fractal Partial

In this first example, partials are created during a fixed time interval and then concatenated to create the overall sound. During the first time interval a single partial is created at a reference frequency. During the second time interval the partial's frequency linearly increases in "octave space" from the reference frequency to a frequency two octaves above the reference frequency. In the third interval the partial bifurcates into two partials, where one increases by an octave and the other decreases by an octave. In the fourth interval, each of the two partials bifurcates again to make a total of four partials, each increasing or decreasing by half an octave. This behavior repeats in each subsequent time interval, doubling the number of partials, and halving the amount of frequency increase or decrease.

The screencast video of Figure 7.2 shows how the frequency trajectories are designed in "octave space", and then reviews the key LabVIEW programming techniques needed to implement this design. The video also includes an audio demonstration so you can hear the design of this "audible fractal."



The LabVIEW VI demonstrated within the video is available here: [genfnc.zip](#)³. This VI requires installation of the TripleDisplay⁴ front-panel indicator.

Image not finished

Figure 7.2: [video] Design of the "audible fractal," LabVIEW implementation, and audio demonstration

7.1.4 Example 2: Spectrogram Art

The design of a sound using additive synthesis typically begins with a spectrogram representation of the desired sound. In this second example, straight line segments define the frequency trajectories of nine distinct partials that create a spectrum of a recognizable object, specifically, a cartoon drawing of an individual who is happy to be wearing a French beret.

The screencast video of Figure 7.3 shows how the frequency trajectories are designed in "octave space" and specified according to the coordinates of the line segment endpoints. The design of the corresponding amplitude trajectories necessary to make the partials start and stop at the correct times is likewise discussed. Key LabVIEW programming techniques needed to implement this design and an audio demonstration are also presented.

³See the file at <http://cnx.org/content/m15445/latest/genfnc.zip>

⁴"[LabVIEW application] TripleDisplay" <http://cnx.org/content/m15430/latest/>



The LabVIEW VI demonstrated within the video is available here: [face.zip](#)⁵. This VI requires installation of the TripleDisplay⁶ front-panel indicator.

Image not finished

Figure 7.3: [video] Design of the cartoon face, LabVIEW implementation, and audio demonstration

7.2 Additive Synthesis Concepts⁷

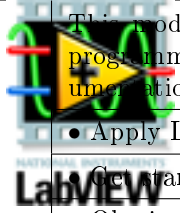
	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide⁸ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p>
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 7.2

7.2.1 Overview

Additive synthesis creates complex sounds by adding together individual sinusoidal signals called **partials**. A partial's frequency and amplitude are each time-varying functions, so a partial is a more flexible version of the **harmonic** associated with a Fourier series decomposition of a periodic waveform.

In this module you will learn about partials, how to model the timbre of natural instruments, various sources of control information for partials, and how to make a sinusoidal oscillator with an instantaneous frequency that varies with time.

7.2.2 Partials

A **partial** is a generalization of the **harmonic** associated with a Fourier series representation of a periodic waveform. The screencast video of Figure 7.4 introduces important concepts associated with partials.

⁵See the file at <http://cnx.org/content/m15445/latest/face.zip>

⁶"[LabVIEW application] TripleDisplay" <http://cnx.org/content/m15430/latest/>

⁷This content is available online at <http://cnx.org/content/m15444/1.2/>.

⁸"NI LabVIEW Getting Started FAQ" <http://cnx.org/content/m15428/latest/>

Image not finished

Figure 7.4: [video] Important concepts associated with **partials**

7.2.3 Modeling Timbre of Natural Instruments

Perception of an instrument's **timbre** relies heavily on the attack transient of a note. Since partials can effectively enter and leave the signal at any time, additive synthesis is a good way to model physical instruments. The screencast video of Figure 7.5 discusses three important design requirements for partials necessary to successfully model a physical instrument.

Image not finished

Figure 7.5: [video] Three design requirements for partials to model physical instruments

7.2.4 Sources of Control Information

The time-varying frequency of a partial is called its **frequency trajectory**, and is best visualized as a track or path on a spectrogram display. Similarly, the time-varying amplitude of a partial is called its **amplitude trajectory**. Control information for these trajectories can be derived from a number of sources. Perhaps the most obvious source is a spectral analysis of a physical instrument to be modeled. The screencast video of Figure 7.6 discusses this concept and demonstrates how a trumpet can be well-modeled by adding suitable partials.



The code for the LabVIEW VI demonstrated within the video is available here: trumpet.zip⁹. This VI requires installation of the TripleDisplay¹⁰ front-panel indicator.

Image not finished

Figure 7.6: [video] Modeling a trumpet tone by adding together increasing numbers of partials

Control information can also be derived from other domains not necessarily related to music. The screencast video of Figure 7.7 provides some examples of non-music control information, and illustrates

⁹<http://cnx.org/content/m15444/latest/trumpet.zip>

¹⁰"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

how an edge boundary from an image can be "auralized" by translating the edge into a partial's frequency trajectory.

Image not finished

Figure 7.7: [video] Skyline of Houston translated into a frequency trajectory

7.2.5 Instantaneous Frequency

The frequency trajectory of a partial is defined as a time-varying frequency $f(t)$. Since a constant-frequency and constant-amplitude sinusoid is mathematically described as $y(t) = A\sin(2\pi f_0 t)$, intuition perhaps suggests that the partial should be expressed as $y(t) = a(t)\sin(2\pi f(t)t)$, where $a(t)$ is the amplitude trajectory. However, as shown in the screencast video of Figure 7.8 this intuitive result is incorrect; the video derives the correct equation to describe a partial in terms of its trajectories $f(t)$ and $a(t)$.

Image not finished

Figure 7.8: [video] Derivation of the equation of a partial given its frequency and amplitude trajectories

7.3 [mini-project] Risset Bell Synthesis¹¹


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹² module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 7.3

7.3.1 Objective

Additive synthesis builds up complex sounds from simple sounds (sinusoids). Additive synthesis implies more than just doing Fourier series, though: each sinusoidal component is assigned its own frequency and amplitude trajectory (resulting in a partial), so complex, time-varying sounds can be generated by summing these partials together.

¹¹This content is available online at <<http://cnx.org/content/m15476/1.1/>>.

¹²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

In this project, use additive synthesis to emulate the sound of a bell using a technique described by Jean-Claude Risset, an early pioneer in **computer music**.

7.3.2 Prerequisite Modules

If you have not done so already, please study the pre-requisite modules Additive Synthesis Concepts (Section 7.2) and Additive Synthesis Techniques (Section 7.1). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing¹³ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

7.3.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

7.3.4 Description of the Risset Bell

Jean-Claude Risset's book *Introductory Catalogue of Computer-Synthesized Sounds (Bell Telephone Laboratories, 1969)* includes an additive synthesis method for a bell-like sound. After analyzing the ringing characteristics of physical bells, he determined that a bell-like tone could be created by using non-harmonic partials whose decay times are approximately inversely proportional to frequency. In addition, pairs of low-frequency partials with a slight frequency offset create a "beating" effect.

The partials for the Risset bell are available in this CSV-type spreadsheet: `risset_bell.csv`¹⁴. The columns are:

1. partial number
2. intensity in dB
3. duration multiplier (indicates fraction of overall duration)
4. frequency multiplier (indicates interval ratio from base frequency)
5. frequency offset (in Hz)

7.3.5 Part 1: Attack/Decay Envelope Generator

Create an attack/decay intensity envelope composed of two straight-line segments as shown in Figure 7.9:

¹³*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
 <<http://cnx.org/content/col10440/latest/>>

¹⁴http://cnx.org/content/m15476/latest/risset_bell.csv

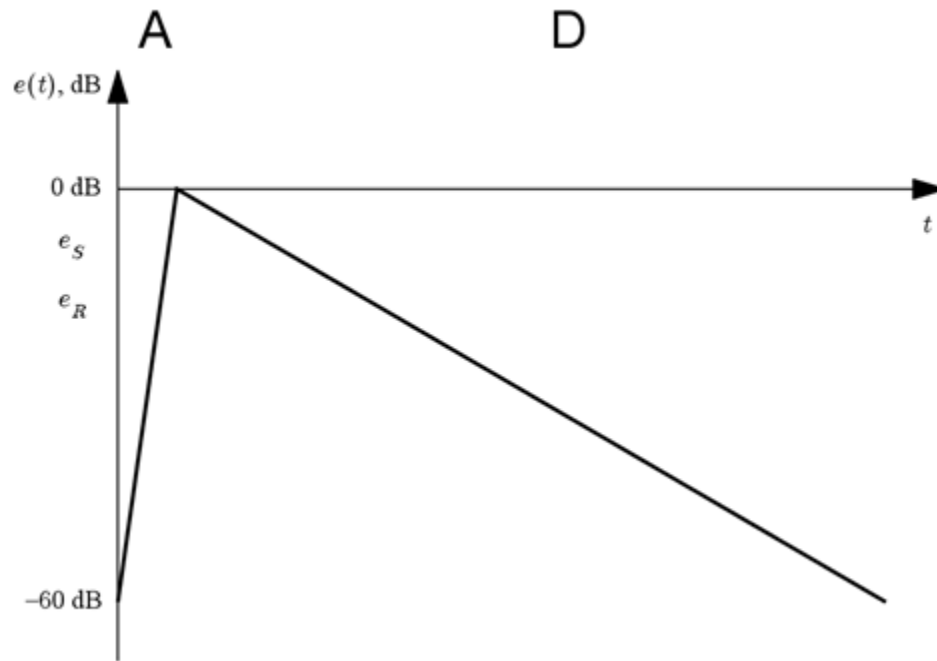


Figure 7.9: Attack/decay envelope specification

The value "-60 dB" corresponds to an amplitude of 0.001, effectively silence. The total duration of the envelope should be based on a front-panel control called "duration [s]" (its unit is seconds), and another control called "attack [ms]" (its unit is milliseconds).

Convert the intensity envelope into an amplitude envelope by "undoing" the equation for decibels.

7.3.6 Part 2: Multiple Envelopes

Enclose your envelope generator in a for-loop structure which takes values from `risset_bell.csv`¹⁵. Connect a control to the count terminal so that you can adjust how many envelopes to view.

Use "auto indexing" on the for-loop to create an array of envelopes, then view the array as graph (you will see all of the envelopes superimposed on the same graph, with each envelope a different color). Confirm that your envelopes have the correct amplitudes and durations.

Plot both the intensity envelopes and amplitude envelopes for all partials, and include these plots with your deliverables.

7.3.7 Part 3: Sinusoidal Tone Generator

Add a sinusoidal tone generator to your main loop. The base frequency should be set by a front-panel control called "freq [Hz]", and the frequency multiplier and offset for each partial should be used to set the actual frequency. Also include a front-panel control to select the system sampling frequency "fs [Hz]".

¹⁵http://cnx.org/content/m15476/latest/risset_bell.csv

Apply the amplitude envelope, and then add the partial to the sound that was generated on the previous pass of the loop. Regardless of whether you use a shift register or a feedback node, initialize the audio signal with an array of zeros which is of the same length as your envelopes.

Remember to use the "QuickScale" built-in subVI on the finished audio waveform to ensure that all of its values lie in the range ± 1 .

7.3.8 Part 4: Experiment with Parameters

Your front panel controls should include the following adjustable parameters: number of partials, total duration, base frequency, sampling frequency, and attack time. Listen to the audio and view the spectrogram as you adjust the parameters.

To get started, listen to the sound with the first partial only, and then with the first two partials, and so on until you hear all eleven partials. Building up the sound from silence by adding more and more partials is the essence of "additive synthesis."

Try varying the attack time. What is the maximum attack time that still sounds like the striking of a bell?

Try adjusting the total duration and base frequency. Remember to adjust the sampling frequency high enough so that you do not produce aliasing. If the sampling frequency is too high, however, all of the partials will compress into the bottom of the spectrogram plot.

Overall, what values produce a realistic-sounding bell?

Create .wav files for three distinct sets of parameters, and discuss the motivation for your choices. Include a spectrogram plot for each of the three bell sounds.

7.4 [mini-project] Spectrogram Art¹⁶


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ¹⁷ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 7.4

7.4.1 Objective

Additive synthesis builds up complex sounds from simple sounds (sinusoids). Additive synthesis implies more than just doing Fourier series, though: each sinusoidal component is assigned its own frequency and amplitude trajectory (resulting in a partial), so complex, time-varying sounds can be generated by summing these partials together.

In this project you will create an oscillator whose output tracks a specified amplitude and frequency trajectory. With this general-purpose oscillator you can define multiple frequency/amplitude trajectories that can be combined to create complex sounds. In particular, you will design the sound so that its spectrogram makes a recognizable picture!

¹⁶This content is available online at <http://cnx.org/content/m15446/1.1/>.

¹⁷"NI LabVIEW Getting Started FAQ" <http://cnx.org/content/m15428/latest/>

7.4.2 Prerequisite Modules

If you have not done so already, please study the prerequisite modules Additive Synthesis Concepts (Section 7.2) and Additive Synthesis Techniques (Section 7.1). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing¹⁸ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

7.4.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

7.4.4 Part 1: General-Purpose Sinusoidal Oscillator

Develop a subVI called **gposc.vi** that accepts a frequency trajectory (in Hz), an amplitude trajectory, and a sampling frequency (in Hz) to produce a sinusoidal output whose amplitude and frequency tracks the two input trajectories, respectively. The two trajectories are arrays that should be of the same length.

Demonstrate that your oscillator works properly by showing the output of your VI (spectrogram and .wav file) for the amplitude and frequency trajectories produced by a LabVIEW MathScript node that contains the following code:

```
ff=[linspace(200,1600,2.5*fs) ...
    linspace(1600,800,1.5*fs)];

aa=[linspace(1,0,3*fs) ...
    linspace(0,0.75,fs)];
```

where **fs** is the sampling frequency in Hz, **ff** is the output frequency trajectory (also in Hz), and **aa** is the amplitude trajectory (between 0 and 1). Use a sampling frequency of 5 kHz when you make the spectrogram and soundfile.

Plot the trajectories **ff** and **aa** and compare to your spectrogram.

Remember, the instantaneous frequency of your general-purpose sinusoidal oscillator is related to the time-varying phase of the sine function. That is, if the sinusoidal signal is defined as $y(t) = \sin(\theta(t))$, then the instantaneous frequency of the sinusoid is $\omega(t) = d\theta(t)/dt$ radians per second. Because you are given a frequency trajectory that relates to $\omega(t)$, which mathematical operation yields the phase function $\theta(t)$?

Here's a LabVIEW coding tip: You will find the built-in VI "Mathematics | Integ and Diff | Integral x(t)" to be essential for this part of the project.

7.4.5 Part 2: Frequency Trajectory Design

You can make your spectrogram art project sound more musically appealing when you design the frequency trajectories to account for frequency **perception**; refer to Perception of Sound (Section 2.1) for a detailed treatment of this subject. Design your trajectories in "log space" (using logarithmic graph paper) and then convert to actual frequency just before invoking your general-purpose sinusoidal oscillator.

Review Additive Synthesis Techniques (Section 7.1) to learn how to create your frequency trajectories for this part of the project.

¹⁸*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
 <<http://cnx.org/content/col10440/latest/>>

7.4.6 Part 3: Amplitude Trajectory Design

The discussion of Part 2 pertains to the design of your amplitude trajectories, as well. Perception of intensity (loudness) is also logarithmic (refer to Perception of Sound (Section 2.1) and review the section on intensity perception). In this part you will design your amplitude trajectory in "log space," but now using traditional decibels (dB). An intensity trajectory can be converted to amplitude by "undoing" the equation that relates a value to the same value expressed in decibels: $X_{\text{dB}} = 20\log_{10}(X)$.

Experiment with your spectrogram display device to learn the intensity-to-color mapping. Specifically, you could produce a sinusoidal signal with increasing intensity values over time, then match up the plotted colors to the known intensity values.

7.4.7 Part 4: Spectrogram Art

Design a spectrogram picture using multiple frequency/amplitude trajectories. Include your paper-and-pencil drawing of the spectrogram as part of your deliverables. Use your creativity to make an interesting and recognizable picture.

Better designs will go beyond straight lines to include curved lines such as arcs, exponentials, parabolas, sinusoids, polynomials, spline interpolations, and so on.

Include a .wav file of the sound associated with your spectrogram picture.

Chapter 8

Subtractive Synthesis

8.1 Subtractive Synthesis Concepts¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	<ul style="list-style-type: none">• Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none">• Obtain a fully-functional evaluation edition of LabVIEW

Table 8.1

8.1.1 Subtractive Synthesis Model

Subtractive synthesis describes a wide range of synthesis techniques that apply a filter (usually time-varying) to a wideband excitation source such as noise or a pulse train. The filter shapes the wideband spectrum into the desired spectrum. The excitation/filter technique describes the sound-producing mechanism of many types of physical instruments as well as the human voice, making subtractive synthesis an attractive method for **physical modeling** of real instruments.

Figure 8.1 illustrates the general model of subtractive synthesis. The excitation source parameters may include amplitude, bandwidth, and frequency (for repetitive pulse train input), and each parameter may change independently as a function of time. The filter parameters likewise may vary with time, and include center frequency, bandwidth, and resonant gain.

¹This content is available online at <<http://cnx.org/content/m15456/1.1/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

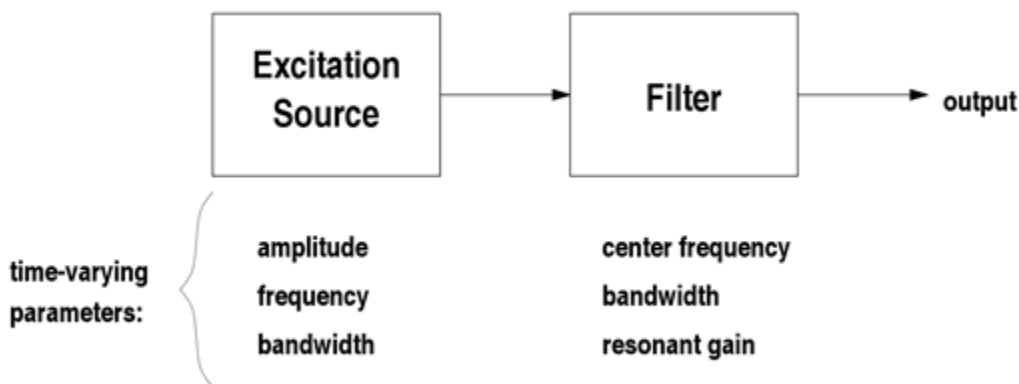


Figure 8.1: Model of the subtractive synthesis process

8.1.2 Excitation Sources

Excitation sources for subtractive synthesis must be **wideband** in nature, i.e., they must contain significant spectral energy over a wide frequency range. A **white noise** source is an idealized source that contains constant energy over all frequencies. Practical noise sources do not have infinite bandwidth but can create uniform spectral energy over a suitable range such as the audio spectrum.

Random number generators form the basis of a variety of noise sources on digital computers. For example, the LabVIEW "Signal Processing" palette contains the following noise sources: uniform, Gaussian, random, gamma, Poisson, binomial, and Bernoulli.

A **pulse train** is a repetitive series of pulses. It provides an excitation source that has a perceptible pitch, so in a sense the excitation spectrum is "pre-shaped" before applying it to a filter. Many types of musical instruments use some sort of pulse train as an excitation, notably wind instruments such as brass (trumpet, trombone, tuba) and woodwinds (clarinet, saxophone, oboe, bassoon). For example, consider the trumpet and its mouthpiece (Figure 8.2 and Figure 8.3, respectively). Listen to the "buzzing" sound of the trumpet mouthpiece alone `trumpet_mouthpiece.wav`³, and compare it to the mouthpiece plus trumpet `trumpet.wav`⁴. Figure 8.4 compares the time-domain waveform and frequency spectrum of each sound. Both sounds are the same pitch, so the same harmonics are visible in each. However, the mouthpiece buzz contains more spectral energy at higher frequencies.

³http://cnx.org/content/m15456/latest/trumpet_mouthpiece.wav

⁴<http://cnx.org/content/m15456/latest/trumpet.wav>



Figure 8.2: Trumpet instrument, a member of the brass family (click picture for larger image)

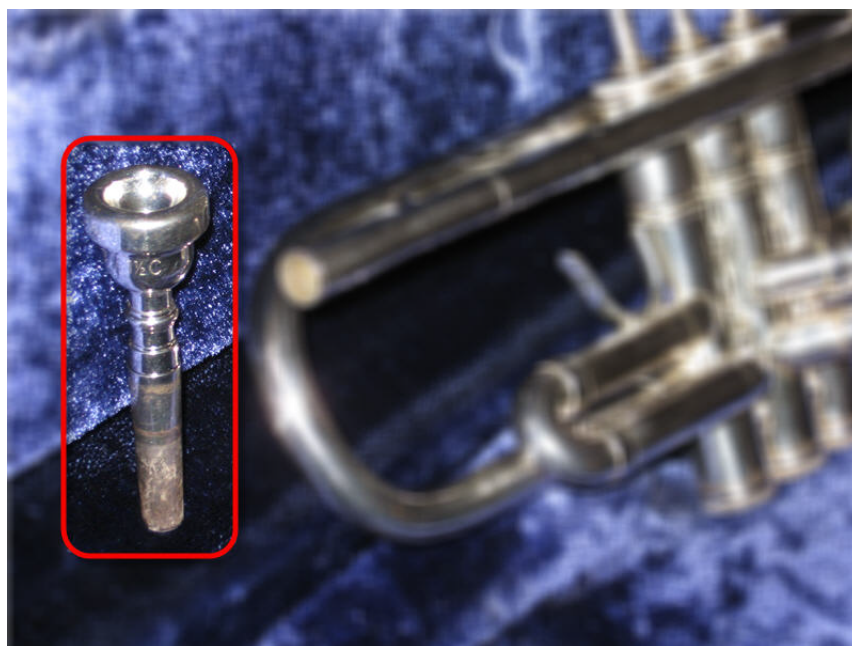


Figure 8.3: Trumpet mouthpiece, approximately 3 inches in length (click picture for larger image)

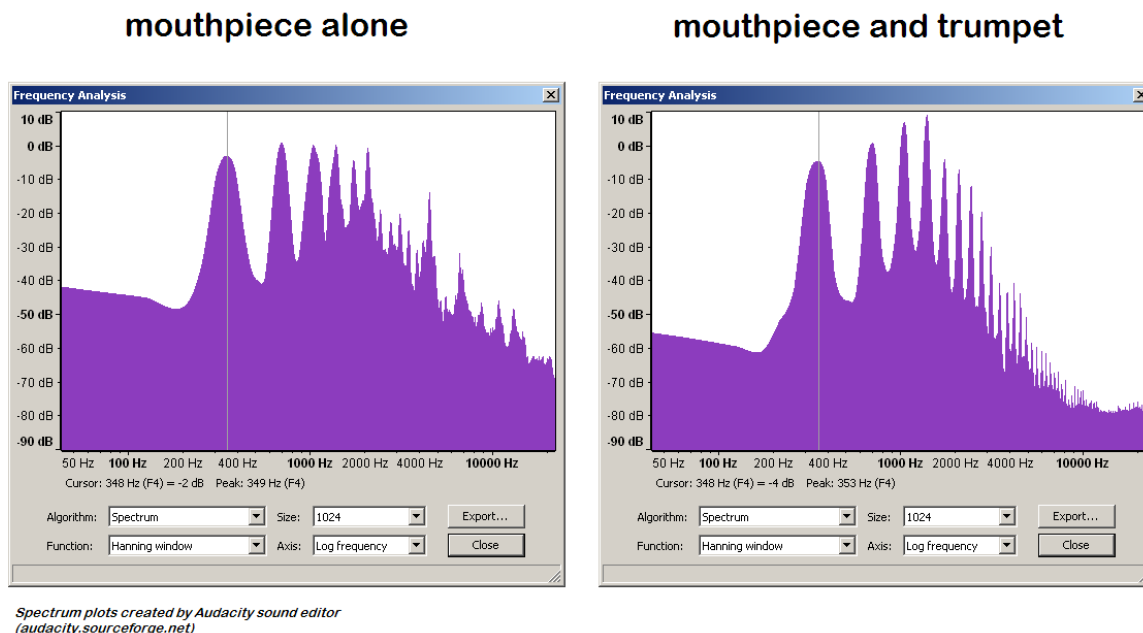


Figure 8.4: Spectra of the trumpet mouthpiece alone and mouthpiece-plus-trumpet signals (click picture for larger image)

8.1.3 Time-Varying Digital Filters

Time-varying digital filters are typically implemented with **block processing**, in which an input signal is subdivided into short blocks (also called **frames**) for filtering. Each frame is processed by a constant-coefficient digital filter. However, the constants change from one frame to the next, thereby creating the effect of a time-varying filter.

The choice of **frame length** involves a trade-off between the rate at which the filter coefficients must change and the amount of time required for the filter's transient response. Higher-order filters require more time to reach steady state, and the frame length should be no shorter than the length of the filter's impulse response.

Digital filters may be broadly classified as either **finite impulse response (FIR)** or **infinite impulse response (IIR)**. The latter type is preferred for most implementations (especially for real-time filtering) because IIR filters have many fewer coefficients than comparable FIR filters. However, IIR filters have the disadvantage of potential stability problems, especially when finite-precision calculations are used.

The digital filter coefficients usually are calculated independently for each frame. That is, it is generally not possible to calculate only two sets of filter coefficients and then interpolate in between. For example, suppose a digital filter is required to have a cutoff frequency that varies anywhere from 100 Hz to 5,000 Hz. Ideally one would be able to calculate a set of filter coefficients for the 100 Hz filter and another set for the 5,000 Hz filter, and then use linear interpolation to determine the coefficients for any intermediate frequency, i.e., 650 Hz. Unfortunately the interpolation technique does not work. For off-line or batch-type processing, filter coefficients can be computed for each frame. For real-time implementation, the filter coefficients must be pre-computed and stored in a lookup table for fast retrieval.



Download and run the LabVIEW VI `filter_coeffs.vi`⁵. This VI illustrates why it is generally not possible to interpolate filter coefficients between blocks. Try this: increase the "low cutoff frequency" slider and observe the values of the coefficients. Some coefficients vary monotonically (such as `a[1]`), but others such as `a[2]` decrease and then increase again. Still others such as the "b" coefficients remain at a constant level and then begin increasing. You can also try different filter types (highpass, bandpass, bandstop) and filter orders.

8.2 Interactive Time-Varying Digital Filter in LabVIEW⁶


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ⁷ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none"> • Get started with LabVIEW • Obtain a fully-functional evaluation edition of LabVIEW

Table 8.2

A time-varying digital filter can easily be implemented in LabVIEW. The screencast video of Figure 8.5 walks through the complete process to develop a digital filter that operates in real-time and responds to parameter changes from the front panel controls. The video includes an audio demonstration of the finished result and discusses practical issues such as eliminating clicks in the output signal.



Download the source code for the finished VI: `filter_rt.vi`⁸. Refer to TripleDisplay⁹ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 8.5: [video] Building an interactive real-time digital filter in LabVIEW

⁵http://cnx.org/content/m15456/latest/filter_coeffs.vi

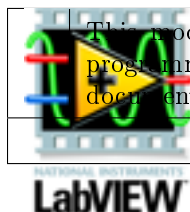
⁶This content is available online at <http://cnx.org/content/m15477/1.2/>.

⁷"NI LabVIEW Getting Started FAQ" <http://cnx.org/content/m15428/latest/>

⁸http://cnx.org/content/m15477/latest/filter_rt.vi

⁹"[LabVIEW application] TripleDisplay" <http://cnx.org/content/m15430/latest/>

8.3 Band-Limited Pulse Generator¹⁰

	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹¹ module for tutorials and documentation that will help you:</p>
	<p><i>continued on next page</i></p>

¹⁰This content is available online at <<http://cnx.org/content/m15457/1.3/>>.

• Apply LabVIEW to Audio Signal Processing
• Get started with LabVIEW
• Obtain a fully-functional evaluation edition of LabVIEW

Table 8.3

8.3.1 Introduction

Subtractive synthesis techniques apply a filter (usually time-varying) to a wideband excitation source such as noise or a pulse train. The filter shapes the wideband spectrum into the desired spectrum. The excitation/filter technique describes the sound-producing mechanism of many types of physical instruments as well as the human voice, making subtractive synthesis an attractive method for **physical modeling** of real instruments.

A **pulse train**, a repetitive series of pulses, provides an excitation source that has a perceptible pitch, so in a sense the excitation spectrum is "pre-shaped" before applying it to a filter. Many types of musical instruments use some sort of pulse train as an excitation, notably wind instruments such as brass (e.g., trumpet, trombone, and tuba) and woodwinds (e.g., clarinet, saxophone, oboe, and bassoon). Likewise, the human voice begins as a series of pulses produced by vocal cord vibrations, which can be considered the "excitation signal" to the vocal and nasal tract that acts as a resonant cavity to amplify and filter the "signal."

Traditional rectangular pulse shapes have significant spectral energy contained in harmonics that extend beyond the **folding frequency** (half of the sampling frequency). These harmonics are subject to **aliasing**, and are "folded back" into the **principal alias**, i.e., the spectrum between 0 and $f_s/2$. The aliased harmonics are distinctly audible as high-frequency tones that, since undesired, qualify as noise.

The **band-limited pulse**, however, is free of aliasing problems because its maximum harmonic can be chosen to be below the folding frequency. In this module the mathematics of the band-limited pulse are developed, and a band-limited pulse generator is implemented in LabVIEW.

8.3.2 Mathematical Development of the Band-Limited Pulse

By definition, a **band-limited pulse** has zero spectral energy beyond some determined frequency. You can use a truncated Fourier series to create a series of harmonics, or sinusoids, as in (8.1):

$$x(t) = \sum_{k=1}^N \sin(2\pi k f_0 t) \quad (8.1)$$

The Figure 8.6 screencast video shows how to implement (8.1) in LabVIEW by introducing the "Tones and Noise" built-in subVI that is part of the "Signal Processing" palette. The video includes a demonstration that relates the time-domain pulse shape, spectral behavior, and audible sound of the band-limited pulse.



Download the finished VI from the video: blp_demo.vi¹². This VI requires installation of the TripleDisplay¹³ front-panel indicator.

¹¹"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

¹²http://cnx.org/content/m15457/latest/blp_demo.vi

¹³"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

Image not finished

Figure 8.6: [video] Band-limited pulse generator in LabVIEW using "Tones and Noise" built-in subVI

The truncated Fourier series approach works fine for off-line or batch-mode signal processing. However, in a real-time application the computational cost of generating individual sinusoids becomes prohibitive, especially when a fairly dense spectrum is required (for example, 50 sinusoids).

A closed-form version of the truncated Fourier series equation is presented in (8.2) (refer to Moore in "References" section below):

$$x(t) = \sum_{k=1}^N \sin(k\theta) = \sin\left[(N+1)\frac{\theta}{2}\right] \frac{\sin\left(N\frac{\theta}{2}\right)}{\sin\left(\frac{\theta}{2}\right)} \quad (8.2)$$

where

$\theta = 2\pi f_0 t$. The closed-form version of the summation requires only three sinusoidal oscillators yet can produce an arbitrary number of sinusoidal components.

Implementing (8.2) contains one significant challenge, however. Note the ratio of two sinusoids on the far right of the equation. The denominator sinusoid periodically passes through zero, leading to a divide-by-zero error. However, because the numerator sinusoid operates at a frequency that is N times higher, the numerator sinusoid also approaches zero whenever the lower-frequency denominator sinusoid approaches zero. This "0/0" condition converges to either N or -N; the sign can be inferred by looking at adjacent samples.

8.3.3 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.

8.4 Formant (Vowel) Synthesis¹⁴


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹⁵ module for tutorials and documentation that will help you:</p> <ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing <p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW
---	---

Table 8.4

¹⁴This content is available online at <<http://cnx.org/content/m15459/1.2/>>.

¹⁵"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

8.4.1 Introduction

Speech and singing contain a mixture of voiced and un-voiced sounds. Voiced sounds associate with the vowel portions of words, while unvoiced sounds are produced when uttering consonants like "s." The spectrum of a voiced sound contains characteristic resonant peaks called **formants**, and are the result of frequency shaping produced by the **vocal tract** (mouth as well as nasal passage), a complex time-varying resonant cavity.

In this module, a **formant synthesizer** is developed and implemented in LabVIEW. The subtractive synthesis model of a wideband excitation source shaped by a digital filter is applied here. The filter is implemented as a set of parallel two-pole resonators (bandpass filters) that filter a band-limited pulse. Refer to the modules Subtractive Synthesis Concepts (Section 8.1) and Band-Limited Pulse Generator (Section 8.3) for more details.

8.4.2 Formant Synthesis Technique

The Figure 8.7 screencast video develops the general approach to formant synthesis:

Image not finished

Figure 8.7: [video] Formant synthesis technique

The mathematics of the band-limited pulse generator and its LabVIEW implementation are presented in the module Band-Limited Pulse Generator.

The two-pole resonator is an IIR (infinite impulse response) digital filter defined by (8.3) (see Moore in the "References" section for additional details):

$$H(z) = (1 - R) \frac{1 - Rz^{-2}}{1 - 2R\cos\theta z^{-1} + R^2 z^{-2}} \quad (8.3)$$

where $\theta = 2\pi(f_C/f_S)$, $R = e^{-\pi B/f_S}$, f_C is the center frequency, B is the bandwidth, and f_S is the sampling frequency, all in units of Hz.

The Figure 8.8 screencast video shows how to create a subVI that implements the two-pole resonator.

Image not finished

Figure 8.8: [video] Implementing the two-pole resonator in LabVIEW

8.4.3 Formants for Selected Vowel Sounds

Peterson and Barney (see "References" section) have compiled a list of formant frequencies for common vowels in American English; refer to Figure 8.9:

Phonetic Symbol	Example Word	F_1 (Hz)	F_2 (Hz)	F_3 (Hz)
/ow/	bought	570	840	2410
/oo/	boot	300	870	2240
/u/	foot	440	1020	2240
/a/	hot	730	1090	2440
/uh/	but	520	1190	2390
/er/	bird	490	1350	1690
/æ/	bat	660	1720	2410
/e/	bet	530	1840	2480
/i/	bit	390	1990	2550
/iy/	beet	270	2290	3010

Figure 8.9: Formant frequencies for common vowels in American English (from Peterson and Barney, 1952)

8.4.4 Formant Synthesizer

The previous sections have laid out all of the pieces you need to create your own formant synthesizer. See if you can set up a LabVIEW VI that pulls the pieces together. The Figure 8.10 screencast video shows how your finished design might operate. The video also discusses how to choose the relative formant amplitudes and bandwidths, as well as the BLP source parameters.

Image not finished

Figure 8.10: [video] Formant synthesis LabVIEW VI

8.4.5 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Peterson, G.E., and H.L. Barney, "Control Methods Used in a Study of the Vowels," Journal of the Acoustical Society of America, vol. 24, 1952.

8.5 Linear Prediction and Cross Synthesis¹⁶

¹⁶This content is available online at <<http://cnx.org/content/m15478/1.2/>>.


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide¹⁷ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none"> • Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 8.5

8.5.1 Introduction

Subtractive synthesis methods are characterized by a wideband excitation source followed by a time-varying filter. **Linear prediction** provides an effective way to estimate the time-varying filter coefficients by analyzing an existing music or speech signal. **Linear predictive coding (LPC)** is one of the first applications of linear prediction to the problem of speech compression. In this application, a speech signal is modelled as a time-varying digital filter driven by an **innovations sequence**. The LPC method identifies the filter coefficients by minimizing the prediction error between the filter output and the original signal. Significant compression is possible because the innovations sequence and filter coefficients require less space than the original signal.

Cross synthesis is a musical adaptation of the speech compression technique. A musical instrument or speech signal serves as the original signal to be analyzed. Once the filter coefficients have been estimated, the innovations sequence is discarded and another signal is used as the filter excitation. For example, a "singing guitar" effect is created by deriving the filter coefficients from a speech signal and driving the filter with the sound of an electric guitar; listen to the audio clips below:

- speech.wav¹⁸ – Speech signal for digital filter coefficients (audio courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech¹⁹; the sentences are two of the many phonetically balanced **Harvard Sentences**, an important standard for the speech processing community)
- eguitar.wav²⁰ – Electric guitar signal for filter input
- speech_eguitar.wav²¹ – Cross-synthesized result (filter output)

8.5.2 Linear Prediction Theory

The Figure 8.11 screencast video develops the theory behind linear prediction and describes how the technique is applied to musical signals. Practical issues such as choosing the filter block size and filter order are also discussed.

Image not finished

Figure 8.11: [video] Theory of linear prediction and cross synthesis

¹⁷"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

¹⁸<http://cnx.org/content/m15478/latest/speech.wav>

¹⁹http://www.voiptroubleshooter.com/open_speech

²⁰<http://cnx.org/content/m15478/latest/eguitar.wav>

²¹http://cnx.org/content/m15478/latest/speech_eguitar.wav

8.5.3 Linear Prediction Implementation

The previous section explains how you can use the all-pole filter model to implement cross synthesis. But how are the all-pole filter coefficients actually created?

The LabVIEW MathScript feature includes the function **lpc**, which accepts a signal (1-D vector or array) and the desired filter order and returns an array of filter coefficients. For details, select "Tools | MathScript Window" and type `help lpc`.

8.6 [mini-project] Linear Prediction and Cross Synthesis²²

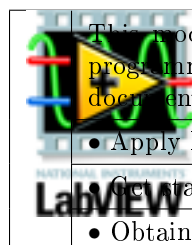
	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide²³ module for tutorials and documentation that will help you:</p> <ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing • Get started with LabVIEW • Obtain a fully-functional evaluation edition of LabVIEW
---	--

Table 8.6

8.6.1 Objective

Linear prediction is a method used to estimate a time-varying filter, often as a model of a vocal tract. Musical applications of linear prediction substitute various signals as excitation sources for the time-varying filter.

This mini-project will give you chance to develop the basic technique for computing and applying a time-varying filter. Next, you will experiment with different excitation sources and linear prediction model parameters. Finally, you will learn about cross-synthesis.

8.6.2 Prerequisite Modules

If you have not done so already, please study the prerequisite modules Linear Prediction and Cross Synthesis (Section 8.5). If you are relatively new to LabVIEW, consider taking the course LabVIEW Techniques for Audio Signal Processing²⁴ which provides the foundation you need to complete this mini-project activity, including working with arrays, creating subVIs, playing an array to the soundcard, and saving an array as a .wav sound file.

8.6.3 Deliverables

- All LabVIEW code that you develop (block diagrams and front panels)
- All generated sounds in .wav format
- Any plots or diagrams requested
- Summary write-up of your results

²²This content is available online at <http://cnx.org/content/m15479/1.1/>.

²³"NI LabVIEW Getting Started FAQ" <http://cnx.org/content/m15428/latest/>

²⁴*Musical Signal Processing with LabVIEW – Programming Techniques for Audio Signal Processing*
<http://cnx.org/content/col10440/latest/>

8.6.4 Part 1: Framing and De-Framing

Time-varying filters operate by applying a fixed set of coefficients on short blocks (or "frames") of the signal; the coefficients are varied from one frame to the next. In this part you will develop the basic technique used to "frame" and "de-frame" a signal so that a filter can be applied individually to each frame.



Download and open framing.vi²⁵.

The "Reshape Array" node forms the heart of framing and de-framing, since you can reshape the incoming 1-D signal vector into a 2-D array of frames. The auto-indexing feature of the "for loop" structure automatically loops over all of the frames, so it is not necessary to wire a value to the loop termination terminal. You can access the individual frame as a 1-D vector inside the loop structure. Auto-indexing is also used on the loop output to create a new 2-D array, so "Reshape Array" is again used to convert the signal back to a 1-D vector.

Study the entire VI, including the unconnected blocks which you will find useful. Complete the VI so that you can select frame sizes of between 1 and 9. Enable the "Highlight Execution" option, and display your block diagram and front panel simultaneously (press Ctrl-T). Convince yourself that your technique works properly. For example, when you select a frame size of 2, you should observe that the front-panel indicator "frame" displays "0,1", then "2,3", then "4,5", and so on. You should also observe that the "out" indicator matches the original.

8.6.5 Part 2: Time-Varying Filter Using Linear Prediction



Download the file part2.zip²⁶, a .zip archive that contains three VIs: part2.vi, blp.vi (band-limited pulse source), and WavRead.vi (reads a .wav audio file). Complete this VI by creating your own "Framer" and "DeFramer" VIs using the techniques you developed in Part 1.

Create or find a speech-type .wav file to use as a basis for the linear prediction filter. Vary the frame size and filter order parameters as well as the various type of excitation sources. Study the effect of each parameter and discuss your results. Submit one or two representative .wav files.

8.6.6 Part 3: Cross Synthesis

"Cross synthesis" applies the spectral envelope of one signal (e.g., speech) to another signal (e.g., a musical instrument). Find or create a speech signal and use it to generate a time-varying filter. Find or create a music signal and use it as the excitation. The sound files should have the same sampling frequency.

Repeat for a second set of signals. You might also try cross synthesizing two different speech signals or two different music signals.


Show your results, particularly the spectrograms of the two original signals and the spectrogram of the output signal.

Select your favorite result and submit .wav files of the two source signals and the output signal.

²⁵<http://cnx.org/content/m15479/latest/framing.vi>

²⁶<http://cnx.org/content/m15479/latest/part2.zip>

8.7 Karplus-Strong Plucked String Algorithm²⁷

	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide²⁸ module for tutorials and documentation that will help you:</p>
	<p><i>continued on next page</i></p>

²⁷This content is available online at <<http://cnx.org/content/m15489/1.2/>>.

• Apply LabVIEW to Audio Signal Processing
• Get started with LabVIEW
• Obtain a fully-functional evaluation edition of LabVIEW

Table 8.7

8.7.1 Introduction

In 1983 Kevin Karplus and Alex Strong published an algorithm to emulate the sound of a plucked string (see "References" section). The Karplus-Strong algorithm produces remarkably realistic tones with modest computational effort.

As an example, consider the sound of a violin's four strings plucked in succession: violin_plucked.wav²⁹ (compare to the same four strings bowed instead of plucked: violin_bowed.wav³⁰). Now compare to the Karplus-Strong version of the same four pitches: ks_plucked.wav³¹.

In this module, learn about the Karplus-Strong plucked string algorithm and how to create a LabVIEW virtual musical instrument (VMI) that you can "play" using a MIDI music file.

8.7.2 Karplus-Strong Algorithm

The Figure 8.12 screencast video develops the theory of the Karplus-Strong plucked string algorithm, which is based on a closed loop composed of a delay line and a low pass filter. As will be shown, the delay line is initialized with a noise burst, and the continuously circulating noise burst is filtered slightly on each pass through the loop. The output signal is therefore quasi-periodic with a wideband noise-like transient converging to a narrowband signal composed of only a few sinusoidal harmonic components.

Image not finished

Figure 8.12: [video] Theory of the Karplus-Strong plucked string algorithm

8.7.3 LabVIEW Implementation

The Karplus-Strong algorithm block diagram may be viewed as a **single** digital filter that is excited by a noise pulse. For real-time implementation, the digital filter runs continuously with an input that is normally zero. The filter is "plucked" by applying a burst of white noise that is long enough to completely fill the delay line.

As an exercise, review the block diagram shown in Figure 8.12 and derive the difference equation that relates the overall output $y(n)$ to the input $x(n)$. Invest some effort in this so that you can develop a better understanding of the algorithm. Watch the video solution in Figure 8.13 only **after** you have completed your own derivation.

²⁸"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

²⁹http://cnx.org/content/m15489/latest/violin_plucked.wav

³⁰http://cnx.org/content/m15489/latest/violin_bowed.wav

³¹http://cnx.org/content/m15489/latest/ks_plucked.wav

Image not finished

Figure 8.13: [video] Difference equation for Karplus-Strong block diagram

The Figure 8.14 screencast video shows how to implement the difference equation as a digital filter and how to create the noise pulse. The video includes an audio demonstration of the finished result.

Image not finished

Figure 8.14: [video] Building the Karplus-Strong block diagram in LabVIEW

8.7.4 Project Activity: Karplus-Strong VMI

In order to better appreciate the musical qualities of the Karplus-Strong plucked string algorithm, convert the algorithm to a **virtual musical instrument** (VMI for short) that can be played by "MIDI Jam Session." If necessary, visit MIDI Jam Session (Section 4.6), download the application VI .zip file, and view the screencast video in that module to learn more about the application and how to create your own virtual musical instrument. Your VMI will accept parameters that specify frequency, amplitude, and duration of a single note, and will produce a corresponding array of audio samples using the Karplus-Strong algorithm described in the previous section.

For best results, select a MIDI music file that contains a solo instrument or perhaps a duet. For example, try "Sonata in A Minor for Cello and Bass Continuo" by Antonio Vivaldi. A MIDI version of the sonata is available at the Classical Guitar MIDI Archives³², specifically Vivaldi_Sonata_Cello_Bass.mid³³.

Try experimenting with the critical parameters of your instrument, including sampling frequency and the low-pass filter constant g . Regarding sampling frequency: lower sampling frequencies influence the sound in **two** distinct ways – can you describe each of these two ways?

8.7.5 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Karplus, K., and A. Strong, "Digital Synthesis of Plucked String and Drum Timbres," Computer Music Journal 7(2): 43-55, 1983.

8.8 Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy³⁴

³²<http://www.classicalguitarmidi.com>

³³http://www.classicalguitarmidi.com/subiviv/Vivaldi_Sonata_Cello_Bass.mid

³⁴This content is available online at <<http://cnx.org/content/m15490/1.2/>>.


	<p>This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide³⁵ module for tutorials and documentation that will help you:</p>
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<p>Get started with LabVIEW</p> <ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 8.8

8.8.1 Introduction

In the project activity of the prerequisite module Karplus-Strong Plucked String Algorithm (Section 8.7), you undoubtedly noticed that the pitch accuracy of the basic Karplus-Strong algorithm needs improvement. For example, listen to the short MIDI test sequence `ksdemo.mid`³⁶ rendered to audio with the basic algorithm using a sampling frequency of 20 kHz: `ksdemo_20kHz.wav`³⁷. The individual notes sound reasonable, but when the notes are played simultaneously as a chord, the pitch inaccuracy becomes noticeable. The accuracy gets worse at lower sampling frequencies such as 10 kHz: `ksdemo_10kHz.wav`³⁸; increasing the sampling frequency improves the accuracy, as at 44.1 kHz: `ksdemo_44kHz.wav`³⁹, however, a discerning ear can still detect some problems.

The pitch of the plucked string tone is determined by the loop time, which must be made continuously variable in order to precisely control the pitch. In the basic algorithm, the length of the delay line determines the loop time, and the delay line can only be varied in integer amounts. The **all-pass filter** will be introduced and studied in this module as a means to introduce an adjustable fractional delay into the loop.

As a preview of the results that you can achieve, consider the same MIDI test sequence rendered to audio using the techniques introduced in this section: `ks2demo_10kHz.wav`⁴⁰ and `ks2demo_20kHz.wav`⁴¹.

8.8.2 Lowpass Filter Delay

In the prerequisite module (Section 8.7), the loop time was determined to be the product of the delay line length and the sampling interval. The reciprocal of the loop time is the pitch (frequency) of the output signal f_0 :

$$f_0 = \frac{f_s}{N} \quad (8.4)$$

where f_s is the sampling frequency in Hz and N is the length of the delay line in samples. Because the delay line length must be an integer number of samples, the pitch cannot be set arbitrarily.

Try the following exercises to explore this concept in more detail.

Exercise 8.1

(Solution on p. 136.)

The sampling frequency is 40.00 kHz, and the length of the delay line is 40 samples. What is the pitch of the output signal? If the delay line length is decreased by one sample, what is the new pitch?

³⁵"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

³⁶<http://cnx.org/content/m15490/latest/ksdemo.mid>

³⁷http://cnx.org/content/m15490/latest/ksdemo_20kHz.wav

³⁸http://cnx.org/content/m15490/latest/ksdemo_10kHz.wav

³⁹http://cnx.org/content/m15490/latest/ksdemo_44kHz.wav

⁴⁰http://cnx.org/content/m15490/latest/ks2demo_10kHz.wav

⁴¹http://cnx.org/content/m15490/latest/ks2demo_20kHz.wav

Exercise 8.2*(Solution on p. 136.)*

The sampling frequency is 10.00 kHz, and the length of the delay line is 10 samples. What is the pitch of the output signal? If the delay line length is decreased by one sample, what is the new pitch?

For each of the two exercises, the first pitch is exactly the same, i.e., 1000 Hz. However, the change in pitch caused by decreasing the delay line by only one sample is substantial (1026 Hz compared to 1111 Hz). But how perceptible is this difference? The module Musical Intervals and the Equal-Tempered Scale (Section 2.2) includes a LabVIEW interactive front panel that displays the frequency of each key on a standard 88-key piano. Pitch C6 is 1046 Hz while pitch C#6 (a half-step higher) is 1109 Hz. These values are similar to the change from 1000 Hz to 1111 Hz caused by altering the delay line length by only one sample, so the change is certainly very audible. The abrupt "jump" in frequency becomes less pronounced at lower pitches where the delay line length is longer.

Flexibility to adjust the overall loop time in a continuous fashion is required to improve pitch accuracy. Moreover, any sources of delay in the loop must be accurately known. So far the delay of the low pass filter has been taken as zero, but in fact the low pass filter introduces a delay of its own.

The Figure 8.15 screencast video describes first how to calculate the delay of an arbitrary digital filter with transfer function $H(z)$.

Image not finished

Figure 8.15: [video] Calculating the delay of a filter given $H(z)$

In general, therefore, the delay is the negated slope of the filter's phase function, and the delay varies with frequency.

Now, consider the specific low pass filter used in the basic Karplus-Strong algorithm. The filter coefficient "g" will be taken as 0.5, making the filter a true two-point averager:

$$H_{\text{LPF}}(z) = \frac{1 + z^{-1}}{2} \quad (8.5)$$

The Figure 8.16 screencast video continues the discussion by deriving the delay of the low pass filter of (8.5). Several techniques for working with complex numbers in LabVIEW are presented and used to visualize the magnitude and phase response of the filter.

Image not finished

Figure 8.16: [video] Calculating the delay of the low pass filter

Because the delay of the low pass filter is always $1/2$, the pitch may be expressed more precisely as

$$f_0 = \frac{f_s}{N + \frac{1}{2}} \quad (8.6)$$

While this result more accurately calculates the pitch, it does nothing to address the frequency resolution problem.

8.8.3 All-Pass Filter Delay

Now, consider the **all-pass filter (APF)** as a means to introduce a variable and fractional delay into the loop. The all-pass filter has a unit magnitude response over all frequencies, so it does not "color" the signal passing through. However, the all-pass filter has a phase response that is approximately linear for all but the highest frequencies, so it introduces an approximately constant delay. Even better, the slope of the phase response is continuously variable, making it possible to adjust the delay as needed between 0 and 1 samples.

The all-pass filter transfer function is

$$H_{\text{APF}}(z) = \frac{C + z^{-1}}{1 + Cz^{-1}} \quad (8.7)$$

where $|C| < 1$ to ensure stability.

The Figure 8.17 screencast video continues the discussion by exploring the delay of the all-pass filter of (8.7) as a function of the parameter C .

Image not finished

Figure 8.17: [video] Calculating the delay of the low pass filter

8.8.4 Implementing the Pitch-Accurate Algorithm

Including the all-pass filter in the basic Karplus-Strong algorithm allows the loop time to be set to an arbitrary value, making it possible to sound a tone with any desired pitch.

This section guides you through the necessary steps to augment the basic algorithm with an all-pass filter, including the derivation of necessary equations to calculate the delay line length and the fractional delay. Work through the derivations requested by each of the exercises.

To begin, the pitch of the output signal is the sampling frequency f_s divided by the total loop delay in samples:

$$f_0 = \frac{f_s}{N + \frac{1}{2} + \Delta} \quad (8.8)$$

where Δ is the fractional delay introduced by the all-pass filter.

Exercise 8.3

(Solution on p. 136.)

Refer to (8.8). Derive a pair of equations that can be used to calculate the length of the delay line N and the value of the fractional delay Δ .

The all-pass filter delay can be approximated by (8.9) (see Moore):

$$\Delta = \frac{1 - C}{1 + C} \quad (8.9)$$

Exercise 8.4

(Solution on p. 136.)

Refer to (8.9). Solve the equation for the all-pass filter coefficient C in terms of the required fractional delay.

The all-pass filter can be inserted at any point in the loop; Figure 8.18 shows the all-pass filter placed after the low pass filter.

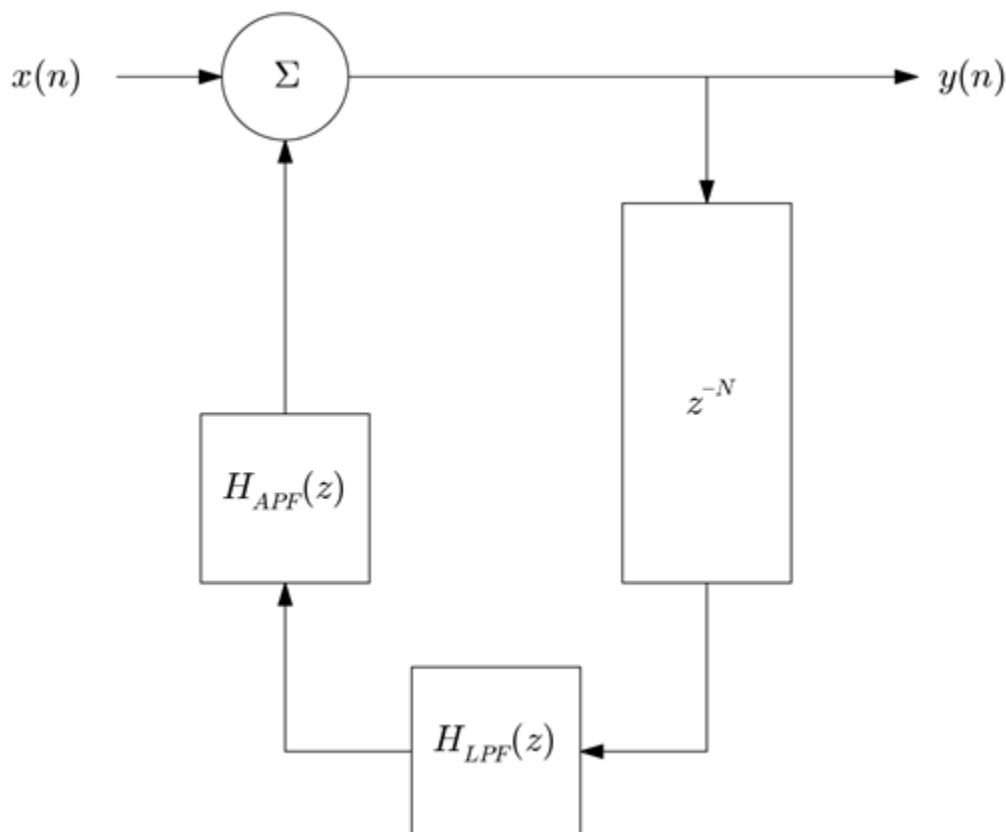


Figure 8.18: Block diagram of the pitch-accurate Karplus-Strong algorithm

Exercise 8.5

(Solution on p. 136.)

To simplify the derivation of the overall filter transfer function that relates $y(n)$ to $x(n)$, consider the three feedback elements (delay line, low pass filter, and all-pass filter) to be a single element with transfer function $G(z)$. Derive the transfer function of the combined element.

Exercise 8.6

(Solution on p. 136.)

Refer to the block diagram of Figure 8.18. Considering that all three elements are represented by a single feedback element $G(z)$, derive the overall transfer function $H(z)$ for the digital filter in terms of $G(z)$.

Recall the low pass filter transfer function ((8.10)):

$$H_{LPF}(z) = g(1 + z^{-1}) \quad (8.10)$$

The all-pass filter transfer function is described by (8.7).

Exercise 8.7

(Solution on p. 136.)

Derive the overall transfer function $H(z)$ in terms of the filter parameters g and C . Write the transfer function in standard form as the ratio of two polynomials in z .

8.8.5 Project Activity: Karplus-Strong VMI

As in the prerequisite module (Section 8.7), convert the pitch-accurate Karplus-Strong algorithm into a **virtual musical instrument (VMI)** that can be played by "MIDI Jam Session." If necessary, visit MIDI JamSession (Section 4.6), download the application VI.zip file, and view the screencast video in that module to learn more about the application and how to create your own virtual musical instrument. Your VMI will accept parameters that specify frequency, amplitude, and duration of a single note, and will produce a corresponding array of audio samples using the Karplus-Strong algorithm described in the previous section.

For best results, select a MIDI music file that contains a solo instrument or perhaps a duet. For example, try "Sonata in A Minor for Cello and Bass Continuo" by Antonio Vivaldi. A MIDI version of the sonata is available at the Classical Guitar MIDI Archives⁴², specifically Vivaldi_Sonata_Cello_Bass.mid⁴³.

8.8.6 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Karplus, K., and A. Strong, "Digital Synthesis of Plucked String and Drum Timbres," Computer Music Journal 7(2): 43-55, 1983.

⁴²<http://www.classicalguitarmidi.com>

⁴³http://www.classicalguitarmidi.com/subiviv/Vivaldi_Sonata_Cello_Bass.mid

Solutions to Exercises in Chapter 8

Solution to Exercise 8.1 (p. 131)

1000 Hz (40 kHz divided by 40 samples); 1026 Hz

Solution to Exercise 8.2 (p. 132)

1000 Hz (10 kHz divided by 10 samples); 1111 Hz

Solution to Exercise 8.3 (p. 133)

$N = \lfloor \frac{f_s}{f_0} - \frac{1}{2} \rfloor$ (the "floor" operator converts the operand to an integer by selecting the largest integer that is less than the operand); $\Delta = \frac{f_s}{f_0} - N$

Solution to Exercise 8.4 (p. 133)

$$C = \frac{1-\Delta}{1+\Delta}$$

Solution to Exercise 8.5 (p. 134)

The elements are in cascade, so the individual transfer functions multiply together: $G(z) = z^{-N} H_{LPF}(z) H_{APF}(z)$

Solution to Exercise 8.6 (p. 134)

$$H(z) = \frac{1}{1-G(z)}$$

Solution to Exercise 8.7 (p. 134)

$$H(z) = \frac{1+Cz^{-1}}{1+Cz^{-1}-gCz^{-N}-g(1+C)z^{-(N+1)}-gz^{-(N+2)}}$$

Chapter 9

Sound Spatialization and Reverberation

9.1 Reverberation¹


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ² module for tutorials and documentation that will help you:
	• Apply LabVIEW to Audio Signal Processing
	• Get started with LabVIEW
	• Obtain a fully-functional evaluation edition of LabVIEW

Table 9.1

9.1.1 Introduction

Reverberation is a property of concert halls that greatly adds to the enjoyment of a musical performance. The on-stage performer generates sound waves that propagate directly to the listener's ear. However, sound waves also bounce off the floor, walls, ceiling, and back wall of the stage, creating myriad copies of the direct sound that are time-delayed and reduced in intensity.

In this module, learn about the concept of reverberation in more detail and ways to emulate reverberation using a digital filter structure known as a **comb filter**.

The Figure 9.1 screencast video continues the discussion by visualizing the sound paths in a reverberant environment. The **impulse response** of the reverberant environment is also introduced. Understanding the desired impulse response is the first step toward emulating reverberation with a digital filter.

Image not finished

Figure 9.1: [video] Sound paths in a reverberant environment and impulse response model

¹This content is available online at <<http://cnx.org/content/m15471/1.2/>>.

²"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

9.1.2 Comb Filter Theory

The **comb filter** is a relatively simple digital filter structure based on a **delay line** and **feedback**. Watch the Figure 9.2 screencast video to learn how you can develop the comb filter structure by considering an idealized version of the impulse response of a reverberant environment.

Image not finished

Figure 9.2: [video] Development of the **comb filter** structure as a way to implement an idealized reverberant impulse response

The **difference equation** of the comb filter is required in order to implement the filter in LabVIEW. In general, a difference equation states the filter output $y(n)$ as a function of the past and present input values as well as past output values.

Take some time now to derive the comb filter difference equation as requested by the following exercise.

Exercise 9.1

(Solution on p. 147.)

Derive the difference equation of the comb filter structure shown at the end of the Figure 9.2 video.

9.1.3 Comb Filter Implementation

Once the difference equation is available, you can apply the coefficients of the equation to the LabVIEW "IIR" (infinite impulse response) digital filter. The Figure 9.3 screencast video walks through the complete process you need to convert the comb filter difference equation into a LabVIEW VI. The LabVIEW **MathScript node** creates the coefficient vectors. Once the comb filter is complete, its impulse response is explored for different values of delay line length and feedback gain.

Please follow along with the video and create your own version of the comb filter in LabVIEW. Refer to TripleDisplay³ to install the front-panel indicator used to view the signal spectrum.

Image not finished

Figure 9.3: [video] Implementing the comb filter in LabVIEW; exploration of the impulse response as a function of delay line length and feedback gain

9.1.4 Loop Time and Reverb Time

As you have learned in previous sections, the comb filter behavior is determined by the delay line length N and the feedback coefficient g . From a user's point of view, however, these two parameters are not very intuitive. Instead, the comb filter behavior is normally specified by **loop time** τ and **reverb time** denoted T_{60} . Reverb time may also be written as R_{T60} . Loop time indicates the amount of time necessary for a given sample to pass through the delay line, and is therefore the delay line length N times the sampling

³"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

interval. The sampling interval is the reciprocal of sampling frequency, so the loop time may be expressed as in (9.1):

$$\tau = \frac{N}{f_S} \quad (9.1)$$

Reverb time indicates the amount of time required for the reverberant signal's intensity to drop by 60 dB (dB = decibels), effectively to silence. Recall from the Figure 9.2 video that the comb filter's impulse response looks like a series of decaying impulses spaced by a delay of N samples; this impulse response is plotted in Figure 9.4 with the independent axis measured in time rather than samples.

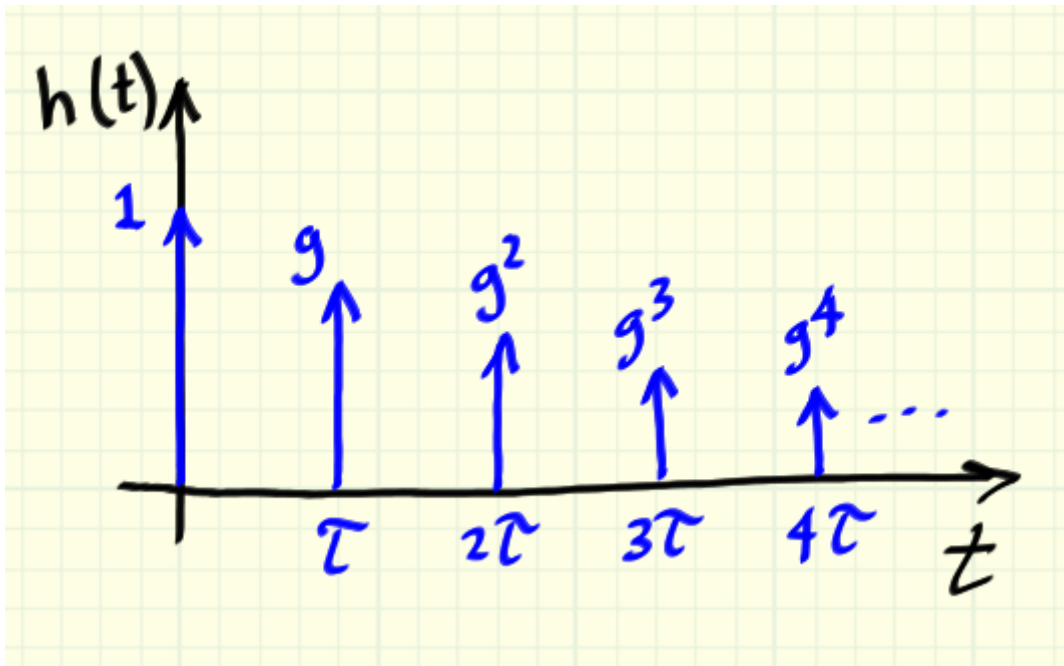


Figure 9.4: Comb filter impulse response

Take a few minutes to derive an equation for the comb filter feedback gain "g" as a function of the loop time and the reverb time. The following pair of exercises guide you through the derivation.

Exercise 9.2

(Solution on p. 147.)

Given the comb filter impulse response pictured in Figure 9.4, derive an equation for the reverb time T_{60} in terms of the loop time τ and the comb filter's feedback constant g . Hint: Recall the basic equation to express the ratio of two amplitudes in decibels, i.e., use the form with a factor of 20.

Exercise 9.3

(Solution on p. 147.)

Based on your previous derivation, develop an equation for the comb filter gain "g" in terms of the desired loop time and reverb time.

Exercise 9.4

(Solution on p. 147.)

To finish up, derive the equation for the comb filter delay "N" in terms of the desired loop time.

Now, return to your own comb filter VI and modify the front-panel controls and LabVIEW MathScript node to use loop time and reverb time as the primary user inputs. Experiment with your modified system to ensure that the spacing between impulses does indeed match the specified loop time, and that the impulse decay rate makes sense for the specified reverb time.

9.1.5 Comb Filter Implementation for Audio Signals

In this section, learn how to build a comb filter in LabVIEW that can process an audio signal, specifically, an impulse source. Follow along with the Figure 9.5 screencast video to create your own VI. The video includes an audio demonstration of the finished result. As a bonus, the video also explains where the "comb filter" gets its name.

Image not finished

Figure 9.5: [video] Building a LabVIEW VI of a comb filter that can process an audio signal

Next, learn how you can replace the impulse source with an audio .wav file. Speech makes a good test signal, and the Figure 9.6 screencast video shows how to modify your VI to use a .wav audio file as the signal source. The speech clip used as an example in the video is available here: [speech.wav](http://www.voiptroubleshooter.com/open_speech/speech.wav)⁴ (audio courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech⁵ ; the sentences are two of the many phonetically balanced **Harvard Sentences**, an important standard for the speech processing community).

Image not finished

Figure 9.6: [video] Modifying the LabVIEW VI of a comb filter to process a .wav audio signal

9.1.6 References

- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Dodge, C., and T.A. Jerse, "Computer Music: Synthesis, Composition, and Performance," 2nd ed., Schirmer Books, 1997, ISBN 0-02-864682-7.

⁴<http://cnx.org/content/m15471/latest/speech.wav>

⁵http://www.voiptroubleshooter.com/open_speech

9.2 Schroeder Reverberator⁶

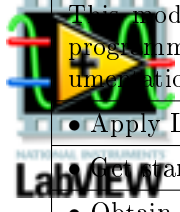
	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ⁷ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 9.2

9.2.1 Introduction

Reverberation is a property of concert halls that greatly adds to the enjoyment of a musical performance. The on-stage performer generates sound waves that propagate directly to the listener's ear. However, sound waves also bounce off the floor, walls, ceiling, and back wall of the stage, creating myriad copies of the direct sound that are time-delayed and reduced in intensity.

In the prerequisite module Reverberation (Section 9.1), you learned how the **comb filter** structure can efficiently create replicas of a direct-path signal that are time delayed and reduced in intensity. However, the comb filter produces replicas that are time delayed by exactly the same amount, leading to the sensation of a pitched tone superimposed on the signal. Refer back to Reverberation (Section 9.1) to hear an audio demonstration of this effect. Put another way, the impulse response of the comb filter contains impulses with identical spacing in time, which is not realistic.

The **Schroeder reverberator** (see "References" section) uses a combination of comb filters and **all-pass filters** to produce an impulse response that more nearly resembles the random nature of a physical reverberant environment.

This module introduces you to the Schroeder reverberator and guides you through the implementation process in LabVIEW. As a preview of what can be achieved, watch the Figure 9.7 screencast video to see and hear a short demonstration of a LabVIEW VI that implements the Schroeder reverberator. The speech clip used in the video is available here: [speech.wav](http://www.voiptroubleshooter.com/open_speech/speech.wav)⁸ (audio courtesy of the Open Speech Repository, www.voiptroubleshooter.com/open_speech⁹ ; the sentences are two of the many phonetically balanced **Harvard Sentences**, an important standard for the speech processing community).

Image not finished

Figure 9.7: [video] Demonstration of the **Schroeder reverberator** as implemented in LabVIEW

9.2.2 Structure of the Schroeder Reverberator

The Figure 9.8 screencast video presents the structure of the Schroeder reverberator and describes the rationale for its design.

⁶This content is available online at <<http://cnx.org/content/m15491/1.2/>>.

⁷"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

⁸<http://cnx.org/content/m15491/latest/speech.wav>

⁹http://www.voiptroubleshooter.com/open_speech

Image not finished

Figure 9.8: [video] Structure of the **Schroeder reverberator** and rationale for its design

9.2.3 All-Pass Filter

The Schroeder reverberator uses **all-pass filters** to increase the pulse density produced by the parallel comb filters. You perhaps are familiar with the frequency response of an all-pass filter: its magnitude response is unity (flat) for all frequencies, and its phase response varies with frequency. For example, the all-pass filter is used to create a variable fractional delay as described in Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy (Section 8.8).

From an intuitive standpoint it would seem that a flat magnitude response in the frequency domain should correspond to an impulse response (time-domain) containing only a single delta function (recall that the impulse function and a constant-valued function constitute a Fourier transform pair). However, the all-pass filter's impulse response is actually a large negative impulse followed by a series of positive decaying impulses.

In this section, derive the transfer function and difference equation of the all-pass filter structure shown in Figure 9.9. Note that the structure is approximately of the same level of complexity as the comb filter: it contains a delay line of N samples and an extra gain element and summing element.

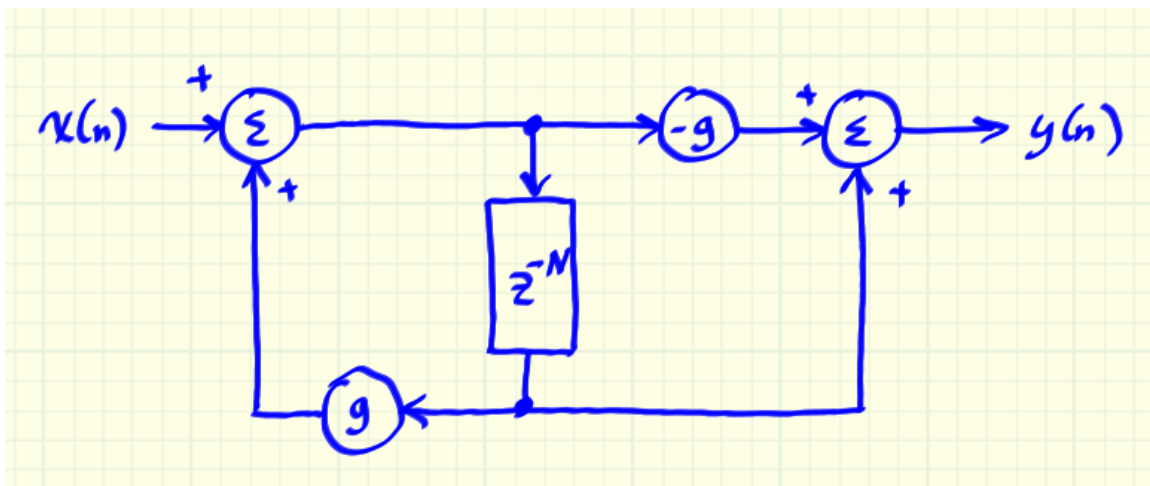


Figure 9.9: All-pass filter structure

Exercise 9.5

(Solution on p. 147.)

Determine the transfer function $H(z)$ for the all-pass filter structure of Figure 9.9.

Exercise 9.6

(Solution on p. 147.)

Based on your result for the previous exercise, write the difference equation for the all-pass filter.

9.2.4 All-Pass Filter Impulse Response

As described in the video of Figure 9.8, two all-pass filters are placed in cascade (series) with the summed output of the parallel comb filters. An understanding of the all-pass filter impulse response reveals why a cascade connection increases the pulse density of the comb filter in such a way as to emulate the effect of natural reverberation.

The Figure 9.10 screencast video derives the impulse response of the all-pass filter; the loop time and reverb time of the all-pass filter are also presented.

Image not finished

Figure 9.10: [video] Derivation of the all-pass filter impulse response

The Figure 9.11 screencast video demonstrates the sound of the all-pass filter impulse response compared to that of the comb filter. Moreover, the audible effect of increasing the comb filter pulse density with an all-pass filter is also demonstrated in the video.



Download the LabVIEW VI presented in the video: apfdemo.zip¹⁰ Refer to TripleDisplay¹¹ to install the front-panel indicator required by the VI.

Image not finished

Figure 9.11: [video] Audio demonstration of the all-pass filter impulse response combined with comb filter

9.2.5 Project: Implement the Schroeder Reverberator in LabVIEW

As described in the Figure 9.8 screencast video, two all-pass filters are placed in cascade (series) with the summed output of four parallel comb filters. The video of Figure 9.10 explains how the all-pass filter "fattens up" each comb filter output impulse with high density pulses that rapidly decay to zero. Selecting mutually-prime numbers for the loop times ensures that the comb filter impulses do not overlap too soon, which further increases the effect of randomly-spaced impulses.

The table in Figure 9.12 lists the required reverb times (T_{60}) and loop times (τ) of the Schroeder reverberator. Note that the comb filters all use the same value (the desired overall reverb time).

¹⁰<http://cnx.org/content/m15491/latest/apfdemo.zip>

¹¹"[LabVIEW application] TripleDisplay" <<http://cnx.org/content/m15430/latest/>>

Filter	T_{60} (ms)	τ (ms)
CF_1	T_{60}	29.7
CF_2	T_{60}	37.1
CF_3	T_{60}	41.1
CF_4	T_{60}	43.7
APF_1	5.0	96.83
APF_2	1.7	32.92

Figure 9.12: Reverb time and loop time values for the comb filters and all-pass filters of the Schroeder reverberator

The Figure 9.13 screencast video provides everything you need to know to build your own LabVIEW VI for the Schroeder reverberator.



Download the .wav reader subVI mentioned in the video : WavRead.vi¹².

Image not finished

Figure 9.13: [video] Suggested LabVIEW techniques to build your own Schroeder reverberator

9.2.6 References

- Schroeder, M.R., and B.F. Logan, "Colorless Artificial Reverberation," Journal of the Audio Engineering Society 9(3):192, 1961.
- Schroeder, M.R., "Natural Sounding Artificial Reverberation," Journal of the Audio Engineering Society 10(3):219-223, 1962.
- Moore, F.R., "Elements of Computer Music," Prentice-Hall, 1990, ISBN 0-13-252552-6.
- Dodge, C., and T.A. Jerse, "Computer Music: Synthesis, Composition, and Performance," 2nd ed., Schirmer Books, 1997, ISBN 0-02-864682-7.

9.3 Localization Cues¹³

¹²<http://cnx.org/content/m15491/latest/WavRead.vi>

¹³This content is available online at <<http://cnx.org/content/m15475/1.2/>>.


	This module refers to LabVIEW, a software development environment that features a graphical programming language. Please see the LabVIEW QuickStart Guide ¹⁴ module for tutorials and documentation that will help you:
	<ul style="list-style-type: none"> • Apply LabVIEW to Audio Signal Processing
	<ul style="list-style-type: none"> • Get started with LabVIEW
	<ul style="list-style-type: none"> • Obtain a fully-functional evaluation edition of LabVIEW

Table 9.3

9.3.1 Introduction

Our enjoyment of synthesized or recorded sound is greatly enhanced when we perceive the actual locations of the various musicians on stage. In a high quality stereo recording of a jazz trio, for example, you can tell that the drummer is perhaps located slightly to the left of center, the saxophonist is on stage right and the bass player is on stage left. If you have ever flipped on the "mono" (monaural) switch on your stereo amplifier, then you know that the resulting sound is comparatively unpleasant, especially when wearing headphones.

We live in a three-dimensional soundfield, and our ears continually experience slightly different versions of any given sound. These variations allow us to place (or **localize**) the sound source, even when we cannot see it.

In this module, learn about two **localization cues** called **interaural intensity difference** and **interaural timing difference**, abbreviated **IID** and **ITD**, respectively. Each cue relies on presenting a slightly different version of a sound to each ear.

9.3.2 Interaural Intensity Difference (IID)

The interaural intensity difference (IID) localization cue relies on the fact that an off-center source produces a higher intensity sound in one ear compared to the other. This technique is also called **intensity panning**, and is most effective when the listener is wearing headphones. When a multi-speaker arrangement is used in a room, a given sound propagates to both ears. Lower frequencies have longer acoustic wavelengths and are therefore less directional than higher frequency sounds. Consequently, the IID effect works better at higher frequencies above 1400 Hz (Dodge and Jerse, 1997).

The Figure 9.14 screencast video continues the discussion by deriving the equations needed to implement the IID effect using a two-speaker arrangement.

Image not finished

Figure 9.14: [video] Development of the equations needed to implement the **interaural intensity difference (IID)** localization cue

9.3.3 Interaural Timing Difference (ITD)

The interaural timing difference (ITD) localization cue relies on the fact that sound waves from an off-center source arrives at one ear slightly after the other ear. We can perceive this slight difference in time down

¹⁴"NI LabVIEW Getting Started FAQ" <<http://cnx.org/content/m15428/latest/>>

to about 20 microseconds (Dodge and Jerse, 1997), and this difference helps us to place the sound source either to the left or right. The ITD cue works best in the lower frequency range of 270 to 500 Hz (Dodge and Jerse, 1997).

The Figure 9.15 screencast video continues the discussion by deriving the equation needed to implement the ITD effect using a two-speaker arrangement.

Image not finished

Figure 9.15: [video] Development of the equation needed to implement the **interaural timing difference** (ITD) localization cue

9.3.4 Project: Implement the IID and ITD Localization Cues in LabVIEW

You can easily experiment with both the IID and ITD localization cues in LabVIEW. The cues are probably easier to perceive when you choose a speech signal for your source.

The ITD cue requires a delay or time shift between the two stereo channels. The delay line can be constructed as a digital filter with the transfer function shown in :

$$H(z) = \frac{z^{-N}}{1} \quad (9.2)$$

Stated as a difference equation, the filter is $y(n) = x(n - N)$, which states that the output is the same as the input but delayed by N samples. The forward (or "b") coefficients are therefore zero for all but $b_N = 1$, and the reverse (or "a") coefficients are zero for all but $a_0 = 1$.



The Figure 9.16 screencast video provides LabVIEW coding tips to implement the equations and to generate a stereo audio signal.

Image not finished

Figure 9.16: [video] LabVIEW coding tips for the IID and ITD localization cues

9.3.5 References

- Dodge, C., and T.A. Jerse, "Computer Music: Synthesis, Composition, and Performance," 2nd ed., Schirmer Books, 1997, ISBN 0-02-864682-7

Solutions to Exercises in Chapter 9

Solution to Exercise 9.1 (p. 138)

Solution to Exercise 9.2 (p. 139)

$$T_{60} = \frac{-3\tau}{\log_{10} g}$$

Solution to Exercise 9.3 (p. 139)

$$g = 10^{-\frac{3\tau}{T_{60}}}$$

Solution to Exercise 9.4 (p. 139)

$$N = \tau f_S$$

Solution to Exercise 9.5 (p. 142)

$$H(z) = \frac{-g + z^{-N}}{1 - gz^{-N}}$$

Solution to Exercise 9.6 (p. 142)

$$y(n) = -gx(n) + x(n - N) + gy(n - N)$$

Index of Keywords and Terms

Keywords are listed by the section with that keyword (page numbers are in parentheses). **Keywords** do not necessarily appear in the text of the page. They are merely associated with that section. *Ex.* apples, § 1.1 (1) **Terms** are referenced by the page they appear on. *Ex.* apples, 1

- , 27
- ((text event, 50
 (track name, 50
- . .mid, 48
- 1** 1000nnnn, 44
 1001nnnn, 44
 1011nnnn, 45, 46
 1100nnnn, 45
 1110nnnn, 46
 11110000, 47
 11110111, 47
- 8** 8n, 44
- 9** 9n, 44
- A** a standard MIDI file, 48
 additive synthesis, § 7.1(105), 105, § 7.2(107),
 107, § 7.3(109), § 7.4(112)
 ADSR, § 3.1(33), 35, 35, § 3.2(36)
 ADSR.vi, 37, 38
 Alex Strong, § 8.7(128)
 aliasing, 85, § 6.2(87), § 8.3(120), 121
 aligning, § 1.2(12), 12
 all-pass filter, § 8.8(130), 131, 133, § 9.2(141)
 all-pass filters, 141, 142
 AM, § 6.1(83), 83, 88
 amplitude, § 2.1(23), 23, 24
 amplitude modulation, § 5.1(69), 71, § 6.1(83),
 83, 88, § 6.3(89)
 amplitude trajectory, § 7.1(105), 105,
 § 7.2(107), 108, § 7.3(109), § 7.4(112)
 analog synthesis, § 3.1(33), § 3.2(36)
 Analog synthesizers, 33
 APF, 133
 application VI, 62
 array, § 1.5(18)
 array constant, § 1.5(18)
 array control, § 1.5(18)
 array dimensions, § 1.5(18)
- arrow keys, 12
- audio, § 1.6(20), § 1.9(21), § 4.6(64), § 9.1(137)
- audio signal processing, § (1)
- audio source, § 1.8(20)
- B** band-limited pulse, § 8.3(120), 121, 121,
 § 8.4(122)
 bandpass filter, § 8.4(122)
 bank select, § 4.1(41)
 bank select value, 45
 baseband, 83, 91
 beating, 26
 Behavior Requirements, 57
 bell, § 7.3(109)
 Bessel function, § 6.4(91)
 big-endian, 48
 binary file, § 4.5(55)
 block, 9
 block processing, 118
 Bn, 45, 46
 Boolean, 10, 16
 Boolean Array, 60
 breakpoints, 11
 Broken wires, 11, § 1.2(12), 12
 Build Waveform, § 1.6(20), 20
- C** carrier frequency, § 6.4(91)
 case structure, § 1.3(12), 12, 16, 16, 38
 channel number, § 4.1(41), 44
 channels, 64
 chunk length, 48, 48
 chunk type, 48
 chunks, 48
 clean up wire, § 1.2(12), 12
 Clear, 20
 click noise, § 8.2(119)
 Close File, 57
 Cn, 45
 coercion indicator, 10
 comb filter, § 9.1(137), 137, 138, 138,
 § 9.2(141), 141
 computer music, § (1), 110

- Concatenate Strings, 58
 - concatenated, 15
 - concert A, 27
 - configure, § 1.7(20), 20
 - connector pane, § 1.4(18), 18
 - Connexions, 23
 - consonant, 26
 - constants, § 1.2(12), 12, 12
 - Context Help, § 1.2(12), 12
 - continuous, 33
 - control change, § 4.1(41), 45, 46
 - control voltage (CV), 34
 - controller, 35
 - controller number, 45, 46
 - controller value, 45, 46
 - controllers, 34
 - controls, 9, 10, § 1.2(12), 12, 12
 - copyright notice, 50
 - cross synthesis, § 8.5(124), 125, § 8.6(126)
 - CSV, § 1.5(18)
 - Ctrl+B, 12
 - CV, 35
- D**
- daisy chain, 42
 - data byte, § 4.1(41)
 - data bytes, 41
 - data type conversion, 10
 - data types, § 1.1(9), 10
 - dataflow, § 1.1(9), 10
 - DAW, 48
 - dB, 24
 - debugging, § 1.1(9)
 - decibel, 24
 - decorations, § 1.2(12), 12
 - delay line, § 8.7(128), § 9.1(137), 138, § 9.3(144)
 - delta time, § 4.2(48), § 4.5(55)
 - delta times, 48
 - delta-time, 49, § 4.4(52)
 - delta-times, 48
 - depth, 71, 76
 - diagram, 9
 - diagram disable, § 1.3(12), 13, 17
 - difference equation, 138
 - digital audio workstation, 48
 - digital signal processing, § (1)
 - dimensions, 18
 - DIN-5, 41
 - dissonant, 26
 - distributing, 12
 - distributing objects, § 1.2(12)
 - division, 49
 - DSP, § (1)
 - duration [s], 37, 38
- E**
- e values, 38
 - echo, § 9.1(137)
 - En, 46
 - end-of-track, 50
 - enumerated, 16
 - enumerated data type, 38
 - envelope, 38
 - envelope generator, § 3.1(33), 34, 35, § 3.2(36), § 5.2(72), § 5.4(78), § 7.3(109)
 - envelope generators, 34
 - equal temperament, § 2.1(23), 26, § 2.2(27), 38
 - equal-tempered, 27, 29, 30
 - event, 49
 - event structure, § 6.1(83), 85, 86
 - events, 48
 - exponential, 19
- F**
- F0, 47
 - F7, 47
 - fast Hilbert transform, 90
 - feedback, 138
 - feedback node, 15
 - file type, 48
 - files chunks, 48
 - filter delay, § 8.8(130)
 - filter state, § 8.2(119)
 - finite impulse response, 118
 - FIR, 118
 - FIR filter, § 8.1(115)
 - FM, § 6.4(91), 91
 - FM synthesis, § 6.5(95), 95, § 6.6(96), 97, § 6.7(98), 99
 - folding frequency, 121
 - for loop, § 1.3(12), 12, 15, 15, 15, 15
 - formant, § 8.4(122)
 - formant synthesizer, 123
 - formants, 123
 - Fourier transform, § 6.1(83)
 - frame length, 118
 - frames, 118
 - framing, § 8.6(126)
 - fref [Hz], 38
 - frequency, § 2.1(23), 23, 24
 - frequency modulation, § 5.3(75), 76, § 6.4(91), 91, § 6.5(95), § 6.6(96)
 - Frequency modulation synthesis, 95, 97
 - frequency trajectory, § 7.1(105), 105, § 7.2(107), 108, § 7.3(109), § 7.4(112)
 - front panel, 9

- fs [Hz], 38, 38
- function, § 1.4(18)
- fundamental, 25, 28
- fuse, 25
- G** General MIDI (GM) standard, 45
- General MIDI sound set, § 4.1(41), 100
- gposc.vi, 113
- graphical, 9
- H** half step, 26, 28
- harmonic, 107, 107
- harmonicity ratio, § 6.4(91), 94
- harmonics, § 2.1(23), 23, 25
- harmonious, 26
- Harvard Sentences, 86, 88, 125, 140, 141
- header chunk, § 4.2(48), 48, 48, § 4.5(55)
- help page, 12
- Highlight Execution, 11
- I** icon, 18
- icon editor, § 1.4(18)
- icons, § 1.2(12), 12
- IID, 145
- IIR, 118
- IIR filter, § 8.1(115)
- impulse response, § 9.1(137), 137, § 9.2(141)
- index, § 1.5(18)
- indexing, 15
- indexing, shift register, 15
- indicators, 9, 10, § 1.2(12), 12
- infinite impulse response, 118
- inharmonious, 26
- innovations sequence, 125
- Input Requirements, 56
- instantaneous frequency, § 7.2(107)
- instrument, 38
- instrument name, 50
- integer, 16
- intensity, § 2.1(23), 23, 24
- intensity panning, 145
- interactive, § 1.7(20)
- interactive controller, 35
- interactive parameter control, § 8.2(119)
- interaural intensity difference, § 9.3(144), 145, 145
- interaural timing difference, § 9.3(144), 145, 146
- interval, 28
- ITD, 145
- iterator, 15
- J** JAZZ++, § 4.3(51), 52
- John Chowning, § 6.4(91), § 6.6(96), § 6.7(98)
- just-tempered, 29, 29
- K** Karplus-Strong plucked string algorithm, § 8.8(130)
- Kevin Karplus, § 8.7(128)
- L** LabVIEW, § (1), § 1.1(9), 9, § 1.2(12), § 1.3(12), § 1.4(18), § 1.5(18), § 1.6(20), § 1.7(20), § 1.8(20), § 1.9(21), § 2.2(27), § 3.2(36), § 4.5(55), § 4.6(64), § 5.1(69), § 5.2(72), § 5.3(75), § 5.4(78), § 6.1(83), § 6.3(89), § 6.4(91), § 6.5(95), § 6.6(96), § 6.7(98), § 7.1(105), § 7.3(109), § 7.4(112), § 8.1(115), § 8.2(119), § 8.3(120), § 8.4(122), § 8.5(124), § 8.6(126), § 8.7(128), § 8.8(130), § 9.1(137), § 9.2(141), § 9.3(144)
- latency, 43
- LFO, 35
- linear prediction, § 8.5(124), 125, § 8.6(126), 126
- Linear predictive coding, 125
- localization cues, § 9.3(144), 145
- localize, 145
- logarithmic, 24
- loop count terminal, 15
- loop delay, § 8.8(130)
- loop time, § 9.1(137), 138
- loop tunnel, 15, 15
- low frequency, § 5.1(69), § 5.3(75)
- low-frequency oscillator, 35
- lowpass filter, § 8.7(128), § 8.8(130)
- LPC, 125, 126
- lyric text, 50
- M** MathScript, 16
- MathScript interactive window, 16
- MathScript node, § 1.3(12), 12, 16, 16, 138
- MATLAB, 16
- mean-tempered, 29
- meta-event, § 4.2(48), 49, § 4.4(52), § 4.5(55)
- meta-events, 48, 50
- mf2t, § 4.3(51), 51
- microseconds per quarter note, 50
- middle C, 27
- MIDI, 48, § 4.6(64)
- MIDI IN, 42, 43
- MIDI Jam Session, § 5.4(78)
- MIDI JamSession, 73, § 6.7(98), 99, § 8.7(128), § 8.8(130)
- MIDI message, 41, § 4.4(52), § 4.5(55)
- MIDI messages, § 4.1(41)

- MIDI OUT, 42, 43
- MIDI patch bay, 52
- MIDI sequencer, 52
- MIDI THRU, 43
- MIDI-OX, § 4.3(51), 52
- MIDI-Yoke, 52
- midi_AttachHeader.vi, 58
- midi_FinishTrack.vi, 59
- MIDI_JamSession, 64, 64, 64, 65
- MIDI_JamSession.vi, 64, 65
- midi_MakeDtEvent.vi, 60
- midi_MakeDtMeta.vi, 62
- midi_PutBytes.vi, 57
- midi_ToVLF.vi, 60, 61, 62, 62
- MIDI_UpDown.vi, 56, 56, 56, 56
- modular synthesis, § 3.1(33), § 3.2(36)
- modulation frequency, § 6.4(91)
- modulation index, § 6.4(91)
- modulation property, § 6.1(83), 83
- modules, 34
- moving, § 1.2(12), 12
- multitimbral, 45
- music synthesis, § (1)
- Musical Instrument Digital Interface, 48
- Musical Instrument Digital Interface (MIDI), § 4.1(41)
- Musical Instrument Samples, 99
- musical intervals, § 2.2(27)
- Musical Signal Processing with LabVIEW, 1
- N** Navigation Window, § 1.2(12), 12
 - nodes, 9
 - Normalized frequency, § 1.6(20), § 1.8(20), 20
 - note, 38, 38, 38
 - note number, 44
 - note-off, § 4.1(41)
 - Note-Off event, 44
 - note-on, § 4.1(41)
 - Note-On event, 44
 - number of tracks, 48
 - numeric, 10
- O** octave interval, 26
 - octave space, § 7.1(105)
 - Open/Create/Replace File, 57
 - Output Requirements, 56
 - overtones, § 2.1(23), 23, 25
- P** partial, § 7.1(105), 105, § 7.2(107), 107, § 7.3(109), § 7.4(112)
 - partials, 105, 107, 108
 - patched, 34
 - patches, 34
 - perfect fifth, 28
 - phase function, § 6.5(95)
 - physical modeling, 115, 121
 - pitch, § 2.1(23), 23, 24
 - pitch shifter, § 6.2(87), 88, § 6.3(89), 89
 - pitch shifting, § 6.2(87)
 - pitch wheel, § 4.1(41), 46
 - Pitch Wheel Change, 46
 - Play Waveform, 20
 - Play Waveform Express VI, § 1.6(20)
 - plucked string algorithm, § 8.7(128)
 - portamento, 36, 40
 - power, 24
 - pre-filter, § 6.2(87)
 - Pre-filtering, 89
 - principal alias, 121
 - probes, 11
 - processors, 34
 - program change, § 4.1(41), 45
 - program number, 45
 - programmed controller, 35
 - pulse train, § 8.1(115), 116, § 8.3(120), 121
 - Pythagorean tuning, 29
- Q** Quick Scale, 22, 22
 - QuickScale, 39
- R** Ramp Pattern, § 1.5(18), 19
 - rate, 71, 76
 - ratio, 24, 24
 - re-entrant, § 1.8(20), 20
 - real-time audio, § 1.7(20)
 - removing, 12
 - rendering, § 4.6(64)
 - replace, § 1.2(12)
 - replacing, § 1.2(12), 12, 12
 - replicating, § 1.2(12), 12
 - reshape, § 1.5(18)
 - reshaping, 19
 - Retain Wire Values, 11
 - reverb time, § 9.1(137), 138
 - reverberation, § 9.1(137), 137, § 9.2(141), 141
 - ring modulation, 83, § 6.3(89)
 - Ring modulation (AM), 89
 - running status, § 4.2(48), 48, 49, 55
- S** scales, § 2.2(27)
 - scaling, § 1.9(21)
 - Schroeder reverberator, 141, 141, 142
 - screencast, § (1), 9
 - Screencasts, 1

- selector terminal, 16
- semitone, 28
- semitones, 26
- sequencer, 48
- sequencer application, 48
- sequencer-specific, 50
- sequencers, § 4.2(48)
- set-tempo, 50
- sharps, 27
- sidebands, § 6.4(91), 93
- Simple Read, 21, 21
- Simple Write, 21, 21
- sine wave, § 1.8(20), 20
- single sideband modulation, § 6.2(87)
- single-sideband (SSB) modulation, 88
- Single-sideband AM (SSB-AM), 89
- single-sideband modulation, § 6.3(89)
- single-stepping, 11
- sinusoid, § 1.8(20)
- sinusoidal oscillator, § 6.5(95)
- Sinusoidal source, § 1.6(20)
- SMF, 48
- Sound File Simple Read, 21
- Sound File Simple Write, 21
- Sound Output, § 1.7(20), 20
- soundcard, § 1.6(20)
- SoundGen.vi, 38, 38, 39
- sources, 34
- spectral envelope, § 8.5(124), § 8.6(126)
- spectrogram, § 7.4(112)
- speech, § 8.4(122)
- spreadsheet, § 1.5(18)
- SSB, § 6.2(87)
- standard MIDI file, 48, § 4.4(52), § 4.5(55), § 4.6(64)
- status byte, § 4.1(41), 41
- stereo, § 1.9(21)
- string, 10, 16
- String Length, 59
- subarray, § 1.5(18)
- subarrays, 18
- subdiagram, 15, 15
- subroutine, § 1.4(18)
- subtractive synthesis, § 8.1(115), 115, § 8.3(120), 121, 125
- subVI, 3, § 1.4(18), 18, 57
- subVIs, 9
- SysEx, 47
- System Exclusive, 47
- System Exclusive End, 47
- System Exclusive Start, 47

- system-exclusive, § 4.1(41), 50

T

- t values, 38
- t2mf, 51
- terminal, 12
- terminals, 10, § 1.4(18)
- text labels, § 1.2(12), 12
- tick, 49
- ticks per quarter note, 49
- timbre, 25, § 7.2(107), 108
- time-varying digital filter, § 8.1(115), § 8.2(119), § 8.5(124), § 8.6(126)
- time-varying spectra, § 6.6(96)
- To Unsigned Long Integer, 59
- To Variant, 58
- track chunk, § 4.2(48), 49, § 4.5(55)
- track chunks, 48
- track length, 49
- tracks, 48
- trajectory, 24, 25
- tremolo, § 5.1(69), 71, § 5.2(72)
- truncated Fourier series, § 8.3(120)
- tuning system, 26
- tuning systems, § 2.1(23), 23
- two-pole resonator, § 8.4(122)

U

- UART, 41
- USB, 43

V

- variable-length format, § 4.2(48), 48, 49, 49, § 4.5(55)
- Variant to Flattened String, 58
- VCA, § 3.1(33), 34, 34
- VCF, § 3.1(33), 34
- VCO, § 3.1(33), 34, 34
- velocity, 44
- velocity profile, 63
- VI, 9
- VI Description, 18
- VI Properties, 18
- vibraphone, § 5.1(69), 69, § 5.2(72), 73
- vibrato, 71, § 5.3(75), 76, § 5.4(78)
- virtual instrument, 9
- virtual instrument (VI), § 1.1(9)
- virtual instruments (VIs), 1
- virtual musical instrument, 64, § 5.2(72), 73, 74, § 5.4(78), 79, 80, § 6.7(98), 100, § 8.7(128), 130, § 8.8(130), 135
- virtual musical instruments, 4, 99
- VLF, 49, 49
- VMI, 64, § 5.2(72), 73, 74, 79, 80, § 6.7(98), 100, 130, 135

- VMIIs, 99
 - vocal tract, § 8.4(122), 123, § 8.5(124), § 8.6(126)
 - voltage-controlled amplifier, 34
 - voltage-controlled filter, 34
 - voltage-controlled oscillator, 34
- W** WAV file, § 1.9(21)
- waveform, 20
 - well-tempered, 29
 - While Loop, 10, § 1.3(12), 12, 15, 15
 - while-loop structure, 85
- white noise, 116
 - white noise burst, § 8.7(128)
 - wideband, 116
 - wideband source, § 8.1(115)
 - wires, 9
 - Write, 20
 - write clear, § 1.7(20)
 - Write to Binary File, 57
- X** XVI32, § 4.3(51), 51
- Y** Yamaha DX7, § 6.4(91)

Attributions

Collection: *Musical Signal Processing with LabVIEW (All Modules)*

Edited by: Sam Shearman

URL: <http://cnx.org/content/col10507/1.3/>

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Musical Signal Processing with LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15510/1.3/>

Pages: 1-7

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Getting Started with LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14764/1.4/>

Pages: 9-12

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Editing Tips for LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14765/1.3/>

Page: 12

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Essential Programming Structures in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14766/1.6/>

Pages: 12-17

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Create a SubVI in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14767/1.4/>

Page: 18

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Arrays in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14768/1.4/>

Pages: 18-19

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Audio Output Using LabVIEW's "Play Waveform" Express VI"

By: Ed Doering

URL: <http://cnx.org/content/m14769/1.3/>

Page: 20

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Real-Time Audio Output in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14772/1.5/>

Page: 20

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Audio Sources in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14770/1.3/>

Pages: 20-21

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Reading and Writing Audio Files in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m14771/1.6/>

Pages: 21-22

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Perception of Sound"

By: Ed Doering

URL: <http://cnx.org/content/m15439/1.2/>

Pages: 23-27

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Musical intervals and the equal-tempered scale"

By: Ed Doering

URL: <http://cnx.org/content/m15440/1.1/>

Pages: 27-30

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Analog Synthesis Modules"

By: Ed Doering

URL: <http://cnx.org/content/m15442/1.2/>

Pages: 33-36

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Compose a piece of music using analog synthesizer techniques"

By: Ed Doering

URL: <http://cnx.org/content/m15443/1.2/>

Pages: 36-39

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "MIDI Messages"

By: Ed Doering

URL: <http://cnx.org/content/m15049/1.2/>

Pages: 41-47

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Standard MIDI Files"

By: Ed Doering

URL: <http://cnx.org/content/m15051/1.3/>

Pages: 48-51

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Useful MIDI Software Utilities"

By: Ed Doering

URL: <http://cnx.org/content/m14879/1.2/>

Pages: 51-52

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Parse and analyze a standard MIDI file"

By: Ed Doering

URL: <http://cnx.org/content/m15052/1.2/>

Pages: 52-55

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Create standard MIDI files with LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15054/1.2/>

Pages: 55-64

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[LabVIEW application] MIDI_JamSession"

By: Ed Doering

URL: <http://cnx.org/content/m15053/1.2/>

Pages: 64-66

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Tremolo Effect"

By: Ed Doering

URL: <http://cnx.org/content/m15497/1.2/>

Pages: 69-71

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Vibraphone virtual musical instrument (VMI) in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15498/1.2/>

Pages: 72-75

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Vibrato Effect"

By: Ed Doering

URL: <http://cnx.org/content/m15496/1.2/>

Pages: 75-77

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] "The Whistler" virtual musical instrument (VMI) in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15500/1.1/>

Pages: 78-81

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Amplitude Modulation (AM) Mathematics"

By: Ed Doering

URL: <http://cnx.org/content/m15447/1.2/>

Pages: 83-86

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Pitch Shifter with Single-Sideband AM"

By: Ed Doering

URL: <http://cnx.org/content/m15467/1.2/>

Pages: 87-89

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Ring Modulation and Pitch Shifting"

By: Ed Doering

URL: <http://cnx.org/content/m15468/1.1/>

Pages: 89-91

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Frequency Modulation (FM) Mathematics"

By: Ed Doering

URL: <http://cnx.org/content/m15482/1.2/>

Pages: 91-95

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Frequency Modulation (FM) Techniques in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15493/1.2/>

Page: 95

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Chowning FM Synthesis Instruments in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15494/1.2/>

Pages: 96-98

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Chowning FM Synthesis Instruments"

By: Ed Doering

URL: <http://cnx.org/content/m15495/1.1/>

Pages: 98-100

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Additive Synthesis Techniques"

By: Ed Doering

URL: <http://cnx.org/content/m15445/1.3/>

Pages: 105-107

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Additive Synthesis Concepts"

By: Ed Doering

URL: <http://cnx.org/content/m15444/1.2/>

Pages: 107-109

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Risset Bell Synthesis"

By: Ed Doering

URL: <http://cnx.org/content/m15476/1.1/>

Pages: 109-112

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Spectrogram Art"

By: Ed Doering

URL: <http://cnx.org/content/m15446/1.1/>

Pages: 112-114

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Subtractive Synthesis Concepts"

By: Ed Doering

URL: <http://cnx.org/content/m15456/1.1/>

Pages: 115-119

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Interactive Time-Varying Digital Filter in LabVIEW"

By: Ed Doering

URL: <http://cnx.org/content/m15477/1.2/>

Page: 119

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Band-Limited Pulse Generator"

By: Ed Doering

URL: <http://cnx.org/content/m15457/1.3/>

Pages: 120-122

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Formant (Vowel) Synthesis"

By: Ed Doering

URL: <http://cnx.org/content/m15459/1.2/>

Pages: 122-124

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Linear Prediction and Cross Synthesis"

By: Ed Doering

URL: <http://cnx.org/content/m15478/1.2/>

Pages: 124-126

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "[mini-project] Linear Prediction and Cross Synthesis"

By: Ed Doering

URL: <http://cnx.org/content/m15479/1.1/>

Pages: 126-127

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Karplus-Strong Plucked String Algorithm"

By: Ed Doering

URL: <http://cnx.org/content/m15489/1.2/>

Pages: 128-130

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Karplus-Strong Plucked String Algorithm with Improved Pitch Accuracy"

By: Ed Doering

URL: <http://cnx.org/content/m15490/1.2/>

Pages: 130-135

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Reverberation"

By: Ed Doering

URL: <http://cnx.org/content/m15471/1.2/>

Pages: 137-140

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Schroeder Reverberator"

By: Ed Doering

URL: <http://cnx.org/content/m15491/1.2/>

Pages: 141-144

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Module: "Localization Cues"

By: Ed Doering

URL: <http://cnx.org/content/m15475/1.2/>

Pages: 144-146

Copyright: Ed Doering

License: <http://creativecommons.org/licenses/by/2.0/>

Musical Signal Processing with LabVIEW (All Modules)

"Musical Signal Processing with LabVIEW," a multimedia educational resource for students and faculty, augments traditional DSP courses and supports dedicated courses in music synthesis and audio signal processing. Each of the learning modules blends video, text, sound clips, and LabVIEW virtual instruments (VIs) into explanation of theory and concepts, demonstration of LabVIEW implementation techniques to transform theory into working systems, and hands-on guided project activities. Screencasts – videos captured directly from the computer screen with audio narration and a hallmark of this resource – use a mixture of hand-drawn text, animations, and video of the LabVIEW tool in operation to provide a visually rich learning environment. *** Unlike other collections that contain sub-sets of modules from this course, this collection contains all modules. ***

About Connexions

Since 1999, Connexions has been pioneering a global system where anyone can create course materials and make them fully accessible and easily reusable free of charge. We are a Web-based authoring, teaching and learning environment open to anyone interested in education, including students, teachers, professors and lifelong learners. We connect ideas and facilitate educational communities.

Connexions's modular, interactive courses are in use worldwide by universities, community colleges, K-12 schools, distance learners, and lifelong learners. Connexions materials are in many languages, including English, Spanish, Chinese, Japanese, Italian, Vietnamese, French, Portuguese, and Thai. Connexions is part of an exciting new information distribution system that allows for **Print on Demand Books**. Connexions has partnered with innovative on-demand publisher QOOP to accelerate the delivery of printed course materials and textbooks into classrooms worldwide at lower prices than traditional academic publishers.