Mads G. Christensen

Introduction to Audio Processing

Springer

Introduction to Audio Processing

Mads G. Christensen

Introduction to Audio Processing

Springer

Mads G. Christensen
Aalborg University
Aalborg, Denmark

ISBN 978-3-030-11780-1 ISBN 978-3-030-11781-8 (eBook)
https://doi.org/10.1007/978-3-030-11781-8

Library of Congress Control Number: 2019933295

This Springer imprint is published by the registered company Springer Nature Switzerland AG.
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

To my children
Silke, Freja, and Erik

Preface

The present book is intended as an introduction to signal processing based on (and for) audio applications for students in media technology bachelor (or similar) programs. It was written based on the material I had developed teaching a course in the B.Sc. program at Aalborg University called Medialogy over a number of years. I took over the course gradually from a colleague in 2011 when the course was called Audio Design and had more of an artistic bent, and I have taught it every Spring since then. When I took over the course, I had the ambition of teaching the students some fundamental signal processing, as I believe it is an extremely powerful and useful tool when solving engineering problems in media technology and for understanding the technology that we use. I quickly came to realize that it was difficult to find suitable literature to base the course on. On one hand, most textbooks on signal processing are developed for electrical engineering programs and are unsuitable for students such as those in our media technology program, which are not as mathematically inclined as electrical engineering students typically are. On the other hand, books on audio effects, sound synthesis, and the like very often lack technical depth, and I felt that important, universally applicable knowledge and skills were thus lost. As a result, with no one book available, I had to rely on a variety of heterogeneous sources for the course, something that became a bit of a challenge for both my students and myself. Over the following years, I refined my course notes and the contents of the course until I was (almost) satisfied with it in the Spring of 2016 and started writing this book.

My philosophy behind the approach taken in this book is that math is at the heart of audio processing, indeed much of media technology, including image processing, computer vision, and computer graphics, and that the math should not be removed or dumbed down because it is difficult to understand or teach. It is easy to strip away the math and simply focus on instances of technologies, but very important universal knowledge and insight is lost this way. Rather, the math should not be stripped away, but it should be presented in a different way, and students should work with the contents in different ways. Accordingly, I have tried to retain the core of signal processing, such as the concepts of difference equations, convolution, and the Fourier transform, but attempt to present them in a context

where they make immediate sense to the student and can readily be applied and used to build interesting artifacts. Each chapter in the book builds on the previous ones, and each chapter is written as a linear, coherent story. Moreover, the exercises in each chapter are mostly focused on applications of the concepts to audio signals, like, for example, building audio effects. The exercises are made specifically for Pure Data (Pd), a graphical programming language and environment for interactive audio, although other and more traditional software, such as MATLAB, can of course be used. The exercises and the hints in the book are tailored specifically for Pure Data. At the time of writing, it is recommended that the reader uses the Purr Data implementation of Pure Data which can be downloaded from GitHub.[1] Also, an audio editor also occasionally becomes useful to anybody working with audio, and for this, I recommend Audacity, which is freely available.[2] Although the book contains an introduction to Pure Data in the appendix, I would also encourage students to make use of the many excellent tutorials that can be found online, including the many video tutorials on YouTube. Additional material, such as Pure Data examples, is also available on my Aalborg University homepage,[3] and videos of my lectures can be found on my YouTube channel.[4]

In closing, I would like to thank a number of people who have contributed, in one way or another, to the book. Firstly, I thank Jedrzej Jacek Czapla for making all the nice illustrations in the book. I would also like to thank all the students, present and former, who have followed my course Audio Processing in the Medialogy B.Sc. program at Aalborg University as they have helped define and refine the material. However, I am particularly indebted to the students who followed the course in the Spring of 2017, as they had to suffer through the first drafts of the book and the consequences of the major overhaul of the course that followed the book's creation, something that might have seemed inconvenient at the time, as the course had been running quite smoothly for some years. I would also like to thank my teaching assistants in the course, namely, my Ph.D. students Martin Weiss Hansen, Mathew Shaji Kavalekalam, and Alfredo Esquivel, for their comments and suggestions that helped tremendously in improving the book. I also wish to thank my friend, and now former colleague at AAU, Bob Sturm, who encouraged me to write more lecture notes when I had only written a few to patch up the biggest glaring holes in the literature and thus got me started on the book. Finally, I thank my family, namely, my kids Silke, Freja, and Erik and my loving wife Mia, for their patience with my professor ways.

Aalborg, Denmark
January 9, 2019

Mads G. Christensen

[1] https://github.com/agraef/purr-data/releases.
[2] https://www.audacityteam.org.
[3] http://madsgc.blog.aau.dk.
[4] https://goo.gl/QO8aiD.

Contents

About the Author

Mads G. Christensen received the M.Sc. and Ph.D. degrees in 2002 and 2005, respectively, from Aalborg University (AAU) in Denmark, where he is also currently employed at the Department of Architecture, Design & Media Technology as Full Professor in Audio Processing and is head and founder of the Audio Analysis Lab. He was formerly with the Department of Electronic Systems at AAU and has held visiting positions at Philips Research Labs, ENST, UCSB, and Columbia University.

He has published 3 books and more than 200 papers in peer-reviewed conference proceedings and journals and has given tutorials at major conferences such as ICASSP, EUSIPCO, and INTERSPEECH and a keynote talk at IWAENC. His research interests lie in audio and acoustic signal processing where he has worked on topics such as microphone arrays, noise reduction, signal modeling, speech analysis, audio classification, and audio coding.

Dr. Christensen has received several awards, including best paper awards, the Spar Nord Foundation's Research Prize, a Danish Independent Research Council Young Researcher's Award, the Statoil Prize, and the EURASIP Early Career Award. He is a beneficiary of major grants from Independent Research Fund Denmark, the Villum Foundation, and Innovation Fund Denmark. He is a former associate editor for IEEE/ACM Transactions on Audio, Speech, and Language Processing and IEEE Signal Processing Letters, a member of the IEEE Audio and Acoustic Signal Processing Technical Committee, and a founding member of the EURASIP Special Area Team in Acoustic, Sound, and Music Signal Processing. He is senior member of the IEEE, member of EURASIP, and member of the Danish Academy of Technical Sciences.

Abbreviations

ADC	Analog-to-Digital Converter
BP	Bandpass
CD	Compact Disc
DAC	Digital-to-Analog Converter
DAFX	Digital Audio Effects
DVD	Digital Versatile Disc
FFT	Fast Fourier Transform
FIR	Finite Impulse Response
FT	Fourier Transform
FX	Effects
HP	Highpass
HRTF	Head-Related Transfer Function
IIR	Infinite Impulse Response
IR	Impulse Response
LP	Lowpass
MIDI	Musical Instrument Digital Interface
Pd	Pure Data
RMS	Root Mean Square
SNR	Signal-to-Noise Ratio
TF	Transfer Function
VoIP	Voice over IP

Definitions and Notation

$x(t)$	Analog (continuous) signal
x_n	Input signal
x_{n-1}	Input signal delayed 1 sample
x_{n-D}	Input signal delayed D samples
y_n	Output signal
h_n	Impulse response
f_s	Sampling frequency in Hz
T_s	Sampling period in s, $f_s = 1/T_s$
δ_n	Kronecker delta function, $\delta_n = 1$ for $n = 0$ otherwise 0
j	Imaginary unit operator $j^2 = -1$
$\angle\cdot$	Argument (angle) of complex number
$\lvert\cdot\rvert$	Magnitude of complex number
Real$\{\cdot\}$	Real value of complex number
Imag$\{\cdot\}$	Imaginary value of complex number
$X(\omega)$	Fourier transform of x_n
$\sum_n x_n e^{-j\omega n}$	Fourier transform of x_n
f	Frequency in Hz
$\omega = 2\pi \frac{f}{f_s}$	Digital frequency in radians
Ω	Analog frequency in radians
$X(z)$	Z-transform of x_n
$\sum_n x_n z^{-n}$	Z-transform of x_n
$H(z)$	Transfer function
$H(\omega)$	Frequency response
$\angle H(\omega)$	Phase response
$\lvert H(\omega)\rvert$	Magnitude response
$\sum_{i=1}^{I} x_i$	$x_1 + x_2 + \ldots + x_I$
$\prod_{i=1}^{I} x_i$	$x_1 \cdot x_2 \cdot \ldots \cdot x_I$
z^{-1}	Unit delay operator
z^{-D}	D-sample delay operator
$\tilde{x}_n$	Signal level
$\tilde{X}_n$	Input signal level in dB
$*$	Convolution

List of Figures

List of Tables

Chapter 1
What Is Sound?

1.1 Introduction

Sound is an integral part of our lives. It is one of our most important ways of sensing the world around us. Moreover, it is fundamental to our communication with our surroundings and fellow human beings. It is also a means for artistic expression and experience, in the form of, for example, music. For most of us, it would be hard to imagine a life without sound.

Sound is often defined as either referring to an auditory sensation or the disturbance in a medium that causes such an auditory sensation. As a physical phenomenon, which is the way that we will here look at sound, it refers to waves that originate somewhere and then travel through some medium to another place, where they can be heard or measured. Such sound waves travel in solids, liquids, and gas, and they come in two forms, namely longitudinal and transversal. In longitudinal waves, the displacement of the medium occurs in the direction of the propagation of the wave while in transversal waves, the displacement is orthogonal to the direction. Both types of waves cause changes in the pressure, measured at single point, which can cause an auditory sensation. Sound waves can be created by vibrating bodies, changing airflow, heat sources, etc. It is clear from this that there are some physics involved with the creation and propagation of sound.

This book is concerned with what is called *audio processing* which is the theory behind and methods for the processing of audio signals. Audio signals simply refer to audible sound signals, while a signal is something that is measured over time (and/or space). That something is, in this context, pressure, because what we perceive with our ears are changes in pressure. What, then, does processing mean? Processing refers to basically anything we might want to do with or to audio signals. We might wish to manipulate audio signals for artistic purposes, to create music, we might wish to compress speech or music signals so that they can be transmitted (like in Skype or our smart phones) or stored efficiently (like mp3, OGG, or AAC), we might wish to identify which person is speaking or what he/she is saying, determine

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_1

from where a sound is coming, and so forth. In media technology, an example of audio processing could be a method for making recorded speech for a computer game sound like it is being spoken in a certain acoustic environment or from a certain angle. It could also be methods for automatically generating contents, such as background music and environmental sounds, for a computer game, or it could be audio effects for live music performance or equipment for recording and producing music.

Before going into audio processing, it is important that we first understand the nature of audio signals. We will, therefore, start out this book by exploring what sound is as a physical phenomenon.

1.2 Simple Vibrations

Before we can do any processing of audio signal, we must first understand what audio and sound even is in the first place. The following is based on [1, 2]. To gain some understanding of this, let us consider a simple example, based on a bar. Vibrating bars are used to create music notes in several instruments, like glockenspiel, marimbas, and xylophones. Suppose we strike such a bar, which is attached at one end, with a hammer of sorts so that it bends, having a displacement $x(t)$ at time t. Due to the stiffness of the bar, there will be a restoring force, F. This is illustrated in Fig. 1.1. The restoring force acts like a spring, so the restoring force is proportional to the displacement, i.e.,

$$F = -kx(t), \tag{1.1}$$

where k is a physical constant for this particular bar, describing its physical properties. We would like to understand how the bar moves, i.e., how $x(t)$ behaves. According to Newton's second law, this force is also equal to $F = ma$, where m is

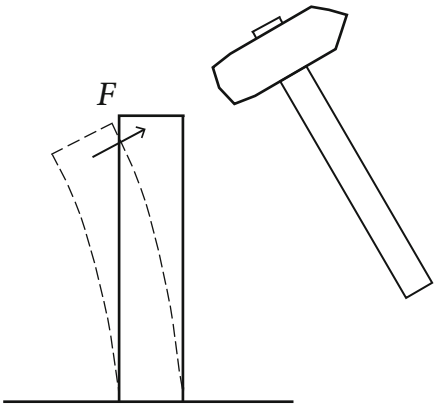


Fig. 1.1 A bar is struck by a hammer, which causes the bar to bend. There is a restoring force, F, that forces the bar towards its equilibrium due to the stiffness of the bar

the mass of the bar. Hence, we have that

$$ma = -kx(t). \tag{1.2}$$

We can relate the acceleration, a, to the displacement, $x(t)$, since the acceleration is the derivative of the velocity at which the bar moves. The velocity is defined as

$$v(t) = \frac{dx(t)}{dt}, \tag{1.3}$$

and the acceleration is the derivative of the velocity, i.e.,

$$a = \frac{dv}{dt} = \frac{d^2x(t)}{dt^2}. \tag{1.4}$$

Inserting this expression back into (1.2), we get

$$m\frac{d^2x}{dt^2} = -kx(t). \tag{1.5}$$

We can rearrange this slightly to get

$$\frac{d^2x(t)}{dt^2} = -\frac{k}{m}x(t). \tag{1.6}$$

The equation can be understood as follows. Due to the physics of the system, there is a relation between the displacement, $x(t)$, and the physical constant, k, and the mass, m. Moreover, the displacement cannot just be anything, it must obey Eq. (1.6). We now ask ourselves, what $x(t)$ obeys this equation? In other words, what kind of function will we see by observing the displacement over time after the bar has been hit by the hammer? The answer is: sinusoids! Hence, the bar will vibrate. This is illustrated in Fig. 1.2. Before going into more details about all this, let us consider the simplest form of such a sinusoid:

$$x(t) = \sin(\Omega t), \tag{1.7}$$

which has a frequency of Ω in radians per second (rad/s). In Fig. 1.3, such a sinusoidal displacement function is shown. The derivative of this sinusoid with respect to time is

$$\frac{d}{dt}\sin(\Omega t) = \Omega\cos(\Omega t). \tag{1.8}$$

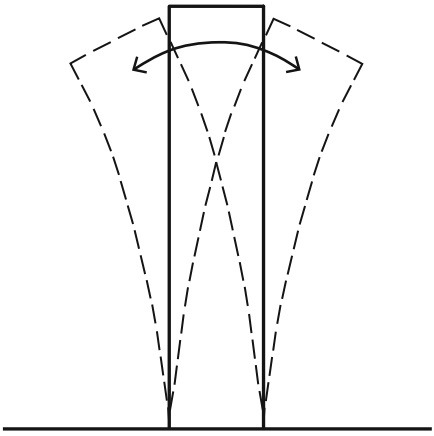


Fig. 1.2 After the initial strike by the hammer, the restoring force will cause the bar to vibrate. The displacement function, as observed over time, is sinusoidal

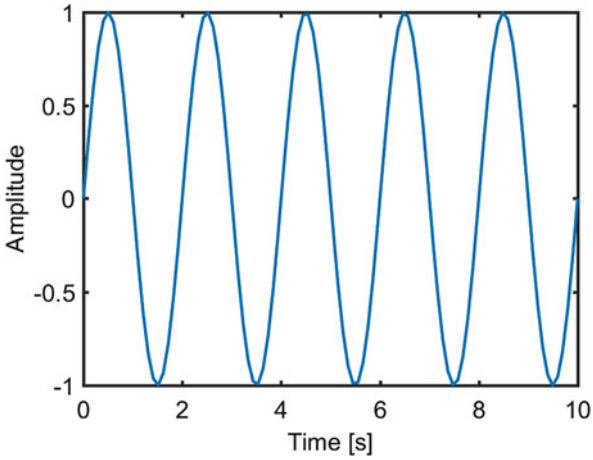


Fig. 1.3 Example of solution of the differential equation (1.6), i.e., the sinusoidal displacement function in (1.7), with $\Omega = 2\pi 0.5$ rad/s

Differentiating this result once again yields

$$\frac{d^2}{dt^2}\sin(\Omega t) = -\Omega^2 \sin(\Omega t). \tag{1.9}$$

By substituting $x(t)$ in (1.6) by (1.7) and inserting the second-order derivative in (1.9), we get

$$-\Omega^2 \sin(\Omega t) = -\frac{k}{m}\sin(\Omega t). \tag{1.10}$$

This confirms that a displacement of the form (1.7) indeed does obey the laws of physics we established for the displacement. Moreover, it also tells us, by comparing the left- and right-hand sides of (1.10), that

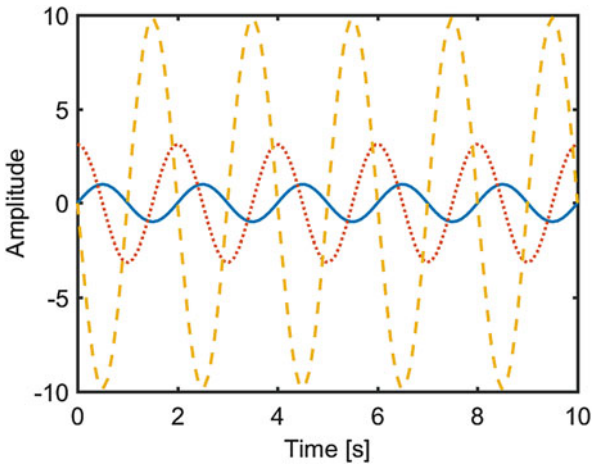


Fig. 1.4 Example of solution (1.7) (solid), and its first-order derivative (1.8) (dotted), and second-order derivative (1.9) (dashed) for $\Omega = 2\pi 0.5$ rad/s

$$\Omega = \sqrt{\frac{k}{m}}, \tag{1.11}$$

i.e., that not just any frequency Ω will do, only one that is determined by that physical constant k of the bar and its mass m will work. Recall that the frequency Ω is related to the propagation speed of the wave in the medium, c, which we will learn more about later, via the wavelength, λ, i.e.,

$$c = \frac{1}{2\pi}\Omega\lambda. \tag{1.12}$$

Interestingly, the sinusoid in (1.7) is not the only solution. A cosine would work just as well, with $x(t) = \cos(\Omega t)$, and so would a scaled or time-shifted sine or cosine function. In Fig. 1.4, an example of the solution in (1.7) is shown for a frequency of $\Omega = 2\pi 0.5$ rad/s along with its derivatives in (1.8) and (1.9), respectively.

So how do the vibrations in the bar turn into sound? The movement of the bar, which we now know must be sinusoidal, causes the air around it to move in a similar fashion, generating a wave that travels from the bar, and depending on the power of the sinusoid and its frequency, this wave can be heard by a listener. For an in-depth treatment of the physics of musical instruments, we refer the interested reader to [3].

1.3 About Sinusoids

Any function of the form

$$x(t) = A\cos(\Omega t + \Phi) \tag{1.13}$$

can be shown to satisfy (1.6) and is thus also a solution. Note that both cosine and sine functions are special cases of this more general function, i.e., with $\Phi = 0$ we get the cosine function and with $\Phi = -\pi/2$ we get the sine function. In this text, we therefore generally refer to all such functions or signals as simply sinusoids. We will now go into a bit more details about sinusoids. The sinusoidal signal in (1.13) has some parameters. These are

- The frequency, Ω, which is here measured in radians per second and determines the number of cycles (in radians) of the sinusoid we get per second. Perceptually, it determines the pitch of the sinusoid in (1.13). It is related to the frequency, f, in Hertz as $\Omega = 2\pi f$. In Fig. 1.5, sinusoids of the form $x(t) = 2\cos(\Omega t)$ are shown for three different frequencies, namely 1, 5, and 10 Hz.
- The amplitude, $A \geq 0$, which scales the sinusoid so that instead of going from -1 and 1, it goes from $-A$ to A. It essentially determines how loud the sinusoid is. The power of a sinusoid is $A^2/2$. In Fig. 1.6 the effect of the amplitude is illustrated for a sinusoid having the same frequency but three different amplitudes, namely 1, 2, and 5.
- The phase, Φ, which can be thought of as a time-shift of the sinusoids by observing that (1.13) can be rewritten as $\cos(\Omega t + \Phi) = \cos(\Omega(t - t_0))$, where t_0 is then the time-shift corresponding to a phase Φ. Note how the same time-shift t_0 results in different phases for different frequencies. This is depicted in Fig. 1.7, which shows $2\cos(2\pi 10(t - t_0))$ for three different values of t_0. As can be seen, the sinusoid is shifted right by t_0.

We can thus manipulate sinusoids and understand these manipulations in terms of changes to the aforementioned parameters or qualities. For example, making A larger increases the loudness of the sinusoid and increasing Ω increases the pitch. Many of the manipulations we will discuss in this book are based on filters. As we will see, a filter changes the amplitude and the phase of a sinusoid when it passes

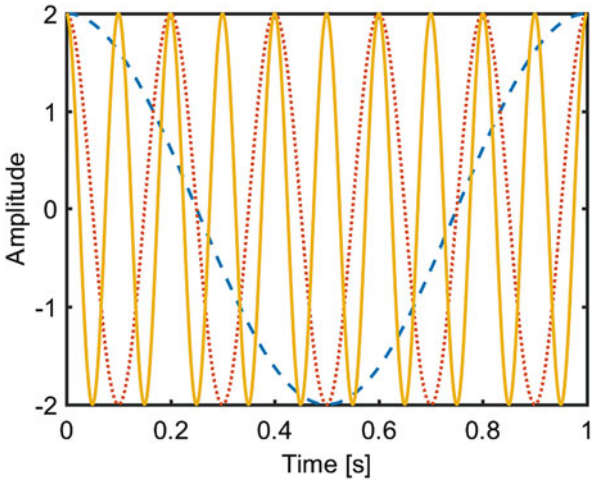


Fig. 1.5 Sinusoids of the form $x(t) = 2\cos(\Omega t)$ for three frequencies, i.e., 1 (dashed), 5 (dotted), and 10 (solid) Hz

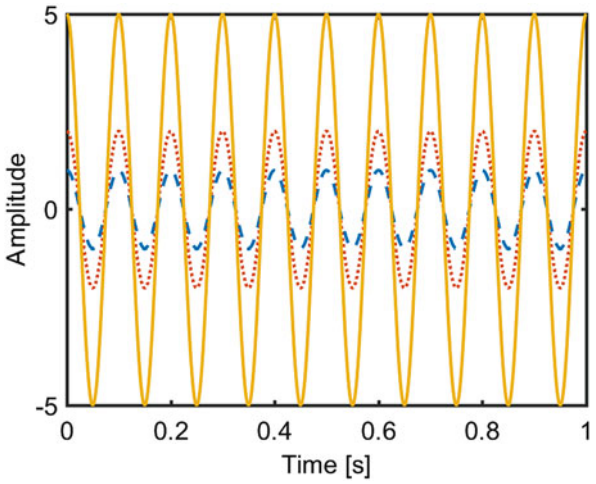


Fig. 1.6 Example of sinusoids having different amplitudes, here 1 (dashed), 2 (dotted), and 5 (solid)

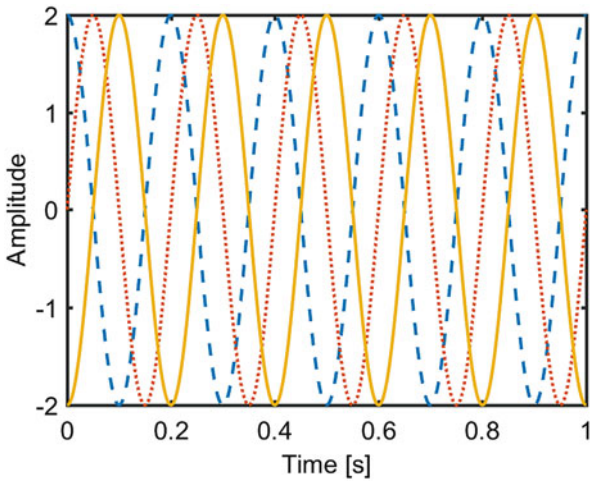


Fig. 1.7 Sinusoids of the form $2\cos(2\pi 10(t - t_0))$ for three different time-shifts, t_0, namely 0 (dashed), 0.25 (dotted), and 0.5 (solid) s

through the filter. The frequency, however, is not changed by filters! Regarding the perceptual attributes of sound, we refer the interested reader to the textbook [4].

That the parameters of the sinusoid (1.13) are simply related to its perceptual properties seems to indicate that sinusoids are something fundamental to sound. Indeed this is the case, although sound in general is more complicated than the sinusoid in (1.13). However, a fundamental result in mathematics is that practically all[1] signals can be thought of as comprised of sinusoids. This is the result of the famous Fourier transform. It basically means that we can think of sinusoids as

[1]The Fourier transform can be shown to exist for a broad class of signals, including absolutely summable/integrable signals, which includes most real-world signals.

the atoms of sound. We will return to these concepts in detail later, but for now just remember that signals can be thought of as the sum of many sinusoids of the form (1.13), having different amplitudes, phases, and frequencies. The effects of filters are then that they change these amplitudes and the phases of the sinusoids according to our design.

Before moving on to introduce complex numbers in the next section and then, later, the phasor, let us see if we can separate the phase and the amplitude (which can be changed by filters) from $\cos(\Omega t)$ (which cannot be changed by filters). Perhaps we can understand how it is mathematically possible to change the amplitude and phase of a sinusoid. According to the angle addition formula for cosine, we can write (1.13) as

$$x(t) = A\cos(\Omega t + \Phi) \tag{1.14}$$

$$= A\cos(\Omega t)\cos(\Phi) - A\sin(\Omega t)\sin(\Phi). \tag{1.15}$$

This still looks somewhat convoluted, and it does not appear to be straightforward to simplify it further. It seems like manipulations of amplitude and phase are going to be complicated (or complex, if you will).

1.4 Complex Numbers

In this section, we will introduce complex numbers. As it turns out, they are tremendously useful when trying to understand sound and filters. A complex number is essentially a two-dimensional representation of the form

$$z = a + jb. \tag{1.16}$$

The number is comprised of two parts, namely a and b, which can be considered coordinates in a plane. They are called cartesian coordinates and this is called the *cartesian* representation (or sometimes the *rectangular* form). j is the so-called imaginary unit while a is called the real part and b is called the imaginary part. Any time we see a complex number, the part that is multiplied by j is the imaginary part while the part that does not is the real part. The real part of the number z is sometimes written as $\text{Real}\{z\}$ while the imaginary part is written as $\text{Imag}\{z\}$. Thus, in our case

$$\text{Real}\{z\} = a \tag{1.17}$$

$$\text{Imag}\{z\} = b. \tag{1.18}$$

Complex numbers are often depicted in a two-dimensional coordinate system, where the x-axis is the real part and the y-axis is the imaginary part. Note that the normal real numbers are special cases of complex numbers, namely where the imaginary

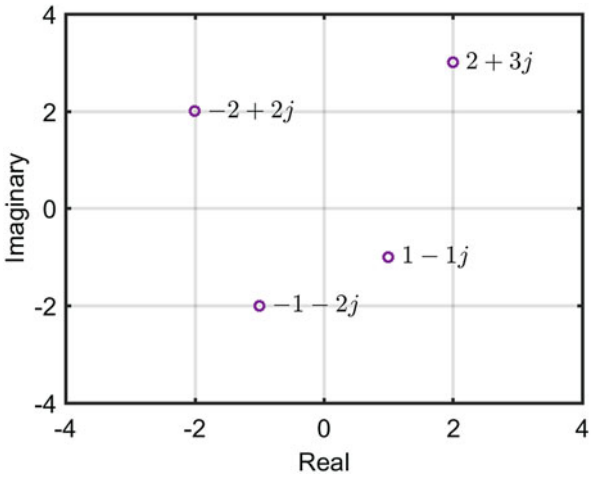


Fig. 1.8 Examples of complex numbers

part is zero. They thus reside on the x-axis in this two-dimensional coordinate system. Some examples of complex numbers are depicted in Fig. 1.8. So, what is special or particularly clever about complex numbers? Nothing, so far. The whole essence of complex numbers has to do with j, and the only thing to remember when doing math with complex numbers is that j multiplied by itself is -1, i.e.,

$$j^2 = -1 \quad \text{and} \quad j = \sqrt{-1}. \tag{1.19}$$

This might seem quite simple, but the consequences of this are quite profound, as we shall see. Let us now consider how to do algebra with complex numbers. Algebra is performed as with real numbers, only we must keep track of the real and imaginary parts and remember the aforementioned rule. Suppose we have two complex numbers

$$z_1 = a_1 + jb_1 \quad \text{and} \quad z_2 = a_2 + jb_2, \tag{1.20}$$

and we would like to add them, i.e.,

$$z_1 + z_2 = a_1 + jb_1 + a_2 + jb_2 \tag{1.21}$$

$$= (a_1 + a_2) + j\,(b_1 + b_2)\,. \tag{1.22}$$

We see that the result is simply that the real part of the result is the sum of the real parts of z_1 and z_2 and that the imaginary part is the sum of the imaginary parts of the two numbers. Subtraction works the same way. Let us now try the multiplication of the same two numbers:

$$z_1 z_2 = (a_1 + jb_1)\,(a_2 + jb_2) \tag{1.23}$$

$$z_1 z_2 = a_1 a_2 + ja_1 b_2 + jb_1 a_2 + j^2 b_1 b_2. \tag{1.24}$$

Now, if we use that $j^2 = -1$ and collect the imaginary and real parts, we get

$$z_1 z_2 = (a_1 a_2 - b_1 b_2) + j\,(a_1 b_2 + a_2 b_1)\,. \tag{1.25}$$

We can see that despite the result perhaps seeming a bit unexpected, the involved algebra is quite simple. We will later investigate the geometrical meaning of the multiplication of complex numbers. There is an operation on complex numbers that will be useful later. It is called the complex conjugation. The complex conjugate of a complex number $z = a + jb$ is written as z^* and is given by

$$z^* = a - jb, \tag{1.26}$$

that is, it simply means that we change the sign on the imaginary part. As we will see shortly, it has a simple geometric meaning.

Since we can think of complex numbers as two-dimensional numbers, we can also think of them as having a length and an angle. For complex numbers, these are called the *magnitude* r, which is the distance from the origin to the point z, and the *argument* ψ, which it forms with the x-axis, and express them in polar coordinates. We can then express the coordinates a and b in terms of these as using simple geometry, i.e.,

$$a = r\cos\psi \tag{1.27}$$

and

$$b = r\sin\psi \tag{1.28}$$

Since $-\sin\psi = \sin(-\psi)$ while $\cos\psi = \cos(-\psi)$, we see that performing the complex conjugation on a complex number changes the sign on the argument ψ, i.e., the complex number simply is reflected across the real axis. The magnitude, r, of a complex number $z = a + jb$ is given by

$$r = |z| = \sqrt{a^2 + b^2}. \tag{1.29}$$

The argument, ψ, of a complex number (which is also written as $\angle z$) is given by

$$\psi = \angle z = \tan^{-1}\left(\frac{b}{a}\right), \tag{1.30}$$

for $a \neq 0$. Note that it is necessary to keep track of in which quadrant z is (i.e., by looking at the signs of the real and imaginary parts) to get the correct angle when computing it using $\tan(\cdot)$. Combined, we can write the complex number compactly as $z = r\angle\psi$. An example of a complex number, represented both in rectangular and polar form, is shown in Fig. 1.9.

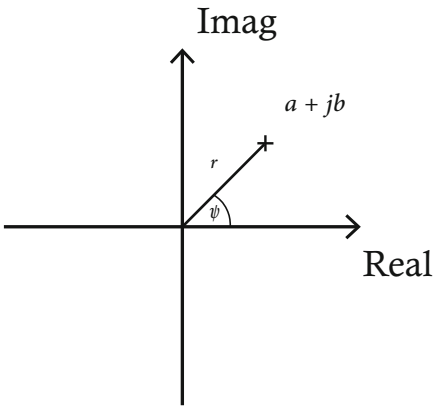


Fig. 1.9 An example of a complex number $a + jb$ and how it can be represented in different ways

Depending on what we are doing, it might be easier to use one representation over another. For example, it was quite simple to add two complex numbers in cartesian coordinates, while it is a bit more complicated to do it with the polar. It was, though, a bit complicated to multiply two complex numbers. As we will see now, the polar representation is very useful for multiplication. Let us revisit the multiplication of the two complex numbers in (1.20), only we will now use their polar representation, i.e.,

$$z_1 z_2 = (a_1 + jb_1)\,(a_2 + jb_2) \tag{1.31}$$

$$= (r_1 \cos\psi_1 + jr_1 \sin\psi_1)\,(r_2 \cos\psi_2 + jr_2 \sin\psi_2)\,. \tag{1.32}$$

Writing this out, this yields

$$z_1 z_2 = r_1 r_2\,(\cos\psi_1 + j\sin\psi_1)\,(\cos\psi_2 + j\sin\psi_2) \tag{1.33}$$

$$= r_1 r_2(\cos\psi_1 \cos\psi_2 + j\cos\psi_1 \sin\psi_2 \tag{1.34}$$

$$+ j\cos\psi_2 \sin\psi_1 + j^2 \sin\psi_1 \sin\psi_2) \tag{1.35}$$

$$= r_1 r_2((\cos\psi_1 \cos\psi_2 - \sin\psi_1 \sin\psi_2) \tag{1.36}$$

$$+ j(\cos\psi_1 \sin\psi_2 + j\cos\psi_2 \sin\psi_1)). \tag{1.37}$$

Using the trigonometric addition formulas, this can be simplified into

$$z_1 z_2 = r_1 r_2\,(\cos(\psi_1 + \psi_2) + j\sin(\psi_1 + \psi_2))\,. \tag{1.38}$$

It can now easily be seen that the result of multiplying the two numbers $z_1 = r_1 \angle \psi_1$ and $z_2 = r_2 \angle \psi_2$ is a new complex number given by $r_1 r_2 \angle \psi_1 + \psi_2$. Hence, multiplying two complex numbers results in a magnitude, that is, the product of the individual magnitudes and an argument which is the sum of the two arguments. This is both quite simple to remember and useful.

1.5 The Phasor

Suppose we define a complex function using (1.27) and (1.28) with $r = 1$, which is then a function of the angle ψ, as

$$f(\psi) = \cos\psi + j\sin\psi. \tag{1.39}$$

This function describes the unit circle in the complex plane. If we differentiate it with respect to the angle, we get

$$\frac{df(\psi)}{d\psi} = -\sin\psi + j\cos\psi. \tag{1.40}$$

Curiously, the result is equal to the function itself multiplied by j, i.e., $jf(\psi) = -\sin\psi + j\cos\psi$, and we therefore have that

$$\frac{df(\psi)}{d\psi} = jf(\psi). \tag{1.41}$$

This is a differential equation, and it begs the question if there is a broader class of functions $f(\cdot)$ that would satisfy it, other than (1.39). What function or functions can we think of whose derivative is the function itself times a constant? The exponential function is probably the easiest non-trivial function of all to find the derivative of, as it has exactly this property. The exponential function of the form e^{kx} has the derivative ke^{kx}, so this would mean that with $k = j$, we would get a function that satisfies (1.41). In other words, we have that

$$f(\psi) = e^{j\psi}. \tag{1.42}$$

It then also follows that this new function must also be equal to (1.39)! We thus have

$$e^{j\psi} = \cos\psi + j\sin\psi. \tag{1.43}$$

This brilliant and very fundamental result is called Euler's formula, and it connects two seemingly completely unrelated types of functions, namely the trigonometric functions and the exponential function. And this new complex exponential function is more general than either cosine or sine, as it encompasses both. Moreover, it can be used to define a number of mathematical constants and concepts, including π, e, and j.

Euler's formula means that we can express complex numbers in a very compact and convenient way using the exponential function. The complex number $z = a + jb$ can now also be written as

$$z = re^{j\psi}. \tag{1.44}$$

We call this the *exponential* form. The complex conjugation still works the same way. We change the sign on the imaginary part, which now appears in the exponent, i.e., $z^* = re^{-j\psi}$. Let us now revisit the multiplication of the two complex numbers z_1 and z_2. We will now use (1.44):

$$z_1 z_2 = \left(r_1 e^{j\psi_1}\right)\left(r_2 e^{j\psi_2}\right) \tag{1.45}$$

$$= r_1 r_2 e^{j\psi_1} e^{j\psi_2} \tag{1.46}$$

$$= r_1 r_2 e^{j(\psi_1+\psi_2)}, \tag{1.47}$$

where we have used that $a^b a^c = a^{b+c}$ to get the last equation. We see that this is consistent with the result we obtained earlier, namely that the result should be equivalent to $r_1 r_2 \angle \psi_1 + \psi_2$, only this is a much easier way to arrive at the result. Furthermore, we can also see that the magnitude of a complex number can be conveniently expressed and computed as $z = \sqrt{z^* z}$.

How about division then? Consider that we may write the division of z_1 by z_2 as

$$\frac{z_1}{z_2} = \frac{z_1}{1} \cdot \frac{1}{z_2}, \tag{1.48}$$

and that

$$\frac{1}{z_2} = z_2^{-1}, \tag{1.49}$$

from which we conclude that the result is:

$$\frac{z_1}{z_2} = \frac{r_1 e^{j\psi_1}}{r_2 e^{j\psi_2}} \tag{1.50}$$

$$= r_1 e^{j\psi_1} \cdot \frac{1}{r_2} e^{-j\psi_2} \tag{1.51}$$

$$= \frac{r_1}{r_2} e^{j\psi_1 - j\psi_2} \tag{1.52}$$

In other words, it means that the resulting magnitude is r_1 divided by r_2 and the angle is ψ_1 minus ψ_2, so this is also quite simple.

We might wonder what the complex equivalent (or generalization) of the sinusoid in (1.7), which was a solution to the differential equation (1.6), would look like, if it exists. Notice how the imaginary part of (1.43) is a sine function. Hence, if we define

$$z^t = e^{j\Omega t} \tag{1.53}$$

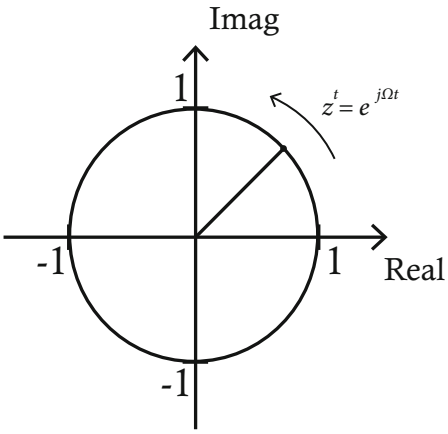


Fig. 1.10 Illustration of a phasor $z^t = e^{j\Omega t}$. It moves around the unit circle counterclockwise as time passes at a speed determined by the frequency Ω

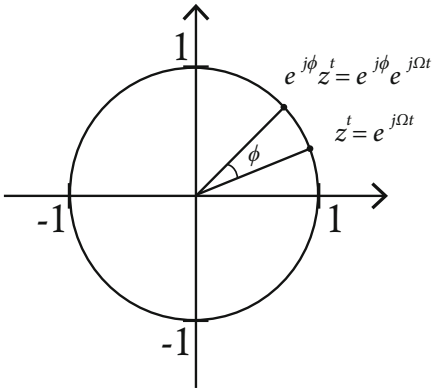


Fig. 1.11 The effect of the multiplication by a complex number, $Ae^{j\Phi}$, and the phasor, $z^t = e^{j\Omega t}$, on the argument of the phasor

then Imag $\left\{z^t\right\} = \sin(\Omega t)$. Recall that $\left(a^b\right)^c = a^{bc}$. The function in (1.53) is called a *phasor* and is instrumental in the approach to audio processing taken in this book, which is similar to that of [2]. It is depicted in Fig. 1.10.

Observe that the cosine function of (1.13) can be obtained from the real part of a complex exponential function, given by

$$Ae^{j\Phi} e^{j\Omega t} = \left(Ae^{j\Phi}\right) z^t. \tag{1.54}$$

We can now see that we can actually separate the phase and amplitude of a sinusoid from the time-varying part involving the frequency, as we were trying to do in (1.15). In Fig. 1.11 the effect of the multiplication by the complex number on the argument is illustrated. Similarly, the effect on the magnitude is depicted in Fig. 1.12. Indeed, it's now clear that we can change the magnitude and the argument of a complex function like this by multiplying it by a complex number.

In Figs. 1.13 and 1.14 two phasors of the form (1.53) are shown for two different frequencies, Ω, namely $2\pi 0.1$ and $2\pi 0.05$. They are both shown for $t = 1, 2, \ldots, 7$.

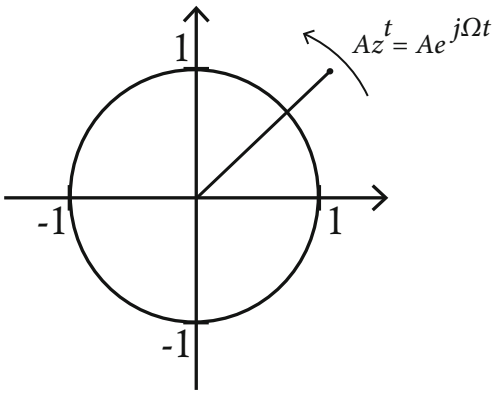


Fig. 1.12 The effect of the multiplication of a complex number, $Ae^{j\phi}$, and the phasor, $z^t = e^{j\Omega t}$, on the magnitude of the phasor

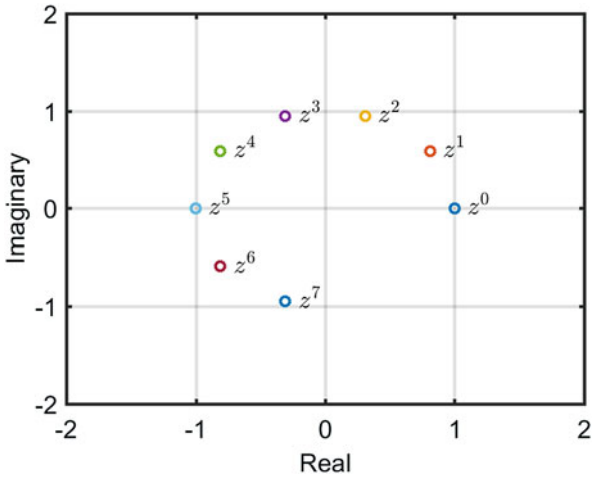


Fig. 1.13 A phasor z^t for $t = 0, 1, \ldots$ where $\Omega = 2\pi 0.1$

It can be seen from the figure that the points of the two phasors lie on the unit circle in the complex plane. Moreover, we can see that the angle changes faster for the phasor having the higher frequency, as expected.

Since both (1.7) and (1.13) were possible solutions to (1.6), we might suspect that (1.53) too is a valid solution. Indeed, differentiating (1.53) twice with respect to the time t yields

$$\frac{d^2 z^t}{dt^2} = -\Omega^2 e^{j\Omega t}, \tag{1.55}$$

which clearly satisfies (1.6). Functions of the form (1.53) are so useful in audio processing that they have their own name. They are called *phasors*. We will use such phasors in the rest of this book for understanding signals, filters, audio effects, and much more. It is useful to remember that we can also get a cosine or a sine function from its real and imaginary parts, i.e.,

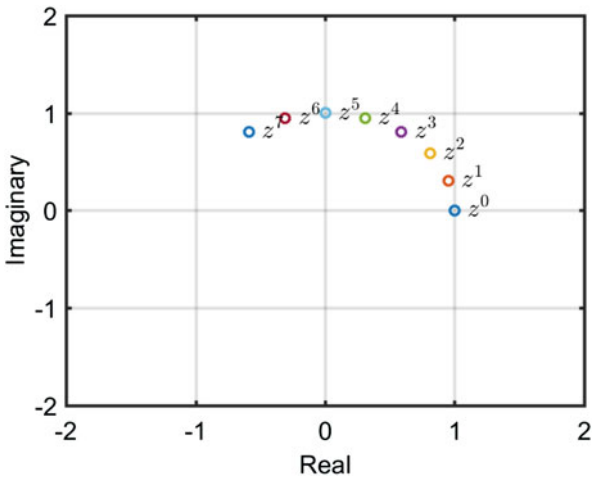


Fig. 1.14 A phasor z^t for $t = 0, 1, \ldots$ where $\Omega = 2\pi 0.05$

Table 1.1 Overview of operations on and quantities in relation to complex numbers for $z = a + jb$, $z_1 = a_1 + jb_1$, and $z_2 = a_2 + jb_2$

Operation/quantity	Notation	Rectangular	Exponential
Complex number	z	$a + jb$	$re^{j\phi}$
Magnitude ($\lvert\cdot\rvert$)	$\lvert z\rvert$	$\sqrt{a^2 + b^2}$	r
Angle ($\angle\cdot$)	$\angle z$	$\tan^{-1}\left(\frac{b}{a}\right)$	ϕ
Conjugation (*)	z^*	$a - jb$	$re^{-\psi}$
Addition ($+$)	$z_1 + z_2$	$(a_1 + a_2) + j\,(b_1 + b_2)$	
Subtraction ($-$)	$z_1 - z_2$	$(a_1 - a_2) + j\,(b_1 - b_2)$	
Multiplication ($\cdot$)	$z_1 \cdot z_2$		$r_1 r_2 e^{j\psi_1 + j\psi_2}$
Division ($/$)	z_1/z_2		$\frac{r_1}{r_2} e^{j\psi_1 - j\psi_2}$

Shown are the results for either/both the exponential or rectangular representations

$$\text{Real}\left\{z^t\right\} = \cos(\Omega t) \tag{1.56}$$

$$\text{Imag}\left\{z^t\right\} = \sin(\Omega t)\,. \tag{1.57}$$

Indeed, were we to draw the real and imaginary parts of the phasors in Fig. 1.13 and Fig. 1.14, we would see sine and cosine functions of frequencies $\Omega = 2\pi 0.1$ and $\Omega = 2\pi 0.05$, respectively.

For future reference, an overview of different operations on complex numbers and the quantities associated with complex numbers are shown in Table 1.1 in their polar and rectangular representations. In closing, we would like to remark that complex numbers also have quite a fundamental impact on polynomials too. Since $\sqrt{-1} = j$, it means that a polynomial like $f(x) = x^2 + 1$, which in primary or secondary school, would have been claimed to have no solutions x (called roots) for which $f(x) = 0$, but now we see that $x = j$ is actually a root of the polynomial,

since $j^2 + 1 = 0$. This actually means that an Mth order polynomial always has M roots, and that we can factor any polynomial using its roots.

1.6 Exercises

The exercises of this chapter focus on how to use complex numbers and their different representations. These exercises are thus intended to be solved with pen and paper (and a calculator for computing trigonometric and exponential functions).

Exercise 1 Plot the following complex numbers: (a) $2+3j$, (b) $-1-4j$, (c) $3-1j$, (d) $2e^{j\frac{\pi}{2}}$, (e) $\frac{1}{2}e^{-j\pi}$.

Exercise 2 Compute and plot the following additions: (a) $(2 + 3j) + (-1 - 4j)$, (b) $(3 - 1j) + 2e^{j\frac{\pi}{2}}$.

Exercise 3 Compute the magnitude and angle of the following complex numbers: (a) $2 + 3j$, (b) $-1 - 4j$, (c) $3 - 1j$, (d) $2e^{j\frac{\pi}{2}}$, (e) $\frac{1}{2}e^{-j\pi}$.

Exercise 4 Compute the following multiplications: (a) $(2 + 3j) \cdot (-1 - 4j)$, (b) $2e^{j\frac{\pi}{2}} \cdot \frac{1}{2}e^{-j\pi}$, (c) $(-1 - 4j) \cdot \frac{1}{2}e^{-j\pi}$.

Exercise 5 Plot the phasor $e^{j0.25\pi t}$ for $t = 0, 1, 2, \ldots$.

Exercise 6 Plot the conjugation $\cdot^*$ of the following complex numbers: (a) $2 + 3j$, (b) $-1 - 4j$, (c) $3 - 1j$, (d) $2e^{j\frac{\pi}{2}}$, (e) $\frac{1}{2}e^{-j\pi}$.

Exercise 7 What are the roots of the polynomials $f(x) = x^2 - 4$ and $f(x) = x^2 + 4$?

Exercise 8 Given $z = a + jb$ draw jz, j^2z, j^3z, ...

Chapter 2
The Wave Equation

2.1 Introduction

In the previous chapter, the concept of sound was introduced and a simple example of a physical system that can produce sound was given. More specifically, we saw how a metal bar struck by a hammer exhibits a displacement over time that is sinusoidal in nature. This was done based on some simple physics that ended up with a first-order differential equation whose solutions come in the form of the aforementioned sinusoids. This caused us to explore sinusoids in more detail and introduce complex numbers as an efficient way of studying and manipulating them. It also led us to the introduction of the so-called phasor, the complex exponential sinusoid, and the idea that signals can be thought of as comprised of such phasors was introduced.

In this chapter, we will go further into detail about the nature of sound by studying vibrating strings, and we will derive the famous wave equation that, when boundary conditions are also taken into considerations, explains why we observe sinusoids everywhere in nature. The wave equation not only holds for acoustic phenomena such as the ones we are interested in, but also explains the behavior of light, water, and electromagnetic waves, and it is one of the most fundamental results of physics. In the process of deriving the wave equation, we will study traveling and standing waves on a string. While the metal bar example from the previous chapter could be used for understanding music instruments like xylophone, marimba, and glockenspiel, standing waves can be seen as the foundation for the inner workings of instruments such as the guitar, pianos, and the violin, although real instruments are of course more complicated than our simple, physical models. In the end, we will see that the kinds of waves we can observe on a string that is attached at both ends are sinusoids whose frequencies are integral multiples of a so-called fundamental frequency, which is given by the length of the string. Sums of such sinusoids give rise to waves and signals that are periodic, cause the perceived phenomenon of pitch, and form what might be considered the atoms of music signals.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_2

2.2 Wave Equation for a String

Let us study the classical example of vibrating strings in a bit more detail. What follows is based on [1]. Vibrating strings are of course a well-known source of sound, as many musical instruments are based on it, like the violin, the guitar, and the piano. Suppose we have a string with uniform mass that is attached at both ends and is stretched to a tension T. At rest, the string is parallel to the y-axis, but we pull it in the direction of the y-axis so that it deforms (i.e., stretches) and has the aforementioned tension. Like for the metal bar, there is a restoring force that pulls the stretched, deformed string towards its equilibrium. This is illustrated in Fig. 2.1. We observe the behavior of the string along a very small piece, Δx, along the x-axis. More specifically, we observe it at a point that we call x and another point $x + \Delta x$, which is a distance Δx further along the x-axis. This is shown in Fig. 2.2. Due to the curvature of the string, there is a difference in the vertical components (the part that points in the direction of the y-axis) of the tension T. At x, there is an angle $\theta(x)$ between the tangent of the string and a horizontal line. At $x + \Delta x$, we will call this angle $\theta(x + \Delta x)$. This is depicted in Fig. 2.3. This difference in the vertical force is then

$$\Delta F = T \sin(\theta(x + \Delta x)) - T \sin(\theta(x)) . \tag{2.1}$$

It can be quite complicated to analyze nonlinear functions, like in our case where we have $\sin(\cdot)$. Moreover, the angle is a function of x, which complicates matters a bit. A mathematical trick that can be used in these circumstances is the Taylor series, which for a function $f(\cdot)$ is given by

$$f(x + \Delta x) = f(x) + \frac{\partial f(x)}{\partial x}\Delta x + \frac{1}{2}\frac{\partial^2 f(x)}{\partial x^2}\Delta x^2 + \cdots . \tag{2.2}$$

Taylor series are a brilliant tool for analyzing nonlinear functions. As we can see, it expresses the function in terms of its derivatives. It is, in general, an infinite series,

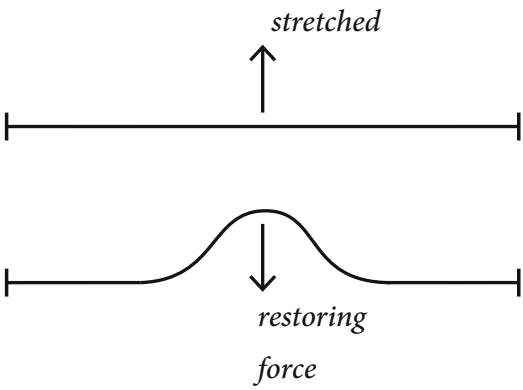


Fig. 2.1 A string that is attached at both ends is stretched by applying a force (e.g., by plucking). This creates a restoring force that pulls the deformed string towards its equilibrium

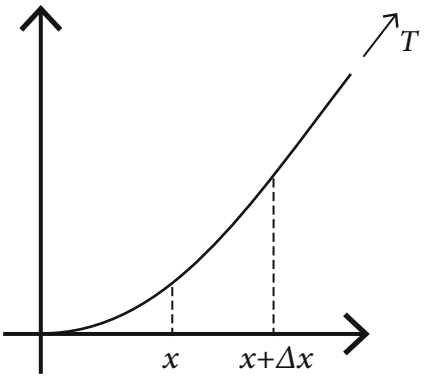


Fig. 2.2 A very small piece of the string is observed at a x and another point $x + \Delta x$. Due to the curvature, there is a difference in the vertical component of the tension, T

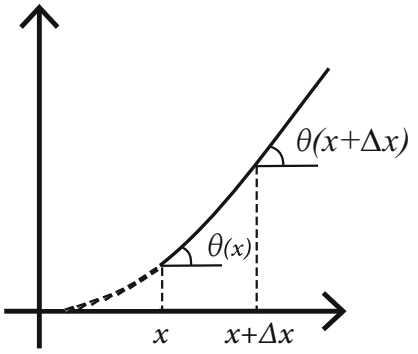


Fig. 2.3 Angles measured at x and at $x + \Delta x$

the number of terms depending on how many derivatives the function has, but if we truncate the series to a finite number of terms, we get an approximation of the function $f(x + \Delta x)$, and the smaller the Δx, the fewer terms are required to obtain a desired accuracy. We will now use it to simplify (2.1). First, note that we can approximate the relationship between the angle $\theta(x + \Delta x)$ and x for small Δx using the Taylor series as

$$\theta(x + \Delta x) \approx \theta(x) + \frac{\partial \theta(x)}{\partial x} \Delta x \tag{2.3}$$

$$= \theta(x) + \Delta\theta(x), \tag{2.4}$$

where we have introduced the definition $\Delta\theta(x) = \frac{\partial \theta(x)}{\partial x} \Delta x$. This is then called a first-order approximation, as we have only used up to order-one derivatives. If we now drop the dependency on x in the notation, we can write the angle at x as θ and the angle at $x + \Delta x$ as $\theta + \Delta\theta$, where

$$\Delta\theta = \frac{\partial \theta}{\partial x} \Delta x. \tag{2.5}$$

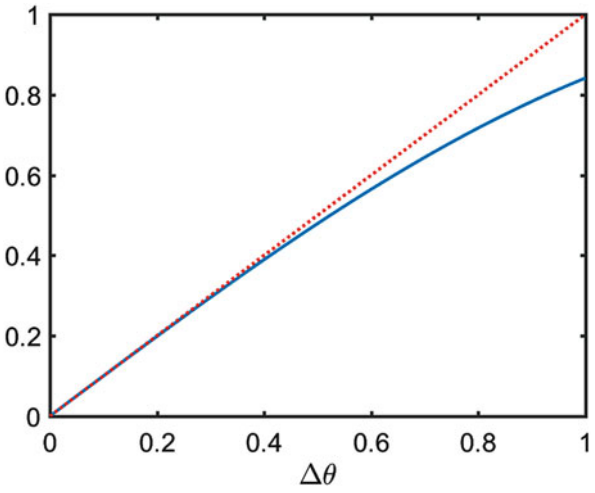


Fig. 2.4 Example of Taylor approximation, here for $\sin(\theta + \Delta\theta)$ for $\theta = 0$. Shown are the original function (solid) and its first-order approximation (dotted)

We will now use the Taylor series again, this time to approximate the sine function, which appears in (2.1), i.e.,

$$\sin(\theta + \Delta\theta) \approx \sin(\theta) + \frac{\partial \sin(\theta)}{\partial \theta}\Delta\theta. \tag{2.6}$$

In Fig. 2.4, the Taylor approximation in (2.6) is shown for $\theta = 0$, i.e., the figures show the left- and right-hand sides of the equation. As can be seen, the Taylor approximation is accurate for small values of $\Delta\theta$ and the difference between the two curves grows bigger as a function of $\Delta\theta$.

By inserting (2.5) into (2.6), we get

$$\sin(\theta) + \frac{\partial \sin(\theta)}{\partial \theta}\Delta\theta = \sin(\theta) + \frac{\partial \sin(\theta)}{\partial \theta}\frac{\partial \theta}{\partial x}\Delta x \tag{2.7}$$

$$= \sin(\theta) + \frac{\partial \sin(\theta)}{\partial x}\Delta x, \tag{2.8}$$

where we have used the chain rule, i.e., that $\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(u)}{\partial u} \cdot \frac{\partial g(x)}{\partial x}$ with $u = g(x)$ to get from (2.7) to (2.8). Finally, if we now use this result to approximate the first term in (2.1), we can simplify it as

$$\Delta F = T\sin(\theta + \Delta\theta) - T\sin(\theta) \tag{2.9}$$

$$= T\left(\sin(\theta) + \frac{\partial \sin(\theta)}{\partial x}\Delta x\right) - T\sin(\theta) \tag{2.10}$$

$$= T\frac{\partial \sin(\theta)}{\partial x}\Delta x. \tag{2.11}$$

Next, we note that for a very small displacement y (i.e., a small value of the function), $\sin(\theta)$ can be approximated by $\tan(\theta)$, which in turn is related to the slope of the tangent of a function, $\frac{\Delta y}{\Delta x}$, and thus also the derivative, i.e.,

$$\tan(\theta) \approx \frac{\partial y}{\partial x}. \tag{2.12}$$

By replacing $\sin(\theta)$ in (2.11) by $\frac{\partial y}{\partial x}$, we obtain

$$\Delta F = T \frac{\partial \left(\partial y/\partial x\right)}{\partial x} \Delta x \tag{2.13}$$

$$= T \frac{\partial^2 y}{\partial x^2} \Delta x. \tag{2.14}$$

At this point, we note that according to Newton's second law, we know that $\Delta F = ma$, where m is the mass and a is the acceleration. With a density of ρ, the mass of the piece of string, which was Δx long, is $m = \rho \Delta x$. Meanwhile, the acceleration along the y-axis is given by

$$a = \frac{\partial^2 y}{\partial t^2}. \tag{2.15}$$

Therefore, we have that

$$\Delta F = \rho \Delta x \frac{\partial^2 y}{\partial t^2}. \tag{2.16}$$

We can now equate the right-hand sides of (2.11) and (2.16), i.e.,

$$\rho \Delta x \frac{\partial^2 y}{\partial t^2} = T \frac{\partial^2 y}{\partial x^2} \Delta x, \tag{2.17}$$

from which we can get the final result:

$$\frac{\partial^2 y}{\partial t^2} = c^2 \frac{\partial^2 y}{\partial x^2}, \tag{2.18}$$

where we have introduced the definition $c = \sqrt{T/\rho}$ which we from the units can see is the speed. The important relation in (2.18) is the famous wave equation! It is, as we can see, a second-order differential equation. While we have here arrived at it for vibrations on a string, the result applies for many more scenarios, including acoustics ones such as columns of air and vibrating bodies. In fact, it is one of the most celebrated results of physics, as the principles apply to not only acoustics but also electromagnetic waves and in fluid dynamics. As for the meaning of the wave equation, we can see that it relates the acceleration of the displacement (i.e., the

second-order derivative with respect to time) to the second-order derivative of the displacement as a function of where we observe the string. This means that there is a special relationship between what we observe as we move along the string and when we observe the displacement at a single point of the string over time.

2.3 Traveling and Standing Waves

We will now explore exactly what kind of displacement functions satisfies the wave equation. We will do this based on a simple, intuitive thought experiment based on a rope. Suppose we tie the one end of a rope to a wall, at position $x = L$, while we hold the other in one of our hands at position $x = 0$, as shown in Fig. 2.5. Actually, we do not need the rope to be tied at all at this point, but for visualization purposes, it would be confusing if the rope was floating in the air. Later we will discuss what happens due to the other end being tied to a wall as well. If we move the hand in question quickly to create a bump, we know from experience that the bump will travel down the rope. Let us say that we describe that bump by a function $f(t)$ and that the bump occurs at $t = 0$. An example of what such a function might look like is depicted in Fig. 2.6. If the wave then travels down the rope with a speed of c, then at position x we will observe a displacement of the rope at time t by

$$y(x, t) = f(t - x/c). \tag{2.19}$$

The bump will look the same at position x at time $t = x/c$ as $f(t)$ did at time zero. That is, the further we move from the origin, the later the bump will arrive.

If we differentiate the equation in (2.19) twice with respect to x, we obtain

$$\frac{\partial^2}{\partial x^2} f(t - x/c) = \frac{1}{c^2} f''(t - x/c), \tag{2.20}$$

while differentiation twice with respect to t yields

$$\frac{\partial^2}{\partial t^2} f(t - x/c) = f''(t - x/c). \tag{2.21}$$

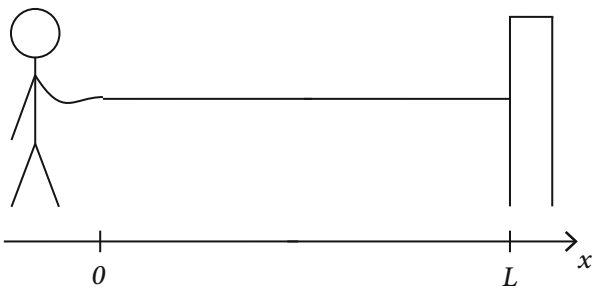


Fig. 2.5 A roped is tied at one end, for $x = L$, and held by a person in the other for $x = 0$

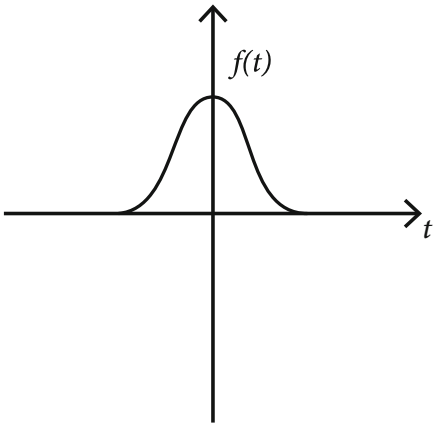


Fig. 2.6 An example of a bump, described by the function $f(t)$

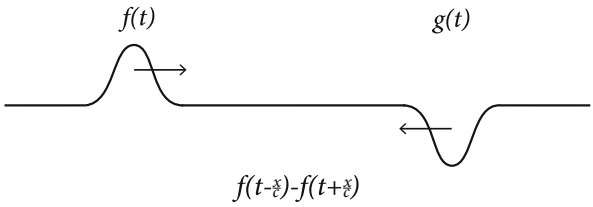


Fig. 2.7 Two superimposed bumps traveling in opposite directions down the rope

If we compare the last two equations with the wave equation in (2.18), we can see that the bump $f(\cdot)$ actually satisfies it, as long as its behavior, in terms of the displacement $y(\cdot)$, over time, t, and as a function of where we look at it, x, is according to (2.19), i.e.,

$$\frac{\partial^2}{\partial t^2} f(t - x/c) = c^2 \frac{\partial^2}{\partial x^2} f(t - x/c) \tag{2.22}$$

If the other end, which we before assumed was tied to a wall, is also held by a person, who also creates a bump, but a different one called $g(\cdot)$, then what would happen? We would now have the first bump $f(\cdot)$ traveling from the left and to the right while the second bump would travel in the opposite direction, both with the same speed c. This is illustrated in Fig. 2.7. The displacement at a point x and at time t in between the two ends would be

$$y(x, t) = f(t - x/c) + g(t + x/c). \tag{2.23}$$

These two bumps are thus superimposed. If the rope has a length of L, the bump at the other end is thus the function $g(t + L/c)$. The displacement function in (2.23) can also be shown to satisfy the wave equation. So, it seems that, more or less, any two functions $f(\cdot)$ and $g(\cdot)$ will do. Indeed, they can be arbitrary functions, although their derivatives must of course exist. They could be impulses, sinusoids, or something else. To say something more specific about these functions, we must

consider additional constraints, which are called initial or boundary conditions in the context of differential equations. Let us return to the example where one end of the rope was tied to a wall somehow. What does this mean for the solutions to the wave equation? Let us say that we are holding the rope in our hand at position $x = L$ while the other end is at position $x = 0$. The rope would then be unable to move at the position $x = 0$ while it could move at position $x = L$. This means that, in terms of the displacement function, we must have that

$$y(0, t) = 0. \tag{2.24}$$

This is an example of a boundary condition. A consequence of it is that we must have that

$$f(t) + g(t) = 0, \tag{2.25}$$

which in turn implies that

$$f(t) = -g(t), \tag{2.26}$$

which must hold for all t. Hence, by imposing the constraint (2.24), $f(\cdot)$ and $g(\cdot)$ can no longer be different functions. Indeed, the displacement function must now be

$$y(x, t) = f(t - x/c) - f(t + x/c). \tag{2.27}$$

This is interesting. It means that at any point x, the displacement of the string will be a superposition of two identical bumps, except that they have opposite signs and travel in opposite directions. It essentially means that by one end being tied to the wall, the bump traveling from the hand towards the wall will be reflected by the wall and return with the opposite sign. This point is illustrated in Fig. 2.8, which shows the bump $f(t)$ traveling down the rope and returning in the opposite direction with the opposite sign. Were we instead to generate a sinusoid with our hand, the returning wave would also be a sinusoid, but with a negative sign.

What if both ends were tied? Then would have that

$$y(0, t) = 0 \quad \text{and} \quad y(L, t) = 0. \tag{2.28}$$

As we saw earlier, the first constraint is satisfied when $f(t) = -g(t)$, but the second one implies that

$$f(t - L/c) - f(t + L/c) = 0 \tag{2.29}$$

$$f(t - L/c) = f(t + L/c) \tag{2.30}$$

for all t. In other words, we must have that

$$f(t) = f(t + 2L/c). \tag{2.31}$$

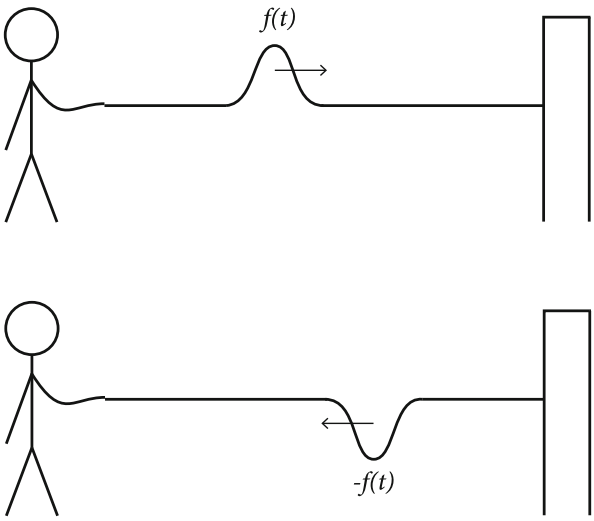


Fig. 2.8 The bump, $f(t)$, traveling down the rope towards the wall where the rope is tied and returning with the opposite sign

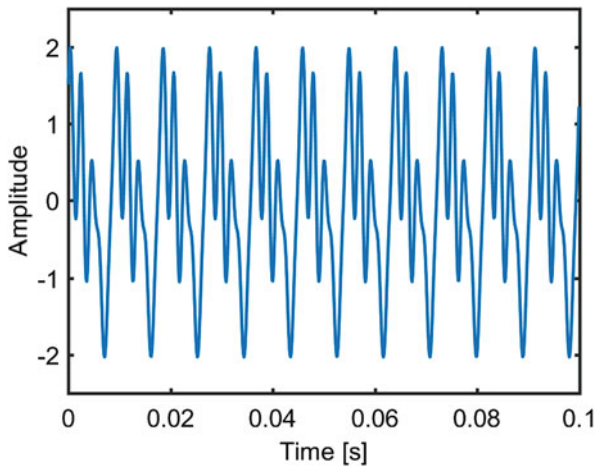


Fig. 2.9 Example of a periodic function

What kind of function is this? As it turns out, it is a kind of function that is very fundamental to sound, and one that is very fundamental to music as well. Functions that obey the relation $f(t) = f(t + D)$ for all (or an interval of) t are said to be periodic with period D, which means that the function is identical to itself at a time D apart. In other words, the function is repeating. In Fig. 2.9, an example of a periodic function can be seen.

In the case of (2.31), the function is repeating every $2L/c$, which means that the periodicity of the function is determined by the length of the rope. A function that repeats every D seconds has a so-called fundamental frequency, which is defined as

$$f_0 = \frac{1}{D} \quad [\text{Hz}], \tag{2.32}$$

which is the number of repetitions per second. Measured in radians per second, it is given by

$$\Omega_0 = 2\pi f_0. \tag{2.33}$$

Sine and cosine functions are of course periodic, so they would qualify, but we have also seen that they are contained in the complex exponential function, the phasor. The phasor $z^t = e^{j\Omega_0 t}$ with $\Omega_0 = \pi c/L$ is periodic with period $D = 2L/c$. Again, we remind the reader that the propagation speed of the wave in the medium, c, is related to the frequency, here f_0, and the wavelength, λ, as $c = f_0\lambda$. From this, we gather that the wavelength thus must be $2L$, i.e., twice the length of the rope. The aforementioned phasor is not the only one that has this period. In fact, any phasor we could construct with frequency $\Omega_0 k = \pi c/Lk$ where $k = 1, 2, \ldots$ would also be periodic with period $2L/c$. Moreover, their wavelengths are $2L/k$. Hence, the more general phasor

$$z^t = e^{j\Omega_0 k t}, \tag{2.34}$$

would be good candidate for further study. In Fig. 2.10, the imaginary value of (2.34), i.e., a sinusoid, is shown with $\Omega_0 = 2\pi$ for $k = 1, 2, \ldots, 5$. It can be seen that all these functions are periodic with a period of 1 s.

Following [2], let us now select a solution of the form

$$f(t) = \alpha e^{j\Omega_0 k t}, \tag{2.35}$$

where α is a complex constant, and examine the displacement function (2.27) further. Inserting (2.35) into (2.27) we get

$$y(x,t) = f(t - x/c) - f(t + x/c) \tag{2.36}$$

$$= \alpha e^{j\Omega_0 k(t-x/c)} - \alpha e^{j\Omega_0 k(t+x/c)} \tag{2.37}$$

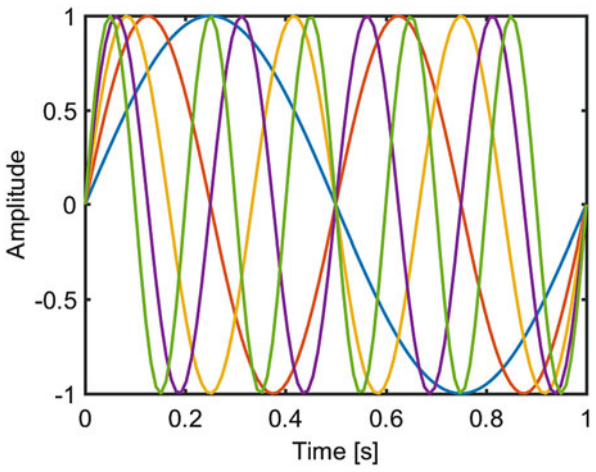


Fig. 2.10 Different sinusoids that are all periodic with the period $D = 1$ s

$$= \alpha e^{j\Omega_0 kt} \left(e^{-j\Omega_0 kx/c)} - e^{j\Omega_0 kx/c} \right) \tag{2.38}$$

$$= \alpha e^{j\Omega_0 kt} 2j \sin\left(\Omega_0 kx/c\right). \tag{2.39}$$

Inserting $\Omega_0 = \pi c/L$ and introducing the definition $\beta = \alpha 2j$, we get

$$y(x,t) = \beta e^{j\frac{\pi ck}{L}t} \sin\left(\frac{\pi k}{L}x\right). \tag{2.40}$$

This equation tells us how the displacement at a point x behaves over time, and how the displacement behaves for a particular time t as we observe the function at different places x. Let us now differentiate the displacement function to check that it satisfies the wave equation. First, we differentiate twice with respect to time:

$$\frac{\partial^2}{\partial t^2} y(x,t) = -\beta \left(\frac{\pi ck}{L}\right)^2 e^{j\frac{\pi ck}{L}t} \sin\left(\frac{\pi k}{L}x\right) \tag{2.41}$$

Next, we differentiate (2.40) twice with respect to place, i.e.,

$$\frac{\partial^2}{\partial x^2} y(x,t) = -\beta \left(\frac{\pi k}{L}\right)^2 e^{j\frac{\pi ck}{L}t} \sin\left(\frac{\pi k}{L}x\right). \tag{2.42}$$

From this, we can see that

$$c^2 \frac{\partial^2}{\partial x^2} y(x,t) = \frac{\partial^2}{\partial t^2} y(x,t), \tag{2.43}$$

and the wave equation thus holds for the displacement function (2.40) defined from the phasor (2.35), and it holds for all $k = 1, 2, 3, \ldots$. Note that the different phasors can have different complex constants, α, although we omitted this dependency in our notation. Not only do the individual phasors satisfy the wave equation, but since it holds for the individual phasors for different k, it also holds for sums of such phasors, as differentiation is a linear operation. Indeed, we can actually express (almost) any periodic function with period D as

$$f(t) = \sum_{k=-\infty}^{\infty} \alpha_k e^{j\frac{2\pi}{D}kt}. \tag{2.44}$$

This is known as a Fourier series, something that plays an instrumental role in signal processing. The individual terms are referred to as *harmonics*. We will return to the subject of Fourier series later on in the book. Note how any of the individual phasors we can construct from (2.35) for different k are special cases of this more general function. The sum results in the following displacement function [2]:

$$y(x,t) = \sum_{k=-\infty}^{\infty} \beta_k e^{j\frac{\pi ck}{L}t} \sin\left(\frac{\pi k}{L}x\right), \tag{2.45}$$

so we observe a whole number (actually, it is here an infinite number) of sinusoids. Remember that if we would rather understand this in terms of a real displacement, we can simply take the real value of the complex one. The displacement function in (2.45) tells us that on a string that is tied at both ends and is excited using some initial displacement, we can observe waves of a certain form, namely periodic ones. More specifically, we can observe sinusoids over both time and along the length of the string. These are called *standing waves*. Moreover, when observed over time, the frequencies of the sinusoids describing the displacement are integral numbers of a fundamental frequency $\frac{\pi c}{L}$ while it is $\frac{\pi}{L}$ when observed over the length of the string. We say that $\frac{\pi c}{L}k$ for $k = 1, 2, \ldots$ are the modes of the system.

We refer the interested reader to [1] for more about the wave equation and the mathematics and physics involved with all things acoustic.

2.4 Exercises

The following exercises focus on understanding the physics behind sound and are intended to be solved with pen and paper. It should be possible to solve all these exercises based on the information in the first two chapters of the book.

Exercise 1 When the tension of a string is increased, does the speed of the sound along the string increase or decrease?

Exercise 2 A tuning fork is similar to a bar in many ways. What happens to the frequency of a tuning-fork when you increase its mass (see the previous chapter for help)?

Exercise 3 It was claimed that any periodic function can be represented by a Fourier series. How many sinusoids do you need to form a square wave?

Exercise 4 What is the frequency in Hz and rad/s, respectively, of a periodic waveform having a period of 2.3 ms?

Exercise 5 On an electric guitar, the ends of the string are attached to the nut and the bridge. This means we are dealing with the case of standing waves. The strings are approximately 64.77 cm (called the scale) on a Fender type guitar and the frequency of the high e string (the note is E_4) is 329.60 Hz. What then is the speed? Suppose the density of the string is 0.401×10^{-3} kg/m, what then is the string tension?

Chapter 3
Digital Audio Signals

3.1 Introduction

As we have discussed in the earlier chapters, the phenomenon of sound can mean either an auditory sensation in the air or the disturbance in a medium that causes such a sensation. An interesting question is exactly how we go from such phenomena, to numbers inside a computer, stored in an array of integers, for example, that we can manipulate in whatever way we desire, and if (and if so how) we can reconstruct an acoustic signal from those numbers.

As a physical phenomenon, sound can be understood as vibration, as movement of molecules that cause changes in the pressure of some medium, like the air around us. These vibrations in the air can then be measured with a microphone, in which the changing pressure causes a diaphragm to move (assuming that we are talking about a dynamic microphone). This movement is then translated into a voltage signal via a voice coil and a magnet. So, we now have an idea about how a signal goes from being movement of molecules in a gas to an electrical signal. The voltage signal produced by a microphone is a continuous signal, i.e., it can take on a value at any time, as is the pressure at a point in space. Moreover, even at a particular time instance, it can take on an infinite number of different values, even if the range is limited.

A computer can only store a finite number of numbers, and those numbers can only take on a finite number of values. So, we must go from measuring infinitely many time-values that can take on an infinite number of different values to that. Put differently, we must go from analog-to-digital signals. In relation to this, there is also the question of how we can reconstruct analog signals. The processes involved with doing all this are called *sampling*, *quantization*, and *reconstruction*, respectively. These are essentially the things that soundcards do. In hardware terms, an analog-to-digital converter (ADC) is a device, or chip, that converts signals from analog to digital by sampling and quantization while a digital-to-analog converter (DAC) converts digital signals to analog ones by reconstruction. In Fig. 3.1, an example of an audio processing system is shown.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_3

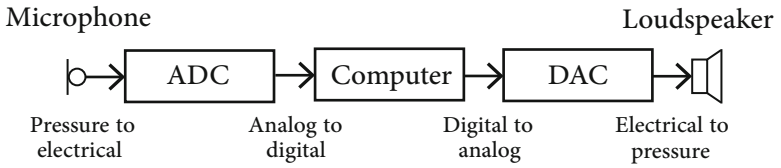


Fig. 3.1 Example of an audio processing system. Vibrations in the air are converted from a pressure signal to an electrical one by the microphone. The analog signal from the microphone is converted into a digital one by the ADC, after which the digital signal can be processed by a computer. The processed digital signal can then be converted into an analog one by the DAC and, finally, converted back into a pressure signal by an active loudspeaker

As for why we would prefer to process digital signals rather than analog ones, there are many technical reasons, the most important ones being that computers are flexible, can do things that we could only dream of doing with analog hardware, and are typically much cheaper than the corresponding analog solution. On a side note it is interesting to observe that while most of the rest of the world has long realized the advantages of digital electronics, many music equipment aficionados are still resisting digital equipment.

In this chapter, we will explain what sampling and quantization are, how they work, and also how we can reconstruct analog signals from digital ones.

3.2 Sampling

Suppose we have a continuous-time signal, $x(t)$, defined for any real t that we would like to process in our computer. The process of sampling then involves measuring the value of this function at particular time instances, t_n, indexed by $n = 0, 1, 2, \ldots$, to obtain the signal

$$x_n = x(t_n), \tag{3.1}$$

which we call a discrete-time signal. The simplest and most common way of sampling is *uniform sampling*, which means sampling at equidistant points, i.e.,

$$t_n = T_\mathrm{s} n, \quad \text{for} \quad n = 0, 1, 2, \ldots \tag{3.2}$$

where T_s is the time (in seconds) between consecutive samples and is called the sampling period. We can see that a sampled, or digital, signal is really just an ordered sequence of numbers. The smaller the T_s, the more finely we sample the original continuous-time signal, $x(t)$. We can represent the number of samples obtained per second using the *sampling frequency*, which is defined as

$$f_\mathrm{s} = \frac{1}{T_\mathrm{s}}. \tag{3.3}$$

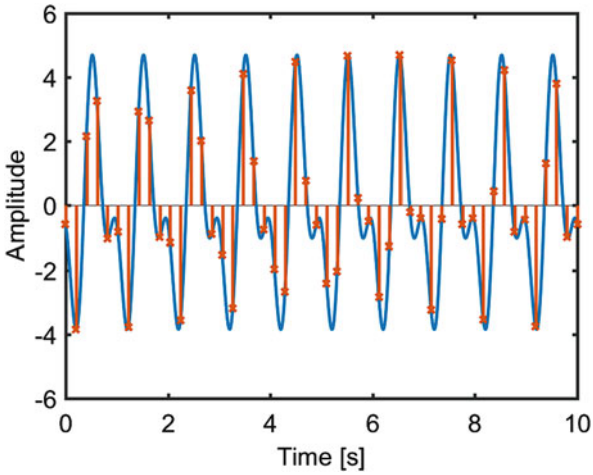


Fig. 3.2 Example of a continuous signal, $x(t)$, and its sampled counterpart, x_n. The sampled values are marked by "x"

The sampling frequency is measured in Hertz. Audio is most often sampled at 44.1 kHz and above while speech can be sampled down to 8 kHz, which results in understandable but certainly not high-quality signals. In Fig. 3.2, the process of sampling is exemplified by a continuous signal, $x(t)$, which is sampled to obtain the discrete-time signal, x_n.

To explore sampling a bit more, let us consider an example. Suppose $x(t)$ is a single sinusoid of frequency f in Hertz, i.e.,

$$x(t) = \sin(2\pi f t)\,. \tag{3.4}$$

If we sample this at regular intervals, i.e., with uniform sampling, with a sampling period of T_s, then the sampled signal, which is now a function of the sample index n, is given by

$$x_n = \sin(2\pi f n T_\mathrm{s})\,. \tag{3.5}$$

Using (3.3), we can also express this as

$$x_n = \sin\left(2\pi \frac{f}{f_\mathrm{s}} n\right). \tag{3.6}$$

From this, we can define the useful quantity

$$\omega = 2\pi \frac{f}{f_\mathrm{s}}, \tag{3.7}$$

which we call the *digital frequency*. It expresses the number of radians per sample. From such a digital frequency alone, we thus cannot tell what the frequency would be in Hertz, unless we know the sampling frequency. A digital frequency of $\omega = 2\pi$ corresponds to the sampling frequency, regardless of what value it is. This makes

sense intuitively in that a digital signal is just a sequence of numbers, so without further knowledge, we would have no way of inferring anything about the sampling frequency.

Now, the fundamental question that we should ask ourselves is this: how high a sampling frequency do we need to select to sample the sinusoid in (3.4)? Technically speaking, what we are concerned about when sampling is whether the spectrum, i.e., the frequency contents, of the sampled signal is the same as that of the original, continuous signal. If it is, it means we have a perfect, digital representation of the analog signal. It can be shown that if we sample the signal with a sampling frequency that is more than twice as large as the frequency of the sinusoid, the sampled signal will have the same spectrum as the original. A couple of questions now spring to mind. Firstly, how do we generalize this result to any signal, not just a sinusoids? The answer to this question is that, as we have already mentioned, (almost) any signal can be thought of as comprised of a number of sinusoids, and we can use the principle from before, that the sampling frequency must be more than twice as high as the frequencies of the sinusoids. So, if the highest frequency of a sinusoid in a signal is $f_{\max}$, then the sampling frequency should obey

$$f_s > 2f_{\max}. \tag{3.8}$$

This is known as the *sampling theorem*. Sometimes, depending on in which field, it is attributed and named after different people, including Nyquist, Shannon, Whittaker, and Kotelnikov. The second question is: how do we know what the maximum frequency of the sinusoids in our signal is? The answer to this question is that we do not know this in general. Instead, we can simply make sure that for a selected sampling frequency, f_s, for our audio system, we will remove all contents above $f_s/2$! This is done with a so-called anti-aliasing filter, which is just an analog low-pass filter with stopband frequency of $f_s/2$, which means that all frequencies above the stopband frequency are removed or attenuated. This is illustrated in Fig. 3.3.

We say that the signal is then *bandlimited*. Aliasing refers to the phenomenon that occurs when we do not obey the sampling theorem. When we sample at a sampling frequency that is less than twice the maximum frequency what happens is that the parts of the spectrum above half the sampling frequency fold back around half the sampling frequency and appears in the lower spectrum as mirror images of sorts. Since these mirror images mix with the frequency contents there and cannot be recovered or removed, we have distorted our signal. For example, returning to the example of the signal being a single sinusoid, a sinusoid of 4500 Hz sampled at 8000 Hz would alias and the sampled signal would be identical to a sinusoid of 3500 Hz! We see that the frequency $f_s/2$ is quite important, and it is important enough that it has been given a name. It is often called the *Nyquist frequency*. In Fig. 3.4, an example of a continuous signal and the resulting digital signal are shown. The continuous signal is a sinusoid with a frequency of 100 Hz and it is sampled at a frequency of 1000 Hz. In that case, the sampling theorem is satisfied, and we can see that the samples represent the signal well. In Fig. 3.5, another example is given.

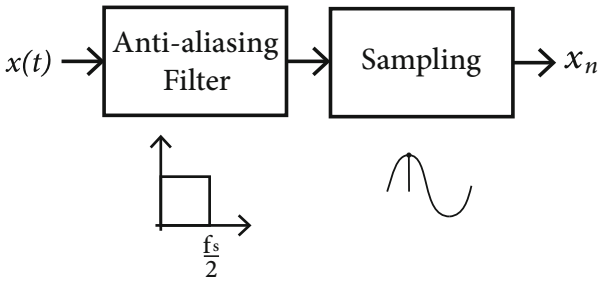


Fig. 3.3 In analog-to-digital converters, the analog signal is first filtered by an anti-aliasing filter to prevent aliasing before it is sampled

Fig. 3.4 Example of the sampling of a signal when the sampling theorem holds. The original continuous-time signal is sampled at a frequency of 1000 Hz

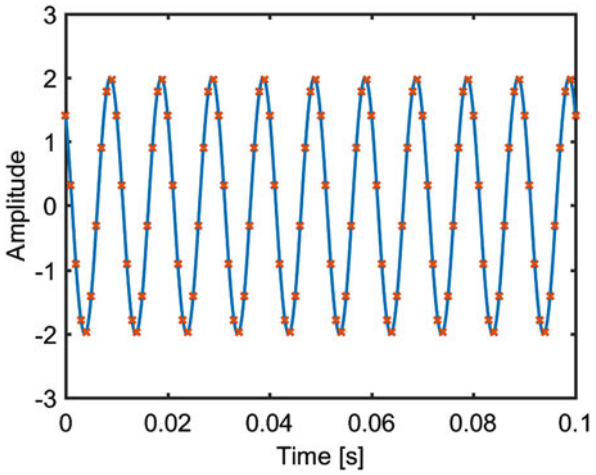


Fig. 3.5 Example of sampling where aliasing occurs. The original signal (solid), which is a 100 Hz sinusoid, is sampled at 150 Hz, which results in samples that are equivalent to those of a 50 Hz sinusoid

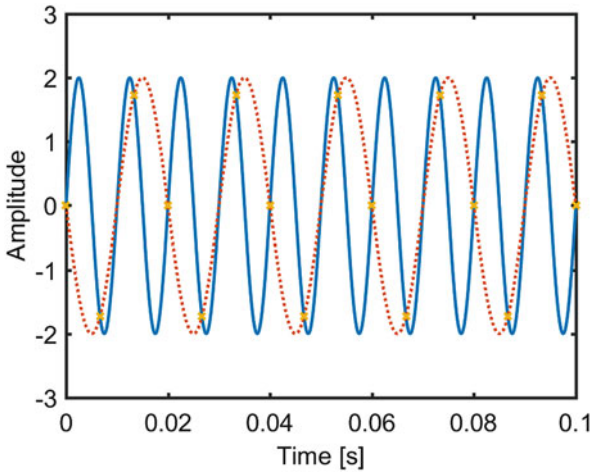


Table 3.1 Common audio sampling frequencies and some examples of their applications

Sampling frequency	Applications
8 kHz	Narrowband speech, telephony
16 kHz	Wideband speech, telephony, VoIP
44.1 kHz	CD, audio equipment, sound cards
48 kHz	DAT, video recorders
96 kHz	DVD-audio, HD DVD, Blu-ray
192 kHz	DVD-audio, Blu-ray

Again, the signal is a sinusoid with a frequency of 100 Hz but now the sampling frequency is only 150. The samples are indicated by "x." A second sinusoid representing the aliasing that now occurs is also shown. It has a frequency of 50 Hz. As can be seen, it also passes through the same samples, and it thus not possible to distinguish between the 100 Hz sinusoid or a 50 Hz sinusoid sampled at 150 Hz!

In Table 3.1, some of the most common audio sampling frequencies and their applications are listed. Modern sound cards usually support a wide range of sampling frequencies and are thus quite flexible. It should be noted that, for technical reasons, it is often beneficial to use a higher sampling frequency in the original recording and during processing than what will be used in the end-product, like a CD. We mention in passing that the process of going from one sampling rate to another, a common task in many audio systems, is called resampling.

3.3 Quantization

As we have discussed, an analog signal measured at a given time instance, say nT_s, can, in principle at least, take on any value. However, we only have a finite number of bits and thus a finite number of different values at our disposal to represent a measurement in a computer. Thus, we must map this infinite number of values to a finite number of values. This is called *quantization*. To perform quantization, which is done by a *quantizer*, we first assume that our measured signal, x_n, which typically represents a voltage, is in a certain range, for example, from $-\alpha$ to α. If we have β bits at our disposal, we can represent 2^β different values. They could, for example, represent the integers from 0 to $2^\beta - 1$ or from $-2^{\beta-1}$ to $2^{\beta-1} - 1$. If we divide the interval between $-\alpha$ and α into 2^β equally large cells, the cells must have the size

$$\Delta = \frac{\alpha}{2^{\beta-1}}. \tag{3.9}$$

This is called *uniform quantization* and is the most common form of quantization for audio. Δ is called the step size. Obviously, the higher the number of bits, β, the smaller the Δ. The number of bits used per sample in audio is typically 16, 24, or 32 bits. It should be mentioned that when the input signal is outside the specified range from $-\alpha$ to α, clipping typically occurs, a problem that is familiar to most people

Fig. 3.6 The processes involved with analog-to-digital conversion, namely anti-aliasing filtering, sampling, and quantization. Each affects the signal in different ways

familiar with music recording and production. This means that any signal values x_n over α are truncated to α and similarly for negative numbers below $-\alpha$. This causes quite severe and audible artifacts and should be avoided, whenever possible.

In Fig. 3.6, the process of analog-to-digital conversion is illustrated in a block diagram. The process involves filtering of the analog signal by an anti-aliasing filter, followed by the sampling of the signal. Finally, the samples are each quantized. The anti-aliasing filter is a low-pass filter, which removes, or rather attenuates, frequencies above half the sampling frequencies. Often, this also causes the frequencies that are close to half the sampling frequency to be affected somewhat. The sampling process then causes aliasing of frequency contents above the half the sampling frequency (if any), and quantization adds noise, with the level of the noise being dependent on the number of bits used.

We know now what the step size of the quantizer should be given the range and the number of bits, but we have not decided on what values we should map our input to. The mapping of x_n to the cells of size Δ is performed by a *quantizer*. We denote that quantizer by $\mathcal{Q}(\cdot)$ and its output as y_n, i.e.,

$$\mathcal{Q}(x_n) = y_n. \tag{3.10}$$

We could implement our quantizer via truncation or rounding of $-\alpha \leq x_n < \alpha$ as

$$y_n = \Delta \left\lfloor \frac{x_n}{\Delta} + 0.5 \right\rfloor. \tag{3.11}$$

Here, $\lfloor \cdot \rfloor$ refers to the floor operation, which maps a real number to its largest previous integer. In Fig. 3.7 the output of the uniform quantizer (3.11) is shown as a function of a continuous input from -1 to 1 for a step size of $\Delta = 0.125$, corresponding to 4 bits.

The quantization error, e_n, is the difference between the input and the output, i.e.,

$$e_n = x_n - y_n = x_n - \mathcal{Q}(x_n), \tag{3.12}$$

and we are interested in understanding the effect of the quantization on the quality of the signal. A good way of understanding the effect of quantization is the so-called additive noise model of quantization. It states that we can think of the effect of quantization as adding noise to the input signal, i.e.,

$$y_n = x_n + e_n. \tag{3.13}$$

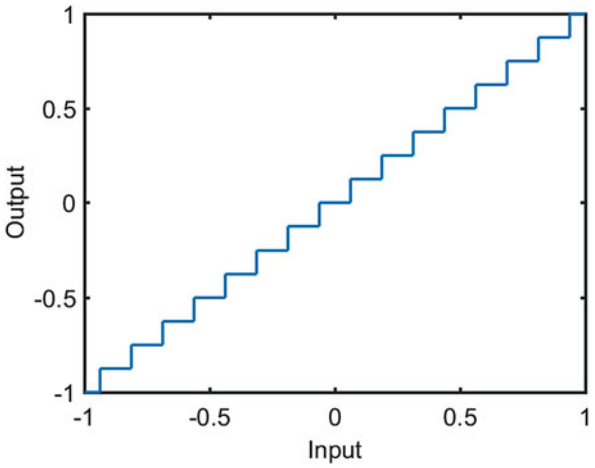


Fig. 3.7 Effect of quantization with the quantized output as a function of a continuous input from -1 to 1 with $\Delta = 0.125$

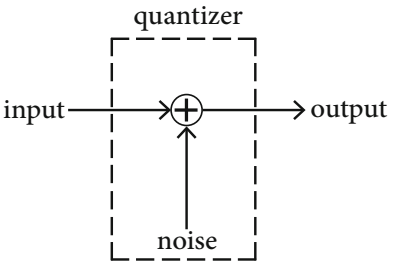


Fig. 3.8 Additive noise model of quantization. The effect of quantization can be thought of as adding noise to the signal. The power of the noise is controlled by the number of bits in the quantizer

This is illustrated in Fig. 3.8. The amount of noise is then determined by the step size, Δ. The smaller the Δ, the less noise is added to the signal. In fact, we can see that the quantization errors are between $-\frac{1}{2}\Delta$ and $\frac{1}{2}\Delta$, due to the rounding in (3.11). In Fig. 3.9, a histogram of quantization errors is shown for 10,000 input values from -1 to 1 with a step size of $\Delta = 0.125$. As can be seen, the different errors occur with approximately the same frequency and the errors lie, as expected, i.e., between $-\frac{1}{2}\Delta$ and $\frac{1}{2}\Delta$.

To analyze the effect of quantization a bit further, we must introduce a new concept, namely the *power* of a signal. We can think of our signal as a digital representation of a voltage signal that was sampled. Similarly, we can think of our signal x_n as a representation of the voltage signal, $v(t)$, that appears on the output of a digital-to-analog converter. If we then connect the output of the DAC to a resistor having resistance $R = 1\ \Omega$, the instantaneous power that is dissipated in the resistor is then simply $v^2(t)$ (see Fig. 3.10). The average power over a time interval of T is given by

$$P_v = \frac{1}{T}\int_{t=0}^{T} v^2(t)dt. \tag{3.14}$$

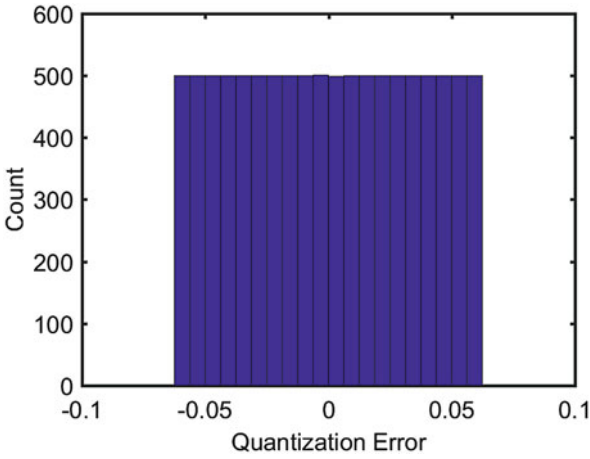


Fig. 3.9 Histogram of quantization errors for 10,000 input values from -1 to 1 with $\Delta = 0.125$

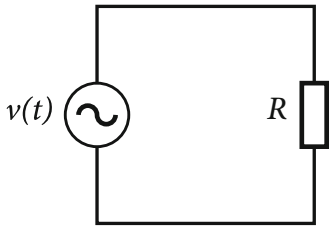


Fig. 3.10 Electrical circuit analogy of signal power

For a digital signal, the equivalent of such an integral is the sum

$$P_x = \frac{1}{N} \sum_{n=0}^{N-1} x_n^2, \tag{3.15}$$

which is here computed over N samples. The power of a signal is directly related to how loud it is—the higher the power, the louder something generally is. We now have an understanding of what the power of a signal is. Returning now to the additive noise model of quantization, we denote the power of the input signal by P_x and the power of the noise signal by P_e. A measure of the quality of the quantization is the ratio between the power of the input and noise signals, i.e.,

$$\frac{P_x}{P_e}. \tag{3.16}$$

This is called the *signal-to-noise ratio (SNR)*. The higher this number is, the better our quantization is and the less audible the quantization error will be. Powers and power ratios are often represented on a logarithmic scale in decibel, and the SNR is also typically computed in dB, i.e.,

$$SNR = 10 \log_{10} \frac{P_x}{P_e} \tag{3.17}$$

$$= 10 \log_{10} P_x - 10 \log_{10} P_e. \tag{3.18}$$

The power of the output signal of the quantizer can (under certain technical conditions [5] which are beyond the scope of this book) be shown to be

$$P_y = P_x + P_e. \tag{3.19}$$

Thus, what we hear on the output of the quantizer is the power of the input plus the power of the output. If our input, x_n, takes on all possible values between $-\alpha$ and α equally often, the power can be shown to be

$$P_x = \frac{1}{3}\alpha^2. \tag{3.20}$$

Furthermore, with this assumed input, the quantization errors will be between $-\frac{1}{2}\Delta$ and $\frac{1}{2}\Delta$ and all values will occur equally often. The result is that the power of the quantization noise signal is given by [6]

$$P_e = \frac{\Delta^2}{12}. \tag{3.21}$$

Inserting (3.20) and (3.21) into the definition of the SNR (3.17), we get

$$SNR = 10 \log_{10} \frac{P_x}{P_e} \tag{3.22}$$

$$= 10 \log_{10} \left(\frac{\alpha^2}{3} \cdot \frac{12}{\Delta^2} \right). \tag{3.23}$$

Inserting the relation between Δ and the number of bits, β, in (3.9), we obtain

$$SNR = 10 \log_{10} \left(\frac{\alpha^2}{3} \cdot \frac{12}{\Delta^2} \right) \tag{3.24}$$

$$= 20 \log_{10} \left(\frac{2\alpha}{\left(\frac{\alpha}{2^{\beta-1}} \right)} \right) \tag{3.25}$$

$$= \beta 20 \log_{10} 2 \tag{3.26}$$

$$\approx \beta \cdot 6 \text{ [dB]}. \tag{3.27}$$

This derivation shows that the SNR is approximately six times the number of bits. Hence, we get an SNR of 96 dB if we use 16 bits. Note that this is a theoretical value that shows what performance can be achieved with uniform quantization. Practical

analog-to-digital converters will, however, generally perform worse than this, due to noisy electrical components and other factors. Moreover, the full range from $-\alpha$ to α is typically not used when recording signals (even though we should strive to do so), and this also leads to a loss in SNR in practice. The derivation here also shows that, limitations of electrical components aside, we can basically get any SNR we desire. To determine how many bits are required to quantize audio transparently, we could carry out a listening experiment, where we try our different number of samples and have human subjects listen to the result, and pick the lowest number of bits where the subjects cannot tell the difference between the input and the output of the quantizer.

It should be mentioned that the noise that is added during uniform quantization is *white*, which means it has a flat spectrum and contains all possible frequencies—just like white light contains all colors. There exist more complicated quantization schemes where the noise is shaped in accordance with human sound perception via the so-called noise shaping. Furthermore, there also exist techniques for making the quantization more noise-like, particularly for low signals, called dithering. These add pseudo-random noise signals to avoid annoying artifacts in the quantized signal. These can occur for certain signals because the quantization in reality is a deterministic function of the input.

Another concept that is frequently associated with audio systems is that of *dynamic range*. The dynamic range of something is defined as the ratio (often measured in dB) between the softest and loudest value. For the human auditory system, the dynamic range is an astounding 120 dB! For a sampled and quantized signal with β bits with values going from 1 to 2^β (the actual choice of range α is not important in this context, since we are interested in ratios), the ratio between the smallest and biggest possible values is thus $2^\beta/1$, which in dB is

$$20\log_{10} 2^\beta = \beta 20\log_{10} 2 \approx \beta 6 \text{ dB}. \tag{3.28}$$

Hence, the dynamic range of a system using 16 bit per sample is 96 dB. The values can be interpreted as the ratio between the smallest and largest pressure that we can measure with our digital system. Later on, we will see how the effective dynamic range can be altered by what is known as *dynamic range control*, which are algorithms for modifying the amplitude of signals depending on how loud they are.

3.4 Reconstruction

Now that we have looked into sampling and quantization the remaining issue is how we go back from the digital signal to an analog one, i.e., how we reconstruct a continuous-time signal from the sampled and quantized signal once we are done with the processing. We have seen how we can think of the effect of quantization as adding noise to our signal, so we actually do not need to differentiate between the

original and quantized signals. Thus, we consider the digital signal, x_n, which was sampled from the original signal, $x(t)$, as

$$x_n = x(T_s n), \tag{3.29}$$

with a sampling period of T_s, from which we wish to reconstruct $x(t)$ for all t. The condition for appropriate sampling of the continuous-time signal was that we sampled at a sampling frequency that was more than twice as high as the highest frequency in the signal. We can think about what this actually means about the signal. One way to think about it is that the signal must be slowly varying between samples compared to the sampling frequencies. The way we enforced this in general was via the anti-aliasing filter, which is just a low-pass filter. Since the effect of a low-pass filter can also be thought of as a smoothing of sorts, we see that even if the signal was varying quickly in-between samples, the anti-aliasing filter would have removed those fast fluctuations. We should, therefore, also be able to reconstruct the signal $x(t)$ from x_n by letting the signal vary slowly from one value to the next. Indeed, this is also the gist of the idea behind analog reconstruction. This is typically achieved in two steps. First, what is known as staircase reconstruction is used to fill in the blanks between consecutive samples. It simply holds the value of a sample x_n until the time of the next one x_{n+1}. This results in an abrupt change, in fact a discontinuity, in the reconstructed signal, $y(t)$ at each $T_s n$. These discontinuities result in the so-called partially attenuated spectral images at higher frequencies. The discontinuities are then simply removed with a low-pass filter that results in a smooth transition from one sample to the next. This is sometimes referred to as an *anti-image postfilter*. In Fig. 3.11, the process of reconstruction is illustrated while in Fig. 3.12 signal examples are shown. The original samples can be seen in the figure, along with the staircase reconstruction of the continuous signal from the samples. The final reconstruction is then obtained by low-pass filtering of the staircase reconstruction.

At this point it should be stressed that although our description of sampling, quantization, and reconstruction here is a bit superficial (on purpose) and the claims hand-wavy, the underlying principles are technically solid. It is a mathematical fact (i.e., it can be proven) that a bandlimited signal can be sampled and later reconstructed perfectly. Thus, we can (in theory at least) reconstruct a signal of any bandwidth, as long as it is finite, it is just a matter of using a high-enough sampling frequency. A caveat here is that it can also be shown that a signal cannot be truly bandlimited while also being limited in time. As we have seen, we can also basically

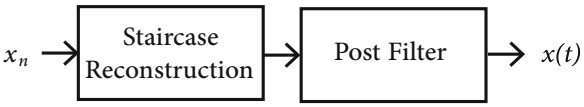


Fig. 3.11 The process of digital-to-analog conversion, which involves staircase reconstruction of a continuous signal from the discrete signal, followed by a low-pass filter to remove the discontinuities

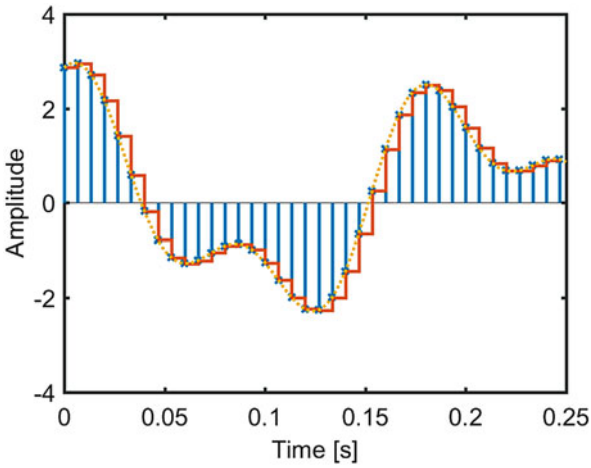


Fig. 3.12 Reconstruction of continuous (analog) signal from discrete (digital) signal. Shown are the samples ("x"), staircase reconstruction (solid), and low-pass filtered (dotted)

quantize to achieve any desired fidelity—it is just a matter of how many bits we are willing to use, i.e., how expensive equipment we are willing to buy (or build). In the end, the quality of ADCs and DACs is limited in practice by a number of factors, including the quality of the analog components involved in the process (for example, in the anti-aliasing and anti-image filters), just like analog solutions would be.

3.5 Exercises

The following exercises are pen-and-paper type exercises focusing on developing an understanding of sampling and quantization and related concepts.

Exercise 1 At what sampling frequency must a signal with a maximum frequency of 12 kHz be sampled?

Exercise 2 A sinusoid with frequency 4500 Hz is sampled at 8000 Hz. What is the frequency of the aliased sinusoid?

Exercise 3 At what frequency must ECG be sampled?

Exercise 4 At what frequency must the following signal be sampled: $x(t) = 2\cos(2\pi 1001t) + 10\sin(2\pi 550t) + 5\cos(2\pi 1501t)$?

Exercise 5 What is the theoretical SNR of a system using 16, 24, and 48 bits, respectively?

Exercise 6 What is the theoretical dynamic range of the systems using 16, 24, and 48 bits, respectively?

Exercise 7 Find examples of aliasing from your everyday life.

Exercise 8 If the range of the input to our quantizer is from -10 to 10, and we use a 24 bit uniform quantizer, what is the step size of the quantizer?

Chapter 4
What Are Filters?

4.1 Introduction

The most basic form of audio processing is filtering and it is possible to do many interesting things to audio signals with filters. In this chapter, we will explore what filters are, what they do and provide some examples of relatively simple filters. What we normally call filters are technically speaking called linear, time-invariant filter. Linear refers to the filters obeying some properties that are associated with the linearity of systems, while time invariant means that the filters do not change over time, something that allows us to analyze filters more easily and understand them independently of time. There exist filters that are not time invariant, examples of which we will see later on in the form of audio effects.

Digital filters come in the form of so-called difference equations, which, in the form we will be discussing here, are sums of delayed versions of the input. Each of these delayed versions is scaled by a real number that we call a filter coefficient. It is interesting to note that digital filters are much simpler than their analog counterparts. Where the output of digital filters is determined by difference equations, the output of analog filters is determined by differential equations and integrals. The math involved in the latter is thus much more advanced than the former. Moreover, it is possible to do much more complicated things with digital filters than with analog filters. There are thus several reasons to prefer digital filters to analog ones.

4.2 A Simple Filter

As we saw in the previous chapter, a digital signal is basically just an ordered sequence of numbers. A digital filter takes as its input such a sequence of numbers, called x_n, where n is the time index, and modifies it to produce another sequence of numbers, another signal y_n, which is called the output. This is illustrated in Fig. 4.1.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_4

Fig. 4.1 The input, x_n, is passed through a filter to obtain the output, y_n

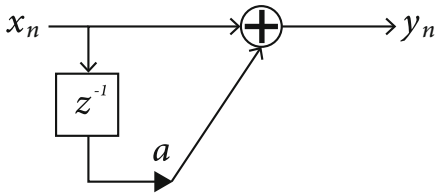


Fig. 4.2 Block diagram of a simple feedforward filter having the difference equation $y_n = x_n + ax_{n-1}$

Digital filters come in the form of so-called difference equations. Such difference equation involves multiplying and adding a number of delayed versions of the input and output. A simple example of such a difference equation is

$$y_n = x_n + ax_{n-1}, \tag{4.1}$$

where x_n is the input at time n and y_n is then the output, also at time n. The term ax_{n-1} deserves a bit more explanation. The factor, a, is what we call a filter coefficient. x_{n-1} is the input delayed by one sample. Remember that a digital signal is just a sequence of numbers. Thus, whatever sequence of numbers appears on the input x_n appears in the form of x_{n-1} one sample later. Filters where the input is delayed, possibly scaled, and added are also called *feedforward filters*. A block diagram of the filter is shown in Fig. 4.2. In the figure, we can see that the signal passes through a block containing a z^{-1}. This means a delay by one sample and is called a unit delay. We will explain this in more detail later, but to develop some understanding of it, consider the following. Remember that a phasor having frequency ω is given by

$$z^n = e^{j\omega n}. \tag{4.2}$$

If we delay the phasor by a number of samples, say D, we replace n by $n - D$ and get

$$z^{n-D} = e^{j\omega(n-D)} = e^{j\omega n}e^{-j\omega D}. \tag{4.3}$$

Here, we have used the rule $a^{b+c} = a^b a^c$. Since D is just a number, $e^{-j\omega D}$ is just a complex number, a constant, that is multiplied onto the phasor z^n. Moreover, that complex number has magnitude 1 and an angle of $-\omega D$. We write this as

$$e^{-j\omega D} = 1\angle -\omega D. \tag{4.4}$$

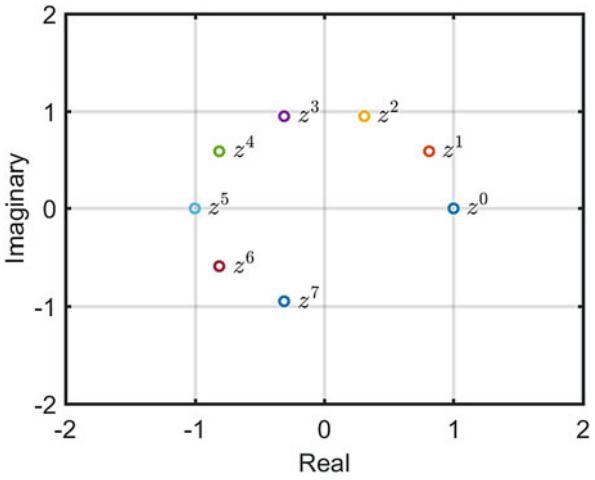


Fig. 4.3 Phasor $z^n = e^{j\omega n}$ with a frequency of $\omega = 2\pi 0.1$

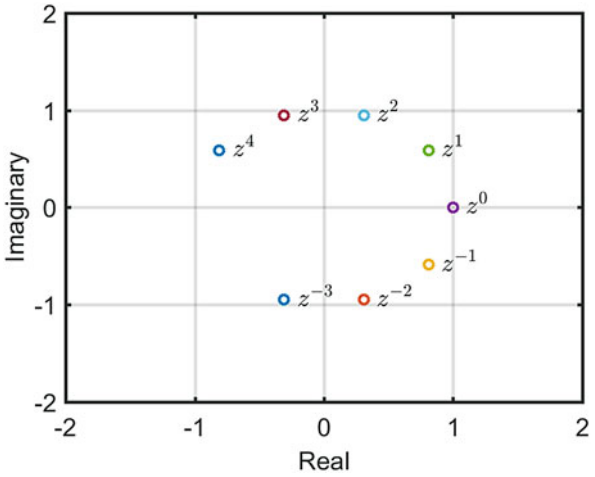


Fig. 4.4 Phasor $z^n = e^{j\omega n}$ with a frequency of $\omega = 2\pi 0.1$ delayed by $D = 3$ samples

We say that the delay changes the *phase* of the phasor z^n. In Fig. 4.3, the phasor $z^n = e^{j\omega n}$ is shown for $n = 0, 1, \ldots, 7$ for a frequency of $\omega = 2\pi 0.1$. In Fig. 4.4, the same phasor is shown, only it is now delayed by $D = 3$ samples, i.e., $z^{n-D} = e^{j\omega(n-D)}$. By comparing the two figures, we can see that they are related by a rotation, a change of angle, corresponding to the multiplication of the original phasor with $e^{-j\omega D}$.

Now, if we put the phasor on the input of the filter in (4.1), i.e., we set $x_n = z^n$, then we get

$$y_n = x_n + ax_{n-1} \tag{4.5}$$

$$= z^n + az^{n-1} \tag{4.6}$$

$$= \left(1 + az^{-1}\right) z^n. \tag{4.7}$$

This shows us that the phasor, z^n, is modified by a multiplication by $\left(1 + az^{-1}\right)$ on its way to the output when we feed it through the filter, so we now have a bit of understanding of what (4.1) does. In fact, we also just learned from this that the output of the filter will also be a phasor, but one that is changed! If we replace z^n by $z^n = e^{j\omega n}$, the result, which we call $H(\omega)$, is

$$H(\omega) = 1 + az^{-1}\Big|_{z=e^{j\omega}} \tag{4.8}$$

$$= 1 + ae^{-j\omega}. \tag{4.9}$$

The quantity $H(\omega)$ is a complex number that depends on the variable ω, i.e., the frequency of the phasor we put on the input. If we think of z^n as a probe signal we can put on the input of a system, and we can think of $H(\omega)$ as the multiplicative change that the phasor is subjected to, we can change the frequency of the phasor to observe how $H(\omega)$ depends on the frequency. This is illustrated in Fig. 4.5. $H(\omega)$ is, therefore, called the *frequency response* of the filter. Since $H(\omega)$ is complex, we can split it into a real part and an imaginary part, but it is more informative to think of it as consisting of a magnitude $|H(\omega)|$ and an angle $\angle H(\omega)$. The angle is often written as

$$\angle H(\omega) = \theta(\omega). \tag{4.10}$$

We call $|H(\omega)|$ the *magnitude response* and $\theta(\omega)$ the *phase response* of the filter. They combine to form the frequency response as

$$H(\omega) = |H(\omega)|\angle H(\omega) = |H(\omega)|e^{j\theta(\omega)} \tag{4.11}$$

Fig. 4.5 The frequency response of a filter is obtained by setting the input equal to a phasor, $z^n = e^{j\omega n}$. The output for a frequency ω is then given by $H(\omega)e^{j\omega n}$, where $H(\omega)$ is the frequency response. The frequency response describes how the magnitude and the angle of the phasor are changed by the filter as a function of the frequency, ω

and specify how the magnitude of a given frequency, here ω, has its magnitude and its phase modified. An easy way to compute the magnitude response is to observe that $|H(\omega)| = \sqrt{H(\omega)H^*(\omega)}$ and $|H(\omega)|^2 = H(\omega)H^*(\omega)$. The phase response of a filter is defined as

$$\theta(\omega) = \angle H(\omega) = \tan^{-1} \frac{\text{Imag}\{H(\omega)\}}{\text{Real}\{H(\omega)\}}, \tag{4.12}$$

for $\text{Real}\{H(\omega)\} \neq 0$.

Most of the time when we design or study filters, we are interested in the magnitude $|H(\omega)|$ because it tells us how the power of a phasor of frequency ω is changed by our filter. The phase response is often not of interest to us, but it is part of the filter's behavior and sometimes it is very important. We might, for example, desire a phase response of a certain kind, for example, linear or what is called minimum phase. In the case of the filter in (4.1), the magnitude response is

$$|H(\omega)| = |1 + ae^{j\omega}| \tag{4.13}$$

$$= \sqrt{|1 + a^2 + 2a\cos(\omega)|}. \tag{4.14}$$

In Fig. 4.6, this magnitude response is shown for $a = \pm 0.9$. For $a = 0.9$ it can be seen that phasors having high frequencies are attenuated, while low ones are amplified. However, for $a = -0.9$ it can be seen that it is the other way around: low frequencies are attenuated, while high frequencies are amplified! We can see that we can get radically different behavior from our filter by simply changing the sign of one filter coefficient. The two filters are what we call low-pass and high-pass filters, respectively.

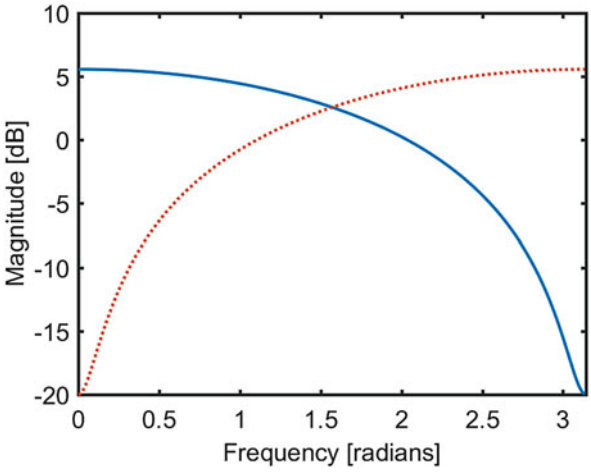


Fig. 4.6 Magnitude response of the filter in (4.1) for $a = 0.9$ (solid) and $a = -0.9$ (dotted)

It should be noted that while we can derive the frequency response of simple filters quite easily with pen and paper, this is most often not done in practice, and particularly not for more complicated filters. Instead, a piece of software, such as MATLAB, is used to compute the frequency response numerically. So, our derivations here serve only to illustrate the concepts and explore the behavior of filters.

Let us proceed to explore a bit more complicated filter. Instead of just the single, delayed term in (4.1), we can add multiple such terms, each scaled by a different value, i.e.,

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + \cdots + b_M x_{n-M}. \tag{4.15}$$

This is still a difference equation and a filter, only it now has multiple filter coefficients, i.e., b_m for $m = 0, 1, \ldots, M$, and there are multiple delays involved, the biggest one being by M samples. We say that the filter has order M. Note that while there are M unit delays involved, there are $M + 1$ filter coefficients, and that we may write the equation in compact form as

$$y_n = \sum_{m=0}^{M} b_m x_{n-m}, \tag{4.16}$$

which will prove useful later on. In Fig. 4.7, a filter of order 3 is shown. As it can be seen, it comprises three delays of the input, in the form of three cascaded unit delays. Such a collection of cascaded unit delays is called a *delayline*. We write the collection of the filter coefficients in (4.15) as $\{b_k\}$. Filter design is then concerned with finding the collection of filter coefficients, $\{b_k\}$, that give us a desired behavior, most often specified in terms of the resulting magnitude response. Filter design is

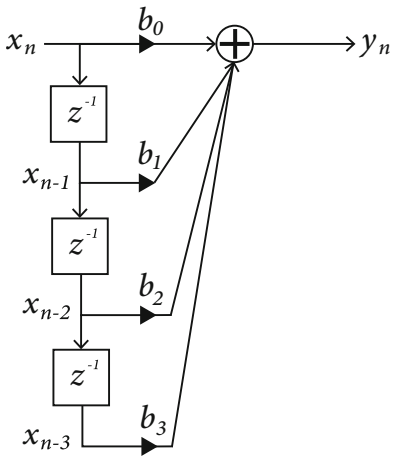


Fig. 4.7 Block diagram of a bigger filter comprising three delays, forming a delayline

a complicated matter, except for the most simple filters having a low order, M, and entire books can be (and have been) written on the subject.

4.3 Analyzing Filters

We will now explore how to analyze filters a bit more. Consider the filter having the following difference equation:

$$y_n = a_0 x_n + a_1 x_{n-1}. \tag{4.17}$$

With $a_0 = 0.5$ and $a_1 = 0.5$ this filter just computes the average (or mean) of two consecutive samples. We have seen how we can understand filters by looking at how they change a phasor of a certain frequency, which is passed through the filter. We have also seen that we can think of the effect of a delay by a sample as multiplication of a phasor by z^{-1}. We can, in fact, use this way of thinking independently of what our input is. This is because we can think of *any* signal as comprised of a number (possible an infinite one) of phasors! This means we do not even need to know what the input is to understand how it will be affected by a filter—we can just pretend that it is comprised of phasors. We can think colloquially of this as a phasor representation of sorts of our input. We write this as $x_n \rightarrow X(z)$. As we have also seen, the output of a filter will also be a phasor when the input is a phasor, albeit a changed one. Hence, when the input is a number of phasors, the output will also be a number of phasors. This follows from the linearity of filters—more on that later. We may thus write the output as $y_n \rightarrow Y(z)$. $X(z)$ and $Y(z)$ are technically speaking the z-transforms of the sequences x_n and y_n, respectively, and we say that we are computing (or deriving) the z-transform when we do this. Admittedly, it is a bit sketchy at this point, but we will go more in depth with the z-transform later on. Returning to our filter in (4.17), we can replace x_n by $X(z)$ and y_n by $Y(z)$, then we get

$$Y(z) = \left(a_0 + a_1 z^{-1}\right) X(z), \tag{4.18}$$

where we have used that $x_{n-1} \rightarrow z^{-1} X(z)$. Similarly to how we defined the frequency response, we can now define a function $H(z)$ that relates the output to the input as

$$Y(z) = H(z) X(z). \tag{4.19}$$

When analyzing the filters using our phasor representation of the input and the output, the function $H(z)$ is called the system's *transfer function*, and it is defined from $X(z)$ and $Y(z)$ as

$$H(z) = \frac{Y(z)}{X(z)}. \tag{4.20}$$

Intuitively, it can be understood in the following way. If we put the input $X(z)$ on the input and measure $Y(z)$ somehow, and we would like to say something about the system independently of the input, $X(z)$, we can divide the output, $Y(z)$, by the input, and the result is the transfer function, $H(z)$. The reason we can do it like this is this the phasor representation of the input. In the case of the filter (4.17), the transfer function is given by

$$H(z) = a_0 + a_1 z^{-1}. \tag{4.21}$$

If we substitute z by $e^{j\omega}$ we get the frequency response $H(\omega)$. To understand filters a bit further, it is helpful to think of the transfer functions as polynomials. The transfer function in (4.21) can also be written as

$$H(z) = \frac{a_0 z + a_1}{z}, \tag{4.22}$$

by multiplying with z in the numerator and in the denominator. We see that we have a first-order polynomial in the numerator, i.e., $A(z) = a_0 z + a_1$. As we know, polynomials have roots, i.e., values for which the polynomial will be zero. In the case of the numerator in (4.21), we can easily see that we have a root for $z = -\frac{a_1}{a_0}$. What does this mean for our filter? As mentioned before, we get the frequency response of a filter by substituting z by $e^{j\omega}$. By doing this, we are essentially evaluating the polynomials in the numerator and the denominator in the transfer function on the unit circle. If the values for which we evaluate the polynomial are close to the root of the polynomial in the numerator, the transfer function will tend to zero and will be exactly zero, if the root happens to be on the unit circle. This means that any input signal will be attenuated, or even cancelled, at frequencies near the roots of the polynomial. For this reason, the roots of the numerator polynomial of a transfer function are referred to as *zeros*.

From the same principles as for (4.17), we can see that the bigger filter in (4.15) has the transfer function

$$H(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \cdots + b_M z^{-M}. \tag{4.23}$$

As before, we can think of this as a polynomial, only this will be one of order M, and it will thus have M roots.

So far, we have analyzed filters by putting phasors, which are complex generalizations of sinusoids, on the input. Another useful way of analyzing filters is by putting a different signal on the input, namely, an impulse. An impulse is a signal with all its energy concentrated at a certain time instance. A digital impulse, for which we will use the symbol δ_n, can be defined as

$$\delta_n = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{4.24}$$

In mathematics, this is called the Kronecker delta function, named after the German mathematician Leopold Kronecker. As a signal, we can think of this function as the infinite sequence of numbers

$$0, \ldots, 0, 1, 0, \ldots, 0, \tag{4.25}$$

where the 1 occurs at time instance 0, or, if we start our sequence at $n = 0$, then it is simply the sequence $1, 0 \ldots$. Perhaps we can learn something useful by putting such an impulse on the input of a filter. In fact, the signal we get by putting δ_n on the input is called the *impulse response*, and it is a very important concept in the theory of filters. This principle is illustrated in Fig. 4.8. Starting out with the simple filter in (4.1), we can see, as shown in Fig. 4.9, that when we put the impulse on the input at time $n = 0$ (assuming that the input is zero for $n < 0$), the output sequence is simply $1, a, 0, \ldots$. We can see this by tracing the numbers, in order, through the block diagram. At time $n = 0$, x_{n-1} is 0, and the impulse passes right through to the output. At time $n = 1$, x_n is now 0 but x_{n-1} is 1. Hence, we get 1 times a on the output. At time $n = 2$ and from then on, both x_n and x_{n-1} are zero, and no signal passes through the filter. Similarly, we can easily see that the filter in (4.17) outputs $a_0, a_1, 0, \ldots$ when δ_n is put on the input. If we try the more complicated case of the bigger filter in (4.15), we can see that the output is $b_0, b_1, b_2, \ldots, b_M$ when $x_n = \delta_n$. Hence, we can observe the filter coefficients on the output of such a filter when we put an impulse on the input. This is, however, only the case for feedforward filters. We can also observe that the impulse response will be zero when $n > M$, i.e., when the time index exceeds the highest delay. For this reason, filters of the form (4.15) are also called *finite impulse response* (FIR) filters.

The conclusion of all this must be that we can actually learn something from a filter by looking at its output when an impulse is put on the input. The impulse response is actually also related to the frequency response and the transfer function. The phasor representation of an impulse is just 1, i.e., $\delta_n \rightarrow 1$. Recall that the input–output relation with the phasor representation is

$$Y(z) = H(z)X(z), \tag{4.26}$$

Fig. 4.8 The impulse response of a filter is obtained by putting a digital impulse, δ_n, on the input and then observing the output. This principle works for both feedback and feedforward filters

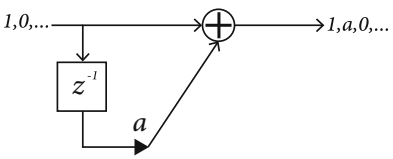


Fig. 4.9 Output of the simple filter in (4.1) when the input is a digital impulse, δ_n

then if $X(z) = 1$, we obtain

$$Y(z) = H(z), \tag{4.27}$$

which confirms that in the phasor representation, the output is indeed equal to the transfer function when the input is an impulse. Hence, our notions of filters and signals are actually closely related and consistent.

4.4 Feedback Filters

The filters we have looked at so far have all been feedforward, or FIR, filters. More powerful filters can be constructed with the use of feedback, where not only delayed versions of the input but also of the output are used. We will now explore such filters a bit. A simple example of a filter that employs feedback is

$$y_n = x_n + a y_{n-1}. \tag{4.28}$$

A block diagram of this filter is shown in Fig. 4.10. As can be seen, the output is fed back and added to the input, hence the name feedback. These types of filters also go by another name, namely, *infinite impulse response* (IIR) filters. To understand this name, consider the input of the filter being an impulse, i.e., $x_n = \delta_n$. By tracing the impulse through the filter in Fig. 4.10, we can see that the output is the sequence (see Fig. 4.11)

$$1, a, a^2, a^3, a^4, \ldots. \tag{4.29}$$

For a non-zero filter coefficient a, the impulse response will be non-zero for all $n = 0, 1, \ldots$. For $a = 0.5$, we can see that the impulse response is the sequence $1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots$, so the values get smaller and smaller but never become exactly zero. If, however, we choose $a = 2$, then the impulse response becomes $1, 2, 4, 8, 16, \ldots$! This is not good. The values of the sequence will grow and grow. This was never a concern for the feedforward filters we looked at so far. Indeed, this a special property of feedback filters. We say that the filter is *unstable*. For this simple case, we can see that as long as $|a| < 1$, the impulse response will decay to zero. However, for filters with multiple delays in the feedback part, it is not straightforward to see if the filter is stable. We will return to the question of how to deal with this later.

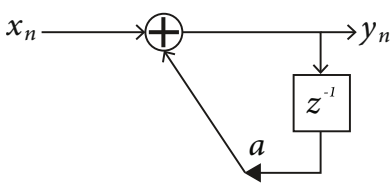


Fig. 4.10 Example of a simple filter with feedback

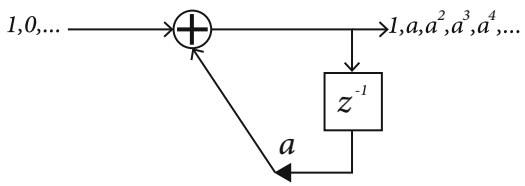


Fig. 4.11 Impulse response of a simple feedback filter

We can analyze feedback filters much like we did for feedforward filters, by using the phasor representation. Doing like we did for the feedback filters yields

$$Y(z) = X(z) + aY(z)z^{-1}. \tag{4.30}$$

This is different in nature from what we have seen so far. Rearranging the terms, we get

$$Y(z) - aY(z)z^{-1} = X(z) \tag{4.31}$$

$$\left(1 - az^{-1}\right) Y(z) = X(z). \tag{4.32}$$

We can see that the transfer function must be

$$H(z) = \frac{Y(z)}{X(z)} \tag{4.33}$$

$$= \frac{1}{1 - az^{-1}}. \tag{4.34}$$

Unlike for the feedforward case, a (non-trivial) polynomial now appears in the denominator. We saw earlier that roots in the polynomial in the numerator cause the input signal to be attenuated or even cancelled at frequencies near the roots. What do the roots in the denominator then do? If we multiply by z in the numerator and denominator of the transfer function, the polynomial in the denominator is $A(z) = z - a$. It is obviously zero for $z = a$. When z comes close to a, we will divide by a small number in the denominator, which will cause the magnitude to grow. So, if $z = e^{j\omega}$ comes the roots for some frequency, ω, we will get an amplification of that phasor! If the phasor happens to fall exactly on the root, we divide by zero and we can think of the magnitude response tending to infinity! As the reader might have guessed, this phenomenon is closely related to the issue of stability. Indeed, a filter that has all roots of the polynomial in the denominator inside the unit circle is guaranteed to be stable! These roots are called *poles*.

Suppose we have some transfer function with a polynomial, $B(z)$, in the numerator and another, $A(z)$, in the denominator, i.e.,

$$H(z) = \frac{B(z)}{A(z)}, \tag{4.35}$$

then the values for which $B(z) = 0$ are the zeros, $\{z_k\}$, and the values for which $A(z) = 0$ are the poles, $\{p_k\}$, of the filter. The zeros cancel signal contents, while the poles amplify it, so the zeros are never a concern in terms of stability but poles are. If the poles fall inside the unit circle, that is, $|p_k| < 1$, the filter is guaranteed to be stable. So, given a set of filter coefficients, we have to compute the roots of the polynomial in the denominator to make sure that the filter is stable. Note that poles or zeros at the origin are equally far from all points on the unit circle and thus do not have any impact—that is why the z in the denominator of (4.22) does not matter. Filters are often analyzed by plotting their poles and zeros in the complex plane. This is then called the z-plane.

4.5 Resonance Filter

A bump in the frequency response of a filter (or a mechanical system) is called a resonance. In what follows, we will investigate how to design a resonance filter, i.e., a filter that produces such a resonance. The following is based on [2]. With our knowledge of filters, we can now guess that we can construct such a bump by using a feedback filter and putting its poles in somewhere in the vicinity of the unit circle, at an angle corresponding to the frequency where we would like to have the bump. The frequency in the middle of such a bump is called the *center frequency* and the width of the bump is called the *bandwidth*. The desired frequency response of such a filter is shown in Fig. 4.12 with center frequency ϕ and bandwidth B. Suppose we use a pair of poles[1] with radius R and angle $\pm\theta$, i.e., the poles are

$$p_1 = Re^{j\theta} \quad \text{and} \quad p_2 = Re^{-j\theta}. \tag{4.36}$$

Notice that $p_1 = p_2^*$, where $\cdot^*$ means the complex conjugate. The location of the poles is shown in Fig. 4.13. The corresponding transfer function is then

$$H(z) = \frac{1}{\left(1 - p_1 z^{-1}\right)\left(1 - p_2 z^{-1}\right)}. \tag{4.37}$$

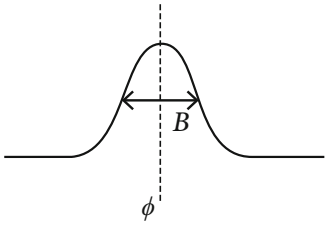


Fig. 4.12 Desired frequency response of resonator filter with a center frequency of ϕ and a bandwidth of B

[1] For filters with real coefficients, poles come in complex-conjugate pairs, unless the poles are real.

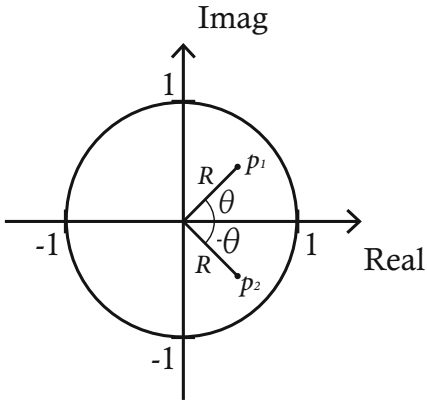


Fig. 4.13 The complex-conjugate pole pair, p_1 and p_2 with $p_1 = p_2^*$, of a resonator filter

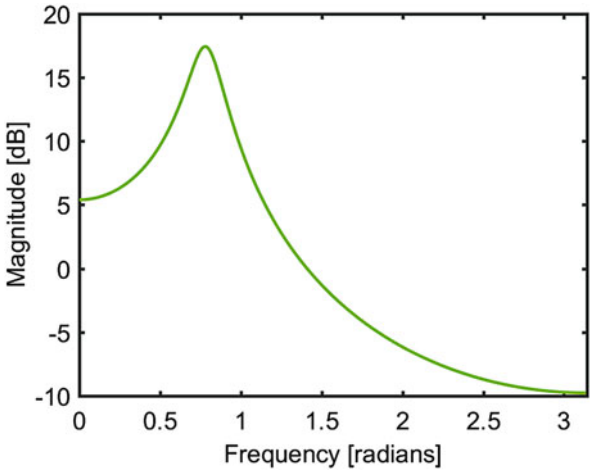


Fig. 4.14 Magnitude response of the resonator filter in (4.39) with $R = 0.9$ and $\theta = \pi/3$

This can be verified by multiplying by z^2 in the numerator and the denominator:

$$H(z) = \frac{1}{\left(1 - Re^{j\theta}z^{-1}\right)\left(1 - Re^{-j\theta}z^{-1}\right)} \tag{4.38}$$

$$= \frac{1}{1 - 2R\cos\theta z^{-1} + R^2 z^{-2}}. \tag{4.39}$$

In Fig. 4.14, the characteristic magnitude response of the resonator filter is shown for $R = 0.9$ and $\theta = \pi/3$. We can see that the corresponding difference equation is given by

$$y_n = x_n + (2R\cos\theta)\, y_{n-1} - R^2 y_{n-2}. \tag{4.40}$$

The question is then how to choose R and θ such that we get the bump we want where we want it. That is the essence of filter design. In this connection, a question is

how to define bandwidth. Bandwidths and cutoff frequencies are usually defined as 3 dB points, that is, the points where the frequency response is 3 dB below or above some reference. In our case, the bandwidth can be defined as the width at which the bump is 3 dB beneath its maximum. Recall that the formula for converting between a linear scale and the dB scale is

$$G = 20 \log_{10} g \text{ [dB]}, \tag{4.41}$$

where g is the linear gain. Using the formula $g = 10^{\frac{G}{20}}$, a G of -3 dB thus corresponds to $g = \frac{1}{\sqrt{2}}$. To find the bandwidth, given some filter coefficients, we would then have to find the ωs for which $|H(\omega)| = \frac{1}{\sqrt{2}}$ assuming that the magnitude response is 1 (i.e., unit gain) at its maximum. In our case, some math shows that the maximum of our resonance filter occurs at

$$\cos \phi = \frac{1 + R^2}{2R} \cos \theta \tag{4.42}$$

for which the gain is

$$|H(\phi)|^2 = \frac{1}{\left(1 - R^2\right)^2 \sin^2 \theta}. \tag{4.43}$$

In our case, the bandwidth, B, is found by solving for $|H(\omega)|^2 = \frac{1}{2}|H(\phi)|^2$, which yields

$$B \approx 2 - 2R. \tag{4.44}$$

From all this, we can gather that to get a resonance at frequency ϕ with bandwidth B, we must first compute the required pole radius, R,

$$R \approx 1 - B/2. \tag{4.45}$$

Then, we can compute the pole angle, θ, as

$$\cos \theta = \frac{2R}{1 + R^2} \cos \phi. \tag{4.46}$$

Finally, to get unit gain at the resonance, we compute the required scaling, the gain g, as

$$g = \left(1 - R^2\right) \sin \theta. \tag{4.47}$$

This gain is then simply applied to the input signal, x_n, to obtain a gain of 1 at the center frequency. The difference equation of the final resonance filter is then given by

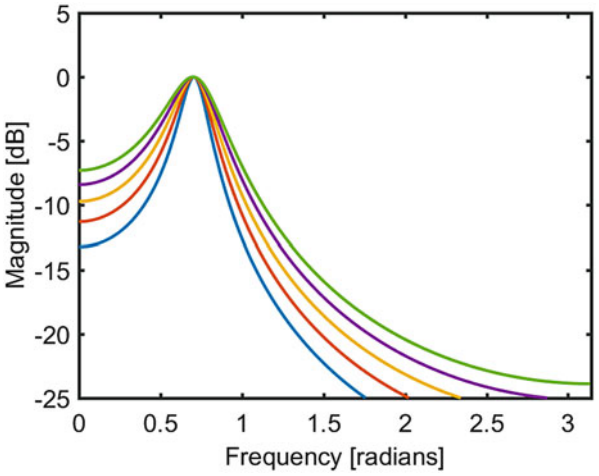
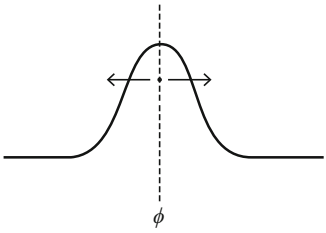


Fig. 4.15 Magnitude response of the resonator filter in (4.48) with $\psi = \pi/4$ for different bandwidths, B. Note how the different resonator filters have a gain of 0 dB at the center frequency

Fig. 4.16 The wah-wah effect illustrated as a resonator filter with time-varying center frequency, ϕ



$$y_n = g x_n + (2R\cos\theta)\, y_{n-1} - R^2 y_{n-2}. \tag{4.48}$$

In Fig. 4.15, some examples of the magnitude responses of resonator filters of the form (4.48) are shown. The shown filters all share the same center frequency, namely, $\psi = \pi/4$, but have different bandwidths. The filter coefficients are then computed from the desired center frequency, ψ, and bandwidth, B. Notice how the gain, g, as computed using (4.47) causes the filters to have 0 dB gain (corresponding a gain of 1) at the center frequency.

The resonance filter can be used for many interesting things, one of them being for the characteristic wah-wah effect that can be heard in such famous songs as Jimi Hendrix's Voodoo Child and Isaac Hayes' theme song for the movie Shaft. The wah-wah effect is basically a resonance filter whose parameters are varied by a footpedal. The frequency response of such an effect is illustrated in Fig. 4.16. Both the center frequency ϕ and the bandwidth B are varied when the pedal angle is changed. Note that the center frequency and the bandwidth are related to the so-called quality factor, Q, which often appears in analog filters, as ϕ/B.

4.6 Exercises

The following exercises, which are of a practical nature, are intended to be solved in Pure Data (or similar software) based on the material presented in this chapter. As such, they aim at developing an intuition and understanding of theory and practice.

Exercise 1 Design and test a simple feedforward filter

$$y_n = x_n + ax_{n-1}$$

in Pd using the `biquad~` object with a slider for changing a (with values from -1 to 1). Use the `noise~` object to generate an input for the filter and connect the output to `dac~`.

Exercise 2 Design and test a simple feedback filter

$$y_n = x_n + by_{n-1}$$

in Pd with a slider for changing b (with values ranging from 0 to 1). Use the `noise~` object to generate an input for the filter and connect the output to `dac~`. How does the output sound compared to the feedforward filter?

Exercise 3 Build and test a resonator filter in Pd using the `biquad~` object, i.e.,

$$y_n = x_n + b_1 y_{n-1} + b_2 y_{n-2}.$$

The filter coefficients b_1 and b_2 should be computed from a desired center frequency and bandwidth (i.e., from (4.45) and (4.46)), with sliders for controlling the center frequency and the bandwidth. Experiment with the possible parameter values for the sliders.

Exercise 4 The resonator filter can be used as a basis for the classical wah-wah effect. Use the resonator filter from Exercise 3 to build an auto-wah. In an auto-wah, the center frequency of the resonator filter (or similar) is controlled by a low-frequency oscillator. Experiment with the settings to achieve the desired result.

Chapter 5
Comb Filters and Periodic Signals

5.1 Introduction

In the previous chapter, we dove into the concept of filters and filtering. In this chapter, we will start out by taking a close look at a particular kind of filter, namely the comb filter and its inverse. The comb filter forms the basis of many of the audio effects that we will go into detail about later in this book. These audio effects include the echo effect, chorus, and flanger, but comb filters are also important building blocks of more complicated things, like artificial reverberators, pitch estimators, and, as we shall see here, synthesizers.

In a comb filter, a delayed and scaled version of the output is added to the input to produce the new output. When the delay is large enough, it produces an infinite number of echos of the original input, but when the delay is low, its output is not perceived as echos. The comb filter has a frequency response that is reminiscent of the teeth of a comb, hence the name. It has an impulse response that is only non-zero at regular intervals, where the interval is determined by the delay, and the scaling determines whether the non-zero values (the echos) grow, decay, or stay the same as a function of time. If the echos decay slowly or stay the same, the impulse response looks and sounds periodic and has a pitch, which is determined by the delay. Hence, we can produce sounds of any pitch we desire by exciting such a comb filter and adjusting the delay. This is the basic idea of the Karplus–Strong algorithm, a famous method for plucked-string synthesis. There is then also a close connection between the output of the comb filter and Fourier series, since periodic signals can be represented using it.

We will here introduce the comb filter, its close relative, the inverse comb filter, and show how the comb filter can be used for synthesis of something akin to the sounds produced by plucked-string instruments. Finally, we will revisit periodic signals and the Fourier series and discuss how they too can be used for synthesis.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_5

5.2 Comb Filters

Comb filters are an important part of many audio processing systems and for this reason we will now introduce comb filters and analyze them a bit. The classical comb filter is a feedback filter, where a version of the output delayed by D samples is added to the input signal, i.e., it has the difference equation [7]:

$$y_n = x_n + R^D y_{n-D}, \tag{5.1}$$

where $0 < R < 1$. The comb filter is illustrated in the block diagram in Fig. 5.1. To inspect the impulse response of the filter, we put the Kronecker delta function, δ_n, on the input, which yields the sequence (see Fig. 5.2):

$$1, 0, \ldots, 0, R^D, 0, \ldots, 0, R^{2D}, 0 \ldots, \tag{5.2}$$

where each non-zero value is followed by $D - 1$ zeros. The sequence can be seen to decay for $R < 1$, and the filter is thus stable. For $R = 1$, the sequence would be $1, 0, \ldots, 0, 1, 0, \ldots, 0, 1, 0 \ldots$, which is called a pulse (or impulse) train, since it is a series of equally spaced pulses. Since the pulses are D samples apart, this corresponds to a period of D/f_s in seconds for a sampling frequency of f_s. Interestingly, a filter of the form $y_n = x_n - a y_{n-D}$ with $a > 0$ is also a comb filter, only it does not have a peak at 0 Hz.

By replacing y_n by $Y(z)$ and x_n by $X(z)$ and using that a delay of L samples can be written as z^{-D}, we get

$$Y(z) = X(z) + R^D Y(z) z^{-D}. \tag{5.3}$$

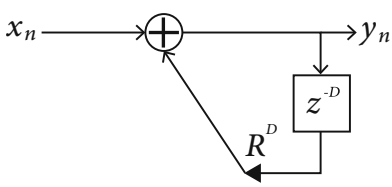


Fig. 5.1 A comb filter which is a feedback filter with a delay of D samples in the feedback path

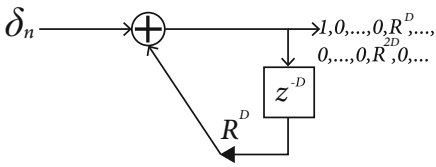


Fig. 5.2 Impulse response of the comb filter

From this, we can see that the transfer function of the comb filter is given by:

$$H(z) = \frac{Y(z)}{X(z)} \tag{5.4}$$

$$= \frac{1}{1 - R^D z^{-D}}. \tag{5.5}$$

Recall that poles are values of z for which the polynomial in the denominator of a transfer function is zero, in this case $1 - R^D z^{-D} = 0$. The filter has poles when $z^D = R^D$, i.e., for $p_l = Re^{j\frac{2\pi}{D}l}$ for $l = 0, 1, 2, \ldots, D-1$. We can verify this by substituting z in (5.5) by $Re^{j\frac{2\pi}{D}l}$, i.e.,

$$H(Re^{j\frac{2\pi}{D}l}) = \frac{1}{1 - R^D \left(Re^{j\frac{2\pi}{D}l}\right)^{-D}} \tag{5.6}$$

$$= \frac{1}{1 - R^D R^{-D} e^{-j2\pi l}} \tag{5.7}$$

$$= \frac{1}{1 - 1^l} \tag{5.8}$$

which is zero in the denominator for $l = 0, 1, \ldots, D-1$. This means that the filter has D poles, and that they are uniformly spaced between 0 and 2π at a radius of R. Since the frequency response exhibits peaks for frequencies corresponding to angles where the poles are the closests to the unit circle, it follows that the comb filter will have uniformly spaced sharp peaks $\frac{2\pi}{D}$ apart in terms of frequency, which leads to the characteristic comb-like magnitude response.

There is also something called the inverse comb filter, whose difference equation is given by:

$$y_n = x_n - R^D x_{n-D}. \tag{5.9}$$

As we can see, this is a feedforward filter, but like the comb filter, it involves a delay of D samples a filter coefficient of R^D. It is shown in Fig. 5.3. Its impulse response is the sequence:

$$1, 0, \ldots, 0, -R^D, 0, \ldots, \tag{5.10}$$

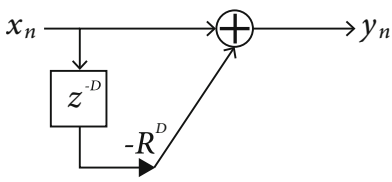


Fig. 5.3 An inverse comb filter which is a feedforward filter with a delay of D samples in the feedforward path

with $D-1$ zeros in between the 1 and $-R^D$. The impulse response is thus finite and we get only a single echo of the impulse. The z-transform of the difference equation can be seen to be

$$Y(z) = X(z) - R^D X(z) z^{-D}. \tag{5.11}$$

The transfer function of this filter is thus given by:

$$H(z) = \frac{1 - R^D z^{-D}}{1} \tag{5.12}$$

If we connected the comb filter and the inverse comb filter in series, we would have to multiply their transfer functions, i.e.,

$$\frac{1}{1 - R^D z^{-D}} \times \frac{1 - R^D z^{-D}}{1} = 1. \tag{5.13}$$

As we can see, the two filters would cancel out, and the filters must thus be each other's inverse, hence the names. From this, we can also clearly see that the inverse comb filters have zeros at the same places the comb filter had poles, i.e., $z_l = Re^{j\frac{2\pi}{D}l}$. When a pole and a zero fall on top of each other in the z-plane, they cancel out. Instead of strong peaks in the magnitude response, the inverse comb filters have deep valleys. In Fig. 5.4, the magnitude response of the comb filter with $D = 16$ and $a = 0.9$ is shown along with the corresponding inverse comb filter.

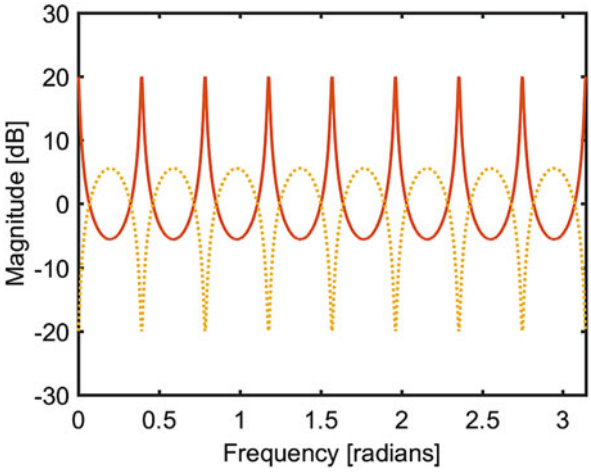


Fig. 5.4 Magnitude response of the comb filter and the inverse comb filter with $D = 16$ and $a = 0.9$

5.3 Plucked-String Synthesis

As we have seen, we can use a comb filter to produce a signal that is a repeating version of itself, i.e., echos. With $R = 1$, the output of the comb filter is a repeating pulse, what we call a pulse train, when we put an impulse on the input. If we put something else on the input, like a short burst of noise, the output will also be echos of that noise burst. With $0 < R < 1$, the echos will get fainter and fainter over time. Signals that repeat with a period of D samples will, when in a certain range, result in a perceived pitch of $f_0 = f_s/D$ with a sampling frequency of f_s. However, if D is large, we will perceive the repetitions of the signal as distinct echos.

As argued above, we can pick a D to produce a tone of our choice. This is exactly the idea behind the Karplus–Strong algorithm [2] which is a method for plucked-string synthesis, i.e., for replicating the sound produced by, for example, a guitar. We will now go into a bit more detail about the Karplus–Strong algorithm. We start with a comb filter of the form:

$$y_n = x_n + a y_{n-D}. \tag{5.14}$$

As mentioned, with $a = 1$, we will get an impulse response that goes on forever and ever and does not decay. In the real world, sounds do not go on forever. Energy is lost over time and the sound will eventually die out. We can mimic this behavior with $|a| < 1$. As we have learned, this filter has a very characteristic frequency response, with resonance peaks at regularly spaced intervals. If we put an impulse on the input of the comb filter, the Fourier transform of the output will be identical to this frequency response. Now, if we compare the frequency response to the spectrum of the sound of a real plucked-string instrument, we will notice that the higher frequencies die out faster than the low ones, and we would like our plucked-string synthesis to exhibit the same behavior. To realize this, the idea in the Karplus–Strong algorithm is to insert a low-pass filter in the comb filter, such that every time our impulse passes the low-pass filter, some energy at higher frequencies will be attenuated. A very simple low-pass filter will do the trick, like this

$$y_n = \frac{1}{2}x_n + \frac{1}{2}x_{n-1}. \tag{5.15}$$

This filter just adds two consecutive samples and divides the result by two, as can be seen in Fig. 5.5. We can thus think of this as an averaging operation that computes the mean. Indeed, this is a common and useful interpretation of low-pass filters in general. Conversely, one can think of high-pass filters as removing slow variations, including the mean, while preserving fast ones. The low-pass filter in (5.15) has the transfer function:

$$H(z) = \frac{1}{2} + \frac{1}{2}z^{-1}, \tag{5.16}$$

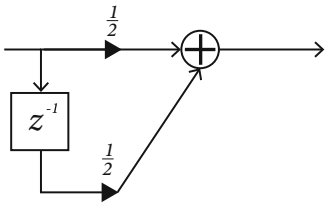


Fig. 5.5 A simple low-pass filter which can be inserted into the comb filter

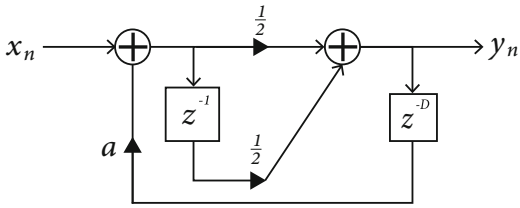


Fig. 5.6 Modified comb filter of the Karplus–Strong algorithm, with a low-pass filter inserted in the comb filter

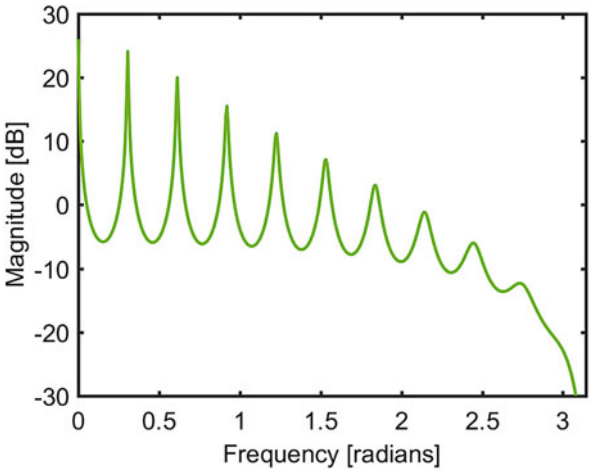


Fig. 5.7 Magnitude response of the combined comb and low-pass filter with $D = 20$ and $a = 0.95$

and its frequency response is $H(\omega) = \cos(\frac{\omega}{2})e^{-j\omega\frac{1}{2}}$. From this, we can easily see that the magnitude response is given by:

$$|H(\omega)| = \left|\cos\left(\frac{\omega}{2}\right)\right|. \tag{5.17}$$

The modified comb filter with the low-pass filter inserted is shown in Fig. 5.6. In Fig. 5.7, the magnitude response of the modified comb filter is shown for $D = 20$ and $a = 0.95$. As can be seen from the figure, the desired effect of attenuated high frequencies has been achieved. The phase response, $\theta(\omega)$, of the low-pass filter is given by $\theta(\omega) = -\omega\frac{1}{2}$. This is what is known as *linear phase*, as the phase is a linear function of ω. This means that all frequencies are delayed by the same amount of samples, here half a sample. This half a sample poses a problem for us. Think of the impulse response of the filter (Fig. 5.6) and how an impulse is affected by the

different blocks. There is a delay of D between samples due to the original comb filter, but now there is an additional delay due to the low-pass filter. In total, this means that there is $D + \frac{1}{2}$ samples delay between non-zero output values! This obviously changes the pitch and we must adjust our desired D accordingly. This leads us to a related issue. Since we would like to be able to generate an output having any desired pitch, this means that D can take on an arbitrary value, and not just an integer. For example, an A4, which is 440 Hz in standard tuning, corresponds to $\frac{f_s}{440} = 100.2273$. If we round this to the nearest integer, i.e., 100, the result would be a pitch of 441 Hz.

So, in summary, we must take the half-sample delay of the low-pass filter into account, and, to be able to output any desired pitch, we must be able to delay a signal by a non-integer number of samples. This can be done in several ways, the most obvious one being some sort of interpolation, but it can also be achieved by filtering, using, for example, what is known as a *fractional delay filter*, which is a particular application of a type of filter known as an all-pass filter. An all-pass filter having a single filter coefficient, b, has the following difference equation:

$$y_n = bx_n + x_{n-1} - by_{n-1}, \tag{5.18}$$

and its transfer function is given by:

$$H(z) = \frac{b + z^{-1}}{1 + bz^{-1}}. \tag{5.19}$$

The frequency response is obtained by substituting z by $z = e^{j\omega}$. This yields

$$H(\omega) = \frac{b + e^{-j\omega}}{1 + be^{-j\omega}}. \tag{5.20}$$

To understand what the all-pass filter does, we will first find the magnitude response, $|H(\omega)|$. The magnitude of the numerator is

$$|b + e^{-j\omega}| = \sqrt{b^2 + 2b\cos\omega + 1} \tag{5.21}$$

while the magnitude of the denominator is

$$|1 + be^{-j\omega}| = \sqrt{1 + 2b\cos\omega + b^2}. \tag{5.22}$$

Hence, we have that $|H(\omega)| = 1$ for all frequencies, ω. As we can see from this, the all-pass filter does not change the magnitude spectrum of the input signal. What is the point of such a filter then? The point is that it changes the phase, and this is thus an example of a filter, where the phase response is not only important but absolutely essential. Recall that the phase response of a filter is defined as:

$$\theta(\omega) = \angle H(\omega) = \tan^{-1} \frac{\text{Imag}\{H(\omega)\}}{\text{Real}\{H(\omega)\}}, \tag{5.23}$$

for Real $\{H(\omega)\} \neq 0$. It is often quite difficult to find nice, simple expressions of the phase response of a filter. We will refrain from going into too many details with the process of finding phase response of the all-pass filter, but the result is

$$\theta(\omega) = -2\tan^{-1}\left(\frac{1-b}{1+b}\tan\left(\frac{\omega}{2}\right)\right). \tag{5.24}$$

This can be simplified further by assuming that ω is small, in which case we can use the approximation $\tan(x) \approx x$ twice. This leads to the following approximation of the phase response:

$$\theta(\omega) \approx -\frac{1-b}{1+b}\omega. \tag{5.25}$$

defining $d = \frac{1-b}{1+b}$, the phase response can be expressed as $\theta(\omega) \approx -d\omega$. The form of the phase response is thus similar to the phase response of the simple low-pass filter in (5.15), in fact, we can say that it is approximately linear phase with a delay d, but the all-pass filter does not attenuate the high frequencies. If we wish to design an all-pass filter with a fractional delay d where $0 < d < 1$, we thus have to solve for a in either (5.24) or (5.25). Obviously, it is simplest to design the filter using (5.25), in which case the filter coefficient should be selected as:

$$a = \frac{1-d}{1+d}, \tag{5.26}$$

but the result would be better if we use (5.24). With the all-pass filter in place, we can now realize an arbitrary pitch with our Karplus–Strong algorithm! The final filter is shown in Fig. 5.8. The delay of the low-pass filter was $\frac{1}{2}$ samples, the all-pass filter has a fractional delay of d (i.e., between 0 and 1), and the comb filter has an integer delay D. Thus, the total delay (in samples) is given by:

$$K = \frac{1}{2} + d + D. \tag{5.27}$$

As an example, let us say that we would like to produce a C5, which is 523.25 Hz. With a sampling frequency of 44,100 Hz, this corresponds to $44{,}100/523.25 = 84.2809$ samples. So, to get this pitch, we need to have $K = 84.2809$. Simple arithmetic reveals that the integer delay then has to be $D = 83$ and the fractional part is then $d = 0.7809$. Now, we can use either (5.24) or (5.25) to determine the coefficient b in the all-pass filter. In Fig. 5.9, an example of the phase response of an all-pass filter is shown for $b = 1/3$, which corresponds to a desired fractional delay of 0.5 samples. It can be seen that the all-pass filter has a phase response that is approximately linear for low frequencies, which results in a delay of the same amount of samples for all frequencies. For high frequencies, however, the function is not linear.

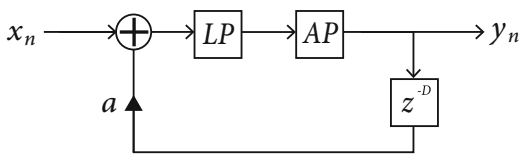


Fig. 5.8 Karplus–Strong algorithm with a low-pass (LP) and an all-pass (AP) filter. The total delay of the combined filter is $\frac{1}{2} + d + D$

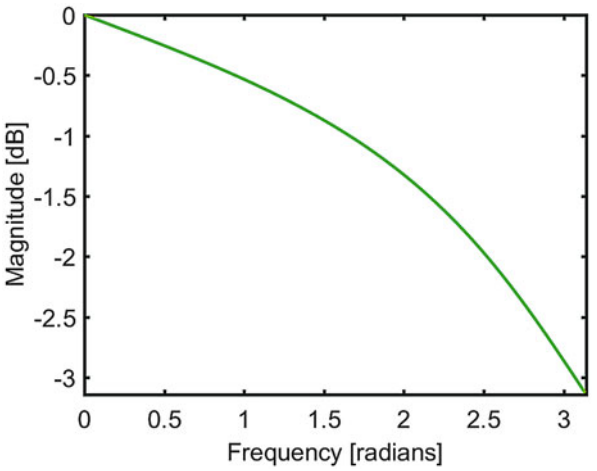


Fig. 5.9 Phase response of the all-pass filter with $b = 1/3$ corresponding to a desired delay of 0.5 samples

There is one final issue that we have not touched upon. To get something out of our filter, we must put something in. We could of course simply put a digital impulse, i.e., the Kronecker delta function, on the input, but this might be too simple. Indeed, the Karplus–Strong algorithm is often excited with a short burst of random noise instead.

5.4 Pitch, Notes, Scales, etc.

We have seen how we can use a comb filter to generate sounds having a desired pitch. But what pitches, or more specifically what frequencies, exactly do we desire to produce then? To answer this question, we must briefly dive into a bit of music theory and introduce some terminology and notation. The atoms of music are notes. A musical scale is a set of musical notes, a chord is a set of notes played simultaneously, and a melody is an ordered set of notes.

Most Western music is based on the twelve-tone equal temperament, which is a tuning system that has twelve different notes, called semitones, within an octave. In music, two notes having frequencies related by an integer (i.e., one is equal to an integer times the other) are said to be an octave apart and belong to the same pitch class or chroma. For example, 220 and 440 Hz are both the note A, but belong to different octaves. Within an octave, the notes are named A, A♯, B, C, C♯, D, D♯, E, F, F♯, G, and G♯. The naming is not unique, as each note can be sharp, indicated

by ♯, or flat, indicated by ♭, which means a semitone above or below the indicated note. For example, A♯ and B♭ are the same note. To indicate to which octave a note belongs, subscript is often used. For example, A_2 is 110 Hz, while A_3 is 220 Hz and so forth.

Ratios between frequencies are called intervals in music and can also be thought of as differences on a logarithmic scale, and these intervals can be measured in terms of semitones, octaves, or cents. These intervals are what define harmony, scales, chords, etc. The reason for this is that equal intervals (i.e., frequency ratios) are perceived as being equivalent in difference in pitch. For example, the difference between the notes A and B is perceived as the same as the difference between F and G. In Hertz, this means that the difference between 110 and 220 Hz is perceived as the same as the difference between 440 and 880 Hz. In equal temperament, consecutive semitones, for example the pair E and F and the pair B and C, have equal frequency ratios, so within an octave we have 12 tones spaced $2^{1/12} \approx 1.05946$ apart, meaning that for a particular note having frequency f_1 the next frequency is $1.05946 f_1$. Since these intervals are relative (i.e., the same ratio can be achieved with many different frequencies), a frequency reference must be chosen to assign absolute values to notes and for instruments to be able to play together in harmony. The note A_4 has, therefore, been agreed upon as the reference with a value of 440 Hz, which is commonly used in Western music, although other values can occur. Thus, all semitones in equal temperament with standard concert pitch 440 Hz can be expressed as $2^{\frac{k}{12}} 440$ Hz where k is an integer.

Sometimes, intervals are also expressed in cents. Cents is a sub-semitone unit of 100 cents per semitone, i.e., 1200 cents per octave. The interval between two frequencies f_1 and f_2 can be computed in cents as:

$$\Delta = 1200 \log_2 \frac{f_1}{f_2}. \tag{5.28}$$

In computers, pitch is often represented using the MIDI Tuning Standard. In this standard, the pitch, F_0, is represented by the number:

$$F_0 = 69 + 12 \log_2 \left(\frac{f_0}{440 \text{ Hz}} \right), \tag{5.29}$$

where f_0 is the pitch in Hertz. When f_0 is equal to a semitone, F_0 is an integer, so rounding can be used in (5.29) to obtain the closests MIDI note from a frequency f_0. The 440 Hz is the standard tuning and corresponds to MIDI note 69. To obtain the frequency in Hertz from the MIDI note number, the following expression can be used:

$$f_0 = 2^{\frac{F_0 - 69}{12}} 440 \text{ Hz}. \tag{5.30}$$

Table 5.1 Notes for four octaves in standard pitch and their frequencies in Hertz and the corresponding MIDI numbers

Note	Freq. (Hz)	MIDI	Note	Freq. (Hz)	MIDI
C_1	32.70	24	C_3	130.81	48
$C_1\sharp$	34.65	25	$C_3\sharp$	138.59	49
D_1	36.71	26	D_3	146.83	50
$D_1\sharp$	38.89	27	$D_3\sharp$	155.56	51
E_1	41.20	28	E_3	164.81	52
F_1	43.65	29	F_3	174.61	53
$F_1\sharp$	46.25	30	$F_3\sharp$	185.00	54
G_1	49.00	31	G_3	196.00	55
$G_1\sharp$	51.91	32	$G_3\sharp$	207.65	56
A_1	55.00	33	A_3	220.00	57
$A_1\sharp$	58.27	34	$A_3\sharp$	233.08	58
B_1	61.74	35	B_3	246.94	59
C_2	65.41	36	C_4	261.63	60
$C_2\sharp$	69.30	37	$C_4\sharp$	277.18	61
D_2	73.42	38	D_4	293.66	62
$D_2\sharp$	77.78	39	$D_4\sharp$	311.13	63
E_2	82.41	40	E_4	329.63	64
F_2	87.31	41	F_4	349.23	65
$F_2\sharp$	92.50	42	$F_4\sharp$	269.99	66
G_2	98.00	43	G_4	392.00	67
$G_2\sharp$	103.83	44	$G_4\sharp$	415.00	68
A_2	110.00	45	A_4	440.00	69
$A_2\sharp$	116.54	46	$A_4\sharp$	466.16	70
B_2	123.47	47	B_4	493.88	71

For reference, the notes for four octaves are listed in Table 5.1 along with their frequency in Hertz and the corresponding MIDI number. It should be noted that while 440 Hz is the most common tuning, others are sometimes used.

A scale is a set of notes defined by the intervals of the notes in relative to the root note, sometimes also called the tonic. For example, the A minor scale, where A is the root note, comprises A, B, C, D, E, F♯, and G. The intervals between consecutive notes, expressed in semitones, can be seen to be 2, 1, 2, 2, 1, 2, ... which can be extended cyclically. The minor scales for the other notes can be obtained using these intervals, and other scales can be obtained from different intervals between the different steps. For example, the diminished pentatonic A minor scale comprises the five notes A, C, D♯, E, and G. A chord is then a set of two or more notes played simultaneously. An A minor chord, for example, consists of the notes A, C, and D, which can be seen to be the root and the third and fifth note of the A minor scale. An A major chord consists of A, C♯, and D, which are the root, third, and the fifth note of the A major scales. Chords and scales are thus both defined by the root note, after which the chord is named, and the involved intervals, which can be expressed in terms of steps.

5.5 Periodic Signals

Signals that repeat are said to be periodic. Earlier in this chapter, we saw an example of a periodic signal, namely the impulse response of a comb filter with $R = 1$, which was a so-called pulse train. For simplicity, we will here go back to discuss analog signals, which are continuous functions of time. Mathematically, we can express periodicity as the property that a signal $x(t)$ is equal to a delayed version of itself $x(t - T)$ with period T. We use t here to indicate that in this context, we are talking about continuous signals (i.e., functions). We can express this property as:

$$x(t) = x(t - T) \tag{5.31}$$

over some interval of t. This is illustrated in Fig. 5.10. It follows that we also have that $x(t) = x(t + kT)$ where $k = 0, \pm 1, \pm 2, \ldots$. Periodicity is at the heart of music signals, not just on a signal level but also on a structural level. What we call notes in music are periodic sounds, they are the atoms we perceive when listening to sounds. The phasors we have studied extensively so far have also been periodic, but many more signals belong to the class of periodic signals. It is these signals that we will study in a bit more detail here.

Mathematically, periodic signals are also very interesting, and it was actually the study of such functions that laid the ground for what is known as function analysis in mathematics. It all started with Jean-Baptiste Joseph Fourier who was studying heat transfer (in canons, no less) and vibrations. Through his work, he discovered what we now refer to as Fourier series. The Fourier series is a decomposition of a periodic signal into atoms or basic building blocks, which turn out to be sinusoids, or phasors as we might be inclined to call them.

The basic building block is of the form:

$$e^{j\omega_0 kt} \quad \text{for} \quad k = 0, \pm 1, \pm 2, \ldots \tag{5.32}$$

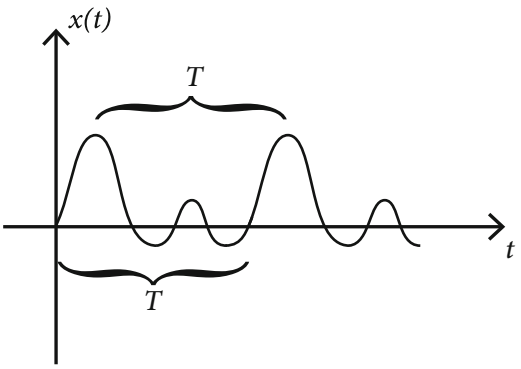


Fig. 5.10 Illustration of a periodic signal, here having a period of T. The signal repeats itself and we thus have that $x(t) = x(t - T)$ for all t

and with the definition $z = e^{j\omega_0}$, we might write this as z^{kt}. ω_0 is a so-called fundamental frequency, which is determined by the period T as $\omega_0 = \frac{2\pi}{T}$. We can see that the frequencies of the phasors in the above equation are all integer multiples of this fundamental frequency, i.e., $w_0 1$, $w_0 2$, $w_0 3$, ... The theory behind the Fourier series states that (almost) any[1] signal of period T can be expressed as an infinite sum of phasor, all of which are periodic with period T, i.e.,

$$x(t) = \sum_{k=-\infty}^{\infty} c_k e^{j\omega_0 kt} \tag{5.33}$$

where c_k are complex coefficients that scale each phasor. So, all periodic signals we can think of can be expressed this way. Since the period T determines ω_0 all that is left to turn the phasors into a specific waveform are the coefficients c_k. In the case of continuous signals, these are given by the integrals:

$$c_k = \frac{1}{T} \int_0^T x(t) e^{-j\omega_0 kt} dt \tag{5.34}$$

for $k = 0, \pm 1, \pm 2, \ldots$. If the signal in question is bandlimited (i.e., all frequency contents above a certain frequency can be assumed to be zero), then the infinite series in (5.33) can be replaced by a finite sum, i.e.,

$$x(t) = \sum_{k=-K}^{K} c_k e^{j\omega_0 kt}. \tag{5.35}$$

Moreover, since all our signals are real, we have that $c_k = c_{-k}^*$, where $\cdot^*$ means the complex-conjugate. Sometimes, it is useful to think of signals as complex or convert them into complex signals, but then the property $c_k = c_{-k}^*$ does not apply.

The period of a periodic sound is closely related to the perceptual property called pitch. In most cases (except pathological ones), the fundamental frequency determines the perceived pitch of signals of the form (5.33). This means we can use Fourier series to generate any pitch we desire. Indeed, Fourier series can be, has been, and is used for music synthesis. The complementary perceptual property of timbre, which is the characteristics that make the sounds of the same volume and pitch sound different, must then be determined by the coefficients $\{c_k\}$. So, by choosing different coefficients, we can get sounds that have the same pitch but sound different. Thinking about phasors and filters, it makes sense. Since a phasor passed through a filter is simply scaled by a complex number (i.e., its magnitude and phase are changed), we can get a result identical to the Fourier series by passing a bunch of phasors, having frequencies $\omega_0 k$ where $k = 0, \pm 1, \pm 2, \ldots$ through filters that change their magnitude and phase.

[1] As mentioned earlier, the Fourier series converges for many types of signals, including those we are interested in, but not all.

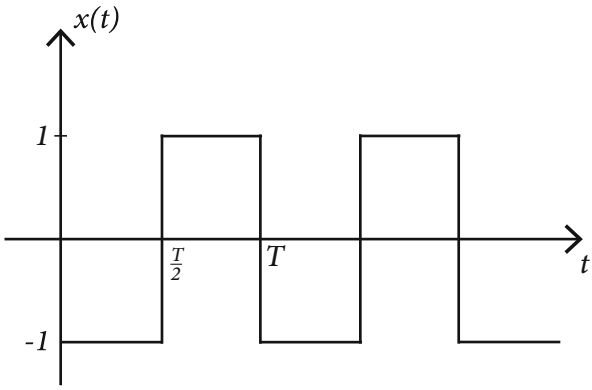


Fig. 5.11 Illustration of a periodic square wave

Let us explore some classical examples of Fourier series. They are very interesting because they are just about as far from nice, smooth sinusoids as we could possibly imagine, yet we can express them with a Fourier series. First, let us look at the square wave, an example of which is shown in Fig. 5.11. Let us now examine the Fourier series of a square wave, which is here defined for one period (i.e., for $0 < t < T$) as:

$$x(t) = \begin{cases} 1 & \text{for } 0 \leq t < \frac{T}{2} \\ -1 & \text{for } \frac{T}{2} \leq t < 1 \end{cases} \tag{5.36}$$

Note that this is slightly different from the one shown in Fig. 5.11. To find the coefficients of the corresponding Fourier series, we must use (5.34). By integration by parts, we arrive at

$$c_k = \begin{cases} -\frac{2j}{k\pi} & \text{for } k = 1, 3, 5, \ldots \\ 0 & \text{otherwise} \end{cases} . \tag{5.37}$$

This means that the real part of the coefficients, c_k, is zero, and the Fourier series is then given by:

$$x(t) = \sum_{k=1,3,5,\ldots}^{\infty} \frac{4}{\pi k} \sin(k\omega_0 t) . \tag{5.38}$$

We can see from the coefficients of the Fourier series that the magnitude of the coefficients, i.e., $|c_k|$, gets smaller and smaller as a function of k. Thus, the terms of the series corresponding to high k contribute less than the low ones. If we truncate the series of either of these signals to only K terms, and throw away the least significant terms, we get something that is only an approximation of the square or triangle wave, and the higher K is, the closer the series will get to them. As mentioned earlier, a bandlimited signal only has a finite number of terms, K.

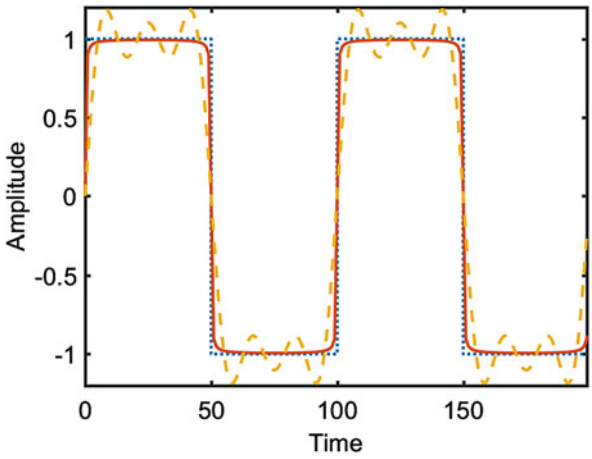


Fig. 5.12 Square wave (dotted) with a period of $T = 100$ and its Fourier series comprising 100 terms (solid) and 5 terms (dashed)

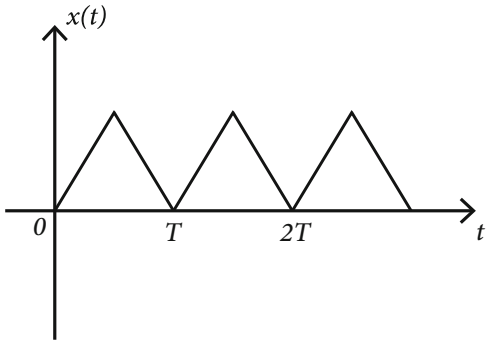


Fig. 5.13 Illustration of a periodic triangle wave

Thus, the effect of using a finite number of terms in (5.38) is identical to passing the signal, $x(t)$, through a low-pass filter. One of the effects of this is that the truncated series, or, equivalently, the low-pass filtered signal has "round corners."

In Fig. 5.12, the square wave can be seen along with its Fourier series comprising $K = 5$ and $K = 100$ terms, respectively, can be seen. A number of things can be observed from the figure. Firstly, even for $K = 100$, the Fourier series is not identical to the square wave. Secondly, we can see that the sharp edges become smooth. Thirdly, we can also see that there are oscillations when a low number of terms are used. These are called Gibb's phenomenon and are the result of the discontinuity in the transition from 0 to 1 (and vice versa) of the square wave.

Another example of a periodic signal is the triangle wave, which is depicted in Fig. 5.13. A triangle wave of period T (with no DC offset) has the following coefficients in its Fourier series:

$$c_k = \begin{cases} \frac{T}{\pi^2 k^2} & \text{for } k = 1, 3, 5, \ldots \\ 0 & \text{otherwise} \end{cases}. \tag{5.39}$$

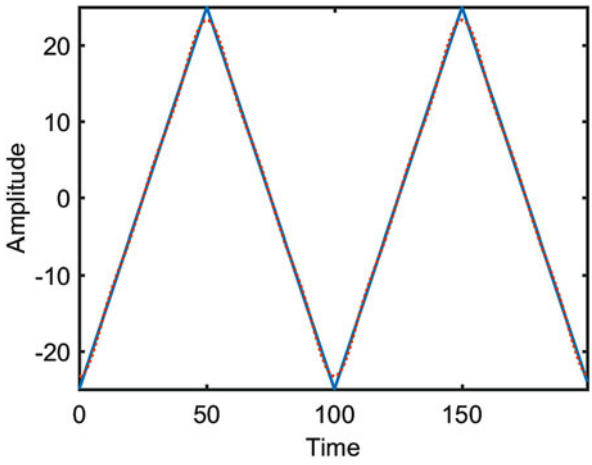


Fig. 5.14 Triangle wave (solid) with a period of $T = 100$ and its Fourier series comprising 100 terms (dotted)

The corresponding Fourier series is then given by:

$$x(t) = \sum_{k=1,3,5,\ldots}^{\infty} -\frac{2T}{\pi^2 k^2} \cos(k\omega_0 t) \,. \tag{5.40}$$

In Fig. 5.14, the Fourier series of the triangle wave is shown. Like for the square wave, we can see that the Fourier series does seem to make a good approximation of the triangle wave, but we can also see that the corners are now smooth.

The square wave and the triangle wave, which are easy to generate in practice, can also be used for music synthesis, and their timbre can be shaped by passing them through various filters, like, for example, the resonator filter.

5.6 Exercises

The exercises in this chapter are of a practical nature and aim at applying the covered theory to design various filters, ending with the Karplus–Strong synthesis, in Pure Data.

Exercise 1 First, we will build a simple echo effect. Implement an inverse comb filter of the form

$$y_n = x_n + bx_{n-D}.$$

Use the objects `delwrite~` and `d elread~` to implement the delays. Use sliders to control the delay D and the gain b. Test your echo filter with microphone input or a wavefile and play back the result by connecting the output to `dac~`.

Exercise 2 Next, implement a feedback version of the echo effect from a comb filter of the form

$$y_n = x_n + ay_{n-D}.$$

Again, build sliders to control the delay D and the feedback gain a. Test the echo filter. How does it sound compared to the one based on the inverse comb filter?

Exercise 3 We will now extend the comb filter to a Karplus–Strong synthesizer. Start by implementing the comb filter

$$y_n = x_n - ay_{n-D}.$$

in an abstraction in Pd and excite it with an impulse using the `dirac~` object. Use sliders to control the two parameters a and L. Extend it to also include the low-pass and all-pass filters of the Karplus–Strong filter by building an abstraction for each.

Exercise 4 You will need some more sophisticated input to excite the Karplus–Strong plucked-string synthesizer to make it sound realistic. A commonly used input is a short noise burst. Implement such a noise burst in Pd. You can use `noise~` for this.

Exercise 5 Build a patch in Pd that allows the user to play notes from the D minor scale using the Karplus–Strong plucked-string synthesizer from before.

Exercise 6 Fourier series can be used for creating practically any periodic waveform. Create a patch where the first five harmonics of a Fourier series, creating using the `osc~` object, are added with different amplitudes. The user should be able to control the fundamental frequency of the Fourier series as well as the amplitudes of each of the harmonics. Display the resulting signal and output it to the soundcard. Listen to the Fourier series and study how the result is affected by the amplitudes.

Chapter 6
More About Filters

6.1 Introduction

By now, the reader is familiar with the concept of filters and has a basic understanding of what they are and what they do. We have seen that digital filters come in the form of difference equations involving delayed versions of the input and output signals that are multiplied by so-called filter coefficients and added up. In this chapter, we will go more into details about such filters. In particular, we will introduce the z-transform and discuss its properties, and we will use it as a tool to represent filters as rational function. By doing this, we can think of filters as polynomials and we can understand and analyze the behavior of filters this way. We will then explore the properties of filters, more specifically their linearity. Finally, we will present some basic filter types that are often encountered in practice.

6.2 The Z-Transform

In the previous chapters, we used what we called the phasor representation of the input and output signals, x_n and y_n, which we wrote as $X(z)$ and $Y(z)$, respectively, and we used them to define the transfer function, $H(z)$, of a filter. The formal tool for doing this is called the z-transform, and we will now go into more details about it (without going overboard on the mathematics). For the input signal, x_n, the z-transform is defined as

$$X(z) = \sum_{n=-\infty}^{\infty} x_n z^{-n}. \tag{6.1}$$

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_6

In essence, we take the sample x_n for each time instance n and multiply it by z^{-n}. We can think of it as a transformation of x_n. Remember that x_n is just a sequence of numbers. Writing out (6.1), we get

$$X(z) = x_{-\infty}z^{\infty} \ldots + x_{-1}z^{1} + x_0 + x_1 z^{-1} + \ldots + x_{\infty}z^{-\infty}. \tag{6.2}$$

In shorthand notation, we write the transformation of x_n to $X(z)$ as $x_n \rightarrow X(z)$ and similarly for other quantities. Note that while we refer to $X(z)$ is a phasor representation of x_n, z^n does strictly speaking not have to be a phasor in the sense that it must lie on the unit circle, i.e., $|z| = 1$. In fact, z is here a more general, complex entity, an abstraction of the phasor concept. We can think of it as a decomposition of the contents of x_n into the terms of $X(z)$.

The summation in (6.3) goes from minus infinity to plus infinity but it is easily changed to take into account that our signal may only be defined for some range of n, for example, from 0 to $N-1$, by considering the signal to be 0 outside this interval. In this case, the z-transform is

$$X(z) = \sum_{n=0}^{N-1} x_n z^{-n}. \tag{6.3}$$

While the z-transform can certainly be applied to signals as described here, it is perhaps the most useful when applied to systems, in our case filters.

Recall that the input–output relation when operating in the z-transform domain is given by the following, as also illustrated in Fig. 6.1:

$$Y(z) = H(z)X(z), \tag{6.4}$$

where $Y(z)$ is the z-transform of the output signal, y_n.

Also, we previously introduced the Kronecker delta function, which was defined as

$$\delta_n = \begin{cases} 1 & \text{for } n = 0 \\ 0 & \text{otherwise} \end{cases}, \tag{6.5}$$

which means it is the sequence

$$0, \ldots, 0, 1, 0, \ldots, 0, \tag{6.6}$$

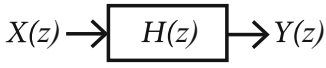


Fig. 6.1 Relationship between the z-transform of the input and the output of a filter, where the output is then given by $Y(z) = H(z)X(z)$

Fig. 6.2 The impulse response of a filter is obtained by putting an impulse on the input of the filter and measuring the output

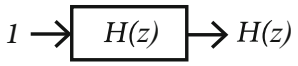


Fig. 6.3 The z-transform of the impulse response as the output of the filter when the input is an impulse which implies that $X(z) = 1$

where the value 1 occurs for $n = 0$. From this and (6.3), we can easily see that if $x_n = \delta_n$, then

$$X(z) = 1. \tag{6.7}$$

So, if we apply a digital impulse on the input of a filter, the output is

$$Y(z) = H(z)X(z) \tag{6.8}$$

$$= H(z). \tag{6.9}$$

In the time-domain, we know that if we apply a digital impulse on the input of a filter, the output is the impulse response, h_n, as illustrated in Fig. 6.2. In the z-transform domain, the output is then simply the transfer function of the filter, as shown in Fig. 6.3. There is thus an intimate relation between the impulse response of a filter and its transfer function. In fact, a different definition of the transfer function might be as the z-transform of the impulse response, h_n, which may be infinite or finite, i.e.,

$$H(z) = \sum_{n=-\infty}^{\infty} h_n z^{-n}. \tag{6.10}$$

This means that we can determine the transfer function of a filter either from the z-transform of the difference equation, or from the impulse response of the filter. For causal filters, i.e., filters that only depend on the present and the past, we have that $h_n = 0$ for $n < 0$. Normally, we design our filters such that they are causal, though, but mathematically they do not have to be, at least when using pen and paper.

When considering infinite sums like in (6.3), we are faced with a concern. How do we know that such an infinite sum converges to something meaningful? If it ends up equalling infinity (or minus infinity for that matter), we say that the series diverges. Indeed, we cannot take it for granted that the series will converge. When introducing the z-transform it is therefore customary in signal processing textbooks to analyze the convergence properties of the z-transform for different kinds of

signals and systems. The values of z for which the z-transform converges are called the *region of convergence* and this region is important for understanding signals and systems. However, we will here refrain from a detailed exposition of these matters and instead refer the interested reader to [7]. Suffice it to say that for filters that are stable and causal, the z-transform of its impulse response converges for values of z that include the unit circle, i.e., for phasors and thus also converge for Fourier transforms of it.

Note that when concerned only with a signal of finite duration, like x_n, y_n, or h_n (remember that the impulse response is also a signal), i.e., when they are only non-zero in some finite interval, we do not have to concern ourself with the problem of convergence, as long as the individual signal values are finite, which they will always be in practice.

Finally, let us explain why the multiplication of a signal by z^{-1} corresponds to a delay by one sample. Consider the signal x_n and its z-transform $X(z) = \sum_{n=-\infty}^{\infty} x_n z^{-n}$. The transform of x_{n-1} is then

$$\sum_{n=-\infty}^{\infty} x_{n-1} z^{-n} = \sum_{n=-\infty}^{\infty} x_n z^{-n-1} \tag{6.11}$$

$$= z^{-1} \sum_{n=-\infty}^{\infty} x_n z^{-n} \tag{6.12}$$

$$= z^{-1} X(z), \tag{6.13}$$

which shows that the z-transform of x_{n-1} is $z^{-1}X(z)$.

6.3 Filters as Rational Functions

We will now develop our understanding of filters a bit further by using the z-transform. In an earlier chapter, we considered a filter of the form

$$y_n = b_0 x_n + b_1 x_{n-1} + \ldots + b_M x_{n-M}. \tag{6.14}$$

This can also be written compactly as $y_n = \sum_{m=0}^{M} b_m x_{n-m}$. The transfer function of the filter is then given by

$$H(z) = \sum_{m=0}^{M} b_m z^{-m}. \tag{6.15}$$

This looks very much like a polynomial, if we treat z like an independent variable, except that the exponents are negative. We can write this into something

with positive exponents by multiplying and dividing by z^M, which results in the following:

$$H(z) = \frac{\sum_{m=0}^{M} b_m z^{-m}}{z^M} z^M \tag{6.16}$$

$$= \frac{\sum_{m=0}^{M} b_m z^{M-m}}{z^M}, \tag{6.17}$$

where we have used that $z^a z^b = z^{a+b}$. The filter now involves polynomials of the usual kind with positive exponents. However, for all practical purposes it makes no difference whether we are analyzing the filters with negative exponents, as in (6.15), or with positive exponents, as in (6.17). The above means that we can think of the transfer functions of filters as polynomials of sorts, which proves to be a key insight in understanding and manipulating filters. In fact, this type of function is called a *rational function*. It has a polynomial in the numerator and one in the denominator.

The frequency response of the filter is obtained by substituting z by $e^{j\omega}$. If we do this for (6.17), we obtain[1]

$$H(\omega) = \frac{\sum_{m=0}^{M} b_m e^{j\omega(M-m)}}{e^{j\omega M}} \tag{6.18}$$

$$= \sum_{m=0}^{M} b_m e^{-j\omega m}, \tag{6.19}$$

from which we can compute the magnitude and phase responses. Note how the multiplication by z^M in the numerator and denominator does not affect the frequency response of the filter. This also means that we can always just convert polynomials involving negative exponents into positives ones.

From the above we can see that the frequency response of the filter in (6.14) involves the quantity $H(\omega) = \sum_{m=0}^{M} b_m e^{-j\omega(m)}$, which is actually the Fourier transform of the coefficients $\{b_m\}$! Not only that, the impulse response of the filter in (6.14) is the sequence $b_0, b_1, \ldots, b_M$, so we have just confirmed the claim that the frequency response of the filter can be obtained by computing the Fourier transform of the impulse response. In fact, we could also see this by making the substitution z by $e^{j\omega}$ in (6.10) instead, from which we would then obtain

$$H(\omega) = \sum_{n=-\infty}^{\infty} h_n e^{-j\omega n}. \tag{6.20}$$

[1]Note that to be consistent in our notation, we should have written the frequency response as $H(e^{j\omega})$ rather than $H(\omega)$, but the latter is usually preferred in the literature as a shorthand notation for the former.

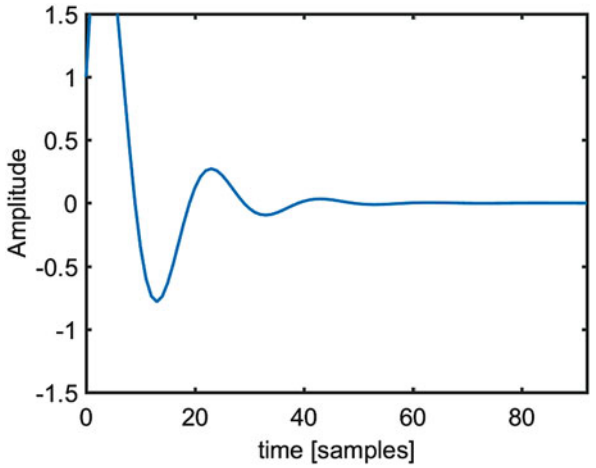


Fig. 6.4 Impulse response of filter having the transfer function $H(z) = 1/(1 - 1.2728z^{-1} + 0.81z^{-2})$

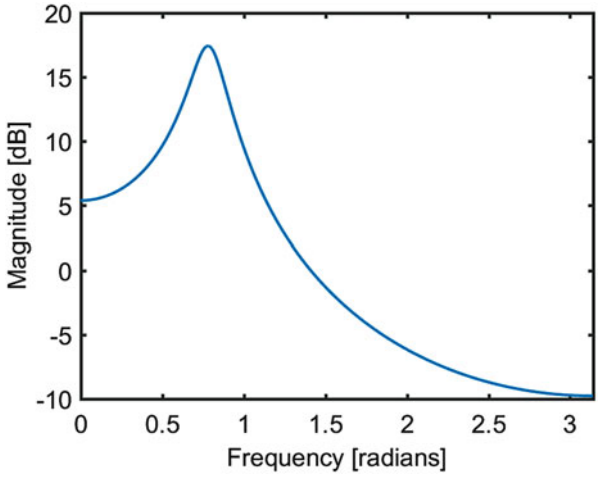


Fig. 6.5 Magnitude of the Fourier transform of the filter having the transfer function $H(z) = 1/(1 - 1.2728z^{-1} + 0.81z^{-2})$

In Fig. 6.4, the impulse response of a filter having the transfer function $H(z) = 1/(1 - 1.2728z^{-1} + 0.81z^{-2})$ is shown and in Fig. 6.5, the magnitude of the Fourier transform of the impulse response is shown. As can be seen, the filter is a resonator filter. From this we gather that the magnitude of the Fourier transform is equivalent to the magnitude response.

Next, consider the filter having the following difference equation having both a feedforward part and a feedback part:

$$\begin{aligned} y_n = b_0 x_n + b_1 x_{n-1} + \ldots + b_M x_{n-M} \\ + a_1 y_{n-1} + a_2 y_{n-2} + \ldots + a_K y_{n-K}. \end{aligned} \tag{6.21}$$

To find the transfer function from this difference equation, we take the z-transform as $x_n \to X(z)$ and $y_n \to Y(z)$ and use that $x_{n-1} \to z^{-1}X(z)$ (and similarly for y_n), whereby we obtain

$$\begin{aligned} Y(z) = b_0 X(z) + b_1 z^{-1} X(z) + \ldots + b_M z^{-M} X(z) \\ + a_1 z^{-1} Y(z) + a_2 z^{-2} Y(z) + \ldots + a_K z^K Y(z). \end{aligned} \tag{6.22}$$

Collecting the parts pertaining to $Y(z)$ on the left-hand side and the parts pertaining to $X(z)$ on the right, we get

$$Y(z)(1 - a_1 z^{-1} - \ldots - a_K z^{-K}) = X(z)(b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}). \tag{6.23}$$

We can now determine the transfer function of the filter as

$$H(z) = \frac{Y(z)}{X(z)} = \frac{b_0 + b_1 z^{-1} + \ldots + b_M z^{-M}}{1 - a_1 z^{-1} - a_2 z^{-2} - \ldots - a_K z^{-K}}. \tag{6.24}$$

As we can see, the transfer function of feedback filters also can be seen as rational functions, having non-trivial polynomials in the numerator and the denominator. The frequency response can be found as before by replacing z by $e^{j\omega}$ in $H(z)$. From the transfer function above let us define two polynomials, namely one for the numerator, which we will call $B(z)$ and is defined as

$$B(z) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots + b_M z^{-M}, \tag{6.25}$$

and one from the denominator, called $A(z)$, defined as

$$A(z) = 1 - a_1 z^{-1} - a_2 z^{-2} - \ldots - a_K z^{-K}. \tag{6.26}$$

The transfer function is thus $H(z) = B(z)/A(z)$. Note that we can easily convert both of the polynomials into something having positive exponents as before. The polynomial $B(z)$ is a Mth order polynomial while $A(z)$ is an Lth order one. As we know, polynomials have values for which they are equal to zero, called roots, i.e., values of z for which $A(z) = 0$ or $B(z) = 0$. The values for which

$$B(z) = 0 \tag{6.27}$$

are called the *zeros* of the transfer function $H(z)$. An Mth order polynomial has M zeros, according to the fundamental theorem of algebra, which are denoted as z_m for $m = 0, 1, \ldots, M$. These roots can be used to factor $B(z)$ as[2]

$$B(z) = (1 - z_1 z^{-1})(1 - z_2 z^{-1}) \cdots (1 - z_M z^{-1}). \tag{6.28}$$

Notice how the polynomial $B(z)$ is in the numerator of $H(z)$. Thus, if for some value z we have that $B(z) = 0$, then we also have that $H(z) = 0$. In other words, zeros cause the input to be attenuated near the zero and disappear completely if it falls on top of the zero. What does this means in terms of the input $X(z)$? It means that the contents of x_n in terms of the phasor representation near the zero will be attenuated. If the zero is on the unit circle, then it means that a phasor having a frequency corresponding to the angle of the zero will be cancelled.

Similarly, the values for which

$$A(z) = 0 \tag{6.29}$$

are called the *poles* of the transfer function, and they are denoted as p_k with $k = 0, 1, \ldots, K$. Just like $B(z)$, $A(z)$ can be factored into its roots, i.e., $A(z) = (1 - p_1 z^{-1})(1 - p_2 z^{-1}) \cdots (1 - p_K z^{-1})$. We could thus also express the transfer function as

$$H(z) = \frac{B(z)}{A(z)} = \frac{(1 - z_1 z^{-1})(1 - z_2 z^{-1}) \cdots (1 - z_M z^{-1})}{(1 - p_1 z^{-1})(1 - p_2 z^{-1}) \cdots (1 - p_K z^{-1})}. \tag{6.30}$$

We can thus think of the big filter in (6.21) as comprised of a number of small filters, corresponding to one (or a pair of) zeros and/or poles. We can also see that if a pole and a zero are located in exactly the same spot, i.e., $z_m = p_k$ for some m and k, they will cancel out.

We saw before that the effect of the zeros was to cancel or attenuate the contents of the input $X(z)$. What then is the effect of the poles? They amplify the signal contents near the poles. To see this, consider that $A(z)$ appears in the denominator of $H(z)$, i.e.,

$$H(z) = \frac{B(z)}{A(z)}. \tag{6.31}$$

Hence, for values of z near the poles where $A(z) = 0$ we are essentially dividing by zero or a very small value. This causes $|H(z)|$ to be become large and thus amplify $X(z)$.

From this we can gather that to design a filter that attenuates the signal contents at certain frequencies, we must place zeros near the unit circle at the corresponding angles. Similarly, to amplify the signal contents at a certain frequency, we must

[2]Note that the root of $(1 - z_k z^{-1})$ is also the root of $(z - z_k)$. We could also have factored $B(z)$ as $B(z) = (z - z_1)(z - z_2) \cdots (z - z_M) z^{-M}$.

place a pole there. Filter design can thus be thought of as the process of deciding where to put the zeros and poles of the transfer function to get some desired frequency response.

Regarding the poles, we must be a bit careful, however. As we saw in an earlier chapter, feedback can cause instability in a filter where the output grows without bound. Moreover, the placement of a pole in a certain region may also cause the system to become non-causal. It turns out that if all the poles are inside the unit circle, then the system is both stable and causal. We can write this condition as

$$|p_k| < 1 \quad \text{for} \quad k = 1, 2, \ldots, K. \tag{6.32}$$

Hence, to check whether a filter is stable and causal given a difference equation, we must form the polynomial $A(z)$ from the feedback coefficients, $\{a_k\}$, and find its roots, i.e., the values for which $A(z) = 0$. If these values all have a magnitude smaller than one, then the filter is stable and causal. As for the zeros, they do not affect stability and causality and can, hence, be placed anywhere in the z-plane without causing us problems. Transfer functions are often analyzed by plotting their zeros and poles in a so-called pole-zero plot that shows the z-plane and the unit circle. If we can see that all the poles are inside the unit circle, the filter is causal and stable.

In Fig. 6.6 an example of such a pole-zero plot is shown. The zeros are indicated by o's while the poles are indicated by x's. In this example, the zeros are located

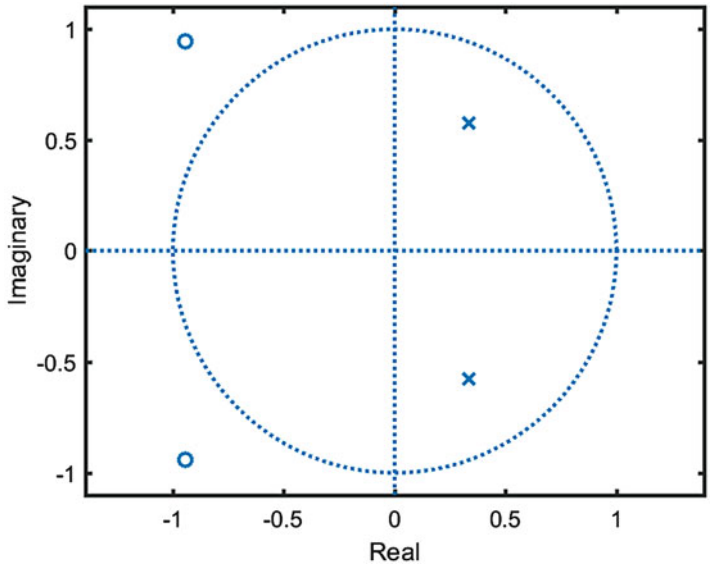


Fig. 6.6 Pole-zero plot of the transfer function $H(z) = (1 + 1.89z^{-1} + 1.78z^{-2})/(1 - 0.67z^{-1} + 0.44z^{-2})$

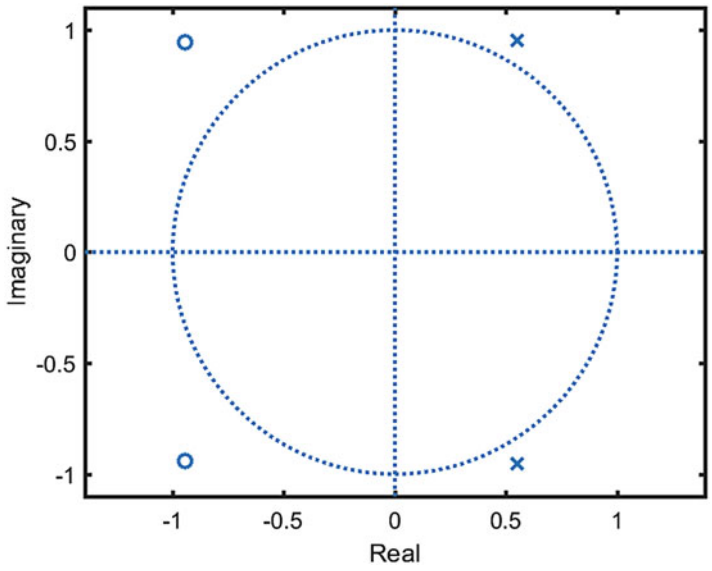


Fig. 6.7 Pole-zero plot of the transfer function $H(z) = (1 + 1.89z^{-1} + 1.78z^{-2})/(1 - 1.10z^{-1} + 1.21z^{-2})$

at $-0.9428 \pm j0.9428$ and the poles at $0.3333 \pm j0.5774$. This corresponds to a transfer function $H(z) = (1 + 1.89z^{-1} + 1.78z^{-2})/(1 - 0.67z^{-1} + 0.44z^{-2})$. Since the poles are inside the unit circle, the corresponding filter is stable. As mentioned before, the location of the zeros is not important for the stability of the filter. In the next example, which is shown in Fig. 6.7 the poles are located at $0.5500 \pm j0.9526$ while the zeros are at $-0.9428 \pm j0.9428$ as before. The transfer function is $H(z) = (1 + 1.89z^{-1} + 1.78z^{-2})/(1 - 1.10z^{-1} + 1.21z^{-2})$. In this case, the corresponding filter is unstable since the poles are outside the unit circle.

The concepts of stability and causality can also be understood in terms of the impulse response, h_n, of a system. Indeed, the causality condition means that $h_n = 0$ for $n < 0$, which is easy to check. The stability condition is, however, a bit more complicated. A necessary and sufficient condition is that the impulse response is absolutely summable, which means that we must have that $\sum_{n=-\infty}^{\infty} |h_n| < \infty$. Put loosely, we can think of this as the condition that the impulse response must decay.

For the case in (6.14), where we had only feedforward, there was a simple relationship between the transfer function and the impulse response, and the impulse response and the frequency response. When a filter also has feedback, as is the case for the filter in (6.21), these relationships are a bit more complicated. As a simple example, consider the filter having the transfer function with $|a| < 1$

$$H(z) = \frac{1}{1 - az^{-1}}, \tag{6.33}$$

which has one pole $p_1 = a$. By multiplying by z in the numerator and denominator and then using polynomial long-division, this can be expanded as

$$H(z) = \frac{z}{z-a} = 1 + az^{-1} + a^2z^{-2} + a^3z^{-3} + \dots. \tag{6.34}$$

As we can see, this results in an infinite series, but it no longer has a denominator. This shows that the filter can be thought of as a feedforward filter, but one that has an infinite number of terms in its transfer function, hence the name infinite impulse response filter. From this we can also deduce that the impulse response of the filter must be

$$h_n = a^n \quad \text{for} \quad n = 0, 1, 2, \dots, \tag{6.35}$$

which is the sequence

$$1, a, a^2, a^3, \dots. \tag{6.36}$$

In this case, a choice of $a > 1$ would yield an increasing impulse response over time (which clearly is not absolutely summable) and a pole outside the unit circle, so the theory checks out from both perspectives. As for the frequency response, it can, in principle, be obtained by substituting z by $e^{j\omega}$ either in the rational function (6.33) or in the infinite series (6.34). The latter corresponds to computing the Fourier transform of the impulse response, just as it did for the filter in (6.14), only that filter had a finite impulse response.

We have seen that we can think of an IIR filter, and filters in general, in different ways. The IIR filter having the difference equation

$$y_n = b_0x_n + b_1x_{n-1} + b_2x_{n-2} + a_1y_{n-1} + a_2y_{n-2}, \tag{6.37}$$

which involves both delayed versions of the input and the output, can also be written as

$$y_n = \sum_{m=0}^{\infty} h_m x_{n-m}, \tag{6.38}$$

where h_n is the impulse response of the filter in (6.37). As we can see, it only involves the input but an infinite number of delayed versions of it.

The curious reader might wonder what happens if we truncate the impulse response of an IIR filter above after certain value, say for $n > L$. We would then have an FIR filter that approximates the IIR filter, which means its frequency response would mimic that of the IIR filter, but not be exactly equal to it.

In its most general form, the operation in (6.38) is

$$y_n = \sum_{m=\infty}^{\infty} h_m x_{n-m}. \tag{6.39}$$

This operation is called *convolution*, and we say that the output is the convolution of x_n by h_n or that x_n is convolved by h_n, and it is often written in the literature as $*$, i.e.,

$$y_n = x_n * h_n = \sum_{m=\infty}^{\infty} h_m x_{n-m}. \tag{6.40}$$

Note that for causal systems, we have that $h_n = 0$ for $n < 0$ and (6.39) and (6.38) are equivalent. For FIR filters, the convolution operation reduces to $y_n = \sum_{m=0}^{M} h_m x_{n-m}$, involving only a finite number of terms. Note how the convolution and the difference equation are equivalent for FIR filters.

6.4 Properties of Filters

Next, we will explore some important properties of filters. These are relatively easy to see from z-transform of signals and transfer functions. In fact, it is much easier to derive some of these properties from the transfer function than the difference equations.

Now, let us return to the aforementioned properties of filters. The first such property is called *homogeneity* and means that if we scale the input of a filter x_n by a number, say a, before passing it through the filter then the result is that the output is simply scaled, i.e., if $Y(z) = H(z)X(z)$, then

$$(aX(z))H(z) = \alpha Y(z) \tag{6.41}$$

The property is illustrated in Fig. 6.8 showing the block diagrams of two filters that are equivalent. The second property is *additivity*. If we add two inputs, let us call them $x_n^{(1)} \to X^{(1)}(z)$ and $x_n^{(2)} \to X^{(2)}(z)$, and pass them through a filter having the transfer function $H(z)$, then the output is the same as passing the two inputs through the filters separately and adding the results, i.e., let $X(z) = X^{(1)}(z) + X^{(2)}(z)$ then

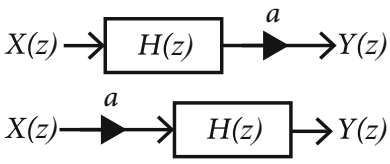


Fig. 6.8 The two systems are equivalent due to the homogeneity property

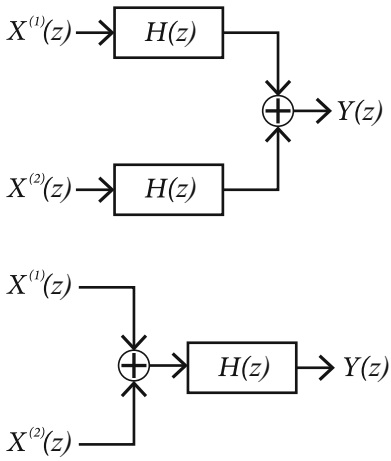


Fig. 6.9 Two equivalent ways of combining and filtering two signals, due to the additivity property of filters

$$Y(z) = X(z)H(z) \tag{6.42}$$
$$= \left(X^{(1)}(z) + X^{(2)}(z)\right) H(z) \tag{6.43}$$
$$= X^{(1)}(z)H(z) + X^{(2)}(z)H(z) \tag{6.44}$$
$$= Y^{(1)}(z) + Y^{(2)}(z), \tag{6.45}$$

where $Y^{(1)}(z) = X^{(1)}(z)H(z)$ and similarly for $Y^{(2)}(z)$. In Fig. 6.9, the additivity property is illustrated with two equivalent filters.

If we pass a signal $x_n \rightarrow X(z)$ through first one filter, having the transfer function $F(z)$, and then a second one, having the transfer function $G(z)$, then the result is

$$Y(z) = G(z)\left(F(z)X(z)\right) \tag{6.46}$$
$$= F(z)\left(G(z)X(z)\right) \tag{6.47}$$
$$= H(z)X(z) \tag{6.48}$$

which means (1) that we can combine the two filters into one by multiplying their transfer functions, i.e., $H(z) = F(z)G(z)$, and (2) that the ordering of the filters does not matter at all. This is illustrated in Fig. 6.10 which shows two equivalent systems. Audio effect aficionados frequently discuss what order audio effects should be applied in. Should the reverb be applied before or after the equalizer? Should it be applied before or after the delay effect? The answer is that as long as the effects are filters, it does not matter. To combine two filters $F(z)$ and $G(z)$ we must multiply their transfer functions. Recall that each transfer function can be put into the form of

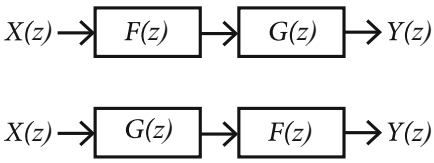


Fig. 6.10 The outputs are identical, as the order of the filters is not important

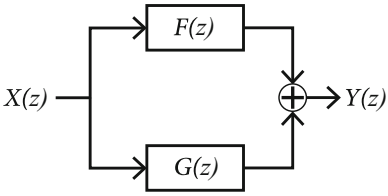


Fig. 6.11 Two filters connected in parallel. They can be seen as one combined filter by adding the transfer functions, which are rational functions

a rational function in (6.17). Then it follows that we must multiply the polynomials of the numerators and denominators of the transfer functions.

The last case is one where we feed the same input, $X(z)$, into two different filters, $G(z)$ and $F(z)$, and then add their outputs. This is shown in Fig. 6.11. We can write this as

$$Y(z) = X(z)F(z) + X(z)G(z) \tag{6.49}$$

$$= X(z)\left(F(z) + G(z)\right) \tag{6.50}$$

$$= X(z)H(z). \tag{6.51}$$

This shows that the two filters connect in parallel can be combined into one filter whose transfer function is simply the addition of the two transfer functions, $F(z)$ and $G(z)$, i.e., $H(z) = F(z) + G(z)$. Since the transfer functions can be thought of as rational functions, it follows that we must add these rational functions. For FIR filters this is rather simple, but it is a bit more complicated for IIR filters, since we must convert the rational functions to common denominators. Suppose the filter $F(z)$ and $G(z)$ are given by

$$F(z) = \frac{A(z)}{B(z)} \quad \text{and} \quad G(z) = \frac{C(z)}{D(z)}. \tag{6.52}$$

Then the two filters put in parallel yield

$$H(z) = F(z) + G(z) \tag{6.53}$$

$$= \frac{A(z)}{B(z)} + \frac{C(z)}{D(z)}. \tag{6.54}$$

Multiplying $F(z)$ by $D(z)$ in the numerator and in the denominator and $G(z)$ by $B(z)$, we obtain

$$H(z) = \frac{A(z)D(z)}{B(z)D(z)} + \frac{C(z)B(z)}{D(z)B(z)} \tag{6.55}$$

$$= \frac{A(z)D(z) + C(z)B(z)}{B(z)D(z)}. \tag{6.56}$$

6.5 Types of Filters

Traditional filter design in digital signal processing aimed at realizing analog filters with difference equations, but the digital signal processing and filter design have since transcended the old, analog filters, and now it is relatively easy to design and implement filters that would be very hard to realize with analog hardware. Standard analog filters were typically designed by first designing a prototype low-pass filter from tables of polynomials (like Chebyshev or Butterworth polynomials), which was then transformed into high-pass, band-pass, or whatever kind of filter was desired with pen and paper. Today, however, filter design is typically done with software such as MATLAB, and now convex optimization methods exist that will allow you to design filters almost any frequency responses you want and can optimize the filters with respect to various properties. The desired filter response is typically specified in terms of an ideal filter response, which may or may not be realizable. For example, a desired response might be that of a brickwall low-pass filter that transitions from a gain of 1 to 0 at a desired frequency, called the cutoff frequency. Such a filter cannot be realized but we can approximate it.

We will now go over some classical, simple filter types, some of which we have already encountered, and examples of their usage.

Low-pass A low-pass filter lets frequency beneath a cutoff frequency pass while it removes or attenuates frequencies above it. As we have seen, low-pass filters are an important part of a sampling and reconstruction, where they were called anti-aliasing and anti-image filters. They are also important in digital resampling of signals, i.e., sample-rate conversion (e.g., going from 44.1 to 48 kHz or vice versa).

High-pass High-pass filters remove the frequencies beneath the cutoff frequency while it preserves everything above it. In audio, they are often used to remove the bass of some instruments before mixing and is also called a low-cut filter.

Bandpass A bandpass filter passes the frequencies of the input signal between one cutoff frequency and another cutoff frequency and otherwise attenuates or removes the contents. It can be obtained by a combination of a low-pass filter and a high-pass filter in series, where the cutoff frequency of the high-pass filter

is lower than that of the low-pass filter. The difference between the two cutoff frequencies is called the bandwidth.

Stopband The stopband filter removes signal contents between two cutoff frequencies while leaving the rest unchanged. Like the bandpass filter, it can be obtained from a combination of a low-pass filter and a high-pass filter in series. More specifically, the cutoff frequency of the high-pass filter must now be higher than that of the low-pass filter to achieve the desired effect.

In Fig. 6.12, the desired (i.e., ideal) magnitude responses of the four basic filter types are shown. The frequency responses of filters are often described in terms of cutoff frequencies, stopband frequencies, center frequencies, passband ripple, stopband attenuation, stopband attenuation, bandwidth, transition bandwidth, and so forth. In Fig. 6.13, these properties of the frequency response of non-ideal filters are illustrated.

For more on different filter types and their properties, we refer the interested reader to [7].

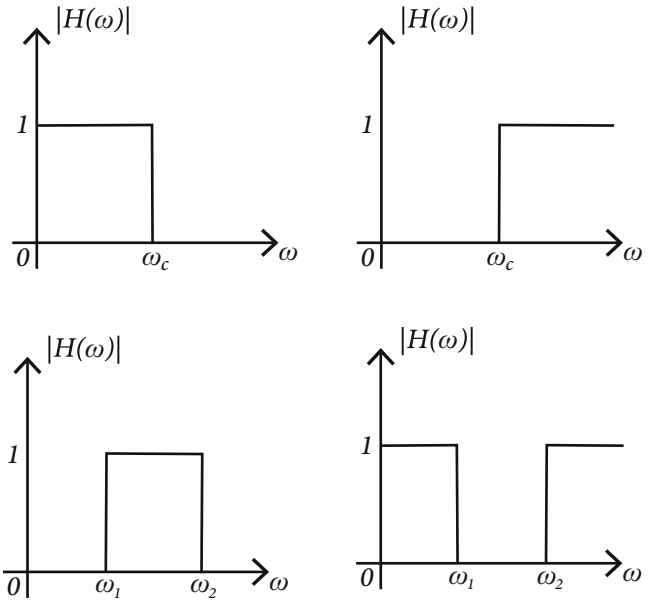


Fig. 6.12 Ideal magnitude responses of four basic filter types: low-pass (top left), high-pass (top right), bandpass (bottom left), and stopband (bottom right)

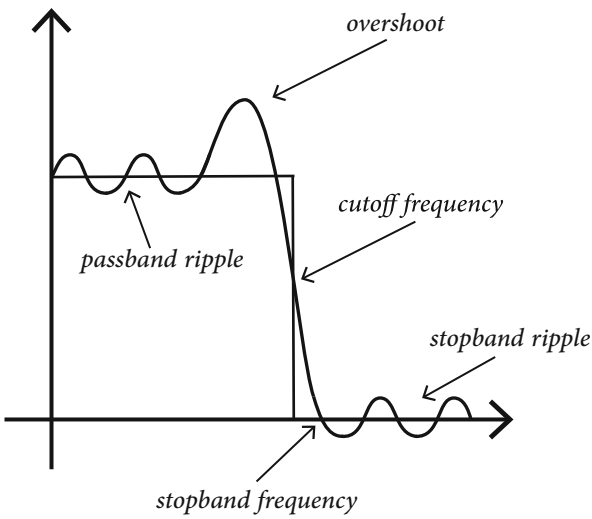


Fig. 6.13 Illustration of the magnitude response of non-ideal filters

6.6 Exercises

The exercises in this chapter aim at developing the intuition and understanding of filters and their properties by working with some simple filters in Pure Data.

Exercise 1 There are some built-in filters in Pure Data, including the `lop~`, `hip~`, and `bp~` objects, which implement low-pass, high-pass, and bandpass filters, respectively. Try feeding white noise through these filters for different parameters. Listen to the results.

Exercise 2 Build a patch in Pure Data for displaying the impulse response of a filter. You can generate an impulse with the `dirac~` object. You have to generate this impulse at regular intervals, say every 250 ms, which can be achieved with the `metro` object. Feed this impulse to a filter of your choice (like the comb filter from a previous exercise). Store the output in an array with `tabwrite~` and plot the result. Remember to trigger the `tabwrite~` object also with a `metro` object.

Exercise 3 Build a patch in Pure Data where the input signal is first fed through a low-pass filter (i.e., use `lop~`) with a cutoff frequency of 1000 Hz and then an high-pass filter (i.e., `hip~`) with a cutoff frequency of 500 Hz. Listen to the output when white noise is used as input. What is the resulting filter?

Exercise 4 Build a patch in Pure Data where the output is given by the input signal minus the input signal fed through a low-pass filter, using `lop~`, with a cutoff frequency of 2500 Hz. Listen to the output when white noise is used as input. What is the resulting filter?

Chapter 7
The Fourier Transform

7.1 Introduction

The Fourier transform is perhaps the most important transform of all, and certainly it is the most used one in audio processing. It is named after Jean-Baptiste Joseph Fourier who discovered what we now know as the Fourier series and thus laid the foundation of the Fourier transform (and modern function analysis in the process) while trying to solve the head equation for a metal plate! He published his results in 1822. It allows us to analyze and interpret the contents of audio signals, we can use it to analyze and understand filters, and how filters affect signals. The Fourier transform of a signal is often referred to as a spectrum. We can think of the Fourier transform as a decomposition of a signal into phasors of different frequencies, and the amplitude of the different phasors then reflects the amount of power or energy that is present in the signal at that frequency, similar to the spectrum of light. When a signal is transformed using the Fourier transform we say that it is transformed from the time domain to the frequency domain. It is often much easier to analyze the contents of audio signals in the frequency domain than in the time domain, unlike, for example, images. However, to do this, we must first understand the Fourier transform and its properties, and we must study how the Fourier transform looks for different kinds of signals that we are likely to encounter. The Fourier transform is a linear transform, which means that the Fourier transform of a sum of signals is equal to the sum of the Fourier transform of the individual parts. Thus, we can look at the Fourier transform of a complicated signal, like that of an orchestra playing a piece of music, and we can, for example, identify the parts corresponding to the different instruments, perhaps we can even recognize the instruments from their Fourier transform. The Fourier transform is also a good example of why complex numbers and algebra are convenient. The Fourier transform is fairly easily understood and applied to signals using complex numbers, while it would be quite cumbersome to

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_7

do the same with only real numbers. We will now proceed to introduce the Fourier transform and explore its properties and applications in detail. For a more in-depth treatment of the Fourier transform, we refer the interested reader to the classical textbook on signal processing [8].

7.2 The Fourier Transform

The Fourier transform is a tremendously important tool in understanding both signals and filters. We will now introduce it and explore its properties a bit. The Fourier transform of a signal x_n measured for $n = 0, \ldots, N-1$ is defined as

$$X(\omega) = \sum_{n=0}^{N-1} x_n e^{-j\omega n}. \tag{7.1}$$

$X(\omega)$ is said to be the Fourier transform of x_n and ω is here a continuous variable between 0 and 2π, so $X(\cdot)$ is a continuous function. The quantity $X(\omega)$ is also called the *spectrum* of the signal x_n. Recall that the digital frequency ω in radians per sample is related to physical frequencies f in Hertz as $\omega = 2\pi f/f_s$, where f_s is the sampling frequency. The Fourier transform can be interpreted as a decomposition of the signal into different frequency components. Thus, we can inspect $X(\omega)$ to learn what components, sinusoids in fact, we have in the signal—more about this later. As we alluded to earlier, practically any signal can be thought of as comprised of sinusoids having different frequencies.

The Fourier transform is an invertible transform, which means that via an inverse transform we can go back to the original signal from the spectrum. For this reason, $X(\omega)$ and x_n are often referred to as each other's duals, and $X(\omega)$ is called the frequency domain representation of the time-domain signal x_n. Similarly, x_n can be thought of as the time-domain representation of $X(\omega)$.

For a continuous frequency ω, the inverse Fourier transform is given by an integral. However, we can only compute a Fourier transform of the form (7.1) with pen and paper. In practice, the frequency variable is discretized, which also makes it simpler to compute the inverse. For a signal comprised of N samples, we must have at least N samples of ω as well, and these are usually distributed uniformly from 0 to 2π, i.e.,

$$\omega_k = 2\pi \frac{k}{N} \quad \text{for} \quad k = 0, \ldots, N-1. \tag{7.2}$$

Using this frequency grid, we can now express the Fourier transform as

$$X(\omega_k) = \sum_{n=0}^{N-1} x_n e^{-j\omega_k n} \quad \text{for} \quad k = 0, \ldots, N-1. \tag{7.3}$$

$X(\omega_k)$ for different ks is now a sequence rather than a function, just like x_n is a sequence of numbers. The expression in (7.3) is called the discrete Fourier transform (DFT). It is the same as (7.1) evaluated with the grid defined in (7.2). This is also said to be an N-point DFT. Given such a sequence, we can now reconstruct the signal x_n from its spectrum as

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X(\omega_k)e^{j\omega_k n} \quad \text{with} \quad \omega_k = 2\pi\frac{k}{N}, \tag{7.4}$$

which is then performed for $n = 0, 1, \ldots, N-1$. This is called the inverse Fourier transform or inverse DFT. This reconstructs x_n perfectly with the division by N ensuring that the energy of x_n is preserved. In light of our previous discussions about phasors and the Fourier series, (7.4) has an interesting interpretation. Inside the summation, we essentially have a phasor of frequency ω_k, and this phasor is multiplied (or scaled) by $\alpha_k = X(\omega_k)/N$, so we are essentially adding up phasors of different frequencies (indexed by k), which have been scaled by different complex numbers, i.e.,

$$x_n = \sum_{k=0}^{N-1} \underbrace{\alpha_k}_{\text{scaling}} \underbrace{e^{j\omega_k n}}_{\text{phasor}}. \tag{7.5}$$

This way of thinking about the Fourier transform and its inverse is illustrated in Fig. 7.1, which shows how x_n is reconstructed by adding a number of phasors scaled by different complex numbers. So, we can think of $X(\omega_k)$ for different k as the amount of that phasor that is in x_n. Since $X(\omega_k)$ is a complex number, it has a magnitude $|X(\omega_k)|$ and a phase $\angle X(\omega_k)$. Therefore, to understand the signal x_n we need only to look at $X(\omega_k)$ for different ω_k. $|X(\omega_k)|^2$ thus represents the power in the signal at that particular frequency.[1] $|X(\omega_k)|^2$ is referred to as the *power spectrum*, $|X(\omega_k)|$ as the *magnitude spectrum*, and $\angle X(\omega_k)$ as the *phase spectrum*.

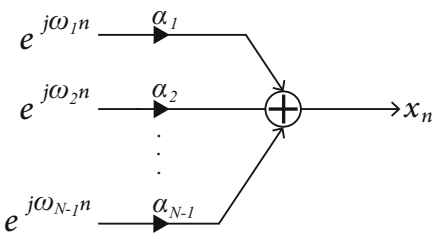


Fig. 7.1 The inverse discrete Fourier transform (DFT) as the reconstruction of the original signal x_n from a number of phasors

[1] Strictly speaking, for a continuous frequency variable, ω, $|X(\omega)|^2$ is a power density function and not a power function.

Much like power in general, the power spectrum is often measured in dB, i.e.,

$$P(\omega_k) = 20 \log_{10} |X(\omega_k)| \quad [\text{dB}]. \tag{7.6}$$

Regarding the earlier claims that any signal can be thought of as comprised of sinusoids, consider the following argument: any sequence of numbers in our signal x_n for $n = 0, 1, \ldots, N-1$ leads to a well-behaved Fourier transform computed using the DFT in (7.3) as long as the individual samples are finite. That the samples have to be finite is no issue, as we cannot really sample and quantize infinite signals anyway. Similarly, any spectrum $X(\omega_k)$ for $k = 0, 1, \ldots, N-1$ can be mapped back to its corresponding time-domain signal using (7.4). Hence, we can find the Fourier transform of any block of signal, and we can reconstruct any block of signal from its spectrum. Voìla!

The Fourier transform has some interesting properties that are good to know:

Linearity It is a linear transformation, which means two things: (1) that the Fourier transform of the sum of two signals, $x_n + y_n$, is just the sum of the individual Fourier transforms, i.e., $X(\omega) + Y(\omega)$, which is called the additivity property, and (2) that the Fourier transform of a signal x_n scaled by α, i.e., the signal is αx_n, is just the scaled Fourier transform of x_n, i.e., $\alpha X(\omega)$, which is called the homogeneity property. The additivity property is illustrated in Fig. 7.2.

Modulation Another interesting and important property is the so-called modulation property that the Fourier transform of a signal x_n multiplied by a phasor of frequency ω_0, i.e., $x_n e^{j\omega_0 n}$ is equal to $X(\omega) = X(\omega - \omega_0)$, which means that the spectrum of x_n is simply shifted by the modulation (i.e., the multiplication by the phasor). This is shown in Fig. 7.3.

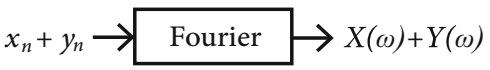


Fig. 7.2 The Fourier transform of the sum of two signals, x_n and y_n, is equal to the sum of their respective Fourier transform, $X(\omega)$ and $Y(\omega)$. This is due to the linearity of the Fourier transform

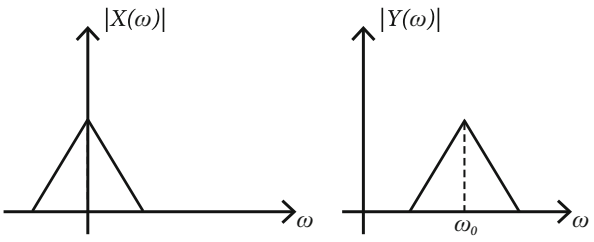


Fig. 7.3 The modulation property of the Fourier transform. The Fourier transform of a signal x_n multiplied by a phasor $x_n e^{j\omega_0 n}$ shifts the spectrum of x_n by ω_0

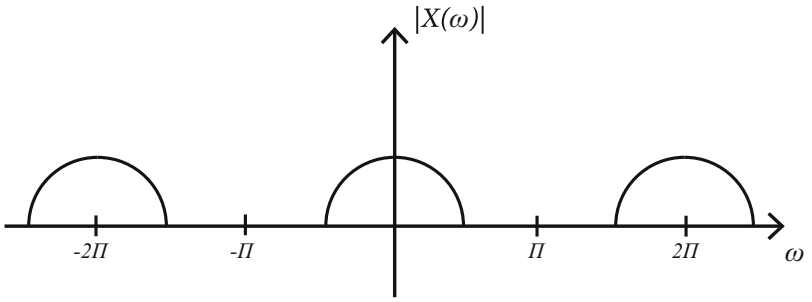


Fig. 7.4 The spectra of real signals are symmetric around zero, i.e., $X(\omega) = X^*(-\omega)$, and due to the sampling we also have that $X(\omega) = X^*(2\pi - \omega)$

Time-Shift Similarly, a time-shift of x_n by n_0, i.e., x_{n-n_0}, causes a change in the phase spectrum of x_n, namely, $X(\omega)e^{-j\omega n_0}$. We actually already knew this since a delay of a phasor simply causes a change in phase.

Parseval The energy of a signal can be computed from both the signal and its spectrum, i.e., $\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{N}\sum_{k=0}^{N-1} |X(\omega_k)|^2$ with $\omega_k = 2\pi\frac{k}{N}$. In fact, since the spectrum is a continuous function of the frequency, the energy can also be computed as the integral $\sum_{n=0}^{N-1} |x_n|^2 = \frac{1}{2\pi}\int_0^{2\pi} |X(\omega)|^2 d\omega$. This is called Parseval's formula.

Symmetry Finally, since audio signals are real, it follows that $X(\omega) = X^*(-\omega)$, which means that the spectrum is symmetric around 0. The symmetry can also be shown to imply that $X(\omega) = X^*(2\pi - \omega)$ for sampled signals, as shown in Fig. 7.4. As a consequence of this, the spectra of sounds are typically only shown for frequencies $0 \leq \omega \leq \pi$ or, equivalently, in Hertz $0 < f < f_s/2$.

7.3 The FFT and Zero-Padding

From (7.3), we can see that the DFT requires N complex multiplications and additions per frequency grid point and there are N such points, so it requires N^2 operations to compute the DFT. It is said that it has complexity[2] $\mathcal{O}(N^2)$. In many cases, however, this kind of complexity is prohibitive or at least it has been. The fast Fourier transform (FFT) is a fast implementation of the DFT, where a divide-and-conquer is employed recursively and breaks the problem into smaller and smaller problems, the solution to which are simple and are combined to form the solution to the bigger problems. It has complexity $\mathcal{O}(N \log_2 N)$ which is much faster than the direct implementation of the DFT for even moderately sized N. The most common

[2]Simply put, the notation $\mathcal{O}(N^2)$ means that the complexity is upper bounded by some (possibly unknown) constant times N^2.

implementations are radix-2 algorithms, which means that they work for N being a power of 2. Other kinds exist, but this is the most common one. The radix-2 FFT was (re-)invented by Cooley and Tukey[3] in 1965, and it has been referred to as one of the most important algorithms in the history of mankind. In signal processing, it made a lot of things possible all of a sudden that were previously impractical, and started the age of digital signal processing. One of the most commonly used FFT implementation today is the FFTW [9], which stands for Fastest Fourier Transform in the West!

In relation to the FFT and its various implementations, one might wonder what to do if we have N samples that we wish to compute the DFT of with a radix-2 algorithm, but N is not a power of two. We can use a trick called zero-padding. Suppose we have the signal x_n defined for $n = 0, \ldots, N-1$ and we wish to compute the F-point DFT of it. We can then simply augment the samples with a bunch of zeros (more specifically $F - N$ of them), i.e.,

$$\underbrace{x_0, x_1, \ldots, x_{N-1}}_{\text{signal}}, \underbrace{0, \ldots 0}_{\text{zero-padding}} . \tag{7.7}$$

This is also illustrated in Fig. 7.5. By observing that the Fourier transform is given by $\sum_{n=0}^{N-1} x_n e^{-j\omega n}$, we can see that adding a number of zeros to this summation does not change the spectrum of x_n. Hence, we can add as many zeros as we like. The result is that any time we wish to compute an F-point DFT of an N-sample signal where $N < F$, we can simply use zero-padding. Aside from the practical issue of a radix-2 algorithm needing a certain number of samples, there is another reason we would like to compute DFTs where $F \gg N$, namely, that the underlying continuous spectrum $X(\omega)$ is sampled on a finer grid this way! Thus, we get a clearer picture of

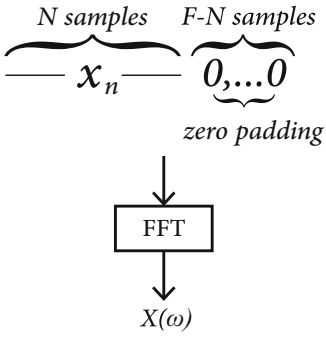


Fig. 7.5 The process of zero-padding the data $x(n)$, which has N samples, by augmenting it with $F - N$ zeros to obtain F samples, from which the F-point Fourier transform can be computed

[3] Carl Friedrich Gauss famously invented the tricks that make the FFT fast already in 1805 while doing data fitting, with pen and paper, of planetary observations but never published the results himself.

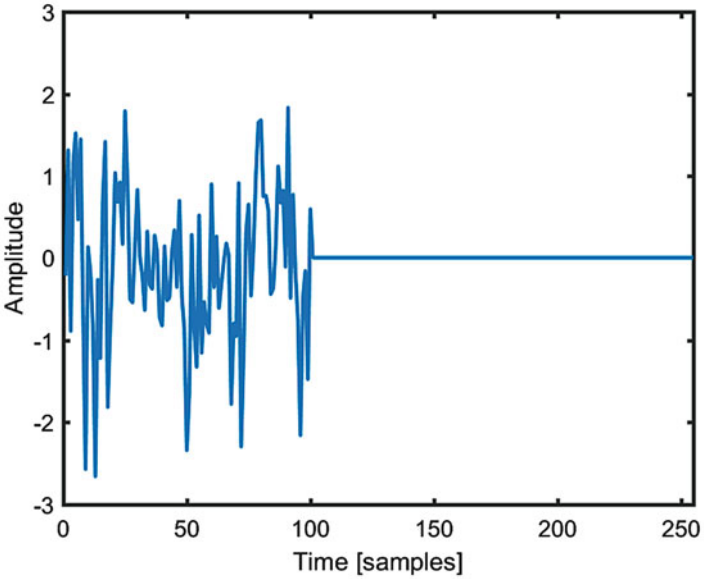


Fig. 7.6 Example of zero-padding. The original signal x_n has a length of $N = 100$ and zero-padding is applied to obtain a signal of length 256

what $X(\omega)$ really looks like. In Fig. 7.6 the effect of zero-padding is illustrated for a signal whose original length is 100 samples. It is zero-padded to length 256 which is 2^8.

7.4 Windowing

Windowing is the operation of multiplying the input signal, x_n, by a so-called window function, which we will call w_n, i.e.,

$$y_n = w_n x_n. \tag{7.8}$$

This begs the question why we would even do such a thing. To answer this, consider the following. We can think of a signal as having infinite duration, i.e., x_n goes from $n = -\infty$ to ∞ and we can think of the signal x_n for $0 \leq n \leq N-1$ as extracted from that signal by applying a window function as

$$w_n x_n \quad \text{for} \quad n = -\infty, \ldots, \infty, \tag{7.9}$$

where

$$w_n = \begin{cases} 1 & \text{for} \quad n = 0, 1, \ldots, N-1 \\ 0 & \text{otherwise.} \end{cases} \tag{7.10}$$

The window analogy is that an infinite signal is passing by our house but we only observe part of it through a window. Thus, we can understand the effect of going from an infinite signal to a finite one as windowing. The window defined in (7.10) is called the rectangular window. Other types of window functions exist but they are all only non-zero for $0 \leq n < N$. An infinite phasor of frequency ω_0 defined as

$$z_n = e^{j\omega_0 n} \quad \text{for} \quad n = -\infty, \ldots, \infty \tag{7.11}$$

has the Fourier transform

$$Z(\omega) = \sum_{n=-\infty}^{\infty} e^{j\omega_0 n} e^{-j\omega n} = \sum_{n=-\infty}^{\infty} e^{j(\omega_0 - \omega)n}. \tag{7.12}$$

This can be shown to be

$$Z(\omega) = \begin{cases} \infty & \text{for} \quad \omega = \omega_0 \\ 0 & \text{otherwise} \end{cases}. \tag{7.13}$$

It is relatively easy to see that for $\omega = \omega_0$ the Fourier transform is just $\sum_{n=-\infty}^{\infty} 1 = \infty$. For $\omega_0 \neq \omega$ it is a bit more complicated to see what the result is and we will not go into further details about this here. From the above, we can see that the spectrum of a phasor having an infinite duration is a very large value, infinite in fact, for the frequency of the phasor while it is zero everywhere else. If we now apply the rectangular window to the phasor and define $x_n = w_n z_n$, the Fourier transform of the signal is now given by

$$X(\omega) = \sum_{n=-\infty}^{\infty} \left(w_n e^{j\omega_0 n} \right) e^{-j\omega n} \tag{7.14}$$

$$= \sum_{n=0}^{N-1} e^{j(\omega_0 - \omega)n} = \frac{1 - e^{j(\omega_0 - \omega)N}}{1 - e^{j(\omega_0 - \omega)}}. \tag{7.15}$$

Meanwhile, the Fourier transform of the rectangular window defined in (7.10) is given by

$$W(\omega) = \sum_{n=0}^{N-1} e^{-j\omega n} = \frac{1 - e^{-j\omega N}}{1 - e^{-j\omega}}. \tag{7.16}$$

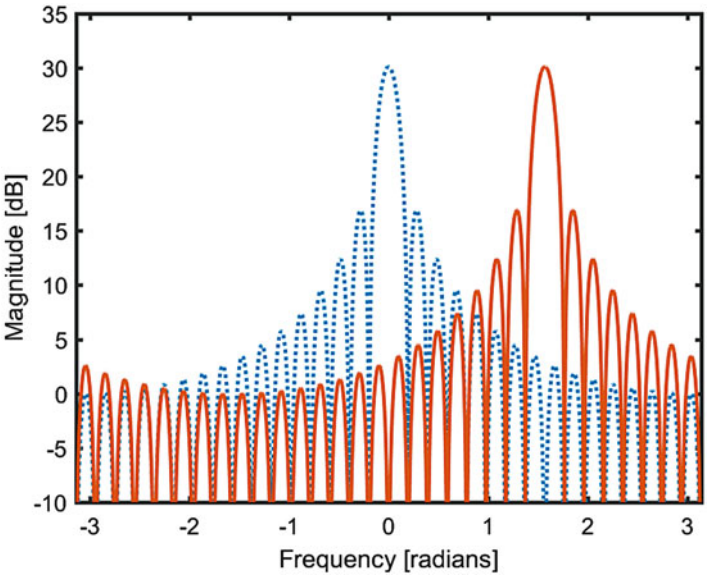


Fig. 7.7 Fourier transform of a window function (dotted) and the window applied to a phasor of frequency $\omega_0 = \pi/2$ (solid)

From this we can see that $X(\omega) = W(\omega - \omega_0)$, which means that the spectrum of the phasor with the window applied is the same as the spectrum of the window, only shifted by ω_0. This observation confirms the modulation property of the Fourier transform and applies regardless of what the window looks like, i.e., not just for the rectangular window. The property is illustrated in Fig. 7.7 which shows the Fourier transform for the rectangular window and a phasor of infinite duration having frequency $\omega_0 = \pi/2$ with the window applied. As can be seen, the Fourier transform of the window is simply shifted to the frequency of the phasor.

The above also means that by applying the rectangular window to the infinite phasor where we would previously see an infinitely large value in the spectrum at ω_0 and zero elsewhere, we now see that spectrum of the window, which still has its maximum value at ω_0, but is not zero everywhere else. While the peak in the infinite phasor's spectrum was infinitely narrow, there is now a distinct width to the peak. The main part of the window's Fourier transform, centered at ω_0, is called the mainlobe of the window, while the parts further away from ω_0 are called the sidelobes. The problem is now this: When we are analyzing the spectrum of a sound, how do we know if there are actually signal contents at a certain frequency or whether we are just looking at the sidelobe of something else? The phenomenon that the signal contents appear at different frequencies than it originally was is called leakage. Also, can we tell two, or more for that matter, phasors apart in the spectrum? From the linearity of the Fourier transform, we can also deduce that if the signal was comprised of two different phasors having

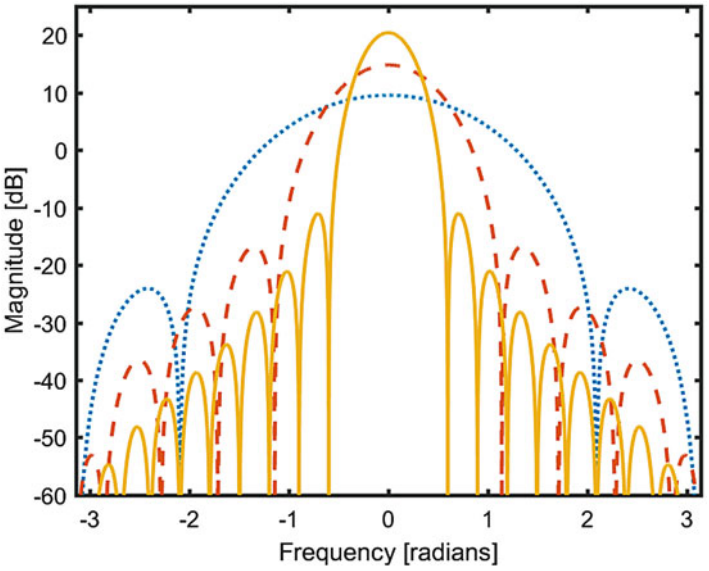


Fig. 7.8 Fourier transform of a window function for different N. Shown are $N = 20$ (solid), $N = 10$ (dashed), and $N = 5$ (dotted)

frequencies ω_1 and ω_2 with the same window on them, then the spectrum would now contain $W(\omega - \omega_1) + W(\omega - \omega_2)$. So, if the mainlobes of the two phasors overlap significantly, they will blend into one in the spectrum. Our ability to analyze spectra thus depends on the shape and the width of the mainlobe and the characteristics of the sidelobes, in particular how much below the mainlobe they are. The width of the mainlobe is controlled by the number of samples, N, and the characteristics of the window, and the higher the N is, the narrower the mainlobe we will get. Thus, to clearly separate phasors in the spectrum having close frequencies, we require a high N. Note that this width is not changed by applying zero-padding, as the effect of zero-padding is only to sample the same spectrum more densely. In Fig. 7.8, the Fourier transform of a common window is shown for three different N. As can be seen, the higher the N, the more narrow the mainlobe and the smaller the sidelobes.

It is possible to design windows that are different from the rectangular window and have different spectra, and the same general observations from the rectangular window still apply. The effect of applying various windows to a phasor is the same, i.e., the spectrum of the window will appear shifted by the frequency of the phasor. However, windows can be designed that have more narrow mainlobes or better attenuation of the sidelobes (or no sidelobes at all), which may make it easier to analyze a given spectrum and separate phasors. There now exists an abundance of different windows having different properties. Some examples of classical windows are the Hamming window, the Hanning window (which should really be called

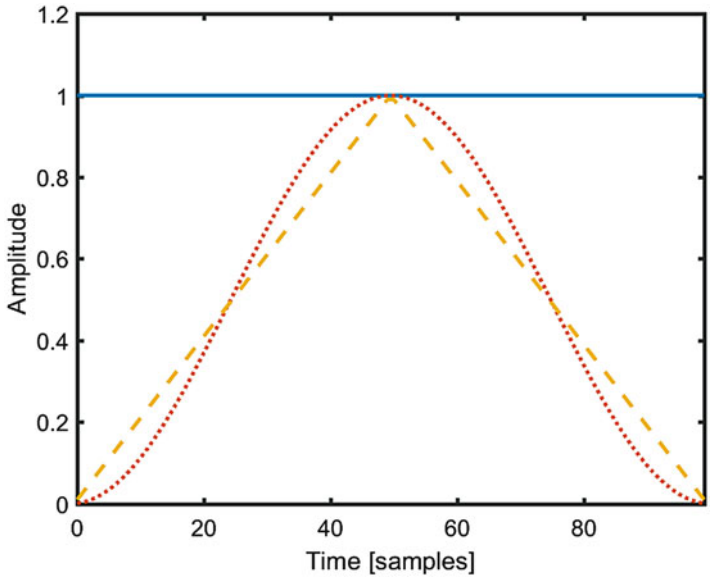


Fig. 7.9 Examples of window functions for $N = 100$. Shown are the rectangular window (solid), the triangular window (dashed), and the Hanning window (dotted)

the von Hann window), the triangular window, and the Gaussian window.[4] As an example of what such windows may look like, consider the Hanning window defined for $n = 0, \ldots, N-1$ as

$$w_n = \frac{1}{2}\left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right). \tag{7.17}$$

As can be seen, it is just an offset slowly varying cosine function. In Fig. 7.9 three examples of window functions are shown for $N = 100$, namely, the rectangular, triangular, and Hanning windows. In Fig. 7.10 their Fourier transforms are shown, only this time for $N = 9$. As can be seen, the three windows have quite different characteristics in terms of how they look in the time and frequency domains. The rectangular window has a narrow mainlobe but high sidelobes. The triangular and Hanning windows have wider mainlobes but their sidelobes are lower than those of the rectangular window. Depending on the application, different windows may thus be preferable. Note that the window whose Fourier transform is shown in Fig. 7.8 for different N is also a Hanning window.

[4] If you want to be famous in signal processing, it appears you just have to come up with a clever window!

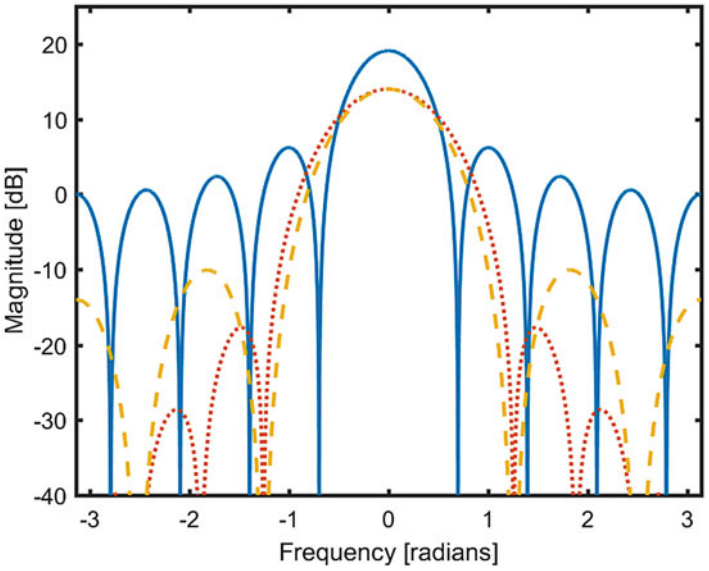


Fig. 7.10 Fourier transforms of some common window functions for $N = 9$. Shown are the rectangular window (solid), the triangular window (dashed), and the Hanning window (dotted)

For a signal x_n whose spectrum is $X(\omega)$, the effect of a window w_n which is only non-zero for $0 \leq n \leq N-1$ is

$$\sum_{n=0}^{N-1} (w_n x_n)\, e^{-j\omega n} = \frac{1}{2\pi} \int_{-\pi}^{\pi} X(\omega') W(\omega - \omega') d\omega', \tag{7.18}$$

which is the continuous equivalent of the discrete convolution operation we have used before. Thus, the effect of applying the window is equivalent to a filtering of the spectrum of x_n by the Fourier transform of the window! The commonly used windows are slowly varying and can be thought of as having a "low-pass filter" effect on the spectrum, which means that they smear the energy over the spectrum. As we have seen, this is unavoidable when we only observe a part of the signal.

7.5 Filtering and the Fourier Transform

The Fourier transform is important not only for understanding what filters are and what they do but also for implementing filters in practice. We will now explore the relation between filtering and the Fourier transform a bit more. Suppose we have an input signal x_n and a filter having the impulse response h_n. For simplicity in the following derivations, we will assume that the signals go from $-\infty$ to ∞. The

spectrum of x_n is then given by

$$X(\omega) = \sum_{n=-\infty}^{\infty} x_n e^{-j\omega n}. \tag{7.19}$$

Similarly, the frequency response of the filter is

$$H(\omega) = \sum_{n=-\infty}^{\infty} h_n e^{-j\omega n}. \tag{7.20}$$

We now convolve the impulse response of the filter with the input to obtain the output y_n, i.e.,

$$y_n = \sum_{m=-\infty}^{\infty} h_m x_{n-m}. \tag{7.21}$$

The question is now how the spectrum of y_n is related to the spectrum of x_n and the frequency response of the filter. To answer this question, we take the Fourier transform of (7.21), which is given by $Y(\omega) = \sum_{n=-\infty}^{\infty} y_n e^{-j\omega n}$, and insert (7.21) into this, whereby we obtain

$$Y(\omega) = \sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h_m x_{n-m} e^{-j\omega n}. \tag{7.22}$$

Noting that only x_{n-m} depends on n, we can rewrite this as

$$\sum_{n=-\infty}^{\infty} \sum_{m=-\infty}^{\infty} h_m x_{n-m} e^{-j\omega n} = \sum_{m=-\infty}^{\infty} h_m \left(\sum_{n=-\infty}^{\infty} x_{n-m} e^{-j\omega n} \right). \tag{7.23}$$

With a substitution of $n - m$ by n', the expression inside the parenthesis can be written as $\sum_{n=-\infty}^{\infty} x_{n-m} e^{-j\omega n} = \sum_{n'=-\infty}^{\infty} x_{n'} e^{-j\omega(n'+m)}$. Substituting this back into (7.23), we finally obtain

$$\sum_{m=-\infty}^{\infty} h_m \left(\sum_{n'=-\infty}^{\infty} x_{n'} e^{-j\omega(n'+m)} \right) = \underbrace{\left(\sum_{m=-\infty}^{\infty} h_m e^{-j\omega m} \right)}_{H(\omega)} \underbrace{\left(\sum_{n'=-\infty}^{\infty} x_{n'} e^{-j\omega n'} \right)}_{X(\omega)}. \tag{7.24}$$

From this, we can see that the expression now involves two Fourier transforms, one of h_n and one of x_n. This means that the spectrum of y_n is given by the multiplication of the spectrum of x_n by the frequency response of the filter (Fig. 7.11), i.e.,

$$Y(\omega) = X(\omega)H(\omega). \tag{7.25}$$

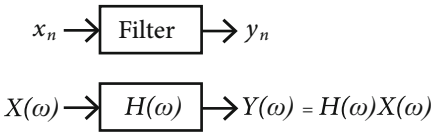


Fig. 7.11 Filtering in the time domain is equivalent to multiplication in the frequency domain

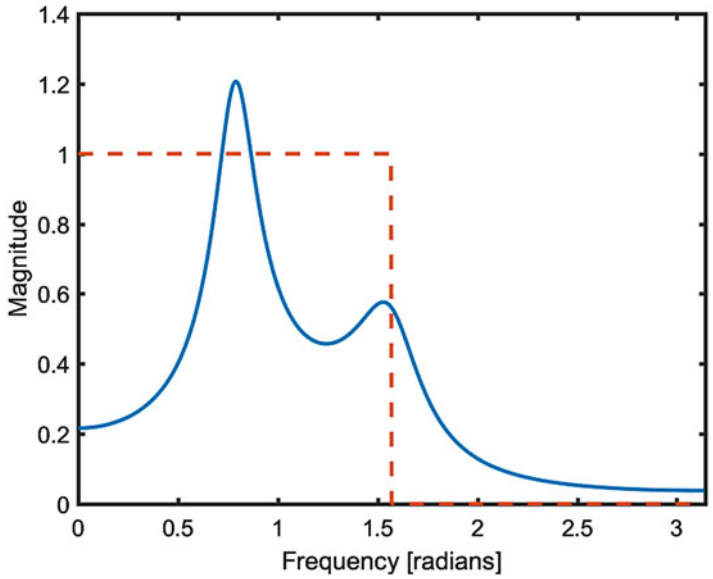


Fig. 7.12 Magnitude response of an ideal filter (dashed) and Fourier transform of input signal (solid). The Fourier transform of the output of the filter is the response of the filter multiplied by the Fourier transform of the input

From this we can see how a signal is affected by a filter. The signal contents at frequency ω are simply multiplied by the frequency response of the filter at ω. Thinking back, we actually already knew this! Earlier, we tested filters by passing a phasor through it. The amplitude and the phase of the phasor were then changed by the filter, corresponding to a multiplication by a complex number. Since we can think of the Fourier transform as a decomposition of signals into phasors, we can surmise that the individual phasors will have their phases and amplitudes changed by filters. It terms of the magnitude (or power spectrum for that matter), we can also deduce from (7.25) that $|Y(\omega)| = |X(\omega)||H(\omega)|$, which means that the magnitude of the spectrum of the input signal is simply multiplied by the magnitude response of the filter. In decibel, i.e., on a logarithmic scale, the effect of the filter can be found by inserting (7.25) into (7.6), which yields

$$20\log_{10}|Y(\omega)| = 20\log_{10}|X(\omega)| + 20\log_{10}|H(\omega)|. \tag{7.26}$$

The effect of filtering is illustrated in Fig. 7.12 showing the magnitude response of a filter, in this case an ideal brickwall low-pass filter, and the magnitude of the

Fourier transform of the input signal. The magnitude spectrum of output is then the magnitude spectrum of the input multiplied by the magnitude response of the filter, which in this case will remove all frequencies above $\pi/2$.

Not only does (7.25) tell us how the spectrum of a signal is affected by a filter, it actually also provides us with a way of implementing filters. Instead of implementing the finite convolution with an FIR filter having coefficients b_m for $m = 0, 1, \ldots, N-1$ directly as

$$y_n = \sum_{m=0}^{N-1} b_m x_{n-m}, \tag{7.27}$$

which requires N additions and multiplications per n (i.e., the complexity is $\mathcal{O}(N^2)$ if we also have N samples of output to compute), we can compute the N-point discrete Fourier transform of x_n and b_m with the FFT and simply multiply the spectrum of x_n by the frequency response of the filter as in (7.25). After multiplication, we simply have to compute the inverse Fourier transform of the result to obtain the output signal y_n. As mentioned earlier, the complexity of the FFT is $\mathcal{O}(N \log_2 N)$, so instead of complexity $\mathcal{O}(N^2)$, we can perform the same computations with two Fourier transforms of complexity $\mathcal{O}(N \log_2 N)$, a point-wise multiplication, and an inverse FFT, which also has complexity $\mathcal{O}(N \log_2 N)$. This is faster for long filter lengths and is called *fast convolution*, and is often used in practice for long FIR filters.

It should be noted that there are some things to be aware of when using fast convolution. Since filtering involves memory of past samples, it follows that we cannot simply process a segment of, say, N samples and ignore past samples. Indeed, by looking at (7.27), we can see that at time $n = 0$, the convolution involves past samples $x_{-1}, x_{-2}, \ldots$. We can also see that even for $n > N-1$, the filter can actually continue to produce output due to x_n for $n = 0, 1, \ldots, N-1$ even when no new input is available. This shows that the outlined procedure is not identical to convolution when executed this way. Indeed, the aforementioned fast convolution procedure is called *circular convolution*, while (7.27) is called *linear convolution*.

Circular convolution can be applied for segments of length N to produce linear convolution by the following procedure. It is performed for a filter having an impulse response h_m of length $M \leq N$ with an FFT size of $F = N + M - 1$, which means that zero-padding is applied . First, the frequency response $H(\omega_f)$ of the filter is computed for $\omega_f = 2\pi f/F$ with $f = 0, 1, \ldots, F-1$. Then, the following is performed for every input segment, x_n, comprised of N samples:

1. Compute the F-point FFTs with zero-padding of x_n to obtain $X(\omega_f)$.
2. Multiply $X(\omega_f)$ and $H(\omega_f)$ for $f = 0, 1, \ldots, F-1$ to obtain $\tilde{Y}(\omega_f) = X(\omega_f)H(\omega_f)$.
3. Compute the F-point inverse FFT of $\tilde{Y}(\omega_f)$ to obtain $\tilde{y}_n$ for $n = 0, 1, \ldots, F-1$, as shown in Fig. 7.13.

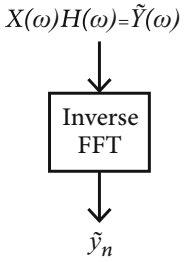


Fig. 7.13 The output $\tilde{y}_n$ is obtained from the multiplication of the Fourier transform of the input, x_n, and the impulse response h_n, which corresponds to circular convolution. To obtain the output corresponding to a linear convolution, the $M-1$ last samples of the previous segment must be added to the first $M-1$ samples of $\tilde{y}_n$

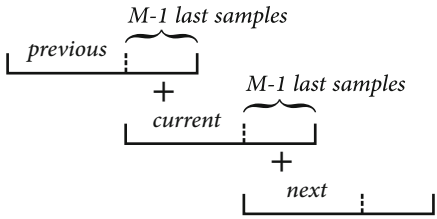


Fig. 7.14 The process of overlap-add to obtain linear convolution via multiplication in the frequency domain using the FFT. To compute the output of the filtering for the current segment, the $M-1$ last output samples of the previous samples must be added to the $M-1$ first output samples of the current segment

4. Add the last $M-1$ samples of the previous segment to the $M-1$ first samples of $\tilde{y}_n$ to obtain y_n for $n=0,1,\ldots,N-1$. Similarly, store the last $M-1$ samples of $\hat{y}_n$ to be added to the first samples of the next segment, as illustrated in Fig. 7.14.

This is called the *overlap-add* procedure. Note that for a time-invariant filter, $H(\omega_f)$ only has to be computed once and not every segment, but if the filter changes, then the frequency response has to be re-computed.

7.6 Spectrograms and Signal Examples

One of the most common ways of analyzing and depicting audio signals is using the *spectrogram*. The spectrogram is a two-dimensional representation, an image, of the contents of audio signals that aims at conveying information about the spectrum and how it behaves over time. On one axis you have time and on the other you have frequency. The amount of signal contents, i.e., the power, is indicated by some color coding. For example, high power could be red, while low power could be blue and grades of yellow in between. The Fourier transform is computed for blocks of audio

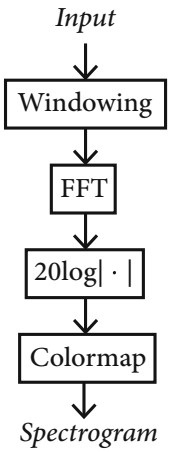


Fig. 7.15 The process of computing the spectrogram of a signal. The signal is segmented into overlapping segments, a window is then applied, and the signal is zero-padded to obtain the desired length. Then, the FFT is computed and the magnitude of the result is first converted to dB and mapped to a color

and typically zero-padding is used to obtain a high-resolution image of the signal contents. The signal's frequency contents have to stay the same for the duration of the blocks of audio. A signal whose statistics (e.g., the spectrum) is constant over some interval is said to be stationary and a signal that changes abruptly is said to be non-stationary or transient. Audio signals are typically stationary for blocks of 20–30 ms, so that is the block size that is typically used when computing spectra. With a sampling frequency of 44.1 kHz, 30 ms corresponds to 1323 samples and these could then be zero-padded to obtain a block of 8192 which is 2^{13}, so we can use radix-2 FFT. The spectrum is then computed for blocks of 1323 samples using the FFT, sometimes with overlap between adjacent blocks. Typically, a window other than the rectangular is also used. The power is then computed from the FFT of each block and converted to dB using (7.6) and the power is then mapped to a color using a colormap. This process is illustrated in Fig. 7.15.

To understand what a spectrogram is and how to interpret it, let us have a look at a relative simple example. In Fig. 7.16, the spectrogram is shown for a glockenspiel signal, i.e., a signal which is a recording of the sound produced by a glockenspiel, a musical instrument which is similar to the more well-known xylophone, only the bars of a glockenspiel are made from metal. Only the lower frequencies are shown here for visual clarity. The power in the signal is indicated with colors ranging from blue (the lowest) over yellow to red (the highest). As discussed earlier, the spectrum of a phasor is a vertical line. However, since the frequency axis is here the y-axis, a phasor will show up as high power concentrated in at a single frequency. If the phasor persists over time, across several blocks, it will show up as a horizontal line. In Fig. 7.16, we can see many such horizontal lines. This means that the glockenspiel produces sinusoids! We can see that the horizontal lines tend to endure for a long time, indicating that the notes sustain for quite some time. We can also see that the different horizontal lines have different starting and stop points. The starting points indicate when the different bars have been hit. It can also be seen that the sinusoids

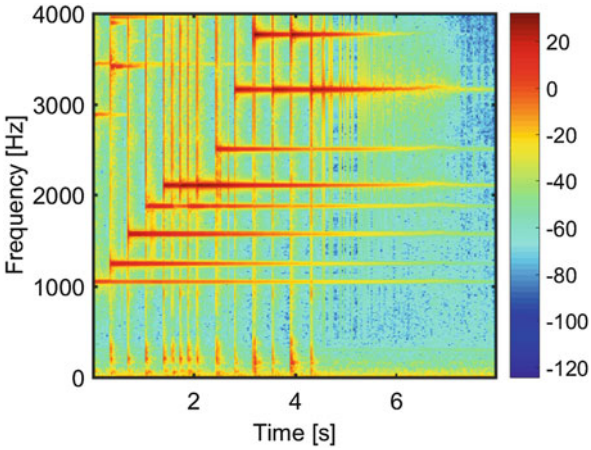


Fig. 7.16 Spectrogram of an audio signal, in this case produced by a glockenspiel

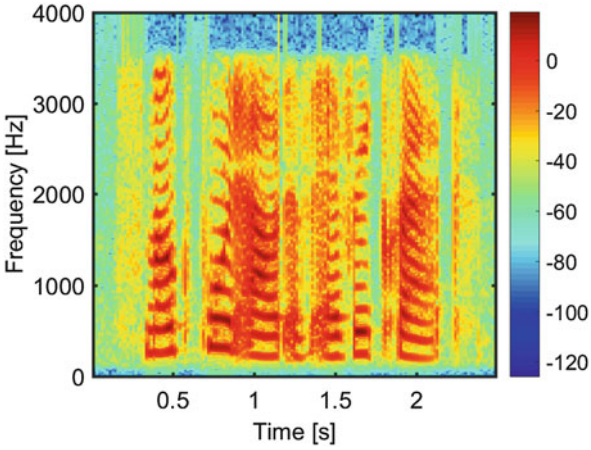


Fig. 7.17 Spectrogram of an audio signal, the female utterance "smoke poured out of every crack"

die out slowly, i.e., they decay. We can also observe other lines in the spectrogram, namely, vertical lines. Vertical lines mean that the energy of a signal is concentrated in time, i.e., we are dealing with something of a short duration, and spread out across a continuum of frequencies. These indicate the onsets of the notes when the metal bars of the glockenspiel have been struck by the mallets. The figure shows that we can actually see directly from the spectrogram which notes have been played on the glockenspiel and when!

Let us have a look at a more complicated example. The spectrogram of a speech signal is shown in Fig. 7.17. The signal is a recording of a natural speech produced by a female. She says "smoke poured out of every crack." In many ways, this is more complicated than the glockenspiel signal. First of all, the signal can be observed to be changing constantly, meaning that the signal is not very stationary. This is characteristic of speech. We can see that the spectrogram does appear to contain some horizontal, parallel lines. The parallel lines indicated that at a particular time

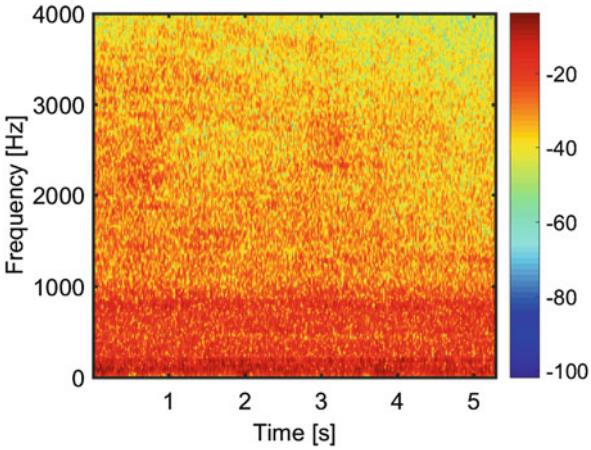


Fig. 7.18 Spectrogram of an audio signal, noise recorded inside a car cabin

instance, for example, around 2 s, the signal is approximately periodic. These parts are called voiced speech, while the other parts are called unvoiced. The distance between the lines is then the fundamental frequency, which is also (in this case) the frequency of the lowest high-energy horizontal line. It can also be seen that the fundamental frequency is constantly changing throughout the signal. Moreover, it is not always present. We can also observe something akin to a vertical line at 1.8 s and again at 2.3, which indicates some kind of transient. For speech, this kind of sound is called a plosive. At around 0.25 s, another type of sound can be seen. It does not have a clearly defined horizontal structure. Rather its energy seems to be spread out over a continuum of frequencies. Sounds having such characteristics are noise-like, i.e., stochastic in nature, and are called fricatives for speech.

The next example is noise recorded in a driving car. Its spectrogram is shown in Fig. 7.18. The characteristics of the noise can be seen to be similar to those of the fricative of speech, only the energy is distributed differently in frequency. The spectrum can be seen to contain energy spread out over the entire frequency range with most energy at low frequencies. This is characteristic of the engine noise in a car cabin. The spectrum can also be seen to vary very little over time, so this kind of signal is fairly stationary.

In Fig. 7.19, the spectrogram of a trumpet signal is shown. The trumpet is playing an ascending series of notes. The individual notes can be identified from the sets of parallel, horizontal lines. It can be seen that the structure of the spectrum is very regular and much more stationary than the voiced speech in Fig. 7.17. The frequency of the notes can be identified from the spacing between the individual harmonics or overtones at a particular time. It can also be seen that the notes appear to be sustaining while the next note is playing. This is due to the reverberation of the room in which the sound was recorded.

As we have seen, the spectrograms are quite useful when trying to understand the contents of signals. They are also useful for analyzing the effect of filters. One way to understand the effect of a filter is to compare the spectrogram before and

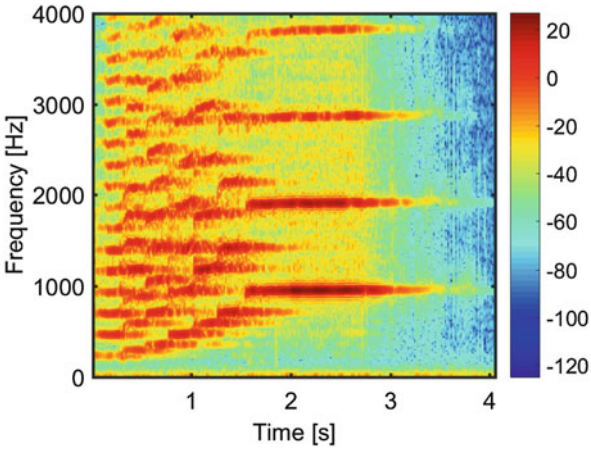


Fig. 7.19 Spectrogram of an audio signal produced by a trumpet

after filtering. As we have seen, the effect of a filter is that the spectrum of the input signal is multiplied by the frequency response of the filter. We have also seen how there exist different types of signals. For example, sinusoids appear as horizontal lines in the spectrogram. Transients, such as the sound of a hammer hitting a metal bar, appear as vertical lines, while noise-like sounds tend to have energy spread out in a smooth way over the spectrum.

In closing, it should be mentioned that spectral analysis is an entire research field in its own right with many sub-fields. Its applications extend into many different disciplines (chemistry, medicine, astronomy, etc.), way beyond audio processing, and entire careers have been devoted to it. Here, we could only give but a glimpse of this fascinating topic. For more on this subject, we refer the interested reader to the following excellent textbooks on the subject [10–12].

7.7 Exercises

The exercises in this chapter aim at exploring the Fourier transform, including also related concepts (e.g., spectrograms), and different types of sound signals, including speech and music.

Exercise 1 Download and install audacity and download some audio files from the EBU SQAM disc. Open the downloaded wave files in audacity and inspect the spectrum. What can we say about them? Also, have a look at the speech signals.

Exercise 2 Construct a test signal in Pd consisting of a number of sinusoids whose frequency and amplitude you can control independently with sliders. Add noise to the sinusoid and a slider for controlling its power.

Exercise 3 Build a simple spectrum analyzer in an abstraction in Pd that displays the power spectrum of the input. Use `rfft~` to compute the Fourier transform. This object operates on blocks, so use `block~` to control the size of the block. Test it with the test signal from Exercise 2 and experiment with the sliders.

Exercise 4 Connect the input of the spectrum analyzer to the microphone input in your laptop. Whistle and determine the frequency using the spectrum analyzer. Try also with sustained vowels (like "aaaaah"). How does the spectrum look?

Chapter 8
Audio Effects

8.1 Introduction

Audio effects are used in many different applications, including movie production, music production, and live music. Their usage is more widespread in some circles than others. For example, electric guitar players usually rely on a host of different effects, and guitar players are thus a big market for manufacturers of audio effects, such as TC Electronic, Strymon, Boss, and Eventide. The guitar effects often come in the shape of a pedal containing a single or a few effects. The effects are then controlled by knobs and switches mounted on the top of the pedal, which can also be turned on and off with one such switch. Guitar players then combine several such effects by connecting them with cables. The ordering and exact placement of the effects in the signal chain is a matter of much debate. Most of the effects are really quite simple from a theoretical point of view and quite similar too, as many of them rely on what is essentially the same basic filters and principles. We will here take a closer look at some of the more common effects, like the vibrato, tremolo, echo (or delay), flanger, and chorus effects. Many of these effects have in common that they are essentially based on comb (or inverse comb) filters and achieve their effect via the delay of those comb filters. As such, they can be thought of as delay-based effects, although all filters, of course, are based on delays in some sense.

It should be emphasized that the effects discussed here can be designed and realized in many different ways, and that the many different manufacturers of audio effects often have their own design. Thus, the designs discussed herein should not be considered canon but rather one way of making these particular effects.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_8

8.2 Echo

One of the simplest effects yet most ubiquitous is the echo effect, which is sometimes also called the delay effect. It can be found in many different applications, but they are very often used by electric guitar players, particular during solos. An echo is the repetition of a sound sometime after the original sound. A single echo can be generated using the difference function

$$y_n = x_n + bx_{n-D}, \tag{8.1}$$

where D is the time (in samples) between the original sound and the echo. b is a filter coefficient that determines how loud the echo is relative to the original sound. The impulse response of this filter can easily be seen to be

$$1, 0, \ldots, 0, b \tag{8.2}$$

followed by zeros and there is $D - 1$ zeros between 1 and b. A block diagram of the filter is shown in Fig. 8.1. To get an echo of T [s] with a sampling frequency of f_s in Hz, we have that $D = Tf_s$. Conversely, we can convert an echo of D samples into delay in seconds via $T = D/f_s$.

Multiple echos can easily be added to the effect like this:

$$y_n = x_n + b_1 x_{n-D_1} + b_2 x_{n-D_2} + b_3 x_{n-D_3} + \ldots. \tag{8.3}$$

The filter in (8.1) is, of course, an inverse comb filter, while (8.3) contains multiple inverse comb filters in parallel. Another alternative is to use a comb filter, which has the difference equation

$$y_n = x_n + ay_{n-D}. \tag{8.4}$$

This, obviously, adds an infinite number of echos, each being an attenuated version of the previous one. In fact, if an infinite number of echos is added and with each echo in (8.3) being D apart, i.e., the kth echo occurs at kD, and the filter coefficient is chosen such that $b_k = a^k$, then (8.3) and (8.4) are mathematically equivalent. In Fig. 8.2 a block diagram of the feedback echo effect is shown. Sometimes two such comb filters are used parallel, which by some manufacturers is called a dual delay, and sometimes a bit of modulation or a low-pass filter is added in the feedback loop.

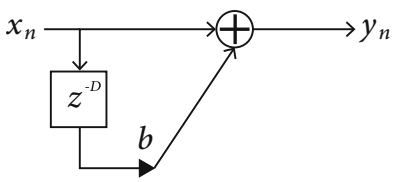


Fig. 8.1 Block diagram of filter structure for a single echo

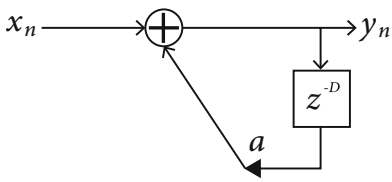


Fig. 8.2 Block diagram of filter structure for echo effect based on feedback

Not only can the comb filter and its inverse be used for these echo effects themselves, they can also be used to produce a number of more complicated effects, some of which we will look more into here. It should be noted that in the comb and inverse comb filters above, the delays have to be sufficiently large for the result being perceived as an echo. A small D, less than 50 ms, in (8.4) or (8.1) will not be perceived as distinct echos but as a change of the timbre or color of the sound. Delay and echo effects thus refer to applications, wherein the delays in the filters are so large that they are perceived as echos, typically 100s of milliseconds.

8.3 Vibrato

An effect that can be implemented using delays, at least in its simplest form, is vibrato. Vibrato refers to the periodic variations in pitch that musicians use as ornaments in musical performances. It can be replicated with filters by passing the input signal through a time-varying delay [13], i.e.,

$$y_n = x_{n-D(n)}. \tag{8.5}$$

This is also called *modulation* or, more specifically, modulation of the frequency. The vibrato effect is depicted in the block diagram in Fig. 8.3. The delay $D(n)$ (which is written as a function of n to emphasize that it changes over time) is typically in the range of 0–10 ms, while it varies at a frequency of 0.1–5 Hz. It cannot be less than zero, since this would imply that it looks into the future (i.e., it is not causal). If the delay was kept fixed at some value, there would be no audible effect. The delay function could be implemented with a sinusoidal function as

$$D(n) = \frac{D}{2}\left(1 - \cos\left(2\pi \frac{f}{f_s} n\right)\right), \tag{8.6}$$

where f is the frequency, or speed, (in Hertz) of the time-varying delay, and D is what you might call the depth, here measured in samples. The delay function, $D(n)$, can be seen to vary between 0 and D.

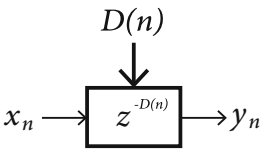


Fig. 8.3 Block diagram of the vibrato effect implemented as a time-varying delay, $D(n)$

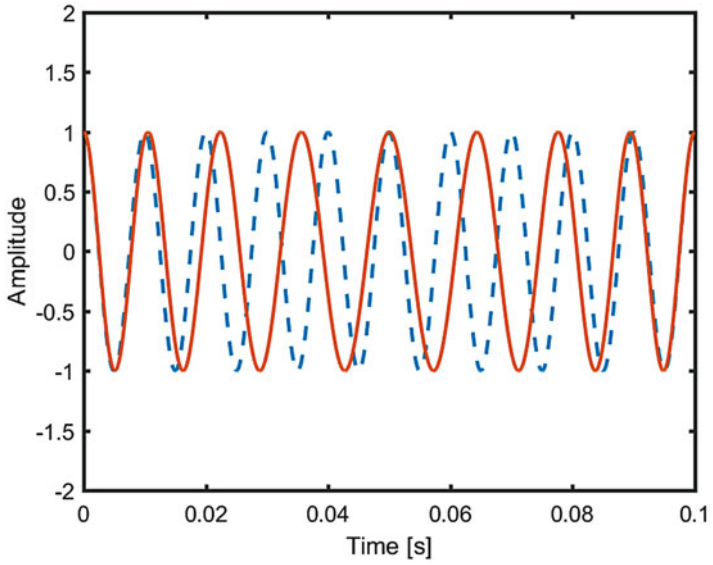


Fig. 8.4 Example of the effect of vibrato on a sinusoid. Shown are the input (dashed) and the output (solid)

The consequences of passing a complex audio signal through a time-varying delay can be a bit hard to understand. The vibrato effect can, however, be analyzed by passing a phasor, $z = e^{j\omega}$, through the time-varying delay, i.e., by setting $x_n = z^n$ and analyzing the output, y_n. This way, it can be shown that a time-varying delay of the form (8.6) will result in a phasor having time-varying frequency given by $\omega(n) = \left(1 - \pi D \frac{f}{f_s} \sin\left(2\pi \frac{f}{f_s} n\right)\right)$. This result generalizes to complex audio signals by observing, as we have done many times by now, that signals can be decomposed into a number of phasors, each of which will be modified in the same manner by the time-varying delay. Note that the frequency of the time-varying delay, f, actually influences not only the rate of change (or speed) of the delay but also the magnitude of the time-varying frequency of the output phasor. This could be avoided by dividing the expression in (8.6) by f.

The effect of the time-varying delay on a sinusoid is illustrated in Fig. 8.4. The figure shows the input, which is a sinusoid of amplitude 1 and frequency 100 Hz sampled at 44.1 kHz, and the output of the vibrato effect, which is here set to operate between 0 and 20 ms at a frequency of 5 Hz. As can be seen, the result is that the frequency of the sinusoid varies periodically, which creates the perception of vibrato.

8.4 Tremolo and Ring Modulation

The tremolo effect is closely related to the vibrato in that it is also based on modulation, only it modulates the amplitude, and thus power, of the input signal rather than its frequency. This creates the perception of a time-varying power. As with the vibrato, a periodically varying function is often used in the tremolo to create this effect. Mathematically, the tremolo effect can be described as the multiplication of the input signal by a so-called modulating signal, i.e.,

$$y_n = x_n \cdot m_n, \tag{8.7}$$

where m_n is this modulating signal. Formally, this type of modulation is called amplitude modulation, and it was a widely used type of modulation in early communication systems, and the reader might recognize it from broadcast radio. In modulation theory, x_n is called the carrier. Amplitude modulation comes in many different shapes and forms, but the type used for the tremolo effect is where the modulating signal m_n has two important properties, namely, that it is non-negative, i.e., $m_n \geq 0$, and it is slowly varying. A modulating signal that possesses these properties is one similar to the delay function in (8.6), i.e.,

$$m_n = \left(1 - A\cos\left(2\pi\frac{f}{f_s}n\right)\right), \tag{8.8}$$

where f is then fairly low, typically below 20 Hz. If the frequency, f, is set too high, the resulting effect will not be perceived as a time-varying loudness but as adding roughness to the input signal, something that is usually undesirable in most applications. For the modulating signal to be non-negative we must require that the parameter A is less than one and larger than zero, i.e., $0 < A \leq 1$. Other types of modulating signals can also be used, such as a clipped version (8.8).

As for the vibrato, the effect can be understood by passing a phasor through the effect. Let the input be equal to a phasor of frequency ω, i.e., $x_n = e^{j\omega n}$, then the output of the tremolo effect in (8.7) is given by

$$y_n = z^n \cdot m_n, \tag{8.9}$$

and the magnitude of the input is thus

$$|y_n| = |z^n||m_n| = m_n, \tag{8.10}$$

since a phasor has unit magnitude. Hence, the magnitude, which is closely related to the power of a phasor and thus its loudness, is given by the magnitude of the modulating signal, i.e.,$|m_n|$, which is simply equal to m_n since the modulating signal is non-negative. It should be noted that the generalization of this analysis of the tremolo effect to more complex signals is not as straightforward as for the vibrato

effect and requires additional mathematics, including the concepts of the Hilbert transform and signal envelopes [14]. In fact, it is not even particularly simple for something as simple as a real sinusoid, as its magnitude is not constant. Instead, we can simply observe that the magnitude of the output will be modified by the modulating signal.

In Fig. 8.5, the tremolo effect is illustrated with a phasor as input. Shown is the real part of the phasor and of the output. Also shown are the magnitudes of the input and the output. It can be seen that the magnitude of the phasor, which is otherwise constant, fluctuates after it has been passed through the tremolo effect. The tremolo effect shown here used a modulating signal as in (8.8) with the parameters $f = 10\,\text{Hz}$ and $A = 0.2$.

Another interesting effect that can be achieved via amplitude modulation is *ring modulation*. It is achieved much like the tremolo, as described in (8.7), but with some important differences. In ring modulation, m_n takes on the role of the carrier and is typically a chosen sinusoidal signal having a frequency higher than 20 Hz, i.e., faster than that of the tremolo. Then, the input signal is the modulating signal. Moreover, neither m_n nor x_n are necessarily non-negative. This produces a distinct and funny effect that is often used in sci-fi movies. For more on modulation theory in general, and not just for audio applications, we refer the interested reader to the textbook [6].

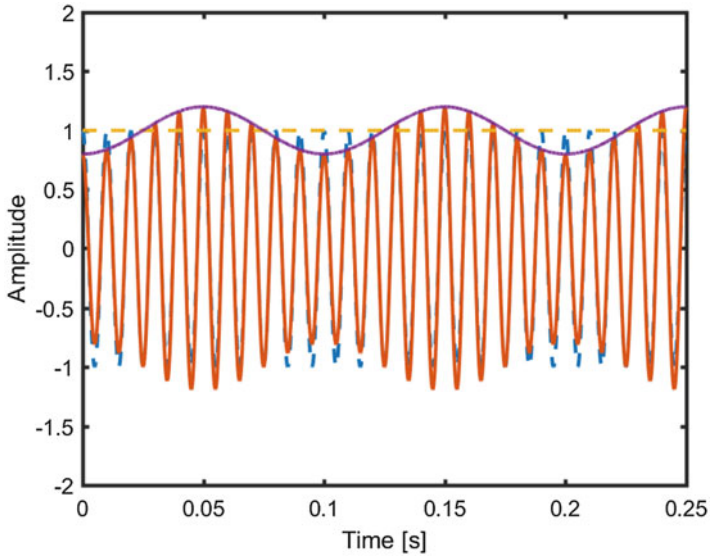


Fig. 8.5 Example of the tremolo's effect on a phasor. Shown are the real part of the input (dashed) and the output (solid) along with the magnitudes of the two signals

8.5 Chorus

The chorus effect imitates the effect of several musical instruments playing, or singers singing, the same part while not being completely in sync and playing at the same volume. When used on single instruments, it leads to a fuller, richer sound. It is often used on clean electric guitar parts in certain up-tempo (and loud) genres and was extremely popular in the 1980s. To emulate multiple instruments playing the same thing, we could add several versions of the signal to itself. This would, however, only make the sound louder and not change its timbre. Instead, we can introduce a slowly varying delay to each copy and multiply it by different gains. To emulate two instruments playing together, we could do it like this:

$$y_n = x_n + g x_{n-D(n)}, \tag{8.11}$$

where g is the gain (or mix) parameter and $D(n)$ is the time-varying delay function that we use to control and vary the amount of delay. A block diagram of the chorus effect is shown in Fig. 8.6. We can recognize this as an inverse comb filter, and the basic filter is thus similar to that of an echo. Only, the involved delays are so small that they are not perceived as distinct echos, and they are time varying! We can envision the effect of the time-varying delay on the inverse comb filter as the troughs of the magnitude spectrum moving around slowly, their distance increasing and decreasing over time as the delay changes. Another way to think about it is that $x_{n-D(n)}$ corresponds to the output of the vibrato effect, meaning that the output of the chorus effect is the input signal mixed with the output of a vibrato effect!

To emulate the sound of multiple such instruments playing together, we could add more terms, corresponding to connecting multiple comb filters in parallel, i.e.,

$$y_n = x_n + g_1 x_{n-D_1(n)} + g_2 x_{n-D_2(n)} + g_3 x_{n-D_3(n)} + \ldots. \tag{8.12}$$

We refer to each of these terms as a branch, and each branch then contains a different vibrato effect. The function $D_k(n)$ is then the delay function for the kth branch and g_k its mix parameter. The delay functions for the different branches should then be chosen or initialized slightly differently. A common choice for the delay function is low-pass filtered random noise [7], which results in a delay function that varies slowly between random points. This ensures that the different branches start with different delays and vary differently. This can be done by passing white noise (for

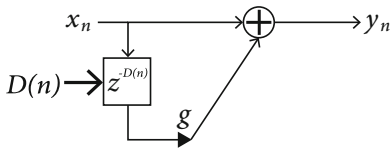


Fig. 8.6 Block diagram of the chorus effect implemented as a feedforward filter with a time-varying delay, controlled by the function $D(n)$

example, uniformly distributed random numbers), through a one-pole low-pass filter having the difference equation (with x_n, the input, now being the noise, and y_n, the output, representing the delay function)

$$y_n = (1 - a)x_n + ay_{n-1}, \tag{8.13}$$

where $0 < a < 1$ results in a low-pass filter. The scaling of x_n by $(1 - a)$ ensures that the gain at 0 Hz is 0 dB, i.e., 1 on a linear scale. For more on this, see [7, 15]. Another option than the low-pass filtered noise is to use a sinusoidal function, like

$$D_k(n) = F_k + \frac{D_k}{2}\left(1 - \cos\left(2\pi \frac{f_k}{f_s} n\right)\right), \tag{8.14}$$

where F_k is the delay offset in samples, D_k the depth in samples, and f_k the frequency (or speed) of branch k. Some typical values that produce the characteristic chorus effect with a single branch is an offset, F_1, corresponding to 10 ms, a frequency, f_1, of 0.2 Hz, with a depth, D_1 of 20 ms.

There exists different implementations of the chorus effect by different manufacturers, and they can vary, for example, in terms of the number of branches, the types of delay functions used, how the delays are implemented, and so forth. Moreover, some fixed amount of feedback, aside from the time-varying feedforward part, can also be added to the chorus effect [13].

8.6 Flanger

The flanger effect is mathematically similar to the chorus effect and the echo effect, but it sounds quite different. The flanger effect is similar to what we experience when a jet plane flies over our heads. The direct sound from the jet is mixed with the sound reflecting off the ground, which has a time-varying delay due to the jet's movement. Like the chorus effect, the flanger effect is quite popular with guitar players and it can frequently be found in their pedal boards and in their racks. From the jet plane analogy, it can be understood that the flanger effect can be achieved with an inverse comb filter as

$$y_n = x_n + ax_{n-D(n)}, \tag{8.15}$$

where a is the mix parameter of the flanger effect and $D(n)$, like for the chorus, is the delay function that controls the amount of delay, which here varies over time. Unlike the chorus effect, typically only a single branch is used in the flanger. Since the flanger is just a time-varying inverse comb filters, we can guess that the effect of the addition of the direct sound with the delayed version with the time-varying delay is that we have signal cancellation at equally spaced frequencies in the magnitude spectrum and that the spacing between these frequencies varies depending on the delay.

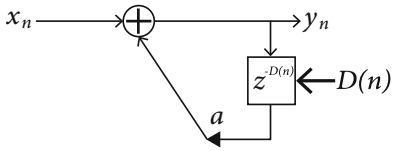


Fig. 8.7 Block diagram of the flanger effect implemented as a feedback filter with a time-varying delay, controlled by the function $D(n)$

The flanger effect can also be implemented using a comb filter, i.e., as a feedback filter, as follows [7]:

$$y_n = x_n + a y_{n-D(n)}. \tag{8.16}$$

What we for the inverse comb filter referred to as the mix parameter, a, now acts as a gain that controls the amount of feedback, and for the filter to be stable, we require that $|a| < 1$ while there is no such requirement for the inverse comb filter. The feedback version of the flanger tends to be more pronounced than the feedforward version. It is, however, a matter of taste whether one prefers one or the other. A block diagram of the feedback version of the flanger effect is depicted in Fig. 8.7.

For both the feedback and the feedforward implementations of the flanger effect, a sinusoidal (or triangular) function is typically used for controlling the delay. It might be of the form

$$D(n) = \frac{D}{2}\left(1 - \cos\left(2\pi \frac{f}{f_s} n\right)\right), \tag{8.17}$$

where D is the depth and f the frequency or speed. This looks quite similar to the delay function used in the chorus, i.e., in (8.14). Notice that, unlike the chorus effect, there is no delay offset in $D(n)$, so the function goes between 0 and D. The frequency in flanger effects is usually between 0.1 and 1 Hz, while the depth is between 0 and 2 ms. In Fig. 8.8, the delay function for the chorus effect (8.14) and for the flanger effect in (8.17) is shown for typical parameter values. As can be seen, the chorus delay function has a non-zero offset and a bigger depth than the flanger delay function.

8.7 Phaser

The phaser effect is another classical effect. Curiously, it was invented as a failed attempt to emulate that flanger effect with few resources. It is achieved by connecting a number of time-varying all-pass (or notch) filters in series and then mixing their output with the input. The all-pass filters cause time-varying phase shifts that in turn cause amplifications and cancellations when mixed with the input signal. As alluded to, the phaser effect can be implemented in different ways [7, 16]. Here, we will now describe how it can be implemented with a number of second-order all-pass filters [17]. Second-order all-pass filters are good for generating

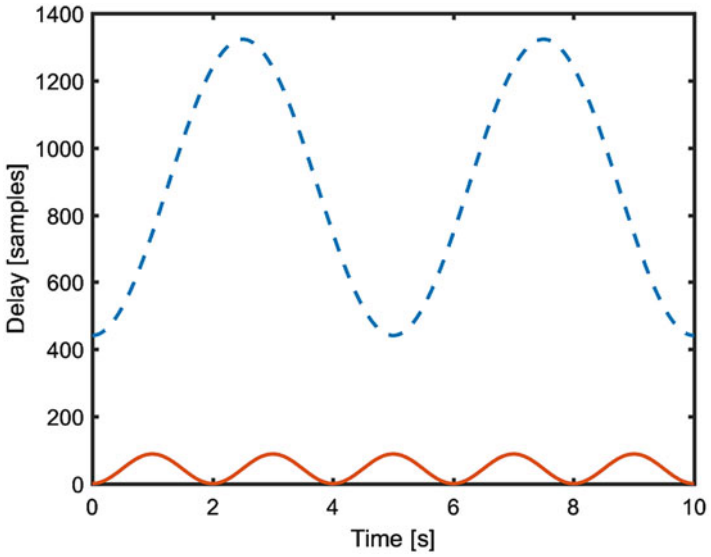


Fig. 8.8 Examples of delay functions for the chorus (dashed) and flanger (solid) effects, as described in (8.14) and (8.17), respectively

notches at particular frequencies that can be controlled independently. The notches are generated at frequencies where the phase response of the combined all-pass filters is π, as phasors at that frequency will cancel out when added to the input.

As mentioned, the phaser effect can be built from a number of all-pass filters. One second-order all-pass filter, the kth is given by the transfer function [18]

$$H_k(z) = \frac{b_k + a_k z^{-1} + z^{-2}}{1 + a_k z^{-1} + b_k z^{-2}}, \tag{8.18}$$

where $a_k = -2R_k \cos(\theta_k)$ and $b_k = R^2$. R_k is the radius of the pole, and the closer it is to the unit circle, the narrower the resulting notch will be. The pole radius is related to the bandwidth of the kth notch, B_k, as $R_k = e^{-\pi B_k/f_s}$ [17]. The parameter θ_k is the angle of the pole which controls the frequency, f_k, at which the notch will occur due to the kth filter. It is related to the frequency in Hertz as $\theta_k = 2\pi f_k/f_s$. If we connect four all-pass filters of the form (8.18) in series and mix the output with the input, we obtain a complete filter with the transfer function

$$H(z) = g + (1 - g)H_1(z)H_2(z)H_3(z)H_4(z), \tag{8.19}$$

where $0 \leq g \leq 1$ is the mix parameter. The angles of the poles of the individual filters, θ_k, can then in principle be varied independently. This can be done, for example, using a low-frequency oscillator, to produce the effect. The pole radius can be set to be the same for all filters, i.e., $R_k = R$ for all k, corresponding to

a fixed bandwidth in Hertz, or a constant Q can be used such that the notch gets wider for higher frequencies. The Q factor is simply the ratio between the frequency and the bandwidth, i.e., $Q = f_k/B_k$. The rate at which the pole angles, θ_k, for $k = 1, 2, \ldots$ change is typically controlled by the same low-frequency oscillator (which is typically chosen from 0.02 to 0.5 Hz), but the range then differs from one all-pass filter to the next. For example, f_1 could vary between 16 Hz and 1.6 kHz, f_2 between 33 Hz and 3.3 kHz, f_3 between 48 Hz and 4.8 kHz, and f_4 between 98 Hz and 9.8 kHz [19]. Note that an amount of feedback can be introduced in the phaser effect (see, e.g., [16]) and that an extension to the phaser to stereo can be found in [17].

8.8 Time-Varying and Fractional Delays

In connection with the discussed effects, there are some practical issue that have to be solved. The chorus and the flanger that we have discussed here rely on a time-varying delay function, here exemplified with a sinusoidal function to control the delay. One might wonder how exactly one goes about generating such sinusoidal functions of any given frequency. This can be done in multiple ways, including the most straightforward approach, namely, lookup tables with interpolation, but an interesting approach is based on filters. Suppose we would like to design a filter that outputs a sinusoid (here it is actually a cosine) when an impulse is put on the input, as shown in Fig. 8.9. Such a filter should therefore have the following impulse response, where ω_0 is the desired frequency in radians:

$$h_n = \begin{cases} \cos(\omega_0 n) & \text{for } n = 0, 1, 2, \ldots \\ 0 & \text{otherwise} \end{cases}. \tag{8.20}$$

Such a filter would be causal, as its impulse response is zero for negative time indices, but also infinite, which would imply that some kind of feedback would be involved. To get a desired frequency in radians, we use the formula $\omega_0 = 2\pi\ f_0/f_s$, where f_0 is the desired frequency in Hertz and f_s is the sampling frequency.

By taking the z-transform of the above impulse response, we obtain the following transfer function [7]:

$$H(z) = \frac{1 - \cos(\omega_0)z^{-1}}{1 - 2\cos(\omega_0)z^{-1} + z^{-2}}. \tag{8.21}$$

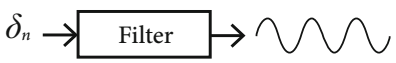


Fig. 8.9 Sinusoidal function generator via a filter whose impulse response is a sinusoid

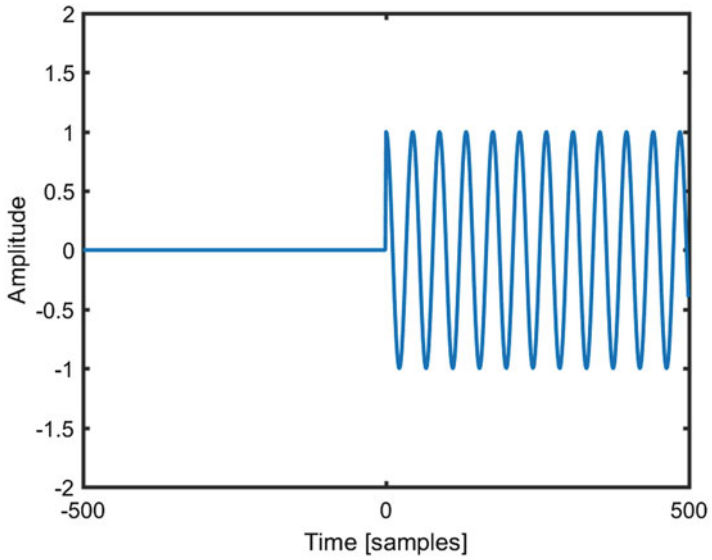


Fig. 8.10 Impulse response of the filter in (8.22) with a frequency of 1000 Hz for a sampling frequency of 44.1 kHz

This can be done by splitting the cosine into two complex exponential functions using Euler's formula and then using that $\sum_{n=0}^{\infty} \alpha^n = \frac{1}{1-\alpha}$. It is a second-order filter, and involves both feedback and feedforward. By solving for the roots of the second-order polynomial in the denominator, we find that the filter has poles at $e^{\pm j\omega_0}$, i.e., at the location on the unit circle corresponding to our desired frequency. Note how this is actually in contradiction with our condition for stability!

From the transfer function, we can easily see that the difference equation that realizes the filter is given by

$$y_n = x_n - \cos(\omega_0)x_{n-1} + 2\cos(\omega_0)y_{n-1} - y_{n-2}. \tag{8.22}$$

By putting a digital impulse (also called the Kronecker delta function) on the input of this filter, i.e., $x_n = \delta_n$, we obtain the desired sequence $\cos(\omega_0 n)$ on the output of our filter. An example of an impulse response obtained this way is shown in Fig. 8.10 for a desired frequency corresponding to 1000 Hz when sampling at 44.1 kHz. A block diagram of this filter is shown in Fig. 8.11. As can be seen, the output of the filter is zero until the impulse occurs at $n = 0$ after which the output is sinusoidal. In practice, it should be noted that because of finite word-length effects, which can cause the poles to not fall exactly on the unit circle, the filter's impulse response might be decaying or growing in practice and special care must be taken to take care of this. For more on how to generate sinusoids, we refer the interested reader to [7, 15].

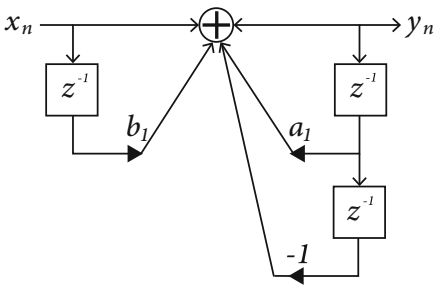


Fig. 8.11 Block diagram of the oscillator filter with $b_1 = -\cos(\omega_0)$ and $a_1 = 2\cos(\omega_0)$. If a Kronecker delta function, δ_n, is put on the input, the output will be a cosine function of frequency ω_0 sampled at $n = 0, 1, \ldots$

Now that we have seen how to generate the function that controls the delay, we can now turn our attention to another question. Since sinusoids are continuous functions, it means that the output of our filter, in principle, is also continuous, and this poses a challenge for us, because how exactly do we implement a delay that is not guaranteed to be an integer number? Such delays are called fractional delays. One way to solve this problem is by interpolation, the simplest incarnation being linear interpolation where the signal value in between two samples is found by drawing a straight line between the two samples and then using this to determine the signal value. Another approach is via filtering with the so-called fractional delay filters [13, 20]. Fractional delays, for example, can be implemented with an all-pass filter of the form

$$y_n = ax_n + x_{n-1} - ay_{n-1}. \tag{8.23}$$

An all-pass filter has a flag magnitude response and, hence, does not change the magnitude of the input. It does, though, change the phase. The phase response $\Phi(\omega)$ of an all-pass filter is approximately given by

$$\Phi(\omega) \approx -\omega d, \tag{8.24}$$

where d is the delay. Recall that a phaser $z = e^{j\omega n}$ of frequency ω passed through a filter has its magnitude changed by the magnitude response at that frequency and the phase changed by the phase response, also at that frequency, i.e., $|H(\omega)|e^{j\omega n+\Phi(\omega)}$. Since the phase response for the all-pass filter is here given by (8.24) and $|H(\omega)| = 1$, the resulting phaser that appears on the output will be $e^{j\omega n-\omega d} = e^{j\omega(n-d)}$, which means that the phaser will be delayed by d samples regardless of its frequency.

This delay, d, is related to the filter coefficient a as

$$d = \frac{1-a}{1+a}, \tag{8.25}$$

which means that for a desired fractional delay $0 < d < 1$, we can compute the required filter coefficient via the following formula:

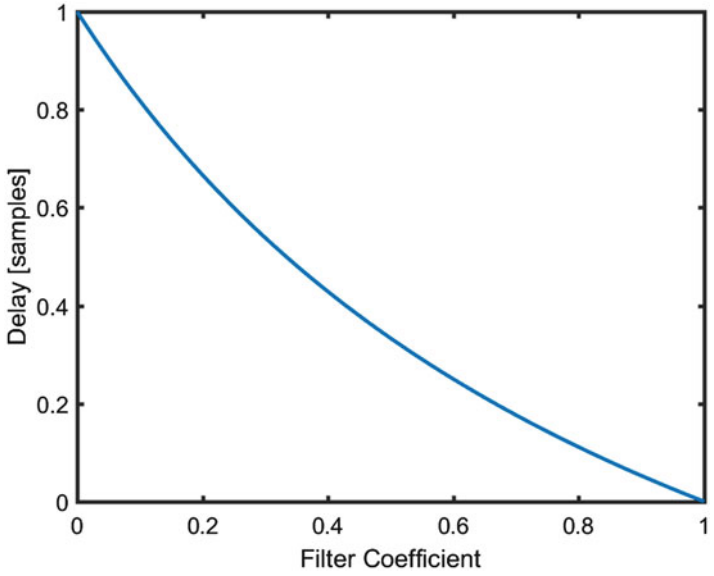


Fig. 8.12 Relationship between the filter coefficient, a, and the resulting delay, d, in the fractional delay filter, as computed using (8.25)

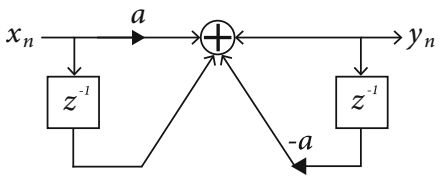


Fig. 8.13 Block diagram of the fractional delay filter, which is an all-pass filter. With $a = (1 - d)/(1 + d)$, where $0 < d < 1$, the filter passes the input with a fractional delay of d samples

$$a = \frac{1-d}{1+d}. \tag{8.26}$$

So, if the time-varying delay for sample n according to our function generator is supposed to be 5.19 samples, we can implement the integer part as a normal integer delay, z^{-5}, while we can realize the fractional part, 0.19, using our all-pass filter. It is illustrated in Fig. 8.12 how different filter coefficients a result in different delays d. More specifically, the figure shows the formula (8.25). Also, in Fig. 8.13 a block diagram of the fractional delay filter can be seen.

8.9 Exercises

The following exercises are practical exercises involving the implementation of the audio effects in Pure Data.

Exercise 1 Implement a vibrato effect in Pd using an oscillator to control the variable with a maximum delay of 20 ms and a frequency of 5 Hz. Remember to use the `vd~` object as the delay is time varying and to use the `block~` object inside an abstraction to realize low delays. Extend it to include sliders for controlling these parameters.

Exercise 2 Design a flanger effect in Pd using an oscillator to control the variable delay of a feedback comb filter. Use a variable delay $d(n)$ between 0 and 10 ms, a frequency of 0.1 Hz for the oscillator, and a mix parameter a of 0.9. As before, use the `vd~` object to obtain a variable delay with fractional delays and `block~`. Expand the patch to use sliders to control the depth, frequency, and mix parameters.

Exercise 3 Design a chorus effect in Pd with just a single feedforward comb filter with an oscillator for controlling the delay. Use a delay offset of 10 ms, a depth of 20 ms, and a frequency of 0.2 Hz to control the delay. Expand the design to include sliders for controlling the parameters.

Exercise 4 Extend the chorus effect from the previous exercise to include three feedforward paths with different parameters. Aside from the delay offset, depth, and frequency, each branch should also have a mix parameter between 0 and 1. Experiment with the settings.

Exercise 5 Implement a sinusoidal function generator using the filter having the difference equation in (8.22). It should produce a sinusoidal signal with a frequency of 1000 Hz. Use the `biquad~` object to implement the second-order filter. Excite the filter with an impulse using the `dirac~` object and plot the output of the filter in Pd with an array.

Exercise 6 Implement a dual delay effect comprising two feedback comb filters connected in parallel. The two delays should have independent controls for their parameters, i.e., delay and feedback filter coefficient. Add a gain to the output of each comb filter before they are added so the outputs can be mixed at different volumes.

Exercise 7 A doubler effect is a stereo effect that makes a single instrument (typically a guitar) sound like two instruments playing the same. The principle is similar to the chorus. In the left channel, the original signal is played, while in the right, the signal is delayed relative to the signal in left channel, typically somewhere between 5 and 60 ms, and a bit of vibrato is added. Implement such an effect in Pd based on the previous patches.

Exercise 8 Implement a phaser effect using four all-pass filters. Start by building an abstraction for the second-order all-pass filters and remember to use the `biquad~` object. Control the notch frequencies with a 0.25 Hz low-frequency oscillator and a pole radius of 0.8. Let the notch frequency of the all-pass filters vary as described next: f_1 varies between 16 Hz and 1.6 kHz, f_2 between 33 Hz and 3.3 kHz, f_3 between 48 Hz and 4.8 kHz, and f_4 between 98 Hz and 9.8 kHz.

Chapter 9
Spatial Effects

9.1 Introduction

Music is usually recorded dry and with only a single microphone, i.e., with a microphone placed closely to the instrument with as little influence from the surroundings as possible. When played back without further modification, signals recorded this way tend to not sound very good and, worse yet, they do not sound natural, as they lack the sound of a natural acoustical environment. To mitigate this, artificial reverberation (reverb for short) is usually applied to such signals before playback. This is done not only in music production, but also in television and movie production. Artificial reverberation basically makes it sound like a sound is being played in a desired environment, such as a large concert hall, a conference center, a cathedral, or a high-quality music recording studio. In this chapter, we will discuss some simple methods for making artificial reverberation. Moreover, even though music is normally recorded with a single microphone, music, and other audio material as well, is most often reproduced in multi-channel loudspeaker configurations like stereo or surround sound, not just in mono. This makes it possible to control the spatial characteristics of different sources by, for example, placing different sources at different angles relative to the listener, which not only makes it sound more natural but also makes it easier to hear the different sources clearly. This can be achieved using simple panning techniques, which we will also cover, although somewhat briefly, in this chapter. In a sense, the two topics, reverb and panning, are closely connected, as they both deal with artificially creating the perception of space and ambience and could be considered spatial effects.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_9

9.2 Impulse Responses and Reverb

As we have seen in earlier chapters, the effect of a linear time-invariant system on a signal can be understood in terms of the impulse response of the system. This is also the case for listening spaces, such as living rooms, music recording studios, cinemas, and so forth. The effect of convolution of a signal with an impulse response can be understood as delaying, scaling, and adding a multitude of different echoes. The impulse response of a normal listening space can usually be described well as comprised of three parts, namely, the direct sound, the early reflections, and the late reflections. The direct sound is the part that travels directly from the source to the listener (or microphone). This sound may be attenuated (i.e., scaled) somewhat and there might be a low-pass filtering effect to account for the propagation of the waves through air, depending on the distance. The early reflections are the sounds that have bounced off the objects or walls a few times on their way to the listener. This part tends to be quite sparse in nature. The late reflections are, however, more stochastic (i.e., noise-like), as they result from multiple echos that have bounced off the walls several times, appearing to come from all directions simultaneously. The impulse response depends on many factors, such as the location of the source and the listener, the room geometry, the material of the surfaces, and the temperature and humidity. The goal of artificial reverberation is to produce, in a simple way, a method for replicating the effect of natural sounding listening environments.

Perhaps the conceptually most straightforward method of applying artificial reverberation is the so-called convolution method. The idea is that if we know the impulse response, we can simply apply it to any signal by convolution, and the result will then sound like the room that it was recorded in. In math, convolution can be written as

$$y_n = \sum_{m=0}^{M-1} h_m x_{n-m}, \tag{9.1}$$

where x_n is the input signal, y_n the output signal, and h_n the impulse response, which is here assumed to consist of M consecutive samples. This also means that we can think of the effect of reverberation as a feedforward filter with filter coefficient h_m. To use this method, we must know the impulse response h_n, which means we must derive it from a simple mathematical model, simulate it with a method such as the image method [21], or measure it from a real acoustical environment. How does one then go about measuring the impulse response of a listening space like, say, a room? The definition of an impulse response is that it is the output of a linear time-invariant system when an impulse is applied to the input. An impulse is an event that is very short in duration. Formally, a digital impulse is defined as

$$\delta_n = \begin{cases} 1 & \text{if } n = 0 \\ 0 & \text{otherwise.} \end{cases} \tag{9.2}$$

Fig. 9.1 The impulse response of a system, like a room, can be obtained by putting an impulse on its input and measuring the output

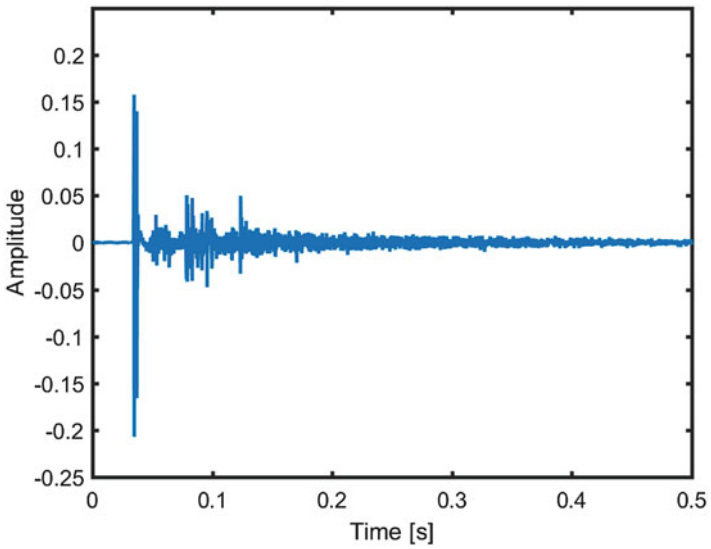


Fig. 9.2 Example of a real impulse response of a room, in this case a hall

This is also known as the Kronecker delta function, as already mentioned. If we set x_n equal to δ_n and insert it in the expression for the convolution in (9.1), we can see that the result is simply $y_n = h_n$. This process for obtaining the impulse response of a system is shown in Fig. 9.1. In the analog world, it is, however, a bit more complicated as the analog equivalent to the Kronecker delta function is the Dirac delta function, which is a so-called generalized function or distribution and not a normal function. It can be heuristically thought of as a function that is infinite at time zero and zero everywhere else. Such a function is difficult, if not impossible, to replicate mechanically, and other ways must be found. In simple terms, the impulse response of a room can be thought of as the sound you hear when you clap your hands one time.

In Fig. 9.2, an example of an impulse response of a room is shown. The impulse response is measured in a hall. A number of observations can be made from the figure. The early reflections can clearly be seen in the form of very distinct spikes early on. Moreover, it can also be seen that the impulse response decays, although it can also be seen to be quite long. It can also be seen that the latter part of the impulse response contains less clear spikes.

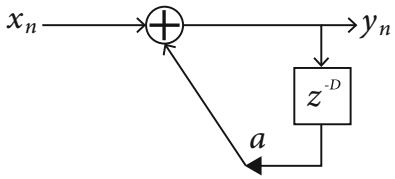


Fig. 9.3 Block diagram of the plain reverberator

While conceptually simple, the convolution method is not very practical for artificial reverberation as it requires many computations. The impulse responses can be very long, like several seconds for, say, a cathedral. With a sampling frequency of 44.1 kHz, this means that for an impulse response that does not decay below some threshold until after, for example, 2 s, we would have $M = 88{,}200$ and would have to perform 88,200 multiplications and additions per sample to compute y_n using (9.1)! Moreover, it is difficult to change the perceptual qualities of the acoustic environment via only a recorded impulse response. We may, for example, wish to change the perceived size or shape of the room or the materials of the surfaces (Fig. 9.3).

9.3 Building Blocks

We will now go over some basic building blocks that are used in many artificial reverberation algorithms [7], including the Schroeder's reverb which we will go into detail later in this chapter. We can think of the effect of a sound reflecting off a wall or an object and then propagating to the listener as an echo of the direct sound. Hence, to mimic this effect, we might use the following filter:

$$y_n = x_n + bx_{n-D}. \tag{9.3}$$

This will produce an echo of x_n D samples later and the echo is attenuated by b. We can recognize this as not only a feedforward but also the inverse comb filter. Since room impulse responses involve many echos, we might add a number of such echos, as many as we desire, and these could have different delays and gains. We could also add many echos of the form (9.3), i.e.,

$$y_n = x_n + bx_{n-D} + b^2x_{n-2D} + b^3x_{n-3D} + \ldots. \tag{9.4}$$

Taking the z-transform of this difference equation yields

$$Y(z) = X(z)\left(1 + bz^{-D} + b^2z^{-2D} + b^3z^{-3D} + \ldots\right) \tag{9.5}$$

and the transfer function is, hence, $H(z) = 1 + bz^{-D} + b^2z^{-2D} + b^3z^{-3D} + \ldots$. This transfer function can also be expressed as $\sum_{k=0} \left(bz^{-D}\right)^k$ and this is called a geometric series. Geometric series are of the form $\sum_{k=0}^{\infty} \alpha^k$ and converge to $\frac{1}{1-\alpha}$ for $|\alpha| < 1$. With $|b| < 1$, our series above converges too and a simple closed-form expression exists for the results, namely,

$$1 + bz^{-D} + b^2z^{-2D} + b^3z^{-3D} + \ldots = \frac{1}{1 - bz^{-D}}, \tag{9.6}$$

which is the transfer function of a feedback filter, or, more precisely, a comb filter with delay D and gain b in the feedback loop. Instead of using the convolution method directly, we might thus think of synthesizing the effect of reverberation with some simple, feedback filters, as even a simple feedback filter with just a single delay can produce infinitely many echos. Recall that the comb filter has the difference equation

$$y_n = x_n + ay_{n-D}. \tag{9.7}$$

It has the transfer function

$$H(z) = \frac{1}{1 - az^{-D}}, \tag{9.8}$$

and its impulse response is

$$h_n = \delta_n + a\delta_{n-D} + a^2\delta_{n-2D} + a^3\delta_{n-3D} + \ldots. \tag{9.9}$$

Written as a sequence of numbers, the impulse response is

$$1, 0, \ldots, 0, a, 0, \ldots, 0, a^2, 0, \ldots, 0, a^3, 0, \ldots, \tag{9.10}$$

with $D - 1$ zeros between non-zero values. The comb filter is stable when $|a| < 1$, which means that the impulse response decays over time. In Fig. 9.4, the impulse response of the comb filter is shown for $a = 0.75$ and $D = 100$. As can be seen, the parameters can actually easily be inferred from the figure, as D is the spacing between the non-zero values and a is the value at $n = D$. Had chosen $|a| > 1$, we can easily see that the impulse response would grow over time, and since it is infinitely long, the impulse response would keep growing and growing, and this will eventually cause overflow in the computer or clipping in the digital-to-analog converter.

The comb filter is one of the basic building blocks of Schroeder's reverb. In this context, it is often referred to as the plain reverberator. A block diagram of the filter is shown in Fig. 9.3. To build a reverb from comb filters, one can simply connect a number of comb filters with different delays and possibly also different gains in parallel and add the output of the filters. This does, however, not sound very natural

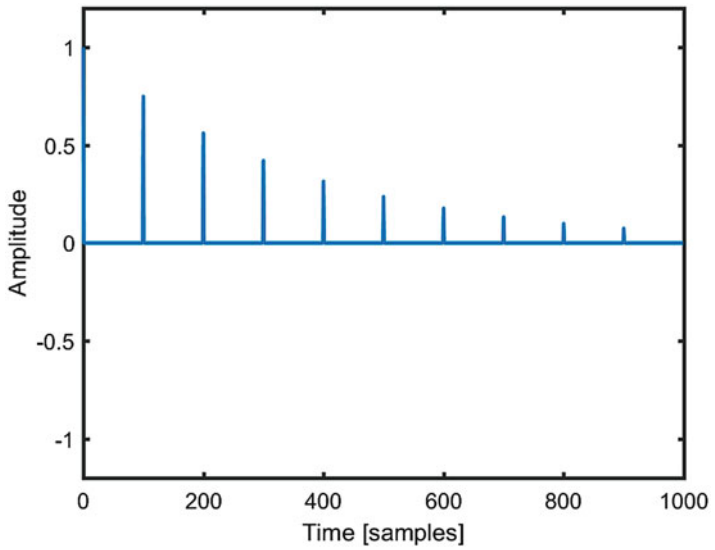


Fig. 9.4 Impulse response of the comb filter with $a = 0.75$ and $D = 100$

or very good, due to the very strong peaks in the frequency response of the comb filter. The teeth of the comb filter cause unnatural correlations in the output signal. Instead, another building block is often used, namely, the all-pass reverberator. It has the difference equation

$$y_n = ay_{n-D} - ax_n + x_{n-D}. \tag{9.11}$$

Its transfer function is given by

$$H(z) = \frac{-a + z^{-D}}{1 - az^{-D}}. \tag{9.12}$$

A block diagram of the all-pass reverberator is shown in Fig. 9.6. Its magnitude response is flat as the numerator and denominator of the transfer function are equal for all frequencies, i.e., $|H(\omega)| = 1$ for all ω. Its impulse response is, however, given by

$$h_n = -a\delta_n + (1-a^2)\delta_{n-D} + (1-a^2)a\delta_{n-2D} + (1-a^2)a^2\delta_{n-3D} + \ldots, \tag{9.13}$$

and when written as a sequence of numbers, it is

$$-a, 0, \ldots, 0, 1-a^2, 0, \ldots, 0, (1-a^2)a, 0, \ldots, 0, (1-a^2)a^2, \ldots, \tag{9.14}$$

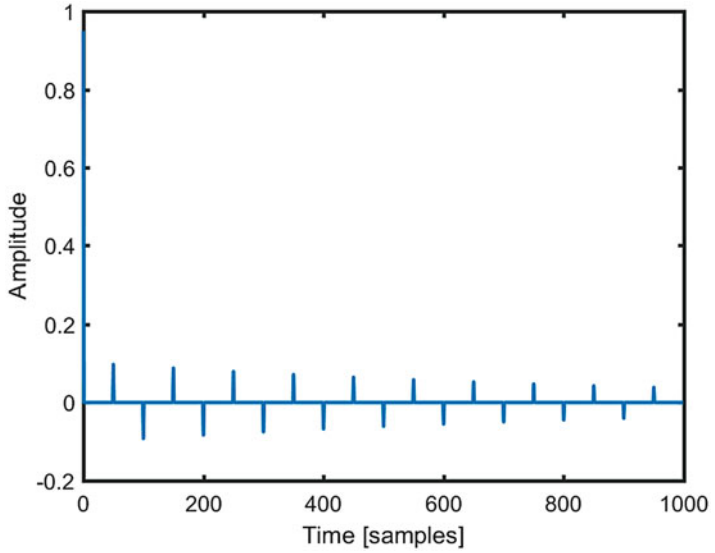


Fig. 9.5 Impulse response of the all-pass filter with $D = 50$ and $a = -0.95$

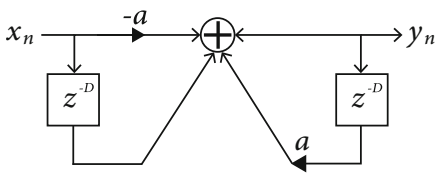


Fig. 9.6 Block diagram of the all-pass reverberator

as before with $D - 1$ zeros in between non-zero values. As with the comb filter, we can easily see that the filter is stable when $|a| < 1$, as the impulse response gets smaller and smaller as a function of n. A typical example of the impulse response of such an all-pass filter is depicted in Fig. 9.5 for $a = -0.95$ and $D = 50$. It should be noted that the all-pass filter discussed in earlier chapters is a special case of this all-pass filter, namely, with $D = 1$.

Inspecting the block diagram of the all-pass reverberator in Fig. 9.6, we can spot something of concern. The all-pass reverberator requires a delayline, i.e., some delays, for both the input sequence, x_n, and the output sequence, y_n. If the delay D is high, which it often is in the context of reverbs, then this requires quite a lot of memory. Since both the feedforward path of the filter and the feedback part constitute filters by themselves, it means that we can change the order of the two parts. The result is shown in Fig. 9.7. It can be seen that by this rearrangement of the parts of the filters, the filter now only requires a single delayline, rather than the two in Fig. 9.6. This means that we have now cut the memory requirements

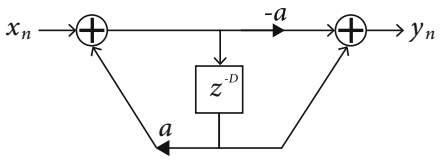


Fig. 9.7 Block diagram of the all-pass reverberator with rearranged parts

of our filter in half. The filter still does exactly[1] the same as before. This can, for example, be verified by putting a Kronecker delta function, δ_n, on the input, which yields the same sequence as (9.14). The two different implementations in Figs. 9.6 and 9.7 are called *realization structures*, and are thus different realizations of the same filter. The one in Fig. 9.6 is called direct form I, while Fig. 9.7 is called direct form II. Many different realization structures of filters exist (see, e.g., [8]), and they have different properties, e.g., different memory requirements, different numerical properties.

9.4 Schroeder's Reverb

Manfred Schroeder introduced the idea of artificial reverberation and later proposed a simple design based on a combination of plain and all-pass reverberators. Not only is it conceptually simple, it is also computationally simple, and it was, and still is, thus also a very practical solution. While we today have many more resources at our disposal and can do more complicated things, Schroeder's reverb remains a classic type of reverb, and reverbs of his design or inspired by his design remain in use today. We here use it mainly as an example for demonstrating the principles of artificial reverberation. His design is based on a number of comb filters, here four, combined in parallel followed by a number, here two, of all-pass filters in series. The outputs of the comb filters are added after which they are fed through first one all-pass filter and then the next. Let $H_1(z)$, $H_2(z)$, $H_3(z)$, and $H_4(z)$ denote the transfer functions of the four comb filters. Each of these is given by

$$H_k(z) = \frac{b_k}{1 - a_k z^{-D_k}}, \text{ for } k = 1, 2, \ldots, 4, \tag{9.15}$$

where D_k is the delay, a_k the feedback coefficient, and b_k a mixing parameter. The mixing parameter, which was not featured in the comb filter so far, is useful when combining the outputs of a number of filters connected in parallel. The higher the

[1] This is the case when analyzing the filters with pen and paper, but they have different behavior in terms of sensitivity to fine word-length effects, so the filters may in some cases yield different results in practice.

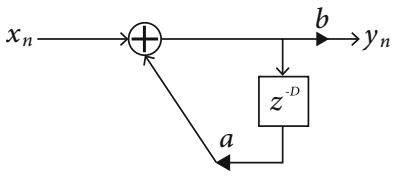


Fig. 9.8 The comb filter in Schroeder's reverb with feedback parameter a, delay D, and mix parameter b

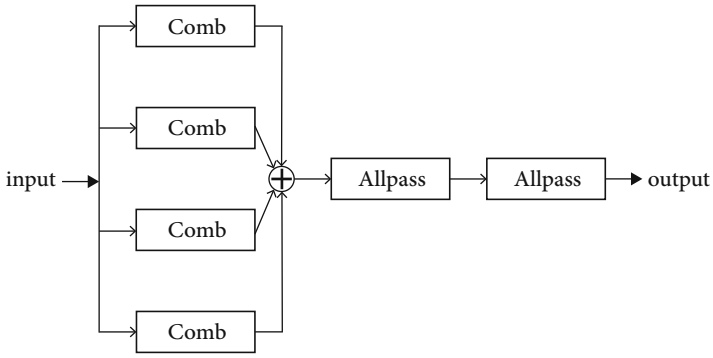


Fig. 9.9 Block diagram of Schroeder's classical reverb, comprising four comb filters in parallel followed by two all-pass filters in series

mixing parameter, the louder the output of that particular filter will be featured in the combined output. The filter is shown in Fig. 9.8.

Similarly, let $H_5(z)$ and $H_6(z)$ denote the transfer functions of the two all-pass filters, whose transfer functions are given by

$$H_k(z) = \frac{-a_k + z^{-D_k}}{1 - a_k z^{-D_k}}, \text{ for } k = 5, 6. \tag{9.16}$$

Since the four comb filters are applied in parallel, it means that the combined transfer function is

$$H_1(z) + H_2(z) + H_3(z) + H_4(z), \tag{9.17}$$

and since the output of this combined filter is then fed through the two all-pass filters, which are connected in series, the transfer function of the entire Schroeder's reverb is

$$H(z) = (H_1(z) + H_2(z) + H_3(z) + H_4(z))\, H_5(z) H_6(z). \tag{9.18}$$

The entire Schroeder's reverb can be seen in Fig. 9.9. It can be thought of as one big, linear, time-invariant filter. In Table 9.1 some suggested parameters for the four comb filters and the two all-pass filters are listed for a sampling frequency

Table 9.1 Some suggested parameters for the Schroeder's reverb for a sampling frequency of 44.1 kHz

Parameter	$H_1(z)$	$H_2(z)$	$H_3(z)$	$H_4(z)$	$H_5(z)$	$H_6(z)$
D_k	1553	1613	1493	1153	223	443
a_k	0.85	0.85	0.75	0.75	0.7	0.7
b_k	0.30	0.25	0.25	0.20	N/A	N/A

of 44.1 kHz. These can be used as a starting point for experimentation. Note that the delays of the comb and all-pass filters should be chosen such that they are mutually prime numbers, i.e., so that they have no common factors.

9.5 Modifications

Some improvements and modifications to Schroeder's original design have been proposed over the years. For example, Moorer suggested to insert a low-pass filter in the feedback loop of the plain reverberator, which results in a transfer function of

$$H(z) = \frac{1}{1 - z^{-D}G(z)}, \tag{9.19}$$

with $G(z)$ being the transfer function of the low-pass filter. This results in a frequency-dependent reverberation time, with the echos being more and more spread out over time and a mellower and more diffuse reverberation in the end, something that is perceived as being more natural and less metallic than the original plain reverberator. Any kind of low-pass filter can in principle be used and both FIR and IIR filters could be used. A simple solution could be a one-pole low-pass filter having the transfer function

$$G(z) = \frac{(1-a)z^{-1}}{1 - az^{-1}}, \tag{9.20}$$

where $0 < a < 1$. This is a low-pass filter with 0 dB gain at $\omega = 0$. This filter has the difference equation

$$y_n = (1-a)x_{n-1} + ay_{n-1}. \tag{9.21}$$

This filter is then inserted in the feedback loop of the plain reverberator. The resulting filter is shown in Fig. 9.10. Another simple modification is to mix the output of the reverberator with the clean, dry signal to give a better control of the amount of reverberation. Let x_n be the input and y_n be the output of the reverberator. Then the input and the output can be combined as $(1-\alpha)x_n + \alpha y_n$, where α then controls the amount of reverberation. With $\alpha = 0$ no reverberation is used, while

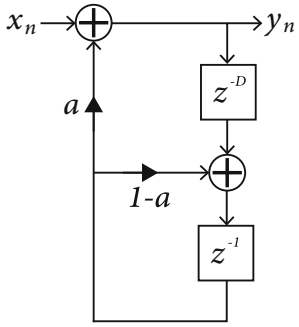


Fig. 9.10 Moorer's modified comb filter with a low-pass filter in the feedback path of the comb filter

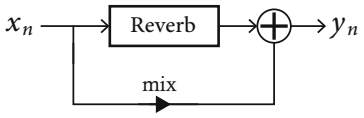


Fig. 9.11 A simple modification to Schroeder's reverb wherein the amount of reverb can be controlled via a mix parameter

only the output of the reverberator is used when $\alpha = 1$. A simpler solution yet is to mix the signals as $x_n + \alpha y_n$. This approach is illustrated in Fig. 9.11.

Different numbers of comb and all-pass filters have also been suggested and investigated, and more complicated solutions have been proposed with the so-called feedback delay networks. So far, we have only discussed how to produce artificial reverberation for a single output, i.e., for a single speaker. To generalize Schroeder's reverb and similar reverbs to multiple output channels, one can use a so-called mixing matrix on the output of the reverberator, which might also have to be rearranged slightly.

In closing, it should be mentioned that a big challenge in designing reverbs is how to adjust the parameters of the constituent parts, like the comb and all-pass filters in Schroeder's reverb, in a way that is meaningful for the user. In most cases, except for expert users, it would not make sense to give the users access to the parameters of the filters directly. Rather, they should interact and control the effect via physically and/or perceptually meaningful parameters, such as the size and shape of a room, the materials of the walls, and so forth.

For more on artificial reverberation and its history, we refer the interested reader to the papers [22, 23].

9.6 Stereo Panning

Having discussed how to artificially create the perception of a sound being played in a reverberant acoustical environment, we will now discuss some simple panning techniques for artificially creating the spatial attributes of sounds from mono recordings. The simplest instance of this is in the case of stereophonic playback, where only two loudspeakers are used. Panning techniques are used in many connections, including in music production where different instruments are panned to different angles to make them spatially distinct from each other. It should be noted that spatial sound is an entire research field in its own right and that we here consider only the simplest techniques. The following is based on [24]. The most common form of panning, which is frequently used in music production and is present in any half-decent mixer, is amplitude panning. In amplitude panning, the original mono sound x_n to be panned is modified for each loudspeaker i as

$$x_n^{(i)} = g_i x_n, \quad \text{for} \quad i = 1, \ldots, I, \tag{9.22}$$

where $x_n^{(i)}$ is the loudspeaker signal for loudspeaker i, I is the total number of loudspeakers, and g_i is a real, non-negative gain factor. The task at hand is then to choose g_i for all i so that a virtual source is perceived from a desired location or direction. Amplitude panning is illustrated in Fig. 9.12.

The most commonly used setup for music production and reproduction is stereo, where two loudspeakers are placed directly in front of the listener, one at an angle of θ_0 and the other at $-\theta_0$, where 0 is the front direction. A typical aperture of the loudspeakers is 60°. For the stereophonic case, we have that $I = 2$ and g_1 denote the gain of loudspeaker 1 (left) and g_2 the gain of loudspeaker 2 (right). There are then a number of different ways to choose g_1 and g_2, which are called *panning laws*. One such law is the sine law where the desired perceived *azimuth*, $\hat{\theta}$, of the virtual source is related to the gains g_1 and g_2 of loudspeaker 1 and 2, respectively, as

$$\frac{\sin\hat{\theta}}{\sin\theta_0} = \frac{g_1 - g_2}{g_1 + g_2}. \tag{9.23}$$

This is shown in Fig. 9.13. The sine law follows from modeling the propagation path from the two loudspeakers to the listener as a straight line. The elevation is assumed to be zero, meaning that the loudspeakers, ears, and the virtual source are all in the same plane. Although easy to use, the model has a number of problems. It is only

Fig. 9.12 In amplitude panning, the signal x_n is modified by a gain g_i for the ith loudspeaker to produce the loudspeaker signal, $x_n^{(i)}$

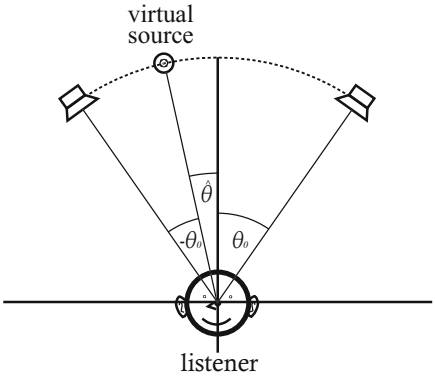


Fig. 9.13 The principle behind stereo panning. Loudspeakers located at $\pm\theta_0$ play signals to produce a perceived azimuth $\hat{\theta}$

valid for low frequencies (i.e., below 500 Hz) and the aforementioned straight line penetrates the head. It should also be noted that the angle cannot extend beyond the aperture of the loudspeakers, i.e., $|\hat{\theta}| < \theta_0$ without causing distortion.

A more sophisticated model that features a head model uses curved lines to model the propagation of the sound from the loudspeakers to the ears, more specifically, a curved line from the contralateral loudspeaker around the head to the ear. The resulting panning law, which is called the tangent law, is given by

$$\frac{\tan\hat{\theta}}{\tan\theta_0} = \frac{g_1 - g_2}{g_1 + g_2}, \tag{9.24}$$

but it suffers from some of the same problems as the sine law.

Both the sine law in (9.23) and the tangent law in (9.24) relate the angles to the difference and sum of the two gains for the stereophonic setup. As such, they form one equation but feature two unknowns, the gains, g_1 and g_2. Thus, there does not exist a unique solution and additional constraints must be imposed before we can determine a set of unique gains. A natural constraint to impose is that the loudness of the virtual source is preserved regardless of the desired direction. This can be achieved with the following:

$$\left(\sum_{i=1}^{I} g_i^p\right)^{1/p} = 1, \tag{9.25}$$

where p can be chosen differently to accommodate different room acoustics. Note that this expression applies for any number of loudspeakers, not just stereophonic setups. We can see that for $p = 2$, we have something reminiscent of the usual measure of power when dealing with signals. However, with a choice of $p = 1$ it

turns out that the loudness is preserved in anechoic conditions, while $p = 2$ works well in reverberant environments. For $p = 1$ we have that

$$g_1 + g_2 = 1, \tag{9.26}$$

which then ensures that we preserve the loudness. By inserting this into the tangent law (9.24), we obtain

$$\frac{\tan\hat{\theta}}{\tan\theta_0} = g_1 - g_2. \tag{9.27}$$

Hence, (9.26) and (9.27) combine to form a system of questions comprised of two equations with two unknowns, which can then be solved. It can be shown that the solutions are

$$g_1 = \frac{1}{2} + \frac{1}{2}\frac{\tan\hat{\theta}}{\tan\theta_0} \tag{9.28}$$

and

$$g_2 = \frac{1}{2} - \frac{1}{2}\frac{\tan\hat{\theta}}{\tan\theta_0}. \tag{9.29}$$

The same procedure can be followed to obtain the gains for the sine law. The result is similar, only $\tan(\cdot)$ is replaced by $\sin(\cdot)$.

The principles of stereophonic panning can easily be extended to setups with more speakers, including also surround sound systems, in a simple way, as long as we are talking about 2-D loudspeaker setups with all loudspeakers and the listener in a plane, most often in the horizontal plane. The panning law can simply be applied to the pair of loudspeakers in the complete setup that are the closest to the desired direction of the virtual source, say loudspeakers n and m, and then the gain parameters g_n and g_m can be computed using (9.23) or (9.24) in combination with (9.25). This is shown in Fig. 9.14.

An alternative to amplitude panning is phase (or time) panning, where the loudspeaker signals are obtained by modifying the original, monophonic signal as

$$x_n^{(i)} = x_{n-\tau_i}, \quad \text{for} \quad i = 1, \ldots, I, \tag{9.30}$$

where τ_i is the phase-delay for the ith loudspeaker, as also shown in Fig. 9.15. The idea behind this is that due to the speed of sound in air, the perception of the source coming from a certain direction causes a difference in time-of-arrival in the ears. In a stereophonic setup, the phase-delay can simply be set to zero for one of the loudspeakers and then it can be adjusted in the other to achieve the desired perceived direction. In practice, phase panning is not frequently used, as it produces inconsistent results. Phase and amplitude panning can also be combined, but it does not work well in setups with only a few speakers.

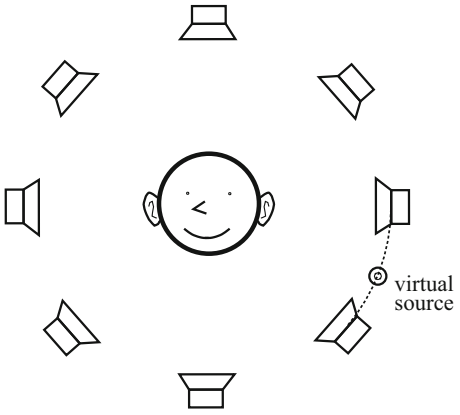


Fig. 9.14 Stereophonic panning techniques can be extended to multiple speakers by considering the pair of speakers closest to the desired direction

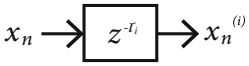


Fig. 9.15 In phase panning, the signal x_n is delayed by τ_i for the ith loudspeaker to produce the loudspeaker signal, $x_n^{(i)}$

The panning techniques discussed here aim at artificially creating the effect of a virtual source coming from a desired angle by modifying the phase and/or amplitude of the loudspeaker signals. The perception of distance, however, can be achieved in another way. The effect of a sound traveling through a certain amount of the atmosphere is that the sound is low-pass filtered, and the further the distance the sound travels, the more its higher frequencies are attenuated. Hence, to simulate the effect of distance, we can apply a simple (say, a first- or second-order) low-pass filter to the signal. To artificially create the perception of the source being played in a certain acoustic environment, where the sound not only travels directly from the source to the listener, but also is reflected off surfaces, we must use more complicated methods.

It should be mentioned that more sophisticated methods for spatial sound reproduction exist than the ones presented here (see, e.g., [25]). In particular, the techniques of ambisonics and wavefield synthesis should be mentioned, as they can produce very good results in 3-D systems with many loudspeakers. For playback on headphones, the technique of *head-related transfer function* (HRTF) can provide excellent results and is used in many applications. The principle is to capture the HRTFs from a desired point in space to the ears (actually, inside the ear at the eardrum) and then apply this HRTF to a desired sound. This is illustrated in Fig. 9.16. The HRTF takes into account the propagation of the sound through the air, the reflections from the acoustical environment, the shape of the listener's body and head, along with the properties of the outer ear.

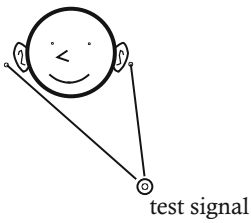


Fig. 9.16 HRTFs are measured by playing a test signal at a point in space and then measuring the pressure in the ear canal with in-ear microphones

9.7 Exercises

The following exercises are concerned with designing and implementing first the basic building blocks of classical reverbs, then a full reverb based on Schroeder's design, and finally some improvements and modifications.

Exercise 1 Build an abstraction (in Pd) for the all-pass filter and one for the comb filter. The abstractions should take the filter coefficients and the delay as input parameters. Make sure the abstractions operate on different buffers (use $0 to generate unique buffer names in the abstractions, e.g., `$0-buffer`).

Exercise 2 Use the abstractions to build the Schroeder's reverb consisting of four comb filers and two all-pass filters in series. Test it with some dry audio signals.

Exercise 3 Create an abstraction with an implementation of the tangent panning law. It should take a mono audio signal as an input as well as the desired perceived angle and should output a stereo signal.

Exercise 4 Extend Schroeder's reverb to include a low-pass filter in the feedback path of the plain reverberator, as shown in Fig. 9.10. Combine the reverb with stereo panning using the abstraction from the previous exercise.

Exercise 5 Use the stereo panning abstract to extend one of the audio effects from the previous chapter (e.g., echo, chorus, or flanger) to stereo.

Exercise 6 Derive the gain parameters for stereo panning with the tangent law with $p = 2$ in (9.25) and implement them in an abstraction in Pure Data.

Exercise 7 Build a panning effect in Pd where the perceived azimuth of the virtual source changes periodically. Use a low-frequency oscillator (LFO), much like in a chorus or flanger, to control the desired angle $\hat{\theta}$ for a loudspeaker aperture of 60° with the tangent law. The angle $\hat{\theta}$ should vary from $-\beta\theta_0$ to $\beta\theta_0$, where $0 \leq \beta \leq 1$ is a user parameter that controls the amount of panning. The user should also be able to control the frequency of the LFO.

Chapter 10
Audio Equalizers

10.1 Introduction

Equalizers are ubiquitous in audio equipment, be it a simple tone control on a stereo, a hi-fi multi-band graphic equalizer, or a parametric equalizer band in a guitar pedal. Although based on the same principles, equalizers serve different purposes depending on their context. In reproduction systems, equalizers serve to cancel any undesired filtering effect due to the involved equipment (loud speakers, amplifiers, etc.) or room acoustics. Musicians use equalizers as a means for shaping their sound, i.e., as part of a musical expression. Equalizers are built from linear time-invariant (as long as you do not touch the controls) filters. There are two kinds of equalizers: graphic and parametric. In graphic equalizers, a desired frequency response (i.e., attenuation and amplification of certain frequencies) is achieved by changing the gain at a set of fixed center frequencies. In parametric equalizers, the desired frequency response is obtained using a number of bands, each of which can be controlled by a center frequency, a bandwidth and a gain (or other similar parameters). In the following, we will cover the design of parametric equalizers although the principles can also be adapted for graphic equalizers.

10.2 Basic Principles

A parametric equalizer is built from a number of filters connected in series. Note that the designs to follow will not work properly if connected in parallel. Each of these filters are called a parametric equalizer filter and are formed as a linear combination of a notch and peak filter [7], i.e.,

$$H_{\text{eq}}(z) = G_0 H_{\text{notch}}(z) + G H_{\text{peak}}(z), \tag{10.1}$$

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_10

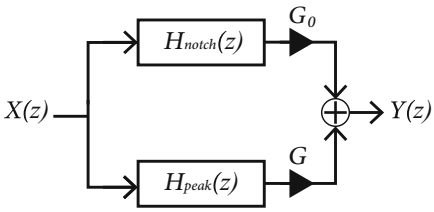


Fig. 10.1 Block diagram of the combination of peak and notch filters into a parametric equalizer filter

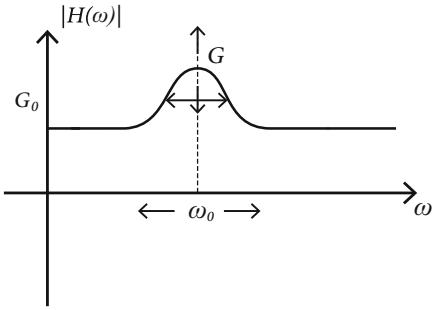


Fig. 10.2 Illustration of the magnitude response of a parametric equalizer filter where the user can adjust the gain, level, bandwidth, and center frequency

where $H_{\text{notch}}(z)$ and $H_{\text{peak}}(z)$ are the transfer functions of the notch and peak filters, respectively, and G_0 and G are their gains. This is illustrated in Fig. 10.1. G is the amplification at the center frequency of the filter relative to G_0. The latter is often chosen as a constant in equalizers and can simply be set to 1, i.e., $G_0 = 1$, and is sometimes also called the level. Boost is achieved by selecting $G > G_0$ and cut by $G < G_0$. The peak and notch filters share a number of parameters (hence parametric), namely, bandwidth $\Delta\omega$, center frequency ω_0, and G, and these can then be adjusted by the user. These digital quantities are related to their analog counterparts as

$$\omega_0 = \frac{2\pi f_0}{f_\text{s}} \quad \text{and} \quad \Delta\omega = \frac{2\pi\,\Delta f}{f_\text{s}}, \tag{10.2}$$

where f_s is the sampling frequency and f_0 and Δf are the center frequency and bandwidth (in Hz). In a graphic equalizer, these quantities are fixed for each band and the user can only change G. The center frequency and the bandwidth are related to the edges of the band, termed cutoff frequencies ω_1 and ω_2 (with $\omega_2 > \omega_1$), as

$$\omega_0 = \sqrt{\omega_1\omega_2} \quad \text{and} \quad \Delta\omega = \omega_2 - \omega_1, \tag{10.3}$$

i.e., the center frequency is the geometric mean of ω_1 and ω_2 and the bandwidth is simply the difference between the two. Note that the analog counterparts of these quantities, i.e., f_1 and f_2 are related to ω_1 and ω_2 similarly to (10.2). In Fig. 10.2, the magnitude response of a parametric equalizer filter, i.e., $H(\omega)$, and how it can be changed via the user parameters gain, level, bandwidth, and center frequency are illustrated.

10.3 Notch and Peak Filters

We will now describe the notch and peak filters in (10.1) that form the basis of the equalizer filter design in this chapter in more detail. We will start with the notch filter. It serves to attenuate, i.e., cut the signal at a certain frequency, namely, the center frequency. A notch filter having a center frequency ω_0 has a transfer function that looks as follows:

$$H_{\text{notch}}(z) = b\frac{1 - 2\cos\omega_0 z^{-1} + z^{-2}}{1 - 2b\cos\omega_0 z^{-1} + (2b-1)z^{-2}}, \tag{10.4}$$

which can be seen to be a second-order IIR filter. The quantity b, which is the gain, is given by

$$b = \frac{1}{1+\beta} \quad \text{where} \quad \beta = \frac{\sqrt{1 - G_{\text{B}}^2}}{G_{\text{B}}}\tan\frac{\Delta\omega}{2}, \tag{10.5}$$

where G_{B} is the gain at the cutoff frequencies ω_1 and ω_2. To yield 3 dB cutoff frequencies we could simply choose $G_{\text{B}}^2 = 0.5$. The above transfer function has been obtained using bilinear z-transformation from an analog equivalent and the frequencies have been pre-warped (see, e.g., [7] for more details). The notch filter results in the following difference equation:

$$y_n = bx_n - 2b\cos\omega_0 x_{n-1} + bx_{n-2} + 2b\cos\omega_0 y_{n-1} - (2b-1)y_{n-2}. \tag{10.6}$$

To implement this, we must simply choose ω_0 and compute b, which is done by selecting a bandwidth $\Delta\omega$. In Fig. 10.3 some examples of magnitude responses of notch filters are shown. These are depicted for a center frequency of $\omega_0 = \pi/2$ and for varying bandwidths with $G_{\text{B}}^2 = 0.5$.

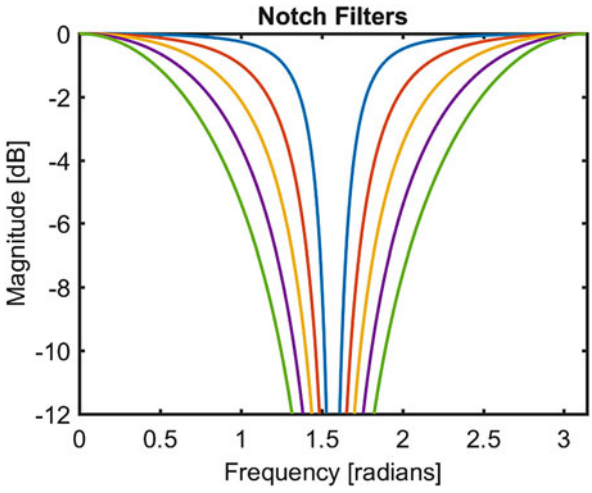


Fig. 10.3 Magnitude responses of notch filters

Next follows a description of the peak filter in (10.1). Such a peak filter boosts the signal at a certain frequency, which is the center frequency ω_0 of the filter. It has the following transfer function:

$$H_{\text{peak}}(z) = (1-b)\frac{1-z^{-2}}{1-2b\cos\omega_0 z^{-1}+(2b-1)z^{-2}}, \tag{10.7}$$

where, similarly to the notch filter, the gain b is given by

$$b = \frac{1}{1+\beta} \quad \text{where} \quad \beta = \frac{G_{\text{B}}}{\sqrt{1-G_{\text{B}}^2}}\tan\frac{\Delta\omega}{2}. \tag{10.8}$$

As can be seen, the peak filter is also a second-order IIR filter. Note that the term z^{-1} does not appear in the numerator of this filter. The parameter $\Delta\omega$ is, just as for the notch filter, the bandwidth of the peak filter at the center frequency ω_0, and G_{B} is the gain at the cutoff frequencies, which again can be chosen as $G_{\text{B}}^2 = 0.5$ to yield 3 dB cutoff frequencies. The peak filter is also obtained from an analog filter via the so-called bilinear z-transform and pre-warped frequencies. It has the following difference equation:

$$y_n = (1-b)x_n - (1-b)x_{n-2} + 2b\cos\omega_0 y_{n-1} - (2b-1)y_{n-2}, \tag{10.9}$$

which can readily be implemented once $\Delta\omega$ and ω_0 have been chosen and b computed from the above expression. In Fig. 10.4, some examples of magnitude

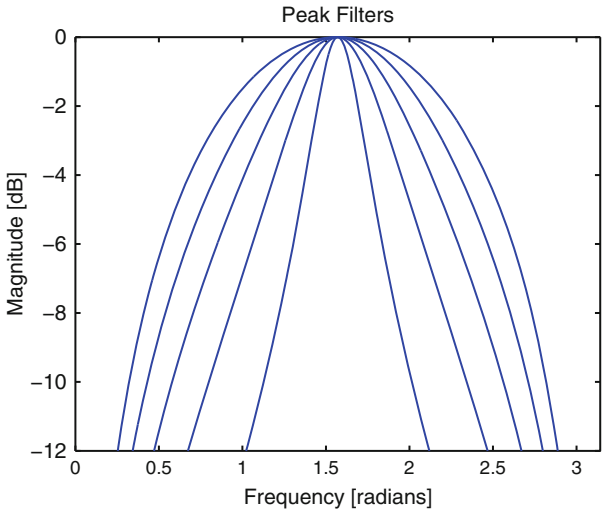


Fig. 10.4 Magnitude responses of peak filters

responses of peak filters are shown. Shown are the responses for a center frequency of $\omega_0 = \pi/2$ and for varying bandwidths with $G_B^2 = 0.5$.

10.4 Parametric Equalizer Filter

We will now combine the peak and the notch filters that we have just introduced as described in (10.1). This is done by inserting (10.4) and (10.7) into (10.1). But first, we note that we can combine the two filters as follows: Let

$$H_{\text{notch}}(z) = \frac{A(z)}{B(z)} \tag{10.10}$$

and

$$H_{\text{peak}}(z) = \frac{C(z)}{D(z)}, \tag{10.11}$$

then we can combine the two filters into a single filter as

$$H_{\text{eq}}(z) = G_0 H_{\text{notch}}(z) + G H_{\text{peak}}(z) \tag{10.12}$$

$$= G_0 \frac{A(z)}{B(z)} + G \frac{C(z)}{D(z)} \tag{10.13}$$

$$= \frac{G_0 A(z) D(z) + G C(z) B(z)}{B(z) D(z)}. \tag{10.14}$$

When combining the two filters, it is interesting to note that

$$H_{\text{notch}}(z) + H_{\text{peak}}(z) = 1, \tag{10.15}$$

meaning that the two filters with $G_0 = 1$ and $G = 1$ will do nothing to the input signal. The filters are thus said to be complementary. If we choose $G_0 = 0$ and $G = 1$ we obtain a peak filter and, conversely, by setting $G_0 = 1$ and $G = 0$ we obtain a notch filter from our parametric equalizer filter. Hence, the filter is very flexible and contains, as special cases, the peak and notch filters as well as a neutral setting that does nothing at all.

Before proceeding in combining the two filters, we must consider how to choose the gain parameter G_B. There are several ways in which this can be done. It is however useful (and common) to set if relative to G and G_0 as either the arithmetic mean or geometric mean of the two, i.e.,

$$G_B^2 = \frac{G^2 + G_0^2}{2} \tag{10.16}$$

or $G_B^2 = GG_0$. Using the aforementioned principle in combining the two filters, we obtain

$$H_{eq}(z) = \frac{\frac{G_0+G\beta}{1+\beta} - 2\frac{G_0 \cos\omega_0}{1+\beta}z^{-1} + \frac{G_0-G\beta}{1+\beta}z^{-2}}{1 - 2\frac{\cos\omega_0}{1+\beta}z^{-1} + \frac{1-\beta}{1+\beta}z^{-2}}, \tag{10.17}$$

where β is now defined as

$$\beta = \sqrt{\frac{G_B^2 - G_0^2}{G^2 - G_B^2}} \tan\left(\frac{\Delta\omega}{2}\right). \tag{10.18}$$

If $G_0^2 < G_B^2 < G^2$ we have a boost at frequency ω_0, and if we have $G^2 < G_B^2 < G_0^2$ we have a cut.

If we use the arithmetic mean in (10.16) for our definition of G_B, it can easily be seen that

$$\sqrt{\frac{G_B^2 - G_0^2}{G^2 - G_B^2}} = 1, \tag{10.19}$$

in which case we have that $\beta = \tan\left(\frac{\Delta\omega}{2}\right)$. Once we have chosen ω_0 and $\Delta\omega$, it is relatively easy to compute the filters coefficients. The resulting difference equation is given by

$$\begin{aligned} y_n = {} & \frac{G_0 + G\beta}{1+\beta}x_n - 2\frac{G_0\cos\omega_0}{1+\beta}x_{n-1} \\ & + \frac{G_0 - G\beta}{1+\beta}x_{n-2} + 2\frac{\cos\omega_0}{1+\beta}y_{n-1} - \frac{1-\beta}{1+\beta}y_{n-2}. \end{aligned} \tag{10.20}$$

In Fig. 10.5 some examples of magnitude responses of parametric equalizer filters are shown for varying gains, G. The filters shown all have a center frequency of $\omega_0 = \pi/2$, a bandwidth of $\Delta\omega = \pi/4$ with $G_B = \sqrt{GG_0}$ and $G_0 = 1$.

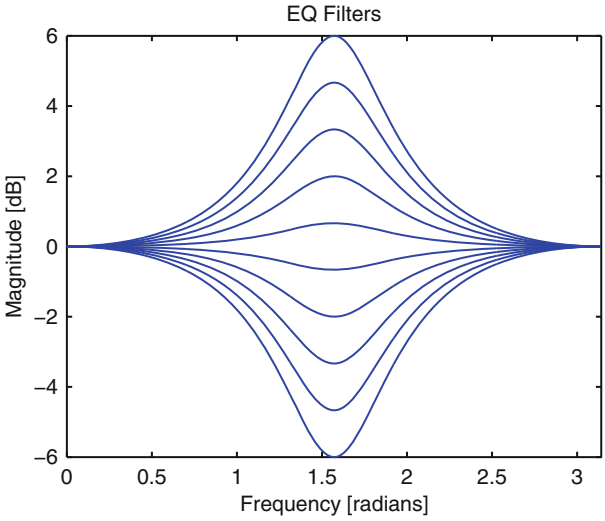


Fig. 10.5 Magnitude responses of parametric equalizer filters

10.5 Shelving Filters

To deal with very low and very high frequencies in the audio signal, it is useful to introduce so-called shelving filters. These are filters that have a flat response and adjustable gain at either low or high frequencies. The filter that is flat for low frequencies is often called a low-pass shelving filter and the one for high frequencies is called high-pass shelving filter. These can be obtained from the general parametric equalizer filter in (10.17) by replacing the center frequency, ω_0 with either 0 or π.

For the low-pass (LP) shelving filter, we insert $\omega_0 = 0$ into (10.17). Since $\cos\omega_0 = 1$, we obtain a simpler filter, i.e.,

$$H_{\mathrm{lp}}(z) = \frac{\frac{G_0+G\beta}{1+\beta} - \frac{G_0-G\beta}{1+\beta}z^{-1}}{1 - \frac{1-\beta}{1+\beta}z^{-1}}, \tag{10.21}$$

which is a first-order filter. Instead of the bandwidth $\Delta\omega$, we now have a single cutoff frequency, which we term ω_{c}, and a corresponding gain G_{C}, which replaces G_{B} but is otherwise computed the same way using (10.16). The quantity β in the equations above is then determined from this as

$$\beta = \sqrt{\frac{G_{\mathrm{C}}^2 - G_0^2}{G^2 - G_{\mathrm{C}}^2}}\tan\left(\frac{\omega_{\mathrm{c}}}{2}\right). \tag{10.22}$$

However, as before, with the definition in (10.16) for G_{C}, this reduces to something somewhat simpler that only depends on the bandwidth. It should be stressed that the general parametric equalizer filter in (10.17) can be used directly with the

substitutions described here. Hence, one does not even have to use a separate implementation of the shelving filters. For the low-pass shelving filter, the resulting difference equation is thus:

$$y_n = \frac{G_0 + G\beta}{1+\beta} x_n - \frac{G_0 - G\beta}{1+\beta} x_{n-1} + \frac{1-\beta}{1+\beta} y_{n-1}. \tag{10.23}$$

For the high-pass (HP) shelving filter, we insert $\omega_0 = \pi$ into (10.17). Noting that $\cos \omega_0 = -1$, this yields

$$H_{\text{hp}}(z) = \frac{\frac{G_0+G\beta}{1+\beta} + \frac{G_0-G\beta}{1+\beta} z^{-1}}{1 + \frac{1-\beta}{1+\beta} z^{-1}}. \tag{10.24}$$

The definition of bandwidth is slightly different in this case, however. Here, it is given by $\Delta\omega = \pi - \omega_{\text{c}}$. Noting that $\tan\left(\frac{\pi-\omega_{\text{c}}}{2}\right) = \cot\left(\frac{\omega_{\text{c}}}{2}\right)$, β turns out, for the high-pass shelving filter, to be

$$\beta = \sqrt{\frac{G_{\text{C}}^2 - G_0^2}{G^2 - G_{\text{C}}^2}} \cot\left(\frac{\omega_{\text{c}}}{2}\right). \tag{10.25}$$

Finally, the difference equation for the high-pass shelving filter is thus:

$$y_n = \frac{G_0 + G\beta}{1+\beta} x_n + \frac{G_0 - G\beta}{1+\beta} x_{n-1} - \frac{1-\beta}{1+\beta} y_{n-1}. \tag{10.26}$$

Finally, some examples of magnitude responses of shelving filters are shown in Fig. 10.6 for varying gains, G. Shown are both low-pass and high-pass filters for

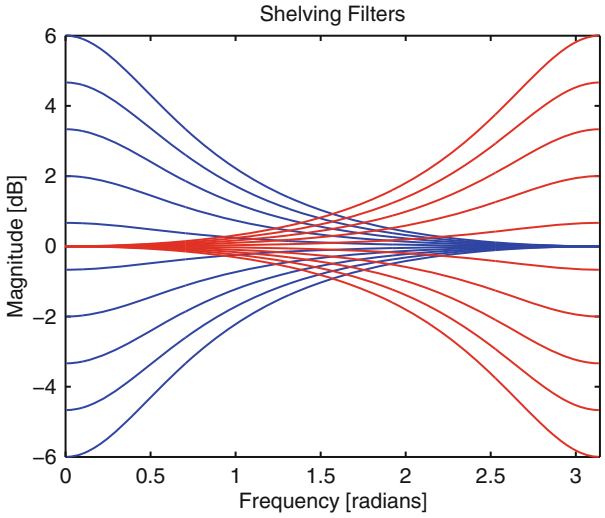


Fig. 10.6 Magnitude responses of low-pass and high-pass shelving filters

$\omega_c = \pi/4$ for the low-pass shelving filter and $\omega_c = 3\pi/4$ for the high-pass with $G_C = \sqrt{GG_0}$ and $G_0 = 1$ in both cases.

10.6 Practicalities

All the filters we have considered in this chapter are special cases of a general second-order IIR filter of the form

$$y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + a_1 y_{n-1} + a_2 y_{n-2}. \tag{10.27}$$

The only difference between the various filters is the choice of the coefficients. A block diagram of the filter structure that can be used for implementing the equalizer filters is shown in Fig. 10.7. In Table 10.1 the filter coefficients for the various filters are listed along with the appropriate definition of β. If we choose the definition in (10.16) for G_B, i.e., $G_B^2 = \frac{G^2+G_0^2}{2}$, then the various definitions of β become simpler. Moreover, with a choice of unit level, i.e., $G_0 = 1$, the expression is simplified further, and the resulting filter coefficients are listed in Table 10.2. For quick reference, the user parameters of the various filters are listed in Table 10.3. It should be noted that instead of the bandwidth, $\Delta\omega$, it is possible to use the so-called Q-factor, which is given by $Q = \frac{\omega_0}{\Delta\omega}$, something that is done in some implementations of parametric equalizers. It controls the sharpness of the filter relative to the center frequency. This means that for a low center frequency, a constant Q factor will result in a lower bandwidth than at higher frequencies. This is well in line with the properties of the human auditory system, more specifically how bandwidth is perceived as a function of center frequency. We refer the interested reader to the classical textbooks on psychoacoustics [26, 27] for more on this.

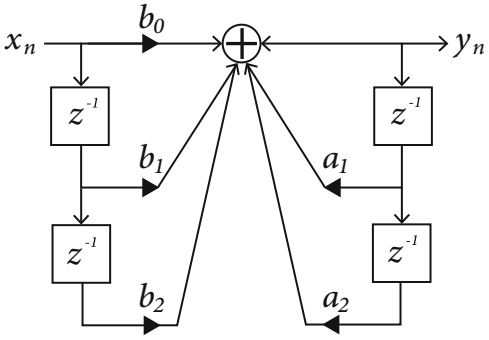


Fig. 10.7 Block diagram of second-order filter structure that can be used for implementing the audio equalizer filters

Table 10.1 General expressions for the filter coefficients for the parametric equalizer filter and the low-pass and high-pass shelving filters for implementation in (10.27)

Type	b_0	b_1	b_2	a_1	a_2	β
BP	$\frac{G_0+G\beta}{1+\beta}$	$-2\frac{G_0\cos\omega_0}{1+\beta}$	$\frac{G_0-G\beta}{1+\beta}$	$2\frac{\cos\omega_0}{1+\beta}$	$-\frac{1-\beta}{1+\beta}$	$\sqrt{\frac{G_B^2-G_0^2}{G^2-G_B^2}}\tan\left(\frac{\Delta\omega}{2}\right)$
LP	$\frac{G_0+G\beta}{1+\beta}$	$-\frac{G_0-G\beta}{1+\beta}$		$\frac{1-\beta}{1+\beta}$		$\sqrt{\frac{G_C^2-G_0^2}{G^2-G_C^2}}\tan\left(\frac{\omega_c}{2}\right)$
HP	$\frac{G_0+G\beta}{1+\beta}$	$\frac{G_0-G\beta}{1+\beta}$		$-\frac{1-\beta}{1+\beta}$		$\sqrt{\frac{G_C^2-G_0^2}{G^2-G_C^2}}\cot\left(\frac{\omega_c}{2}\right)$

Table 10.2 Simplified expressions for the filter coefficients for the parametric equalizer filter and the low-pass and high-pass shelving filters for implementation in (10.27) for the case of $G_0 = 1$ and $G_B^2 = \frac{G^2+G_0^2}{2}$

Type	b_0	b_1	b_2	a_1	a_2	β
BP	$\frac{1+G\beta}{1+\beta}$	$-2\frac{\cos\omega_0}{1+\beta}$	$\frac{1-G\beta}{1+\beta}$	$2\frac{\cos\omega_0}{1+\beta}$	$-\frac{1-\beta}{1+\beta}$	$\tan\left(\frac{\Delta\omega}{2}\right)$
LP	$\frac{1+G\beta}{1+\beta}$	$-\frac{1-G\beta}{1+\beta}$		$\frac{1-\beta}{1+\beta}$		$\tan\left(\frac{\omega_c}{2}\right)$
HP	$\frac{1+G\beta}{1+\beta}$	$\frac{1-G\beta}{1+\beta}$		$-\frac{1-\beta}{1+\beta}$		$\cot\left(\frac{\omega_c}{2}\right)$

Table 10.3 List of user parameters for each filter type with G_0 chosen a priori

Filter type	Parameters	Comment
Notch	ω_0, $\Delta\omega$, G_B	For 3 dB cutoff frequencies set $G_B^2 = 1/2$
Peak	ω_0, $\Delta\omega$, G_B	For 3 dB cutoff frequencies set $G_B^2 = 1/2$
BP	ω_0, $\Delta\omega$, G	Assuming G_B defined from G_0, G
LP	ω_c, G	Obtained by $\omega_0 = 0$ and $\Delta\omega = \omega_c$
HP	ω_c, G	Obtained by $\omega_0 = \pi$ and $\Delta\omega = \pi - \omega_c$

Note that the bandwidth parameter can be replaced by the Q-factor with $Q = \omega_0/\Delta\omega$

On a related matter, we have usually specified the parametric equalizer filter in terms of center frequency, ω_0, and bandwidth, $\Delta\omega$. These are related to the cutoff frequencies, ω_1 and ω_2 (where we assume that these have been ordered so that $\omega_1 < \omega_2$), as described in (10.3). It is possible to determine these from ω_0 and $\Delta\omega$ as follows. Since we have that $\Delta\omega = \omega_2 - \omega_1$, we also have that

$$\omega_0^2 = \omega_1\omega_2 \tag{10.28}$$

$$= \omega_1\left(\Delta\omega + \omega_1\right) \tag{10.29}$$

$$= \omega_1^2 + \Delta\omega\omega_1, \tag{10.30}$$

which means that we can find ω_1 as the values for which

$$\omega_1^2 + \Delta\omega\omega_1 - \omega_0^2 = 0, \tag{10.31}$$

i.e., ω_1 are the roots of the polynomial. Having found ω_1, we can easily find ω_2 as $\Delta\omega + \omega_1 = \omega_2$. This is useful in case we wish to be able to map bandwidth and center frequencies to cutoff frequencies and vice versa, something that might be useful when matching the cutoff frequencies of adjacent filters.

In closing, we remark that, as mentioned earlier, the filters designed from the principles described here should be connected in series. If they are connected in parallel, the filters will not yield the desired result due to the effect of the phase response of the filters. There are, however, other kinds of equalizer filters that can be connected in parallel. Moreover, there also exists FIR equalizer designs for audio, although the IIR filters presented here are typically more computational efficient.

10.7 Exercises

The exercises below are concerned with implementing the elements of equalizers in Pure Data.

Exercise 1 Implement a peak and a notch filter separately in Pd where the user can control the center frequency and the bandwidth of the two filters with sliders. Use $G_\mathrm{B} = 0.5$. It is recommended to use the `biquad~` object. Check that their frequency responses are correct.

Exercise 2 Combine the two filters in a Pd patch from the previous exercise by putting them in parallel as in (10.1). Add a controllable gain, G, on the output of the peak filter and a gain, G_0, on the output of the notch filter before their outputs are added. The two filters should share center frequency and bandwidth control. Inspect the magnitude response of the combined filter.

Exercise 3 Implement the basic building block of an audio equalizer in Pd in an abstraction, namely, a parametric equalizer filter. Use the `biquad~` object and use sliders to control the center frequency ω_0 (e.g., from 320 Hz to 3.2 kHz), the bandwidth $\Delta\omega$ (e.g., from 100 Hz to 1 kHz), and the gain G (e.g., from -10 to 10 dB). Use a level of $G_0 = 1$ and $G_B^2 = (G_0^2 + G^2)/2$. It is recommended to use the `expr` object for computing the filter coefficients from the user parameters.

Exercise 4 Implement abstractions for the low-pass and high-pass shelving filters, respectively, following the principles from Exercise 1. The cutoff frequencies and gains should be controllable by the user.

Exercise 5 Build a complete 3-band EQ from the abstractions made in the previous exercises by combining the low-pass and high-pass shelving filters with a parametric equalizer filter in a patch and test it.

Chapter 11
Dynamic Range Control

11.1 Introduction

The process of automatically adjusting the level of audio is called dynamic range control and is used in many different applications, including television, radio, music production, live music, etc. It is called dynamic range control because the ratio between the largest (i.e., loudest) and smallest (i.e., faintest) sound is in effect controlled in the process. For example, if faint sounds are amplified more than loud sounds, then the dynamic range has been decreased. This is, in fact, what a compressor does. The basic principle of dynamic range control is that the level of the audio signal is measured every so often and then a gain is applied depending on the measured level. This gain typically also depends on what the gain was previously so as to get a smooth transition from one gain to another. In this chapter, we will present some common dynamic range control algorithms, namely, the compressor, expander, limiter, and gate. We will also look into how the level can be measured in different ways and how the gain can be applied in a way that sounds good.

As with many of the other algorithms and effects in this book, there are many ways of realizing them and there is not one right way. Interestingly, the techniques presented herein, more specifically compressors, are also the basic technology that gets blamed, by music aficionados, for the so-called loudness war in modern music, where producers try to make the music sound as loud as possible at the cost of a loss of dynamic range. When compressors get blamed for ruining music, the manufacturers of course say something akin to "guns don't kill people…" and "we didn't tell anybody to turn it up all the way up," so careful with that axe, Eugene!

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_11

11.2 Basic Principles

In dynamic range control, the input signal x_n is multiplied by a time-varying gain $g_n \geq 0$ to produce the output y_n, i.e., [28]

$$y_n = g_n \cdot x_n. \tag{11.1}$$

This is shown in Fig. 11.1. The idea is then that this gain, g_n, should be varied according to the level of the input signal x_n to achieve the desired effect. In what follows, we will denote the input level as $\tilde{x}_n$ and the output level by $\tilde{y}_n$. Later we will discuss how to measure the level. The gain, g_n, is typically varied differently depending on whether the signal level is increasing, a situation that we refer to as the attack state, or whether it is decreasing, which we refer to as the release state. How long it takes for the gain to adjust to an increase or decrease is called attack or release time, respectively. In most cases, the attack time is expected to be low so the gain is turned up quickly, while it should be adjusted more slowly during release, i.e., the release time should be high. Then, based on the measured level, the desired gain is determined. Finally, smoothing is applied to the gain before it is applied to the input signal, as described in (11.1). In Fig. 11.2, the different stages involved with determining the final gain are shown. First, the level is measured from the input signal.

Regardless of how the level is computed, it is most often represented in decibel, as the difference in loudness between something having a power of 1 and 10 is perceived as the same as the difference between 10 and 100. We denote the level (and other similar quantities) in decibel as capital letters with subscript n. Hence, the level in dB $\tilde{X}_n$ is computed from the linear level $\tilde{x}_n$ as

$$\tilde{X}_n = 20 \log_{10} \tilde{x}_n \quad [\text{dB}]. \tag{11.2}$$

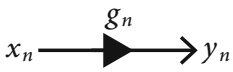


Fig. 11.1 Dynamic range control can be described as applying a time-varying gain, g_n, to the input, x_n, to obtain the output, y_n

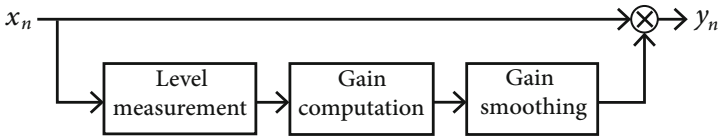


Fig. 11.2 The process of computing the gain, g_n. First, the level is measured, and then the desired gain is computed from the level. Finally, the gain is smoothed before being applied to the signal

Note that although $\tilde{x}_n$ is non-negative by construction, we here emphasize that the expression for converting from linear scale to decibel is only valid for positive values, so when this expression is used on signals, such as x_n, their absolute value, $|x_n|$, should be used. To convert back to linear scale from dB, we use the following expression:

$$\tilde{x}_n = 10^{\tilde{X}_n/20}. \tag{11.3}$$

As already explained, the principle behind dynamic range control is now to choose a linear gain g_n to be applied to the input based on x_n, as described in (11.1). Expressed in terms of level, the operation of the dynamic range control in (11.1) can be expressed as

$$\tilde{y}_n = g_n \cdot \tilde{x}_n. \tag{11.4}$$

Strictly speaking, this is only valid for a constant g_n, but the gain is normally slowly varying. Expressed in dB, the above yields

$$\tilde{Y}_n = G_n + \tilde{X}_n, \tag{11.5}$$

this means that the level of the output in dB is given by the level of the input plus the gain in dB. Recall that the relationship between the gain g_n and the gain in dB G_n is given by

$$G_n = 20 \log_{10} g_n \quad \text{and} \quad g_n = 10^{G_N/20}. \tag{11.6}$$

Hence, if the gain, g_n, is larger than 1, its value is positive in dB and results in an amplification of x_n, while a gain of 1 is equivalent to 0 dB amplification. A value of g_n below 1 results in a negative value when expressed in dB. Hence, a negative value means an attenuation of the input signal. In Fig. 11.3, the conversion from linear scale to logarithmic (i.e., decibel) in (11.6) is shown.

The behavior of all the dynamic range control algorithms considered here is such that they leave the input level unchanged when it is above (gate and expander) or below (compressor and limiter) a threshold, T, while the output level is then either increased faster or slower as a function of the input level below or above the threshold [28], i.e., for a given increase in the input level, the output level will be changed more or less above the threshold, depending on the type of dynamic range control. The output level is normally a linear function of the input level in decibel, and the slope of this line specifies what is called the compression factor. The basic functionality of the different types of dynamic range control is shown in Fig. 11.4. Mathematically, the compression factor R, which is real and positive, is given by

$$R = \frac{\tilde{X}_n - T}{\tilde{Y}_N - T}. \tag{11.7}$$

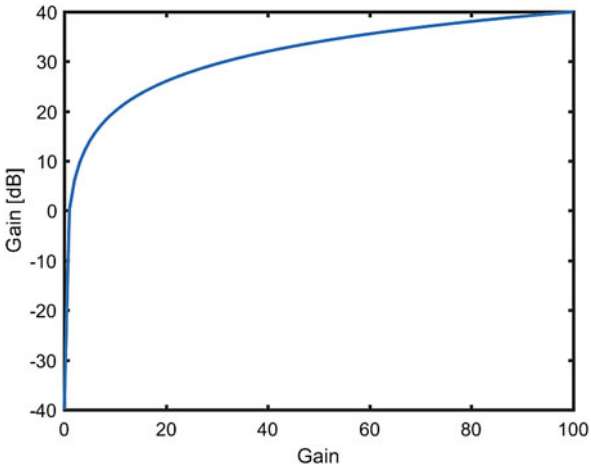


Fig. 11.3 Relationship between linear gain, g_n, and the gain in dB, G_n

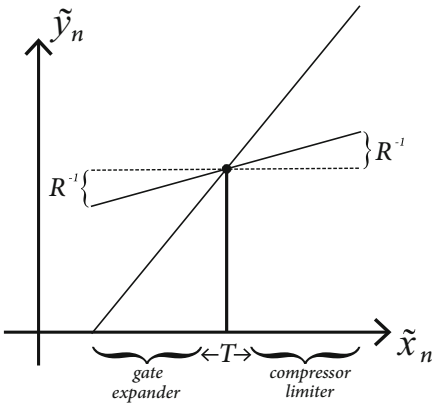


Fig. 11.4 The output level as a function of the input level in dynamic range control. The behavior of the different types of dynamic range control is determined by a threshold, T, and a compression factor, R. Compressors and limiters leave the level unchanged below the threshold, while expanders and gate leave it unchanged above the threshold

Rewriting this, we get

$$R(\tilde{Y}_n - T) = \tilde{X}_n - T \tag{11.8}$$

$$\tilde{Y}_n = T + R^{-1}\left(\tilde{X}_n - T\right), \tag{11.9}$$

which shows how the output level is related to the input level in terms of the compression factor, R, with the slope of the line being R^{-1} while it attains a value of $T - R^{-1}T$ for $\tilde{X}_n = 0$. This value is larger than zero for $R > 1$ and smaller than zero for $0 < R < 1$. Using (11.2) for $\tilde{Y}_n$ and $\tilde{X}_n$, we obtain

$$20\log_{10}\tilde{y}_n - T = R^{-1}(20\log_{10}\tilde{x}_n - T). \tag{11.10}$$

Dividing both sides by 20 and raising to the power of 10, we obtain

$$\tilde{y}_n 10^{-T/20} = \left(\tilde{x}_n 10^{-T/20}\right)^{R^{-1}}. \tag{11.11}$$

The gain, g_n, to be applied using (11.1) can now be determined as

$$g_n = \frac{\tilde{y}_n}{\tilde{x}_n} = \left(\frac{\tilde{x}_n}{10^{T/20}}\right)^{R^{-1}-1}. \tag{11.12}$$

From this we can see that, as expected, the gain is a function of the input level. The different dynamic range control algorithms are then obtained by choosing different R and T and by specifying whether the gain should be modified above or below the threshold. Note that with $R = 1$, the gain simple reduces to $g_n = 1$ meaning that the input level is not modified.

11.3 Level Measurement

As we have seen, we face the problem of first determining the level, which we denote by $\tilde{\cdot}$. There are several ways in which the level can be measured (see, e.g., [29]) and we will now go into details about a few of them, but first we should answer the question what exactly we mean by level. Here, level means some quantity that reflects how loud a signal is. Loudness is of course ultimately a perceived quality but for the purposes of dynamic range control, some simple, mathematically tractable measures are typically used. One such measure is based on the root mean squared (RMS) value of the signal, x_n, which we here denote by $\tilde{x}_n$ and is defined as

$$\tilde{x}_n = \sqrt{\frac{1}{M}\sum_{m=-M/2}^{M/2-1} x_{n-m}^2}, \tag{11.13}$$

where the signal $\tilde{x}_n$ is then the RMS value. Note how the signal squared, i.e., x_n^2, is computed from M values symmetrically around $n = 0$, i.e., before and after the time instance at which we wish to estimate the level. If the RMS value was computed for $n = 0, \ldots, M-1$, then the RMS value would lag behind the instantaneous value when the level increases or decreases and might thus fail to react in a timely fashion. A different way of writing (11.13) is

$$\tilde{x}_n^2 = \frac{1}{M}\sum_{m=-M/2}^{M/2-1} x_{n-m}^2, \tag{11.14}$$

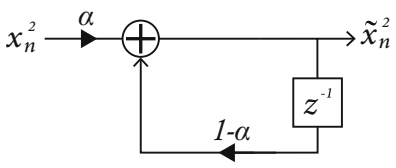


Fig. 11.5 Measuring the input level with an IIR filter. Notice how the input to the filter is the input squared

which is similar to the power of the signal, as discussed earlier. When written this way, it becomes clear that we can think of the operation of computing the RMS value as a filtering of sorts. In fact, we can see that it is just an FIR filter with coefficients $1/M$ and that the filter operates on the squared signal, x_n^2. This gives us the idea that perhaps we could also use an IIR filter on the squared signal to obtain the RMS value. Such a filter is shown in Fig. 11.5 and is given by

$$\tilde{x}_n^2 = (1-\alpha)\tilde{x}_{n-1}^2 + \alpha x_n^2. \tag{11.15}$$

Here, the constant α, which chosen to be between 0 and 1, controls how much averaging is involved with the computation of the RMS value, i.e., how slowly/fast it will react to changes in the signal. An α close to 0 will make the RMS value very slowly varying while a value close to 1 will make it adapt very quickly to changes in the RMS value, resulting in a less smooth RMS value over time.

As can be seen, (11.15) requires fewer computations per n than (11.14), and the former is often preferred for exactly this reason. Moreover, the latter also introduces a delay of $M/2$ samples which may also be a problem in some applications.

A different way of computing the level is based on peak values. In the peak method, the absolute value of the input signal is compared to the current peak value, and depending on whether the input is higher or lower than the peak value, the peak value is updated in different ways. When the absolute value of the input signal is larger than the peak value, the level measurement is in an attack state while if it is lower, it is in release state:

$$\tilde{x}_n = \begin{cases} (1-\beta_\mathrm{A})\tilde{x}_{n-1} + \beta_\mathrm{A}|x_n| & \text{for} \quad |x_n| > \tilde{x}_{n-1} \\ (1-\beta_\mathrm{R})\tilde{x}_{n-1} & \text{for} \quad |x_n| \leq \tilde{x}_{n-1} \end{cases}. \tag{11.16}$$

As can be seen, the peak value is updated recursively when the absolute value of input sample is higher than the peak value, and the updates are controlled by the parameter β_A, which is the filter coefficient in the attack state. If the absolute value of the input sample is lower than the peak value, the old peak value decays towards zero, at the speed controlled by the filter coefficient for the release state, i.e., β_R. Typically, we have that $\beta_\mathrm{A} > \beta_\mathrm{R}$ so the peak value adapts quickly to new peak values but decays slowly towards zero during release. Later, we will go more into details about choosing such parameters.

In Fig. 11.6, a signal example is shown. It is a recording of a xylophone. It can be seen that the signal exhibits very sharp onsets (when the bars are struck) after

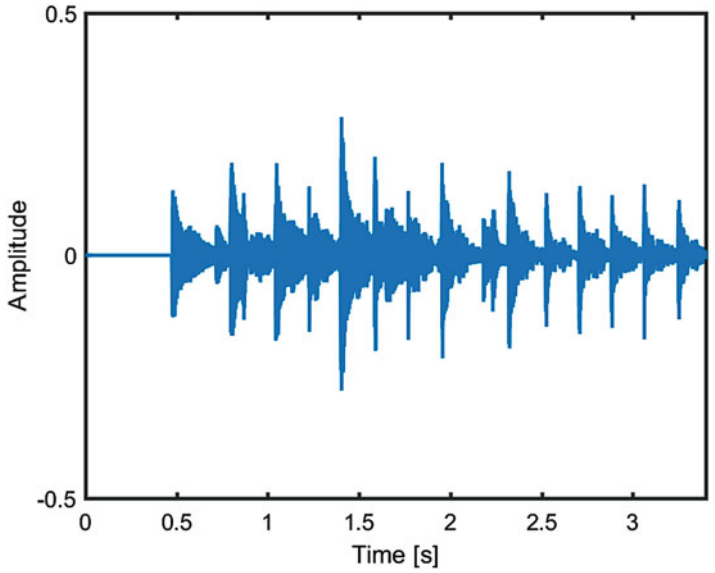


Fig. 11.6 Example of a signal, a xylophone signal, exhibiting a high dynamic range

which the notes die out. In Fig. 11.7, the level measurements obtained using the three different methods (11.13), (11.15), and (11.16) are shown. For the estimate obtained using (11.13), $M = 100$ was used, while (11.15) was used with $\alpha = 0.01$. For the peak level measurement method in (11.16), $\beta_{\mathrm{A}} = 0.1$ and $\beta_{\mathrm{R}} = 0.001$ were used to obtain the figure.

11.4 Limiter, Gate, Compressor, and Expander

Having determined the level in decibel, we can now turn our attention to how to modify the level, i.e., what g_n we should choose. Depending on how this is done, we obtain different effects. In what follows, we will take a look at several instances of dynamic range control, namely, the gate, limiter, compressor, and expander. Let us first take a look at the gate and the limiter, which are both quite simple to understand.

The idea behind the gate, or noise gate as it is also sometimes called, is that when the input level is beneath a certain level, which we will call the threshold T (which is measured in dB), the output should be set to zero which means that the gain should be $-\infty$ dB. Above the threshold, the output level should not be modified, which corresponds to a gain of 0 dB. We can write this desired behavior mathematically as

$$\tilde{Y}_n = \begin{cases} \tilde{X}_n & \text{for} \quad \tilde{X}_n \geq T \\ -\infty & \text{for} \quad \tilde{X}_n < T \end{cases} \quad \text{(GATE)}. \tag{11.17}$$

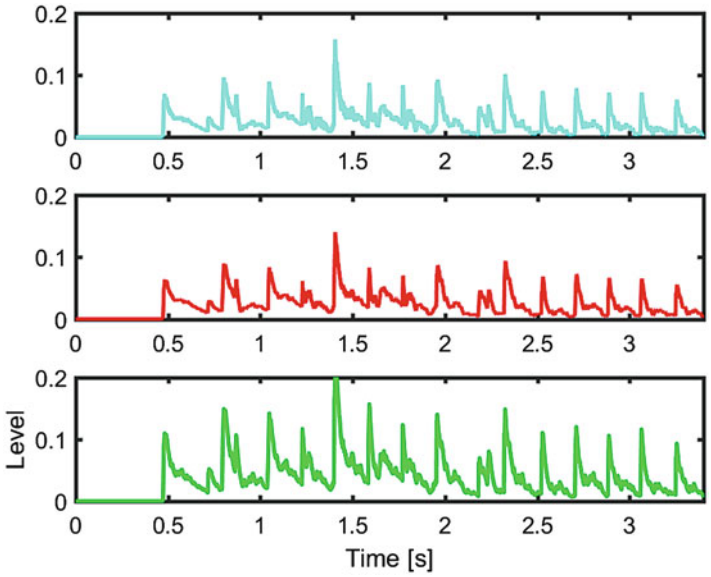


Fig. 11.7 Level measurements for the signal in Fig. 11.6. Shown is the estimated level, $\tilde{x}_n$, obtained with the methods in (11.13) (top), (11.15) (middle), and (11.16) (bottom), respectively

This means that faint sounds below the threshold will be cut. This is useful for getting rid of noise during periods of silence or for other effects, such as removing the tails of reverberation. The threshold value, T, is thus a user parameter. Described in terms of the compression factor, we can see that the desired effect can be achieved by choosing

$$R = \begin{cases} 1 & \text{for} \quad \tilde{X}_n \geq T \\ 0 & \text{for} \quad \tilde{X}_n < T \end{cases} \quad \text{(GATE)}. \tag{11.18}$$

The relation between the input level and the output level in dB for the gate is shown in Fig. 11.8. As can be seen, above the threshold, which is here -50 dB, the output level is simply equal to the input level. Below the threshold, however, the input signal is multiplied by 0, corresponding to $-\infty$ dB.

The limiter is quite similar to the gate. Instead of setting the output to zero when the level is below the threshold, it leaves the input level unchanged below the threshold while not allowing the output level to grow beyond the threshold. This is useful in many connections, including for protecting expensive equipment for overload, for limiting loudness during live performance, and as part of the mastering process in music production. The operation of the limiter can be described mathematically as follows:

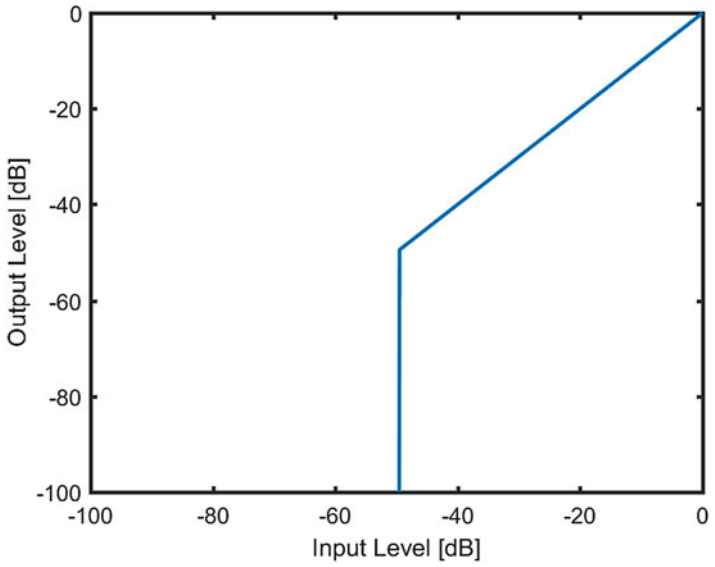


Fig. 11.8 Relationship between input and output levels for the gate with a threshold of −50 dB

$$\tilde{Y}_n = \begin{cases} \tilde{X}_n & \text{for} \quad \tilde{X}_n < T \\ T & \text{for} \quad \tilde{X}_n \geq T \end{cases} \quad \text{(LIMITER)}. \tag{11.19}$$

This corresponds to choosing a compression factor of

$$R = \begin{cases} 1 & \text{for} \quad \tilde{X}_n < T \\ \infty & \text{for} \quad \tilde{X}_n \geq T \end{cases} \quad \text{(LIMITER)}, \tag{11.20}$$

in the formula for the gain (11.12), depending on whether the signal level is above or below the threshold. The characteristic behavior of the limiter is shown in Fig. 11.9, which shows the output level as a function of the input level. As can be seen, the output level is equal to the input level below the threshold of −50 dB, but above it, the output level is kept constant, which effectively caps the output level at some desired maximum value.

The compressor and expander are closely related and are a bit more complicated than the limiter and the gate and their effects are not as obvious. The compressor leaves the input level unchanged below the threshold T. Above the threshold, the output level is modified such that the output level increases slower than the input level, but at a constant rate. As before, its behavior can be understood in terms of the compression factor, R. For a compressor, this factor must thus satisfy $R > 1$ for it to lead to the desired result. The compressor can be seen to decrease the dynamic range, i.e., it compresses the signal to a smaller dynamic range.

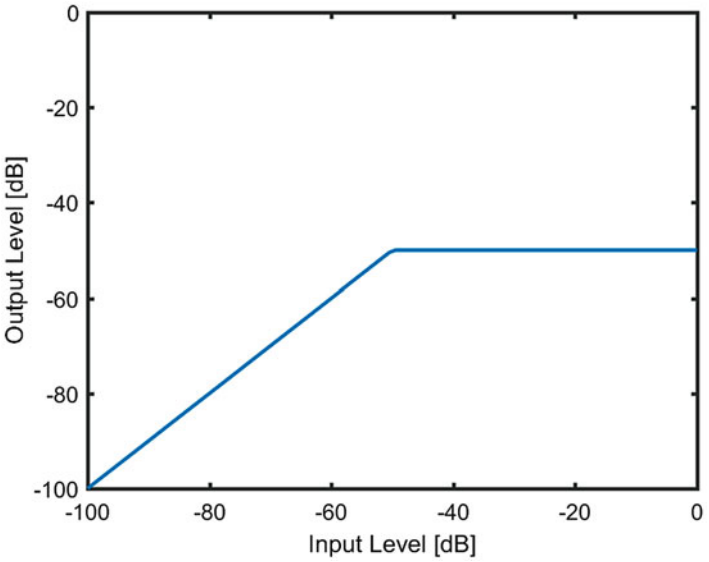


Fig. 11.9 Output level as a function of input level for the limiter, here shown with a threshold of −50 dB

The compressor can thus be specified as

$$\tilde{Y}_n = \begin{cases} \tilde{X}_n & \text{for} \quad \tilde{X}_n < T \\ T + (\tilde{X}_n - T)R^{-1} & \text{for} \quad \tilde{X}_n \geq T \end{cases} \quad \text{(COMPRESSOR)}, \tag{11.21}$$

or in terms of the compression factor as $R = 1$ for $\tilde{X}_n < T$ and $R > 1$ for $\tilde{X}_n \geq T$. The relationship between the input and output levels for the compressor is illustrated in Fig. 11.10 for a threshold of −50 dB and a compression factor of 5. From the figure, it can be seen that below the threshold, the compressor does not alter the level, but above the threshold the output level increases slower as a function of the input level.

In Fig. 11.11, the input and output levels are shown for the signal in Fig. 11.6 with compression applied with peak level measurements. For the example shown here, a threshold of −50 dB was used along with a compression factor of 2.

The expander, on the other hand, does the opposite of the compressor. Above the threshold, it leaves the input level unchanged, but below the threshold, the output level increases faster than the input level, whereby the dynamic range is increased. This means that described in terms of the compression factor, it must now be smaller than 1. The expander is obtained with the following rules for the output level:

$$\tilde{Y}_n = \begin{cases} \tilde{X}_n & \text{for} \quad \tilde{X}_n \geq T \\ T + (\tilde{X}_n - T)R^{-1} & \text{for} \quad \tilde{X}_n < T \end{cases} \quad \text{(EXPANDER)}. \tag{11.22}$$

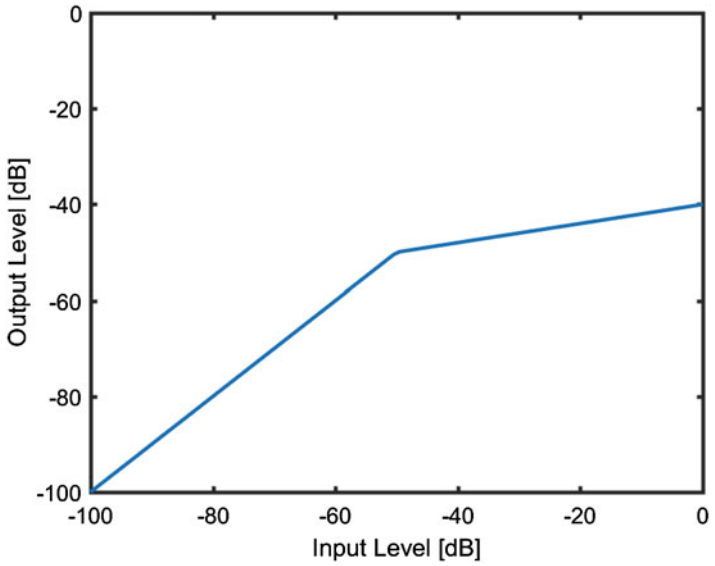


Fig. 11.10 Output level as a function of input level for the compressor, here shown with a threshold of −50 dB and a compression factor of 5

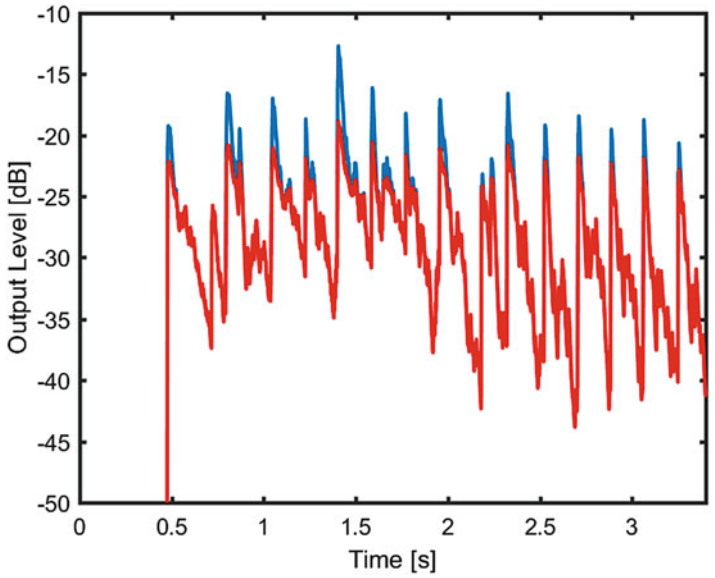


Fig. 11.11 Input and output levels for the signal in Fig. 11.6 when the compressor is applied with peak level measurements and a threshold of −25 dB and a compression factor of 2

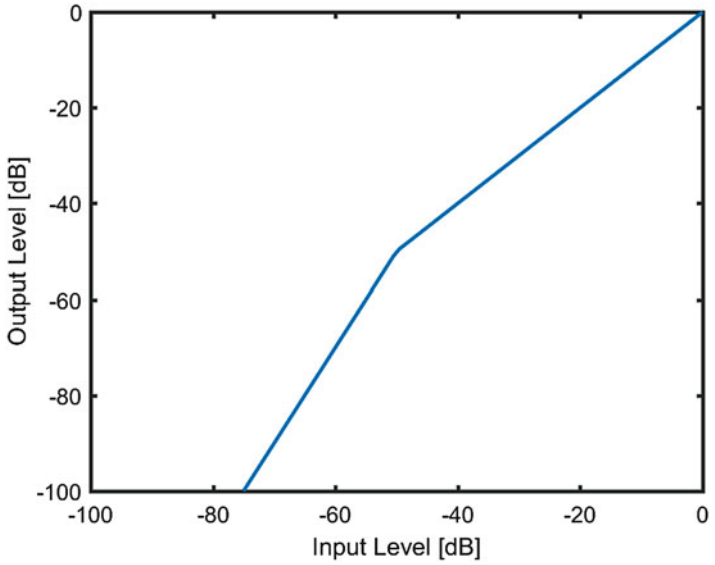


Fig. 11.12 Relationship between input and output levels for the expander with a threshold of 50 dB and a compression factor of 0.5

This can, of course, also be described in terms of the compression factor, which here must be $0 < R < 1$ for $\tilde{X}_n < T$ and $R = 1$ for $\tilde{X}_n \geq T$. In Fig. 11.12, the behavior of the expander is illustrated for a threshold of 50 dB with a compression factor of 0.5. As can be seen, the level is kept unchanged above the threshold, while the output level increases faster as a function of the input level below the threshold. This effectively expands the dynamic range of the signal.

11.5 Gain Smoothing

Before applying the gain, g_n, to the input signal, it is generally desirable to apply some smoothing to it. There are several reasons for this. The gain should not fluctuate wildly from one sample to the next. That will create annoying artifacts in the output signal. Moreover, as we explained earlier, the gain modification can be in two states when it is active, namely, attack or release, and it should behave differently in these states, and this can be achieved by applying different smoothing in the two states. Let f_n denote the raw gain computed as described in (11.12) from the level $\tilde{x}_n$. Then, the smoothed gain, g_n, can be obtained as [30]

$$g_n = (1 - \beta)g_{n-1} + \beta f_n, \tag{11.23}$$

which is a simple first-order IIR filter where the parameter $0 < \beta < 1$ (which is then a filter coefficient) determines how smoothly or fast the gain can change, and it should be set differently depending on whether the algorithm is in the attack or release state. A way to determine if the dynamic range control should be in attack or release state is by comparing f_n to the previous output value, g_{n-1}. If $f_n \geq g_{n-1}$, then it should be in attack state, while if $f_n < g_{n-1}$, it should be in release state. In the attack state, the gain should adapt quickly to the input level, which implies a high β. In the release state, the gain should decay slowly which means that β should be low. β can be chosen in accordance with a desired attack or release time τ with the following formula:

$$\beta = 1 - \exp\left(\frac{-2.2}{\tau f_s}\right), \tag{11.24}$$

where f_s is the sampling frequency and

$$\tau = \begin{cases} \tau_A & \text{when} \quad \text{ATTACK} \\ \tau_R & \text{when} \quad \text{RELEASE.} \end{cases} \tag{11.25}$$

Here, τ_A and τ_R are the attack and release times, respectively, in seconds. Typically, we choose $\tau_A \ll \tau_R$ with τ_A in the order of a few ms, while τ_R can be in the order of hundreds of ms. The expression in (11.24) can also be used for computing the filter coefficients in the peak level measurements in (11.16) and in other similar filters.

In Fig. 11.13, the linear gain, f_n, and the final smoothed version, g_n, are shown for the signal in Fig. 11.6 and the input and output levels in Fig. 11.11, i.e., when compression has been applied (with threshold −25 dB and compression factor 2) with the level being measured with the peak level measurement method. The smoothing was done with $\beta = 0.1$ in the attack state and with 0.001 in the release state.

A subtle implementation detail that can be found in many pieces of dynamic range control equipment deserves some remarks before proceeding. As the dynamic range control often renders the overall loudness changed, a makeup gain, G, is often applied aside from g_n, i.e.,

$$y_n = G \cdot g_n \cdot x_n, \tag{11.26}$$

so the output will be perceived as loud as the input, but with changed dynamic range. Sometimes this makeup gain is a user parameter of dynamic range control algorithms or it is automatically set by the system to counter the perceived effect of the dynamic range control on the loudness. Here, we will simply treat it as a user parameter for simplicity, for which reason we do not write it as a function of n.

In Fig. 11.14 the final output of a compressor run on the signal in Fig. 11.6 is shown. The compressor had a threshold of −25 dB and a compression factor of 2, with gain smoothing, peak level measurements, and a makeup gain of 3 dB.

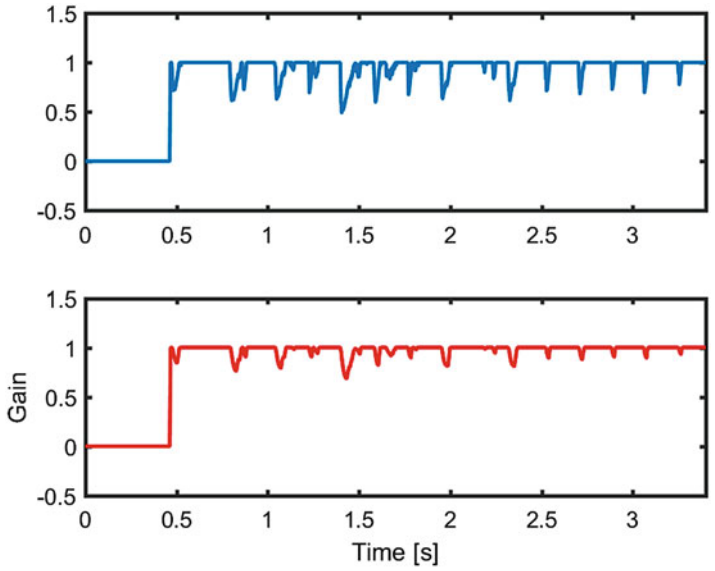


Fig. 11.13 Final smoothed gain for a compressor for the signal in Fig. 11.6. Shown are f_n (top) and g_n (bottom)

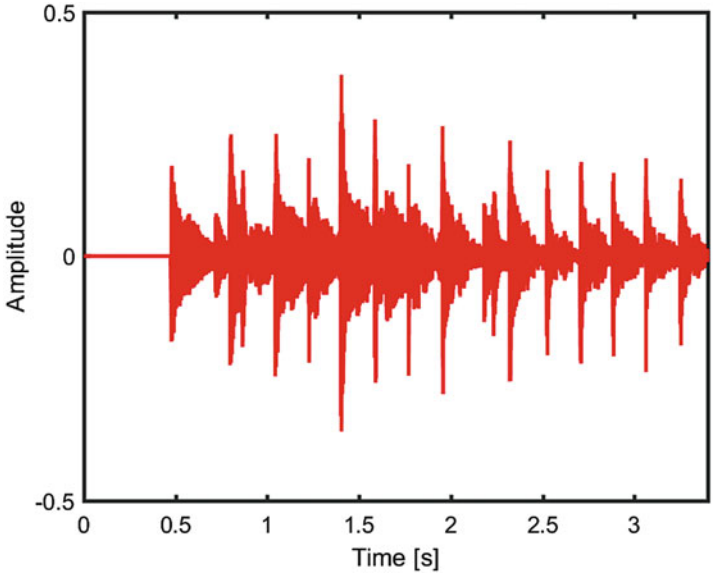


Fig. 11.14 Output of the compressor with the input signal shown in Fig. 11.6. The compressor was run with a threshold of −25 dB and a compression factor of 2, with gain smoothing, peak level measurements, and a makeup gain of 3 dB

Comparing Figs. 11.6 and 11.14, it can be seen that the fainter parts of the signal now have a higher amplitude compared to the louder parts (i.e., the onsets). This is exactly the desired effect of the compressor. Without the makeup gain, the onsets of the output would have been noticeably lower than in the input signal, and the signal as a whole would be less loud.

11.6 Summary

As we have seen, dynamic range control works by modifying the input signal by a gain g_n that depends on the input level as

$$y_n = G \cdot g_n \cdot x_n, \tag{11.27}$$

where G is a makeup gain that can be adjusted so that the overall loudness of the input is preserved. The behavior of this gain depends on whether the signal level is above or below a threshold, T. The level of the input signal is denoted $\tilde{x}_n$ and can be computed in different ways, like RMS and peak values. For the algorithms described here, the gain is given by

$$g_n = \left(\frac{\tilde{x}_n}{10^{T/20}} \right)^{R^{-1}-1}, \tag{11.28}$$

with R being the compression factor, which determines the behavior of the algorithm. By applying this when the input level in dB, $\tilde{X}_n$, is below the threshold, T, we obtain the gate and expander, while when we apply when the level is above the threshold, we get the compressor and limiters. In Table 11.1, an overview of the different algorithms is shown. Before being applied to the input signal, the gain is usually smoothed and this is done differently, depending on whether the algorithm is in the attack or release state. In the attack state, the gain should adapt quickly to increases in the input level, while in the release state, it should decay more slowly.

In closing we note that there exists many different ways of implementing dynamic range control and its different elements. Moreover, the designs presented here can be improved in several ways, like, for example, using a soft knee (as

Table 11.1 Overview of dynamic range control algorithms

Effect	Compression factor	Threshold
Gate	$R = 0$	Below
Limiter	$R = \infty$	Above
Compressor	$R > 1$	Above
Expander	$0 < R < 1$	Below

The threshold indicates whether the indicated compression factor is active above or below the threshold

opposed to the hard knee used here) in a compressor. Here knee refers to the transition at the threshold. Another improvement includes splitting the signal up into different frequencies ranges, which is done using a so-called filterbank, and then running dynamic range control, typically a compressor, independently in each band before adding the output. For more details and more on dynamic range control in general, we refer the interested reader to [7, 28–30].

11.7 Exercises

In the following exercises the different elements of dynamic range control are to be implemented in Pure Data, starting with level measurements over the simple noise gate to a compressor.

Exercise 1 Implement a RMS level measurement method of your choice in Pd and compute the level in dB. To convert to and from dB you can use the `rmstodb~` and `dbtorms~` objects.

Exercise 2 Implement a noise gate that sets the gain factor to zero when the input level is below a certain threshold. Make it possible for the user to control this threshold in dB. Use the level measurement method from the previous exercise.

Exercise 3 Similarly to the previous exercise, implement a limiter in Pd. Again, the user should be able to control the threshold in dB.

Exercise 4 Implement a compressor. It should be possible for the user to control the threshold, the compression factor, and the makeup gain.

Exercise 5 Make an abstraction in Pd that contains all the four different types of dynamic range control, namely, gate, limiter, expander, and compressor. It should be possible for the user to choose the desired type of dynamic control and the various parameters (threshold, compression factor, and makeup gain), but it should only run one at the time.

Chapter 12
Pitch Estimation

12.1 Introduction

A key property of many sounds, and in particular those that we think of as music, is the pitch. In the context of music, the American Standard Association defines the term pitch as "that attribute of auditory sensation in terms of which sounds may be ordered on a musical scale." As such, it is strictly speaking a perceptual phenomenon. It is, however, caused by physical stimuli that exhibit a certain behavior. Signals that cause the sensation of pitch are, broadly speaking, the signals that are well-described by a set of harmonically related sinusoids, meaning that their frequencies are approximately integral multiples of a fundamental frequency. Signals that have frequencies that are integral multiples of a fundamental frequency are what is commonly referred to as periodic. We have already discussed such periodic signals extensively in connection with comb filters, Fourier series, and phasors. Pitch is, of course, of interest in and of itself in music, as it is the essence of music, but it can also be used for many things in audio processing, such as auto-tuning, harmonizers, intelligent pitch shifting, and much more. Thus, no audio processing book would be complete without a chapter about pitch estimation!

In this chapter, we will present some relatively simple methods for pitch estimation. These methods are based on the principles of comb filters, correlation, and knowledge about periodic signals from Fourier series, and they are widely used in audio processing. It should be noted that the treatment herein is rather simplified and does not rely on statistical principles, although all the presented methods can (and should) be interpreted in a statistical framework. For a more in-depth treatment of the matter, we refer the interested reader to [31].

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8_12

12.2 Periodic Signals Revisited

Periodic signals have the following property for $n = 0, \ldots, N-1$:

$$x_n = x_{n-\tau}, \tag{12.1}$$

or, equivalently $x_n = x_{n+\tau}$, where x_n is a real, discrete-time signal (i.e., a sampled signal) and τ is the so-called pitch period, i.e., the smallest time interval over which the signal x_n repeats itself measured in samples. Here, it should be stressed that while x_n is defined for integers n, τ is not generally an integer. In fact, pitch is a continuous phenomenon and it is hence not accurate to restrict τ to only integer values, although this is often done. This means that we need to be able to implement fractional delays to use (12.1). Note that the assumption that (12.1) holds over $n = 0, \ldots, N-1$ implies that the characteristics of the signal x_n do not change over this interval. Such a signal is said to be stationary, and for audio signals N corresponding to anywhere between 20 and 80 ms is common. In Fig. 12.1 an example of a periodic signal is shown. It has a pitch of 150 Hz, which corresponds to a pitch period of 6.7 ms.

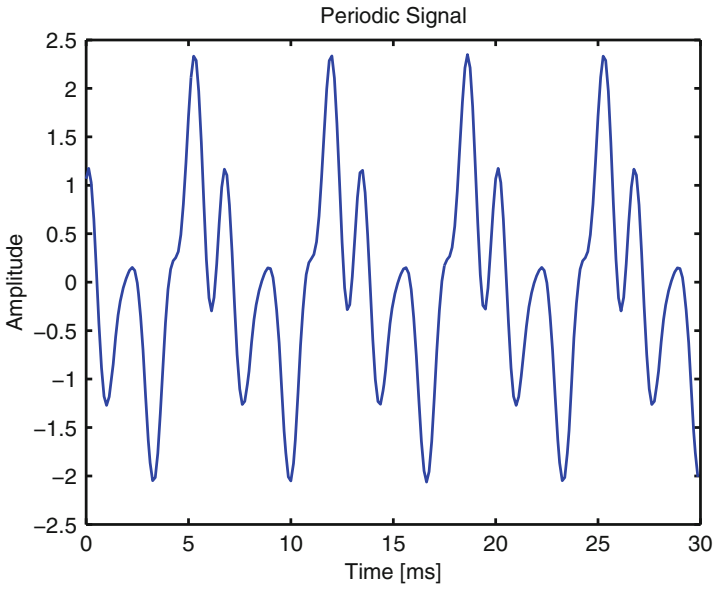


Fig. 12.1 Example of a periodic signal. This signal has a pitch of 150 Hz corresponding to a pitch period of 6.7 ms

Functions that perfectly obey (12.1) can be decomposed using a Fourier series[1] as

$$x_n = \sum_{l=1}^{L} A_l \cos(\omega_0 l n + \phi_l). \tag{12.2}$$

This is also known as the harmonic model. The quantity ω_0 is called the fundamental frequency and L is the number of harmonics, where the term harmonic refers to each term in the sum of (12.2). $A_l > 0$ and $\phi_l \in (-\pi, \pi)$ are the amplitude and the phase of the lth harmonic, respectively. The amplitude determines how dominant (or loud) the various harmonics are, while the phase can be thought of as representing a time-shift of the harmonic as we can express the argument of the cosine function in (12.2) as $\omega_0 l n + \phi_l = \omega_0 l (n - n_l)$ with $n_l = \frac{\phi_l}{\omega_0 l}$. The number of harmonics L can generally be any integer between 1 and $\frac{\pi}{\omega_0}$, and it is generally not possible to say in advance how many harmonics are going to be present. In this connection, it should be stressed that L is absolutely critical when trying to find ω_0 from a signal $x(n)$. For signals that can be expressed using (12.2), the pitch, i.e., the perceptual phenomenon, and the fundamental frequency are the same. It is interesting to note that while the harmonic is comprised as a sum of a number of individual components, these are perceived as being one object by the human auditory system. This object is the same as a musical note played by an instrument (e.g., guitar and flute) and the human voice (for the parts known as voiced speech). Moreover, the perceptually complementary property of timbre is closely related to how the amplitudes A_l change over l. To express the fundamental frequency in Hertz, denoted f_0, one must use the relation $\omega_0 = 2\pi \frac{f_0}{f_s}$, where f_s is the sampling frequency. The pitch period (in samples) and the pitch are each other's reciprocal, i.e., $\omega_0 = 2\pi \frac{1}{\tau}$, and to convert the pitch period into time measured in seconds, one must use $\frac{\tau}{f_s}$. Pitch estimation is then the art of finding ω_0 from an observed signal whose characteristics are not known in detail. The spectrum (i.e., the Fourier transform) of the signal in Fig. 12.1 is shown in Fig. 12.2. It can be seen that the signal has five harmonics and that the peaks occur at frequencies that are integral multiples of a fundamental frequency of 150 Hz.

12.3 Comb Filtering Method

An intuitive approach to finding ω_0, or, equivalently, τ is to use the relation in (12.1) directly. To obtain an estimate of τ from (12.1) we can simply subtract the right-hand side from the left-hand side, i.e., $x_n - x_{n-\tau} = 0$, and then choose the lowest τ for which this holds.[2] However, in doing so, we are faced with a number of problems.

[1] Strictly speaking, this is not a Fourier series, as, among other details, these are usually defined for x_n over a time interval corresponding to the pitch period τ.

[2] If $x_n = x_{n-\tau}$, then it is also true that $x_n = x_{n-2\tau}$, $x_n = x_{n-3\tau}$, and $x_n = x_{n-k\tau}$ for any integer k. The pitch period is the lowest possible value for which $x_n = x_{n-\tau}$ hold.

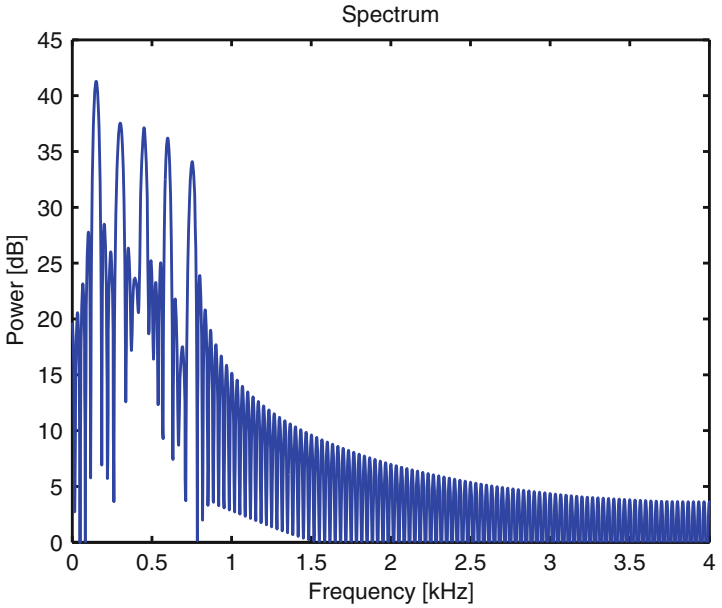


Fig. 12.2 Spectrum of a periodic signal. The signal can be seen to have five harmonics with decreasing amplitudes. The distance between the harmonics corresponds to the pitch, i.e., 150 Hz

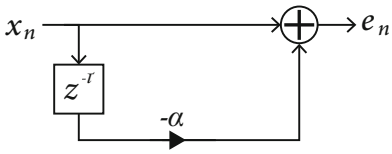


Fig. 12.3 A comb filter where the delay, τ, is adjusted to match the period of the input signal, x_n. The output, e_n is then the modeling error

First, the signal may not be perfectly periodic but may be changing slowly. Second, there will always be background noise present when dealing with real-life signals. In both cases, the relation in (12.1) is only approximate, i.e., $x_n \approx x_{n-\tau}$, so we can instead measure the non-zero difference, e_n as $x_n - x_{n-\tau}$. We call this difference the modeling error. This principle is shown in Fig. 12.3. To account for the periodic signal changing slowly, we can also include a positive scale factor a close to 1 to account for this so that $x_n \approx a x_{n-\tau}$, and define the modeling error as

$$e_n = x_n - a x_{n-\tau}. \tag{12.3}$$

Taking the z-transform of this yields

$$E(z) = X(z) - aX(z)z^{-\tau} \tag{12.4}$$

$$= X(z)(1 - az^{-\tau}). \tag{12.5}$$

From this we see that this operation can be thought of as a filtering of x_n to yield the modeling error signal e_n, and the transfer function of the filter is

$$H(z) = \frac{E(z)}{X(z)} = 1 - az^{-\tau}. \tag{12.6}$$

This is a well-known filter known as the inverse comb filter. As can be seen, this filter contains only a feedforward path and no feedback path, so it is inherently stable. Analyzing the filter as a polynomial, it has zeros located at a distance of a from the origin at angles $\frac{2\pi}{\tau}k$ for $k = 1, 2, \ldots, \tau$. To use the inverse comb filter to find an estimate of the pitch period, we must design this filter for different τs (corresponding to the fundamental frequencies in the audible range), apply it to the signal, and then somehow measure how large the modeling error, as defined in (12.3), is. A normal way of measuring the size of errors is using the mean squared error (MSE), i.e.,[3]

$$J(\tau) = \frac{1}{N - \tau} \sum_{n=\tau}^{N-1} e_n^2. \tag{12.7}$$

A function, like $J(\cdot)$ in (12.7), which we use to measure the goodness (or badness) of something, is generally referred to as a cost function. This principle is illustrated in Fig. 12.4 This cost function is a function of τ since we will get different errors for different τs. We then pick as our estimate, the τ for which (12.7) is the minimum, denoted $\hat{\tau}$. We write this as[4]

$$\hat{\tau} = \arg\min_{\tau} J(\tau), \tag{12.8}$$

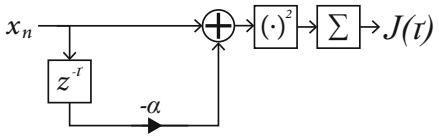


Fig. 12.4 The cost function $J(\tau)$ is computed from the output of the comb filter where the delay, τ, is adjusted to match the period of the input signal, x_n. The error signal, e_n, is squared and added up to compute the cost for a particular value of τ

[3] We have here needed to define the summation range to account for x_n only being defined for $n = 0, 1, \ldots, N - 1$.

[4] The notation means the following: $\max_x f(x)$ denotes the maximum value $f^\star$ of the function $f(x)$ over all possible x. $\arg\max_x f(x)$ is then the value $x^\star$ of x for which $f(x) = f^\star$. So, $f^\star = \max_x f(x)$ and $x^\star = \arg\max_x f(x)$.

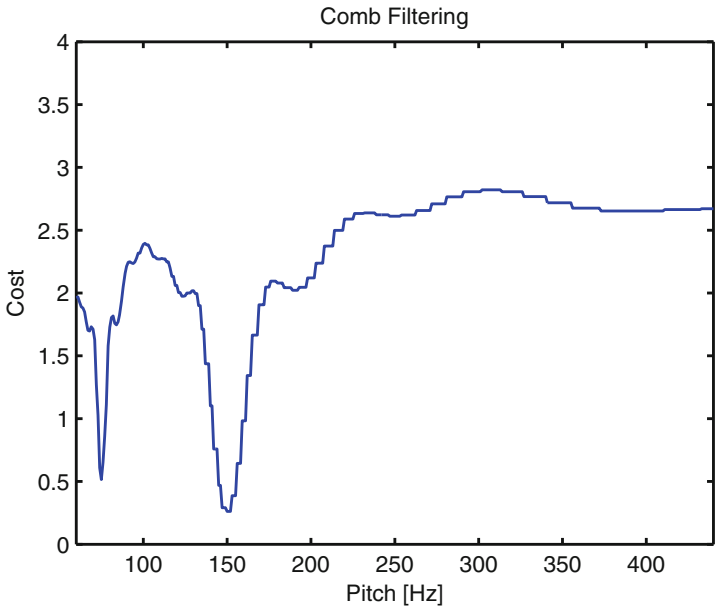


Fig. 12.5 Cost function in (12.7) for the comb filtering method as a function of the pitch for the signal in Fig. 12.1. The pitch estimate is obtained by finding the smallest value of the cost function

which is what we call an *estimator*. The quantity $\hat{\tau}$ is called an *estimate* and the process of finding it is called *estimation*. Note that we generally denote estimates as $\hat{\cdot}$. Using (12.3), we can also express this as

$$\hat{\tau} = \arg\min_{\tau} \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} (x_n - a x_{n-\tau})^2. \tag{12.9}$$

Note that a suitable range over which to compute (12.7) must be chosen. For speech, this would be τs corresponding to fundamental frequencies from 60 to 440 Hz. For a sampling frequency, f_s, of 8 kHz (which is common for speech) this would correspond to τs going from 18 samples to 133 samples. We have until now ignored the issue of which filter coefficient a to use. It is actually possible to find an optimal $a > 0$ in sense of the MSE, but in this connection, it is not that critical, and one can simply choose a to be close to 1 or even 1. The method that we have here presented is known as the comb filtering method for pitch estimation [32]. An obvious problem with this approach is that to directly compute (12.7), we are restricted to using only integer τ. To mitigate this, we would have to include a method for fractional delays in our estimator. In Fig. 12.5 the cost function in (12.7) is shown computed for different pitches (i.e., different τ) for the signal in Fig. 12.1.

12.4 Auto-Correlation and Related Methods

Perhaps the most universally applied principle for pitch estimation is the so-called auto-correlation method. We will here derive it based on the comb filtering approach as follows. Suppose that the signal is perfectly periodic, so that $a = 1$ in (12.3). In that case, we have that we should measure the modeling error for different τ as

$$e_n = x_n - x_{n-\tau}. \tag{12.10}$$

Inserting this expression into the definition of the MSE in (12.7), we obtain

$$J(\tau) = \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} e_n^2 \tag{12.11}$$

$$= \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} (x_n - x_{n-\tau})^2. \tag{12.12}$$

Writing this out, we obtain

$$J(\tau) = \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} (x_n - x_{n-\tau})^2 \tag{12.13}$$

$$= \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_n^2 + \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_{n-\tau}^2 - 2\frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_n x_{n-\tau}. \tag{12.14}$$

From this, we can make a number of observations. The first term, $\frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_n^2$, is the same as the power (or variance) of the signal x_n and does not depend on τ, i.e., it is constant. The second term, $\frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_{n-\tau}^2$, which is the power of the signal $x_{n-\tau}$, does appear at first to depend on τ, but since we have assumed that the signal is stationary, this should be equivalent to the first term and is, hence, also constant. So, the only part that actually changes with τ is this:

$$R(\tau) = \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_n x_{n-\tau}. \tag{12.15}$$

This quantity is known in signal processing terms as the *auto-correlation function*. It is a function of τ, which is commonly referred to as the *lag* in this context, and from (12.14) it can be seen to measure the extent to which x_n and $x_{n-\tau}$ are similar. The principle behind computing the auto-correlation function, $R(\tau)$, is illustrated in Fig. 12.6, which shows that it can be thought of as running a comb filter for different delay τ to obtain $R(\tau)$. In Fig. 12.7, the auto-correlation function is shown for the signal in Fig. 12.1. The periodicity of that signal can be seen

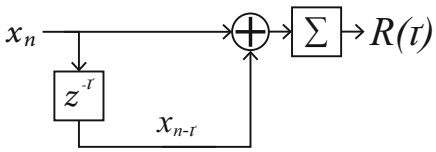


Fig. 12.6 Auto-correlation function computed using a comb filter

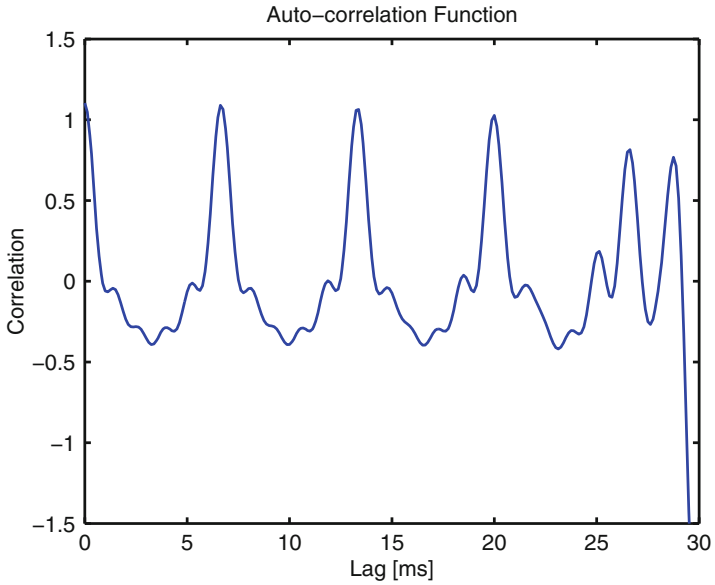


Fig. 12.7 Auto-correlation function for the signal in Fig. 12.1. The peaks indicate lags for which the signal resembles itself, i.e., where it exhibits high correlation

from the auto-correlation function also being periodic. For $\tau = 0$ we get that $R(0) = \frac{1}{N-\tau}\sum_{n=\tau}^{N-1} x_n^2$, which is the power of the signal. If the signal is perfectly periodic with period τ, then $R(\tau) = R(0)$. Moreover, it can easily be shown that $R(\tau) \leq R(0)$ for all τ. Hence, the highest possible value of $R(\tau)$ that we can hope to obtain is the same as $R(0)$. Since we would like the difference $x_n - x_{n-\tau}$ to be small, it follows from (12.14) that we should make $R(\tau)$ as large as possible. This idea leads to the following estimator:

$$\hat{\tau} = \arg\max_{\tau} R(\tau) \tag{12.16}$$

$$= \arg\max_{\tau} \frac{1}{N-\tau} \sum_{n=\tau}^{N-1} x_n x_{n-\tau}. \tag{12.17}$$

This estimator is known as the auto-correlation method [33, 34]. In Fig. 12.8 the cost function for the auto-correlation method is shown as a function of the pitch (in Hertz) for the periodic signal in Fig. 12.1. It suffers from many of the same problems

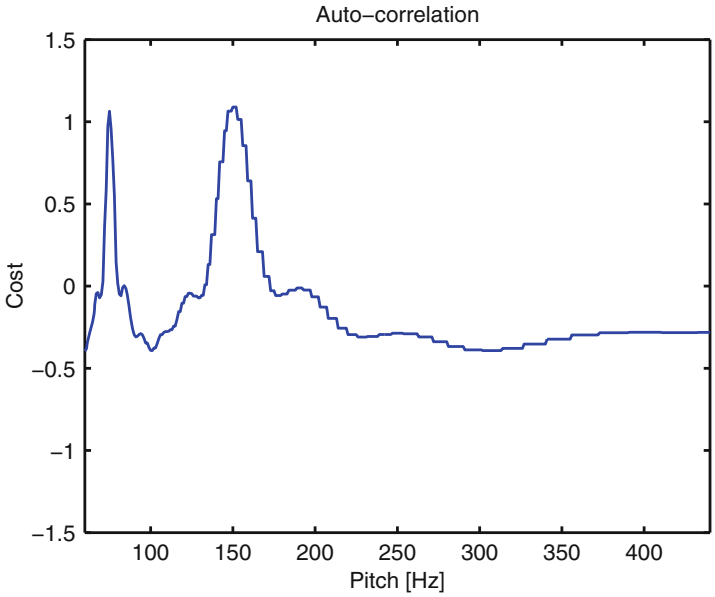


Fig. 12.8 Cost function for the auto-correlation method, i.e. (12.15), as a function of the pitch for the signal in Fig. 12.1. Using this method, the pitch estimate is obtained by finding the highest peak

as the comb filtering approach, which is no surprise since it is based on the same principle (and they are identical in certain special cases). Despite this, it is the most commonly used principle for pitch estimation, and many variations of this method have been introduced throughout the years, many of which basically boil down to modifying (12.12) to different measures of the goodness of fit, which more generally can be written as

$$J(\tau) = \left(\frac{1}{N-\tau} \sum_{n=\tau}^{N-1} (x_n - x_{n-\tau})^p \right)^{\frac{1}{p}}. \tag{12.18}$$

For example, the so-called average magnitude difference function (AMDF) method is obtained from this by setting $p = 1$. Also, various summation limits (e.g., ones that do not depend on τ or use the same number of terms for all computations) and normalization procedures have been considered, but they are all based on the same fundamental principle.

12.5 Harmonic Summation

The next (and final) approach to pitch estimation is based on the Fourier transform of the model in (12.2). The Fourier transform of a signal $x(n)$ over $n = 0, 1, \ldots, N-1$ is defined as

$$X(\omega) = \sum_{n=0}^{N-1} x_n e^{-j\omega n}, \tag{12.19}$$

for $0 \leq \omega \leq 2\pi$. The quantity $|X(\omega)|^2$ is known as the power spectrum. It should be noted that even though the signal $x(n)$ is a discrete function of n, $X(\omega)$ is a complex, continuous function of ω, but in practice we often compute it for a discrete set of frequencies using the fast Fourier transform (FFT). Let us assume that the signal in (12.19) is not perfectly periodic. In that case, we might think of fitting the model in (12.2) to $x(n)$ and choosing the fundamental frequency ω_0 that best fits. In that regard, we can think of (12.2) as a model and we can subtract the right-hand side of (12.2) from the left-hand side and measure the difference, i.e.,

$$e_n = x_n - \sum_{l=1}^{L} A_l \cos(\omega_0 l n + \phi_l). \tag{12.20}$$

We can then apply the MSE , as defined in (12.7), to measure the goodness of our model, i.e.,

$$J(\omega_0) = \sum_{n=0}^{N-1} \left(x_n - \sum_{l=1}^{L} A_l \cos(\omega_0 l n + \phi_l) \right)^2. \tag{12.21}$$

Then, it can be shown that it does not matter whether we measure the MSE on $e(n)$ or its Fourier transform $E(\omega)$, i.e.,

$$J(\omega_0) = \sum_{n=0}^{N-1} e_n^2 = \frac{1}{2\pi} \int_0^{2\pi} |E(\omega)|^2 d\omega. \tag{12.22}$$

It can then be shown, that for large N, this MSE is given by

$$J(\omega_0) = \frac{1}{2\pi} \int_0^{2\pi} |X(\omega)|^2 d\omega - \frac{2}{N} \sum_{l=1}^{L} |X(\omega_0 l)|^2. \tag{12.23}$$

The first term, i.e., $\frac{1}{2\pi}\int_0^{2\pi}|X(\omega)|^2 d\omega$, is just the power of the signal $x(n)$ computed from the Fourier transform, and is constant. The second term, i.e.,

$$\frac{2}{N}\sum_{l=1}^{L}|X(\omega_0 l)|^2 \tag{12.24}$$

is the sum of the power of the individual harmonics, i.e., $\frac{A_l^2}{2}$, when it is computed for the true fundamental frequency. To minimize the MSE in (12.23), it can be observed that we must maximize $\frac{1}{N}\sum_{l=1}^{L}|X(\omega_0 l)|^2$ as it is subtracted from the first term. The basic idea behind the harmonic summation method [35] is that for a noise free, infinitely long, and perfectly periodic signal, we have that

$$\frac{1}{N}|X(\omega)|^2 \approx \begin{cases} \frac{A_l^2}{2} & \text{for } \omega = \omega_0 l \\ 0 & \text{for } \omega \neq \omega_0 l \end{cases}. \tag{12.25}$$

Hence, by measuring $\sum_{l=1}^{L}|X(\omega_0 l)|^2$ for different ω_0, one can obtain an estimate of the fundamental frequency by picking the one for which the sum is the maximum, i.e.,

$$\hat{\omega}_0 = \arg\max_{\omega_0}\sum_{l=1}^{L}|X(\omega_0 l)|^2, \tag{12.26}$$

where we must then search over the audible range of ω_0s that obey $\omega_0 \leq \frac{\pi}{L}$. This estimator is the harmonic summation method. The cost function used in this method is shown in Fig. 12.9 for the periodic signal in Fig. 12.1. The pitch estimate is obtained by locating the highest peak in the cost function. It should be noted that the number of harmonics, L, must be known for this method to work. The determination of L is, though, a difficult problem known as model order estimation or model selection. Some simple heuristics can often be applied in practice, though, by, for example, counting the number of peaks above a threshold in the spectrum.

One can obtain another method from (12.25). Since $\frac{1}{N}|X(\omega)|^2$ is only non-zero for frequencies equal to those of the harmonics, a multiplication of the spectrum evaluated for a set of candidate fundamental frequencies is only non-zero for the true fundamental frequency. This principle can be stated as

$$\hat{\omega}_0 = \arg\max_{\omega_0}\prod_{l=1}^{L}|X(\omega_0 l)|^2, \tag{12.27}$$

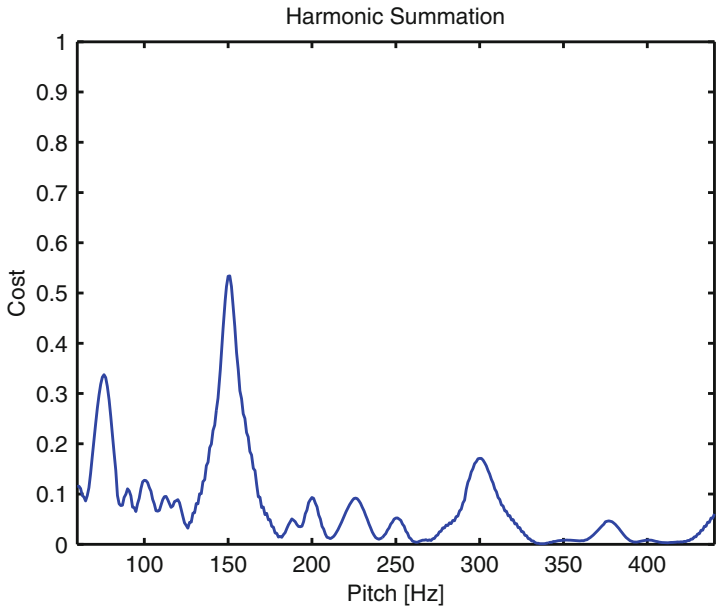


Fig. 12.9 Cost function for the harmonic summation method, i.e., for the estimator in (12.26), as a function of the pitch for the signal in Fig. 12.1. Using this method, the pitch estimate is obtained by finding the highest peak

we refer to this as the harmonic product method [35]. Taking the logarithm,[5] it can also be stated as

$$\hat{\omega}_0 = \arg\max_{\omega_0} \ln \prod_{l=1}^{L} |X(\omega_0 l)|^2 = \arg\max_{\omega_0} \sum_{l=1}^{L} \ln |X(\omega_0 l)|^2, \tag{12.28}$$

which is similar to the harmonic summation method, only it operates on the log-power spectrum. An advantage of using (12.28) over (12.26) is that it is possible to use the former for also finding the number of harmonics L. It is, however, extremely sensitive to deviations in the frequencies of the harmonics from integral multiples of the fundamental frequency. An example of a typical cost function for the harmonic product method is shown in Fig. 12.10, as before for the periodic signal in Fig. 12.1. As can be seen, there is a very sharp peak at the 150 Hz.

[5] We should not really do this as we have assumed that $|X(\omega)|$ is zero for some values.

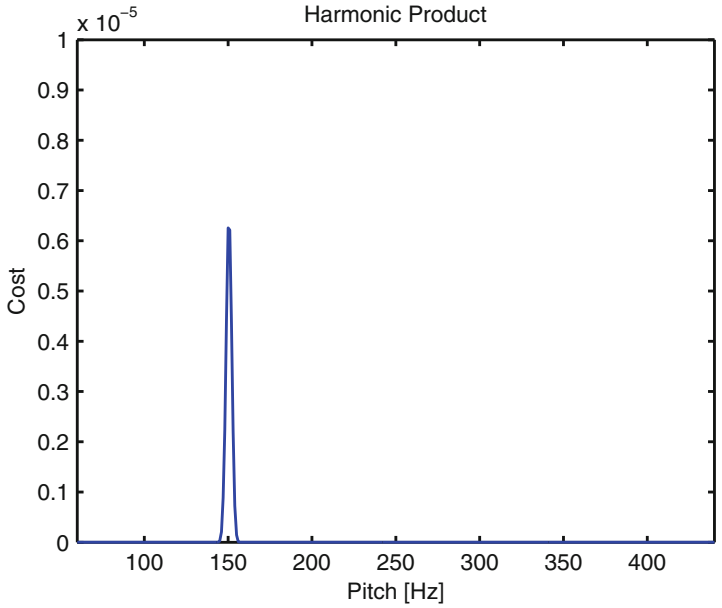


Fig. 12.10 Cost function for the harmonic product method, i.e., for the estimator in (12.28), as a function of the pitch for the signal in Fig. 12.1. Using this method, the pitch estimate is obtained by finding the highest peak

12.6 Exercises

The implementation of pitch estimators is a bit more advanced than the rest of this book so the following exercises aim instead at developing the understanding the nature of periodic signals and pitch rather than implementing complete pitch estimators.

Exercise 1 Implement a patch in Pure Data that reads a file and displays its spectrum for 20–30 ms segments. See if you can identify the pitch of different sounds (vowels, whistling, and single notes from musical instruments) from the spectrum.

Exercise 2 Use the `fiddle~` and `sigmund~` objects to estimate and display the pitch. Try different audio files but focus on single note instruments (trumpet, flute, voice, etc.).

Exercise 3 Implement the filter having the transfer function in (12.6) in a patch in Pure Data. Control the candidate pitch period with a slider. Pass a stationary single-pitched signal (e.g., from a flute, trumpet, or whistling). Display the pitch in frequency. Display the output signal of the filter as you change the candidate pitch and listen to it. What should it sound like when you find the correct pitch?

Exercise 4 Combine your patch with the spectrum analyzer developed in an earlier exercise. Show the spectrum of the input signal and the output signal of the filter in the previous exercise in Pure Data. What should the spectrum look like when the correct pitch period is chosen?

Appendix A
Introduction to Pure Data

A.1 Introduction

Pure Data is a free, open-source visual programming environment and run-time platform for interactive audio applications. It was developed by Miller Puckette and was first released in 1996. Despite its age, it remains popular in electronic music where it is often used in live performances, and it is, as we shall see, a useful tool for learning audio processing. Curiously, there exists a commercial product that was derived from Pure Data called Max/MSP. Technically speaking, Pure Data belongs to the class of dataflow program languages, which take their starting point in the movement of data. For the purpose of learning audio processing, Pure Data is well-suited because it provides a solution that works on a multitude of platforms (e.g., Windows, Linux, and MacOS) and it is easy to use in real-time, interactive applications, something that is difficult with other solutions. This means that students (or other readers) can easily apply what they have learned and make it available for use, for example in live music performances with minimal extra effort. Thus, there is a short way from idea to product with Pure Data. As always, however, there is no free lunch. While it is comparably easy to create interactive, real-time software that is ready for use with Pure Data, it is not so easy to test the algorithms, like for example a filter, in Pure Data, while this would be easy in more traditional software such as MATLAB.

In what follows, a brief introduction to Pure Data will be given. It is intended to get the reader acquainted with Pure Data and introduce the basic principles and objects needed to solve the exercises in this book. Thus, it is not intended as a comprehensive introduction to Pure Data. For that, we refer the interested reader to the many tutorials that can be found online, like FLOSS Manuals.[1] The introduction will also put some emphasis on how to test audio processing algorithms, e.g., filter

[1] http://write.flossmanuals.net/pure-data/.

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8

designs, in Pure Data. When reading this appendix, the reader is encouraged to create the examples him-/herself in Pure Data at the same time, as this is really the only way to really learn how to do it, like any other kind of programming or math. It is recommended that the Purr Data implementation of Pure Data is used, which is based on Miller Puckette's original version. It can be downloaded from GitHub.[2] This introduction is also based on Purr Data, although we shall simply refer to it as Pure Data or Pd for simplicity. A Pure Data reference card with all the most commonly used objects can be found online.[3]

A.2 Getting Started

Once Pure Data has been installed, the user will be met with the window shown in Fig. A.1 upon launching the program. This is the main window where text output, error messages, etc. will be printed. The DSP on/off button in the top left corner indicates whether Pure Data is processing audio. You can turn it on by simply clicking the button or choosing **Media** and then **Audio On**.

Before we get started with creating programs in Pure Data, it is important to check the audio settings. By clicking **Edit** and then **Preferences**, the settings window will open. There are several tabs in this window, but the most important one is the one called **Audio**. Depending on your platform (Windows, Linux, MacOS, etc.), it should look something like Fig. A.2. In this tab, you can see important settings like the sampling frequency, the number of channels, the block size, and the input and output devices. In this case, the input device is the microphone in the Webcam and the output is channels 1 and 2 in a Roland Quad-Capture USB device. The block size determines the size of the block (or frame, segment, buffer, etc.) of samples that Pure Data will use to process the audio. This means that the number of samples of the block size will be collected before the audio will be processed, and it thus introduces a delay. For real-time, interactive applications, you would like this to be as low as possible. However, there is an overhead associated with processing each block regardless of its size, so it is also less efficient to use a small block size.

Pure Data programs are comprised of patches, starting with the main patch. To create a patch, choose **File** and then **New** in the window. This creates an empty patch as shown in Fig. A.3. Programs are then designed by placing objects in this patch and connecting the inputs and outputs of the objects. To place an object on a patch, simply click **Put** and then select the type of object you desire. Then, you can put the object wherever you desire in the patch.

[2] https://github.com/agraef/purr-data/releases.

[3] https://puredata.info/docs/tutorials/pd-refcard.

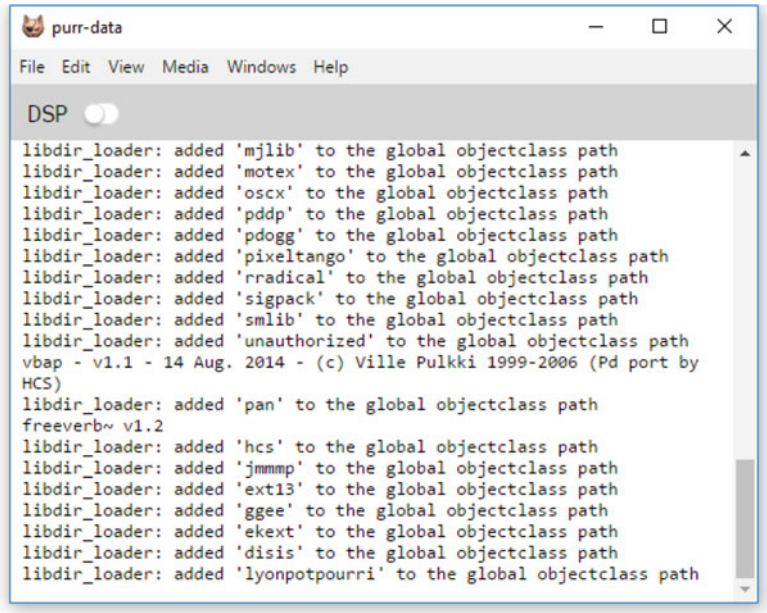


Fig. A.1 The main window in Purr Data

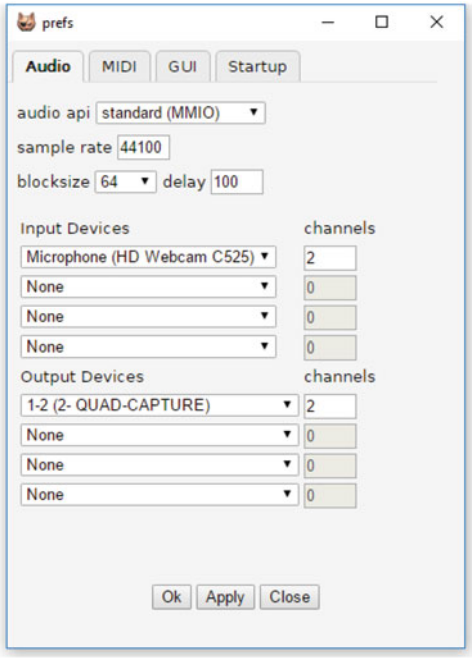


Fig. A.2 Audio settings in Pure Data

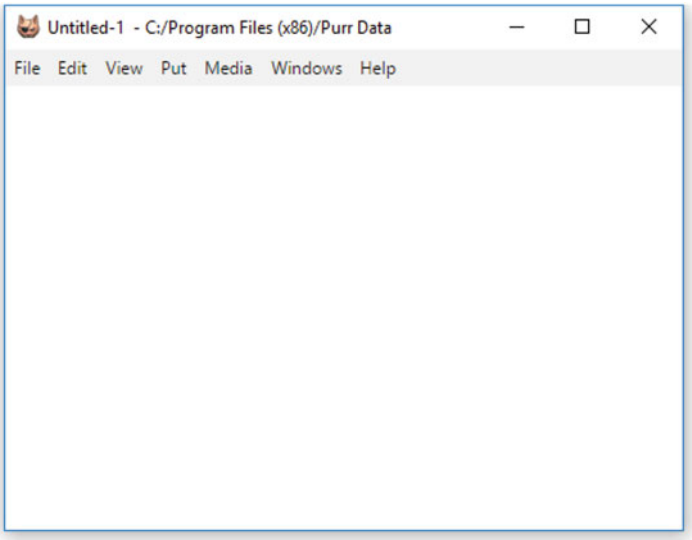


Fig. A.3 An empty patch

A.3 A Simple Example

Let us start by building a simple patch in Pure Data where we simply take the input from the microphone of the computer and pass it on to the output of the soundcard. Such a program is sometimes called talkthru and is used to test that things are working properly. In Fig. A.4, a patch doing this is shown. Before creating this patch, we must make sure that we are in the **Editmode**. To check this, click **Edit** in your patch window and see whether the **Editmode** is turned on or not. When the **Editmode** is turned on, we can modify our patch (e.g., edit, move, and create objects), while we can only interact with a patch via its user interface (e.g., sliders) when it is off. To create the talkthru patch, start by simply selecting **Put** and then **Object** and then writing `adc~` in the object. This object represents your input device, and its name derives from analog-to-digital converter. The inlet at the top of an object is where the input should be connected and the outlet at the bottom of the object is where the output appears. In this case, the object has one inlet (which actually is not used for anything) and two outlets, the left and right channel of the input device. To pass the input on to the output, repeat the process by selecting **Put** and then **Object** but this time the object should be called `dac~`, which refers to digital-to-analog converter. We can now connect the output of the `adc~` object to the input of the `dac~` object. To do this, hover your mouse pointer over the output of one object, click and hold the first mouse button, and then drag the connection to the desired input of the other object. Once the objects have been connected as shown in the figure, we can turn Pure Data on by selecting **Media** and then **Audio On**, but before you do this, make sure you have the audio volume turned down.

Fig. A.4 Talkthru patch where the input is passed on without modification to the output

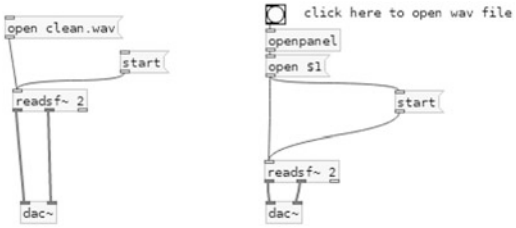


Fig. A.5 Patch for loading and playing back a wav file (**a**), and a patch for selecting, loading, and playing back a wav file using the GUI (**b**)

We also recommend that headphones are used to avoid feedback issues (and to not annoy your neighbors too much).

There are two kinds of data in Pure Data, control messages and signals. Signals are of course the stream of audio that we wish to somehow process, but control messages are all the other quantities that we need, such as filter coefficients. The ~ of the object indicates that the two objects in Fig. A.4 both operate on signals. Objects without ~ thus operate on control messages, and there are several kinds of control messages that we can use, many of which you will encounter in the examples to follow. Similarly, connections and object inlets/outlets can be for signals or control messages. Fat connections and dark inlets/outlets are for signals, while thin connections and light inlets/outlets are for control messages.

To load an audio file from the computer, we can use the `readsf~` object. To understand what a built-in object does, you can, after creating the object, right-click the object and then choose **Help**. Then, a window will appear where the object is explained in detail and its inlets, outlets, and creation arguments are listed. Creation arguments are arguments that are written alongside the object name when the object is created. In this case, for example, the arguments are the number of channels and the buffer size per channel. To open and play an audio file using the `readsf~` object, you must create a patch as shown in Fig. A.5a, only you must replace `clean.wav` with a file on your computer. The two inputs to the `readsf~` object are messages. To create these, simply click **Put** and then **Message**. Once you have created the patch, turn the DSP on, turn the edit mode off and then click the two messages (first `open...` and then `start`). You should now hear your audio file. Check the main window for any error messages if there is no sound. To make the patch more user friendly, we can extend it to allow the user to select the

file to be opened with `readsf~` using the `openpanel` object. This is shown in Fig. A.5b. There are a couple of things to note in this patch. Firstly, the object at the top is an instance of the important `bang` object, here in the form of a button. The `bang` object outputs a bang, which is a form of control message in Pure Data. It serves as a trigger or activator for other objects. In this case, for example, it activates the `openpanel` object. Secondly, the `open` message features an argument, `$1`, which refers to the first inlet of the message object. Hence, the message will be a combination of `open` and whatever comes out of the `openpanel` object's outlet. By right-clicking on the `openpanel` object and then selecting **Help**, we can confirm that the output is the filename of the file selected by the user. Similarly, it is also possible to save the output of a patch to an audio file in Pure Data. This can be done with the `writesf~` and `savepanel` objects which function much like their counterparts for opening and reading sound files.

A.4 Manipulating Signals

As we have seen again and again throughout the book, audio processing involves a number of basic mathematical operations. Filtering, for example, involves adding a number of delayed audio signals each multiplied by a constant, a filter coefficient. We will now take a look at how we can perform these basic operations on audio signals in Pure Data. Later, we will take a look at how to compute filter coefficients. To get started with manipulating signals, let us consider the simple example shown in Fig. A.6a. It is essentially the talkthru example from earlier, but with a volume control added. For the sake of simplicity, only the left channel of the `adc~` is used here. The operation of volume control is equivalent to multiplying signals by a constant. We can use the `*~` object for this. This object has two inlets which are then multiplied inside to object to produce the output. In this case, the left input is the signal and the right input is the constant we wish to multiply the signal with. To allow the user to control the volume, we add a horizontal slider object by choosing **Put** and then **HSlider**. The range of values of the slider, i.e., the minimum and maximum values, can be selected by right-clicking the object and selecting **Properties**, which produces the window shown in Fig. A.6b. Moreover, a logarithmic scaling can also be chosen, something that is usually preferable for quantities such as volumes and frequencies. We now have a Pure Data patch that manipulates an audio signal, albeit in a simple way.

In Fig. A.7a, a slightly more complicated example is given where the left and right channels of a loaded wav file are added using the `+~` object and subtracted using the `-~` object, respectively, before being passed on to the soundcard. From the two examples, we can see that it is quite easy to add, multiply, and subtract signals in Pure Data. It is a bit more complicated to delay signals, however. In fact, we need two objects to implement a delay of a certain amount. The first is `delwrite~` which writes a signal we wish to delay to a buffer or delay line as they are frequently called in audio applications (from which the object also derives its name). This

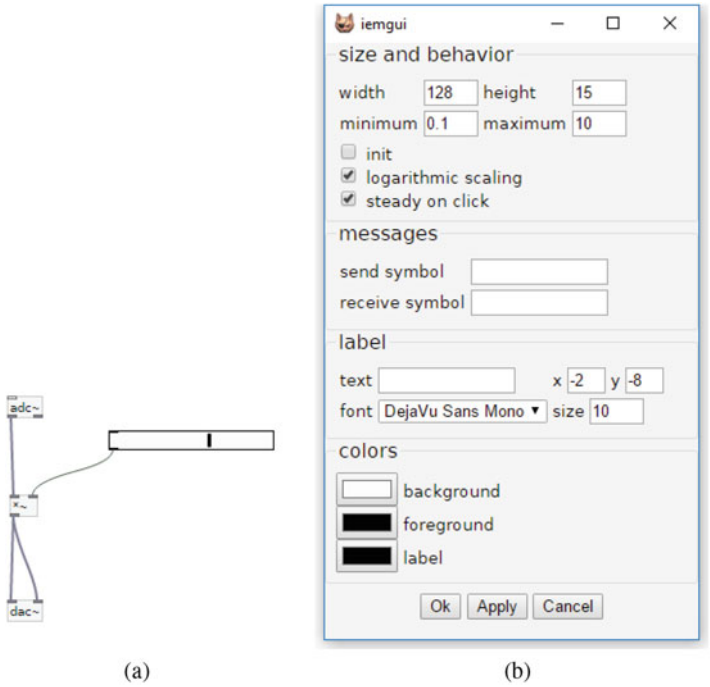


Fig. A.6 Adding a volume control to the talkthru example (**a**), and the properties of the horizontal slider (**b**)

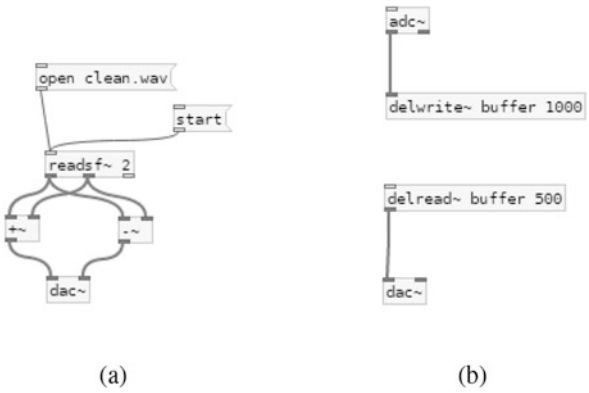


Fig. A.7 Examples of signal manipulations: Adding and subtracting signals, here the left and right input channels of the soundcard (**a**), and delaying a signal by 500 ms (**b**)

object takes as its creation argument two parameters, namely the name of the buffer and the size in milliseconds. The size in milliseconds is then the maximum amount by which the signal can be delayed, so it must be chosen carefully. The second object is `delread~`, which reads the signal from the buffer with a specified amount of delay. It takes the buffer name as a creation argument as well as the desired delay in milliseconds. The `delwrite~` must be created before the `delread~` or you will get an error, as the delay line does not exist and thus cannot be read. In Fig. A.7b, an example of how to use `delwrite~` and `delread~` is shown. Note how the `delread~` object has an inlet. This inlet can be used to control the amount of delay, so it is actually possible to vary the delay this way. Not only that, the object also supports fractional delays, meaning that the delay does not have to correspond to an integer number of samples. The delay is, though, fixed per block, so if a time-varying delay that varies from sample to sample is desired, then a different approach must be taken and the `vd~` object must be used instead. This is important to remember in connection with effects with time-varying delays, such as the chorus and flanger. Aside from the restriction that the delay in the `delread~` object cannot be larger than the size of the buffer, it also cannot be smaller than the block size used by Pure Data, which can be adjusted in the preferences in Pure Data. Later, we will return to the matter of how to achieve very low delays, like a single sample.

When manipulating signals, it is sometimes useful, or perhaps even necessary, to convert signals to control messages and vice versa. This can be done with the two objects `snapshot~` and `sig~`. The `snapshot~` object takes a snapshot of the signal on its inlet when it is triggered by a bang on the first inlet. Similarly, `sig~` converts a control message, a float to be specific, into a signal. In Fig. A.8, a simple example of how these objects can be used is shown. Notice how the line connecting the two objects in question is fat, indicating that it is for signals.

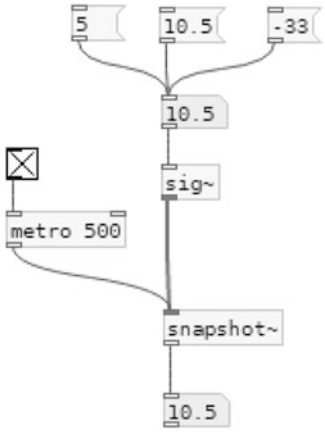


Fig. A.8 An example of how `snapshot~` and `sig~` can be used to covert control messages to/from signals

A.5 Subpatches and Abstractions

When working on big patches or when reusing code, it might be beneficial to use subpatches and abstractions. Subpatches are a way of organizing your main patch by hiding parts of your program in a separate window. You create a subpatch by selecting **Put** and then **Object** and then typing `pd` in the object followed by the name of the subpatch. This opens a new window, the subpatch, where you can put and connect objects. However, to make this work, we must define inputs and outputs, or inlets and outlets as they are called in Pure Data. These are created as with all other objects and should be named `inlet` and `outlet` for control messages and `inlet~` and `outlet~` for signals. When adding inlets and outlets, you can see that the inlets and outlets will appear on the object in the main patch. The ordering of the inlets and outlets is determined by their location in the subpatch. More specifically, the ordering is determined by their horizontal location from left to right. An example of a main patch with parts of its objects placed in a subpatch is shown in Fig. A.9. Instead of connecting inlets and outlets of objects and subpatches, we can also use wireless connections using the `send` and `receive` objects. These objects can be used with control messages only (i.e., messages, symbols, and lists). There are however signal counterparts for these objects, named `send~` and `receive~`. When using these objects, a variable name must be given as a creation argument. Hence, the objects `send msg` and `receive msg` will form a wireless connection where the inlet on the `send` object will be identical to the output of the `receive` object.

When a piece of a patch is used in several places, like a filter type that is used several times, like the comb and all-pass filters of Schroeder's reverb, it is best to put it in an abstraction, which is the Pure Data's equivalent to functions, as known in many other programming languages. You create an abstraction by clicking **File** and then **New**. In the new window, select **File** and then **Save** and choose an appropriate name for your abstraction. You can use the abstraction by simply writing the name you have chosen in an object on the main patch. As for the subpatch, you define the inputs and outputs of the abstraction by placing inlets and outlet objects of

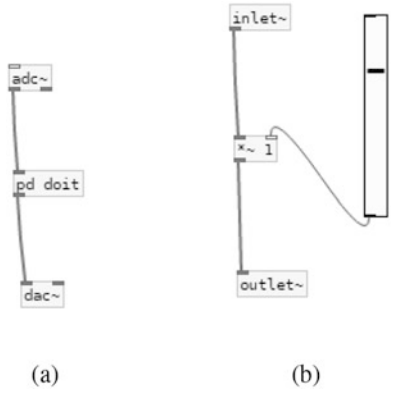


Fig. A.9 The main patch (**a**), and a subpatch (**b**)

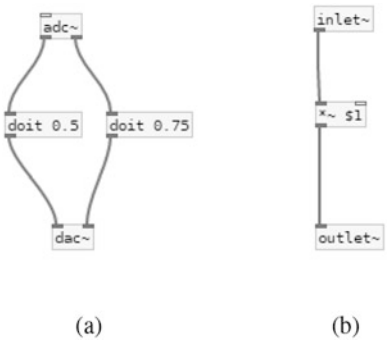


Fig. A.10 The main patch (**a**), and an abstraction (**b**). In the abstraction, the input signal is multiplied by the creation argument, `$1`, which is different for the two channels in the main patch

Fig. A.11 An abstraction implementing a delay with a unique name for the delay line. The amount of delay is specified by the creation argument, `$1`

the abstraction. You can also use creation arguments, which can be used inside the abstraction. This is done using the dollar sign, so `$1` is the first creation argument and `$2` is the second and so forth. An example of an abstraction and how they can be used is shown in Fig. A.10. Concerning subpatches and abstractions, it is important to understand that all variables in Pure Data essentially are global variables, i.e., the main patch, its subpatches, and all abstractions that are used share the same namespace. Therefore, signals and control messages can also be sent to subpatches and abstractions using `send` and `receive`. This is problematic if, for example, we need to operate on different delay lines in different instances of an abstraction implementing filtering. So, for example, an abstraction employing a delay like in Fig. A.7b would be operating on the same buffer. To create a unique name for each delay line, we can use `$0`, as shown in Fig. A.11.

A.6 Plotting Audio

In many cases, it is useful to plot the audio signals to see what is going on. An example of how to do this in Pure Data is shown in Fig. A.12. We can recognize parts of the patch from Fig. A.5a, namely the parts that let the user select an audio file, which is then opened and played back. To plot the audio that is being played back,

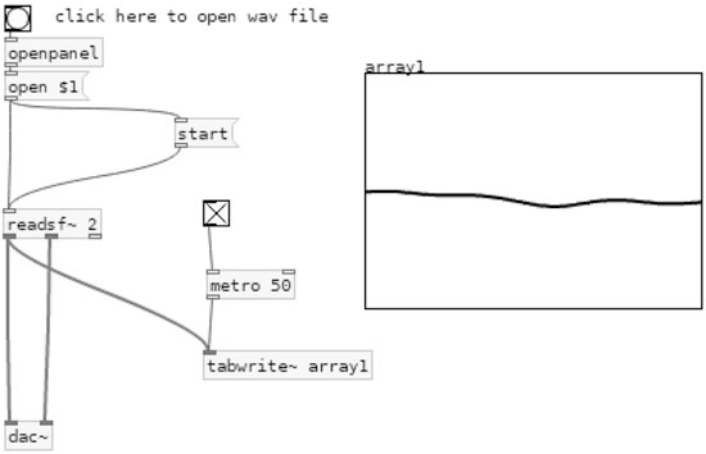


Fig. A.12 Plotting an audio signal using an array

we must add an array to the patch. This is done by selecting **Put** and then **Array** after which a window will appear where you have to choose the name (i.e., the name of the underlying variable) and the length of the array. The array can then be placed on the patch. To plot something in the array, we must do two things. First, we must write something to the array. This is done with the `tabwrite~` object in the case of signals, as is the case in the present example, and `tabwrite` in the case of control messages. Second, we must activate the `tabwrite~` object. This can be done with a bang message, and we could in principle manually do this, but it would be more convenient to do it periodically in an automatic fashion. This can be achieved with the `metro` object, which is a very useful object in many contexts. It sends periodic bangs where the period between bangs, in milliseconds, is specified by the creation argument, which is here 50 ms. The `metro` object, though, also needs to be activated with a bang. In Fig. A.12, this is done with a toggle object, which can be created by selecting **Put** and then **Toggle**. This object sends a bang once the toggle is switched on. This could also be achieved with the useful `loadbang` object, which sends a bang once a patch is loaded, i.e., when it is originally opened.

When working on audio processing, we are also frequently interested in analyzing signals and systems using the Fourier transform. This can also be done fairly easily in Pure Data. To do this, we create the patch shown in Fig. A.13a, which aside from the usual objects, features an array, called `array1`. The actual computation of the Fourier transform, which is done using the Fast Fourier Transform (FFT), happens inside the abstraction called `spectrum` shown in Fig. A.14. The inlet is passed on to an object called `rfft~`, which computes the FFT of a real signal. Its output is a real value, the left outlet, and an imaginary value, right outlet. To compute the power, these are squared and added and then converted to dB using `log~` and multiplied by 10, as the power spectrum in dB is given by $20 \log_{10} |X(\omega)|$, where $X(\omega)$ is the Fourier transform of x_n. Note that the `log~` object takes the base of

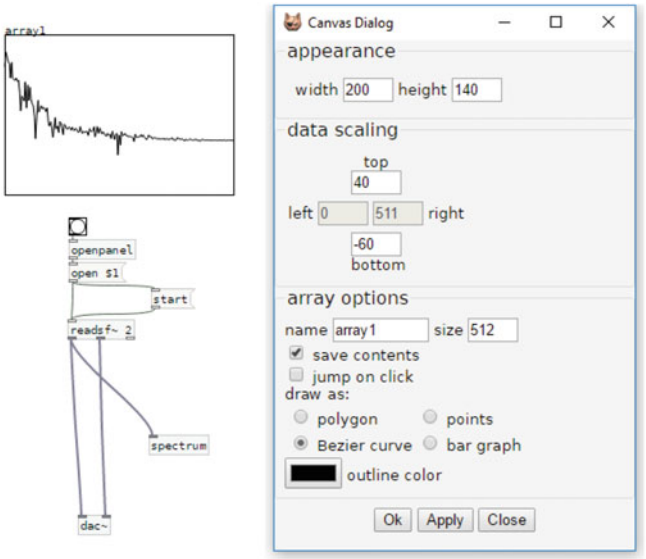


Fig. A.13 The main patch for plotting the spectrum, i.e., the Fourier transform of an audio signal, (**a**) and the properties of the array used for plotting the result (**b**)

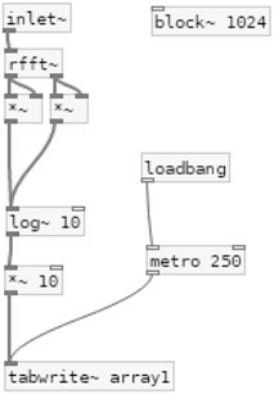


Fig. A.14 Abstraction for computing the Fourier transform and the power spectrum of an audio signal

the algorithm as a creation argument. As with the example in Fig. A.12, the signal to be plotted, here the Fourier transform of the input signal, is written to the array using the `tabwrite~` object. As before, this object needs to be activated using a bang, and we here do this with the `metro` object which itself needs to be activated with a `loadbang` to activate it automatically when loaded. In this case, the array is updated every 250 ms. There is one final and important object to pay attention

to, namely the `block~` object. It sets the block size locally for the abstraction, something that cannot be done for the main patch. Here, we set it to 1024 which means that the output is a 1024 point Fourier transform. Notice in Fig. A.13b that the size of the array is set to 512 to display half the spectrum of the real audio signal.

A.7 Mathematical and Logical Expressions

An important element of audio processing is of course the computation of filters coefficients, delays, etc. There are different ways to go about such computations in Pure Data. As a simple example of how to perform arithmetic in Pure Data, consider the path shown in Fig. A.15a. When the button is pushed, a bang is sent to the message containing the number 2. A message is created by clicking **Put** and then **Message** in the menu. The message is then sent to the addition object, where the number 4 is added to the input, and then the result is passed on to the multiplication object where it is multiplied by 2. The result is then passed to a number object. To create such a number object, simply select **Put** and then **Number**. These objects are useful for displaying the results of computations, such as filter coefficients and delays, in your patches. Another option is to use the `print` object, which prints the input in the main Pure Data window.

Another example is shown in Fig. A.15b, where one number is divided by another and then the result is raised to the power of two using the `pow` object, another useful object. This example is used because it highlights an important detail, namely the order in which things happen in Pure Data. When exactly is the division in the `/` object carried out? The answer is that it is carried out when there is an output on the left-most inlet. This holds for most objects Pure Data. This means that if the number on the right inlet, in this case 5, has not been passed to the division object before the number on the left inlet, the result will not be correct! In this case, it appears that it worked out okay by coincidence. But, we must generally ensure that things happen in the correct order. This can be done in several ways, but the simplest way is probably using the `trigger` object, which is also simply called `t`. The `trigger` object outputs its inputs on the outlets in the order from right to left. The types of the outputs of the object, i.e., the outlets, must be specified. Here, `b` indicates that outlets one and two are both bangs. Other types of outputs can be found by right-

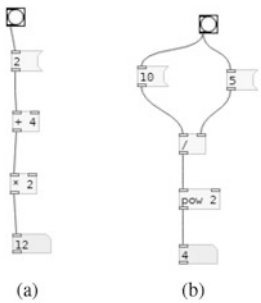


Fig. A.15 An example of how to perform basic arithmetic in Pure Data (**a**), and an example of how the ordering of the inlets is important (**b**)

clicking the `trigger` object and clicking **Help**, and aside from the bang, they also include float indicated by `f`, symbol indicated by `s`, list indicated by `l`, pointer indicated by `p`, and anything indicated by `a`. Another useful object for ensuring that events happen in the correct order is the `bondo` object. The `bondo` object outputs all its values whenever a new control message arrives on any of its inlets, the number of inlets and outlets being specified by the first creation argument. In Fig. A.16b, it is illustrated how it can be used for the example in Fig. A.15b. In Table A.1, a list of the most commonly used math objects in Pure Data is provided. To learn more about these objects, simply create them in Pure Data and see the help.

It is also possible to perform relational and logical operations on numbers in Pure Data. For relational operations, this is done using the objects dedicated to each of these operations, like `==` to check whether two numbers are equal. They have in common that they output a Boolean value, i.e., true or false, represented by 1 or 0, respectively, when comparing the inputs. Moreover, the right inlet of these objects, or, alternatively, its creation argument, forms the value that the left inlet will be compared to. An example of relational operations and how to use the objects is shown in Fig. A.17a. In the example, a random number that is generated when the user pushes the button is compared to a constant. The logical operations in Pure Data function the same way their equivalents do in the C programming language. The inputs to logical operations are in principle Boolean, meaning that they are either

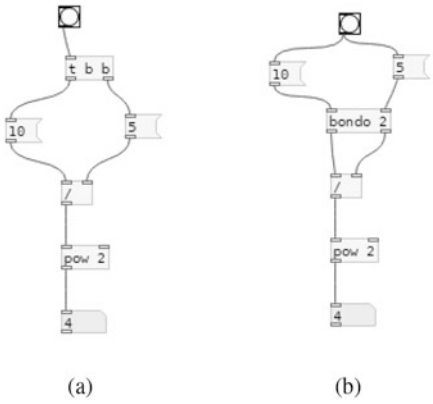


Fig. A.16 An example of how the `trigger` object can be used to ensure that events in Fig. A.15b happen in the right order (**a**), and how the same can be achieved with the `bondo` object (**b**)

Table A.1 List of standard math objects in Pure Data

Object	Description
`+ - * / pow`	Basic arithmetic
`mod div sqrt log exp abs`	Functions
`sin cos tan atan atan2`	Geometric functions
`max min`	Max and min functions
`rmstodb powtodb dbtorms dbtopow`	Convert to/from dB
`random`	Random number generator
`clip`	Clip function

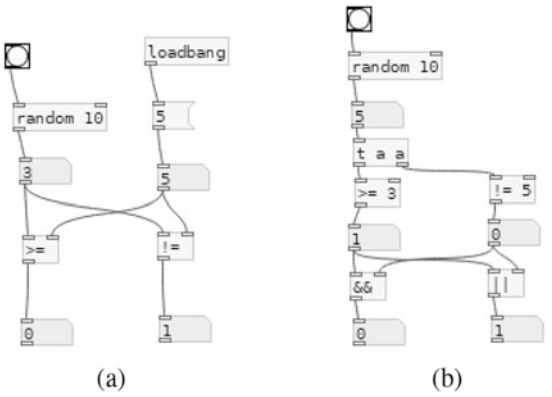


Fig. A.17 Example of how to use relational operators (i.e., comparisons) in Pure Data (**a**) and a combination of relational and logical operators (**b**). Notice how the `trigger` object is used to ensure the correct execution order

Table A.2 List of Pure Data objects for relational and logical operations

Object	Description
==	Equal to
!=	Not equal to
>	Larger than
>=	Larger than or equal to
<	Smaller than
<=	Smaller than or equal to
&&	Logical AND
||	Logical OR

true or false, but in the C programming language, as there is no Boolean data type, the value 0 represents false and everything else represents true. The output of the objects that perform logical operations in Pure Data is then also Boolean. Logical AND is performed using the `&&` object, while logical OR is performed using `||`. In Fig. A.17b, an example combining relational and logical operations in Pure Data is shown. In Table A.2, an overview of objects for performing relation operations, i.e., comparisons, on numbers and for performing logical operations is given. Note that there also exist objects for performing bitwise operations on integers, including `&`, `|`, `<<`, and `>>`. To see what these do in more detail (if you cannot guess it), see the help in Pure Data.

For computing filter coefficients involving somewhat complicated expressions, it might be cumbersome to have to create many objects in Pure Data and figure out the logistics and the ordering. Fortunately, there is an easier way to do this using the `expr` object. It allows for evaluation of mathematical C-style expressions in Pure Data, and multiple expressions can be evaluated in one object by separating them by a semicolon. In that case, expressions are evaluated in the order from last to first with each expression being evaluated from right to left. The inputs of the `expr` objects are defined using `$` followed by an `i` when the input is an integer, an `f` when it is a float, or `s` when it is a symbol, followed by the number of the inlet. So, for example, an expression involving `$i2` and `$f1` would mean that the first inlet is a float and the second is an integer. All the usual operations can be performed

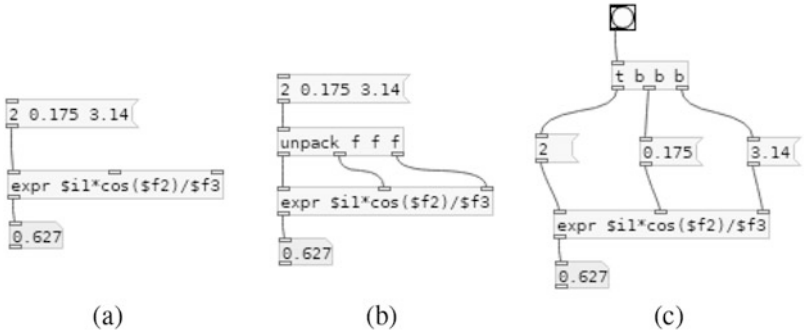


Fig. A.18 Three ways of using the `expr` object in Pure Data that all produce the same result with a list (**a**), the `unpack` object (**b**), and using a `trigger` object (**c**)

within an `expr` object, like addition, multiplication, division, etc., and the rest of the ones listed in Tables A.1 and A.2. When using the `expr` and related objects, it is important to note that computations are performed when there is a new input on the left-most inlet or, alternatively, when the object receives a bang on the left-most inlet. Therefore, special attention must be given to the order of the computations of the objects that connect the inlets. Figure A.18 shows how the `expr` object can be used in Pure Data and how the correct result can be achieved in several ways. The only new object here is the `unpack` object which unpacks a list and distributes its elements, as specified by the creation arguments, among its outlets. Similarly, multiple control messages can be combined into a list with the `pack` object.

There also exists an object for performing similar computations when the input can be a signal. This object is, not surprisingly, called `expr~`. It works, from the user's standpoint, just like the `expr` object, except that its inlets can also be signals and the outputs are signals. Hence, `expr` should be used when the inputs and outputs are control messages, while `expr~` should be used when they are signals. Signals are defined in expressions inside the object using `$v1-$v9`. The final object to be mentioned in this connection is the `fexpr~` object which can be used for implementing filters. It allows for sample-by-sample evaluation of expressions and provides access to past input and output samples. Using this object, input samples can be accessed as `$x1[0]`, `$x1[-1]`, `$x1[-2]`, and so forth in expressions. Here, `$x1` indicates the signal in the first inlet and `[0]`, `[-1]`, `[-2]` refers to signals delayed by 0, 1, and 2 samples, respectively. So, `$x3[-2]` would refer to the signal in the third inlet delayed by two samples. Output samples can also be used in a similar manner, only they are referred to as `$y[0]`, `$y[-1]`, etc. For both inputs and outputs, the delay cannot be larger than the block size of the patch/abstraction. In Fig. A.19a, an example of how this object can be used is shown for computing simple low-pass and high-pass filters using feedforward. Figure A.19b shows a simple example involving feedback. Table A.3 provides an overview of commonly used functions that can be used with the `expr`, `expr~`, and `fexpr~` objects. For more details on these functions, we refer the reader to the help in Pure Data.

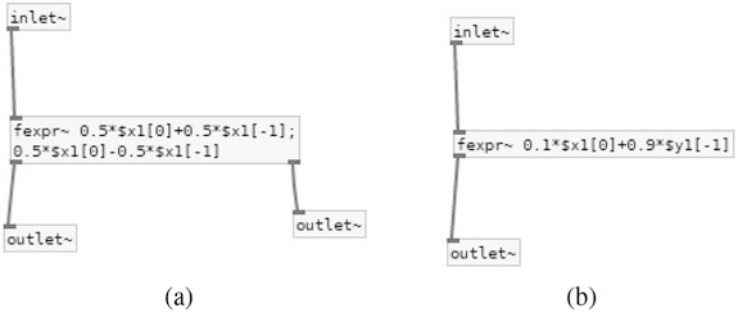


Fig. A.19 Two examples of how the `fexpr~` object can be used for filtering using feedforward (**a**) and using feedback (**b**)

Table A.3 List of functions that can be used in `expr`, `expr~`, and `fexpr~`. Shown are the function names, the number of arguments, and a short description of the function

Function	Args	Description
`pow()`	2	Raise to the power of
`sqrt()`	1	Square root
`exp()`	1	Exponential function
`log`	1	Natural logarithm
`log10()`	1	Base 10 logarithm
`fact()`	1	Factorial
`erf()`	1	Error function
`cbrt()`	1	Cube root
`sin() asin()`	1	Sine and its inverse
`cos() acos()`	1	Cosine and its inverse
`tan() atan()`	1	Tangent and its inverse
`atan2()`	2	2-Argument arctangent
`sinh() asinh()`	1	Hyperbolic sine and its inverse
`cosh() acosh()`	1	Hyperbolic cosine and its inverse
`tanh() atanh()`	1	Hyperbolic tangent and its inverse
`floor() ceil()`	1	Floor/ceil operators
`isnan() isinf()`	1	Check if value is NaN/inf
`fmod`	1	Floating point remainder
`if()`	3	Conditional if
`int() float()`	1	Convert to integer/float
`rint()`	1	Rounding
`min() max()`	2	Minimum/maximum
`abs()`	1	Absolute value

A.8 Generating Signals

In many applications, we need to not only be able to process audio signals read from files or recorded using the sound card but also to generate audio signals ourselves. In the plucked-string synthesizer, for example, we generate the sound of a plucked-string instrument from just a noise burst. Similarly, in additive synthesis audio signals are synthesized by adding components like sinusoids. In other cases, we need to generate signals that control some parameters in filters or audio effects. Example of this is the auto-wah where the resonance frequency of a resonator filer is controlled automatically by a signal and the chorus effect where a time-varying delay of a comb filter is controlled by a signal. There are several objects for doing this in Pure Data, and we will now take a closer look at some of them.

To generate noise, i.e., a random signal, we can use the `noise~` object which draws its samples independently from a uniform distribution between -1 and 1. The power spectrum of such a signal is flat so the noise is also said to be white. White noise is frequently used for various purposes, including testing of filters and systems. As mentioned, the plucked-string synthesis requires that the modified comb filter be excited with a noise burst. A noise burst is a noise signal of a very short duration. To generate such a signal in Pure Data, we can use the `noise~` object. To turn the noise burst on, we can multiply its output by 1 and to turn it off, we can multiply by 0. To control the length of the noise burst, we have to delay the time between the 1 and the 0, something that can be achieved using the `del` object. In Fig. A.20, it is shown how a noise burst can be generated in this way. The length of the burst is determined by the creation argument of the `del` object, which is the delay in milliseconds. The burst is triggered by pressing the button.

In additive synthesis, signals are often synthesized by adding a number of sinusoids having different frequencies and amplitudes and then manipulating the envelope and spectrum of the signal. To generate sinusoids in Pure Data, we can use the `osc~` object. It takes as its first inlet the desired frequency of the oscillator (or sinusoid). This can also be set via the creation argument of the object. The output of the object is then a cosine wave of a desired frequency. The amplitude of the

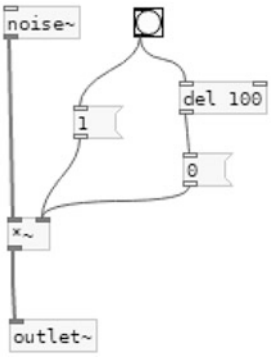


Fig. A.20 Abstraction for generating a noise burst with a duration of 100 ms

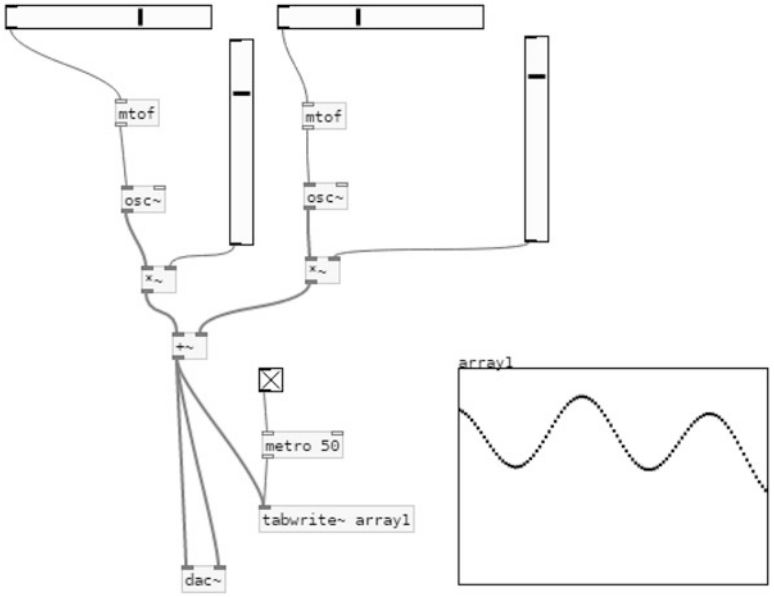


Fig. A.21 An example of additive synthesis with two oscillators whose amplitudes and frequencies are controlled by sliders

wave can be changed by multiplying it by the desired amplitude. In Pure Data, we can easily add sliders to control both the frequency and the amplitude of a number of oscillators, as shown for two oscillators in Fig. A.21. To generate sinusoids on a musical scale, the sliders here control the MIDI note number of the oscillators which are then converted to Hertz using the `mtof` object. There also exists an object for computing the MIDI note number from a frequency named `ftom`.

The two oscillators in the previous example will keep playing until we turn off the DSP in Pure Data. To give the notes a natural envelope and a duration, we can use the `line~` object to change the volume in combination with a multiplication, object, i.e., the `*~` object. The `line~` object changes the current value to the desired value over a specified duration. This is done my sending the object a message comprising two elements, i.e., a list. The first element specifies the desired value, while the second specifies the duration in milliseconds. So, if we wish for the amplitude of the oscillator to go from 0 to 1 over 50 ms and then after 100 ms go from 1 back to 0 over 2000 ms, we can do this as shown in Fig. A.22, which uses a bang in combination with the `del` object from before to trigger the two messages. This principle can be extended to implement the well-known ADSR (attack–decay–sustain–release) envelope commonly used in synthesizers.

The last object for generating signals that we will here mention is the `phasor~` object. This object has little to do with the mathematical concept of a phasor that has been used throughout the book, so the reader should be careful not to confuse these two. It also does not have anything to do with the phaser effect. What it does is that

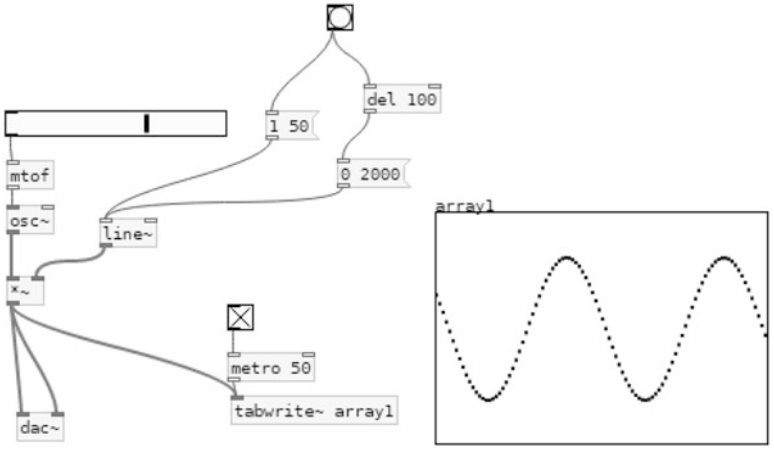


Fig. A.22 Using `line~` to change the envelope of a sinusoid

it creates a sawtooth wave that varies between 0 and 1 with a frequency specified by its creation argument or the first inlet. The `phasor~` object can, for example, be used in combination with the `cos~` object, which computes cosine to the input times 2π. This would simply produce a cosine signal with a desired frequency. A sawtooth wave is of course a periodic signal (or it could not have a frequency), but more importantly, it can be used for generating other periodic signals. In Fig. A.23, it is demonstrated how it can be used for generating a square wave and a triangle wave using some simple expressions implemented using the `expr~` object. Such periodic waveforms are often used in synthesis, in particular in combination with filters that modify the timbre of these basic waveforms. This method is known as subtractive synthesis, and some simple built-in filters in Pure Data that can be used for this is `lop~`, which is a low-pass filter, `hip~`, which is a high-pass filter, and `bp~`, which is a bandpass filter.

A.9 Implementing Filters

Filtering involves a number of basic operations, namely additions, multiplications, and delays. Moreover, some mathematical expressions typically need to be evaluated to compute the coefficients of the filters from the user input. In the previous sections, we have seen how all this can be done in Pure Data. In this section, we will take a look at how it can all be put together to produce filters. We will start with a simple, yet classical filter in the context of audio, namely the comb filter. The comb filter has the difference equation $y_n = x_n + ay_{n-D}$. To implement this in Pure Data, we can see that we must delay the output signal by D samples, then multiply it by a, and then add the result to the input. We have previously seen how we can implement

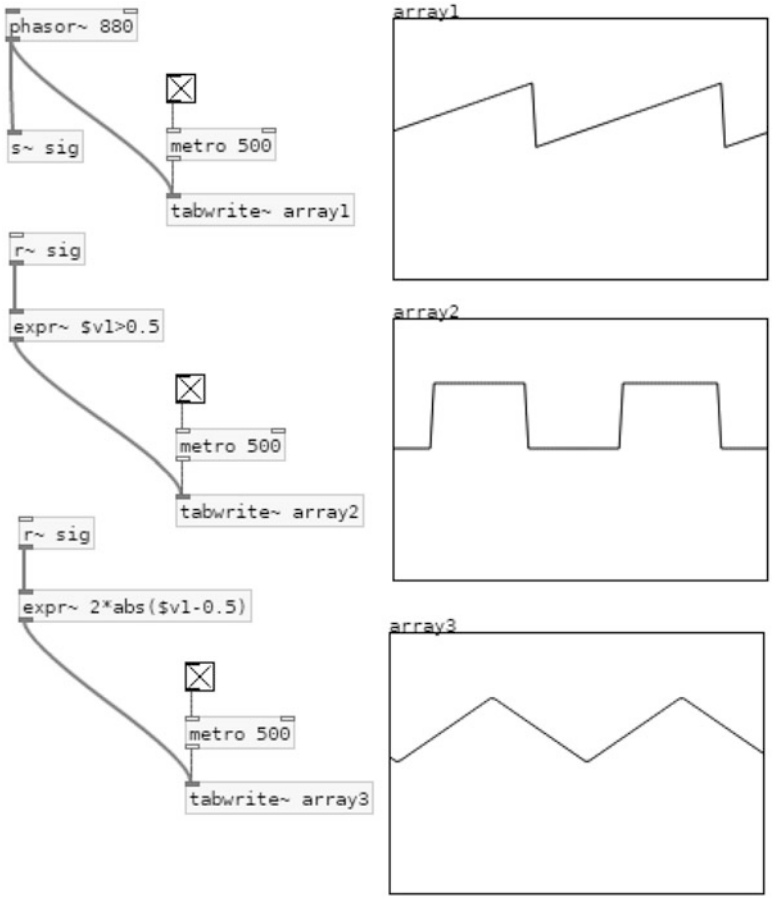


Fig. A.23 Using the `phasor~` object to generate periodic signals, here a sawtooth (top), a square (middle), and a triangle (bottom) wave

delays of this sort in Pure Data with the `dealwrite~` and `delread~` objects. In Fig. A.24, an implementation of a comb filter is shown. In this implementation, the filter coefficient is set to $a = 0.95$ and the delay is set to $D = 44{,}100$ corresponding to 100 ms at a sampling frequency of 44.1 kHz. To implement an inverse comb filter, which has the difference equation $y_n = x_n - ax_{n-D}$, we only need to make a few changes. First, the inlet of the `delwrite~` object should be connected to the `inlet~` object of the abstraction instead of the `+~` object. Second, the `+~` object should be changed to a `-~` object, or the filter coefficient should be modified to -0.95.

To make our abstraction slightly more user friendly and sophisticated, we can make a few changes. Recall that the comb filter can also be expressed as $y_n = x_n + R^D y_{n-D}$, where R is the pole radius. Suppose we would like to make R a user parameter and also let the user select the delay in samples, D, with sliders. To

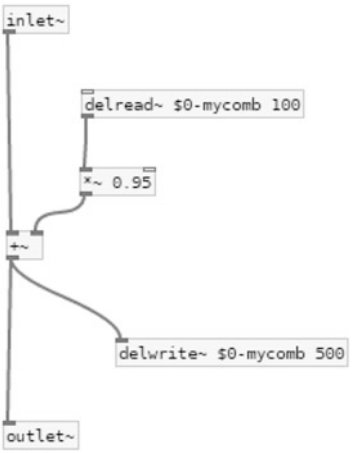


Fig. A.24 Abstraction for implementing a simple comb filter with $a = 0.95$ and a delay corresponding to 100 ms

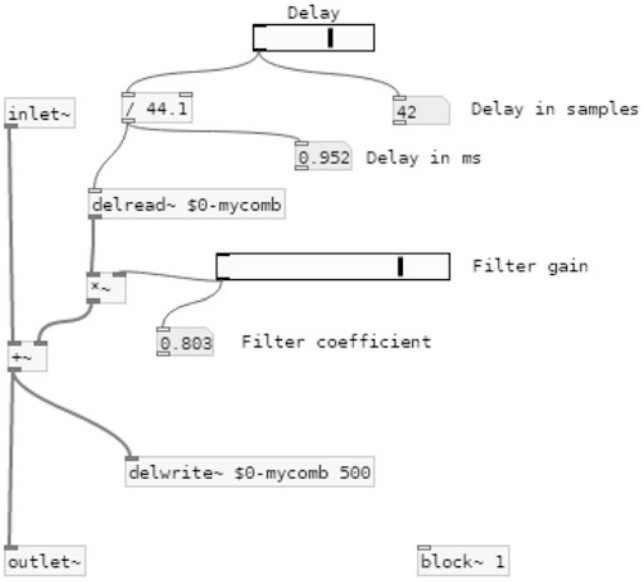


Fig. A.25 Abstraction for implementing a comb filter where the user can control the pole radius, R, and the delay in samples, D

do this, we can create two sliders, one for D and one for R. To compute the filter coefficient, which is now given by R^D, we must use the `pow` object. However, the delay must be converted to milliseconds before we can use it with the `delread~` object. This can be done by dividing the delay in samples by 44.1. In Fig. A.25, it

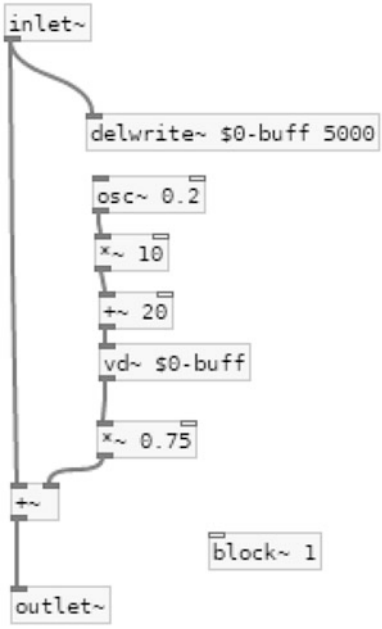


Fig. A.26 A filter employing a time-varying delay which can be implemented with the `vd~` object

is shown how the resulting abstraction may look in Pure Data. Several of the effects discussed in the book, such as the chorus and flanger effects, employ comb filters, but with time-varying delays. To implement such effects, the `vd~` object must be used rather than the usual `delread~`, as the former allows for delays, specified by the signal in its first inlet, to be time varying. In contrast, the inlet of the latter is a control message, and the delay can only be changed once per block. Figure A.26 shows how the time-varying delay can be implemented in a simple comb-like filter structure. In this case, the delay is generated by an oscillator using the `osc~` object.

Many filters involve delays of only a few samples, like simple low-pass and high-pass filters as well as more sophisticated filters such as the resonator and equalizer filter. For these types of filters, it is best to use dedicated objects for this, as we would otherwise have to set the block size to 1 in the involved abstractions, and this is very inefficient. Instead, the `biquad~` object can be used for this. It is an object for implementing a biquad filter, which is a second-order filter involving both feedforward and feedback. It has the difference equation $y_n = b_0 x_n + b_1 x_{n-1} + b_2 x_{n-2} + a_1 y_{n-1} + a_2 y_{n-2}$. The `biquad~` object then takes as its input the signal to be filtered, as well as the filter coefficients, here b_0, b_1, b_2, a_1, a_2. These can either be sent to the object as a list using the first inlet or be used as creation arguments. In the list, they should be ordered as a_1, a_2, b_0, b_1, b_2. As an example of how to use the `biquad~` object, consider a resonator filter having the difference equation $y_n = x_n + 2R\cos(\theta) y_{n-1} - R^2 y_{n-2}$. This filter has two parameters, the pole angle θ and the pole radius R, which we will consider as user parameters from which to compute the filter coefficients. We can see that there are no terms involving x_{n-1} and x_{n-2},

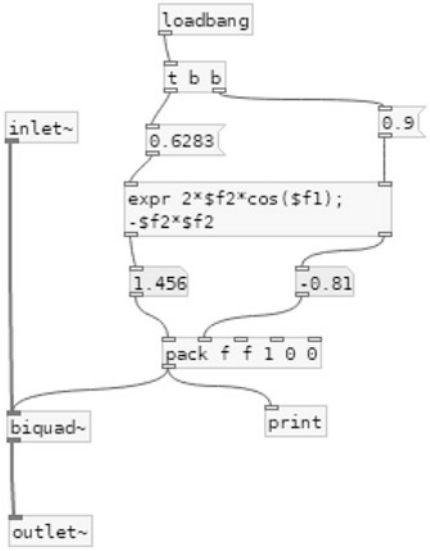


Fig. A.27 Resonator filter implementation using the `biquad~` object

and we must therefore set b_1 and b_2 to 0. We can also see that $b_0 = 1$ and that the two remaining coefficients are $a_1 = 2R\cos(\theta)$ and $a_2 = -R^2$. To compute these, we can use the `expr` object. In Fig. A.27, it is shown how to compute the coefficients and implement the resulting filter with the `biquad~` object. The computation of the filter coefficient is triggered by the `loadbang` object when the abstraction is first loaded, and the trigger ensures that the message for the right inlet of the `expr` object is ready first. The filter coefficients for the `biquad~` object are organized in a list using the `pack` object before being sent to the `biquad~` object. More complicated filters such as the equalizer filters can be implemented using the same principles. Concerning how to implement filters involving delays of more than two samples, we remark that such filters can be factored into a number of second- and first-order filters by finding the roots of the denominator and numerator of the transfer function. This is typically how IIR filters are implemented, while FIR filters are typically implemented in the frequency domain using fast convolution.

A.10 Testing Filters

An important aspect of any kind of engineering is testing. Audio processing is no different. When we design and implement filters and effects, we could like to make sure that they function correctly and according to our specifications and expectations. For example in the case of the comb filter in Fig. A.25, we would like to make sure that the filter that we implemented is indeed a comb filter. For a comb filter, we know what the impulse response should look like. It should have regularly

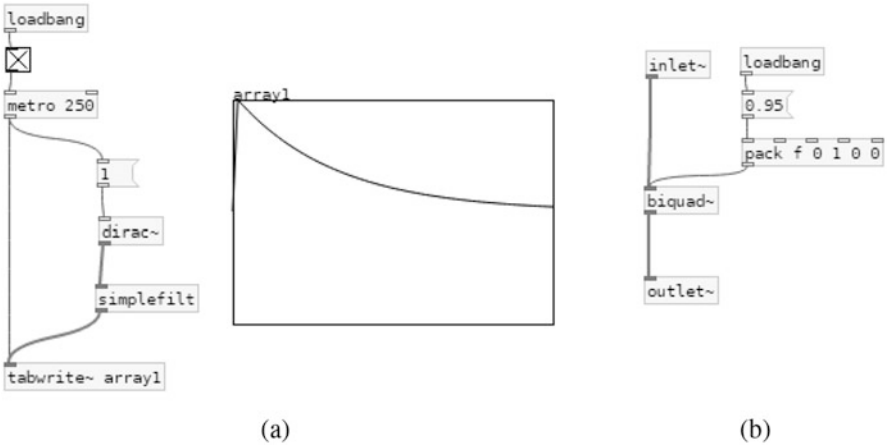


Fig. A.28 Measuring and plotting the impulse response (**a**) of the abstraction named `simplefilt` (**b**)

spaced non-zero values that decay over time. Similarly, the frequency response should look like the teeth of a comb. Indeed, looking at the impulse responses and frequency responses of filters is the most common way of testing them. Therefore, we will now explore how we can compute and plot these in Pure Data.

To find the impulse response of a filter in Pure Data, all we have to do is feed an impulse into the filter to be tested and then write the output of the filter to an array, which we can then plot. To generate an impulse, we can use the `dirac~` object (although its name is really a misnomer), which generates a unit impulse, i.e., a digital impulse, as shown in Fig. A.28a and we can write the output of the filter, here the simple filter in Fig. A.28b, i.e., the impulse response, to an array using `tabwrite~`. The `dirac~` object has to be triggered, which is here done by a `toggle` and a `loadbang` object and the message containing `1` indicates that impulse should occur one sample after the bang. In this case, the bang is generated by the `metro` object every 250 ms. Notice how the object is used to trigger both the message and the `tabwrite~` object. It is important to choose the correct interval for the `metro object`. It should update only after the response of the previous impulse has decayed close to zero. Similarly, it is also important that the length of the array is chosen such that it captures the impulse response. The impulse response of a comb filter with a high delay, for example, can be very long, and if the array size is too small, we may not capture shape of the impulse response at all. If, for example, we only chose an array size of 64 samples and the delay in the comb filter is more than 64, we will not even capture the first echo of the impulse.

To plot the frequency response of the filter in Fig. A.28b, we must make some minor changes to the patch in Fig. A.28a. What we must do is simply combine the patch with the abstraction for computing the Fourier transform in Fig. A.14. The reason for this is that the frequency response of a filter can be found in multiple

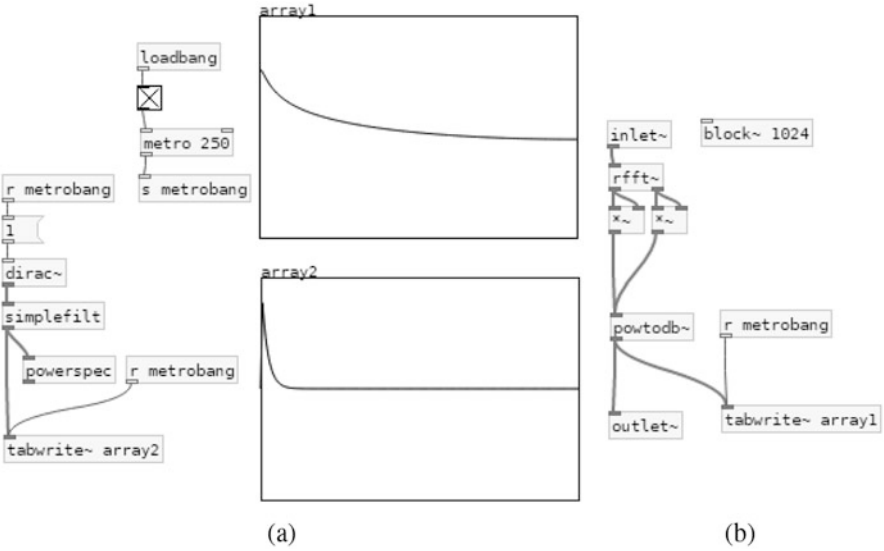


Fig. A.29 Patch for plotting the impulse response and the frequency response of a filter (**a**) and the abstraction `powerspec` used for computing the power spectrum (**b**)

ways, but one of them is to compute the Fourier transform of the impulse response of the filter. Figure A.29a shows how the frequency response, and in the process also the impulse response, can be computed and plotted using the abstraction shown in Fig. A.29b. Note that the size of the FFT used in Pure Data is determined by the block size of the patch or abstraction. Hence, to set the size of the FFT to a desired value, we must compute it in an abstraction and set the block size using the `block~` object. The bangs generated periodically by the `metro` object are used to trigger the `dirac~` object and the `tabwrite~` objects in both the main patch and in the abstraction to synchronize the plotting of the signals (i.e., the impulse response and the frequency response) with the generation of the impulse using the `s` and `r` objects. An alternative way to compute the frequency response would be using white noise as an input, which can be done by replacing the `dirac~` object with the `noise~` object. Since the spectrum of white noise is flat, the spectrum of the filter output would reflect the frequency response of the filter.

References

1. L.E. Kinsler, A.R. Frey, A.B. Coppens, J.V. Sanders, *Fundamentals of Acoustics*, 3rd edn. (Wiley, New York, 1982)
2. K. Steiglitz, *A Digital Signal Processing Primer with Applications to Digital Audio and Computer Music* (Addison-Wesley Publishing Company, Menlo Park, 1996)
3. N.H. Fletcher, T.D. Rossing, *The Physics of Musical Instruments*, 2nd edn. (Springer, New York, 1998)
4. T.D. Rossing, *The Science of Sound*, 2nd edn. (Addison-Wesley Publishing Company, Reading, 1990)
5. A. Gersho, R.M. Gray, *Vector Quantization and Signal Compression* (Kluwer Academic Publishers, Norwell, 1993)
6. S. Haykin, *Communication Systems*, 4th edn. (Wiley, New Delhi, 2000)
7. S.J. Orfanidis, *Introduction to Signal Processing* (Prentice-Hall International, Englewood Cliffs, 1996)
8. A.V. Oppenheim, R.W. Schafer, *Discrete-Time Signal Processing*, 1st edn. (Prentice-Hall, Englewood Cliffs, 1989)
9. M. Frigo, A fast Fourier transform compiler, in *Proceedings ACM SIGPLAN Conference on Programming Language Design and Implementation* (1999)
10. P. Stoica, R. Moses, *Spectral Analysis of Signals* (Pearson Prentice Hall, Upper Saddle River, 2005)
11. S.M. Kay, *Modern Spectral Estimation: Theory and Application* (Prentice-Hall, Englewood Cliffs, 1988)
12. B.G. Quinn, E.J. Hannan, *The Estimation and Tracking of Frequency*. Cambridge Series in Statistical and Probabilistic Mathematics (Cambridge University Press, Cambridge, 2001)
13. J. Dattorro, Effect design, part II: delay-line modulation and chorus. J. Audio Eng. Soc. **45**(10), 764–788 (1997)
14. E. Bedrosian, A product theorem for Hilbert transforms. Proc. IEEE **51**(1), 868–869 (1963)
15. J. Dattorro, Effect design, part III: oscillators: sinusoidal and pseudonoise. J. Audio Eng. Soc. **50**(3), 115–146 (2002)
16. P. Dutilleux, M. Holters, S. Disch, U. Zölzer, Filters and delays, in *DAFX: Digital Audio Effects*, 2nd edn., ed. by U. Zölzer (Wiley, Chichester, 2011)
17. J.O. Smith, An all-pass approach to digital phasing and flanging, in *Proceedings of the International Computer Music Conference* (1984), pp. 103–109
18. V. Välimäki, S. Bilbao, J.O. Smith, J.S. Abel, J. Pakarinen, D. Berners, Virtual analog effects, in *DAFX: Digital Audio Effects*, 2nd edn., ed. by U. Zölzer (Wiley, Chichester, 2011)

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8

19. W. Pirkle, *Designing Audio Effect Plug-Ins in C++* (Focal Press, Taylor & Francis Group, Burlington, 2013)
20. V. Valimaki, T.I. Laakso, Principles of fractional delay filters, in *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing* (2000)
21. J.B. Allen, D.A. Berkley, Image method for efficiently simulating small-room acoustics. J. Acoust. Soc. Am. **65**(4), 943–950 (1979)
22. V. Valimäki, J.D. Parker, L. Savioja, J.O. Smith, J.S. Abel, Fifty years of artificial reverberation. IEEE Trans. Audio Speech Lang. Process. **20**(5), 1421–1448 (2012)
23. J. Dattorro, Effect design, part I: reverberator and other filters. J. Audio Eng. Soc. 45(9), 660–684 (1997)
24. V. Pullki, Spatial sound generation and perception by amplitude panning techniques, Ph.D. dissertation, Helsinki University of Technology, 2001
25. V. Pulkki, T. Lokki, D. Rocchesso, Spatial effects, in *DAFX: Digital Audio Effects*, 2nd edn., ed. by U. Zölzer (Wiley, Chichester, 2011)
26. B.C.J. Moore, *An Introduction to the Psychology of Hearing*, 4th edn. (Academic Press, London, 1997)
27. E. Zwicker, H. Fastl, *Psychoacoustics – Facts and Models*, 2nd edn. (Springer, Berlin, 1999)
28. P. Dutilleux, K. Dempwolf, M. Holters, U. Zölzer, Nonlinear processing, in *DAFX: Digital Audio Effects*, 2nd edn., ed. by U. Zölzer (Wiley, Chichester, 2011)
29. D. Giannoulis, M. Massberg, J.D. Reiss, Digital dynamic range compressor design–a tutorial and analysis. J. Audio Eng. Soc. **60**, 399–408 (2012)
30. U. Zölzer, *Digital Audio Signal Processing*, 2nd edn. (Wiley, New York, 2008)
31. M.G. Christensen, A. Jakobsson, *Multi-Pitch Estimation*. Synthesis Lectures on Speech & Audio Processing, vol. 5 (Morgan & Claypool Publishers, San Rafael, 2009)
32. J. Moorer, The optimum comb method of pitch period analysis of continuous digitized speech. IEEE Trans. Acoust. Speech Signal Process. **22**(5), 330–338 (1974)
33. R.L. Miller, E.S. Weibel, Measurement of the fundamental period of speech using a delay line. Presented at the 51st meeting of the Acoustical Society of America, 1956
34. J.S. Gill, Automatic extraction of the excitation function of speech with particular reference to the use of correlation methods. Presented at the 3rd I.C.A., Stuttgart, Germany, 1959
35. M. Noll, Pitch determination of human speech by harmonic product spectrum, the harmonic sum, and a maximum likelihood estimate, in *Proceedings of the Symposium on Computer Processing Communications* (1969), pp. 779–797

Index

M. G. Christensen, *Introduction to Audio Processing*,
https://doi.org/10.1007/978-3-030-11781-8