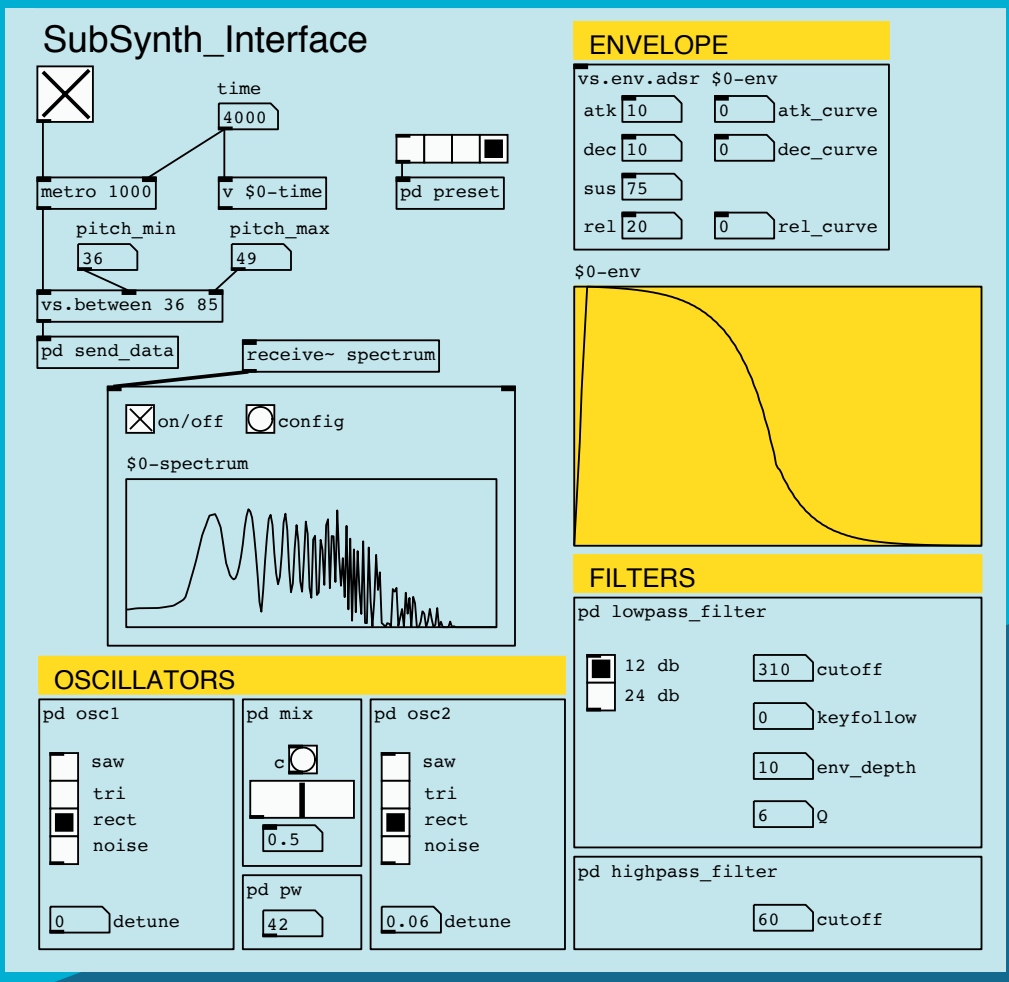
Francesco Bianchi • Alessandro Cipriani • Maurizio Giri



Pure Data: Electronic Music and Sound Design

Theory and Practice • volume 1

ConTempoNet

Francesco Bianchi • Alessandro Cipriani • Maurizio Giri

This is a demo copy of

PURE DATA:
ELECTRONIC MUSIC AND SOUND DESIGN

Theory and Practice - Vol. 1

full version at:
www.contemponet.com
www.virtual-sound.com

Bianchi Francesco. Cipriani, Alessandro. Giri, Maurizio.
Pure Data: Electronic Music and Sound Design : Theory and Practice Vol. 1. /
Includes bibliographical references and index.
ISBN 978-88-992122-1-6
1. Computer Music - Instruction and study. 2. Computer composition.

Original Title: Pure Data: Musica Elettronica e Sound Design - Teoria e Pratica Vol. 1

Theory chapters translation: David Stutz
Practice chapters translation: Richard Dudas, Simone Micheli and David Stutz

Figures realized by: Gabriele Cappellani and Maurizio Refice
Index: Salvatore Mudanò
Language education consultant: Damiano De Paola

Contemponet s.a.s., Rome (Italy)
e-mail posta@virtual-sound.com
posta@contemponet.com
URL: www.virtual-sound.com
www.contemponet.com

CONTENTS

FOREWORD

by Miller Puckette

There was a time when programming a digital computer (or an analog one, which also existed) involved using switches, knobs, and patch cords to program the thing and to feed it its input data. As late as 1980 or so, the Digital Equipment Corporation PDP11 had rows of switches on the front. You could put numbers directly into the machine's registers or memory.

This was the machine on which Music 11, Barry Vercoe's music compiler that later morphed into CSOUND, once ran. To use Music 11 was to function at two levels of software abstraction from the raw PDP11 hardware: the music compiler itself, and the operating system that ran underneath it. These two layers of abstraction saved the user the trouble of learning the PDP11 instruction set or its memory addressing scheme, or even the binary number system that all its digital circuitry was based on. But on the other hand, the user had to buy into various assumptions that the authors of both of these layers made about what their users might want to do.

Some of these assumptions were overt: for instance, to store data persistently one organized it into "files", which weren't real files at all — those were made of paper — but were a convenient abstraction built on an easily grasped metaphor.
At the level of Music 11, an assumption was made that the composer's input could divide neatly into a "score", a sequence of "notes" that presumably was of the composer's primary and individual concern, and an "orchestra" consisting of virtual instruments that composers and researchers frequently swapped around among themselves.

Music 11 itself played the role of the performer, and did so exactly as a computer program could be expected to: it caused the instruments in the orchestra to play the notes in the score, one by one, exactly as prescribed in the score. Vercoe himself saw this as a compromise that had to be made because computers of that time weren't able to generate sound samples interactively, as we would need to do to allow a human performer into the process. The available hardware thus constrained the design of the software, which in turn constrained the composer.

To be sure, the composer had other options. Other electronic instruments of the time offered different trade-offs: "tape studios" offered a much more tactile experience and far more sound storage capacity; analog synthesizers could be played in real time; and hybrid computer/analog systems such as GROOVE combined computers and analog gear to achieve some of the specific advantages of each of the two technologies. To choose to work with Music 11 was to decide that its particular advantages were important and its particular constraints were acceptable.

With this historical example in mind, readers of this book could think about its own particular choice of technology in a similar way. The technology consists of three layers: first, the Virtual Sound library, a toolset for synthesizing, analyzing, and processing digital sounds in real time along with many example applications and audio materials. This library was designed specifically for users of this book. Second, underneath *Virtual Sound*, the Pure Data environment (abbreviated as "Pd") provides the necessary fundamental audio building blocks, and the real-time interactive programming capabilities, upon which *Virtual Sound* is built. Pd itself is used in a wide variety of other situations (forinstance, inside video games or audio "plug-ins", in concert with video and graphics extensions, and/or with customizations appropriate to particular production studios) and often appears with various specialized libraries; so the reader of this book is seeing Pd from one specific angle. Third, underneath Pd is the operating system and hardware, which is presumed to be a laptop or desktop computer with a keyboard, mouse, and display.

I can think of three good reasons to use a computer (instead of, say, a collection of music-specific hardware devices such as synthesizers, hardware controllers, drum machines, and/or turntables). The first is cost. Computers are cheap, and since they can also be used to do many other tasks, there's a good chance a reader of this book already has had to get one for some reason. Second, computers are far more flexible and open-ended in their capabilities than any special-purpose audio hardware could be. Third, and of particular importance: computers are transparent in the sense that you can know what they're doing. Nothing that matters is secret about them. This is often not the case with hardware instruments.

As to Pure Data, its design also aims for the highest possible level of generality and transparency. Unlike much computer software, Pd is open-source, so that there are no secrets about its inner workings. And while many of the design choices underlying Pd will seem constraining to users, they are much less so than those of most music software, which is typically designed with particular workflows and/or musical styles in mind.

And finally, among the many libraries that can be run atop Pd, Virtual Sound is particular in that it is primarily designed for pedagogy. Although a student using this book can't escape having to learn many of the particularities of the Pd environment itself, the authors have worked hard to make the process as straightforward as possible for the beginner, who can then focus on the audio and musical concepts treated in this book, avoiding much of the trouble that besets the student trying to learn computer music technique using Pd alone.

All this is not to claim that this particular stack of technologies is the answer to everyone's needs at all times, but only that it has been crafted with the goals of this book in mind: primarily, as I see it, to open a path along which the reader (whether a student or an independent reader) can get a theoretical and practical understanding of the fundamental techniques for making music with a computer, to make this process as direct and straightforward as possible, and to require a minimum of specialized knowledge in advance.

Returning to the historical context into which I've thrown this introduction, one last point is worth thinking about. What did composers in the decades from 1957 to about 1983 see in the computer as a music-making tool? I think the computer workflow of that day was one that was familiar to Western-trained composers of that time, with their attention to planning, organization, and above all detailed manipulation of numbers and sets, and their ability to work for months or years on a project without getting to hear the music itself until afterward (if even then). Most humans would be discouraged from working like that. Today, on the other hand, the sound pours from your loudspeakers while you're making it up. In the world of electronic music, the special role of "composer" disappears, or perhaps more precisely, the roles of composer, performer, and listener have converged. Music making is a much freer and wider-ranging activity than it was before the advent of electronics, mediated as they are almost always today by computers. There are many possibilities out there to explore, and studying both the theory and the practice of audio programming is the best first step of the way.

(Miller Puckette is the original author of *Max* and *Pure Data*)

INTRODUCTION

This is the first of a series of three volumes dedicated to digital synthesis and sound design. The next volumes will cover a range of additional topics in the realm of sound synthesis and signal processing, including dynamics processing, delay lines, reverberation and spatialization, digital audio and sampled sounds, MIDI, non-linear techniques (such as AM, FM and wave terrain synthesis), granular synthesis and granulation, analysis and resynthesis, convolution, and computer-aided composition.

PREREQUISITES

This first volume will be useful to several levels of reader. Prerequisites for its study are minimal, and include nothing more than rudimentary musical knowledge such as an understanding of notes, scales, and chords, as well as basic computer skills such as saving files, copying and pasting text.

The volume should be equally useful for self-learners and for those studying under the guidance of a teacher. It is laid out as chapters of theoretical background material that are interleaved with chapters that contain practical computer techniques. Each pair of chapters stands together as a unit. We suggest that curricula follow this structure, first touching on theory, then following up with hands-on material, including computer activities. The theoretical chapters are not intended to substitute for more expansive texts about synthesis; they provide, instead, an organic framework for learning the theory that is needed to invent sounds on the computer and to write signal processing programs.

TIME NEEDED FOR STUDY

The time needed for this material will, of course, vary from person to person. Nonetheless, here are two estimates to help in planning, one for learning under the guidance of an expert teacher, and the other for self-learners:

Self-learning
(300 total hours of individual study)

Chapters	**Topic**	**Total hours**
1T+1P+IA	Sound synthesis	100
2T+2A	Additive Synthesis	60
3T+3P+IB	Subtractive Synthesis and Filtering	110
4T+4P	Control Signals	30

Teacher-assisted learning
(60 hours of classroom-based learning + 120 hours of individual study)

Chapters	**Topic**	**Lessons**	**Feedback**	**Studio time**	**Total hours**
1T+1P+IA	Sound synthesis	16	4	40	60
2T+2P	Additive Synthesis	10	2	24	36
3T+3P+IB	Subtractive Synthesis	18	4	44	66
4T+4P	Control Signals	5	1	12	18

THE INTERACTIVE EXAMPLES

The path laid out in the theoretical sections of this book is meant to be accompanied by numerous interactive examples, which are available on the website at the support page for this text. Using these examples, the reader can immediately refer to the example sounds being discussed, as well as their design and elaboration, without having to spend intervening time on the practical work of programming. In this way, the study of theory can be immediately connected to the concrete experience of sounds. The integration of understanding and experience in the study of sound design and electronic music is our objective. This principle is the basis for the entire set of three volumes, as well as for future online materials that will help to update, broaden, and clarify the existing text.

THEORY AND PRACTICE

As we just said, the teaching approach for this book is based, first and foremost, upon an interplay between theory and practice, which we believe is indispensable. One of the glaring problems in the field of digital sound processing is the knowledge gap that exists between experts in theory (who often have neither the time nor the need to tackle concrete technical problems that are so relevant to the actual practice of creating sound) and those enthusiasts, much more numerous, who love to invent and modify sounds using their computers. These enthusiasts persevere, despite gaps in their theoretical awareness and/or in their understanding of how sounds may be modified within the rigid confines forced upon them by their specific software. It is our intention help these users of music software to acquire the deeper understanding that will take them beyond the confines of specific software to access the profound power inherent in the medium.

TEACHING APPROACH AND METHOD OF THIS BOOK

On the basis of the problems and concepts described above, we have tried to fill the information gap by continuing in the direction already begun with the book titled "Virtual Sound" (Cipriani and Bianchini, 2000), also dedicated to sound synthesis and signal processing. The innovations in this new text are substantial, with regard to both the examples provided and a completely different teaching approach. Because very little academic literature is available concerning methods for teaching electronic music, we have approached the problem directly, considering various promising ways to plumb the depths of the subject material. This exercise has led us to an organic teaching method, in which we adopt various ideas and techniques from foreign language textbooks in order to develop a more context-based, open-ended and interactive concept of teaching and learning. In addition to interactive examples, we have included "learning agendas" that detail the specific objectives for each chapter, that include listening and analysis activities, exercises and tests, glossaries, and suggestions for recordings to which to listen. The practical chapters of the book also include many other new features and activities, including the correction, completion, implementation, debugging, testing and analysis of algorithms, the construction of new algorithms from scratch, the replacement of parts

of pre-built algorithms, and reverse engineering (in which the reader listens to a sound and then tries to invent an algorithm to create a similar sound). These activities and tasks are intended to activate the knowledge and practical skills of the reader. When learning a foreign language, there is a gap between what one knows and what one is able to use in practice. It is common for a student's passive vocabulary (the total number of terms that the student can recognize) to be much larger than the active vocabulary that he or she can actually use while speaking or writing. The same is true of a programming language: a student can understand how algorithms work without being able to build them from scratch. The activities in this book that concentrate on replacing parts of algorithms, completing unfinished algorithms, correcting algorithms with bugs, and reverse engineering, have been included in order to pose problems to which the reader is encouraged to find his or her own solutions, causing the learning process to become more active and creative.
When learning a foreign language, students are given replacement exercises (e.g. "replace the underlined verb in the following phrase: I wish I could go out"), correction exercises (e.g. "correct the following phrase: I want to went home"), and sentences to be completed (e.g. "I'd like to ... home"). In this context, it is vitally important for the student to work at these activities in order to avoid an excessively passive approach to learning. Our approach, likewise, not only involves interactions between the perception of sounds and the knowledge deriving from reading the book and doing the practical activities, but also interactions between these two factors and the user's own *skills* and *creativity*.

This method is not based on a rigidly linear progression, but is rather a network that enables the reader to acquire knowledge and practical skills through an interaction of four separate dimensions: learning of the theoretical concepts, learning to use the Pure Data program, interacting with example material, and constructing algorithms.

This text is based on the book "Electronic Music and Sound Design – Theory and Practice with Max 8" by A. Cipriani and M.Giri, whose structure and educational vocation are maintained. The section dedicated to practice, where the techniques described in the theory chapters are exemplified, refers to the software Pure Data, which (just like Max) was created by Miller Puckette, and shares the general settings and many other features with Max.

Pure Data is a visual programming language, that allows you to connect graphic objects. These objects, which can perform calculations or process audio signals, can be connected to create very complex entities such as synthesizers and signal processors, or even authentic automatic sound devices.

Pure Data is free and *open source*. It can be downloaded for free and used on the most common operating systems. You can also modify the source code in order to create custom versions of the software. This set of features has led to the birth of an ever-growing community of musicians, programmers, and enthusiasts, and even to some interesting projects, such as *Pd-extended*

(which, unfortunately, is no longer updated, and that, before ceasing to exist, was the reference version of the community). Hence, this book deals with the original version of Pure Data (sometimes referred to as *Pd-vanilla*), which is currently kept up-to-date and supported by Miller Puckette himself. The advantage of the *vanilla* version is that it ensures the compatibility with every platform, and most importantly, a longer continuity over time than the alternative versions.

The *vanilla* version of the software contains a rather small native-object library, and for this reason this book is accompanied with a library of *abstractions* (which are objects created within the environment, i.e.: made up of already-existing Pd objects) that allows you to boost the features of the program and to easily perform all those operations that would otherwise be really difficult to carry out (especially for a newbie who has just started learning the basics).

PRACTICAL INFORMATION

Many indispensable materials accompany this book, among them, interactive examples, patches (programs written in Pure Data), sound files, programming libraries, and other materials.
These can be found at the support page for this text.

Interactive Examples

During the study of a theory chapter, before moving on to the related practical chapter, it will help to use the interactive examples or video examples. Working with these examples will aid in the assimilation of the concepts raised by the theory.

Example Files

The example files (patches), are created to be used with Pure Data, which is freely downloadable from Miller Puckette's site http://msp.ucsd.edu/software.html. As we have already mentioned, the book deals with the *vanilla* version (without extensions) of Pure Data.

Alternating Theory and Practice

In this book, theoretical chapters alternate with chapters which are geared towards programming practice. Because of this, the reader will find himself taking on all of the theory for a given chapter before passing to the corresponding practical chapter. An alternative to this approach would be to read a single section from the theory, and then go directly to the corresponding section of the practical chapter. (For example, 1.1T and 1.1P, then 1.2T and 1.2P, etc.

The Interludes

Note that there are two "technical interludes", the first between the first and second chapters, and the second between the third and fourth chapters. These interludes, named respectively "Interlude A" and "Interlude B", are dedicated specifically to the Pure Data language. They don't relate directly to any of the theoretical discussions, but they are very necessary for following the code

traced out in the book. After having tackled the theory and practice of the first chapter, before moving on to the second chapter, it will benefit the reader to study Interlude A. Likewise, Interlude B is meant to be studied between Chapters 3 and 4.

Learning Pure Data

Learning Pure Data (and, in general, learning synthesis and sound processing) requires effort and concentration. In contrast to much commercial music software, Pure Data provides flexibility to the programmer, and this design choice provides those programming with Pd many alternative ways to build a given algorithm. To benefit from this the recommendations of the book and to code in a systematic way. Pd is a true musical instrument, and learning to play it should be approached as one would approach the study of a traditional instrument (such as a violin). As with any instrument, the reader will find it necessary to practice regularly, and to stay sharp on basics while gradually acquiring more complex techniques. By approaching the software in this way, fundamental techniques and technical insights can be retained once they have been acquired.

Bibliography

The decision was made to limit the bibliography in this book to a list of only the most absolutely essential reference works, and, of course, a list of the books and articles cited in the text. A more comprehensive bibliography is available online.

Before Beginning

To begin working with this book, you will need to download the interactive programming examples, which you will find at the support page for this text. While reading the theory chapters, you will find constant references to the examples contained in this downloadable archive. To work interactively with the programming chapters of the book, you will need to download the Virtual Sound Macro Library from the support page mentioned above. It will also be necessary to install Pure Data, which is available at Miller Puckette's website: http://msp.ucsd.edu/software.html.

The support page for this text contains detailed instructions regarding how to install Pd and the macro library correctly. Always check the support page for patches (Pd programs) related to the practice chapters of this book, as well as the audio files for the reverse engineering exercises.

Comments and Suggestions

Corrections and comments are always welcome.
Please contact the authors via email at:

Francesco Bianchi frabianchi72@gmail.com
Alessandro Cipriani a.cipriani@edisonstudio.it
Maurizio Giri maurizio@giri.it

THANKS

We wish to thank Gabriele Cappellani, Vincenzo Core and Simone Micheli for their patience and long hours of work, and Andrew Bentley, Richard Dudas and Miller Puckette for their generosity and support.

DEDICATIONS

This text is dedicated to Riccardo Bianchini, who would have wanted to participate in the production of this teaching text, but who, unfortunately, passed away before the work began. We have collected some of his materials, revised them, and cited them in a few of the sections on theory. This seemed to be a way to have Riccardo still with us. A particular thanks goes to Ambretta Bianchini for her great generosity and sensitivity during these years of work.

Enjoy the reading!
Francesco Bianchi, Alessandro Cipriani and Maurizio Giri

LIST OF SYMBOLS

• ACTIVITIES AND INTERACTIVE EXAMPLES

• INTEGRATED CROSS-FUNCTIONAL PROJECTS

• FUNDAMENTALS CONCEPTS

• TECHNICAL DETALS

• TEST WITH SHORT ANSWERS

1T
INTRODUCTION TO SOUND SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Basic skills in using computers
  (operating a computer, managing files and folders, audio formats, etc.)
- Minimal knowledge of music theory (semitones, octaves, rhythms, etc.)

LEARNING OBJECTIVES

Knowledge

- To learn about the signal paths one uses in sound synthesis and signal processing
- To learn about the principal parameters of sound and their characteristics
- To learn how pitch and sound intensity are digitally encoded
- To learn about musical intervals in different tuning systems
- To learn about audio file formats

Skills

- To be able to hear changes of frequency and amplitude and to describe their characteristics
- To be able to hear the stages of the envelope of a sound or a glissando

CONTENTS

- Computer-based sound synthesis and signal processing
- Theory of timbre, pitch, and sound intensity
- Theory of glissandi and amplitude envelopes
- The relationship between frequency, pitch, and MIDI encoding
- Introduction to sampled sound
- Introduction to panning

ACTIVITIES

- Interactive examples

TESTING

- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS

- Fundamental concepts
- Glossary

1.1 SOUND SYNTHESIS AND SIGNAL PROCESSING

The use of computers in music has enabled composers and musicians to manage and manipulate sound with a precision and a freedom that is unthinkable with acoustic instruments. Thanks to the computer, it is now possible to model sound in every way imaginable. One might say that while the traditional composer working with traditional instruments composes *using* sounds, the electronic composer composes *the sounds themselves*.

The same thing has happened in animation graphics: thanks to the computer it is now possible to create images and film sequences that are extremely realistic, and that would have been impossible to produce by other means. Almost all cinematic special effects are now produced with computers; it is becoming commonplace to find virtual entities sharing the screen with flesh-and-blood actors.

These newfound possibilities are the result of passing from the analog world into the digital world. The digital world is a world of numbers. Once an image or a sound has been converted into a sequence of numbers, those numbers can be subjected to transformations, since numbers are easily and efficiently analyzed and manipulated by computers. The process of digitization, precisely defined as that of transforming an item of data (a text, a sound, an image) into a sequence of numbers, is the technique that makes this all possible.[1]

This text will concentrate on two subjects: sound synthesis and signal processing. **Sound synthesis** means the electronic generation of sound. In practice, you will find that the possibilities for creating sound are based largely on a few selected parameters, and that you can obtain the sonorities you seek by manipulating these parameters.

Signal processing in this context means the electronic modification of a sound, whether the sound of a recorded guitar or a sound generated by using a particular type of sound synthesis.

DIGITAL SYNTHESIS OF SOUND

When generating sound using a programming language designed for sound synthesis and signal processing, we specify a desired sound by constructing a "virtual machine" of our own design (realized as an **algorithm**[2]) , and by specifying a series of instructions which this machine will use to create the sound.

Once we have written this sequence of instructions, the programming language we're using (Pure Data for example) will *execute* our instructions to create a *stream of digital data* in which all of the characteristics of the

[1] We will broaden this concept during the course of the chapter.

[2] An algorithm is a sequence of instructions, written in a programming language, that enables a computer to carry out a defined task.

sound or sounds that we have specified will be rendered.[3] Between the time that this stream of digital data is generated and the time that we actually hear the sound, another fundamental operation occurs. The computer's **audio interface** transforms the digital data into an electrical signal that, when fed to an amplifier and loudspeakers, will produce the sound. The audio interface, in other words, converts the digital data into an analog voltage (a process often abbreviated as "D/A conversion"), allowing us to hear the sounds that are represented by the stream of digital data. (fig. 1.1).

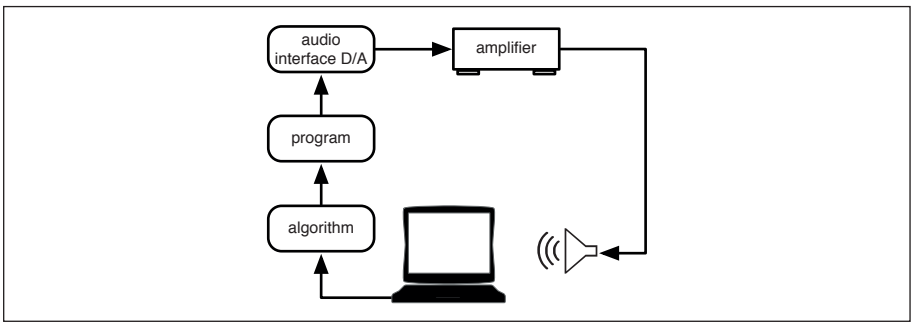


Fig. 1.1 Realtime synthesis

We can also capture the stream of data to our hard disk as an *audio file*, which will enable us to hear the result of our algorithmic processing as many times as we'd like.

When the stream of data goes directly to the audio interface as it is processed, so that there are only few milliseconds between the processing and the listening of the synthesized sound, one speaks of **realtime synthesis**. When the processing of sound is first calculated entirely and saved to an audio file (which can be listened to later) one speaks of **non-realtime** or **offline synthesis**. (In this context the latter term is not a technical one, but it is widely used.)

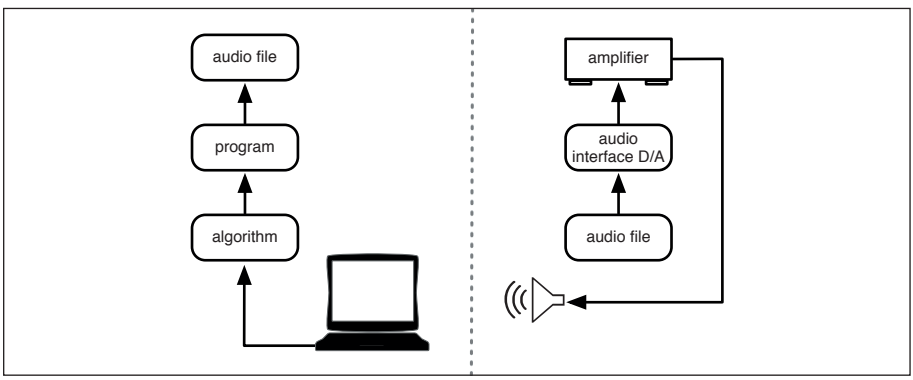


Fig. 1.2 Non-realtime synthesis and listening as separate actions

[3] In numeric form.

SIGNAL PROCESSING

Signal processing is the act of modifying a sound produced by a live source, for example through a microphone, or from a pre-existing audio file already stored in your computer. It is possible to do signal processing in various ways. We see three possibilities:

Pre-existing sound, saved separately as a sound file which is processed offline

The sound of a flute, for example, is recorded to disk using a microphone connected to the audio interface, which performs the analog-to-digital conversion.[4] We implement an algorithm in which we specify the sonic modifications to be made to the original audio file. Once executed, this program will create a new audio file containing the now-modified sound of the flute. We can then listen to the processed sound file at any time by *playing the file* (through the audio interface).

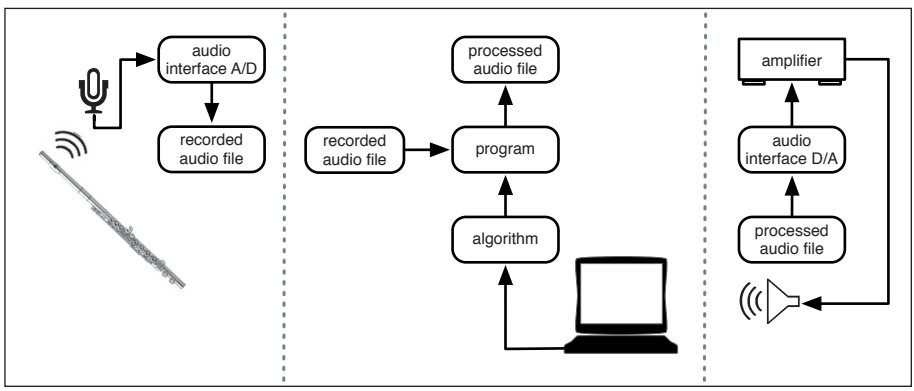


Fig. 1.3 Example of offline sound processing

Pre-recorded sound, which is then processed in realtime

A sound, already recorded in the computer as in the first example, is streamed from a pre-existing sound file. The processing program, while executing commands to modify the streamed sound file, also routes the processed sound file directly to the audio interface for listening. The program, although it is processing in real time, can also record the resulting stream into an audio file for later listening, as in fig. 1.4.

Realtime sound, processed immediately

Sound comes from a live source. As in the preceding example, the processing program, executing commands, routes the processed sound directly to the audio interface.

[4] A transformation of a physical sound into a sequence of numbers.

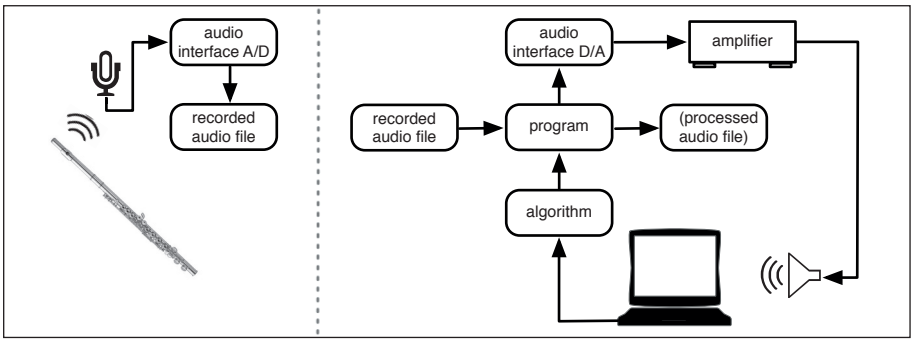


Fig. 1.4 Example of realtime sound processing on pre-existing sound

Naturally, in this case also, the program can record the processed sound as an audio file, as shown in figure 1.5.

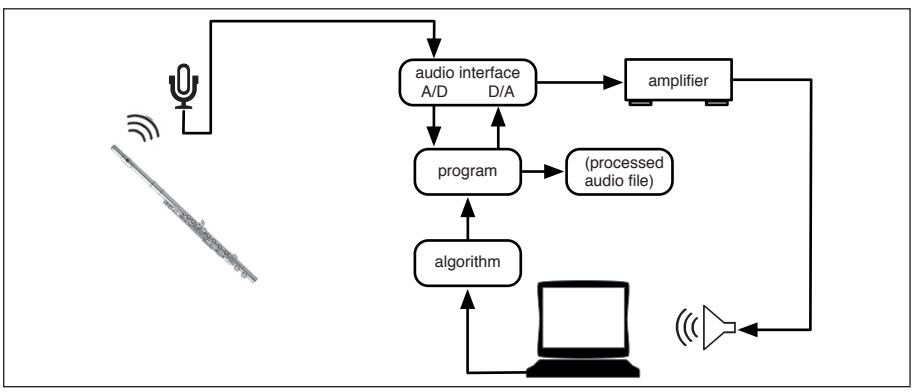


Fig. 1.5 Example of realtime sound processing on live sound

We define a **DSP system** as an integrated hardware and software system (computer, audio interface, programming language.) that enables the processing and/or synthesis of sound. The term **DSP** is an acronym for digital signal processing.

REALTIME VERSUS OFFLINE PROCESSING

We have seen that both synthesis and signal processing can occur either in realtime or offline. At first glance, the more valuable approach would seem to be realtime, because this method provides immediate feedback and an opportunity to weigh the appropriateness of the algorithm being evaluated, as well as to tune and tweak the code if necessary.

What cause is served, then, by deferring processing to offline status?

The first reason is simple: to implement algorithms that the computer cannot execute in realtime, due to their complexity. If, for example, the computer needs two minutes of time in order to synthesize or to process one minute

of sound, one has no alternative but to record the result to disk in order to be able to listen to it without interruption once the processing has finished.
At the dawn of computer music, all of the processing done for synthesis and effects was performed offline, because the processing power to do realtime calculation did not exist. With the increasing power of computers, it began to be possible to perform some processing directly in realtime, and, over time, the processing power of personal computers grew enormously, enabling them to do most synthesis and processing in realtime. But as computing power continues to grow, new possibilities are continually imagined, some of which are so complex that they can only be achieved offline. The need for offline processing will never disappear.
There also exists a second reason: a category of processing that is *conceptually* offline, independent of the power of the computer. If we want, for example, to implement an algorithm that, given a sequence of musical sounds from an instrument, will first break the sequence down into singles notes and then reorder those notes, sorting from the lowest to the highest pitch, we *must* do this processing offline. To realize this algorithm, we would first need the entire sequence, most likely recorded into an audio file in a way that the computer could analyze; the algorithm could then separate the lowest note, then the next-lowest, and so forth until finished. It should be obvious that this kind of analysis can only take place offline, only after the completion of the entire sequence; a computer that could handle this kind of algorithm in realtime (that is to say, while the instrument was playing the sequence) would be a computer so powerful that it could see into the future!
A final advantage of non-realtime processing is the prospect of *saving time*! Contrary to what one might initially think, realtime processing is not the fastest computing speed possible. We can imagine, for example, that we might modify a 10 minutes sound file using a particular type of processing. If this modification were to happen in realtime, it would obviously take 10 minutes, but we might also imagine that our computer had enough power to render this processing offline in 1 minute. In other words, the computer could render the calculations for this particular hypothetical operation at a speed 10 times faster than realtime. Offline processing, in this case, would be far more convenient than realtime processing.

1.2 FREQUENCY, AMPLITUDE, AND WAVEFORM

Frequency, amplitude and waveform are three basic parameters of sound.[5] Each one of these parameters influences how we perceive sound, and in particular:

a) our ability to distinguish a lower pitch from a higher one (frequency)
b) our ability to distinguish a loud sound from a soft sound (amplitude)
c) our ability to distinguish different timbres (waveform)

[5] We refer here to the simplest forms of sound. (i.e. we will later see how the parameter of timbre actually depends on several factors.)

Let's look at a table (taken from Bianchini, R., 2000) of the correspondences between the physical features of sound, musical parameters, and perceived sonority.

CHARACTERISTIC	PARAMETER	PERCEPTUAL SENSATION
Frequency	Pitch	High ↔ Low
Amplitude	Intensity	Forte ↔ Piano
Waveform	Timbre	Sound color

TABLE A: correspondences between sound characteristics, musical parameters and perceived sonority.

FREQUENCY

Frequency is the physical parameter that determines the pitch of a sound, that is, it is the feature that allows us to distinguish between a high-pitched sound and a low-pitched sound. The range of frequencies that is audible to humans extends from about 20 to about 20,000 hertz, that is to say, from about 20 to about 20,000 cycles per second.[6] (We'll define cycles per second in a moment.) The higher the frequency of a sound, the higher its pitch will be.
But what do we mean by hertz or "cycles per second"? To understand this, we refer to the definition of sound given by Riccardo Bianchini:

"The term 'sound' signifies a phenomenon caused by a mechanical perturbation of a transmission medium (usually air) which contains characteristics that can be perceived by the human ear.[7] Such a vibration might be transmitted to the air, for example, by a vibrating string (see fig. 1.6). The string moves back and forth, and during this movement it pushes the molecules of air together on one side, while separating them from each other on the other side. When the motion of the string is reversed, the molecules that had been pushed together are able to move away from each other, and vice versa.
The compressions and expansions (that is to say, the movements of air molecules) propagate through the air in all directions. Initially, the density of molecules in

[6] The highest frequency that someone can hear varies from individual to individual. Age is also a factor. As we get older, our ears become less sensitive to high frequencies.

[7] There are many theories about the nature of sound: Roberto Casati and Jérôme Dokic argue that the air is a medium through which the sound is transmitted, but that sound itself is a localized event that resonates in the body, or in the mechanical system that produces the vibration. (Casati, R., Dokic, J. 1994). Another point of view is expressed by Frova: "with the term 'sound', one ought to signify the sensation, as manifest in the brain, of a perturbation of a mechanical nature, of an oscillatory character, which affects the medium interposed between source and listener." (Frova, A., 1999, p.4).

air is constant; each unit of volume (for example, a cubic centimeter) contains the same number of molecules.

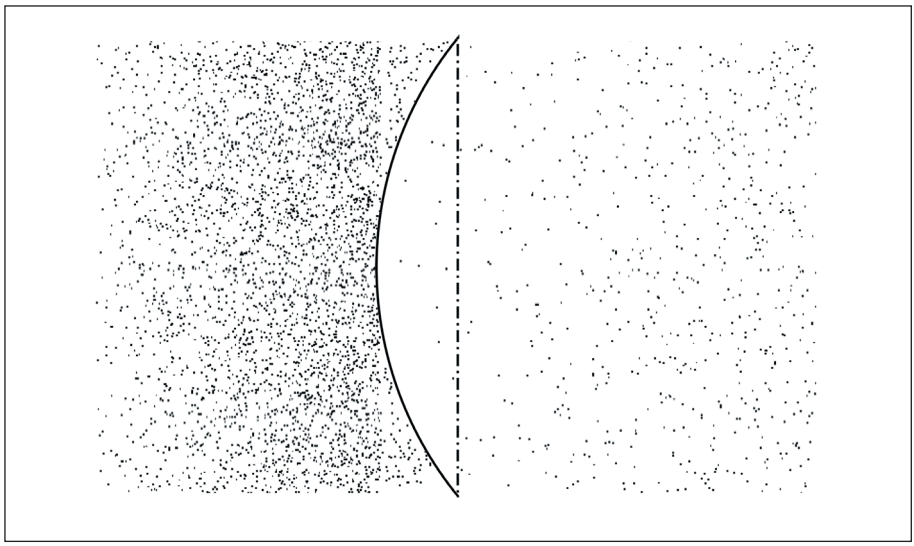


Fig. 1.6 Vibration of a string

This density can be expressed as a value called *pressure*. Once the air is disturbed, the pressure value is no longer constant, but varies from point to point, increasing where molecules are pushed together and decreasing where the density of the molecules is rarefied (see fig. 1.7).

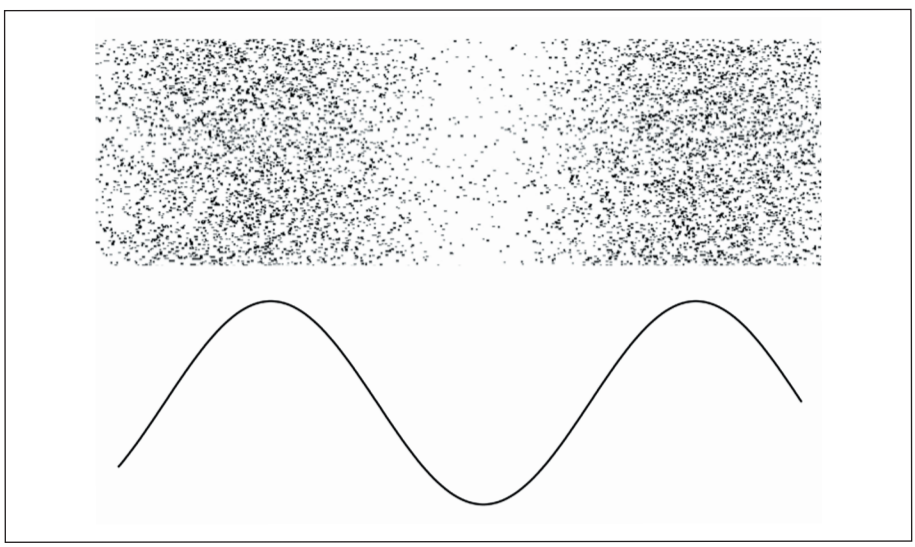


Fig.1.7 Compression and rarefaction of air molecules

Pressure can be physically studied either in terms of space (by simultaneously noting the pressure at multiple points at a given moment), or from the point of

time (by measuring the pressure at a single location as a function of time). For example, we can imagine that if we were located at a specific point in space, we might observe a series of condensations and rarefactions of the air around us, as in figure 1.8.

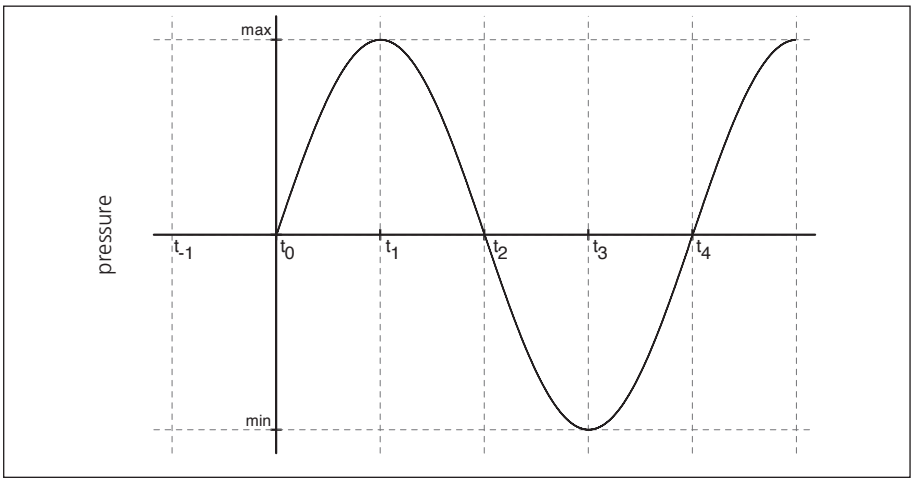


Fig.1.8 A graphical representation of compression and rarefaction

At time t_{-1}, which occurs immediately before t_0, the air pressure has its normal value, since the cyclic disturbance has not yet reached our point of observation. At instant t_0, the disturbance arrives at our observation point, pressure starts to rise, reaches a maximum value at time t_1, and then decreases until it returns to normal at time t_2. It continues to decline, reaching its minimum value at t_3, after which pressure returns to its normal value at t_4; the pattern then repeats. What has been described is a phenomenon called a **cycle**, and an event that always repeats in this way is called *periodic*.[8] The time required to complete a cycle is said to be the **period** of the wave, which is indicated by the symbol T and is measured in seconds (s) or in milliseconds (ms). The number of cycles that are completed in a second is defined as *frequency*, and is measured in hertz (Hz) or cycles per second (cps).
If, for example, a sound wave has period T = 0.01s (1/100 of a second), its frequency will be 1/T = 1/0.01 = 100 Hz (or 100 cycles per second)."(ibid)

While examining figure 1.9, listen to the sounds of Interactive Example 1A.[9] We can see (and hear) that increasing the number of cycles per second (Hz) corresponds to making a sound higher in pitch.

[8] Mathematically a waveform is said to be periodic if it is repeated regularly for an infinite time. In the practice of music, of course, we can satisfy ourselves with periods much shorter than infinity! We will say that a wave is "musically periodic" when it displays enough regularity to induce a perception of pitch that corresponds to the period of the wave. We'll discuss this issue in more detail in Chapter 2.

[9] Please note that interactive examples and other supporting materials to the book can be found on the website at the support page for this text

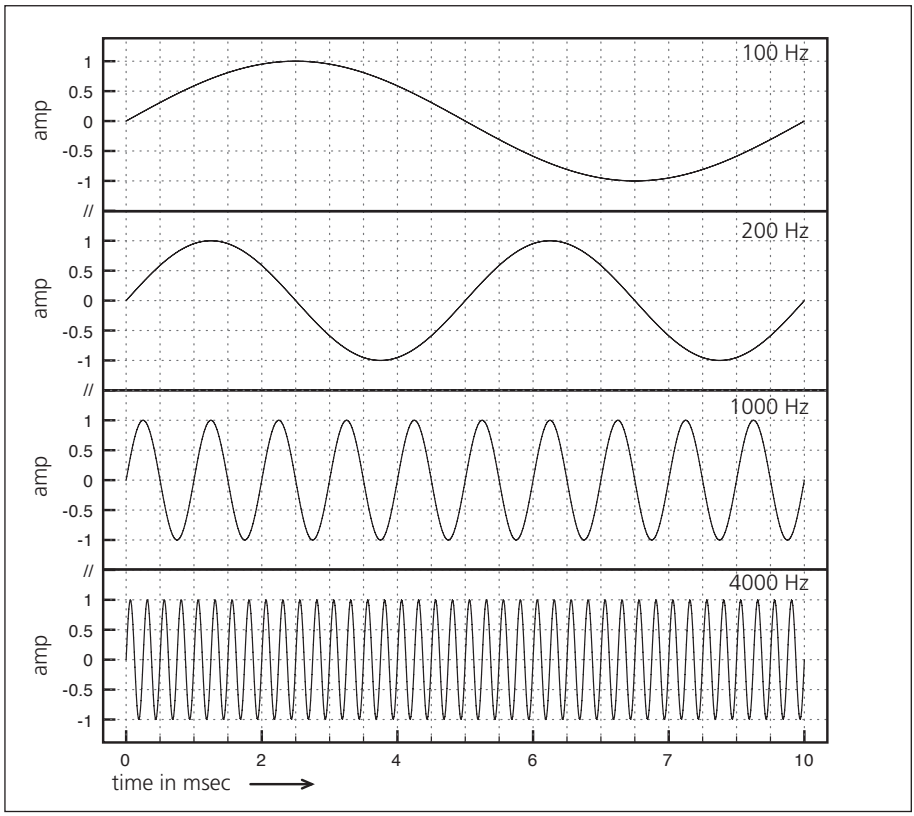


Fig.1.9 Four sounds of different frequencies

INTERACTIVE EXAMPLE 1A • *FREQUENCY*

From the instant that it propagates in space, a wave has a length that is inversely proportional to its frequency. Let's clarify this concept: the speed of sound in air (the speed at which waves propagate from a source) is about 344 meters per second.[10] This means that a hypothetical wave of 1 Hz would have a length of about 344 meters, because when it has completed one cycle, one second will have passed, and during this second, the wavefront will have traveled 344 meters. A wave of 10 Hz, however, completes 10 cycles in a single second, which fill 344 meters with an arrangement of 10 cycles of 34.4 meters each; each cycle physically occupies a tenth of the total space available.

[10] For the record, this speed is reached when the temperature is 21°C (69,8°F). The speed of sound is, in fact, proportional to the temperature of the medium.

By the same reasoning, a 100 Hz wave has a wavelength of 3.44 meters. We see that frequency decreases with increasing wavelength, and the two quantities are, as we have said, inversely proportional.

AMPLITUDE

The second key parameter for sound is **amplitude**, which expresses information about variations in sound pressure, and which allows us to distinguish a loud sound from one of weaker intensity.

A sound pressure that is weaker than the human ear can hear is said to lie below the **threshold of hearing**, while the maximum sound pressure that can be tolerated by the human ear is defined as the **threshold of pain**. Exposure to sounds above the threshold of pain results in physical pain and permanent hearing damage.

In the wave depicted in figure 1.10, the maximum pressure value is called the **peak amplitude** of the sound wave, while the pressure at any point is called **instantaneous amplitude**.

When we generically refer to the "amplitude of a sound", we are referring to the peak amplitude for the entire sound (see figure 1.10).

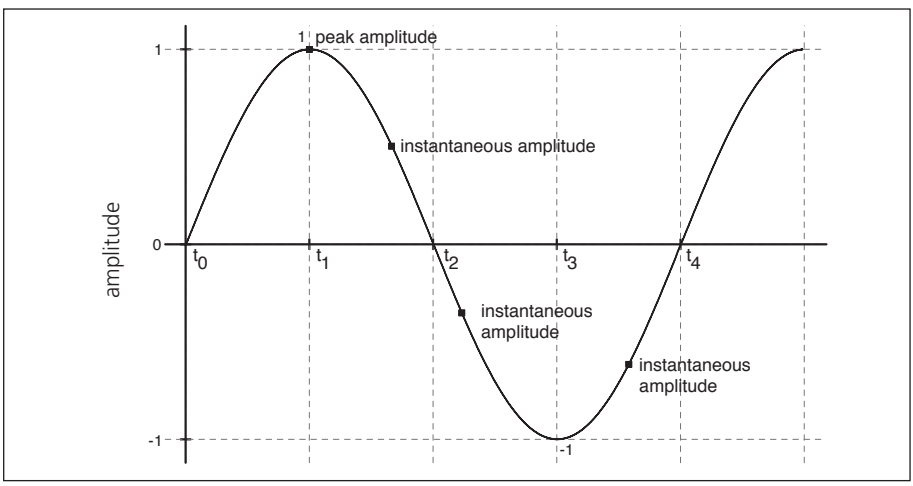


Fig.1.10 Amplitude of a sound

If we show a wave that has a peak amplitude of 1, as in the example, we will see a wave that starts from an instantaneous amplitude of 0 (at time t_0), rises to 1 at time t_1, returns to pass through 0 at time t_2, continues to drop until it reaching its minimum value of -1 at time t_3, after which it rises again to the value 0 at time t_4, and so on. When we represent amplitude this way, we are looking at it as a function of time. The process of digitization transforms such a function into a series of numbers between 1 and -1, and the values thus obtained can be used to graph the wave form (fig. 1.11). The relative position that a wave cycle

occupies at a given instant is called its **phase**, and we will explore the concept of phase in more detail in Section 2.1.

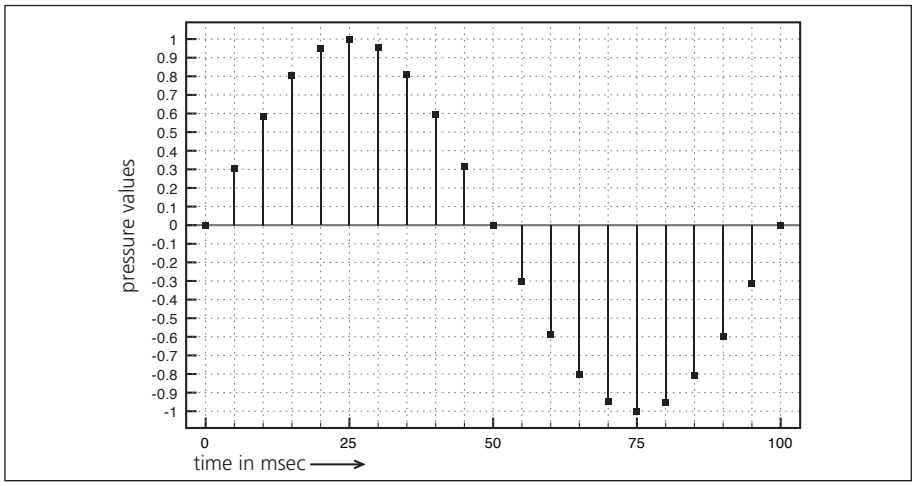


Fig. 1.11 Digital representation of a waveform

Comparing the graph to the real wave (i.e. the physical succession of air compressions and rarefactions), we can see that compression corresponds to positive numbers, rarefaction to negative numbers, and that the number 0 indicates the original stable pressure. (The absence of any signal is, in fact, digitally represented by a sequence of zeros.) Values representing **magnitude** (or amplitude values) are conventionally expressed as decimal numbers that vary between 0 and 1. If we represent the peak amplitude with a value of 1, we will have oscillations between 1 and -1 (as in the previous example), whereas if we were to use 0.5 as the peak amplitude (defined as half of the maximum possible amplitude), we would have oscillations between the numbers 0.5 and -0.5, and so on. (See figure 1.12.)

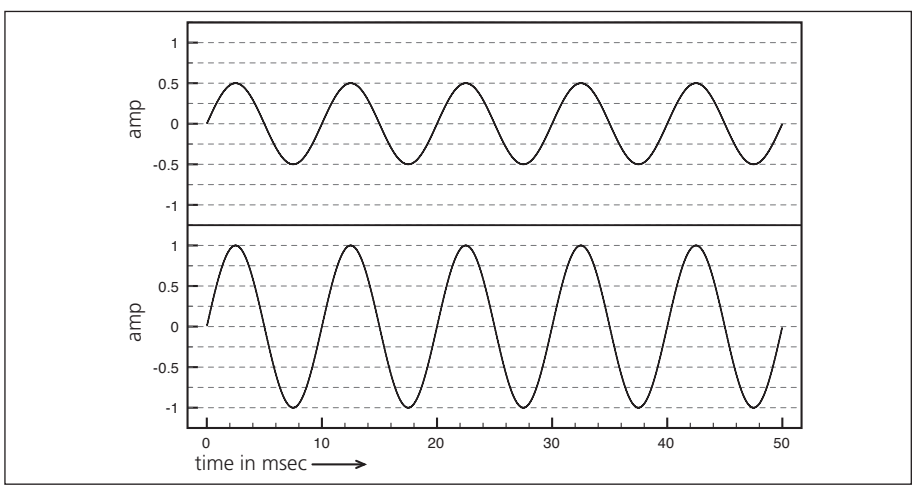


Fig.1.12 Two sounds with differing amplitudes

If the amplitude of a wave being output by an algorithm exceeds the maximum permitted by the audio interface (a wave, for example, that ranges between 1.2 and -1.2, being output by an interface that cannot accurately play values greater than 1), all of the values exceeding 1 or falling below -1 will be limited respectively to the maximum and the minimum value: offending values will be "clipped" to the values 1 or -1. Clipped waves are deformed, and because of this, their sound is distorted[11] (see fig. 1.13).

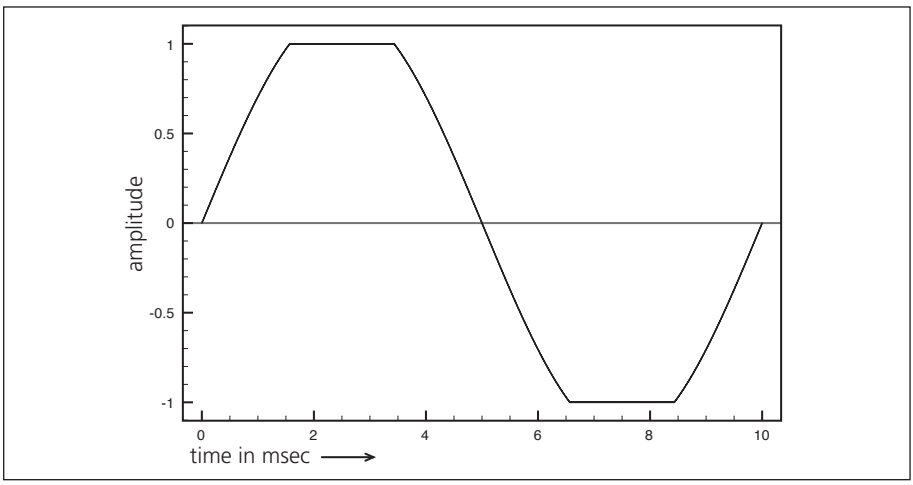


Fig.1.13 A "clipped" waveform

In most software, besides using "raw" numbers to represent amplitude, it is also possible to indicate levels by using **dBFS**: the symbol dB indicates that the level is measured in *deciBels*, and the acronym FS stands for *Full Scale* – thus the entire abbreviation can be read as *"deciBels relative to full scale"*. Whereas raw amplitude measurements represent the difference between a sound pressure measurement and some normal pressure, dBFS is instead defined as the relationship of a sound pressure at a given moment to a reference pressure (which is typically 0 dB in digital audio). 0 dBFS represents the highest level of accurately reproducible pressure (corresponding to the maximum amplitude), and lower levels are indicated by negative values.

Using this scale, the raw amplitude 1, as used in the preceding examples, would correspond to 0 dBFS, while a magnitude of 0.5 would correspond to approximately -6 dB, and an amplitude of 0.25 would fall still lower on the scale at approximately -12 dB. It follows that a reduction of 6 dB corresponds to a halving of the amplitude, whatever the level may be. This kind of relative measurement is very useful because you can use it while working with sounds of unknown loudness.

[11] As we will see in Section 5.1, harmonic distortion is the modification of a signal due to the alteration of its waveform, which results in the introduction of spectral components that are not found in the original signal.

No matter how strong the signal, we know that in order to double it, it will need to increase by 6 dB. Measurement in dB, in contrast to other measurements, is not absolute but relative; it allows us to measure and manipulate the relationship between one sound pressure level and another without knowing their absolute magnitudes.

Here is a useful rule to remember: to reduce the magnitude of a signal by a factor of 10 (in other words, to reduce it to one tenth of the original amplitude) we must reduce the signal by 20 dB. Likewise, to increase a signal tenfold, raise it by 20 dB. It follows that an increase of 40 dB would increase a signal by 100 times, 60 dB by 1000, etc. For a more detailed discussion of this, see "Technical Details" at the end of this section.

Let's look at a table relating raw amplitudes, normalized to a maximum value of 1, to amplitudes measured in dBFS.

Amplitude	dBFS
1	0
0.5	-6
0.25	-12
0.125	-18
0.1	-20
0.01	-40
0.001	-60
0.0001	-80
0	-inf

TABLE B: relationship between raw amplitude and dBFS

As we said, the deciBel is not an absolute magnitude, but is instead a relationship between two quantities, and so there is no absolute measure of 0 dB. Instead, you are free to define 0 dB as you wish, to use as a benchmark against which you will measure a given sound pressure. Unlike in digital audio, where we will usually specify that 0 dB is the *maximum* value reproducible in a given system, analog acousticians often use 0 dB to represent the *minimum* level for their amplitude scale, with *positive* numbers representing louder values.

The following list itemizes, in an approximate way, pressure levels for some common environments (measured in dB SPL at 1 meter of distance)[12] . Amplitude in this table, as you can see, is not represented using 0 dB as the maximum pressure level (as it would be in digital audio, where the amplitudes below the maximum possess negative values, such as -10 dB or -20 dB). On the contrary, these amplitudes are represented using 0 dB as a reference point for the "weakest perceptible sound," leaving all other values to be positive numbers greater than 0.

[12] The acronym SPL stands for Sound Pressure Level.

140	the threshold of pain
130	a jet taking off
120	a rock concert
110	a symphony orchestra fortissimo
100	a truck engine
90	heavy traffic
80	a retail store
70	an office
60	normal conversation
50	a silent house
40	a night in the countryside
30	the rustle of leaves
20	wind
10	a light breeze
0	the weakest perceptible sound

INTERACTIVE EXAMPLE 1B • *AMPLITUDE*

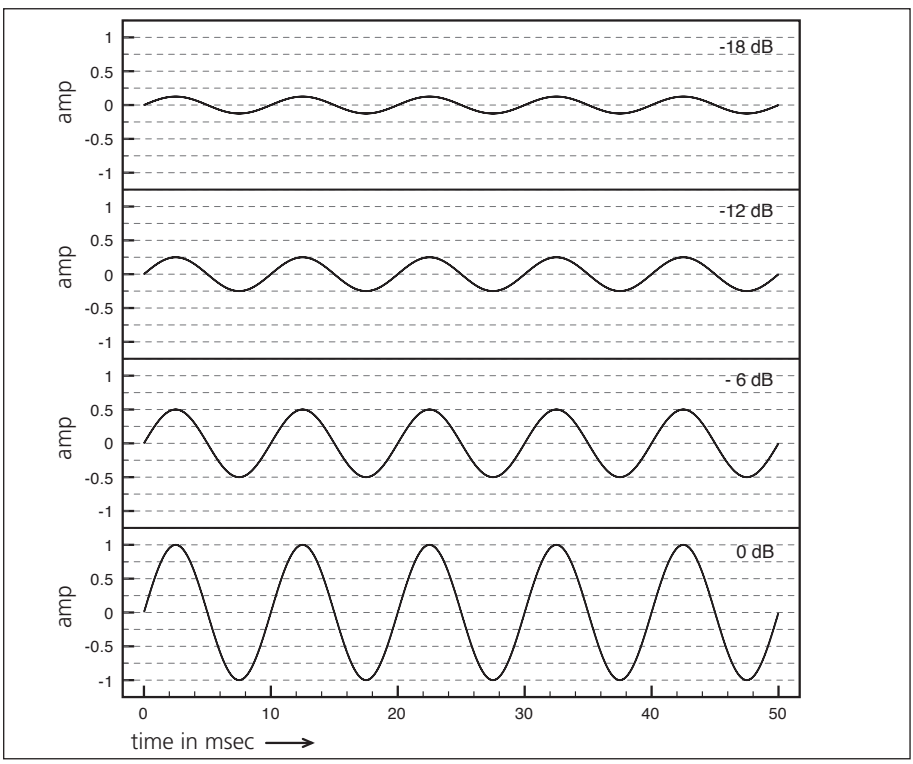


Fig.1.14 Four sounds with their amplitudes progressively doubled

From the psychoacoustic point of view, the intensity of a sound influences the perception of its pitch. Without going into too many details, it suffices to note that above 2,000 Hz, if we increase the intensity of a sound while maintaining fixed frequency, we will perceive that the pitch is rising, while below 1,000 Hz, as intensity increases, there will be a perceived drop in the pitch. On the other hand, frequency also influences our perception of its intensity: the sensitivity of the ear to volume decreases at higher frequencies, increases in the midrange, and decreases greatly at low frequencies. This means that the amplitudes of two sounds must differ, depending on their frequencies, in order to produce the same perceived sensation of intensity. A low sound needs more pressure than is required for a midrange sound to register with the same impact.
There is a graphical representation of the varying sensitivity of the ear to different frequencies and sound pressures. In figure 1.15 we see this diagram, which contains **isophonic curves** that represent contours of equal loudness. The vertical axis indicates the level of pressure in dB, while the horizontal axis represents frequency. The curves are measured using a unit called a **phon**[13] and indicate, within the audible frequency range, the sound pressure needed to produce equal impressions of loudness for a listener.[14]

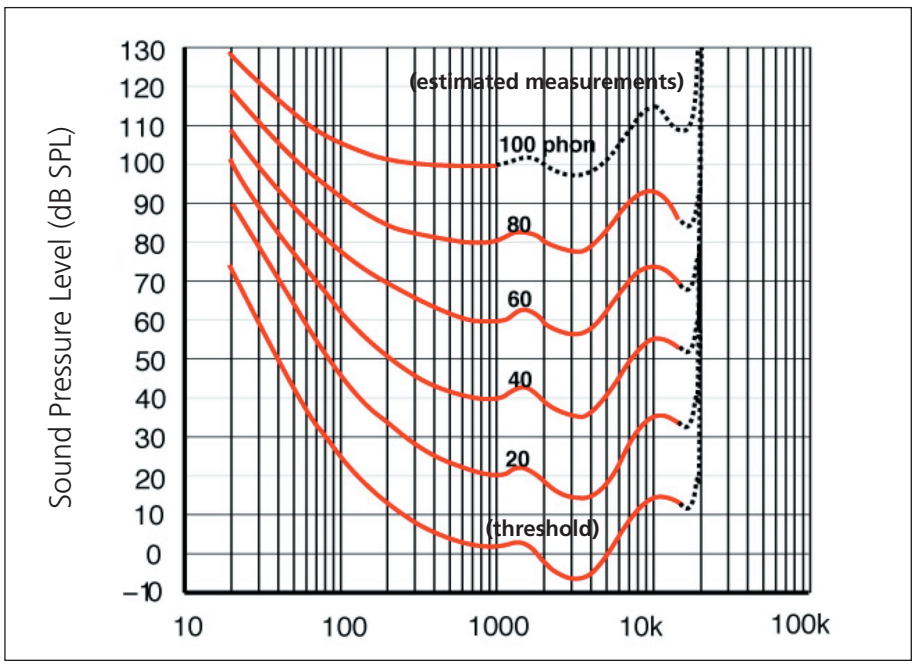


Fig. 1.15 Diagram of equal loudness contours (ISO 226:2003)

[13] The phon is a measure of perceived level of intensity which takes psychoacoustics into account. 1 phon is equal to 1 dBFS at a frequency of 1000 Hz.

[14] The diagram of equal loudness contours is named after H. Fletcher and W.A. Munson, who created the chart used for many years in psychoacoustic experiments all over the world. Recently, this diagram has been refined, and the new measures have been adopted as a standard by the International Organization for Standardization as ISO code 226:2003 (see fig. 1.15).

1000 Hz was chosen as the reference frequency for the phon, because at this frequency, a measurement in phon and one in dB often coincide. (100 dB corresponds to the feeling of 100 phon, 80 dB of 80 phon, etc.) For example, if we examine the 60 phon curve, 60 dB of pressure are necessary at 1000 Hz to produce a certain sensation, but as the pitch drops in frequency, more and more dB are required to maintain the same sensation in the listener.

(...)

other sections in this chapter:

ACTIVITIES
- Interactive examples

TESTING
- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS
- Fundamental concepts
- Glossary

1P
SOUND SYNTHESIS WITH PURE DATA

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Basic Computer Knowledge (operating a computer, managing files and folders, audio input/output, etc.)
- Basic Knowledge of Music Theory (semitones, octaves, rhythm, etc.)
- Contents of the Theory part of Chapter 1 (It is best to study one chapter at a time, starting with the theory and the progressing to the corresponding chapter on practical Pd techniques)

LEARNING OBJECTIVES

Skills

- To be able to use all of the basic functions of Pd
- To know how to synthesize sounds – both sequentially and simultaneously – using sine wave oscillators, as well as square wave, triangle wave, and sawtooth wave oscillators
- To be able to control the amplitude, frequency, and stereo spatialization of a sound continuously (using linear and exponential envelopes for glissandi, amplitude envelopes, and the placement of sound in a stereo image)
- To know how to generate random sequences of synthesized sounds
- To be able to work with sampled sounds at a rudimentary level

Competence

- To be able to successfully create your first sound etude based on the techniques you have acquired in this chapter, and save your work as an audio file.

CONTENTS

- Sound synthesis and signal processing
- The timbre, pitch and volume of sound
- Glissandi and amplitude envelopes
- Relationships between frequency, pitch, and midi
- Introduction to working with sampled sound
- Introduction to panning
- Some basics of the Pd environment

ACTIVITIES

- Substituting parts of algorithms, correcting algorithms, completing algorithms, analyzing algorithms, constructing new algorithms

TESTING

- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of principal commands, list of native Pd objects, List of Virtual Sound library Objects, list of messages for specific objects, glossary

1.1 INSTALLATION AND FIRST STEPS WITH PD

Installation and System Configuration

Before continuing to read this chapter, you will need to perform the following operations:

1. Download and install Pure Data (Pd);
2. Download and install the *Virtual Sound* Library [for Pd];
3. Download the supporting materials created to accompany this text;
4. Configure your audio and MIDI system correctly.

The Pure Data[1] software distribution can be downloaded via the website http://msp.ucsd.edu/software.html, which includes versions of the software for both Windows and Mac OSX (including 64-bit) operating systems. To install the software, you simply need to execute the downloaded file and follow the instructions that subsequently appear. Installation is slightly more complicated on Linux operating systems, but Pd can generally be installed using the *synaptic* application without any particular problem. For a more in depth discussion about installing Pd on different operating systems, please refer to the document EMASD_Pd_Installation.pdf which can be found on the website at the support page for this text.

This text is based on Pd-0.51-4 (the latest stable version at the time of this book's publication), but the examples should also work with older versions back to and including 0.45.

While you are on the above mentioned website, be sure to download the *Virtual Sound Library*, which is absolutely essential for the proper functioning of the examples used in this text. To install it correctly, follow these steps:

1. Download the file VirtualSoundLibraryPd.zip to your computer and uncompress it. You should obtain a folder called Virtual_Sound_Library_Pd;
2. Launch Pd and go to the Path section of the Pd/Preferences menu (Mac) or File/Preferences menu (other operating systems). Click on New, and a window will open allowing you to locate and select the Virtual_Sound_Library_Pd folder. Once you have found and selected it, click on Ok.
3. Relaunch Pd.

[1] If you search the internet, you will certainly come across several different versions of Pure Data, including *Pd-extended*. Although this is one of the more widespread versions and includes a very rich library of extensions, it was built upon an outdated version of Pure Data, unfortunately, and consequently is no longer being maintained. This text is therefore based on the *vanilla* release of Pure Data – the original version of the of the program which continues to be maintained by its creator, Miller Puckette. The examples in this book will NOT all work with *Pd-extended*.

If everything went well, Pd will now automatically locate the objects in the Virtual Sound Library each time one of them is created by the user[2]. For more information, refer to the related documents also found on the website.

The supporting materials for this volume have been gathered together and compressed in the file **EMASD_PD_MATERIAL_vol_1.zip**. You can download and uncompress it to any location on your computer of your choosing. From there, you can open the files designed to accompany each of the chapters, as well as those pertaining to the activities and reverse engineering exercises.

Audio Setup

To check that your audio is working correctly, perform the following steps:

1. Use the Media/Audio Settings menu to make sure that the Input Device 1 and Output Device 1 fields contain the name of your audio interface. If not, then simply select your audio interface using the drop down menus (for both input and output devices).
2. Use the Media/Test Audio and MIDI menu to open its window (shown in figure 1.1). On the upper left hand side of this window, in the section labeled 'TEST TONES', click on the selections labeled '80' and 'noise'. Double check that your computer's audio is not muted!

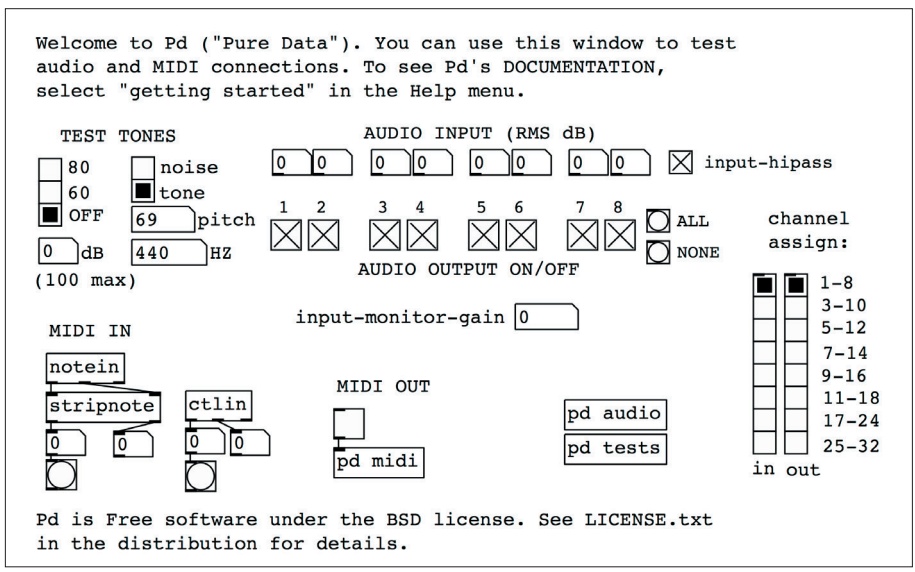


fig. 1.1: The built-in Pd window used to verify audio configuration/operation.

Now we can really begin!

[2] The names of all the objects in the Virtual Sound Library begin with the prefix *vs.* followed by the name of the object.

First Steps with Pd

Launch the Pd program and select the New option from the File menu (or type <Command-n> on the Mac or <Control-n> on Windows or Linux[3]: An empty **Patcher Window**, also known as a **Canvas**, will appear. This window is where we will begin to assemble our first algorithm, which appears as a collection of "boxes" in the window. These boxes, which contain strings of text and numbers, are generically referred to as **text boxes**. A collection of text boxes that are connected together is called a **patch** (a reference to the old world of analog modular synthesizers that were programmed using physical cable connections called **patch cords**).

After you open a new *patcher window*, type the key combination <*Cmd/Ctl-1*>, or alternately go to the *Put* menu and select the *Object* option. This will create a rectangle with dashed sides (as shown in figure 1.2), containing a blinking cursor prompting you to type a string of alphanumeric characters.

fig. 1.2: A generic object box

This rectangle is called an **object box**. Now, go ahead and type the character string 'osc~ 440'[4] into it and then move the cursor and click outside the rectangle with the mouse. You have just created the [osc~] object (see figure 1.3) – a sine wave oscillator.

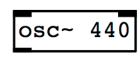


fig. 1.3: The [osc~] object

In Pd a character string without spaces is known as an **atom**. An atom can be composed of a number, a literal string, or any kind of symbol. A Pd object is made up of an initial atom that defines the object's name, followed by one or more atoms that are optionally supplied as **creation arguments**. In performing the above actions, we have just created an object whose class is `[osc~]` with a creation argument of 440.

---

[3] On Mac OSX, hold the Command key (⌘) down and press 'n', or on Windows and Linux, press 'n' while holding the Control key down. This will be abbreviated as <*Cmd/Ctl-n*> from this point on. Whenever you see this notation, use <*Command*> if you are using a Mac or use <*Control*> if you are using Windows or Linux.

[4] Note the character "~", which is called a tilde, that follows the string 'osc'. The tilde character is almost always the last character in the name of objects used to process streams of digital audio; it is an important naming convention in Pd. It will be important that you know where to find the tilde on your keyboard, especially if you have a non-English one!.

The little rectangles located on the upper and lower parts of the object are inlets and an outlet, respectively, and we will soon see how they are used. If the object you just created does not have them (i.e., if it does not resemble the one shown in the figure), then there is some kind of problem, and you should read the FAQ at the end of this section.

Now let's create another object, `[vs.gain~]`, which is part of the Virtual Sound Library, and looks somewhat like a fader on a mixing board (see figure 1.4).

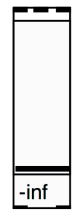


fig. 1.4: The `[vs.gain~]` object

Just as you did with `[osc~]`, type *<Cmd/Ctl-1>* to create a generic object box, then type the string '`vs.gain~`' inside it, and click outside the box on an empty part of the patcher window[5]. The graphical object which you have just created falls into the GUI (Graphical User Interface) category, as it allows user interaction via the mouse. Move this object so it is underneath the `[osc~]` and connect the outlet of `[osc~]` to the left inlet of `[vs.gain~]` using a **patch cord** connection.

To do this, you will need to perform a *click*, *drag and drop* action with the mouse. First, move the mouse pointer (which looks like a hand with a pointing finger) over the outlet of the `[osc~]` object (located at the left side of its bottom edge), at which point the pointer will turn into a circle. Click and hold down the mouse button, drag the pointer to the left inlet of `[vs.gain~]` (at which point it will turn into a circle again), and release the mouse button. If you did this correctly, the result should look similar to the two connected objects shown in figure 1.5.

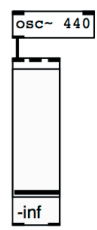


fig. 1.5: Connecting objects

[5] If the object does not appear as shown, double-check to make sure you have typed its name correctly. If you are still experiencing problems, the *Virtual Sound Library* may not be properly installed, in which case you should go back to the beginning of this chapter and follow the installation instructions carefully.

The connection from the oscillator to **[vs.gain~]** was made in order to be able to adjust the oscillator's volume. We now need to send the resulting audio signal to an audio output.
We do this by creating a **[dac~]** object (shown in figure 1.6), positioning it underneath **[vs.gain~]**, and connecting the left outlet of **[vs.gain~]** to both the left and right inlets of **[dac~]**, by dragging patch cords, one at a time, from outlet to inlet, as we did earlier.

fig. 1.6: The [**dac~**] object

The result should look similar to figure 1.7.

fig. 1.7: Our first patch

Watch out! Make sure you used the *left* outlet of **[vs.gain~]** for *both* connections. If you notice that one of the two patch cords you just created extends from the right outlet of **[vs.gain~]**, you should delete it! Do this by moving the mouse pointer over the patch cord (the pointer will turn into an X), click on the patch cord to select it (it will turn blue), then type the *delete* key on the computer keyboard (the same one you use to delete text when you are typing) to delete the connection. You can now reconnect the objects correctly.

Now would be a good time to save your new patch to disk, keeping this warning in mind: DON'T EVER save your patches to a file that shares a name with a pre-existing Pd object! For example, don't call this patch "osc~". Doing this would be a recipe for confusing Pd, and for causing unexpected results the first time that you tried to reload your patch. Given that it is impossible to remember the names of all the Pd objects, a good technique for avoiding the use of object names, and therefore averting the danger of a confusing name, is to give your files a name composed of multiple words with spaces between them: "test oscillator", for example, or "osc~ object test", or any other combination. No Pd object possesses a name composed of more than one word. Don't forget this advice! A large percentage of the problems encountered by Pd beginners relates to saving files that share a name with some existing object.

Good! We've finished implementing our first patch, and we are ready to make it run. It lacks one more touch, however: up till now we've been in **Edit Mode**, in which we assemble patches by inserting, moving and connecting objects together, and now we need to make the transition into **Run Mode**, where we will be able to hear and test our patch. To do this, press <*Cmd/Ctl-e*>, or use the *Edit* menu to uncheck the *Edit Mode* menu option. Once the patch is in **Run Mode**, type <*Cmd/Ctl-/*> (or use the *Media* menu to select *DSP On*) to start the audio engine[6], and then slowly raise the level of the slider in the `[vs.gain~]` object.[7] You should hear a sound, pitched at A above middle C. To disable sound output, you can type <*Cmd/Ctl-.*> (or use the equivalent menu selection: *Media/DSP Off*). If you are not hearing any sound at all, you should try consulting the FAQ at the end of this section.

Now that we've built a patch and seen and heard it work, let's revisit the patch we made by analyzing the algorithm behind it: The `[osc~]` object is an oscillator (a sound generator that produces a periodic waveform, in this case, a sine wave), and the number 440 that we typed into its interior indicates the frequency that we want it to produce; in this case, we specified that the sine wave should repeat itself 440 times per second.[8] In more formal terms, `[osc~]` is the name of the object, and 440 is an **argument** – a value used by the object to specify its operation. In this case, the argument 440 caused `[osc~]` to produce a tone at 440 Hz.

The signal outlet of the `[osc~]` object is connected to the inlet of the `[vs.gain~]` object, which causes the signal generated by `[osc~]` to be passed on to the `[vs.gain~]` object, which, as we have seen, modifies the signal's volume when its volume fader is moved. The modified signal is then passed on to the `[dac~]` object, which routes it to the computer's sound driver. The sound driver sends the signal to the audio interface, which performs a digital-to-analog conversion on the signal to transform the numeric (i.e. digital) representation of the sound into an audio wave that can be heard via headphones or speakers. (This conversion, by the way, is where `[dac~]` gets its name: DAC is an acronym for Digital-to-Analog Converter.)
Let's now broaden the function of this patch. Let's make it possible to actually see what is happening in addition to hearing the sound. Save the current patch (which you will need again in the next section) into an appropriate folder, for example "My Patches", and close the Patcher Window. If you haven't already downloaded and unpacked the "EMASD_PD_MATERIAL_vol_1.zip" archive that can be found on the support web page mentioned at the beginning of this chapter, do this now.

---

6 The audio engine in Pd is not active by default: it must be turned on (and off) by the user as needed. This is because Pd, more generally speaking, is an actual programming language which, in addition to sound synthesis, can perform other tasks that do not require the use of audio.

7 When a `[vs.gain~]` object is created, its slider is set to the minimum value (silence) by default.

8 All of these concepts were laid out in Theory Section 1.2.

Open the file 01_01.pd, which you will find in the "EMASD_PD_MATERIALS/VOL_1/01_patch" folder (see Figure 1.8).

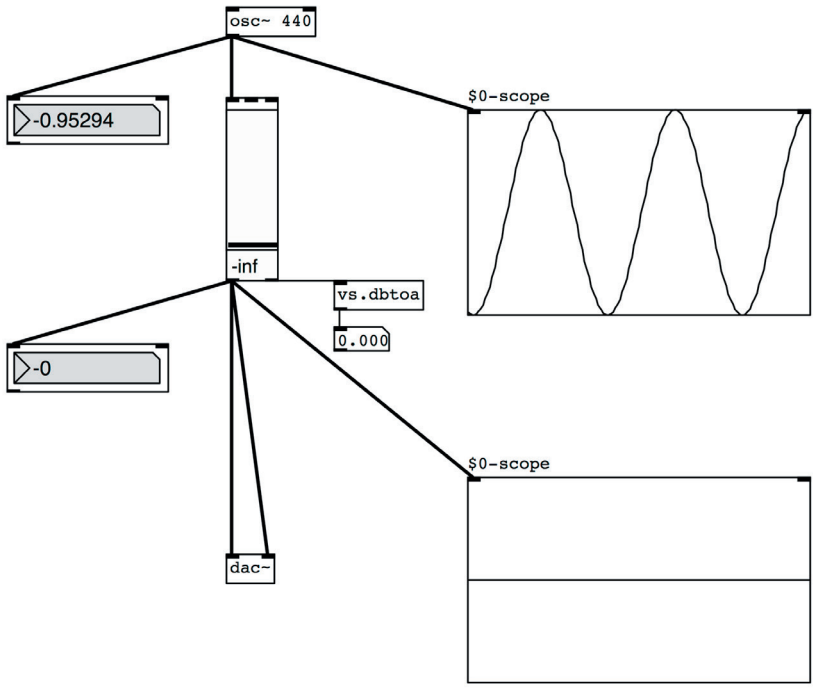


fig. 1.8: The file 01_01.pd

When this file opens, you will see that we have added new objects to the original patch. The new objects on the left, in which you can see numerical values, are called `[vs.number~]` objects, and they show, via the numerical value that they display, a snapshot of the signal that they are receiving. The larger rectangular objects on the right are called `[vs.scope~]` objects, which act as oscilloscopes allowing an incoming signal to be viewed as a graphic waveform. The `[vs.dbtoa]` object and the object that is connected to its outlet (which is called a *number box*[9]) are used to view exactly how much amplification or attenuation is being applied to the signal by `[vs.gain~]`.

Once again, start the patch by typing <*Cmd*/*Ctl*-/>, and observe the changing numbers displayed in the `[vs.number~]` object on the upper left side of the patch. These numbers are being produced by the `[osc~]` object and, if we observe them for a bit, we can see that their values, both positive and negative, fall between 1 and -1.

[9] A number box can be created by typing <*Cmd*/*Ctl*-3>, or via the *Put*/*Number* menu option.

On the upper right side of the patch we can see the `[vs.scope~]` object displaying the same numbers in graphical form; the upper half of its panel corresponds to positive numbers, and the lower half to negative numbers. In the `[vs.scope~]` panel, hundreds of different values are shown, visualized as a sequence of points, rather than the single number shown by the `[vs.number~]` object. The points fall very close to each other on the screen, and so they appear as a solid curve. The values that they represent – i.e., the numbers that make up the digital signal itself – are called *samples* in the terminology of digital music. And the curved line made from these sample values, as they undulate high and low across the oscilloscope panel, is precisely the sinusoidal wave produced by the `[osc~]` object.

If the `[vs.gain~]` fader is lowered to its minimum value, you will notice that the lower set of `[vs.number~]` and `[vs.scope~]` objects display the number 0 and a flat line (which is, of course, a sequence of zeros), respectively, since the volume fader is at its lowest setting, resulting in a volume of 0. If the `[vs.gain~]` fader is moved upwards, you should see the lower `[vs.number~]` begin to display values that start out very small and gradually grow larger as the volume rises, and at the same time, the flat line of the lower `[vs.scope~]` should begin its undulation and assume the same look as the other `[vs.scope~]`. We can infer from this that `[vs.gain~]` is controlling the amplitude of the signal – the more we raise the fader, the greater the amplitude of the oscillations becomes. If we go too far, and raise the value of the `[vs.gain~]` fader to be close to its maximum setting, we see `[vs.number~]` begin to exceed the amplitude limits of 1 and -1, and the waveform on the oscilloscope becomes clipped at the top and bottom. More important than these visual clues, you should be able to actually *hear* the sound change, as it becomes distorted.

We can now draw some conclusions from what we've seen:

1. The `[osc~]` object produces a sequence of numeric values that follow the shape of a sine wave[10];
2. The numerical limits for samples in this sine wave are 1 and -1. The actual sequence that these values follows can be seen on the upper `[vs.scope~]`, which shows the waveform at its maximum amplitude, above which the quality of the sound would be distorted;
3. The `[vs.gain~]` object modifies the amplitude of the sine wave, causing the sample values at its outlet to be different than the corresponding sample values received on its inlet. How does it do this? By multiplying the values that it receives by a quantity that depends upon the position of the fader. When the fader is in its lowest position, the signal is multiplied by 0, and the result is a stream of zeros (because any number multiplied by 0 is 0). One can see that as we raise the fader, the multiplication factor rises.

[10] In reality, as we will see later on in this book, `[osc~]` actually generates values that define the shape of a *cosine* wave.

If, for example, we raise it from -inf to -6, the corresponding multiplication factor will be 0.5 (as can be seen in the number box connected to the right outlet of `[vs.gain~]`). This means the amplitudes of the samples that enter the `[vs.gain~]` object are diminished (because multiplying a number by 0.5 is the same as dividing by 2). As we raise the fader to 0 (by moving it upwards until the numerical value below the fader reads 0)[11], the sample values entering the object are identical to those leaving, since they are multiplied by 1.

Finally, if we raise the fader all of the way (the maximum allowed by this object is a value of +6), the most extreme of the sample values will exceed the limits of 1 and -1, although these samples will then be brought back into line during the digital-to-analog conversion. When that happens, however, the waveform will no longer be a sine wave. Instead, it will be clipped (as we can see in the lower oscilloscope). Sample values that fall outside of the -1 to 1 range are actually simply reassigned the maximum possible amplitude during conversion, and the distorted sound that we hear reflects the resulting truncated waveform.

We have continued the exploration of the concepts of frequency, amplitude, and waveform, which we began in Section 1.2 of the theory chapter. Let's recap some practical aspects of these basic concepts:

- *Amplitude* is the physical parameter that corresponds to the intensity of a sound; it is the parameter that controls the perception of whether a given sonic event is forte or piano. In Pd, the absolute value of amplitude (that is, the value of the parameter, independent of its sign) always lies between 0 and a maximum of 1.
- *Frequency* is the physical parameter that relates to pitch; it is the parameter that controls the perception of whether a given sonic event is high or low in pitch. The values are expressed in hertz (Hz), and we need to keep in mind that sounds audible to humans fall in the range between 20 and around 20,000 Hz
- *Waveform* is a fundamental element of timbre, which we define as the overall quality of a sound. *Timbre* enables us to perceive, for example, the difference between the middle C played on a guitar and the same note played on a saxophone. We have seen and heard the timbre of the sine wave produced by [osc~].

FAQ (Frequently Asked Questions)

In this section, we will try to give answers to some of the more common problems that are encountered by new users of Pd. You should probably read these carefully even if you have not experienced a problem, since they contain information that you will use in the following sections of the book.

[11] For reasons that will be clear later on, `[vs.gain~]` uses the deciBel scale (see section 1.2 in the previous Theory chapter), therefore a value of 0 does not mean an amplitude of zero (silence), but rather corresponds to an amplitude of 1, the peak amplitude value.

Question: I created an object called "osc~440", as instructed in this section, but the object has no inlets or outlets. What went wrong?

Answer: Be sure that you typed a space between "osc~" and "440", because the first is the name of the object, while the second is an argument, which in this case represents the frequency of the sound. If the two words are run together, Pd will search for a non-existent object named "osc~440", and when nothing is found, Pd will not be able to create an object box with the correct number of inlets and outlets.

Q: Very good. Why, then, didn't Pd give me an error message?
A: There is an error message, which can be found in the **Pd Window**, which is a window that the program uses to communicate with you. If you cannot see this window, press <*Cmd/Ctl-r*> to bring it up (or from the menu Window/Pd Window). In the window, you will probably find a message such as:
"
osc~440
… couldn't create"

Q: I inserted a space between "osc~" and "440", but the object has no inlets or outlets just the same!

A: There is a less obvious error that often turns up for some new users when using objects that have a tilde ('~') at the end of their name. If you have a keyboard on which there is no tilde key, and so you need to press a combination of keys in order to produce the character (for example, <*Alt-5*> on some regional localized Mac keyboard layouts), you may have continued pressing one of the modifier keys when typing the space (for example, <*Alt-Space*> on the Mac). The resulting combination is not recognized by Pd, as is not able to separate the object name from its argument. Delete the space and re-insert it, avoiding pressing modifier keys at the same time.

Q: There is no sound.

A: Have you turned on the audio by typing <*Cmd/Ctl-r*>? Have you raised the volume fader above zero? Are you sure that sound on your computer is not muted, or that you are able to produce sound by using a program other than Pd? Have you checked the *Audio Settings* window (which you can find in the *Media* menu), to see that the correct sound driver is selected?

A Compact "Survival Manual" for Pd

In this section, we will give some essential information for easily navigating the Pd environment.

BASIC KEYBOARD COMMANDS
Let's start by reviewing the keyboard commands we have learned so far:

<*Cmd/Ctl-n*> will create a new *Patcher Window*, which is the workspace for implementing *patches*.

<*Cmd/Ctl-e*> is used to toggle between *Edit Mode* and *Run Mode* in the *Patcher Window*. In *Edit Mode*, we assemble *patches* by creating objects and other types of text boxes, and connecting them together. In *Run Mode* we can activate patches and interact with their graphical user interface objects (such as number boxes or the `[vs.gain~]` object).

<*Cmd/Ctl-r*> will call up the *Pd Window* if it is not already visible. The *Pd Window* is a window used by the Pd software to communicate with its user, using brief text messages. We will learn more about this window shortly.
<*Cmd/Ctl-1*> will create a generic *object box* which can be placed in the patch. An object name, often followed by one or more *creation arguments*, can be typed into this box to create a specific Pd object. A generic object box can also be created using the menu selection *Put/Object*.

<*Cmd/Ctl-3*> will create a number box that can be placed in the patch. This can also be done using the menu selection *Put/Number*.

SELECTION, DELETION, AND COPYING
To delete a patch cord or an object, you need to ensure that you are in *edit mode*,[12] select the patch cord or object with the mouse, and then press the <*Delete*> key (which may also sometimes be referred to as the <*Backspace*> key). Several objects can be selected at the same time by clicking on a blank spot in the *Patcher Window* and dragging a rectangular selection that encompasses the objects to be selected. Selected objects are highlighted in blue. At this point, if we move one of the selected objects, all of them will move. Likewise, if we press the <*Delete*> key, all of the selected objects (as well as connections between them) will be removed from the patch.

Things work a little differently if we only want to delete patch cords. To delete a single patch cord, you simply need to select it by clicking on it with the mouse and typing the <*Backspace*> key. Deleting a group of connections together, however, is impossible because Pd does not allow you to select multiple patch cords. Therefore, you will always need to select and delete patch cords one at a time.

[12] When you move the mouse pointer over a *patcher window* in *Edit Mode*, it should look like a hand with a pointing index finger.

To duplicate an object or group of objects, you need to select it and type the key combination <*Cmd/Ctl-d*>. The duplicate objects will appear selected, and positioned very close to the originals, as shown in figure 1.9.

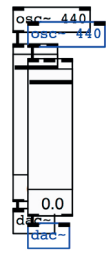


fig. 1.9: Duplicating objects

The duplicated objects (highlighted in blue) can then be moved by using the arrow keys on the computer keyboard. By holding down the <*Shift*> key[13] while using the arrow keys, the duplicated objects can be moved in larger steps. Naturally, the duplicated objects may also be moved by clicking and dragging them with the mouse, however it is very easy to accidentally deselect them this way, which is rather inconvenient, given their close initial placement to the originals.

To copy an object or group of objects from one *Patcher Window* to another, you simply need to make a selection and use the key commands <*Cmd/Ctl-c*> to copy and <*Cmd/Ctl-v*> to paste, as with most other software.

If you make a mistake (for example, you delete one object rather than another), you can undo the action by selecting the *Undo* command from the *Edit* menu (or by using the standard key combination <*Cmd/Ctl-z*>). If, after undoing one or more actions, you decide that there wasn't actually an error (for example, if you realize you wanted to delete the object after all), you can restore the action by using the Redo command, which is also found in the *Edit* menu (or by using <*Cmd/Ctl-Shift-Z*>). Caution! Pd only stores *one* action at a time – the last action which was carried out. Therefore, you can only use *Undo* or *Redo* to cancel or reinstate your previous patch editing action.

AUTOPATCH MODE

Pd provides a convenient way of creating objects that are already connected to each other, called *autopatch mode*. Let's take a look at how this feature can be used. Open a new *patcher window* (<*Cmd/Ctl-n*>) and create an **`[osc~ 440]`** object inside it using <*Cmd/Ctl-1*>. Now, select the object and type <*Cmd/Ctl-1*> once again: a new generic object box is created pre-connected to the **`[osc~]`** object with a patch cord (as shown in figure 1.10).

[13] <*Shift*> is the modifier key used to type capital letters.

Fig. 1.10: Using *autopatch* mode to create pre-connected objects

Now type 'dac~' into this object box and click in an empty part of the *patcher window* to create a `[dac~]` object. You have just created a second object connected to the first, without needing to click, drag and release a patch cord between them. Note that this technique can only be used to create connections between the *leftmost* inlets and outlets of objects.

DOCUMENTATION AND HELP

This book is self-contained: within it you will find all that you need for understanding and using the patches with which we illustrate synthesis and signal processing techniques using Pd. But the Pd software also comes equipped with an array of built-in help materials – practically every object in Pd has its own dedicated *Help* patch. To open up an object's *Help* patch, right-click on the object (or *Control-click* on Mac) while the patch is in *Edit Mode* and select *Help* from the contextual drop-down menu which appears. (In *Run Mode*, however, Windows and Linux users will need to hold down the CTRL or Control key and left-click with the mouse to obtain the contextual drop-down menu.) *Help* patches are fully functional Pd *patches* that generally summarize the principal characteristics of an object, and provide simple examples of its use (see figure 1.11).

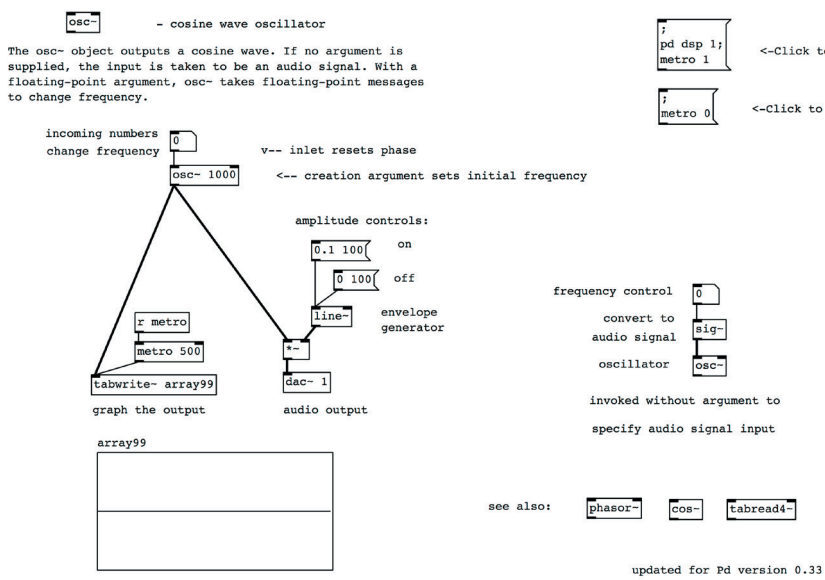


fig. 1.11: The *Help* patch for the `[osc~]` object

Additionally, you can also select the *Browser*... option in Pd *Help* menu to access a simple file *Browser* that lets you navigate through the various folders that contain help patches and files that make up the official Pd Documentation written by Miller Puckette, the creator of Pure Data. The *Help* menu additionally contains a link to the online HTML Documentation for the software.

EXERCISE

Create a new *patcher window* and attempt to reproduce the patch contained in the file **01_01.pd**. Pay careful attention that you don't confuse the `[vs.number~]` object with the *number box*! The former is created by typing the string 'vs.number~' into a generic *object box*, whereas the latter is created using the key command <*Cmd/Ctl-3*> (or via the menu selection *Put/Number*).

(...)

other sections in this chapter:

ACTIVITIES
- Correcting algorithms
- Analyzing algorithms
- Completing algorithms
- Substituting parts of algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of principal commands
- list of native Pd objects
- List of Virtual Sound library Objects
- List of messages for specific objects
- Glossary

Interlude A
PROGRAMMING WITH PD

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of chapter 1 (theory and practice)

OBJECTIVES

Skills

- To learn how to use the basic features of Pure Data that pertain to numbers
- To learn how to generate and Control sequences of random numbers, optionally with the use of a metronome
- To learn how to construct algorithms within subpatch and *abstraction*
- To learn how to repeat messages across multiple object outlets
- To learn how to assemble and disassemble lists
- To learn how to manage and use variable arguments
- To learn how to manage communication between objects without using virtual patch cords

CONTENTS

- Numbers in Pd
- Generating and controlling random numbers with the objects `[random]`, `[vs.drunk]` etc.
- Generating regular rhythmic events using the `[metro]` object
- Constructing subpatch and *abstraction*
- Managing lists and variable arguments
- Using the `[send]` and `[receive]` objects for wireless communication between objects

ACTIVITIES

Activities on your computer:

- Analyzing algorithms, completing algorithms, replacing parts of algorithms, correcting algorithms, and constructing new algorithms

TESTING

- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of Pd native objects – List of Virtual Sound library objects – List of messages for specific objects - Glossary

In this first "interlude" we will examine a few aspects of programming Pd more in depth, so to provide you with useful information. Because of the essential nature of this information, we encourage you not to skip over this section unless you are already truly expert in Pd. It is important that you implement all the tutorial *patches* that we propose throughout the text, as these small efforts yield the biggest results.

IA.1 BINARY OPERATORS AND ORDER OF OPERATIONS

Installation and system configuration

Like any respectable programming language, Pd can do many things with numbers. We will begin this chapter by looking at the simplest operations with numbers. Recreate the simple patch shown in figure IA.1. (Make sure there is a space between '+' and '5'!).

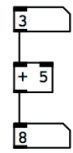


fig. IA.1: addition

The `[+]` object adds its argument (which is, in this case, 5) to whatever number it receives on its inlet. If we feed some numbers to the object via the number box above it (by selecting it, for example, in *run mode* and then writing a value), we can track the results of the operation in the lower number box.
The right inlet is used to change the argument, and if we send a number to this inlet, by using another number box, the number will be substituted for the argument of the `[+]` object in the sum that will be produced, when new numbers are sent on to the left inlet.
Try this by adding a number box on the right, as shown in fig. IA.2 .

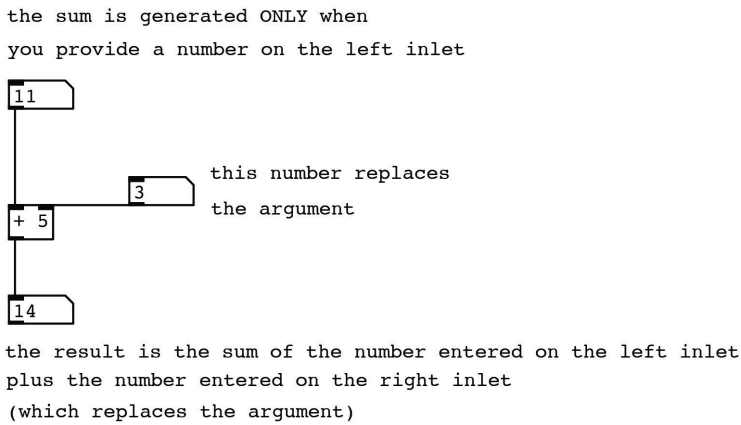


fig. IA.2: addition with a changeable argument

When you play with this patch, note that the operation is executed only when a number is sent to the left inlet, and not when a number is sent to the right inlet (which is used to replace the argument instead).

Let us clarify this point. The numbers entering the two inlets of the `[+]` object are stored into two memory cells, which we refer to as *internal variables*, and that are contained within the object itself. Every time a new number enters one of the two inlets, the corresponding internal variable is updated. (The old number is deleted and replaced with the new one.) When the number enters the left inlet, the `[+]` object sums the values of the two internal variables. This is a feature shared by most of the Pd objects, which execute their functions only when a message is received on their left inlet. Messages that are sent to other inlets are used either to modify the arguments (by updating the internal variables), or else to change the behavior of an object.

In the lexicon of Pd, the left inlet is defined as the *hot* inlet, one that causes the operation to be carried out on the object, while the other inlets are defined as *cold*, which update the internal variables without producing output.
To sum up, for an operation to be carried out properly, it is important that data are first sent to the cold inlets and then to the hot inlet.
In order for the left inlet (the *hot* one) to be the last to receive data, messages are output from right to left too. Let us look at an example using the `[swap]` object, which we have already studied at the end of the first chapter. This object swaps the order of the first two incoming numbers. When returning the values, it outputs a number through the right outlet first, and then a number through the left one (see fig. IA.3).

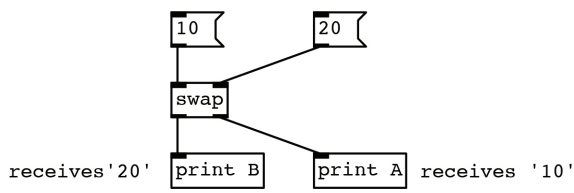


fig. IA.3: output order of an object

Once you have recreated the patch, first click on the right message box ('20') and then on the left one ('10') to test if it works. The object will first output the value 10 through the right outlet, and then the value 20 through the left outlet.

THE TRIGGER OBJECT

The purpose of this kind of behavior (i.e., the hot and cold inlets combined with the right-to-left order) is to avoid ambiguity. Just remember to follow this protocol and you will not encounter any problems. What happens when a single outlet outputs two *patch cords*? (This situation is illustrated in the example IA.4.)

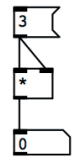


fig. IA.4: output order of a single outlet

Pd gives priority to the *patch cord* you create first. In this example (which shows you how to calculate the square of 3), the operation is carried out properly only if you connect the right cable first. In fact, in this case, when activating the message box, the message '3' is sent to the right inlet first and stored in the corresponding internal variable, and then a copy of the same message is sent to the left inlet as well. Finally, the operation is carried out. If you created the left cable first, when activating the message box, the number '3' would be sent to the left inlet first, and the calculation would be performed immediately. Therefore, the number 3 would be multiplied by 0, as the right inlet would not receive any value, and you would get an unpredictable result.

Just by looking at the patch, it is impossible to determine in what order it was built, and therefore we suggest you do not use this type of configuration. The object that allows you to specify the order in which messages are sent is `[trigger]`, which can also be used in its abbreviated form `[t]` (see fig. IA.5).

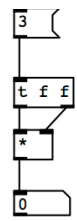


fig. IA.5: the `[trigger]` object

This object accepts an arbitrary number of *creation arguments*, each of which indicates the *type* of message to be returned. For each individual argument, an outlet is created.
When `[trigger]` receives a message on its single inlet, it outputs the same message through all its outlets, starting from the rightmost one. In the example shown, two numeric (*float*) outlets have been created. When the message '3' is received, this value is immediately put out through the right outlet first, and then through the left one. This way, the `[*]` object receives the elements of the multiplication in the correct order, causing it to carry out the operation in the desired manner.
Oftentimes, you may find it useful to convert the outlet type, since `[trigger]` can also convert the type of atoms it receives. Figure IA.6 shows a `[trigger]` object which contains two arguments. The first one, *b*, (which stands for bang), indicates that whatever number is sent to the inlet, the first outlet will always output a bang. The second argument, *f*, (which stands for *float*), indicates that the right outlet of `[trigger]` will always output the number received at its inlet.

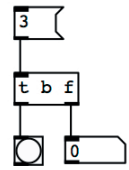


fig. IA.6: type conversion using `[trigger]`

In the example shown in the figure, the message box sends the number 3 to `[trigger]`, which in turn outputs this number through the right outlet, while the left outlet outputs a bang.

In figure IA.7 we used the conversion system to implement a simple algorithm that makes the cold inlet of a `[+]` object behave like a hot one. When we send the addend to the left inlet, the `[+]` object will carry out the operation and output the result. When we send a value using the number box on the top right instead, this will send the value to the `[trigger]` object, which in turn will output the same value through its right outlet first (also storing it into the corresponding internal variable), and then send a bang to the left inlet of the `[+]` object (which 'forces' the object to produce the result by summing the two values that were stored in the internal variables).

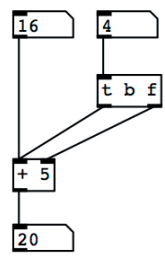


fig. IA.7: how to make a cold input behave like a hot one

To conclude, let us review the various arguments that the `[trigger]` object can accept, also introducing some new ones we do not know yet.

f: *float* number
s: *symbol*
b: bang
l: list (a message consisting of several atoms)
a: *anything* (this argument lets anything through it, without performing any conversion)

DEPTH FIRST

The last principle that regulates the order of the operations is called the *depth first*. Basically, this rule states that a message can "travel" only once, until it reaches an end point in order for it to be able to "travel" again. In figure IA.8, the initial bang is first sent through the right section of the algorithm, then it reaches the `[print A]` object, and then is sent through the middle section to

reach `[print B]`. Finally, it is sent through the left section of the algorithm until it reaches the `[print C]` object.

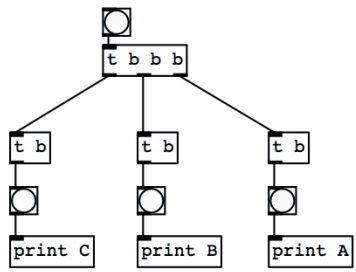


fig. IA.8: the *depth first* principle

If you click on the upper bang you will notice that the three bangs light up simultaneously. That is because it takes a very short time for the message to travel through the three sections of the patch.
However, if you look at the *pd window* you can see that the first `[print]` that receives the bang is the one labelled as 'A', then 'B', and eventually 'C'.

Back to the Pd `[+]` object, we will now see a simple musical application. Open the file **IA_01_transposition.pd** (which is shown in fig. IA.9).

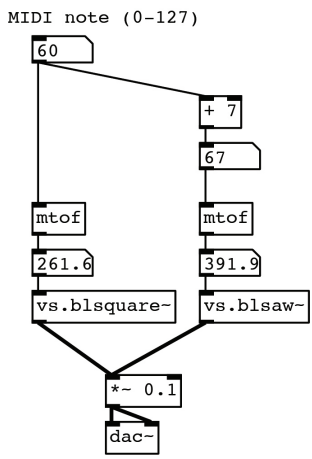


fig. IA.9: the file IA_01_transposition.pd

In this patch, every time you enter a value in the upper number box, two notes are generated. The second note is seven semitones above the first one, which is a perfect fifth. In practice, every time we enter a value in the number box, the corresponding MIDI note is sent to the left `[mtof]` object, which converts it into a frequency. In the meantime, however, the same note is sent to an addition operator which adds the value '7' to the note, thus adding seven semitones to the base note. The result of the sum is then sent on to a second `[mtof]` object.

Hence, by using the addition operator it is possible to transpose MIDI notes by any interval of your liking (meaning you can arbitrarily decide the distance between two notes). After the MIDI values of the two notes have been converted into frequency values, they are sent to two oscillators (`[vs.blsquare~]` and `[vs.blsaw~]`), which will play the fifth interval. Play with the patch shown in fig. IA.9 and try using various intervals (such as a third, four semitones, or a fourth, five semitones, and so on). Then add a new addition operator, connect it to a new `[mtof]` object, which is in turn connected to an oscillator, and make sure that for each value entered in the upper number box, a three-note major chord is generated. If, for example, you enter the value 60 in the number box, you will obtain the chord 60, 64, 67.

HOW NUMBERS ARE REPRESENTED IN PD AND OTHER OPERATIONS

Before delving deeper into binary operators, it is important to understand how numbers are represented in Pd. Unlike most of the programming languages, which distinguish integer from floating-point values, there are only *float* numbers inside Pure Data.

This can lead to some problems when you need to carry out those operations that require integer numbers. In the example shown on the left of figure IA.10, a programmer would expect a division between integer numbers (of course, the `[/]` object should carry out the division operation), which generally returns an integer result. In Pd the two input numbers are considered as *float* and the result itself is returned as a *float* value. In order to carry out an integer division (with an integer result) you need to convert the result into an integer value using the `[int]` object. To be precise, the `[int]` object removes the fractional part of a *float* value, also returning a new *float* number (see the right half of fig. IA.10).

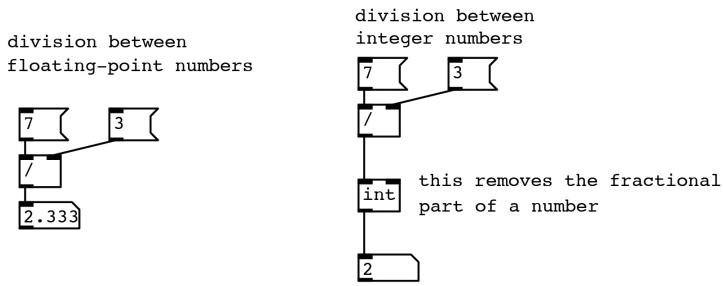


fig. IA.10: the division

The `[int]` object, as well as its counterpart used for floating-point numbers (`[float]`) – abbreviated to `[i]` and `[f]`, respectively – also perform another important function. In fact, they store the last value received on their right or left inlet. If, for example, we send a value to the right inlet of one of these objects, this same value can be put out through the outlet by sending a bang to the left inlet (see fig. IA.11).

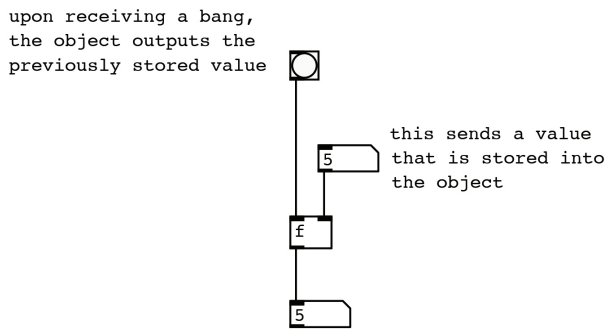


fig IA.11: the `[float]` object

The binary operators we have already seen (`[+]`, `[*]` and `[/]`), along with `[-]` and `[pow]`, are a group of arithmetical binary operators which we will discuss several times throughout the course of this book. Open a new patch*er window*, create some objects for the arithmetical operations and test their behaviors.

To conclude, let us now analyze a simple example of how to use binary operators in order to implement a system that, when working with two oscillators, uses a certain frequency for the first one, and a transposition of a perfect fifth for the other. Open the file **IA_02_fifths.pd** (see fig. IA.12).

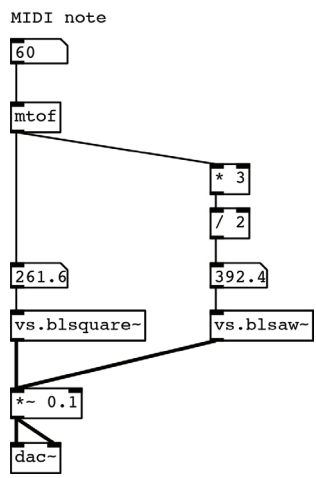


fig. IA.12: the file IA_02.fifths.pd

In this patch you can see that the MIDI note we entered in the left number box is converted into frequency using a `[mtof]` object, and then sent to a band-limited square wave oscillator (`[vs.blsquare~]`). Furthermore, this frequency is multiplied by 3/2, so to obtain a frequency which is a fifth above, and then sent to a band-limited sawtooth wave oscillator (`[vs.blsaw~]`). Hence, the output of the `[mtof]` object is split in two; a cable is connected to the left number box, while the other cable is connected to the `[*]` object using the argument 3.

If you compare figures IA.9 and IA.12, you will notice that the frequency of the highest note (the fifth) is slightly different (so the resulting timbre is different as well).
In fact, using the two `[mtof]` objects we calculated the interval of an *equal-tempered* fifth (the one which is commonly used in occidental music), which is equal to seven equal-tempered semitones. Conversely, the 3/2 ratio allows us to calculate a *perfect* fifth, which is about two *cents* (cents of an equal-tempered semitone one) 'wider' than an equal-tempered fifth.

If you are curious about which and how many binary operators are available in Pd, open the file **help-intro.pd**, which can be opened via the menu *Help/List of objects*, that shows the list of all Pd objects (see fig. IA.13). This is a good starting point to learn about Pd objects and their features.

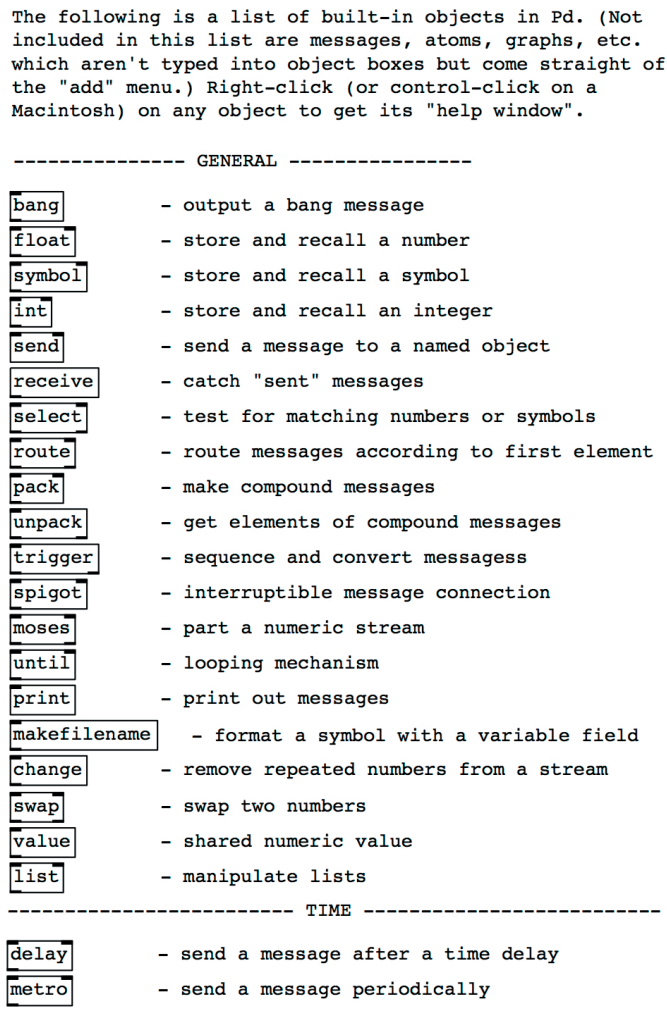


fig. IA.13: part of the file help-intro.pd

(...)

other sections in this chapter:

ACTIVITIES
- Analyzing algorithms
- Completing algorithms
- Replacing parts of algorithms
- Correcting algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Pd native objects
- List of Virtual Sound Library objects
- List of messages for specific objects
- Glossary

2T
ADDITIVE AND VECTOR SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapter 1 (Theory)

LEARNING OBJECTIVES

Knowledge

- To learn about the theory behind adding waveforms (phase, constructive interference, destructive interference)
- To learn about the theory and use of basic additive synthesis, using both fixed and variable spectra to produce both harmonic and non-harmonic sounds
- To learn about the relationship between phase and beats
- To learn how to use wavetables, and how interpolation is implemented
- To learn some theory to support basic vector synthesis

Skills

- To be able to differentiate between harmonic and non-harmonic sounds
- To be able to recognize beats upon hearing them
- To identify the different segments of a sound envelope, and to describe their characteristics

CONTENTS

- Additive synthesis using both fixed and variable spectra
- Harmonic and non-harmonic sounds
- Phase and beats
- Interpolation
- Vector synthesis

ACTIVITIES

- Interactive examples

TESTING

- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS

- Fundamental concepts
- Glossary
- Discography

2.1 FIXED SPECTRUM ADDITIVE SYNTHESIS

A sound produced by an acoustic instrument, any sound at all, is a set of complex oscillations, all produced simultaneously by the instrument in question. Each oscillation contributes a piece of the overall timbre of the sound, and their sum wholly determines the resulting waveform. However, this summed set of oscillations, this complex waveform, can also be described as a group of more elementary vibrations: sine waves.

Sine waves are the basic building blocks with which it is possible to construct all other waveforms. When used in this way, we call the sine waves frequency components, and each frequency component in the composite wave has its own frequency, amplitude, and phase. The set of frequencies, amplitudes, and phases that completely define a given sound is called its **sound spectrum**. Any sound, natural or synthesized, can be decomposed into a group of frequency components. Synthesized waveforms such as we described in Section 1.2 are no exception; each has its own unique sound spectrum, and can be built up from a mixture of sine waves. (Sine waves themselves are self-describing – they contain only themselves as components!).

SPECTRUM AND WAVEFORM

Spectrum and waveform are two different ways to describe a single sound. Waveform is the graphical representation of amplitude as a function of time.[1] In figure 2.1, we consider the waveform of a complex sound in which the x-axis is time and the y-axis amplitude. We note that the waveform of this sound is bipolar, meaning that the values representing its amplitude oscillate above and below zero. A waveform graph is portrayed in the **time domain**, a representation in which instantaneous amplitudes are recorded, instant by instant, as they trace out the shape of the complex sound.

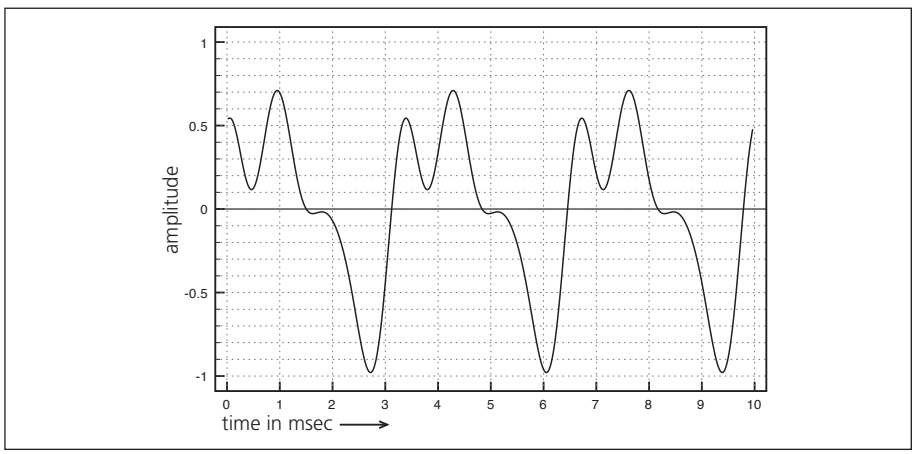


Fig. 2.1 The waveform of a complex sound

[1] In the case of periodic sounds, the waveform can be fully represented by a single cycle.

In figure 2.2, we see the same complex sound broken into frequency components. Four distinct sine waves, when their frequencies and amplitudes are summed, constitute the complex sound shown in the preceding figure.

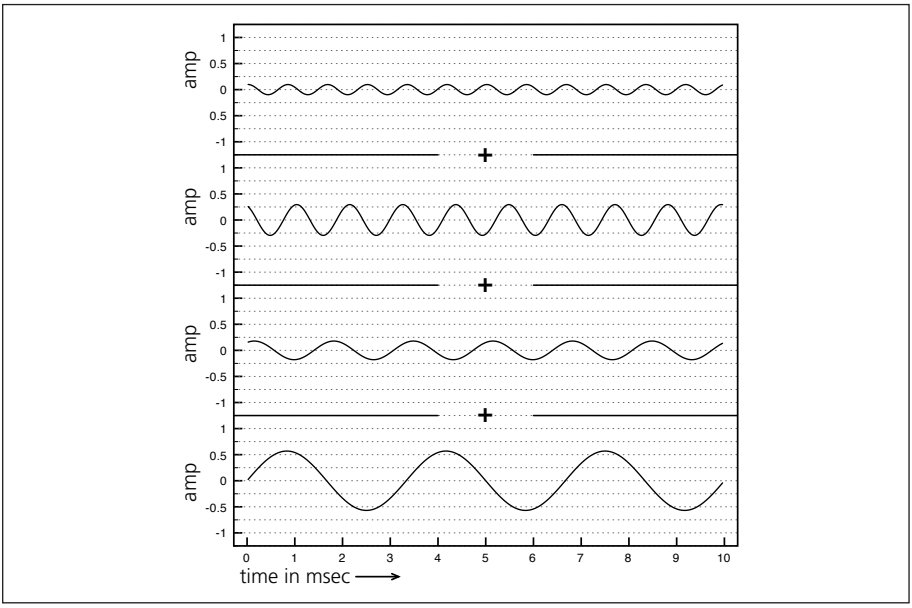


Fig.2.2 Decomposition of a complex sound into sinusoidal components

A clearer way to show a "snapshot" of a collection of frequencies and amplitudes such as this might be to use a graph in which the amplitude of the components is shown as a function of frequency, an approach known as **frequency domain** representation. Using this approach, the x-axis represents frequency values, while the y-axis represents amplitude. Figure 2.2b shows our example in this format: a graph displaying peak amplitudes for each component present in the signal.

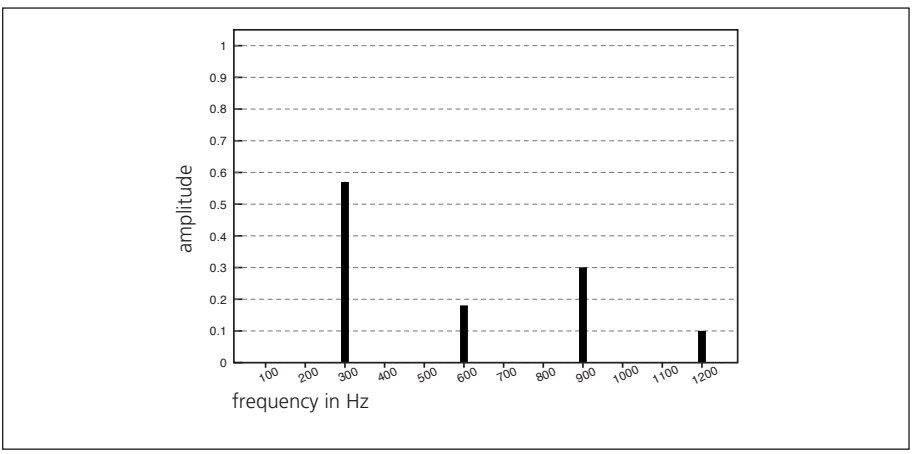


Fig. 2.2b A sound spectrum

In order to see the evolution of components over time, we can use a graph called a **sonogram** (which is also sometimes called a spectrogram), in which frequencies are shown on the y-axis and time is shown on the x-axis (as demonstrated in figure 2.2c). The lines corresponding to frequency components become darker or lighter as their amplitude changes in intensity. In this particular example, there are only four lines, since it is a sound with a simple fixed spectrum.

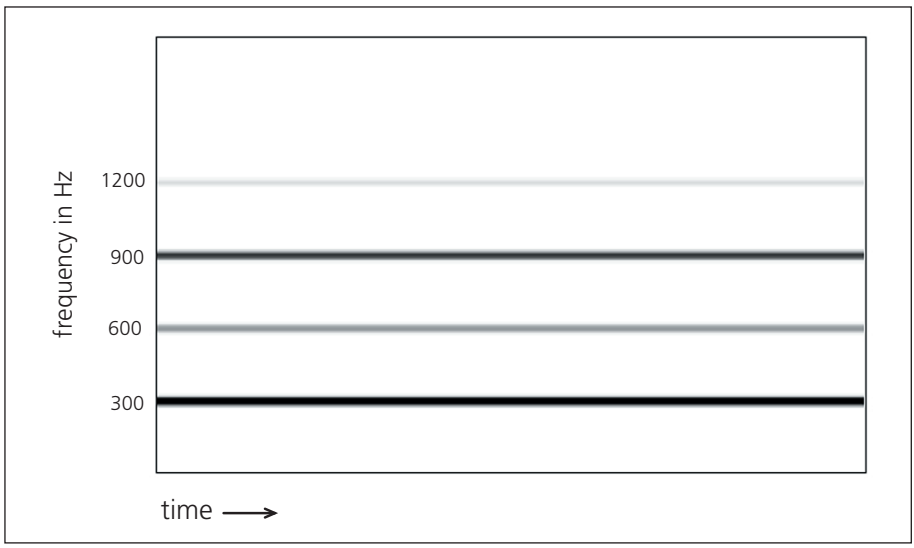


Fig. 2.2c A sonogram (also called a spectrogram)

Now we will consider a process in which, instead of decomposing a complex sound into sine waves, we aim to do the opposite: to fashion a complex sound out of a set of sine waves.

This technique, which should in theory enable us to create any waveform at all by building up a sum of sine waves, is called **additive synthesis**, and is shown in diagrammatic form in figure 2.3.

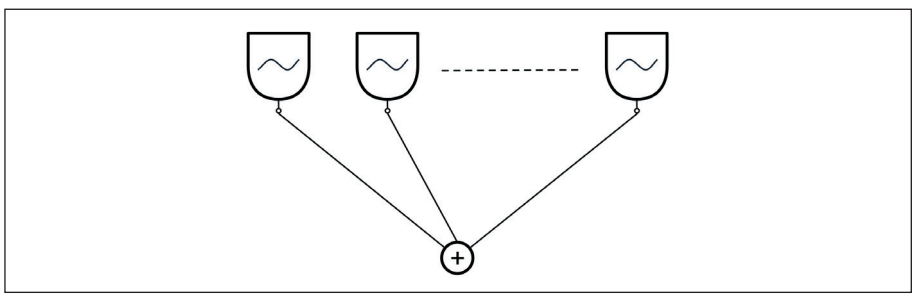


Fig. 2.3 A sum of signals output by sine wave oscillators

In figure 2.4, two waves, A and B, and their sum, C, are shown in the time domain.

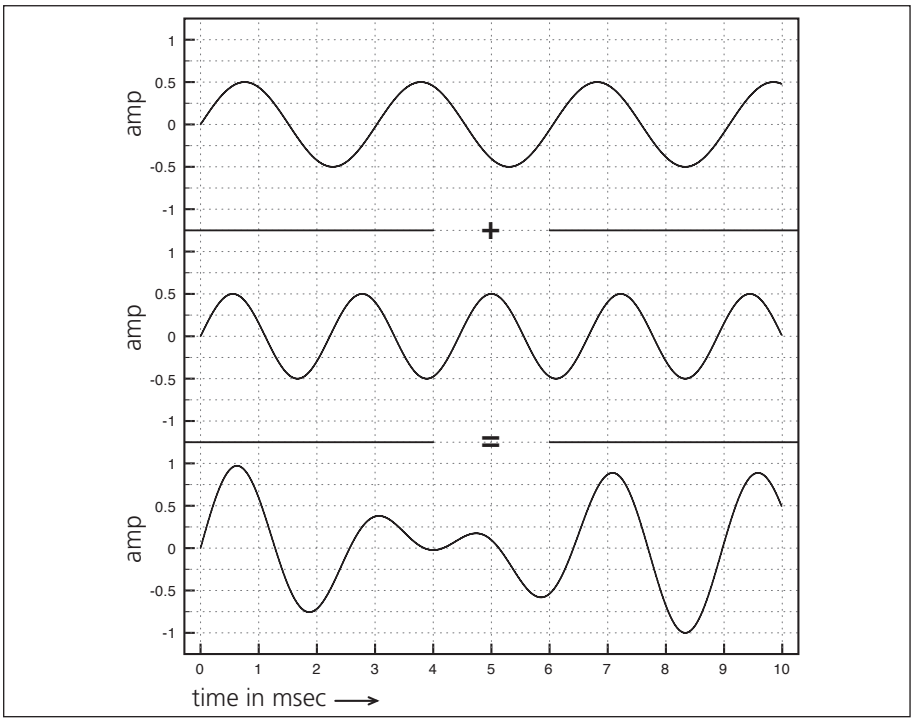


Fig.2.4 A graphical representation of a sum of sine waves

As you can easily verify by inspection, instantaneous amplitudes for wave C are obtained by summing the instantaneous amplitudes of the individual waves A and B. These amplitude values are summed point-by-point, taking their sign, positive or negative, into consideration. Whenever the amplitudes of A and B are both positive or both negative, the absolute value of the amplitude of C will be larger than that of either of the component, resulting in **constructive interference**, such as displayed by the following values:

A = -0.3
B = -0.2
C = -0.5

Whenever the amplitudes of A and B differ in their signs, one being positive and the other negative, the absolute value of their sum C will be less than either one or both components, resulting in **destructive interference**, as shown in the following example:

A = 0.3
B = -0.1
C = 0.2

"The largest part, indeed nearly the entirety, of sounds that we hear in the real world are not pure sounds, but rather, **complex sounds**; sounds that can be

resolved into bigger or smaller quantities of pure sound, which are then said to be the components of the complex sound. To better understand this phenomenon, we can establish an analogy with optics. It is noted that some colors are pure, which is to say that they cannot be further decomposed into other colors (red, orange, yellow, and down the spectrum to violet). Corresponding to each of these pure colors is a certain wavelength of light. If only one of the pure colors is present, a prism, which decomposes white light into the seven colors of the spectrum, will show only the single color component. The same thing happens with sound. A certain perceived pitch corresponds to a certain **wavelength**[2] of sound. If no other frequency is present at the same moment, the sound will be pure. A pure sound, as we know, has a sine waveform."
(Bianchini, R., Cipriani, A., 2000, pp. 69-70)

The components of a complex sound sometimes have frequencies that are integer multiples of the lowest component frequency in the sound. In this case the lowest component frequency is called the **fundamental**, and the other components are called **harmonics**. (A fundamental of 100 Hz, for example, might have harmonics at 200 Hz, 300 Hz, 400 Hz, etc.) The specific component that has a frequency that is twice that of its fundamental is called the second harmonic, the component that has a frequency that is three times that of the fundamental is called the third harmonic, and so on. When, as in the case we are illustrating, the components of a sound are integer multiples of the fundamental, the sound is called a harmonic sound. We note that in a harmonic sound the frequency of the fundamental represents the greatest common divisor of the frequencies of all of the components. It is, by definition, the maximum number that exactly divides all of the frequencies without leaving a remainder.

INTERACTIVE EXAMPLE 2A – *HARMONIC COMPONENTS*

(...)

[2] "The length of a cycle is called its **wavelength** and is measured in meters or in centimeters. This is the space that a cycle physically occupies in the air, and were sound actually visible, it would be easy to measure, for example, with a tape measure." (Bianchini, R. 2000)

other sections in this chapter:

Phase
Harmonic and non-harmonic spectra
Periodic versus aperiodic, and harmonic versus non- harmonic
Interpolation

2.2 BEATS

2.3 CROSSFADING BETWEEN WAVETABLES: VECTOR SYNTHESIS

2.4 VARIABLE SPECTRUM ADDITIVE SYNTHESIS

ACTIVITIES
- Interactive examples

TESTING
- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS
- Fundamental concepts
- Glossary

2P
ADDITIVE SYNTHESIS AND VECTOR SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapter 1 (theory and practice), Chapter 2 (theory), interlude A

LEARNING OBJECTIVES

Skills

- To learn how to synthesize a complex sound from simple sine waves
- To learn how to synthesize harmonic and non-harmonic sounds using additive synthesis and wavetables, and to transform one into the other (and vice versa) by using amplitude and frequency control
- To learn how to implement triangle waves, square waves, and sawtooth waves approximately by adding component harmonic sine waves together
- To learn how to control beats between two sine waves or harmonics
- To learn how to synthesize sounds using vector synthesis

Competence

- To successfully realize a sound study based on additive synthesis and save it to an audio file

CONTENTS

- Additive synthesis using both fixed and variable spectra
- Harmonic and non-harmonic sounds
- Phase and beats
- Interpolation
- Vector synthesis

ACTIVITIES

- Correcting algorithms
- Completing algorithms
- Replacing parts of algorithms
- Analyzing algorithms
- Constructing new algorithms

TESTING

- Integrated cross-functional project: reverse engineering
- Integrated cross-functional project: composing a brief sound study

SUPPORTING MATERIALS

- List of native Pd objects, list of Virtual Sound Library objects, list of messages for specific objects

2.1 FIXED SPECTRUM ADDITIVE SYNTHESIS

The first patch of this chapter implements an additive synthesis algorithm with a *harmonic spectrum*. As we have already learned a harmonic spectrum is one whose partials' frequencies are whole number multiples of a given fundamental frequency. The patch shown here is based on figure 2.12 in the theory part of this chapter, where 10 oscillators are added together using a mixer.
Each oscillator receives a value to set its frequency and another to determine its amplitude. To create a harmonic spectrum, all you need to do is select a fundamental frequency and multiply it by a series of successive whole numbers. This way, the first oscillator is set to the fundamental, the second to twice that frequency, the third to triple the frequency, and so forth. Figure 2.1 shows the patch contained in the file **02_01_harmonic_spectrum.pd**.

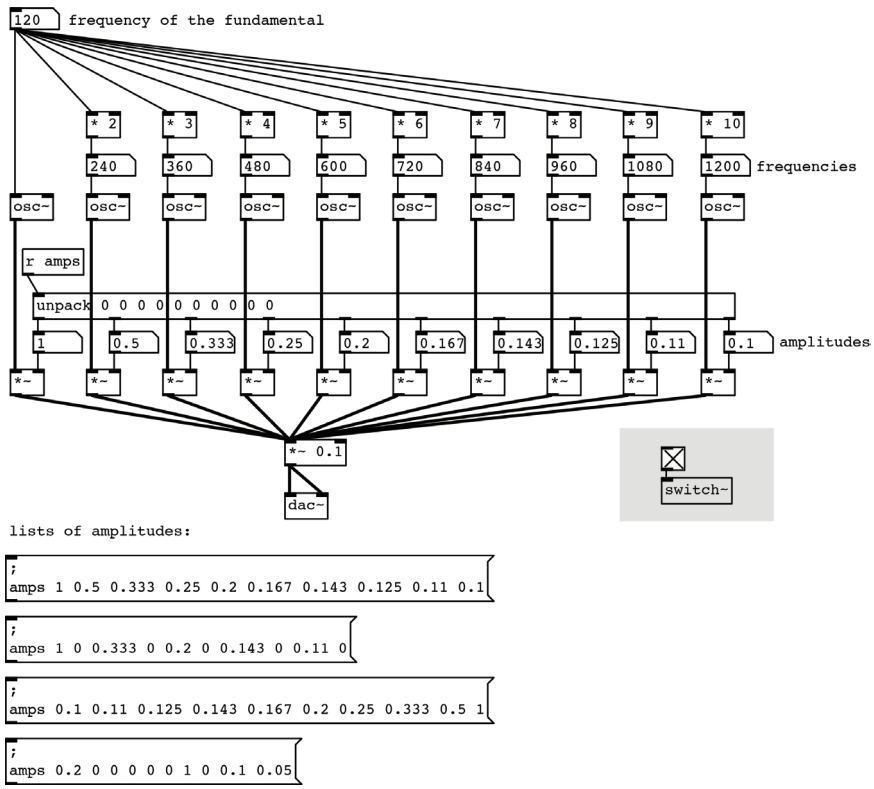


fig. 2.1 the file 02_01_harmonic_spectrum.pd

The number box on the upper left-hand side of the patch is used to set the fundamental frequency. This value goes directly into the left inlet of the first `[osc~]` object, while the other objects receive a whole number multiple of this value calculated using a `[*]` object. The output signal of each `[osc~]` is connected to a `[*~]` object in order to scale its amplitude – you will notice that each `[*~]` receives a different multiplication factor from the number box connected to it. The 10 scaled signals are sent to the left inlet of another `[*~]`

with an argument of 0.1: this means the summed signals will be scaled by 0.1 so the resulting signal does not exceed the maximum amplitude threshold[1].

This type of algorithm creates a fixed spectrum, because the partial frequencies always have the same ratio to each other and to the fundamental frequency, even if it changes. The amplitude of each partial, on the other hand, can be changed at will, in order to create different timbres. The message boxes at the bottom of the patch contain preset amplitude lists which are sent to the `[receive]` (`[r]`) object with the argument *amps*, and the unpacked into individual numbers using the `[unpack]` object.

Note that there is nothing preventing us from using the mouse to manually set the amplitudes of the partials one by one, in each number box. We just need to be sure to hold down the SHIFT key before clicking and dragging on the number box, in order to create decimal values, so we do not exceed the maximum amplitude value of 1. Listen to the different timbre presets by clicking on the message boxes (naturally, after having turned on the audio engine).

On the lower right-hand side of the patch you will notice a toggle connected to a `[switch~]` object. This can be used to locally turn the audio on and off for the patch where it is located – this includes any subpatches and abstractions used in the patch[2]. Essentially, `[switch~]` lets you decide which patches you want to "play" and which ones you want to mute, in order to avoid unwanted superimposition of multiple audio algorithms when multiple patches are open. When the toggle is used to send the '1' message to `[switch~]`, audio will be turned on in the patch where it is located. Conversely, when a '0' message is sent, audio is turned off for that patch, leaving other open patches unaffected. It goes without saying that for `[switch~]` to do its job, the main audio engine must be turned on.[3]
sinsum
The `[switch~]` object will not appear in the patch images in subsequent figures in this text for reasons of simplicity and clarity, however it will be included in the patches, themselves, for the sake of convenience. When you are creating your own patches and/or copying algorithms from this book, it can often be useful to include a toggle connected to a `[switch~]` object so you can easily control the audio in each of your open patches. However, you should be sure to create the `[switch~]` before you put any other tilde objects in your patch! If you add the `[switch~]` to your patch after other audio objects, it will not work unless you save your patch to a named file on disk.

---

[1] Each oscillator outputs a signal whose maximum amplitude value can be equal to 1. If the output signals of all the oscillators are at maximum amplitude, the maximum amplitude of their summed signal will be 10. By multiplying this signal by 0.1, we can be sure that the maximum possible amplitude value after summation will be equal to 1.

[2] The `[switch~]` object additionally has some other uses which will be discussed later.

[3] Remember that the key combination <C-/> can always be used to turn the audio engine on.

ACTIVITIES

1. Take the patch **02_01_harmonic_spectrum.pd** as a starting point and create two new amplitude lists in two new message boxes. In the first, give all odd harmonics an amplitude of 1 and even harmonics an amplitude of 0, and the second give the even harmonics an amplitude of 1 and odd harmonics an amplitude of 0. What is the most obvious difference between them? Why?

2. Again, using the patch **02_01_harmonic_spectrum.pd** as a starting point, and referring to the topic of the missing (or phantom) fundamental discussed in section 2.1 of the theory part of this chapter, create a spectrum where all the harmonic amplitudes are at 1, then slowly attenuate the amplitude of the fundamental to down to zero. You should still hear a spectrum whose fundamental is equal to the frequency that has just been silenced. Now try attenuating the amplitudes of the other harmonics to zero, in succession, one by one. At which point do you no longer hear the fundamental frequency?

PHASE

As you have probably already noticed, the `[osc~]` object has two inlets. The left inlet (the one we have been using up until now) is used to set the oscillator's frequency; its input can be either a message or a signal. The right inlet, on the other hand, is used to set the oscillator's phase, and can only receive messages. The values which can be sent to this inlet are decimal values between 0 and 1, which represent the normalized phase: a value of 0 corresponds to an angle of 0° or 0 radians, a value of 0.25 corresponds to an angle of 90° or $\pi/2$ radians, a value of 0.5 corresponds to 180° or π radians, and so forth. A normalized phase value of 1 corresponds to the end of a cycle – 360° which is also equal to 0°.

When a message is sent to the right inlet of `[osc~]`, it forces the oscillator to immediately jump to the point in the waveform indicated by the message, and continue outputting the waveform from that point onward. For example, the message '0' causes the waveform to be output from a phase value corresponding to 0° or 0 radians. Try this out by recreating the patch shown in figure 2.2.

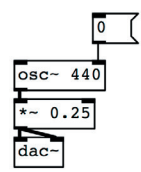


fig. 2.2 resetting the phase of an `[osc~]` object.

Turn on the audio and click repeatedly on the message box to reset the waveform's phase. You should notice a click each time you send the message, since the waveform gets interrupted mid-stream and reset to output from the beginning of the waveform (i.e., the value 0 indicated in the message box).

Figure 2.3 shows what happens to the oscillator when its phase is reset. As soon as the object receives the value from the message box, telling it to reset its phase, it jumps immediately to the point in the waveform indicated by the message, creating a *discontinuity* in the output waveform, which is heard as an audible click. When using the message '0' to reset the phase (as in figure 2.2), the message forces the oscillator to begin outputting the waveform again from the beginning of its cycle.
The beginning of a sine wave's cycle, as we learned in the theory chapter 1.2, corresponds to an amplitude value of 0. So why does the `[osc~]` object begin again from a value of 1, when its phase is reset to 0? In reality, the `[osc~]` object does not use a sine function for its output, but rather a cosine function, whose cycle starts with the maximum amplitude value (i.e., 1).

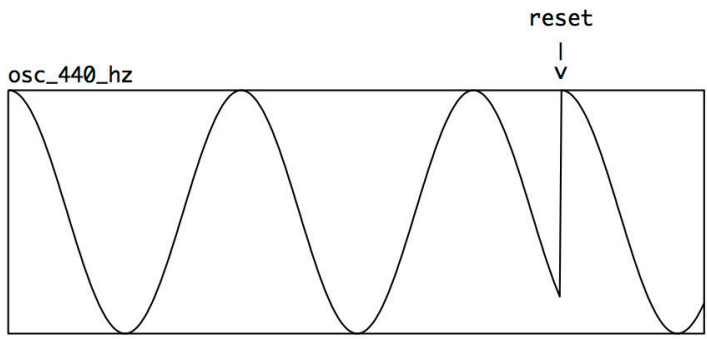


fig. 2.3 a graphical representation of a discontinuity when resetting the phase

The phase parameter is not really very interesting on its own when used this way. However, it does become interesting when applying it to two waveforms of the same frequency that are summed together. When both waveforms are exactly in phase with each other their sum is a signal with the same waveform, but having double the amplitude.

Copy the patch shown in figure 2.4, turn on the audio, and click on the message box on the left, labelled 'in phase'. You will notice that the phase of both oscillators gets reset immediately to the same value at the same time, and the sum of the oscillators' output is a waveform with double the amplitude. What happens when you click on the number box labelled 'out of phase' on the right? The phase of the oscillator on the left is reset to its initial value, while the one on the right is simultaneously reset using a phase value of 0.5, corresponding to a phase of 180°. This value creates a waveform whose phase is opposite that output by the other oscillator. Therefore, when the two waveforms are added together, sample by sample, a constant signal of 0 is created, causing silence.

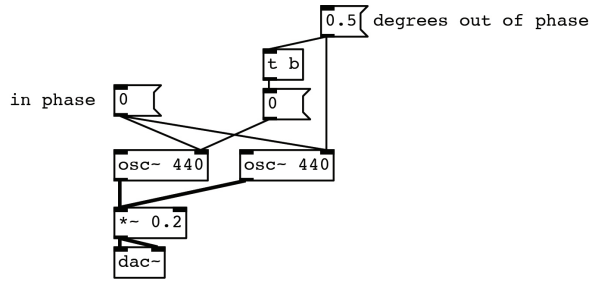


fig. 2.4 resetting the phase of two oscillators

Now open the file **02_02_phase.pd**, shown in figure 2.5. In this patch, the phase of the oscillator on the left is always reset to 0, whereas the one on the right can be reset to any value via the number box at the top of the patch. When the phase is changed, a bang causes waveforms to be written to three arrays, allowing us to visualize each of the two oscillators' waveforms and the resulting waveform when they are added. If you set the phase of the right oscillator to 0.5, you will see that the resulting sum is a constant signal equal to 0. When you set the phase to a value near 0 or 1, the amplitude of the resulting waveform exceeds the limits of -1 and 1. Try changing the phase to different values (with the audio turned on, naturally!), and you will see that the behavior is cyclic – when the phase value is near 0.5, 1.5, 2.5, etc., the two waveforms cancel each other out, and when the phase is near a whole number (0, 1, 2, 3, etc.) the sum is double the amplitude of the individual waveforms. This cyclic pattern is a result of the fact that a phase value of 1 resets the waveform to the value at 360°, which is the same as an angle of 0° – this is true of all integer multiples of 1.

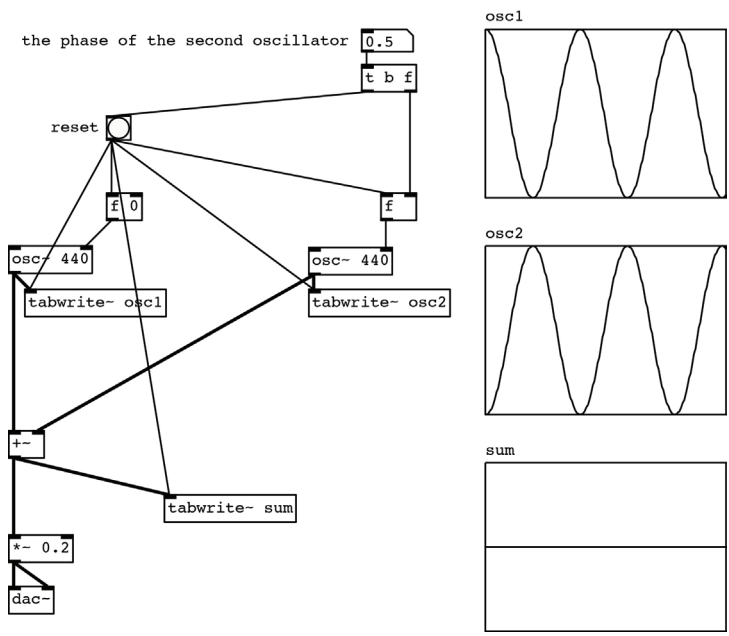


fig. 2.5: file 02_02_phase.pd

The `[vs.osc~]` object, unlike `[osc~]`, allows us to use a signal in the right inlet to control the oscillator's phase. To try this out, rebuild the patch shown in figure 2.6.

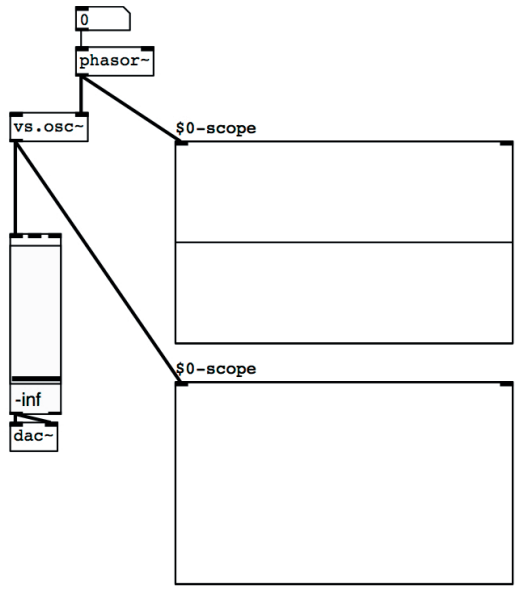


fig. 2.6: An oscillator controlled by a `[phasor~]`

In this patch, there is a `[vs.osc~]` object at 0 Hz (since the object has no argument and no numerical message has been sent to its left inlet) whose phase is controlled by a `[phasor~]` object whose frequency, as shown in the image, is 0 – in other words it is stopped. Looking at the oscilloscope, we can see that the `[phasor~]` is outputting a stream of samples whose value is 0 and the `[vs.osc~]` object is outputting a stream of samples whose value is 1 (actually, we can't readily notice this because this value corresponds to the upper border of the `[vs.scope~]` object's rectangle) – therefore both objects are stopped at the beginning of their cycle.

Now let's see what happens when we give `[phasor~]` a frequency that is greater than 0, let's say 400 Hz (as shown in figure 2.7). Try reconstructing this patch. You will notice that the `[phasor~]` controls the phase of the `[vs.osc~]` object continuously, thereby causing it to oscillate at the same frequency (and naturally also the same phase). We already learned in section 1.2 that `[phasor~]` outputs a ramp whose values go from 0 to 1, and this ramp, when applied to the phase of `[vs.osc~]` causes it to oscillate. This explains the name of the `[phasor~]` object: one of its main purposes is to be used to control the phase of another object.

(...)

other sections in this chapter:

ACTIVITIES
- Analyzing algorithms
- Completing algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Pd native objects
- List of Virtual Sound Library objects
- List of messages for specific objects

3T
NOISE GENERATORS, FILTERS, AND SUBTRACTIVE SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1 and 2 (Theory)

LEARNING OBJECTIVES

Knowledge

- To learn about the theory of subtractive synthesis
- To learn about the theory and use of the main filter parameters
- To learn about the differences between the theory of ideal filters and the actual responses of digital filters
- To learn about the theory and the response of FIR and IIR filters
- To learn how to use filters routed in series or in parallel
- To learn about the theory and use of graphic and parametric equalizers
- To learn how to use filters on different types of signals
- To learn the main elements of a typical subtractive synthesizer

Skills

- To be able to hear the basic effects caused by filters, and to describe their characteristics

CONTENTS

- Subtractive synthesis
- Lowpass, highpass, bandpass, and bandreject filters
- High shelving, low shelving, and peak/notch filters
- The Q factor
- Filter order
- Finite impulse response and infinite impulse response filters
- Graphic and parametric equalization
- Filtering signals produced by noise generators, sampled sounds, and impulses

ACTIVITIES

- Interactive examples

TESTING

- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS

- Fundamental concepts
- Glossary
- Discography

3.1 SOUND SOURCES FOR SUBTRACTIVE SYNTHESIS

In this chapter we will discuss filters, a fundamental subject in the field of sound design and electronic music, and subtractive synthesis, a widely-used technique that uses filters. A **filter** is a signal processing device that acts selectively on some of the frequencies contained in a signal, applying attenuation or boost to them.[1] The goal of most digital filters is to alter the spectrum of a sound in some way. **Subtractive synthesis** was born from the idea that brand-new sounds can be created by modifying, through the use of filters, the amplitude of some of the spectral components of other sounds.

Any sound can be filtered, but watch out: you can't attenuate or boost components that don't exist in the original sound. For example, it doesn't make sense to use a filter to boost frequencies around 50 Hz when you are filtering the voice of a soprano, since low frequencies are not present in the original sound.

In general, the source sounds used in subtractive synthesis have rich spectra so that there is something to subtract from the sound. We will concentrate on some of these typical source sounds in the first portion of this section, and we will then move on to a discussion of the technical aspects of filters.

Filters are used widely in studio work, and with many different types of sound:

> Sounds being produced by noise generators, by impulse generators, by oscillator banks, or by other kinds of signal generators or synthesis

> Audio files and sampled sounds

> Sounds being produced by live sources in real time (the sound of a musician playing an oboe, captured by a microphone, for example)

NOISE GENERATORS: WHITE NOISE AND PINK NOISE

One of the most commonly used source sounds for subtractive synthesis is **white noise**, a sound that contains all audible frequencies, whose spectrum is essentially flat (the amplitudes of individual frequencies being randomly distributed). This sound is called white noise in reference to optics, where the color white is a combination of all of the colors of the visible spectrum. White noise makes an excellent source sound because it can be meaningfully filtered by any type of filter at any frequency, since all audible frequencies are present. (A typical white noise spectrum is shown in figure 3.1.)

[1] Besides altering the amplitude of a sound, a filter modifies the relative phases of its components.

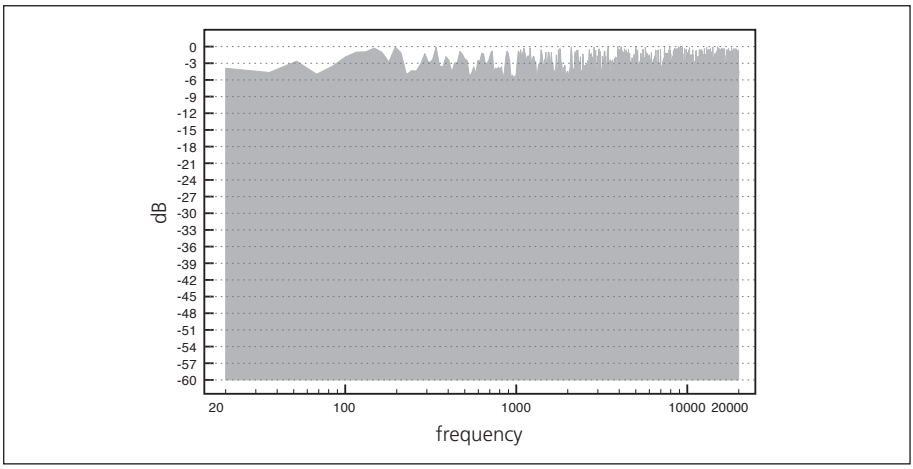


Fig. 3.1 The spectrum of white noise

Another kind of noise that is used in similar ways for subtractive synthesis is **pink noise**. This kind of sound, in contrast to white noise, has a spectrum whose energy drops as frequency rises. More precisely, the attenuation in pink noise is 3 dB per octave;[2] it is also called 1/f noise, to indicate that the spectral energy is proportional to the reciprocal of the frequency. (See figure 3.2.) It is often used, in conjunction with a spectral analyzer, to test the frequency response of a musical venue, in order to correct the response based on some acoustic design.

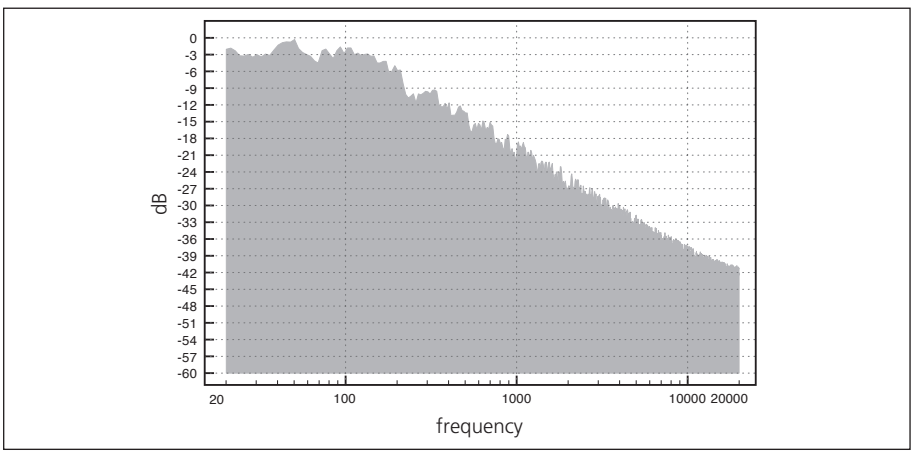


Fig. 3.2 The spectrum of pink noise

[2] Another way to define the difference between white noise and pink noise is this: while the spectrum of white noise has the same energy at all frequencies, the spectrum of pink noise *distributes the same energy across every octave*. A rising octave, designated anywhere in the spectrum, will occupy a raw frequency band that is twice as wide as its predecessor's; pink noise distributes equal amounts of energy across both of these frequency bands, resulting in the constant 3 dB attenuation that is its basic property.

In digital systems, white noise is generally produced using random number generators: the resulting waveform contains all of the reproducible frequencies for the digital system being used. In practice, random number generators use mathematical procedures that are not precisely random: they generate series that repeat after some number of events. For this reason, such generators are called **pseudo-random generators**.

By modifying some of their parameters, these generators can produce different kinds of noise. A white noise generator, for example, generates random samples at the sampling rate. (If the sampling rate is 48,000 Hz, for example, it will generate 48,000 samples per second.) It is possible, however, to modify the frequency at which numbers are generated – a generating frequency equal to 5,000 numbers a second, for example, we would no longer produce white noise, but rather a noise with strongly attenuated high frequencies.

When the frequency at which samples are generated is less than the sampling rate, "filling in the gaps" between one sample and the next becomes a problem, since a DSP system (defined in the glossary for Chapter 1T) must always be able to produce samples at the sampling rate. There are various ways of resolving this problem, including the following three solutions:

> **Simple pseudo-random sample generators**
These generate random values at a given frequency, maintaining a constant value until it is time to generate the next sample. This results in a waveform resembling a step function. In figure 3.3 we see the graph of a 100 Hz noise generator; the random value is repeatedly output for a period equal to 1/100 of a second, after which a new random value is computed. If the sampling rate were 48,000 Hz, for example, each random value would be repeated as a sample 48,000 / 100 = 480 times.

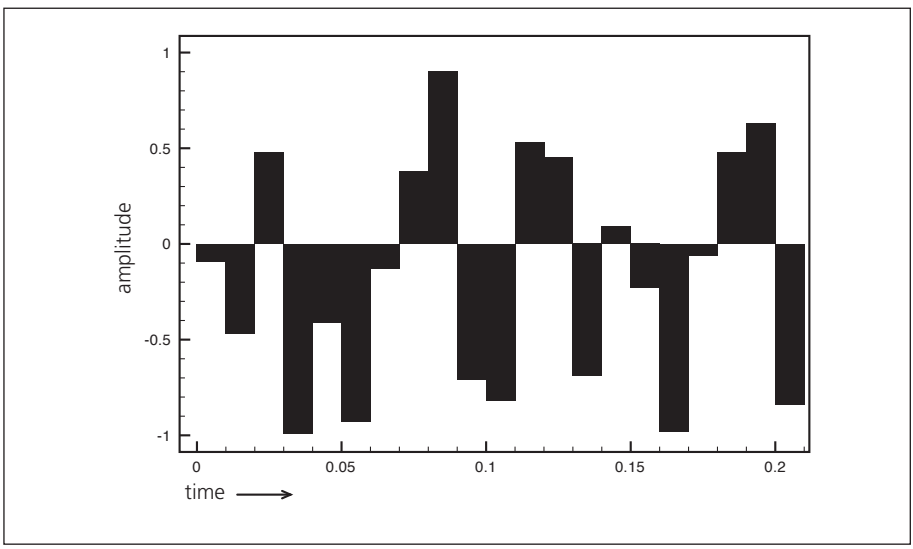


Fig. 3.3 Generation of pseudo-random values

> **Interpolated pseudo-random sample generators**
These generators use interpolation between each random number and the next. (See the section on linear interpolation in Chapter 2.1.) As you can see in figure 3.4, intermediate samples, produced during the gaps between random value computations, follow line segments that move gradually from one value to the next.

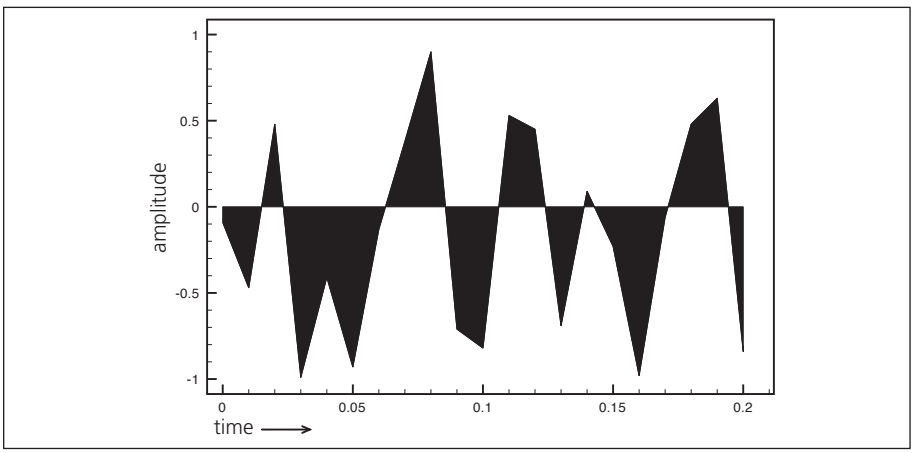


Fig. 3.4 Generation of pseudo-random values with linear interpolation

Interpolation between one value and the next can be linear, as shown in the figure, or polynomial, implemented using polynomial functions to connect the values using curves rather than line segments. (Polynomial interpolation is shown in figure 3.5, however, we will not attempt to explain the details here.) The kinds of polynomial interpolation most common to computer music are quadratic (which use polynomials of the second degree) and cubic (which use polynomials of the third degree). Programming languages for synthesis and signal processing usually have efficient algorithms for using these interpolations built in to their runtimes, ready for use.

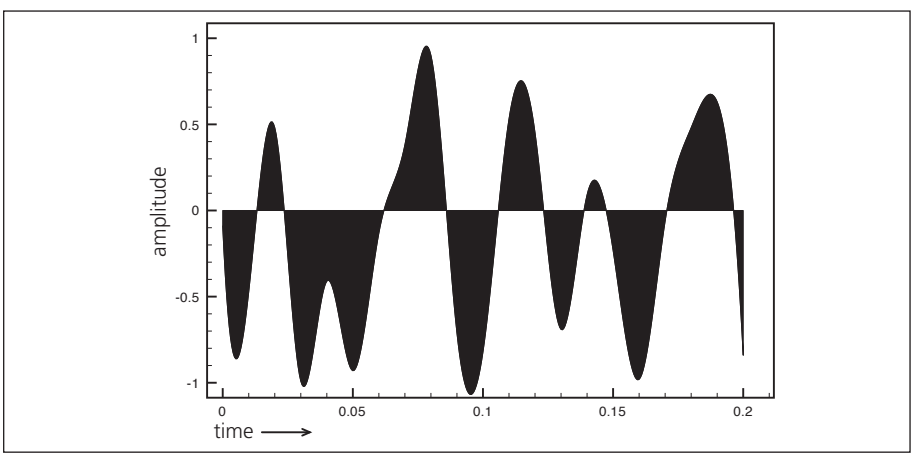


Fig. 3.5 Generation of pseudo-random values with polynomial interpolation

> **Filtered pseudo-random sample generators**
In this kind of approach, the signal produced is filtered using a lowpass filter. We will speak further of this kind of generator in the section dedicated to lowpass filters.

INTERACTIVE EXAMPLE 3A – *NOISE GENERATORS* – PRESETS 1-4

OSCILLATORS AND OTHER SIGNAL GENERATORS

In Section 1.2T, we examined the "classic" waveforms that are often found in synthesizers, such as the square wave, the sawtooth wave, and the triangle wave. Section 2.1T explained how these waveforms, when geometrically perfect (perfect squares, triangles, etc.), contain an infinite number of frequency components. The presence of infinitely large numbers of components, however, causes nasty problems when producing digital sound, since an audio interface cannot reproduce frequencies above half of its sampling rate.[3] (We will discuss this topic in much greater detail in Chapter 5.) When you attempt to digitally reproduce a sound that contains component frequencies above the threshold for a given audio interface, undesired components will appear, which are almost always non-harmonic. To avoid this problem, **band-limited oscillators** are often used in digital music. Such oscillators, which produce the classic waveforms, are built so that their component frequencies never rise above half of the sampling rate. The sounds generated by this kind of oscillator therefore make a good point of departure for creating sonorities appropriate for filtering, and as a result, they are the primary source of sound in synthesizers that focus on subtractive synthesis. In Section 3.5 we will analyze the structure of a typical subtractive synthesizer.

It is, of course, also possible to perform subtractive synthesis using synthetic sounds, rich in partials, that have been realized using other techniques such as non-linear synthesis or physical modeling. We will cover these approaches in following chapters.

FILTERING SAMPLED SOUNDS

Beyond subtractive synthesis, one of the everyday uses of filters and equalizers is to modify sampled sounds. Unlike white noise, which contains all frequencies at a constant amplitude, a sampled sound contains a limited number of frequencies, and the amplitude relationships between components can vary from sound to sound. It is therefore advisable, before filtering, to be conscious of the frequency content of a sound to be processed.

[3] It is for this reason that sampling rate of an audio interface is almost always more than twice the maximum audible frequency for humans.

Remember that you can only attenuate or boost frequencies that are already present. This is true for all sounds, sampled or otherwise, including those captured from live sources.

(...)

other sections in this chapter:

ACTIVITIES
- Interactive examples

TESTING
- Listening and analysis

SUPPORTING MATERIALS
- Fundamental concepts
- Glossary

3P
NOISE GENERATORS, FILTERS, AND SUBTRACTIVE SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1 and 2 (theory and practice), Chapter 3T, Interlude A

LEARNING OBJECTIVES

Skills

- Generate and control different types of complex signals for subtractive synthesis (white noise, pink noise, impulses, etc.)
- Construct algorithms using lowpass, highpass, bandpass, band-reject, shelving, and resonant filters, and how to control them using various parameters, Q, and filter order
- Implement FIR (non-recursive), as well as IIR (recursive) filters
- Build a simple subtractive synthesizer
- Write algorithms using filters connected in series and in parallel
- Build graphic and parametric equalizers

Competence

- To be able to realize a short sound study based on the techniques of subtractive synthesis and save it to an audio file.

CONTENTS

- Sources for subtractive synthesis
- Lowpass, highpass, bandpass, band-reject, shelving, and resonant filters
- The Quality factor and filter order
- FIR and IIR filters
- Connecting filters in series and in parallel
- Graphic and parametric equalizers

ACTIVITIES

- Activities using computer: replacing parts of algorithms, correcting, completing and analyzing algorithms, constructing new algorithms

TESTING

- Composing a brief study - Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of Pd native objects - List of Virtual Sound library objects – List of messages for specific objects - Glossary

3.1 SOUND SOURCES FOR SUBTRACTIVE SYNTHESIS

As you learned in section 3.1 of the theory chapter, the purpose of a filter is to modify the spectrum of a signal in some manner. So, let us first introduce an object of the *Virtual Sound* library that is used to view such spectrum: `[vs.spectroscope~]`. To create this object, simply write its name inside a generic object box.

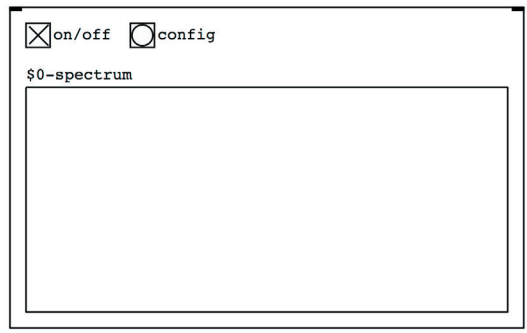


fig. 3.1: the `[vs.spectroscope~]` object

Now open the file **03_01_spectroscope.pd** (see fig. 3.2).

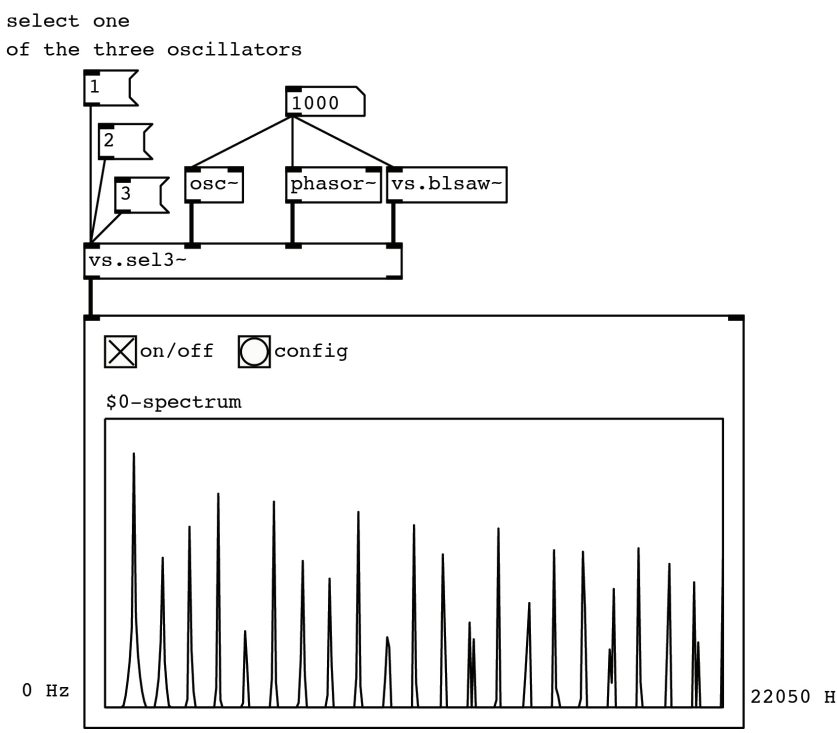


fig. 3.2: file 03_01_spectroscope.pd

In this case, the spectroscope has been connected to a `[vs.sel3~]` object which allows us to select one of three oscillators: a sine wave (`[osc~]`), a non-bandlimited sawtooth wave (`[phasor~]`) and a bandlimited sawtooth wave (`[vs.blsaw~]`)[1]. In order to select one of the three oscillators, three message boxes have been connected to the left input of `[vs.sel3~]`; by setting up the frequency of the number box and selecting the oscillators in turn, we can view their spectra. Notice that the sine wave shows a single partial, whereas the `[phasor~]` object, which generates a non-bandlimited waveform, shows the richest spectrum, comprising many partials[2].

Make some tests by changing the frequency and selecting the various waveforms while looking at the images displayed in the spectroscope. As previously mentioned, what is displayed is the spectrum of the input sound; the partials are displayed in ascending order from left to right over a frequency range from 0 to 22050 Hz.

Try adding the `[vs.spectroscope~]` object to the patches you have already created, so as to get the hang of the relation between the sound and its spectral content; you can also add the object to the patches of the previous chapters. For example, try adding it to **01_13_audiofile.pd** (connect the spectroscope to the left output of the object `[vs.splayer~]`) or **IA_06_random_walk.pd** (connect it to the output of `[pd monosynth]`): do you understand the relation between the sound and the shape of its spectrum in this last patch?

Let us move on to a discussion about white noise, which in Pd can be generated using the `[noise~]` object (see figure 3.3).

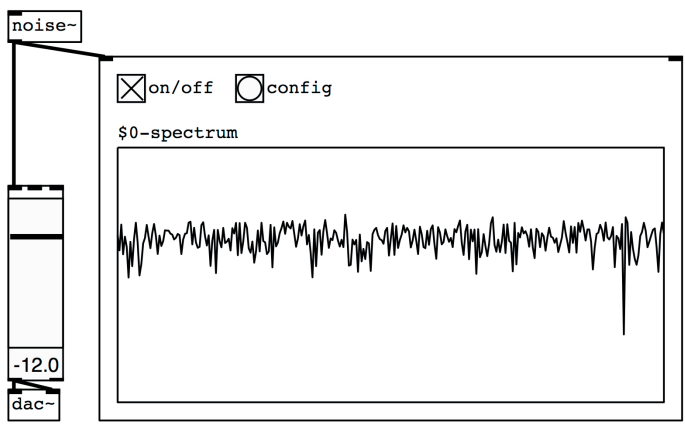


fig. 3.3: white noise generator

[1] See section 1.2

[2] The `[vs.spectroscope~]` object employs a spectral analysis algorithm called the *Fast Fourier Transform*, which we will cover in detail in a later chapter

For the patch in this image (which we strongly encourage you to recreate) we connected the noise generator to the object **[vs.spectroscope~]** and we can now see how the noise spectrum contains energy at every frequency. Unlike the other sound generators we dealt with so far, the white noise generator does not need any parameter; in fact, it is solely used to generate a signal from random values in the range between -1 and 1 at the current sample rate (refer to section 3.1 of the theory chapter). Another type of noise generator we can find in the *Virtual Sound* library is the object **[vs.pink~]** which generates pink noise (fig. 3.4).

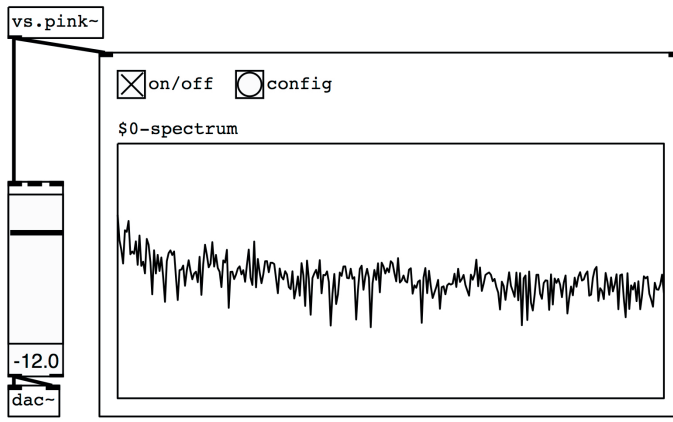


fig. 3.4: pink noise

Note that the spectrum of pink noise, unlike that of white noise, gradually diminishes in amplitude as frequencies get higher, and that this attenuation (as we know from section 3.1T) is 3 dB per octave.
Recreate the simple patch shown in the figure and listen carefully to the difference between pink noise and white noise. Which of the two seems more pleasant (or maybe just less unpleasant), and why?
Add a (**[vs.scope~]**) oscilloscope to the two patches you have just created and look at the differences between the waveforms of the white and pink noise. Fig. 3.5 shows these two waveforms side by side.

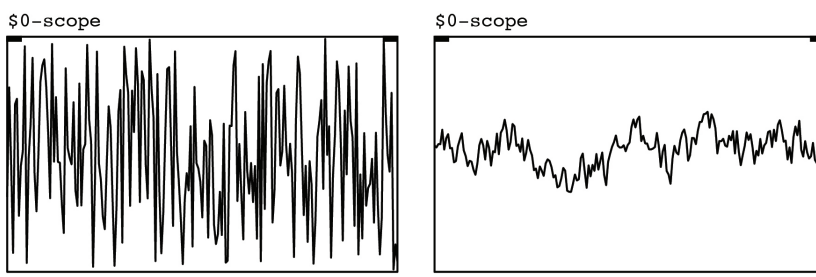


fig. 3.5 white noise and pink noise waveforms

Without getting bogged down in technical details, while white noise is basically a stream of random values, pink noise is generated using a more complex algorithm in which a sample, although randomly generated, cannot stray too far from the value of its predecessor. This results in the "serpentine" waveform that we see in the figure. The behavior of the two waveforms demonstrates their spectral content: when the difference between one sample and the next is larger, the energy of the higher frequencies in the signal is greater[3]. White noise has more energy at higher frequencies than pink noise.

Another interesting generator is `[vs.rand1~]`[4], which generates random samples at a selectable frequency and connects these values using line segments (as shown in figure 3.6). Unlike `[noise~]` and `[vs.pink~]`, which generate random samples on every tick of the DSP "engine" (producing a quantity of samples in one second that is equal to the sample rate), with `[vs.rand1~]` it is possible to choose the frequency at which random samples are generated, and to make the transition from one sample value to the next gradual, using linear interpolation.

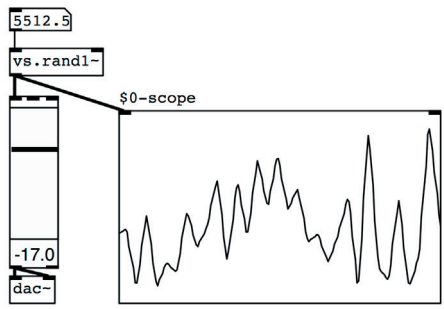


fig. 3.6: the `[vs.rand1~]` object

Obviously, this generator produces a spectrum that varies according to the frequency setting; it shows a primary band of frequencies that ranges from 0 Hz up to the frequency setting, followed by additional bands that are gradually attenuated and whose width are also equal to the frequency setting. Figure 3.7 shows this interesting spectrum.

[3] To understand this assertion, notice that the waveform of a high sound oscillates quickly, while that of a low sound oscillates slowly. At equal amplitudes, the difference between succeeding samples for the high frequency case will be larger, on average, than for the low frequency case.

[4] The prefix *vs* means that the object is part of the *Virtual Sound* library, just like all the other objects using this prefix.

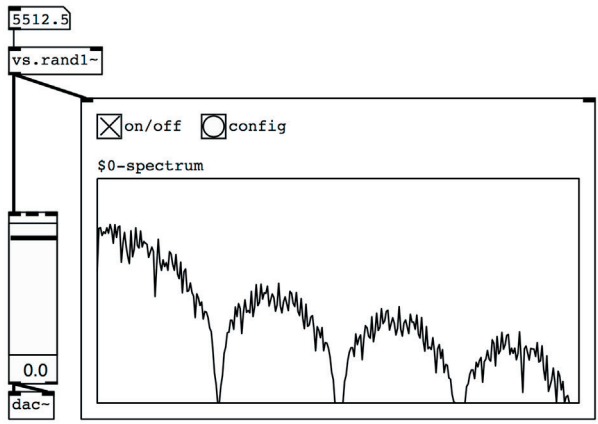


fig. 3.7: the spectrum generated by the `[vs.rand1~]` object

In the example, we can see that the frequency of the `[vs.rand1~]` object is 5,512.5 Hz (a quarter of the maximum frequency visible in the spectroscope in the figure), and the first band goes from 0 to 5,512.5 Hz. After this come secondary bands, progressively attenuated, all 5,512.5 Hz wide. Changing the frequency of `[vs.rand1~]` changes the width of the bands as well as their number. If you double the frequency to 11,025 Hz, for example, you will see precisely two bands, both 11,025 Hz in width.

Another noise generator is `[vs.rand0~]` (the last character before the tilde is a zero), that generates random samples at a given frequency like `[vs.rand1~]`, but does not interpolate between the values produced. Instead, it maintains the value of each sample until a new sample is generated, producing stepped changes.

Its spectrum forms bands in the same way as that of `[vs.rand1~]` in figure 3.7, but as you can see in figure 3.8, the attenuation of the secondary bands is much less because of the abrupt changes between sample values.

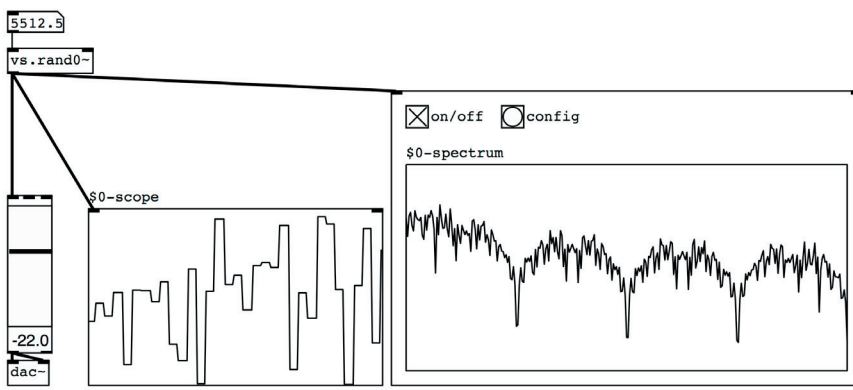


fig. 3.8: the `[vs.rand0~]` object

A noise generator with polynomial interpolation `[vs.rand3~]` can also be found in the *Virtual Sound* library (see fig. 3.9).

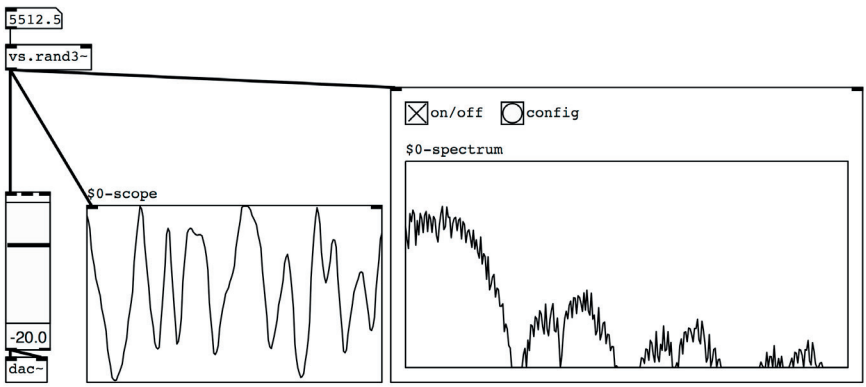


fig. 3.9: the `[vs.rand3~]` object

Thanks to the polynomial interpolation in this object, the transitions between one sample and another appear "smooth", as you can see on the oscilloscope. The transitions form a curve rather than a series of connected line segments, and the resulting effect is a strong attenuation of the secondary bands. Recreate the patches found in figures 3.6 to 3.9 in order to experiment with various noise generators.
"Classic" oscillators – those that produce sawtooth waves, square waves, and triangle waves – are other examples of sound sources that are rich in components, which makes them effective for use with filters. In section 1.2, we examined three band-limited oscillators that generate such waveforms: `[vs.blsaw~]`, `[vs.blsquare~]` and `[vs.bltri~]`. We will use these oscillators frequently during the course of this chapter.

In section 3.1T, we also learned about the possibility of filtering sampled sounds. For this reason, we will also give examples in this chapter of filtering sampled sounds, using the `[vs.splayer~]` object (first introduced in section 1.5P).

(...)

other sections in this chapter:

ACTIVITIES
- Replacing parts of algorithms
- Correcting algorithms
- Completing algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Pd native objects
- List of Virtual Sound Library objects
- List of messages for specific objects
- Glossary

Interlude B
ADDITIONAL ELEMENTS OF PROGRAMMING WITH PURE DATA

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1, 2, and 3 (theory and practice), Interlude A

LEARNING OBJECTIVES

Skills

- Use simple midi objects and signals
- Implement recursive operations in Pd
- Build an arpeggiator that exploits randomly generated intervals
- Route signals and messages to switched inlets and outlets
- Compare values and analyze their relation
- Take apart lists of data
- Program repeating sequences using iterative structures
- Generate random lists for simulating resonant bodies
- Implement the Shepard tone, or "infinite glissando"

CONTENTS

- Basic use of the MIDI protocol
- Recursive operations and repeating sequences
- Arpeggiators and random intervals
- Comparing values, converting, and routing signals and messages
- Taking lists apart, and generating random lists
- Shepard Tones

ACTIVITIES

Activities using computer:

- Replacing parts of algorithms, correcting, completing, analyzing algorithms, constructing new algorithms

TESTING

- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of Pd native objects – List of Virtual Sound library objects - Glossary

IB.1 INTRODUCTION TO MIDI

MIDI is a protocol for communicating between electronic and/or digital musical instruments and computers. Using a physical MIDI cable, it is possible to connect a synthesizer to a computer, enabling the computer to "play" the synthesizer, for example. Using the MIDI protocol, the computer sends the synthesizer information on what notes to play at what volume, and other information.
Furthermore, MIDI instruments do not need to exist in physical form: they can also run as "virtual" instruments, which are computer applications that simulate the behavior of real instruments and produce sound through audio interfaces. Such digital instruments can communicate via MIDI, creating a virtual connection (i.e. using no physical cables), between the application that sends MIDI messages (like Pd) and the application that receives MIDI messages (the virtual instrument itself).
As we will later investigate more closely, Pd features various objects that use the MIDI protocol and since we are going to look at some of them in this Interlude, it is important that you configure your system properly, so that Pd can communicate with the MIDI devices in your computer. Generally speaking, in order to connect Pd to a virtual MIDI device on a *Windows* OS, all you need to do is open Pd and set the *Output Device 1* (found in the *Media/MIDI Settings* menu) to *Microsoft GS Wavetable*[1].

For Mac users:

1. open the MIDI Audio configuration window (accessible from the folder */Application/Utility* or from *Launchpad*) and make sure the IAC driver is enabled
2. open Pd and select IAC Driver in the *Output Device 1* menu, found in the *Media/MIDI* settings menu
3. open a software that can host virtual instruments (*Garageband* or the like).

Once you have configured MIDI, open the file **IB_01_MIDI_note.pd**; fig. IB.1 shows the upper section of the patch.

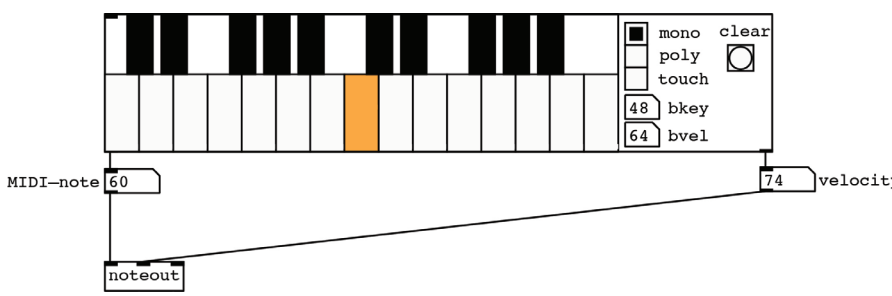


fig. IB.1: the upper section of the file IB_01_MIDI_note.pd

[1] The name could be different depending on the version of Windows OS, but do not worry, you will still recognize the device!

We have connected the `[vs.midikeyboard]` object (the musical keyboard) to some number boxes, which are in turn connected to a `[noteout]` object. As we have already learned, clicking on one of the `[vs.midikeyboard]` keys will generate the MIDI note value for the selected key on the left outlet. (We first used this object in section 1.4.) Pressing a key also generates a velocity value on the right outlet that represents the dynamic level of a note; clicking on the upper part of a `[vs.midikeyboard]` key will produce a higher velocity value, while clicking on the lower part will produce a lower value. Values for velocity can vary between 1 and 127.[2] MIDI note and velocity values are sent to the left and center inlets of a `[noteout]` object, which then sends the appropriate command to any MIDI instruments (real or virtual) that are connected to it.[3]

In the MIDI protocol, this message is called the *note-on* command. When you click close to the top of a `[vs.midikeyboard]` key, generating a high enough velocity value (let's say above 90), you should hear the sound of a piano, assuming you have configured MIDI correctly[4]. This sound is not coming from Pd, but it is generated by a virtual instrument that is part of the operating system of your computer, which, by default, plays MIDI notes using the sound of a piano. If you try playing more notes, you will realize that everytime `[noteout]` receives note and velocity, a new note is played, without interrupting or stopping the notes previously played. A slight problem remains. Using `[noteout]`, we have told the virtual instrument when to begin playing, without telling it when to stop!
To interrupt a note and solve this problem, we will need to send another MIDI command: a matching MIDI note with a velocity equal to 0, which is called the ***note-off*** command. (A sort of "remove finger from key" command.)
In order to "turn a MIDI note off" using `[vs.midikeyboard]`, we need to change the way it manages MIDI note messages; select the mode called *poly* in the right section of the object (up until now, you have been working in *mono* mode).
The first time that you click on a `[vs.midikeyboard]` key, the note will sound at the velocity chosen, and the second time that you click on the same key, the note will be sent again, but this time with a velocity of 0, which will make the sound stop: try this! This mode is called *poly* which stands for polyphonic, since – unlike *mono*, i.e. monophonic – it allows you to play more than one note at the same time).

---

2 The value 0 turns the note off.

3 The right inlet of the `[noteout]` object is used to set the MIDI channel, which we do not need at this moment. Further details will be revealed later on.

4 If you cannot hear anything, it means you did not manage to configure the system properly. If that is the case, go back at the beginning of the chapter.

Now turn your attention to the lower half of the file **IB_01_MIDI_note.pd**, shown in figure IB.2.

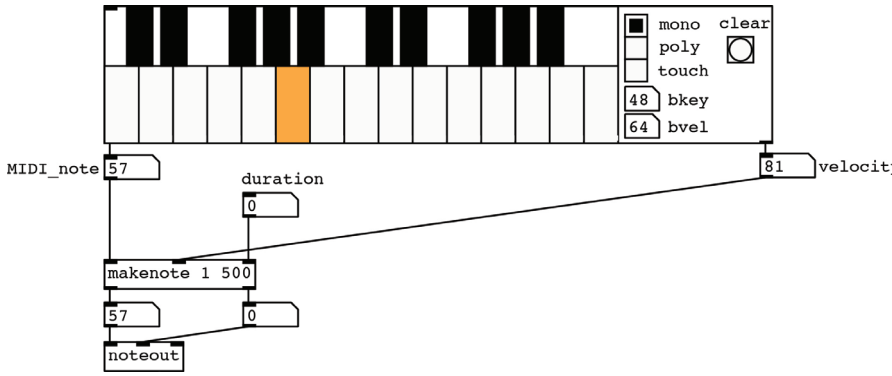


fig. IB.2: the lower half of the file IB_01_MIDI_note.pd

In this case, we have connected the **[vs.midikeyboard]** object (in *mono* mode) to the **[makenote]** object. Every time this object receives a MIDI *note-on* command, it generates a corresponding MIDI *note-off* command after a specified length of time has elapsed. The object has three inlets, one for MIDI note number, one for velocity, and one for the duration in milliseconds (i.e., the amount of time to pass before the *note-off* command is sent). It also has two outlets, one for MIDI note number and the other for velocity.

There are two arguments, a velocity and a duration expressed in milliseconds. In the patch, we have set velocity to 1, and duration equal to 500 milliseconds (or half a second). When the object receives a *note-on*, it will send its value directly to the outlets, where, after the duration given by the argument (500 milliseconds in this example) a *note-off* is sent. Note that the velocity sent by the **[vs.midikeyboard]** (which in the example shown in the figure is a value of 81) overrides the value of 1 that had been provided as an argument. In fact, we only added this value in order to be able to insert the second argument (duration), since both arguments are obligatory!

Duration can also be modified by sending a new value to the right inlet, which overwrites the value originally specified in the second argument.

Try playing some notes and changing the duration of the **[makenote]** object: observe how velocity values first adopt the values generated by **[vs.midikeyboard]**, and then, after the time specified by the duration parameter, they go back to 0.

Now insert an addition object to the lower part of the patch as shown in figure IB.3:

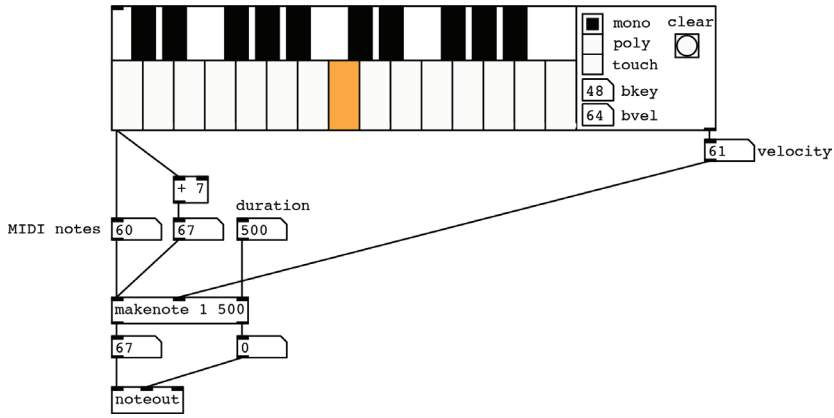


fig. IB.3: Transposing MIDI notes

This patch is similar to the one in the file **IA_01_transposition.pd**, which we examined in the first section of interlude A. Also in this case, every key pressed on the `[vs.midikeyboard]` object generates two notes simultaneously, separated by a distance of 7 semitones, the interval of a perfect fifth. Each time that a `[vs.midikeyboard]` key is pressed, in fact, the value of the corresponding MIDI note is sent to the `[makenote]` object and simultaneously to the addition object, that adds 7 to it and then sends its output to `[makenote]`. In creating these pairs of notes, the velocity and duration values need not be repeated, since they are stored in the internal variables of [makenote] until new values come along. They are recalled from the object every time a note value arrives at the left, hot inlet. In the figure, for example, both notes (the middle C and G) will have a velocity of 61 and a duration of 500 milliseconds.
From this example, you can see that the rule about "hot" inlets being on the left is complementary with right-to-left execution, as discussed. The first messages to be processed are those on the right, and processing these "cold" inlets initializes the internal variables in the object (such as the value for velocity used by [makenote]); the last message to be processed is to the "hot" inlet on the left, which causes output to occur only after internal variables have been updated.

(...)

other sections in this chapter:

ACTIVITIES
- Analyzing algorithms
- Completing algorithms
- Replacing parts of algorithms
- Correcting algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Pd native objects
- List of Virtual Sound Library objects
- Glossary

4T
CONTROL SIGNALS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1, 2, and 3 (Theory)

LEARNING OBJECTIVES

Knowledge

- To learn about the theory and practice of low frequency oscillators
- To learn about the use of DC Offset with LFOs
- To learn about the use of frequency modulation for vibrato
- To learn about the use of amplitude modulation for tremolo
- To learn about the use of pulse-width modulation
- To learn how to use LFOs to control filters
- To learn about the use of pseudo-random signal generators for control
- To learn how to use LFOs to control location in stereo and multichannel systems

Skills

- To be able to hear and describe LFO-controlled modulations of basic parameters

CONTENTS

- Low frequency oscillators: depth, rate, and delay
- Managing LFO parameters and using DC offset
- Managing vibrato, tremolo, and PWM using LFOs
- Managing filter parameters using LFOs
- Positioning and moving sound in stereo and multichannel systems
- Modulating control oscillators with pseudo-random LFOs

ACTIVITIES

- Interactive examples

TESTING

- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS

- Fundamental concepts
- Glossary

4.1 CONTROL SIGNALS: STEREO PANNING

As we have seen throughout this book, envelope generators can be used to vary the parameters of a sound, such as frequency or amplitude, in time. Signals that come from envelope generators – signals that are expressly produced to vary parameter values rather than to produce sound – are called **control signals**.

To this point, we have used line segments and exponential curves to describe control signals. These techniques are efficient because they require only a few values to describe parameter changes; think of the four segments that we use for ADSR envelopes, or of the numbers that define an exponential curve, which can completely describe a glissando. Other control signals, however, may need to vary in more complex ways. Take the vibrato associated with a string instrument as an example: there is a continuous change in the frequency of a note played with vibrato that can best be described as an oscillation around a central pitch. To simulate such a vibration using an envelope, we might need to use tens, or even hundreds, of line segments, which would be both tedious and impractical. Instead of resorting to such primitive methods, we might instead choose to use a **control oscillator**, an oscillator whose output is produced for the sole purpose of providing parameter values to *audio oscillators* or other parameterized devices used in sound synthesis and signal processing.

Control oscillators are usually **Low Frequency Oscillators (LFOs)**; their frequency is usually below 30 Hz. They produce continuously changing control values that trace waveforms in the same way that audio oscillators do. Every instantaneous amplitude of a wave generated by a control oscillator corresponds to a numeric value that can be applied to audio parameters as needed.

Here is an example demonstrating the use of an LFO: in Figure 4.1, you see a graphic representation of an LFO controlling the position of a sound in space. This LFO generates a sine wave that oscillates between MIN (representing its minimum value) and MAX (its maximum value).

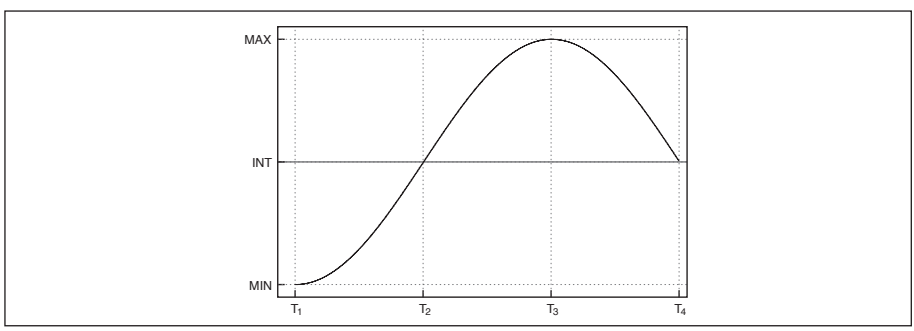


Fig. 4.1 An LFO for controlling the position of a sound in space

The minimum and maximum values, or the **depth** of the oscillator, define the limits of the amplitude values being produced, while the **rate** of the oscillator is a value that defines its frequency.

In the example, the instantaneous amplitude values of the sine wave generated by the LFO is used as input to two multipliers that inversely scale the amplitude of an audio oscillator on two output channels. Whenever the control signal reaches MIN (its minimum value), the sound is panned completely left, and whenever it reaches MAX (its maximum value), the sound is panned completely right. While intermediate values are being produced (represented in the figure by the central value INT), the sound is smoothly mixed between the two channels.

It should be obvious that it would be possible to use other waveforms (triangle, random, etc.) to control parameter values in the same way; a square wave, for example, could control the location of a sound bouncing between left and right channels without intermediate positions. In this case, there would be no continuous change, as there is when using a sine wave; the values would simply alternate between MIN and MAX.

INTERACTIVE EXAMPLE 4A • *Panning using different LFO waveforms*

The rate with which values change depends on the frequency assigned to a given control oscillator. If you use a frequency of 1 Hz, you will move from MAX to MIN and back again in one second; if you use a frequency of .2 Hz, you will have 1 complete oscillation in 5 seconds. What if you use a frequency of 220? In this case, the 220 oscillations per second would be too fast to allow us to hear the location moving between left and right; this frequency would instead enter the audio range and would generate new components in the spectrum of the resulting sound. We will cover this phenomenon, *amplitude modulation*, in Chapter 10.

INTERACTIVE EXAMPLE 4B • *Panning using a sine wave LFO at various frequencies*

By using control oscillators, we can control the depth and the rate of a vibrato, of a tremolo, or of variations in filter parameters, all of which we will cover in the following sections.

(...)

other sections in this chapter:

ACTIVITIES
- Interactive examples

TESTING
- Listening and analysis

SUPPORTING MATERIALS
- Fundamental concepts
- Glossary

4P
CONTROL SIGNALS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1, 2, and 3 (theory and practice), Chapter 4 (theory), interludes A & B

LEARNING OBJECTIVES

Skills

- Move a sound within a stereo field
- Implement Vibrato
- Simulate instruments whose frequency is controlled, such as a theremin
- Implement tremolo
- Build pulse width modulation algorithms
- Vary cutoff frequency, center frequency, and Q of filters using oscillating control signals
- Use pseudo-random signal generators for Control
- Locate and move sounds in a system of 4 or more Channels using Control signals

Competence

- To be able to create a short sound study based on the technique of controlling parameters using LFOs

CONTENTS

- Low frequency oscillators: depth, rate, and delay
- Managing LFO parameters and using DC offset
- Managing vibrato, tremolo, and pulse width modulation using LFOs
- Managing filter parameters using LFOs
- Pseudo-random control signals
- Positioning and moving sound in stereo and multi-Channel systems

ACTIVITIES

- Replacing parts of algorithms - Correcting algorithms - Analyzing algorithms - Completing algorithms - Constructing new algorithms

TESTING

- Integrated cross-functional project: composing a brief sound study
- Integrated Cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of Pd native objects – List of Virtual Sound library objects - Glossary

4.1 CONTROL SIGNALS: STEREO PANNING

You can use the output of an ordinary `[osc~]` object as a sine wave control signal for positioning a signal within a stereo field, as described in section 4.1T. The frequency of the `[osc~]` object, in this case, should be low enough to be below the audio range.
As you learned in section 1.6, you can specify the position of a signal within the stereo field using a value *n* between zero and one. The square root of this value is the amplitude multiplication factor of a channel, while the square root of 1 – *n* is the same for the other channel.
The patch shown in figure 4.1 (which we strongly encourage you to recreate), shows the algorithm used to control the position within the stereo field of a signal, which, in this case, is a bandlimited sawtooth wave.

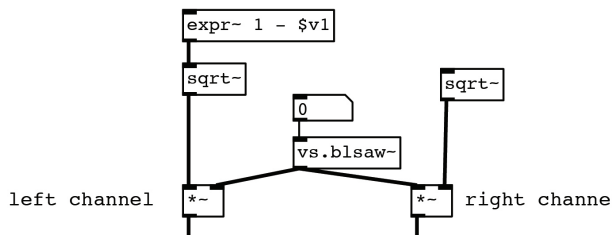


fig. 4.1: the algorithm for stereo panning

Now we need to connect the signal generated by `[osc~]` to the inlet of the `[expr~]`[1] object and to the inlet of the right `[sqrt~]`, in order to control the stereo positioning. As we already know, `[osc~]` generates a sine wave with values ranging from -1 to 1, but what we need is a signal whose values range from 0 to 1. We might as well adjust the oscillation range of `[osc~]` by doing some simple math, but we are going to cover such method in the next section. In this case, we will use an object we have already discussed in Interlude IB.9.

[1] Remember that the `[expr~]` works just like `[expr]`, but it is used for signals. The variables used for a signal are always preceded by the prefix '$v', followed by the number of the variable. The left inlet should always be a signal, hence the first variable will always be '$v1', while the others (i.e., the other inlets) can be either "f" or "i" values.

Complete the patch to resemble that shown in figure 4.2.

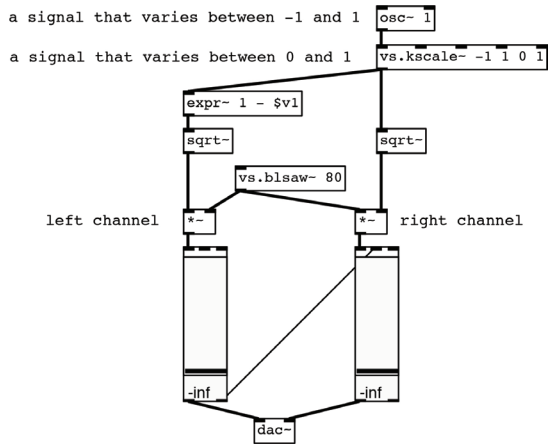


fig. 4.2: stereo panning controlled by an LFO

As we already know, the **[vs.kscale~]** object takes four arguments. The first two arguments define the input range, while the last two define the output range. In our case, the arguments [-1 1 0 1] indicate that if we send a signal that varies between -1 and 1 to **[vs.kscale~]**, the output signal will be rescaled between 0 and 1, which is exactly what we are looking for. The **[osc~]** object generates a control sine wave at a frequency of 1 Hz. So, it takes one second for the sound to go from the left channel to the right channel and back. If we connect a number box to **[osc~]**, we can adjust the frequency of the oscillation. Try using various frequencies, but do not exceed 20 Hz: in fact, higher frequencies will produce modulation phenomena, which are part of a topic we will discuss later.

At this point, you can simplify the patch by using the **[vs.pan~]** object from the *Virtual Sound* library, an object that implements a stereo panning algorithm; the object takes the sound to be positioned on its left inlet and positions it in the stereo field according to the control signal it takes on its right inlet (see fig. 4.3).

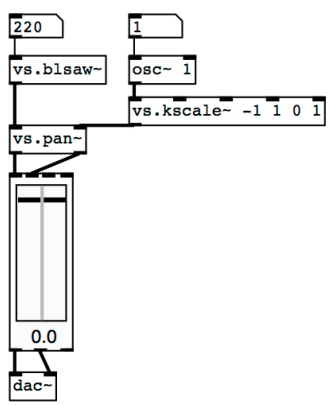


fig. 4.3: stereo panning using the **[vs.pan~]** object

You can see that the `[vs.pan~]` object performs the same function as the algorithm shown in figure 4.2. We are using it just to free up room in the graphical display of our patch. In this patch we also replaced the two `[vs.gain~]` objects with `[vs.sgain~]`, which unifies their behavior in a single object.
Try this patch, substituting control signals made with other waveforms, such as the square wave shown in figure 4.4.

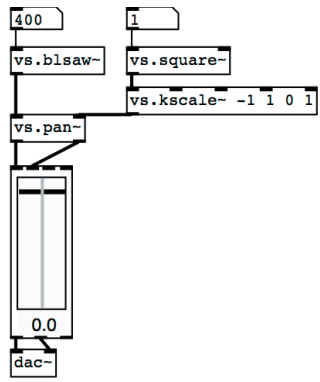


fig. 4.4: controlling panning with a square wave LFO

In this case, the sound moves from channel to channel without passing through intermediate positions. The sudden discontinuity, however, generates an undesirable click in the output signal. Fortunately, this can be eliminated by filtering the control signal with a low-pass filter (`[lop~]`), which 'smooths' the sharp corners of the square wave. (See Figure 4.5 for this modification.)

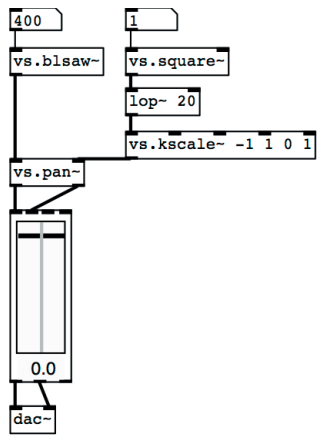


fig. 4.5: filtering an LFO

In this patch, we have set a cutoff frequency of 20 Hz, which means that the control signal cannot 'jump' from one value to the other faster than 20 times a second. Try changing the cutoff frequency for the filter to better understand how it influences the path of the sound; the lower the cutoff frequency, the smoother the transitions between channels will be.

(...)

4.2 DC OFFSET

4.3 CONTROL SIGNALS FOR FREQUENCY
Simulating a theremin

4.4 CONTROL SIGNALS FOR AMPLITUDE

4.5 VARYING THE DUTY CYCLE (PULSE-WIDTH MODULATION)

4.6 CONTROL SIGNALS FOR FILTERS

4.7 OTHER GENERATORS OF CONTROL SIGNALS

4.8 CONTROL SIGNALS: MULTICHANNEL PANNING

ACTIVITIES
- Analyzing algorithms
- Completing algorithms
- Correcting algorithms

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Pd native objects
- List of Virtual Sound Library objects
- Glossary

Francesco Bianchi • Alessandro Cipriani • Maurizio Giri

Pure Data: Electronic Music and Sound Design

Theory and Practice • volume 1

Topics

Sound Synthesis and Processing - Frequency, Amplitude and Waveform - Envelopes and Glissandi - Additive Synthesis and Vector Synthesis - Noise Generators - Filters - Subtractive Synthesis - Virtual Synthesizer Programming - Equalizers, Impulses and Resonant Bodies - Control Signals and LFOs - Pure Data Programming Techniques

This is the first in a series of volumes dedicated to digital synthesis and sound design. It is part of a structured teaching method incorporating a substantial amount of online supporting materials: hundreds of sound examples and interactive examples, programs written in Pure Data, as well as a library of Pd objects created especially for this book.

"Music making is a much freer and wider-ranging activity than it was before the advent of electronics, mediated as they are almost always today by computers. There are many possibilities out there to explore, and studying both the theory and the practice of audio programming is the best first step of the way"(...) "This particular stack of technologies has been crafted with the goals of this book in mind: primarily, as I see it, to open a path along which the reader (whether a student or an independent reader) can get a theoretical and practical understanding of the fundamental techniques for making music with a computer, to make this process as direct and straightforward as possible, and to require a minimum of specialized knowledge in advance."
(from the foreword by Miller Puckette, the original author of Max and Pure Data)

FRANCESCO BIANCHI is mainly involved with music composition and computer music. He composed various pieces of both vocal and instrumental electronic music, which have been performed in many music festivals and institutions of several countries (such as the Huddersfield University, the Leeds University, and the Music Conservatories of Rome, Turin, and Perugia). He won the third price at the contest "Valentino Bucchi" in 2008, thanks to his piece "Cercle". On the occasion of Expo 2015 he collaborated on a project on designing the sound for the data captured by a food testing device, developed at the University of Parma. He created several software libraries to extend the features of Pure Data and Max. Also, for these two software applications he is currently working on biosLib, which is a collection of objects that implement artificial life algorithms.

ALESSANDRO CIPRIANI co-authored "Virtual Sound", a textbook on Csound programming. His compositions have been performed at major festivals and electronic music venues, and released on CDs and DVDs issued by Computer Music Journal, International Computer Music Conference as well as others. He has written music for the Peking Opera Theater, as well as for films and documentaries in which ambient sound, dialog, and music all fuse together, interchangeably. He is a tenured professor in electronic music at the Conservatory of Frosinone, a founding member of the Edison Studio in Rome, and a member of the editorial board of the journal Organised Sound (published by Cambridge Music Press). He has given seminars at many European and American universities, including the University of California - Santa Barbara, Sibelius Academy in Helsinki, and Accademia di S. Cecilia in Rome.

MAURIZIO GIRI is a professor of composition as well as a teacher of Max programming techniques at the conservatories of Latina and Frosinone. He is an instrumental and electroacoustic composer of music, specializing in digital sound processing, improvisation and computer-assisted composition. He has written computer applications for algorithmic composition and live performance, and has published numerous tutorials on Max. He founded Amazing Noises, a software house that develops music applications and plug-ins for mobile devices and computers.

ISBN 978-88-99212-21-6

www.contemponet.com

€ 42.00 vat included