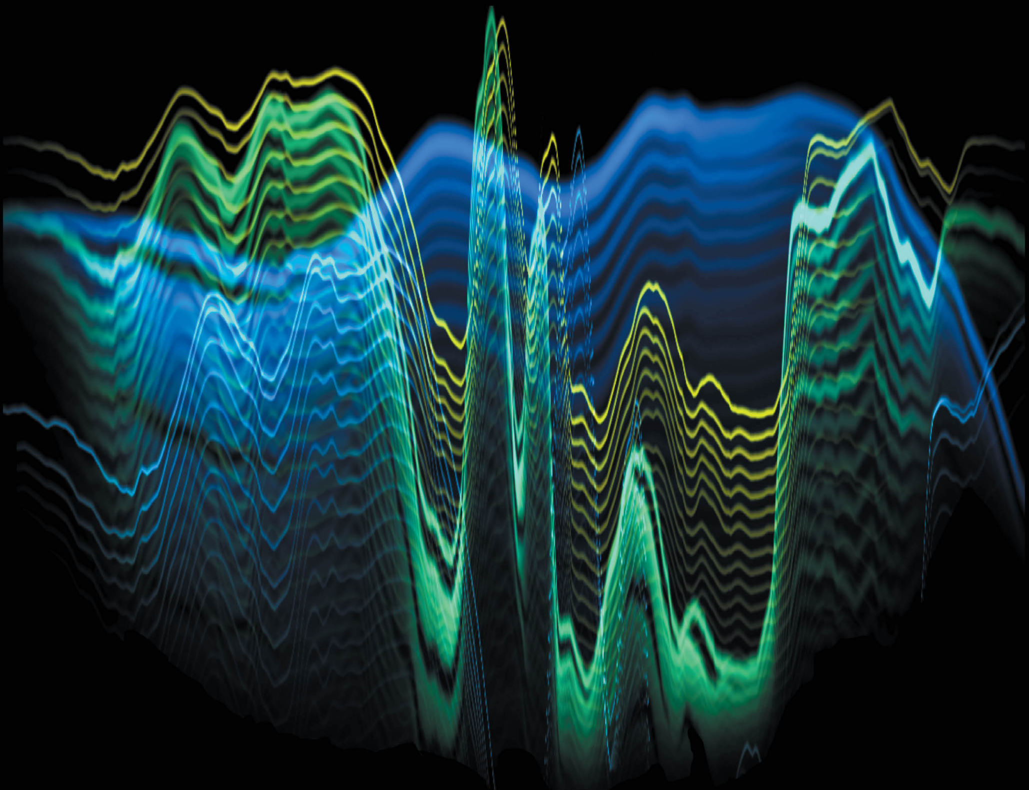
Alessandro Cipriani • Maurizio Giri

Electronic Music and Sound Design

Theory and Practice with **Max 7** • volume 1



ConTempoNet

Alessandro Cipriani • Maurizio Giri

ELECTRONIC MUSIC AND SOUND DESIGN

Theory and Practice with Max 7 - Vol. 1

Cipriani, Alessandro. Giri, Maurizio.
Electronic Music and Sound Design : theory and practice with Max 7. Vol. 1. / Alessandro Cipriani, Maurizio Giri.
Includes bibliographical references and index.
ISBN 978-88-992120-2-5
1. Computer Music - Instruction and study. 2. Computer composition.

Original Title: Musica Elettronica e Sound Design - Teoria e Pratica con Max 7
Copyright © 2009 Contemponet s.a.s. Rome - Italy

Translation by David Stutz

Copyright © 2010 - 2016 - ConTempoNet s.a.s., Rome - Italy
First edition 2010
Second edition 2013
Third edition 2016

Figures produced by: Gabriele Cappellani and Maurizio Refice
Interactive Examples: Francesco Rosati
Index: Salvatore Mudanò
Language education consultant: Damiano De Paola
Cover image: Federico Foderaro

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 David Zicarelli

It might seem odd to you, but many years ago, I spent a lot of time learning about making sound with a computer by reading books and articles while trying to imagine what the synthesis techniques being described would actually sound like. While I suppose my imagination might have been stimulated by this practice, I am happy that real-time synthesis has progressed to the point where you no longer have to be deprived of the perceptual experience that is such an important part of learning the techniques of digital synthesis.

Alessandro Cipriani and Maurizio Giri's book is one of the first courses on electronic sound that explicitly integrates perception, theory, and practice using examples of real-time sound synthesis you can manipulate and experience for yourself. In my view, the manipulation aspect of learning about sound is critically important. It helps lead you to what Joel Chadabe terms "predictive knowledge" -- the ability to intuit what will happen to a sound before you take an action to change it. We all have some level of predictive knowledge. For example, most of us know that by turning a volume knob clockwise, the sound coming from our amplifier will get louder. Once we enter the realm of digital sound synthesis, things quickly get more complicated than a volume knob, and we need the first-hand experience of manipulation and perception in order to deepen our predictive knowledge.

However, to educate ourselves fully about digitally produced sound, we need more than predictive knowledge. We need to know why our manipulations make the perceptual changes we experience. This theoretical knowledge reinforces our intuitive experiential knowledge, and at the same time, our experience gives perceptual meaning to theoretical explanations.

In my opinion, Cipriani and Giri have done a masterful job of allowing experiential and theoretical knowledge to reinforce each other. This book will work either as a textbook or as a vehicle for the independent learner. As a bonus, the book includes a thorough introduction to digital signal processing with Max and serves as a wonderful introduction to the programming concepts in that software.

As you will see, the theoretical chapters are the "T" chapters, while practical and experiential knowledge is imparted by the "P" chapters. These chapters alternate, in the form of a ladder, refining the concepts at ever higher levels of sophistication.

I hope you will take advantage of the excellent Max examples the authors have created. They are simultaneously fun and enlightening, and they sound good enough to use on stage. They are also worth examining as models for your own Max patches, or for extending in new ways. But a few minutes of messing around with the examples is not the same thing as studying the concepts in the book. The book provides the language for expressing the concepts in terms of the underlying theory. Knowing the theory is essential, because presumably you're reading this book because you want to be more than someone who can turn a volume knob.

That is the authors' wish for you, and mine as well. I want to wish you good luck on this new adventure, and also thank my two Italian friends for creating such a comprehensive resource for learning about digital music – the one I wish existed when I was a student!

David Zicarelli, publisher of Max

INTRODUCTION

This is the first of a series of three volumes dedicated to digital synthesis and sound design. The second volume 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, OSC and realtime synthesis. The third volume will be concerned with non-linear techniques (such as AM and FM synthesis), granular synthesis, analysis and resynthesis, convolution, physical modeling, micromontage, 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. 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), 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 Max program, interacting with example material, and constructing algorithms.

MAX

The practical parts of this book are based on the software Max. This program, written originally by Miller Puckette, was extensively revised and expanded by David Zicarelli, and is published as a supported product by Cycling '74 (www.cycling74.com). Max is an interactive graphic environment for music, audio processing, and multimedia. It is used throughout the world by musicians, composers, sound designers, visual artists, and multimedia artists, and it has become a *de facto* standard for modern technologically-enabled creative projects in both the musical and in the visual spheres.

It is a graphic programming language, and is therefore *relatively* easy to learn, especially given its great power and expressivity. In Max one creates programs by connecting onscreen graphic objects with virtual cables. These objects can perform calculations, produce or process sounds, render visuals, or be configured as a graphical user interface. Using its sound synthesis and signal processing capabilities, one can fashion soft-synths, samplers, reverbs, signal-processing effects, and many other things.
In practice, Max adopts the metaphor of the modular synthesizer: each module

handles a particular function, exchanging data with the modules to which it is connected. The difference between a traditional modular synthesizer and Max is that with Max, one can access and control a level of detail that would be inconceivable in a preconfigured synthesizer or extension module (whether hardware or software).

PRACTICAL INFORMATION

Many indispensable materials accompany this book, among them, interactive examples, patches (programs written in Max), sound files, programming libraries, and other materials.
These can be found at the book web site.

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. 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 Max version 5 or higher, which is downloadable from the official Cycling '74 website, www.cycling74.com.

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 Max 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 Max

Learning Max (and, in general, learning synthesis and sound processing) requires effort and concentration. In contrast to much commercial music software, Max provides flexibility to the programmer, and this design choice provides those programming with Max many alternative ways to build a given algorithm. To benefit from this freedom, however, it is advisable to consider the recommendations of the book and to code in a systematic way. Max 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 Max, which is available at the Cycling74 website: www.cycling74.com. The same web page contains detailed instructions regarding how to install Max and the macro library correctly; look for the document entitled "How to Install and Configure Max". Always check the support page for patches (Max 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: a.cipriani@edisonstudio.it and maurizio@giri.it
http://www.edisonstudio.it/alessandro-cipriani
http://www.giri.it

THANKS
We wish to thank Gabriele Cappellani, Salvatore Mudanò and Francesco "Franz" Rosati for their patience and long hours of work, and Dario Amoroso, Joel Chadabe, Mirko Ettore D'Agostino, Luca De Siena, Eugenio Giordani, Gabriele Paolozzi, Giuseppe Emanuele Rapisarda, Fausto Sebastiani, Alvise Vidolin and David Zicarelli for their generosity.

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.

Alessandro Cipriani and Maurizio Giri

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 (Max for example) will *execute* our instructions to create a *stream of digital data* in which all of the characteristics of the sound or sounds

[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.

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).

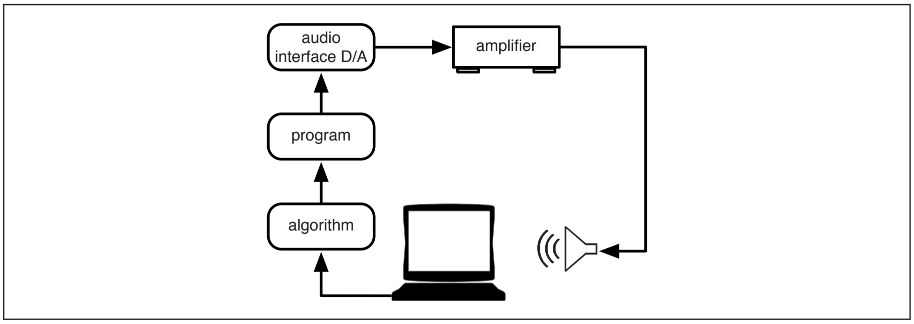


Fig. 1.1 Realtime synthesis

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.)

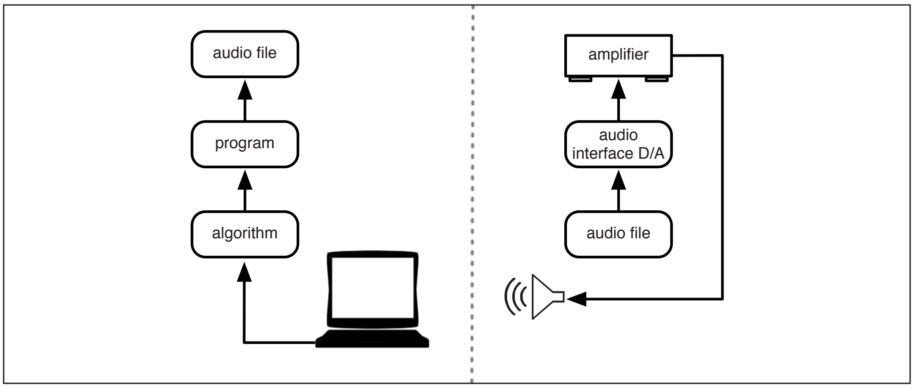


Fig. 1.2 Non-realtime synthesis and listening as separate actions

[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).

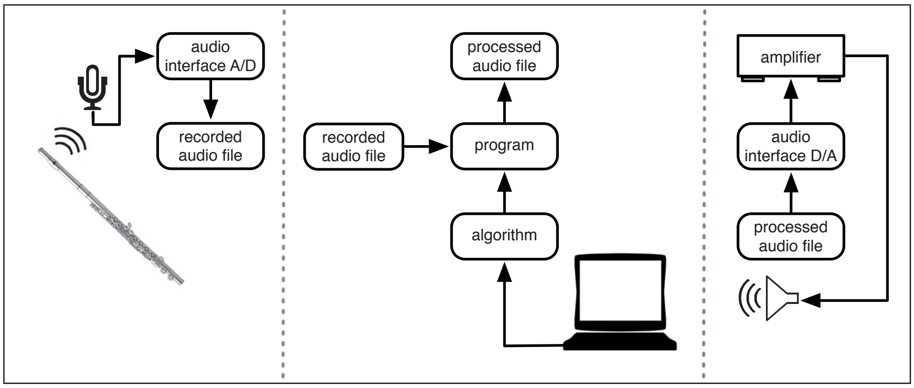


Fig. 1.3 Example of offline sound processing

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.

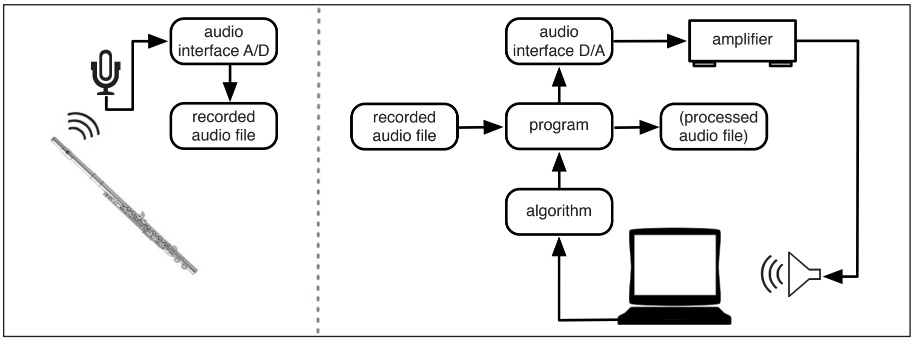


Fig. 1.4 Example of realtime sound processing on pre-existing sound

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

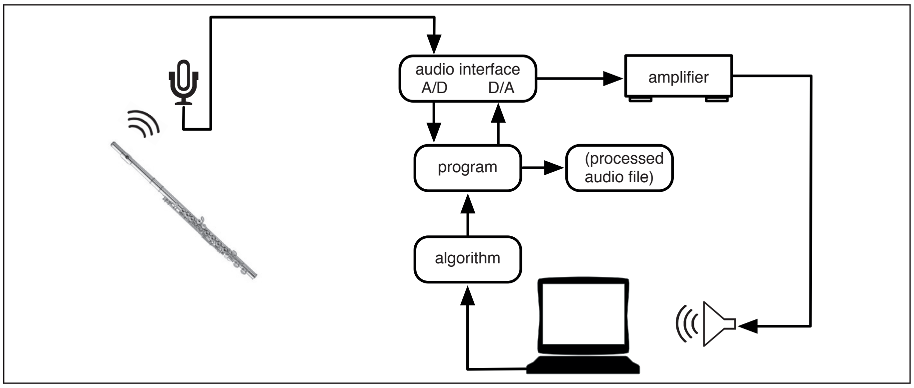


Fig. 1.5 Example of realtime sound processing on live sound

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., 2003) 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

[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).

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

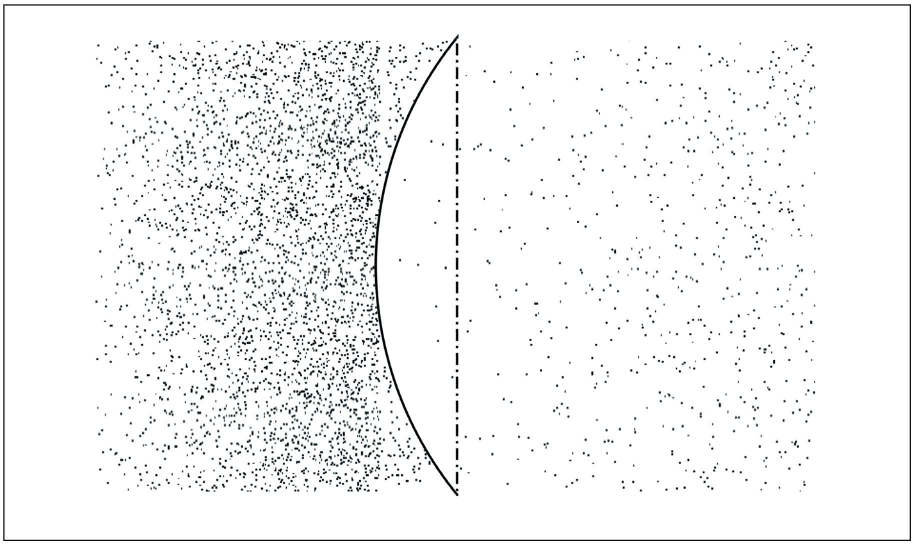


Fig. 1.6 Vibration of a string

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).

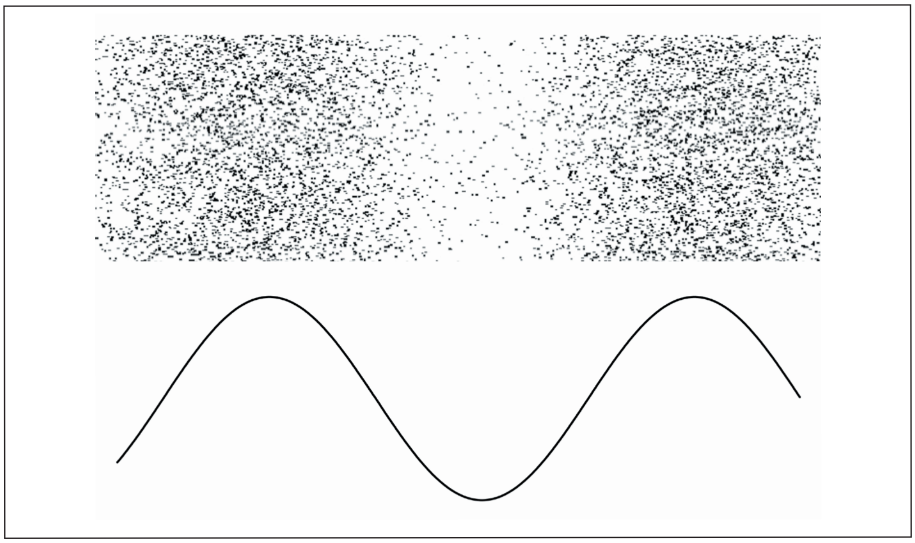


Fig.1.7 Compression and rarefaction of air molecules

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.

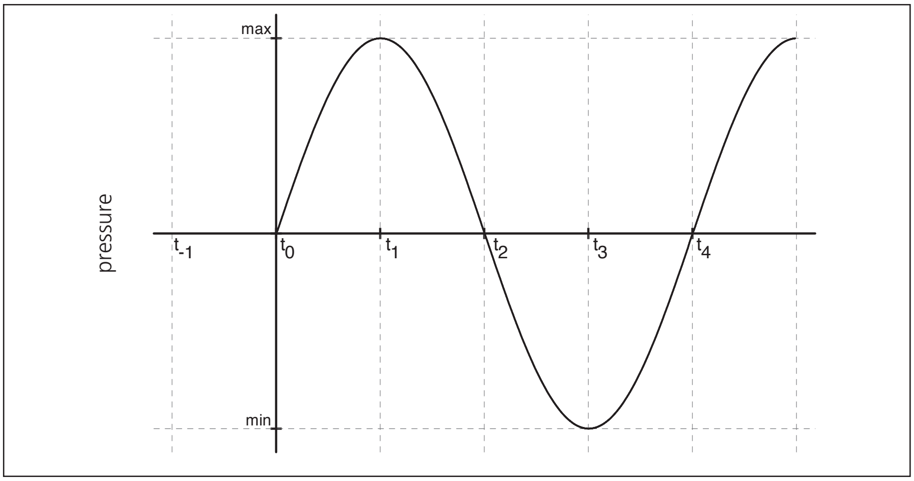


Fig.1.8 A graphical representation of compression and rarefaction

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 web site.

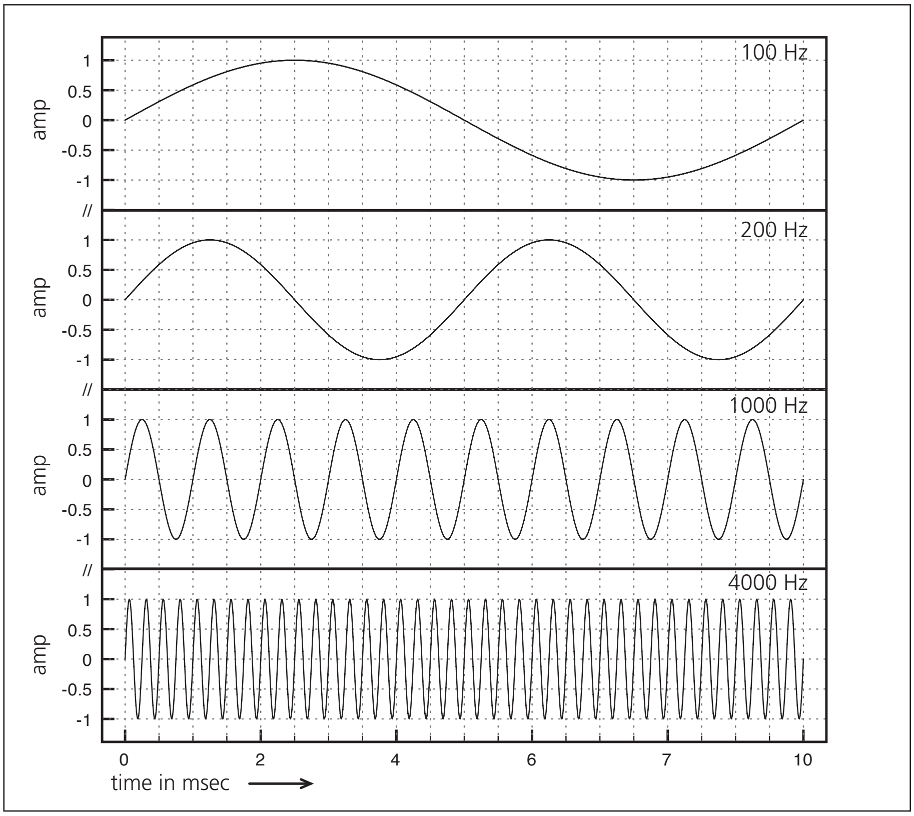


Fig.1.9 Four sounds of different frequencies

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).

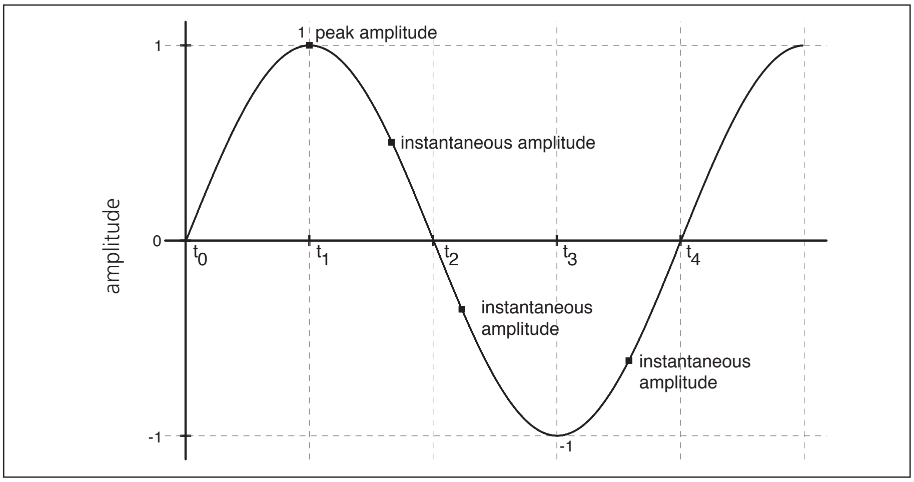


Fig.1.10 Amplitude of a sound

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 numbers 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.

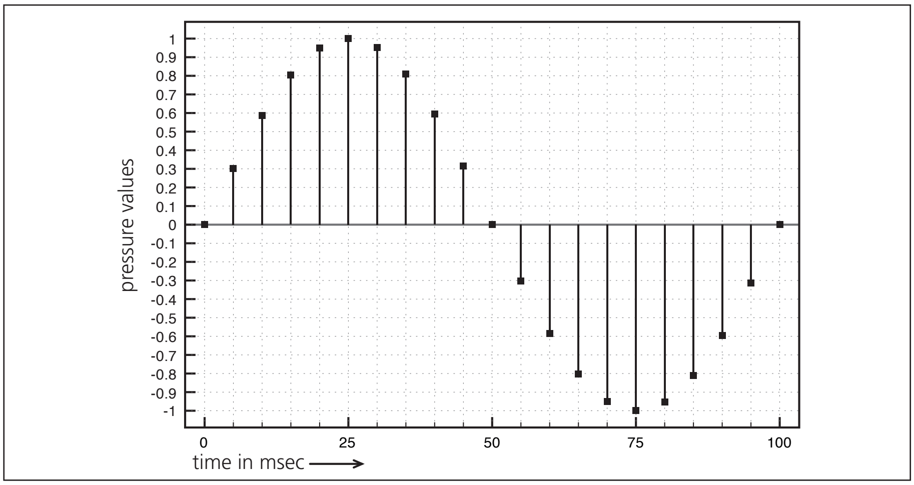


Fig. 1.11 Digital representation of a waveform

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.)

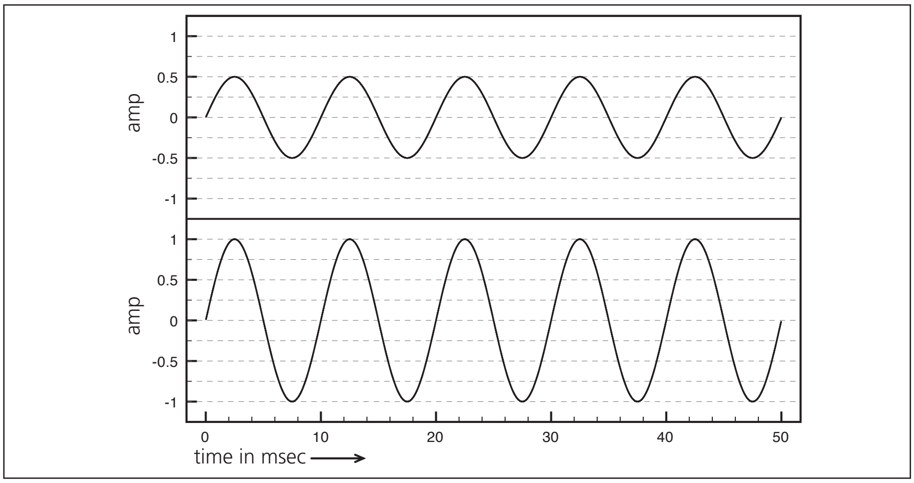


Fig.1.12 Two sounds with differing amplitudes

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).

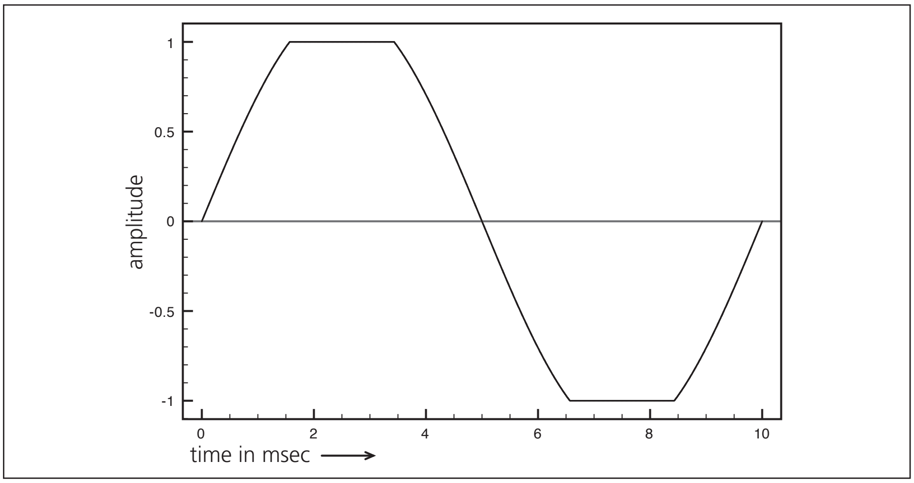


Fig.1.13 A "clipped" waveform

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 to 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, FS 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 at 1 meter of distance). 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.

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*

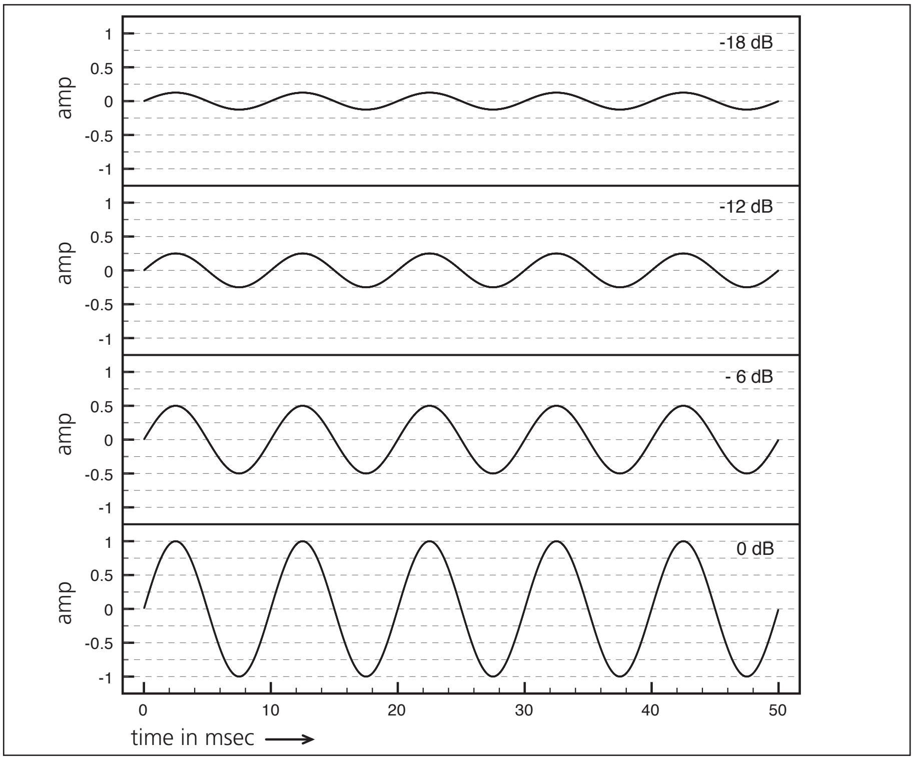


Fig.1.14 Four sounds with their amplitudes progressively doubled

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**[12] and indicate, within the audible frequency range, the sound pressure needed to produce equal impressions of loudness for a listener.[13]

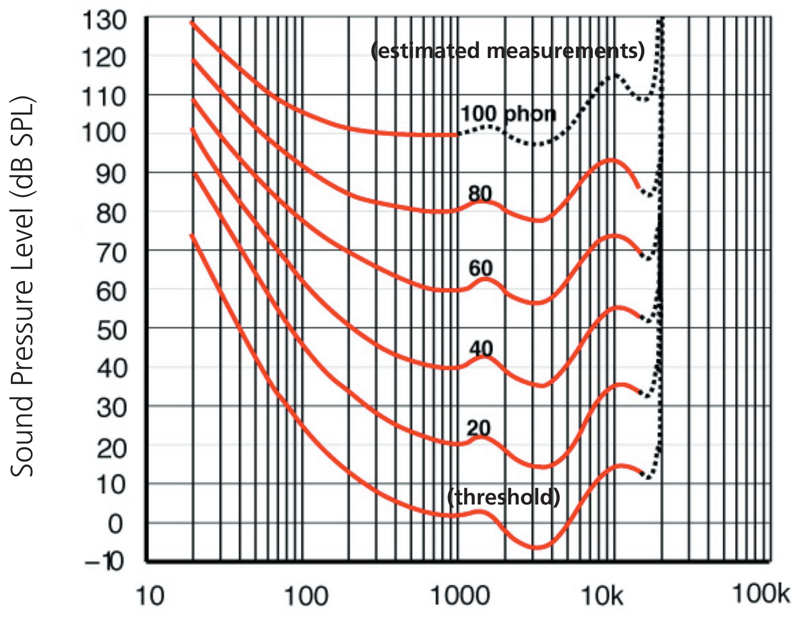


Fig. 1.15 Diagram of equal loudness contours (ISO 226:2003)

[12] 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.
[13] 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:

1P
SOUND SYNTHESIS WITH MAX

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Basic computer knowledge
  (operating a computer, managing files and folders audio formats, 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 then progressing to the corresponding chapter on practical Max techniques.)

LEARNING OBJECTIVES

Skills

- To be able to use all of the basic functions of Max
- To be able to synthesize both sequenced and overlapping sounds using sine wave oscillators, as well as square wave, triangle wave, and sawtooth wave oscillators
- To be able to continuously control the amplitude, frequency, and stereo spatialization of a sound (using linear and exponential envelopes for glissandi, amplitude envelopes, and the placement of sound in a stereo image)
- To be able to generate random sequences of synthesized sound
- To be able to work with and manage elements of sampled sound

Competence

- To successfully realize a first sound study 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 intensity 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 Max 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 Max commands
- List of Max objects used in this chapter
- Messages, attributes, and parameters for specific Max objects
- Glossary of terms used in this chapter

1.1 FIRST STEPS WITH MAX

To proceed with this chapter you will need to have the Max application correctly installed on your computer. If you haven't done this, you can download it from the Cycling74 website at http://www.cycling.74.com. It will also be necessary for you to download the support materials for this book, which can be found on the support web page. We especially recommend that you install the *Virtual Sound Macros library*; to do this, follow the installation instructions found at the same website.

Launch Max, and select the New Patcher entry from the *File* menu (or press Command-n on the Mac or Control-n on Windows).[1] The **Patcher Window** will appear. The Patcher window is where you connect Max objects to create algorithms. Before you proceed, notice that the Patcher Window is framed by a dark border that contains a number of icons (see fig. 1.1). The four sides of this border are named *Patcher Window Toolbars*; we will explain each of the icons found in them gradually, as the need arises.

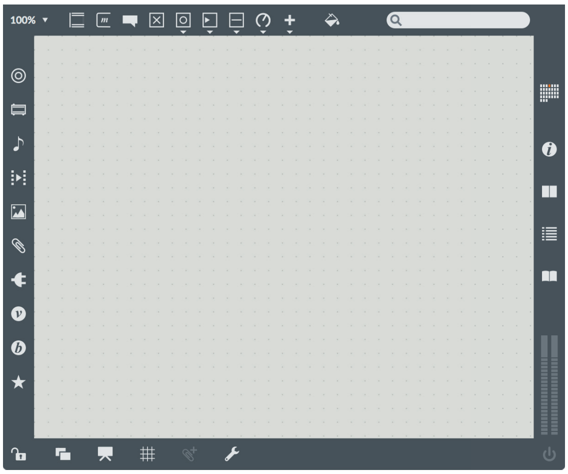


Fig.1.1 The *Patcher Window*

We will begin with the upper *Toolbar*, whose icons represent Max objects that you can use to construct "virtual machines" that embody algorithms for sound synthesis and signal processing. Max objects can be interconnected with each other: information flows (in the form of numbers, digital signals, or other kinds of data) from one object to the next through these connections. Each object is designed to perform a specific operation on the information that it receives, and each object will pass its results to any other objects that are connected to it.

[1] On Mac OSX, hold the Command key (⌘) down and press 'n', or on Windows, press 'n' while holding the Control key down. This will be abbreviated as <Command-n/Control-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.

A collection of objects 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**).
Let's create our first patch. If you click on the first icon on the left in the upper *Toolbar*, you will cause a new object to appear in the Patcher Window (as seen in figure 1.2).

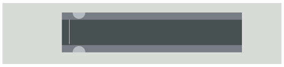


Fig.1.2 The object box

The object that appears is a generic object called the **object box**. It is the object that we will use most often when patching. Its function depends upon the name that we give it when we type a *string*[2] into its interior.
We want to turn this object box into a sine wave oscillator; in order to accomplish this, you will need to type the word "cycle~" into the object box's interior. Try this now, and notice that as you begin to type, a menu appears that lists all of the objects whose names or whose description contain the characters which you have typed. This very useful feature, shown in figure 1.3, is called **autocompletion**.

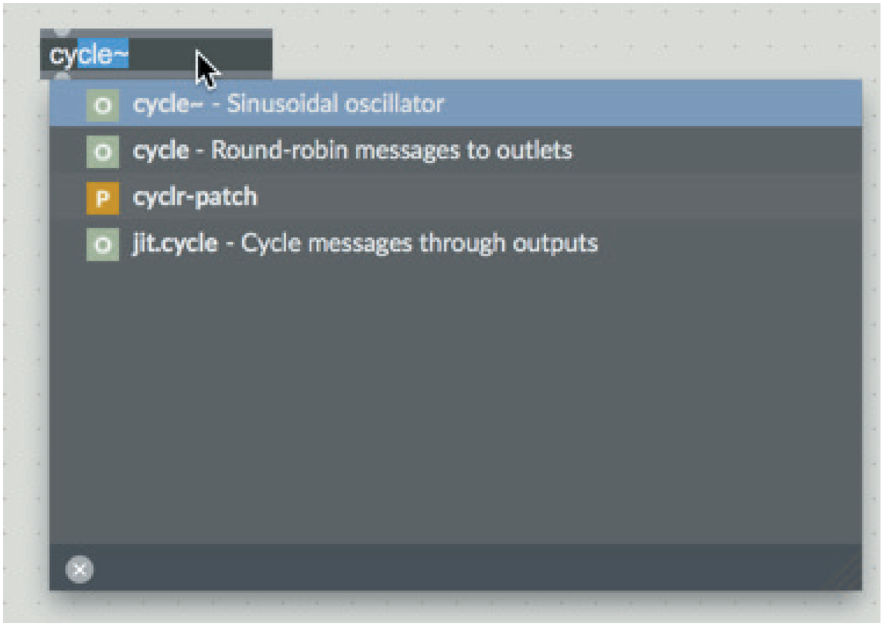

cycle~
O cycle~ - Sinusoidal oscillator
O cycle - Round-robin messages to outlets
P cyclr-patch
O jit.cycle - Cycle messages through outputs

fig.1.3: the autocompletion menu

[2] The word "string" is computer programming lingo that refers to a combination of letters and numbers, such as "print", "save", "astringcontainingthenumber1", or "subsection22".

In the figure, you can see what the *autocompletion* menu looks like after typing the first two characters of "cycle~" (the 'c' and the 'y'). The stylized letter that precedes each *autocompletion* menu option specifies the type of the individual elements being suggested. In the figure, for example, we can see two letters: "o" which indicates an object and "p" which indicates an *abstraction*. (We will speak later about what *abstractions* are, and also about other types of elements that can be found within a Max patch.) Inside the object box itself, immediately following the characters that you have typed, you can see a proposed completion for the object that Max deems most likely to be used (and which is most often a name that you have typed before).

At this point, you can select the object name that you are looking for by clicking on its menu item, or by simply typing the rest of the name; for the purposes of this tutorial, make sure that you select "cycle~" and not "cycle"![3]
With "cycle~" in place, now type a space in order to see how the autocompletion menu changes depending on context. Two new categories, "Arguments" and "Attributes", appear. Without going into too much detail, the elements that populate these categories represent words that can optionally be typed in after the name of the object. Ignoring these placeholder suggestions for now, continue by typing the number 440 after the space that you have typed. (The space is very important!) After you have done this, click in an empty part of the Patcher Window[4], and the object box will now resemble figure 1.4.

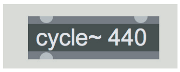


Fig.1.4 The `cycle~` object

The little lighter zones that appear on the upper and lower edges of the object are the **inlets** and **outlet** for `cycle~`, respectively, which we will examine shortly. (NB: If the object doesn't display these, then you have a problem. Look for debugging tips in the FAQ at the end of this section.)

[3] Note the character '~', which is called a **tilde**, that follows the word `cycle`. 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 Max. Since some external objects do not appear in the auto completion menu, you will sometimes need to type (or want to, because it is quicker!) object names directly into the object box. It will be important that you know where to find the tilde on your keyboard, especially if you have a non-English one! .
[4] Or alternatively, press <Enter> on the Mac or <Shift-Enter> on Windows.

Now we will create another object, **gain~**, which has an appearance, shown in figure 1.5, that is similar to the faders found on a mixing console.
By clicking on the seventh icon of the upper *Toolbar*, you will cause a *palette* to appear that contains all of the objects in the "Slider" category. Click on the third icon of the *palette*, or alternatively, drag this icon [directly into the Patcher Window using the mouse.]

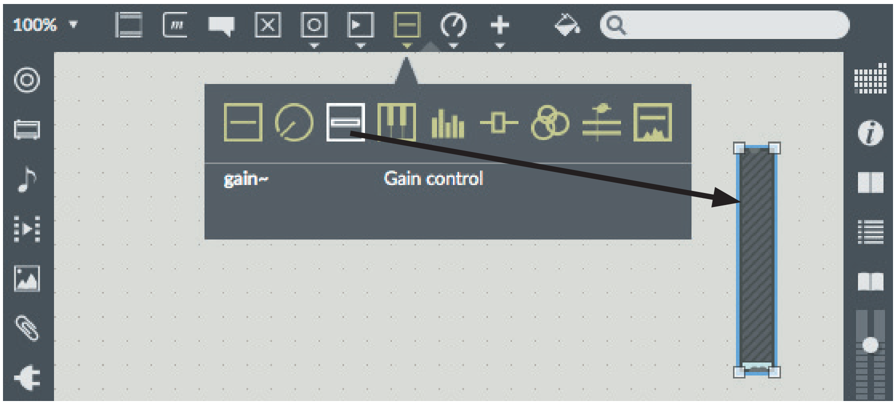


Fig.1.5 The gain~ object

When you click on the icon, a new kind of object will appear: instead of an object box, a **new user interface object** (also known as a **ui object**) will be created. Here's a little tip: if you have a hard time finding an object in the *Top Patcher Window Toolbar*, just create an object box (the same generic object that we just used to create cycle~), and then type the name of the object that you want to create (such as gain~) into its interior. Once you've done this, click outside of the object, and the object box will immediately morph into the ui object whose name you typed.
Drag the newly created gain~ object to a spot in the Patcher Window underneath the cycle~ object, and connect the outlet of cycle~ to the inlet of gain~ in the following manner: first, bring the mouse pointer near the outlet on the lower edge of the cycle~ object. A yellow circle, along with a box containing descriptive text (a feature officially called "*Assistance Bubble*" by Max), should appear. The circle, shown in figure 1.6a, indicates that the outlet is selected, and by clicking and dragging the mouse towards the gain~ object, a yellow-and-black striped patch cord can be made to appear. If you drag the end of this patch cord towards the inlet of the gain~ object (the small dark patch on its upper edge), you will see a red circle (and more descriptive text) appear, as shown in figure 1.6b. This second circle indicates that you have located an inlet that is compatible – at this point, release the mouse button, and you will have connected the two objects!

a

b

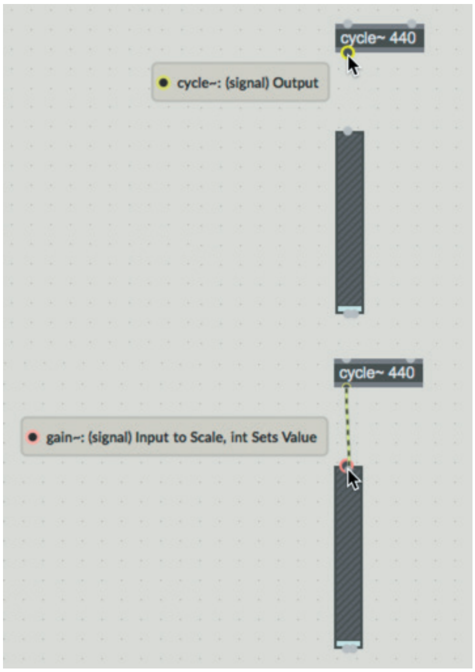


Fig. 1.6 Connecting two objects

We have connected the oscillator to the `gain~` object, which will serve to regulate the volume. We now must route the audio signal to the output: click on the next-to-last icon in the upper *Toolbar* to cause a six-category *palette* to appear[5], select the "Audio" category within this *palette*, and then click on the second icon within this category (or else drag it into the *Patcher Window*, as shown in fig. 1.7). The name of the object thus obtained is **`ezdac~`**.

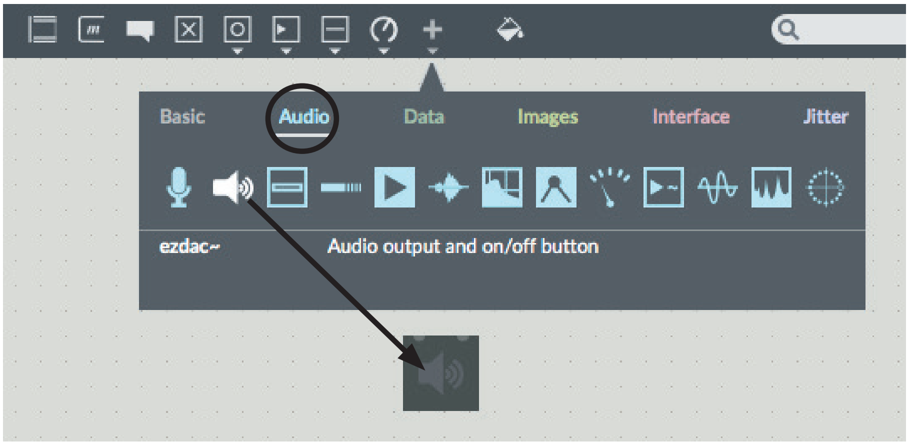


Fig.1.7 The `ezdac~` object

[5] A bit further in this same paragraph, we will better see how the upper *Toolbar* is organized.

We place the **ezdac~** under the **gain~** object and connect its left outlet to the two inlets of the **ezdac~** (as shown in figure 1.8).

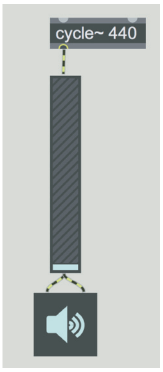


Fig.1.8 Our first patch

Watch out! The **gain~** object has two outlets, which are poorly distinguished from each other. Because of this, you should verify that you've used the left outlet for making both connections. The best way to ascertain that you've used the correct outlet is to read the *Assistance Bubble* text that appears when you make the connection, and to make sure that it contains the text *gain~: (signal) Scaled Output*. If one of the two cables were to be grey rather than yellow-and-black striped as they appear in the figure above, this would indicate the you've mistakenly used the right outlet, in which case you would need to cancel the patch cord by selecting the cord with a click (causing it to appear fatter) and pressing the <Delete> key (which is the same key that you press to delete text). The cord would disappear, and you could then reconnect the objects, using the correct inlet/outlet pair.
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 Max object! For example, don't call this patch **cycle~** (or **cycle** without the tilde, for that matter!) Doing this would be a recipe for confusing Max, and for causing unexpected results the first time that you tried to reload your patch. Given that it is impossible to remember all of the names for Max 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 more than one word: "test oscillator", for example, or "cycle~ object test", or any other combination. No Max object possesses a name composed of more than one word.

Don't forget this advice! A large percentage of the problems encountered by Max beginners are related to saving files that share a name with some internal object. We will return to this topic and explain the logic behind it in the interlude that follows this chapter.

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 **performance mode**, where we will be able to hear and test our patch. To do this, click on the small padlock that appears at the bottom left of the Patcher Window, or else press <Command-e/Control-e>.[6] Once we are in performance mode, the padlock at the bottom left will appear to be locked. (If it appears to be open, you are still in edit mode!)
Having switched to performance mode, click on the `ezdac~` object (the small loudspeaker), and then slowly raise the level of the `gain~` slider. You should hear a sound, pitched at A above middle C. By clicking repeatedly on the `ezdac~` loudspeaker, it is possible to alternately disable and enable sound output for the patch. (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 it work, let's revisit what we did:
The `cycle~` 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[7] should repeat itself 440 times per second.[8]
In more formal terms, `cycle~` was the name of the object, and 440 was the **argument**; a value used by the object to specify its operation. In this case, the argument 440 caused `cycle~` to produce a tone at 440 Hz.
The signal outlet of the `cycle~` object was connected to the inlet of the `gain~` object, which caused the signal generated by `cycle~` to be passed on to the `gain~` object, where it could be modified, as we have seen, by moving the volume fader. The modified signal was then passed on to the `ezdac~` object (the little loudspeaker), which attenuated the signal and routed it to the sound driver,[9] which managed a digital-to-analog conversion that transformed the numeric representation of the sound into an audio wave that you could hear on your headphones or speakers. (This conversion, by the way, is where `ezdac~` got its name; it is a quasi-acronym for EaZy 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 "Max Chapter Materials Vol 1" archive that can be found on the support web page, do this now.

[6] Alternatively, you can move back and forth between edit and performance modes by holding down the <Command> key (on the Mac) or the <Control> key (on Windows) and clicking the left mouse button on an empty part of the Patcher Window.
[7] In this case, it is actually a cosine wave, as we will see in the next chapter.
[8] All of these concepts were laid out in Theory Section 1.2.
[9] A sound driver is a piece of system software that enables programs such as Max to manage incoming as well as outbound audio signals.

Open the file **01_01.maxpat**, which you will find in the "*Max Chapter Materials Vol 1/Max Patches Vol 1/Chapter 01 Patch*" folder (see figure 1.9).

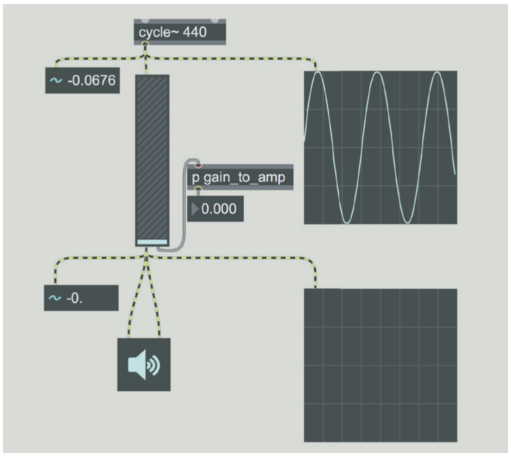


Fig.1.9 The file 01_01.maxpat

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 **`number~`** objects, and they show, via the value that they display, a snapshot of the signal that they are receiving. The larger rectangular objects on the right are called **`scope~`** objects,[10] which act as oscilloscopes on which it is possible to view an incoming signal as graphic waveform. The object [`p` gain_to_amp] and the object that is connected to its outlet (which is called a `flonum` or a float number box) are used to view exactly how much `gain~` is being applied to amplify or attenuate the signal.

Once again, start the patch by clicking on the `ezdac~`, and observe the numbers changing in the `number~` object high on the left. These numbers are being produced by the `cycle~` object and, if we observe them for a bit, we can see that their values, both positive and negative, fall between 1 and -1. On the upper right side we can see the `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 `scope~` panel, hundreds of different elements are shown, visualized as a sequence of points, rather than the single number shown by the `number~` object. The points fall very close to each other on the screen, and so they appear as a solid curve. The elements that they represent, the numbers themselves, are called *samples* in the terminology of digital music. And the line made from these sample values, as they undulate high and low across the oscilloscope panel, is precisely the sinusoidal wave produced by the `cycle~` object.

[10] The `number~` and `scope~` objects can be found in the upper *Toolbar* in the previously described "Audio" category, which you will remember can be accessed by clicking on the next-to-last icon and selecting "Audio". Also remember that if you can't find an object here, you can always use the trick that we demonstrated above: take an object box and type the name of the desired object into its interior.

The patch also contains a second **`number~`** and a second **`scope~`**, each connected to the **`gain~`** object. These objects should be displaying the number 0 and a flat line (which is, of course, a sequence of zeros), because the volume fader is at its lowest setting, resulting in a volume of 0. If we move the **`gain~`** fader upwards, we should see **`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 **`scope~`** should begin its undulation and assume the same look as the other **`scope~`**. We can infer from this that **`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 **`gain~`** fader to be close to its maximum setting, we see **`number~`** begin to exceed the amplitude limits of 1 and -1, and the waveform on the oscilloscope becomes clipped. More important than these visual clues, you should be able to actually *hear* the sound change, as it becomes distorted.

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

1) The **`cycle~`** object produces a sequence of digital values that follow the pattern of a (co)sine wave.
2) The numerical limits for samples in this sine wave are 1 and -1. The actual sequence that these values follow can be seen on the upper **`scope~`**, which shows the waveform at its maximum amplitude, above which the quality of the sound would be distorted.
3) The **`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. If, for example, we move it to 0.5, the amplitudes of the samples that enter the **`gain~`** object are diminished (because multiplying a number by 0.5 is the same as dividing by 2).[11]

As we move it to 1.0 (which is about 3/4 of the way up the fader), the sample values entering the object are identical to those leaving. Finally, if we raise the fader all of the way, 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

[11] To adjust the fader to a position that corresponds to a multiplication factor of 0.5, watch the number box that is connected to the [`p` gain_to_amp] object, which is set up precisely for the purpose of displaying the value of the multiplication factor. The fader is actually divided into logarithmic increments, and using a formula that we'll explain later, the [`p` gain_to_amp] object converts these fader positions (which can be read from one of the outlets of the `gain~` object) into an amplitude level. We will return to this topic in Interlude A, which follows this chapter. One thing to note is that when the multiplication factor is near 0.5, the sine wave takes up about half of the oscilloscope screen.

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 Max, 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 `cycle~`.

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 Max. 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.

Question: I created an object called "cycle~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 "cycle~" 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, Max will search for a non-existent object named "cycle~440", and when nothing is found, Max will have no inlet or outlet information with which to work.

Q: Very good. Why, then, didn't Max give me an error message?

A: There *is* an error message, which can be found in the **Max Console**, which is a window that the program uses to communicate with you. To call it from the keyboard, use <Command-m / Control-m>, or click on the fourth icon of the right *Toolbar*: a window will appear on the right of the Patcher Window. In the window, you will probably find a message such as "*cycle~440: No such object*". If you double-click on the error message, the object that caused it (in this case, the bogus "cycle~440" object) will be highlighted in the Patcher Window.

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

A: There is a more subtle 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 Macs), 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 Max, and Max 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 clicked on the **ezdac~** object (which is the loudspeaker icon)? 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 Max? Have you checked the Audio Status window (which you can find in the *Options* menu), to see that the correct sound driver is selected?

A COMPACT "SURVIVAL MANUAL" FOR MAX

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

BASIC KEYBOARD COMMANDS
Mac <Command-n> or Windows <Control-n> will create a new Patcher Window, which is the workspace for implementing patches.

Mac <Command-e> or Windows <Control-e> will toggle between edit mode and performance mode in the Patcher Window. In edit mode, we assemble patches by creating objects using the Top Patcher Window Toolbar; in performance mode we can activate patches and interact with their graphical user interface objects (such as float number boxes or the **gain~** object).

Mac <Command-m> or Windows <Control-m> will call up the Max Console if it is not already visible. The Max Console is a window used by the Max environment to communicate, using brief text messages, with its user. We will learn more about this window shortly.

Instead of using the Patcher Window, it is possible to create some common objects by typing simple one-character commands, without having to use <Command> or <Control> keys. The character 'n', for example, will create (when your Patcher is in edit mode) an empty object box at the position of the mouse pointer. There are other keys that also enable object creation: in edit mode, try positioning the mouse pointer and pressing 'f', 'i', 't', and 'b', and observe the various objects (which probably mean nothing to you at the moment!) that are created. We will use these objects frequently in the course of the following chapters.

SELECTION, DELETION, AND COPYING

To delete a patch cord or an object, you need to insure 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 be referred to as the <Backspace> key). We can select multiple objects at a time by clicking on a blank spot in the Patcher Window and dragging a rectangle that "touches" the objects to be selected (see fig 1.10).

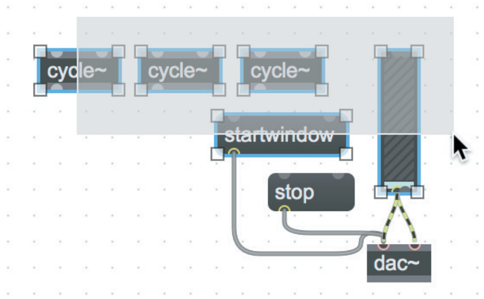


Fig. 1.10 Selecting objects

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 will vanish. Using this selection technique, objects will be selected but not patch cords. If we need to select the patch cords at the same time we are selecting objects (for example, to delete them) we need to hold the <Alt> key down while dragging the selection rectangle, and make sure that we "touch" the cables that interest us (as seen in figure 1.11).

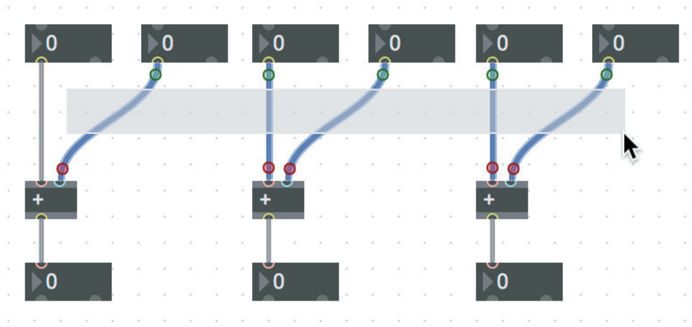


Fig. 1.11 Selecting patch cords

[12] Check to make sure the padlock at the bottom left of the window containing the patch is open.

With the <Alt> key held, we can also copy an object. Simply click-and-drag to tear off a copy. If you first select multiple objects and then drag them using <Alt-Click>, you can copy them all (as in figure 1.12).

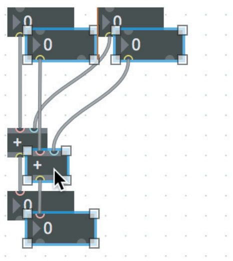


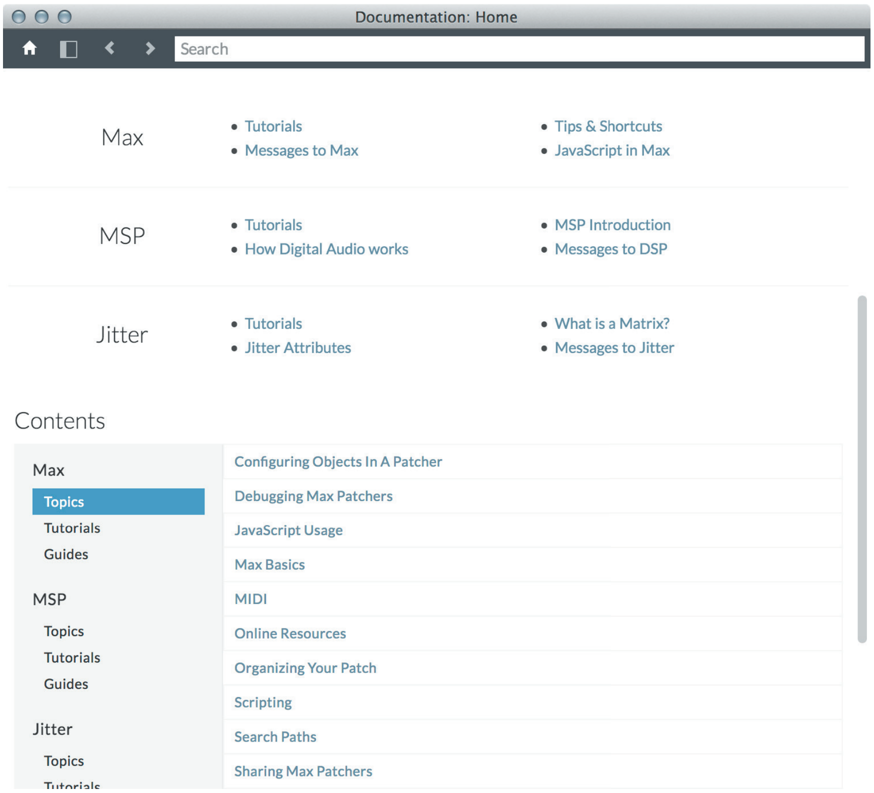
Fig. 1.12 Copying multiple objects at once

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. Repeatedly selecting Undo can cancel an entire sequence of commands and return the patch to the state that preceded those actions. If, after undoing one or more actions, you decide that there wasn't actually an error (for example, you wanted to delete the object after all), you can restore actions by using the Redo command, which is also found in the Edit menu.
The command equivalents of undo and redo are <Command-z> and <Shift-Command-z> on the Mac and <Control-z> and <Shift-Control-z> on Windows.[13]

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 Max synthesis and signal processing techniques. But the Max environment also provides extensive illustrative materials that can be accessed by selecting the **Max Documentation** item from the ***Help*** menu.

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

When you select this item, you will obtain the window shown in figure 1.13 (which may be quite different looking in different versions of Max).


Documentation: Home

Search

Max
- Tutorials
- Messages to Max
- Tips & Shortcuts
- JavaScript in Max

MSP
- Tutorials
- How Digital Audio works
- MSP Introduction
- Messages to DSP

Jitter
- Tutorials
- Jitter Attributes
- What is a Matrix?
- Messages to Jitter

Contents

Max: Topics, Tutorials, Guides
MSP: Topics, Tutorials, Guides
Jitter: Topics, Tutorials

Configuring Objects In A Patcher
Debugging Max Patchers
JavaScript Usage
Max Basics
MIDI
Online Resources
Organizing Your Patch
Scripting
Search Paths
Sharing Max Patchers

Fig. 1.13 The Documentation window

In the figure we see a part of the *Documentation* window for Max. At the top are the three principal categories: **Max** (the software that manages control functions - basically everything that doesn't have to do with generating or processing audio, graphics, and/or video), **MSP** (which manages the generation and processing of audio signals), and **Jitter** (which does video and graphics processing as well as matrix management). For each of these categories there are Tutorials that introduce the elements in a systematic way, as well as a few more that cover central topics. Lower down the page, there is a table of contents containing "Topics" for every category, a list of "Tutorials", and a series of "Guides" that examine in depth numerous aspects of programming and operating the Max environment. You should definitely try to take time to tour the documentation; it will help you better understand how things work.
There are also help patches for all of the individual objects found in Max. When you are in edit mode, <Alt-Click> on an object (without dragging), and a help patch relevant to the clicked object will open. This patch will be fully functional, and will summarize the principal characteristics of the object. <Alt-Click> while in edit mode on the `cycle~` object, for example, will bring up the **help patch** seen in figure 1.14 (which again may be quite different in different versions of Max).

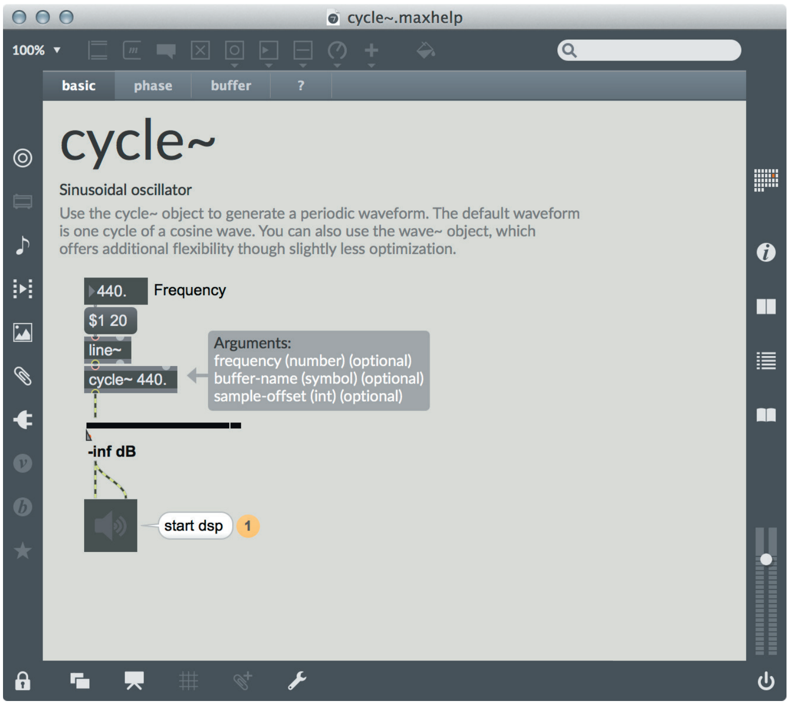

cycle~.maxhelp
100%
basic
phase
buffer
?
cycle~
Sinusoidal oscillator
Use the cycle~ object to generate a periodic waveform. The default waveform is one cycle of a cosine wave. You can also use the wave~ object, which offers additional flexibility though slightly less optimization.
440.
Frequency
$1 20
line~
cycle~ 440.
Arguments:
frequency (number) (optional)
buffer-name (symbol) (optional)
sample-offset (int) (optional)
-inf dB
start dsp
1

Fig. 1.14 A help patch

Help patches all share a very specific structure: they are divided into panes that can be displayed by clicking on tabs that are visible in the upper left part of the window: each individual pane explains some specific characteristics or behaviors for the object in question. The number of tabs and the names on these tabs will vary from object to object, with the exception of the first and last tabs, which are shared by all objects and are labeled consistently. The first of these shared tabs, labeled "basic", highlights the fundamental features of the object being examined, while the last shared tab, labeled with a question mark, contains a menu whose first item, "Open Reference", can be used to call up a detailed page from the reference manual, and whose succeeding menu items lead to help patches for related objects, to objects that are similar to the object in question, and finally, to tutorials illustrating the use of the object. Whether or not you read the deeper explanatory material contained in a help patch such as this one, we still recommend that you take a peek! They are working patches that can teach a great deal about the practical techniques and specialized vocabulary used in Max.
Another useful source of information is the **Clue Window**, which can be called up via the *Window* menu. This window, when viewed using the default (assumed) color scheme and settings,[14] appears as a small yellow window that

[14] Default colors and settings can be easily modified by Max users, resulting in alternative an look and feel for the program.

floats above other windows and that displays information relating to whatever lies beneath the mouse pointer. Try activating it and then moving the mouse pointer over the various elements of a patch, or over any of the numerous menu items. The clue window will display a short description for all of these different items.

The comprehensive help system is undoubtedly one of the strongest points of the Max environment. We have already encountered other aspects of it: remember the Assistance Bubble text, which we see every time that we work with patch cords. The Assistance Bubble dynamically describes messages that can be sent to or received from an object's inlets and outlets. Recall that to see it, we need only hover with the mouse over an object's inlet or outlet when in edit mode (as in figures 1.6a and 1.6b). Another resource in the same vein is the **Quickref menu**. Open 01_01.maxpat once again, and put the patch into edit mode by clicking on the lock at the bottom left of the Patcher Window. Now hover with the mouse pointer over the left inlet of `cycle~` so that the red circle and Assistance Bubble appear. Right-clicking while inside the red circle, and then holding the mouse button down, will cause the *Quickref* menu to appear (as shown in figure 1.15).

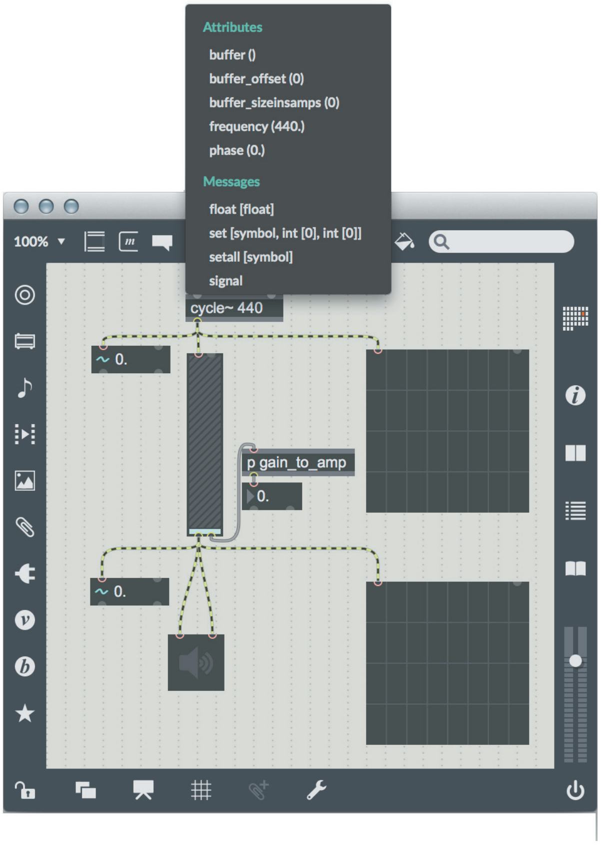


Fig. 1.15 The Quickref menu

This menu is categorized into two types of elements: **Attributes**, about which we will speak more in a bit, and **Messages** which correspond to the messages that a given object is able to "understand" and use. By selecting a message from the list, it is possible to create a new object that is automatically connected to the target object (in this case, **cycle~**). Click, for example, on the float [float] item, and a new object will appear, already connected to **cycle~**, as in figure 1.16.

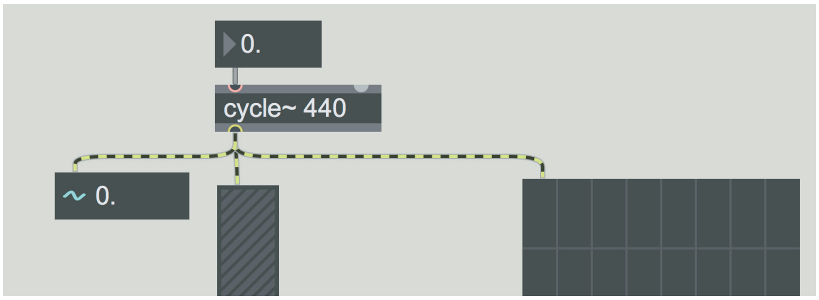


Fig. 1.16 Connecting an object by means of Quickref

If you now put the Patcher into performance mode (by closing the lock in the lower left with <Click>), and make vertical sliding strokes of the mouse over the new object while holding the mouse button down, you will see the numbers change in the object's UI. This object's purpose is, in fact, to store and enter numeric values; it is named **number**, or *number box*. Numbers that we generate by sliding the mouse over this object are sent to **cycle~** to modify its frequency. (You will note, however, that the argument of 440 that you can see inside of the **cycle~** object doesn't change. Instead, the frequency is overridden invisibly by the newly transmitted values.) Run the patch by clicking on the loudspeaker icon and raising the fader to about 3/4 of its range, and then set the number in the number box to values between 500 and 1000 by sliding the mouse. You will hear the oscillator change pitch, driven by the frequencies being output by the number box. We will speak at much greater length about the number box in following sections of this chapter.
If you now move the mouse pointer to the left side of an object (in *edit* mode, of course) you will see a yellow circle appear, containing a small triangle (fig. 1.16b). Click on this icon, and a new context menu will appear, called the *Object Action Menu.*

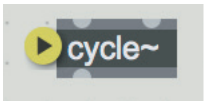


Fig. 1.16b The *Object Action* icon

This menu (fig 1.16c) contains the most useful elements for managing, transforming, and finding information about an object. In practice, all of the items found in this menu are also reachable in other ways (and we will progressively find out what use they serve); this menu has the advantage of placing them all where they can be invoked with a single click.

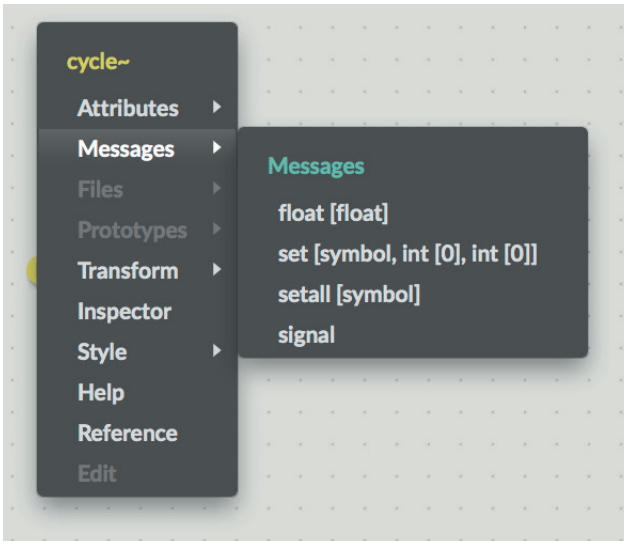


fig. 1.16c The Object Action Menu

PATCHER WINDOW TOOLBARS

We will now examine in more detail the four *Toolbars*, which, as we have said, surround the *Patcher Window* (which is, to say, the window in which we write our Max programs). The **Upper Toolbar** contains the interface objects available in Max, grouped into categories (see 1.17), as well as a Zoom menu in the upper left corner, by which it is possible to enlarge or shrink the view of the *patch*. The first four icons after the Zoom menu correspond to four frequently used objects: the *object box* (which you already are familiar with), the *message box*, the *comment box*, and the `toggle` object, whose acquaintance we will make in a bit. Five icons follow, from which it is possible to open the same number of *palettes*, each containing objects from a pre-determined category:

Buttons: this *palette* contains interface objects that act like buttons.
Numbers: this *palette* contains interface objects that display and generate values (such as the *number box* that we already encountered in fig. 1.16).
Sliders: this *palette* contains interface objects that function as cursors; in other words, they produce ranged values when you drag them using the mouse.
Max for Live: this *palette* contains the interface objects for Max for Live (which we will discuss in the second volume).
Add (other objects): this *palette* contains all of the other relevant objects, and is sub-divided into the sub-categories *Basic, Audio, Data, Images, Interface*, and *Jitter*.

We have already seen the *Audio* sub-category in fig. 1.7, and we will explore the contents of the other categories as we find the need.

There is a final icon in the upper *Toolbar* that looks like a can of paint. This icon doesn't serve to create objects, but rather, it is used to configure the graphical appearance of the *Patcher Window* and of the Max objects contained in the *Patcher Window*: a click on the icon brings up the *Format Palette*. We will not occupy ourselves with details of appearance during this first section, but you should explore the icon anyway: select (in *edit* mode) an object box within patch 01_01.maxpat and, after bringing up the *Format Palette*, try modifying its color, or the size and type of font that it exhibits, for example.

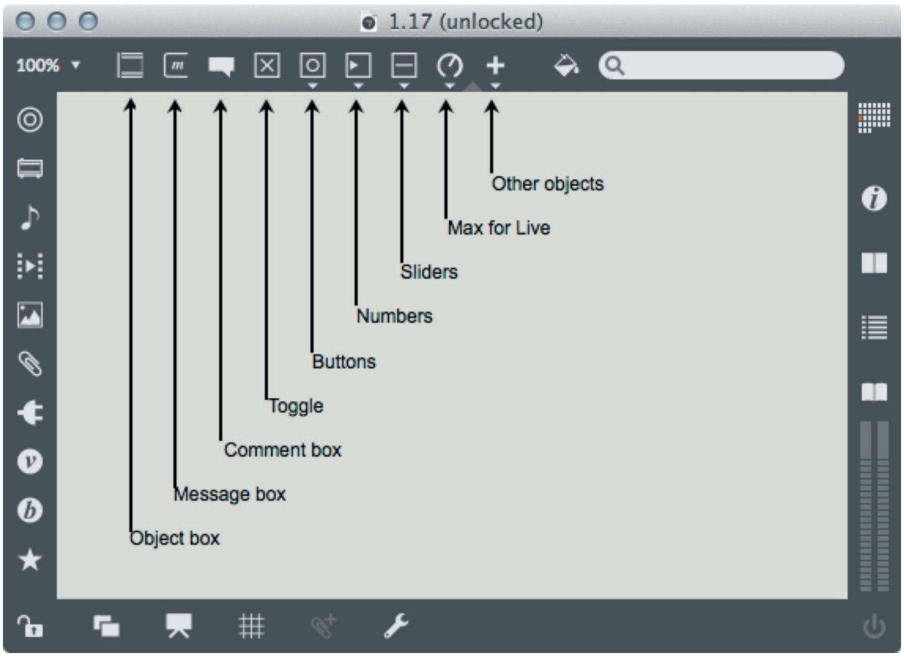


fig 1.17 the upper *Toolbar*

The right part of the upper *Toolbar* is occupied by a text field that enables us to search the Max documentation: try, for example, to search for the term "oscil-lator".

By using the **Left Toolbar**, it is possible to access any or all of the many types of files used when programming Max (fig. 1.17b).

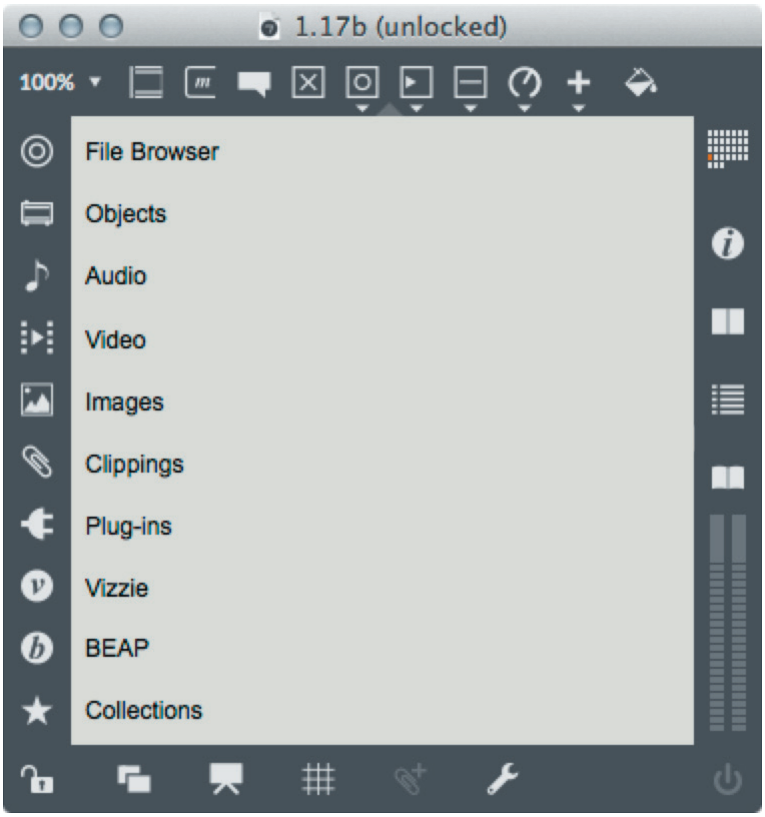


fig 1.17b The left *Toolbar*

Skipping over the first icon for a moment, which we will discuss after we introduce the *File Browser* near the end of the chapter, click on the second icon: a slide-out menu appears that contains all of the objects available in Max (not only interface objects, reachable from the upper *Toolbar*). Objects in this menu are divided into categories and sub-categories, and a double-click on the name of an object will cause it to appear within the patch. Alternatively, you can also drag a name into the *Patcher Window,* causing an instance of the object to appear. (These two alternative approaches to creating objects are consistently available from the *Toolbar* menus.)

The third icon enables us to locate and use audio files within a *patch*, the fourth, video files and the fifth, images. Note that to be visible in these menus, a file must be found in Max' *File Search Path*, which we will revisit later. As long as you are here, try, with the padlock icon open, dragging some files from the audio menu into the patching area and observe what happens; we will revisit this menu in Section 1.5. The sixth icon, shown as a paper clip, enables us to access *Snippets*, which are reusable fragments of code captured using the lower

Toolbar (see below). The seventh icon gives us access to Max for Live plug-ins, VSTs, and Audio Units that are installed on our computer. Just for the moment, we will ignore the remaining icons and move on.

The **Lower Toolbar** contains tools used while programming Max. The first icon is the padlock, which, as you already know, enables us to transition from edit mode into *performance* mode, and vice-versa. We will discuss each of the other icons contained in this *Toolbar* at the appropriate time, and for the moment occupy ourselves with the fifth icon, which is a paper clip embellished with a plus sign that closely resembles the paper clip found in the left Toolbar. This icon enables us to create *new snippets*. To understand the *snippet* feature, open the patch file named 01_01.maxpat (fig. 1.9), put the *Patcher Window* into *edit* mode, and select the `gain~` objects (the vertical faders), the `ezdac~` object (the little speaker), and the `[p gain_to_amp]` object[15]. At this point, click on the paper clip icon in the lower *Toolbar* and a small window will appear in which we can name the snippet being saved: type "audiostart" and hit enter. Next, create a new patch and click on the sixth icon of the left *Toolbar* (the other paper clip). A menu will appear that, among its other entries, now contains an item called "audiostart". Drag this element into the Patcher Window, and you will see the snippet that you previously saved appear. Using this technique, it is possible to capture frequently needed pieces of code for reuse. As you can see, the menu already contains some pre-defined snippets for you to use; you can easily create more while you work with Max.

We now move to the **Right Toolbar**. Besides the first icon, which opens a calendar with which you can look for recently used patches, all of the other icons will display some form of the **Sidebar**, which is a drawer-like window that opens on the right side of the *Patcher Window* and which contains a variety of useful tools and information. Open, for example, patch 01_01.maxpat (fig. 1.9), put it into edit mode, select the `ezdac~` object (the little speaker), and click on the second icon (shown as the letter "i" inscribed in a circle). This will open the *Sidebar* and display the *inspector* for the selected object. (The *inspector* contains all of the attributes that define the appearance and the behavior of an object, and we will speak about it in great detail later in the book.) The third icon enables quick access to documentation about the selected object, while the fourth opens the Max Console, a window through which, as we have already learned, Max displays error messages and other useful information. Finally, the fifth icon can be used to access a series of interactive lessons about Max. At the bottom of this Toolbar, the combined Audio Meter/Gain controls can be seen, which display and control the global audio volume of the patch, as well as an Audio On/Off button, which can be used to activate or disable the production of audio signals in the patch (in exactly the same way as the `ezdac~` object can).

[15] To select multiple objects, you can use the technique shown in fig. 1.10 or else click on the objects, one after another, while holding the shift key down.

SOME ORDER FOR A CHAOTIC WORLD

You probably noticed that some of the patch cords in patch 01_01.maxpat (figure 1.9) had angles in them which divided their connections into tidy segments. How was this done? The answer is simple: by selecting **Segmented Patch Cords** from the *Options* menu.

If you use this option, the procedure for connecting two objects will be slightly different than what we've seen already: begin connecting with a click on the outlet that will form the source of the connection, and then pull the patch cord out from the object *without holding the mouse button down*. (The cable will remain hooked to the mouse pointer.) Segments can be created at any time with a click of the mouse at the point at which you want to change direction; for every click, there is a new segment. The final click should be made on the inlet of the object to which you wish to connect.

If you make an error and want to release the cable without connecting, you need to <Command-Click> on the Mac or <Control-Click> on Windows, or else hit the escape key (<Esc>).

If you select some objects that are lined up roughly horizontally, and then press <Command-y/Control-Shift-a>, the objects will become perfectly aligned. The same command works to create vertical columns of objects one over the other. (The two `scope~` objects and the two `number~` objects in 01_01.maxpat were aligned in this way.) Objects can also easily be aligned using the **Snap to Object** function (which is active by default). Every time that you position an object in a patch, this function will align that object with other nearby objects. Another related function is **Distribute**, which can be found in the *Arrange* menu; this function makes it possible to distribute a group of selected objects at equal intervals horizontally or vertically.

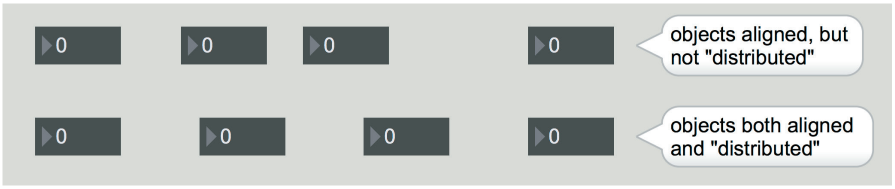

0
0
0
0
objects aligned, but not "distributed"
0
0
0
0
objects both aligned and "distributed"

fig. 1.18: the *Distribute* function

A complex patch can be very crowded, with dozens of objects and patch cords stretched between them. In this case, it is possible to render a set of the objects and cords invisible when in performance mode, but still visible while in edit mode, so that they can do their work behind the scenes. To hide one or more objects and patch cords, select them while in edit mode and press <Command-k/Control-k>. When the Patcher is put into performance mode, the objects will vanish. To reverse this action, and make the objects reappear during performance mode, use <Command-l/Control-l> (lower key "el") in edit mode. By selecting multiple objects, it is possible to hide multiple objects at once using these commands. An alternative to the keystroke commands is found in the *Object* menu, where **Hide on Lock** and **Show on Lock** can be found. Play with 01_01.maxpat, making various objects and patch cords in the patch appear and disappear.

There is yet a still more efficient way to put order into patches, which is to use a feature called presentation mode. We will explain this mode once we find ourselves building a more complex patch.

EXERCISE

Create a new Patcher Window and attempt to reproduce the patch contained in the file **01_01.maxpat**.
Use the snippet that you saved in the preceding paragraph (if you haven't yet done this, make it now!), because otherwise it will be impossible for you to create the `[p` gain_to_amp`]` object.
Pay careful attention that you don't confuse the `number~` object with the number box! If you don't succeed in finding `scope~` and `number~` in the Top Patcher Window Toolbar, remember that you can always take an object box, type the name of the object into its interior, and morph the object box into the graphic object that you seek. Note that the displayed waveform on the oscilloscope created by you is different from that in the original file. We will see why in the following section.

(...)

other sections in this chapter:

Interlude A
PROGRAMMING WITH MAX

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER
- Contents of Chapter 1 (Theory and Practice)

LEARNING OBJECTIVES
Skills
- To learn how to use the basic features of Max that pertain to integers and floating point numbers
- To learn how to generate and control sequences of random numbers, optionally with the use of a metronome
- To learn how to construct embedded objects and abstractions
- To learn how to repeat messages across multiple object outlets
- To learn how to assemble and manipulate lists, using both non-graphic and graphic methods
- To learn how to use variable arguments
- To learn how to manage communication between objects without the use of patch cords

CONTENTS
- Integers and floating point numbers in Max
- Generating and controlling random numbers with the objects `random`, `drunk`, etc.
- Generating regular rhythmic events using the `metro` object
- Constructing subpatches and abstractions
- Managing list and variable arguments
- Using the `send` and `receive` objects for wireless communication between objects

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

TESTING
- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS
- List of Max objects
- Messages, attributes, and parameters for specific Max objects
- Glossary

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

IA.1 MAX AND THE NUMBERS: THE BINARY OPERATORS

Like any respectable programming language, Max can do many things with numbers. We will begin this chapter by looking at the simplest operators, those for addition, subtraction, multiplication, and division.

INTEGER ADDITION

Recreate the simple patch shown in figure IA.1 (and make sure that there is a space between '+' and '5'!).

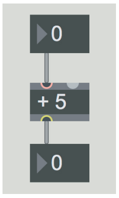


Fig. IA.1 Addition

The **+** object adds its argument (which is, in this case, 5) to whatever number it receives on its left inlet. If we feed some numbers to the object via the number box above it (by selecting it, for example, in performance mode and then using the arrow keys on the keyboard to change its value), we can track the results of the operation in the lower number box.
The right inlet of + changes the argument, and if we enter a number on 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 input via the left inlet.
Try this by adding a number box on the right, as shown in figure IA.2:

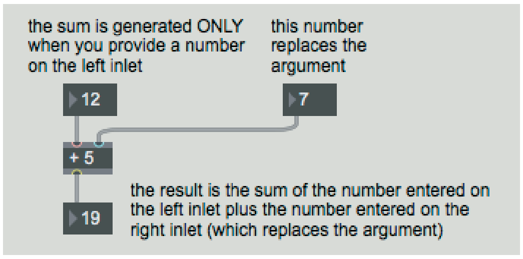

the sum is generated ONLY when you provide a number on the left inlet
this number replaces the argument
12
7
+ 5
19
the result is the sum of the number entered on the left inlet plus the number entered on the right inlet (which replaces the argument)

Fig. IA2 Addition with a changeable argument

When you play with this patch, note that the + operation is executed only when a number is sent to its left inlet, and never when a number is sent to its right inlet. Every time a new number is sent to one of the two inlets, an internal variable corresponding to that inlet is updated. The old contents of the variable are deleted, and replaced by the new value. In addition, a number sent to the left inlet triggers the + operation, summing the contents of the two internal variables and emitting the result. This is a common pattern: a large majority of Max object execute their functions and emit results only when a message is received on their left inlet. Messages that enter other inlets either modify arguments by updating internal data structures or else modify the behavior of the objects without causing visible results.

In the lexicon of Max objects, the left inlet is defined as a "**hot**" **inlet**, one that causes the operation to be carried out on the object in addition to updating the inlet's corresponding internal values. Most other inlets are defined as "**cold**" **inlets**, which update internal variables without producing output. Note that the circle that appears around the inlet of an object when you hover above it with the mouse in edit mode is red for the "hot" inlets and blue for the "cold" inlets.

Is there a way to update the value of the sum when we send a number to the right inlet of an addition object? In other words, is there a way to convert a "cold" inlet into one that behaves like a "hot" one? Of course there is, and we can examine the technique in figure IA.3:

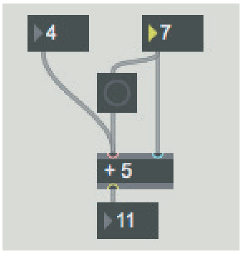


Fig. IA.3 How to make a "cold" inlet behave like a "hot" one

In this patch, we first send a number to the right inlet, and then a bang to the left inlet (remembering the right-to-left order for Max messages). Since `button` transforms anything that it receives into a bang, inputting a number on the right inlet produces this bang message, which, as you know, will force the receiving object to produce a result, which, in this case, is our sum.

What is added? The numbers within the object's internal variables, which in this case are the number last sent to the right inlet (in the figure, the number 7) and the last number sent to the left inlet (which is 4 in the figure).[1]

[1] Recall that in this figure, just as in the next, the original argument '5' is replaced by any numbers sent to the right inlet.

In this style of patch, the "cold" inlet of the addition object behaves as though it were a "hot" inlet. Try building it, and verifying that the number box on the right triggers a result every time that its value is changed.

It is essential that the positions of the objects in your patch are absolutely the same as the positions in the figure. Placing the `button` to the right of the + object, for example, will produce undesired results, since the bang will fire before the new value is stored, causing the addition object to use the old value that had been copied into the internal variable previously in its calculation rather than the new value (as shown in figure IA.4).

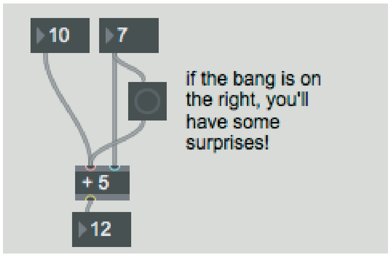


Fig. IA.4 Buggy results produced by an error in the order of execution

Delete the two number boxes above the + object and connect a message box to its left inlet. After this is done, type two numbers (separated by a space) inside the message box, as in figure IA.5:

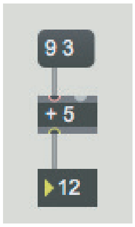


Fig. IA.5 Summing a list

If you now click on the message box in performance mode, the + object will sum its values; the operator behaves as though it had received the second number on the right inlet (and, as usual, the argument 5 inside the object box is replaced by that new value). This behavior (the ability to accept lists of two or more elements on their left inlet and then route the individual list items to the other inlets) is also a common feature of many Max objects, and works not only with binary operators, but often with objects that have three or more inlets.

To see a simple musical application of Max's addition object, open the file **IA_01_transposition.maxpat** (seen in figure IA.6).

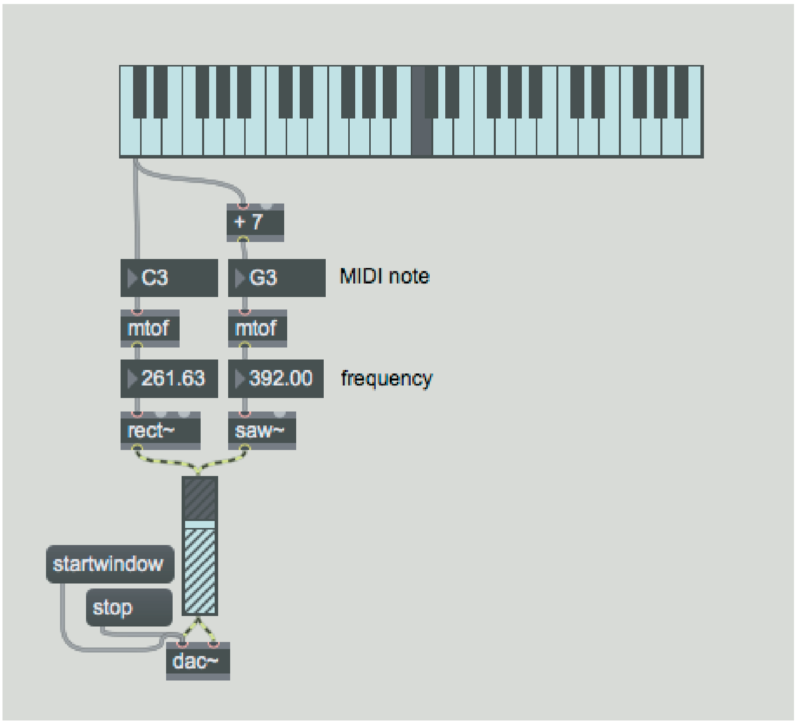


Fig. IA.6 The file IA_01_transposition.maxpat

Every time a key is pressed on the **kslider** object in this patch, two notes are generated, the second of which is 7 semitones above the first – the musical distance of a fifth. Every time that we press a key on the **kslider**, the value of the corresponding MIDI note number (a C in figure IA.6, for example) is sent to the **mtof** object on the left, which converts it into a frequency value. At the same time, the note number is also sent to a **+** object that adds 7 (producing the MIDI note number for a G in the example). The resulting sum is sent to the **mtof** object on the right side of the patch.
By using an addition object in this way, it is possible to transpose MIDI notes by arbitrary intervals of our own choosing. In this example, after the MIDI values of the two notes are converted into frequencies, they are sent to two oscillators, **rect~** and **saw~**, which sound at the interval of a fifth. Try modifying the patch to use other intervals (a third, or 4 semitones; a fourth, or 5 semitones; an octave, or 12 semitones, etc.). After this, try adding another addition object, connected to another dedicated **mtof** object and oscillator, so that every **kslider** keypress produces a three note major chord. For example, if the C2 key were pressed (the key that corresponds to the note one octave below middle C), a chord of C E G in the second octave (C2 E2 G2) would be produced.

NUMBERS WITH A DECIMAL POINT, AND OTHER OPERATIONS

Up until now we've used only integers. When the + object is given an integer as its argument (or has no argument at all), Max will assume that you would like to perform integer math. To make the object work with floating point numbers, as shown in figure IA.7, we need to connect float number boxes to its inlets, and, as you can see, to provide an argument of "0.". (The decimal point is important, but it is not necessary to put any numbers after the decimal point.) The decimal point in the argument declares to the + object that we intend to use non-integer numbers on its inlets, or in other words, that we intend to use floating point math. Duplicate the patch yourself and do some sums using floating point numbers (remembering that the "hot" inlet is on the left).

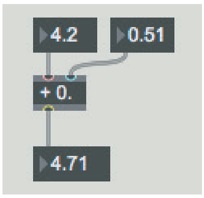


Fig. IA.7 Summing numbers that contain decimal points

Everything that we have covered to this point about addition also applies to subtraction, multiplication, and division. Try building the patch shown in figure IA.8 to verify that this is the case:

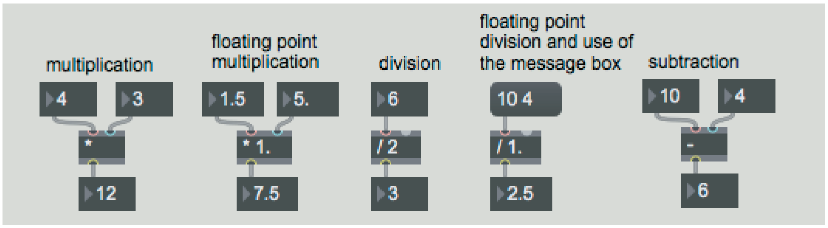

multiplication
4
3
*
12
floating point multiplication
1.5
5.
* 1.
7.5
division
6
/ 2
3
floating point division and use of the message box
10 4
/ 1.
2.5
subtraction
10
4
-
6

Fig. IA.8 Other mathematical operators

A brief piece of advice is in order at this point: do these exercises! Although they may seem trivial, doing them will help you to notice the details and the quirks of the Max environment, details whose importance will be revealed in the patches that we will pursue in the following chapters. For example, in the test case with the comment "floating point division and the use of the message box", you see that we are submitting a list of *integers* ("10 4") for evaluation. Despite the fact that integers are submitted, the operator was already declared to represent a floating point operation, thanks to the floating point argument provided in its object box, and as a consequence, the result emitted will be a floating point number.
Let's again use some of these simple operations in a musical context. You'll remember how the distance between two notes, an interval, can be expressed as a ratio between the two frequencies. For example, the interval of a fifth, corresponding to 7 semitones, can be expressed using the ratio 3/2.

Given a frequency (let's say 261.63 Hz, which is middle C), if we multiply the frequency by 3 and then divide by 2, we will obtain the frequency for the note a fifth above (which in this case would turn out to be 392.44 Hz, a G). To see this in action, open the file **IA_02_fifth.maxpat** (shown in figure IA.9):

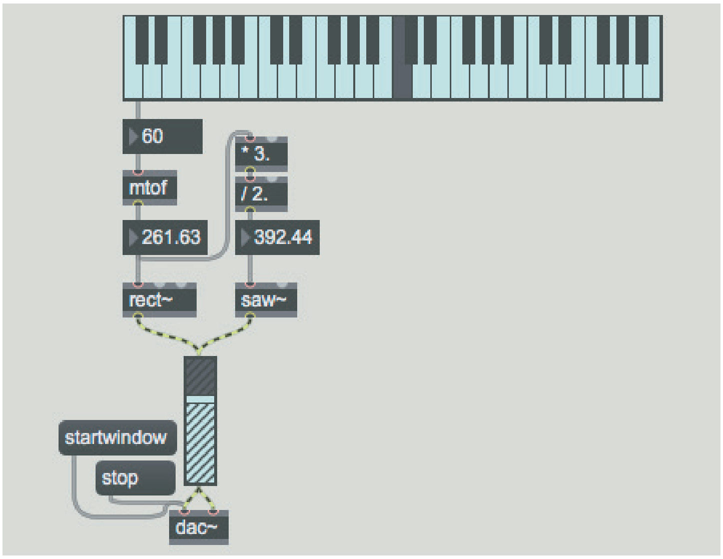


Fig. IA.9 The file IA_02_fifth.maxpat

In the patch we see that the note number generated by the `kslider` is transformed into frequency though the use of `mtof`, and is then sent to a `saw~` oscillator. The output of `mtof` is also multiplied by 3/2, in order to obtain the frequency needed to sound the note a fifth above, which is then sent to a second `saw~` oscillator. Note that the arguments to the multiplier object and to the divider object have a decimal point as their last character, which declare that Max should use floating point operations. Also note the routing of the two patch cords that exit the float number box under the `mtof` object: one cord is connected directly to the `saw~` object below it, but the second heads off to the right, heads upwards, and finally winds up connected to the multiplier object that has the argument 3.
If you compare figures IA.6 and IA.9, notice that the frequencies calculated for the note a fifth above are different. By using two distinct `mtof` objects in the first patch, we calculated a *tempered* fifth (the fifth normally found in western music), which is defined as the interval formed by combining 7 tempered semitones. When we used the ratio 3/2 on the output of a single `mtof`, however, we calculated a *natural* fifth, which is slightly wider that a tempered fifth. The difference is around 2 *cents* (which is a unit defined to be 1/100 of a tempered semitone).

OPERATIONS THAT USE THE EXCLAMATION POINT

All of the operators spoken of so far are *binary operators*, so called because they need two inputs (also called operands in computer lingo) to produce their output. In the objects that we have already seen, the first operand is the number that is input on the left inlet and which triggers the operation, while the second operand is the argument (or else the number on the right inlet).
For subtraction and division, however, there also exist two objects for which the operands are reversed. Their second operand is input on the left inlet and triggers the operation, while the first operand is the object's argument.
The name for these "reversed" objects is made up of an exclamation point plus the character for the underlying operation: **`!-`** for subtraction (which we have already encountered in patch 01_17_pan.maxpat), and **`!/`** for division.
See figure IA.10 for examples:

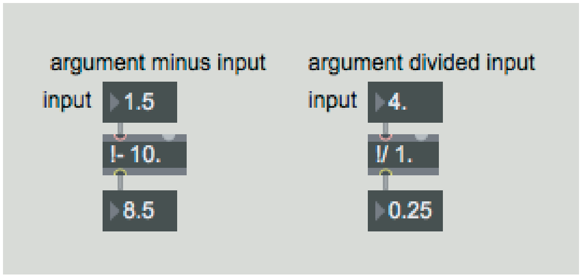


Fig. IA.10 Operators that contain the exclamation point

In the first case, the value being input (1.5) is subtracted from the argument (10), and the result is 8.5 (10 - 1.5). In the second case, the argument (1) is divided by the value being input, which results in 0.25, since 1/4 = 0.25.
The operands are reversed with respect to the normal order for subtraction and division. Build this patch and compare these operations with their analogs that lack an exclamation point. We already encountered the **`!-`** operator in action in Section 1.6, in the patch 01_17_pan.maxpat.

All of these operators also exist in MSP versions that generates a signal (**`+~`**, **`-~`**, **`*~`**, **`/~`**), which we have already used a few times in the previous chapter. MSP operators require a signal to be present on one of the two inlets (usually the left), and can receive either a signal or numeric values on their other inlet. (They can also use a single value provided as an argument.) For more information about operators and objects, remember that <Alt-Click> will bring up help patches when in edit mode.

THE INT AND FLOAT OBJECTS

Two objects exist that allow you to store values and recall these values later using a bang. They are **`int`** (for the storage of integers) and **`float`** (for floating point numbers).

In figure IA.11, we see that these objects possess two inlets; if a value is sent to the "hot" left inlet, the value is both stored and also immediately transmitted on the outlet, while if a value is sent to the "cold" right inlet, it is stored but not transmitted. To retrieve the stored value at any time from either of these objects, a bang can be sent to their left inlet, which will cause the value to be transmitted. In both cases, incoming values are copied into the object's memory (and can therefore be recalled with a bang whenever we'd like) until new values take their place.

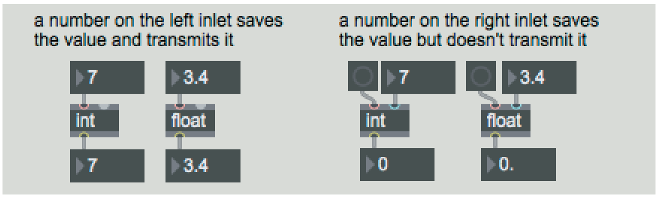

a number on the left inlet saves the value and transmits it
a number on the right inlet saves the value but doesn't transmit it
7
3.4
int
float
7
3.4
7
3.4
int
float
0
0.

Fig. IA.11 The `int` and `float` objects

(...)

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 Max/MSP objects
- Commands, attributes, and parameters for specific Max/MSP 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.

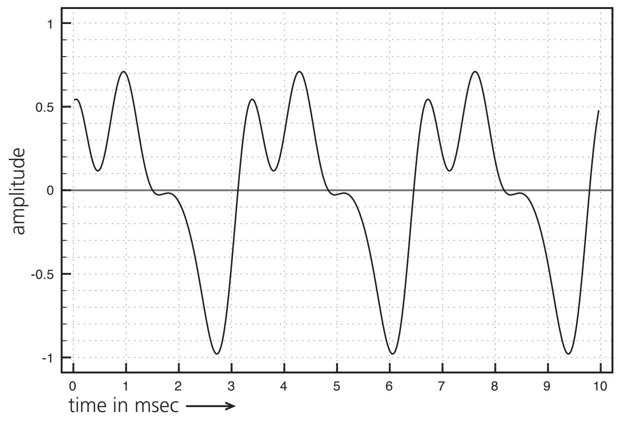


Fig. 2.1 The waveform of a complex sound

[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.

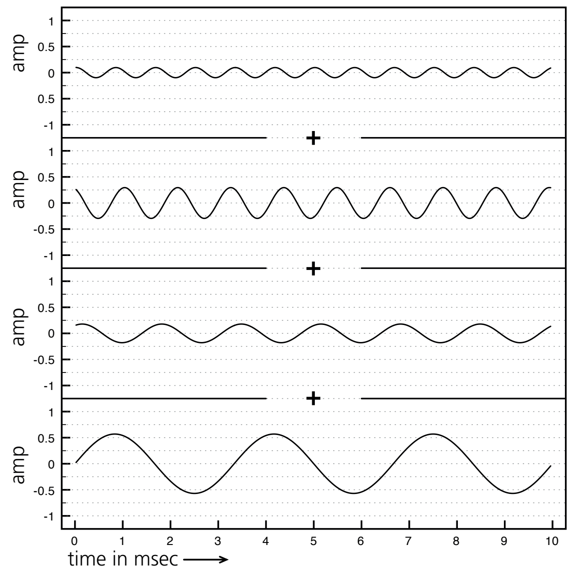


Fig.2.2 Decomposition of a complex sound into sinusoidal components

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.

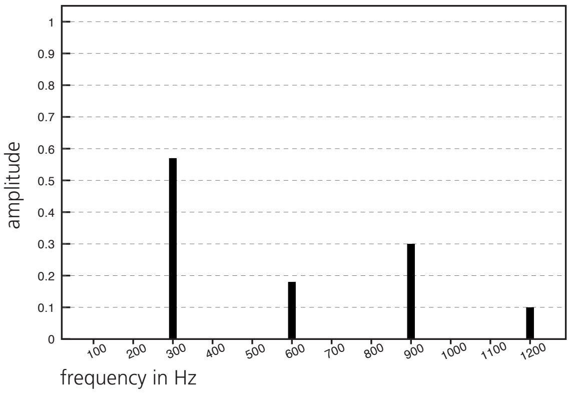


Fig. 2.2b A sound spectrum

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.

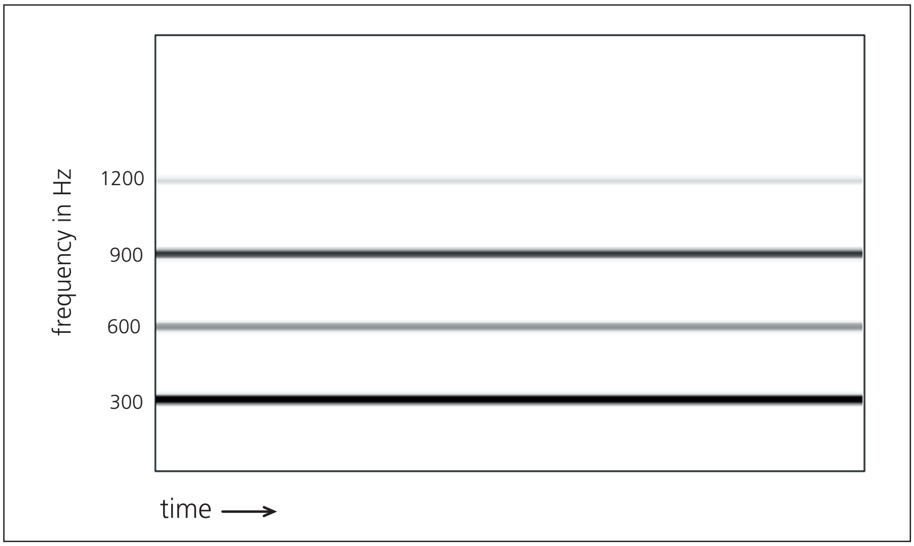


Fig. 2.2c A sonogram (also called a spectrogram)

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.

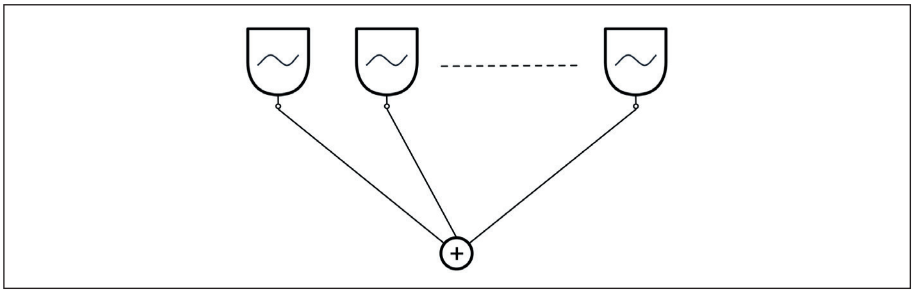


Fig. 2.3 A sum of signals output by sine wave oscillators

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

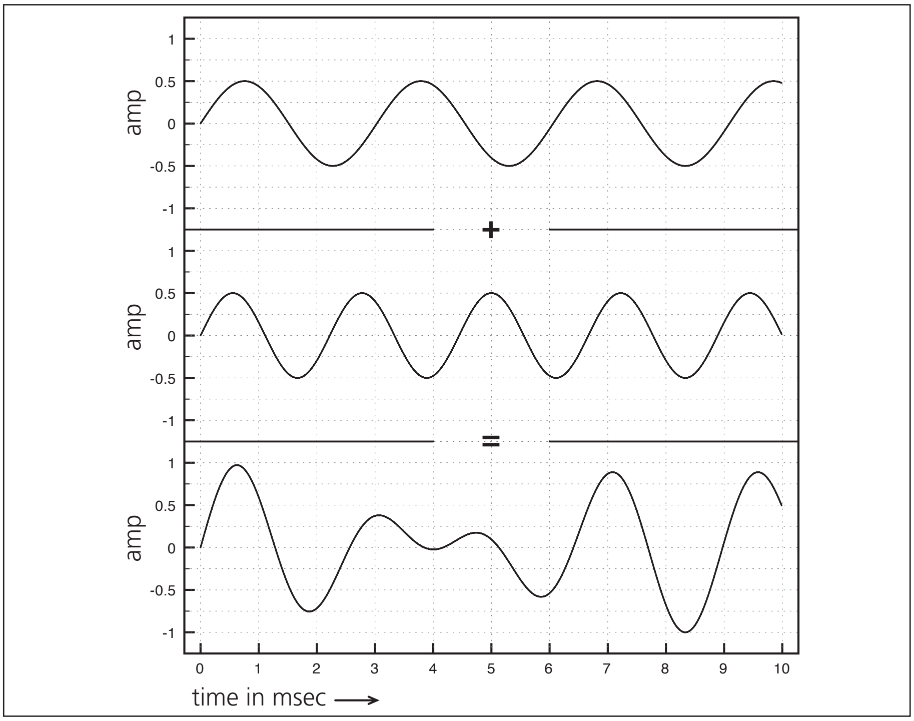


Fig.2.4 A graphical representation of a sum of sine waves

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., 2001, 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*

If the pure sounds composing a complex sound are not integer multiples of the lowest frequency component, we have a non-harmonic sound and the components are called **non-harmonic components**, or **partials**.

INTERACTIVE EXAMPLE 2B – *NON-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. 2003)

other sections in this chapter:

2P
ADDITIVE 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

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

SUPPORTING MATERIALS
- List of principal Max commands
- List of Max objects
- Messages, attributes, and parameters for specific Max objects
- Glossary

2.1 FIXED SPECTRUM ADDITIVE SYNTHESIS

To start things off, let's create a patch for producing harmonic sounds in Max using additive synthesis, with figure 2.12 of the theory chapter as our implementation guide. The diagram in that figure shows 10 oscillators, all summed using a mixer, and so we will begin with 10 `cycle~` objects that will furnish the 10 sine waves, each producing a frequency which is an integer multiple of the fundamental (in order to be harmonic). To calculate the frequency values needed, we will simply multiply the frequency of the fundamental by the first 10 integers. The resulting patch is shown in figure 2.1 (**02_01_additive.maxpat**).

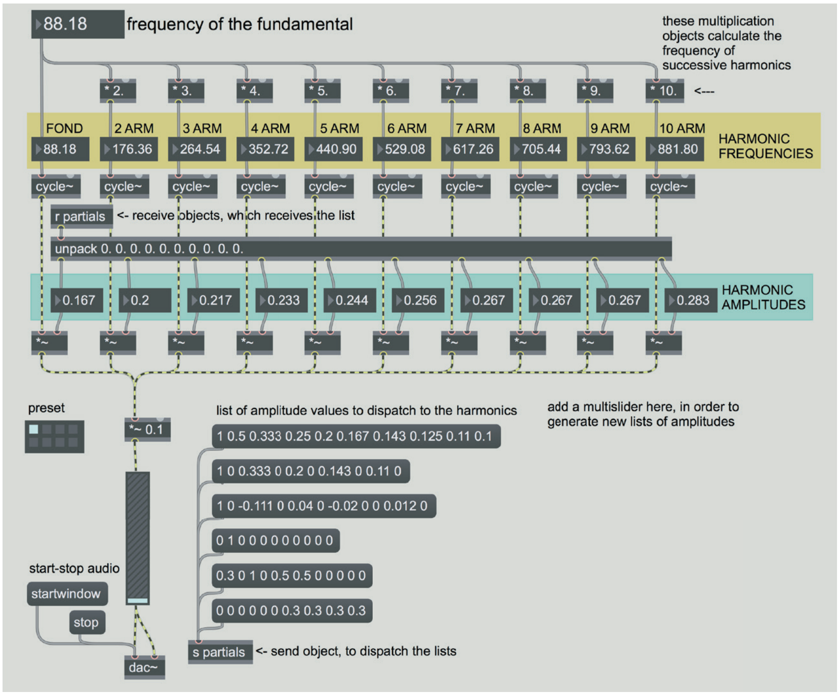


Fig. 2.1 The file 02_01_additive.maxpat

The number box at the top of the patch is used to set the fundamental frequency for the entire patch, which is then passed to the 9 `*` operators using successive integers to produce harmonic frequencies that are multiples of the fundamental. The frequency of the fundamental and of the frequencies being produced by these multipliers can be viewed in the 10 number boxes below, which are themselves connected to as many `cycle~` objects. These, in turn, are connected to signal multipliers (`*~`) that rescale their outputs. Normally the amplitude of a signal produced by `cycle~` will have a peak value of 1, and for this reason the value given to each multiplier serves to directly reflect the relative amplitude of each harmonic (for example, 1.0 · 0.5 = 0.5). The signal multipliers enable each harmonic to have its own distinct amplitude, determined by the value fed into its multiplier. Given that we have 10 oscillators, 10 values completely specify the amplitudes to be used.

As we learned in Section IA.10 of Interlude A, we can create a list of numbers by using either a message box or a **multislider**. The first technique is immediately obvious in this patch: desired amplitudes are grouped into lists within message boxes, and then dispatched to the 10 multipliers by using a send object ([**s** partials]). The corresponding **receive** object is connected to an **unpack** object that takes care of distributing single values to appropriate multipliers (as seen in figure 2.1b).

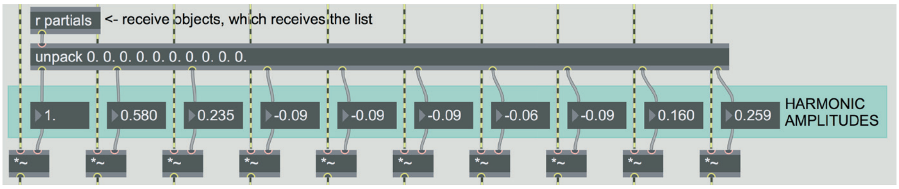

r partials
<- receive objects, which receives the list
unpack 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.
1.
0.580
0.235
-0.09
-0.09
-0.09
-0.06
-0.09
0.160
0.259
HARMONIC AMPLITUDES
*~

Fig. 2.1b The **unpack** object breaking a list into single elements

Amplitude lists might also, however, be created using a **multislider**. Create one now in the lower right of the patch, and configure it using the Inspector to have 10 sliders, each ranging from 0 to 1. This **multislider** will need to be connected to a **send** object with the argument “partials”. It’s also possible to manipulate the float number boxes connected to the right inlet of the signal multipliers in order to set different amplitudes, and then save the result by using the **preset** object on the left (by shift-clicking on the small dots).

The component signals in the patch, after being rescaled by their individual amplitude multipliers, are routed to a final multiplication object in the signal path that reduces the overall amplitude of the summed signal to 1/10th of its original strength. We know that any two or more signals that enter a single inlet are summed. So in the case of 10 oscillators, if all of the components have the maximum amplitude of 1, the resulting summed waveform would have an amplitude of 10. This multiplier, therefore, serves to return the absolute value of the signal to a value between 0 and 1, thus avoiding distortion.[1]

The amplitude **multislider** must dispatch data continuously, since we would like variations to be sent out as they are made, even while the value of a slider is being changed. By default, however, **multislider** updates the list of values that it outputs only when the mouse is released, at the end of an edit. To configure the object to produce continuous updates, you must call up the Inspector for the object, and after selecting the *Style* tab, check the box that enables “Continuous Data Output when Mousing” (“Style” category).

Using this patch, create various spectral profiles, saving them as presets and changing between them to compare the differences in timbre made by lower, middle, and upper components, as we covered in the theory chapter.

[1] For a discussion of distortion due to excessive amplitude see Section 1.2 of Chapter 1T.

ACTIVITIES

Using the patch 02_01_additive.maxpat, create two presets, one that has the amplitudes of the odd harmonics set to 1, and those of the even harmonics set to 0. Apart from timbre, what is the obvious difference? Why?

Continuing to use 02_01_additive.maxpat (and referring to the discussion of missing fundamentals in Section 2.1T), begin with a spectrum in which the amplitudes of all harmonics are 1, and then zero out the amplitude of the fundamental. Listen for the illusion that the zeroed fundamental appears to be present. Try reducing the amplitude of successive harmonics, one by one. At what point can you no longer perceive the fundamental?

PHASE

We now take on a topic discussed at length in Section 2.1T (Section 2.1 of Chapter 2T), the *phase* of a periodic waveform. It is a basic concept that will develop over many chapters; because of this, it is important to understand how phase is managed using Max. Probably you have noticed that the **cycle~** object has two inlets: the left, as we already know, allows us to set the frequency, and the right allows us to set the *normalized phase*, a phase that is expressed as a value between 0 and 1 (rather than a value between 0 and 360 degrees or between 0 and 2π radians, both of which we encountered in the box dedicated to phase in Section 2.1T).
Build the patch shown in figure 2.2 in a new Patcher Window.

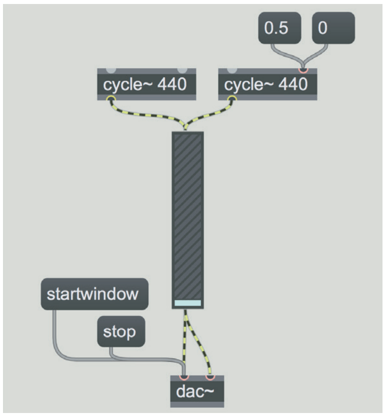


Fig. 2.2 Constructive and destructive interference

We have two **cycle~** objects at 440 Hz, the second of which has two message boxes, containing the numbers 0.5 and 0, connected to its right inlet (its phase inlet, as we just learned). Start the patch by clicking on "*startwindow*" and raising the **gain~** fader: you will hear a perfectly in-phase A440 generated by the two oscillators.

Clicking on the message box that contains 0.5, the sound vanishes. What has happened? We've set the phase of the second oscillator to 0.5, which is exactly half a cycle (180 degrees). The two oscillators become polarity-reversed[2] and because of this, their sum is always zero. Clicking on the message box that contains the number 0 will reset the two oscillators to being in-phase, enabling us to hear the sound once again.
To better understand the phase combinations, modify the patch to look like figure 2.3.

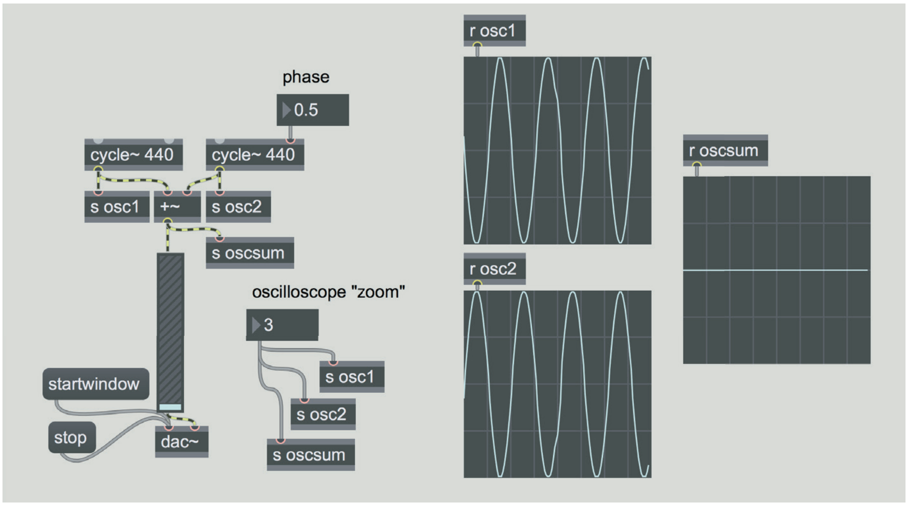


Fig. 2.3 Variations of phase

We've now replaced the two message boxes connected to the right inlet of the second oscillator with a float number box, and we've added three **send** objects to route the signals produced by the oscillators and their sum to three **scope~** objects. Besides this, we now vary the "Samples per Pixel" attribute[3] of the three oscilloscopes to control their displays. This is done by using three additional **send** objects, all connected to an integer number box. You will note that both signals and numerical values are sent to the oscilloscopes using the same name ("osc1", "osc2", and "oscsum") – recall that the **scope~** object accepts both the signals to be displayed and the parameter values to control that display on the same inlet.
After setting the number boxes to the values shown in the figure and clicking on "*startwindow*", you should see polarity-reversed sine waves displayed in the first two oscilloscopes, and a "null" waveform (a flat line) in the third. Click on "*stop*" and check that the two sine waves actually have opposite phase, as we have done in the figure. Now, after clicking "*startwindow*" again, modify the value in the float number box to control the phase of the second oscillator.

[2] See the box dedicated to phase in Section 2.1T.
[3] See Section 1.2, where we demonstrated that this attribute can be considered, only a little inappropriately, as a kind of "zoom factor" for the oscilloscope.

When it begins to drop from 0.5 towards 0, the oscillators are no longer in anti-phase, and you hear a sound growing progressively in amplitude until the two oscillators are completely in phase and their summed amplitude ranges between -2 and 2.[4] Try using phase values larger than 1, and note that whenever the phase is equal to an integer, the sound reaches its maximum amplitude, and whenever the phase is exactly halfway between two integer values (such as 1.5, 2.5, 3.5, etc.), the sound is cancelled out. Phase values larger than 1 (larger than 360 degrees) cause the cycle to wrap around, beginning anew, as we explained in the box dedicated to phase in Section 2.1T. Of course, all of these observations hold true only if the two generators share exactly the same frequency. If they don't, the ratio between the phases of the two oscillators would vary continuously, giving rise in the case that frequencies are only slightly different (for example 440 and 441 Hz), to the phenomenon of beats that will be covered in Section 2.2.
Let's look at another possibility for the right inlet of **`cycle~`** (as shown in figure 2.4).

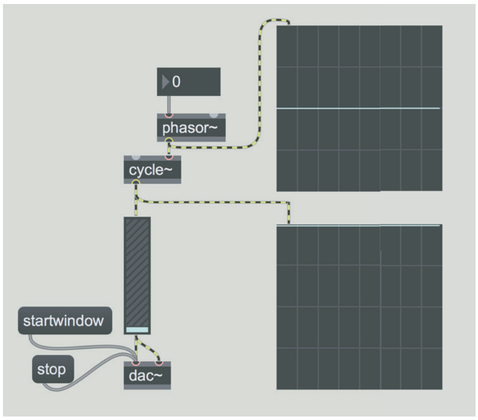


Fig. 2.4 Oscillator controlled by a **`phasor~`**

Here we have a **`cycle~`** set to 0 Hz, since it has neither an argument nor has it received a numeric message on its left inlet. The phase of this **`cycle~`** is set to be modified by a **`phasor~`**, which in the figure also has a frequency of 0 Hz, so to remain still. Looking at the oscilloscopes, we see that the **`phasor~`** generates a stream of zero-valued samples, which implies that it is frozen at the beginning of its cycle, and **`cycle~`** generates a stream of ones, which means that it too is frozen at the beginning of its cycle. It is important to stress that the cycle of a waveform generated by **`cycle~`** begins with the value 1; **`cycle~`** generates a cosine wave, as we pointed out in Section 1.1, which explains why the cycle starts at 1, its maximum positive value.[5] This is not a general rule: **`phasor~`**, for example, begins its cycle at 0.

[4] The resulting wave oscillates between -2 and 2 because it is the sum of two cosine waves (more on this shortly) that oscillate between -1 and 1 and are perfectly in phase

[5] For a definition of the cosine function, see theory sections 1.2 and 2.1.

Now we will give **phasor~** a frequency higher than 0 Hz; we set it to 5 Hz in the example shown in figure 2.5.

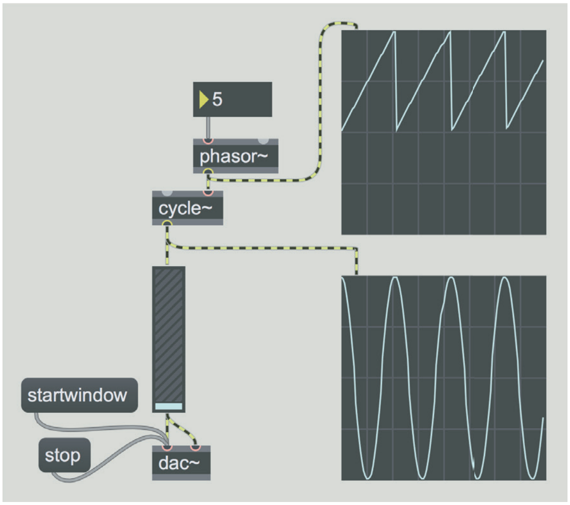


Fig. 2.5 **phasor~** set to 5 Hz

Build this patch yourself. The phase of the **cycle~** object is slaved to the **phasor~** object, causing it to oscillate at the same frequency and with the same phase as the **phasor~**. As you will recall from Section 1.2, **phasor~** generates a ramp from 0 to 1, and this ramp, when applied to the phase inlet of **cycle~**, produces the oscillation. (This explains the motivation behind the name **phasor~**: one of its main functions is to control the phase of other objects.)

As we see in figure 2.5, every ramp generated by **phasor~** results in one complete oscillation of **cycle~**, which is running at 5 Hz in this example. If we were to eliminate the **phasor~** and give **cycle~** a frequency of 5 Hz by sending it a message on its inlet, or by providing an argument, we would obtain an identical oscillation. In other words, **cycle~** behaves as though it has an "internal phasor". Such an "internal phasor", interestingly, is lacking in the **triangle~** object (encountered in Section 1.2), which can only oscillate when controlled by an external **phasor~**. Yet in the case of the **triangle~** object, we still speak about its phase (or index), thinking of this as the indexed absolute position within the waveform cycle (see Section 2.1T). And **triangle~** is not alone; we will see shortly another important example of an object that produces oscillations when connected to a **phasor~**.

One might ask what would happen if the **cycle~** object in figure 2.5 were to have its own frequency in addition to the frequency being provided by **phasor~**? The answer is: nothing particularly interesting. The two frequency values (that of the **cycle~** and that of the **phasor~**) are simply summed. Try this, by giving an argument of 440 to **cycle~**, for example, and then setting the frequency of **phasor~** to 220 Hz. The result will be a (co)sine wave with a frequency of 660 Hz.

(...)

other sections in this chapter:

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.

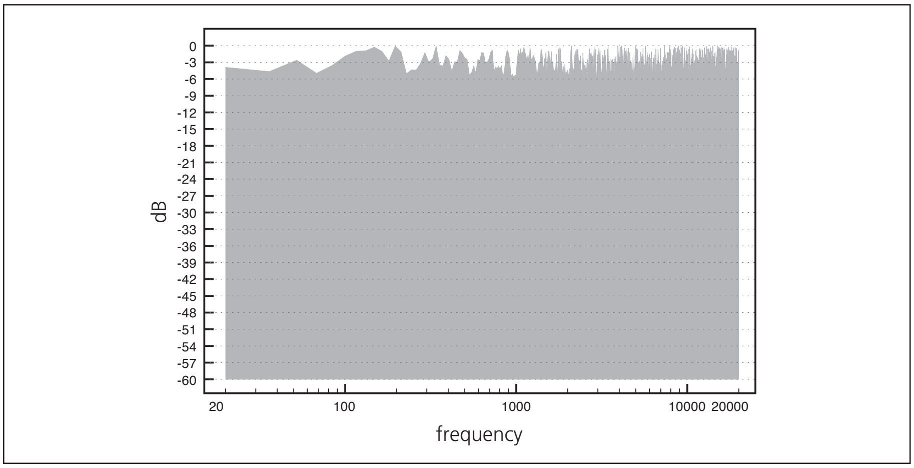


Fig. 3.1 The spectrum of white noise

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.

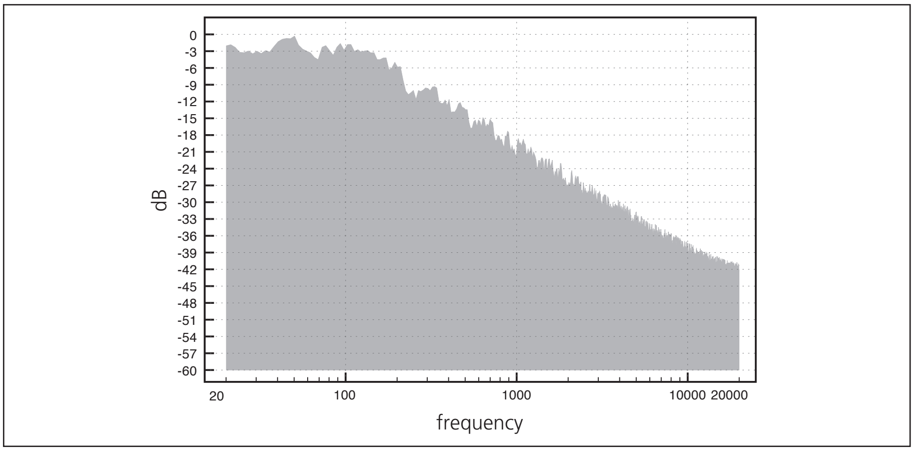


Fig. 3.2 The spectrum of pink noise

[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.

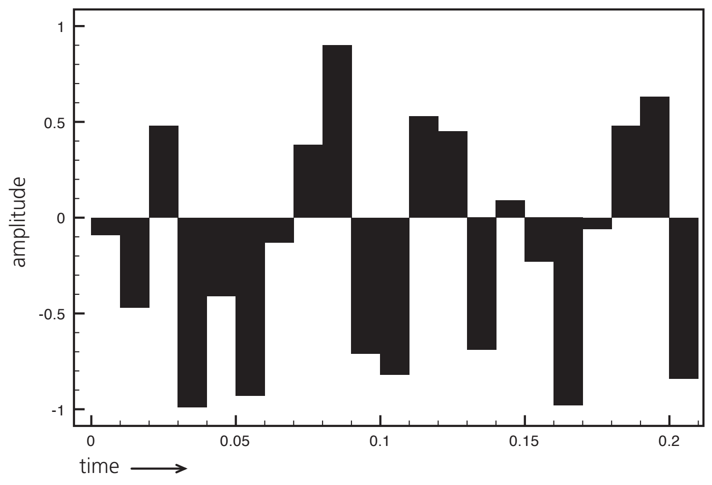


Fig. 3.3 Generation of pseudo-random values

> **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.

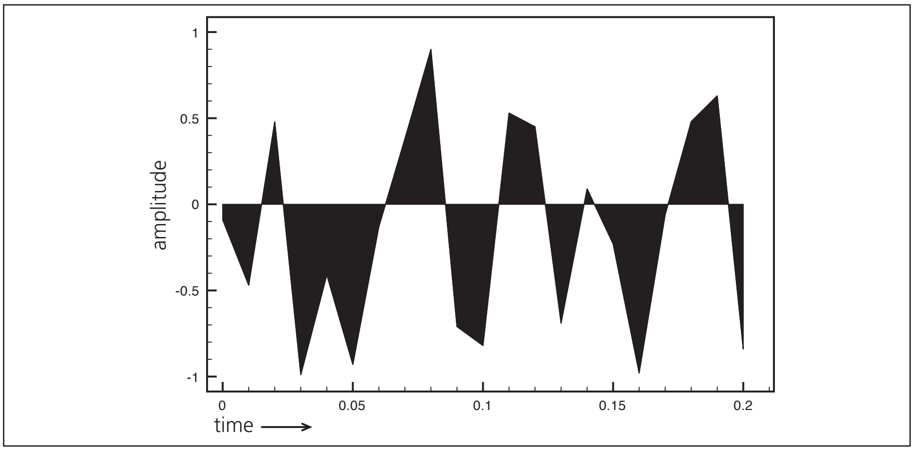


Fig. 3.4 Generation of pseudo-random values with linear interpolation

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.

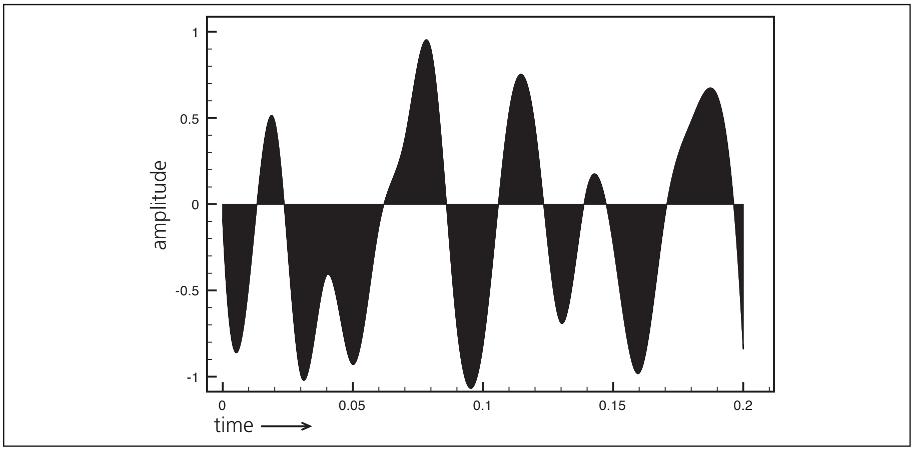


Fig. 3.5 Generation of pseudo-random values with polynomial interpolation

> **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:

3P
NOISE GENERATORS, FILTERS, AND SUBTRACTIVE SYNTHESIS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- Contents of Chapters 1 and 2 (Theory and Practice), Chapter 3 (Theory), Interlude A

LEARNING OBJECTIVES

Skills

- To learn how to generate and control different types of complex signals for subtractive synthesis (white noise, pink noise, impulses, etc.)
- To learn how to construct algorithms using lowpass, highpass, bandpass, bandreject, shelving, and resonant filters, and also how to control them using various parameters, Q, and filter order
- To learn how to implement FIR (non-recursive), as well as IIR (recursive), filters
- To learn how to build a simple subtractive synthesizer
- To learn how to write algorithms using filters connected in series and in parallel
- To learn how to 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, bandreject, shelving, and resonant filters
- The Q factor and filter order
- Finite impulse response and infinite impulse response filters
- Connecting filters in series and in parallel
- Graphic and parametric equalization

ACTIVITIES

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

TESTING

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

SUPPORTING MATERIALS

- List of Max objects
- Attributes for specific Max objects

3.1 SOUND SOURCES FOR SUBTRACTIVE SYNTHESIS

As you learned in Section 3.1T, the purpose of a filter is to modify the spectrum of a signal in some manner. In this section, before introducing filtering in Max, we will introduce an object that you can use to display a spectrum: the **`spectroscope~`** object. This graphical object can be found in the "Audio" category of the Add *palette* (as shown in figure 3.1). As usual, if you can't find it in the Add *palette*, create an object box and type the name "spectroscope~" into it.

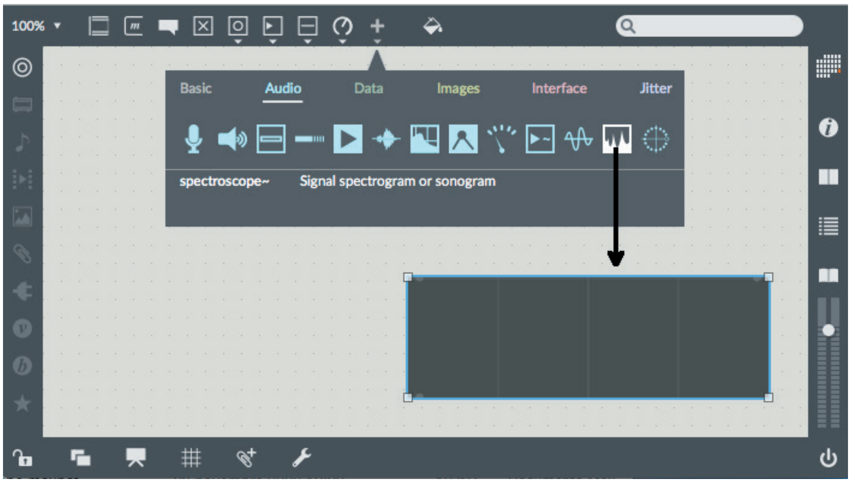


Fig. 3.1 The `spectroscope~` object

Open the file **03_01_spectroscope.maxpat** (shown in figure 3.2).

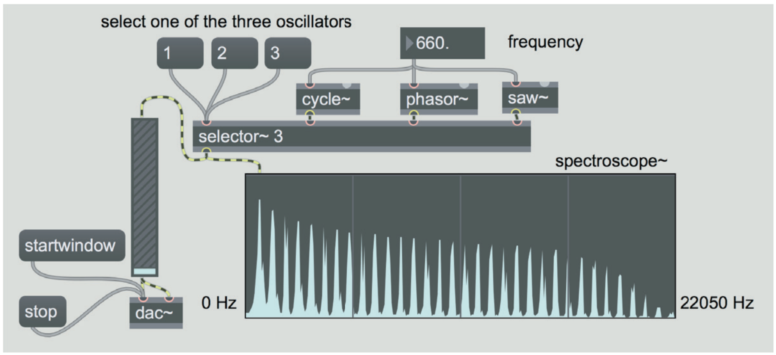


Fig. 3.2 The file 03_01_spectroscope.maxpat

In this patch, we have connected a `selector~` object to the spectroscope so that we can switch between three oscillators, one generating a sine wave (`cycle~`), one generating a non-band-limited sawtooth wave (`phasor~`), and one generating a band-limited sawtooth wave[1] (`saw~`).

[1] See Section 1.2.

The three message boxes connected to the left input of the **selector~** object are used to pick one of the three oscillators; by setting the frequency in the float number box and then moving between the three oscillators, you can compare their spectra. Observe that the sine wave has only one component, while the **phasor~**, being a non-band-limited object, has a much richer spectrum.[2]
Make sure to try several frequencies for all of the waveforms, while watching the image being produced by the spectroscope.
As stated, the spectrum of the sound being input is what is being displayed: the components of the sound are distributed from left to right across a frequency band that ranges from 0 to 22,050 Hz by default. These two values, the minimum and maximum frequencies that the spectroscope can display, can be modified in the *Inspector*, using the "Lo and Hi Domain Display Value" attribute in the *Value* category.
Try adding a **spectroscope~** object to a patch that you have already created, so to familiarize yourself with the relationship between sound and its spectral content. You might add the object to the patches found in previous chapters. In 01_14_audiofile.maxpat, for example, you might try connecting the spectroscope to the left outlet of the **sfplay~** object, or in IA_06_random_walk.maxpat, you might try connecting it to the outlet of [**p** monosynth]. (In this last patch, are you able to see the relationship between the frequency of the sound and the shape of the spectrum? Try preset number 5.)
Let's move on to a discussion about ways to produce various types of noise in Max. The first, white noise, is generated using the **noise~** object (as shown in figure 3.3).

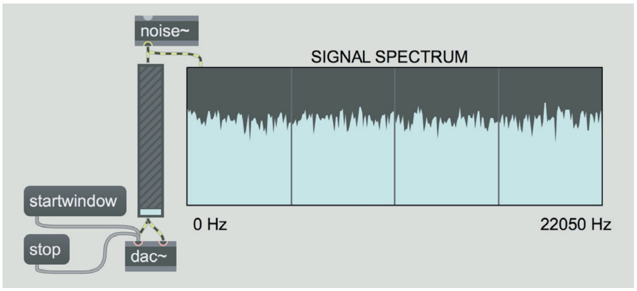


Fig. 3.3 The white noise generator

In the figure (which we encourage you to recreate on your own) we have connected the noise generator to a **spectroscope~** object, through which we can see that the spectrum of white noise contains energy at all frequencies. Unlike the sound generators that we have already encountered, the white noise generator doesn't need any parameters; its function is to generate a signal at the sampling rate consisting of a stream of random values between -1 and 1 (as we discussed in Section 3.1T).

[2] The **spectroscope~** object uses an algorithm for spectral analysis called the Fast Fourier Transform. We already mentioned the Fourier theorem at the end of Chapter 2T, and we will return to a much more detailed discussion of it in Chapter 12.

The second type of noise generator that is available in Max is the **pink~** object, which, unsurprisingly, generates pink noise (as seen in figure 3.4).

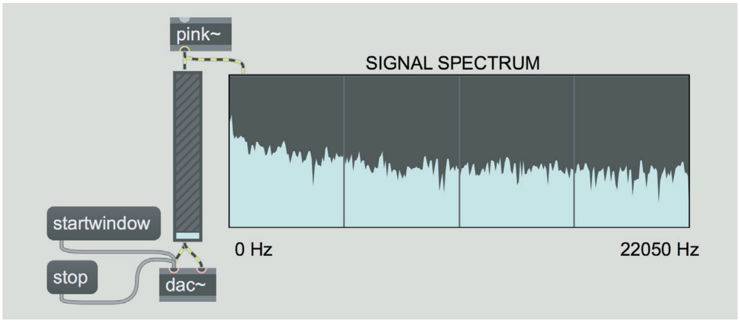


Fig. 3.4 Pink noise

Note that the spectrum of pink noise, unlike white noise, is gradually attenuated as frequencies get higher, and that this attenuation (as we know from Section 3.1T) is 3 dB per octave.
Reconstruct 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 an oscilloscope (scope~) to the two patches just built, remembering to set the "Calcount - samples per pixel"[3] attributes in the *Value* category of the Inspector, and observe the difference between the waveforms of pink noise and white noise.
In figure 3.5, we see these two waveforms side-by-side.

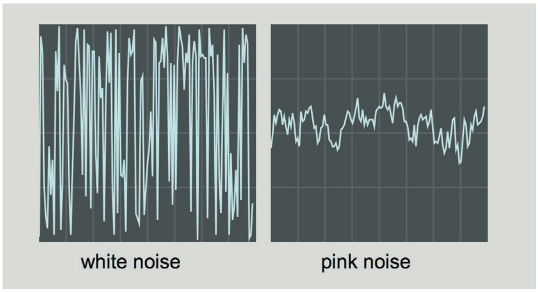


Fig. 3.5 Waveforms of white noise and pink noise

Without bogging down in technical details, you can see that 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

[3] We explained this attribute in Section 1.2P.

next is larger, the energy of the higher frequencies in the signal is greater.[4] As you know, white noise has more energy at higher frequencies than pink noise.

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

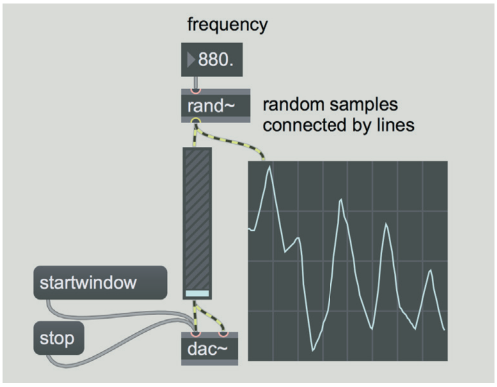


Fig. 3.6 The rand~ object

Obviously, this generator produces a spectrum that varies according to the frequency setting; it shows a primary band of frequencies that range 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.

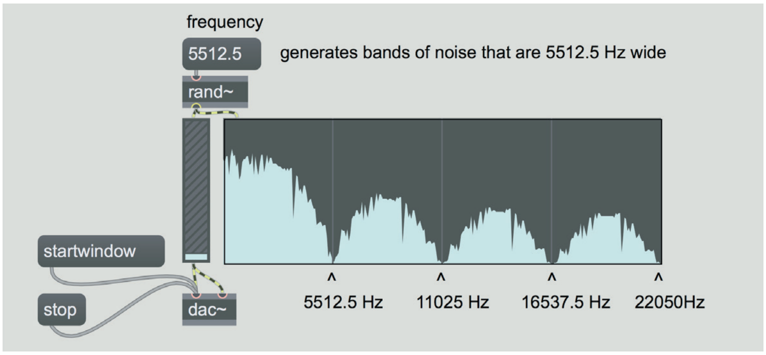


Fig. 3.7 The spectrum generated by the rand~ object

[4] 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.

In the example, we can see that the frequency of the `rand~` 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 `rand~` 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 wide bands, both 11,025 Hz wide.

Another noise generator is **`vs.rand0~`**[5] (the last character before the tilde is a zero), which generates random samples at a given frequency like `rand~`, but that doesn't interpolate. Instead, it maintains the value of each sample until a new sample is generated, producing stepped changes in value. The spectrum of this object is divided into bands in the same way as that of `rand~` was 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.

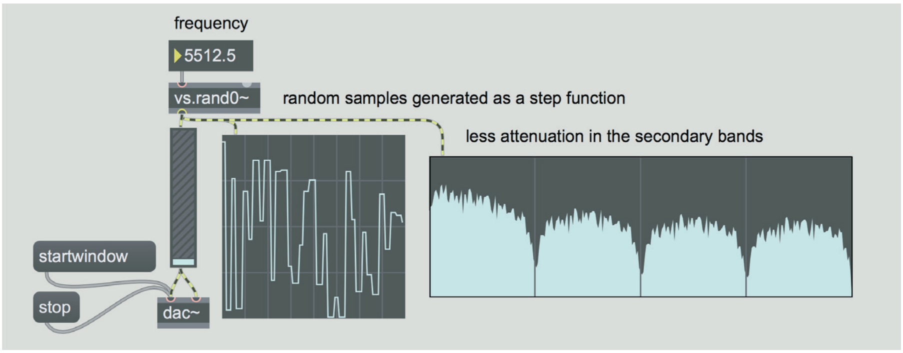

frequency
5512.5
vs.rand0~
random samples generated as a step function
less attenuation in the secondary bands
startwindow
stop
dac~

Fig. 3.8 The `vs.rand0~` object

In the *Virtual Sound Macros* library, we also have provided a noise generator that uses cubic interpolation called **`vs.rand3~`** (as shown in figure 3.9).

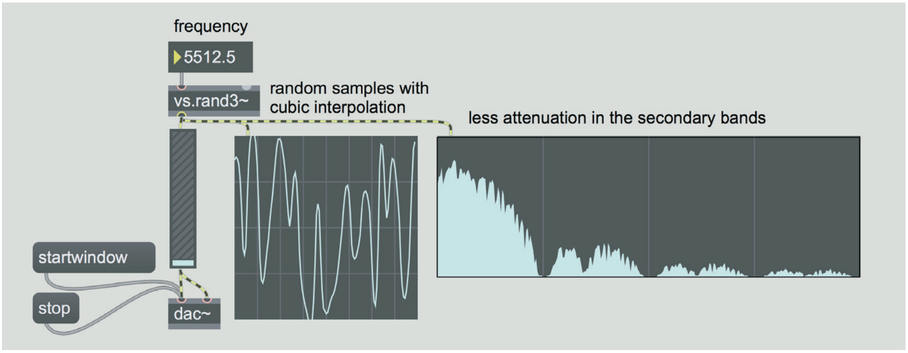

frequency
5512.5
vs.rand3~
random samples with cubic interpolation
less attenuation in the secondary bands
startwindow
stop
dac~

Fig. 3.9 The `vs.rand3~` object

[5] You should know from preceding chapters that the "vs" at the beginning of this object's name means that it a part of the *Virtual Sound Macros* library.

Thanks to the polynomial interpolation in this object, the transitions between one sample and the next 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 another source of sounds 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 these waves: `saw~`, `rect~`, and `tri~`. We will use these oscillators frequently in 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 `sfplay~` object (first introduced in Section 1.5P).

(...)

other sections in this chapter:

3.9 OTHER SOURCES FOR SUBTRACTIVE SYNTHESIS: IMPULSES AND RESONANT BODIES

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

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

SUPPORTING MATERIALS
- List of Max/MSP objects - Commands, attributes, and parameters for specific Max/MSP objects

Interlude B
ADDITIONAL ELEMENTS OF PROGRAMMING WITH MAX

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

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

LEARNING OBJECTIVES

Skills

- To learn how to use simple MIDI objects and signals
- To learn how to implement recursive operations in Max
- To learn how to use Max's conversion operators
- To learn how to build an arpeggiator that exploits probabilistically generated intervals
- To learn how to route signals and messages to switched inlets and outlets
- To learn how to compare values using relational operators
- To learn how to take apart lists of data
- To learn how to program repeating sequences using iterative structures
- To learn how to generate random lists for simulating resonant bodies
- To learn how to implement Shepard tone, or "infinite glissando"

CONTENTS

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

ACTIVITIES

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

TESTING

- Integrated cross-functional project: reverse engineering

SUPPORTING MATERIALS

- List of Max objects
- Messages, and attributes for specific Max 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" synthesizer, for example. Using the MIDI protocol, the computer sends notes to the synthesizer along with the intensity with which they should be played, their durations, and other information.
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, just as real-world instruments do. Programs like Max can send MIDI commands directly to a receiving program such as a virtual instrument. Indeed, Max has many objects that exploit the MIDI protocol, as we will learn in more detail in Chapter 9, but for the moment, we will stick to a basic subset that is used to manage MIDI messages.
Open the file **IB_01_MIDI_note.maxpat**; figure IB.1 shows the upper part of the Patcher Window.

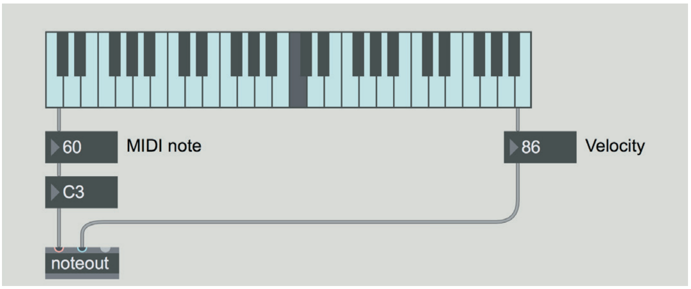


Fig. IB.1 The upper part of the patch contained in IB_01_MIDI_note.maxpat

We have connected the `kslider` object (the musical keyboard) to some number boxes, which then connect to a **`noteout`** object. As we have already learned, clicking on one of the `kslider` 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 an intensity for the note; clicking on the upper part of the key will produce a higher velocity value, while clicking on the lower part will produce a lower value. (On physical keyboards, velocity actually reflects the velocity with which the key is pressed, hence the origin of the term.) Values for velocity can vary between 1 and 127 in 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.[1] In the MIDI protocol, this message is

[1] The right inlet of the `noteout` object is used to set the MIDI channel, which we don't need at the moment. The details of this will be forthcoming in Chapter 9.

defined to be the "***note-on***" command. When you click close to the top of a kslider key, generating a high enough velocity value (let's say above 90), you should hear the sound of a piano. This sound is not coming from Max, but instead, 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 noteout plays a sound for every new note that it receives, 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 kind of "remove finger from key" command.)

One way to properly "extinguish" a MIDI note generated by kslider is to change the way that the object manages MIDI notes. In edit mode, call up the Inspector for the upper kslider, in the "Value" category change the ***Display Mode*** parameter value from "**Monophonic**" to "**Polyphonic**". Then return to performance mode. The first time that you click on a kslider key, the note will sound at the velocity chosen, and the second time that you click on the same key, the note will be triggered again, but this time with a velocity of 0, which will make the sound stop: try this. Besides adding the handy note-stopping feature, this also now enables you to activate more than one note at the same time: the performance capability has become polyphonic.
Now turn your attention to the lower half of the patch in **IB_01_MIDI_note.maxpat.**

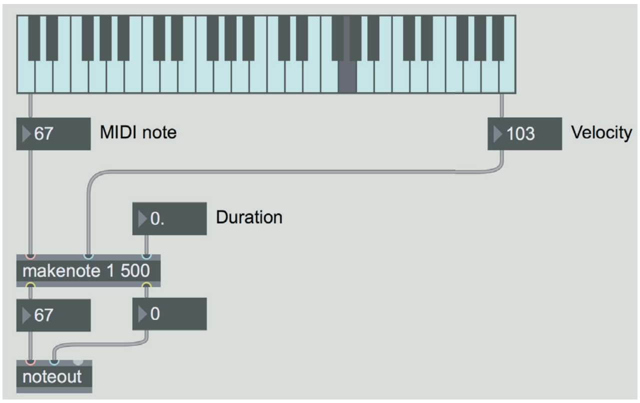


Fig. IB.2 The lower part of the patch in IB_01_MIDI_note.maxpat

We have connected the kslider object (in monophonic mode) to the **makenote** object, which generates a MIDI *note-on/note-off* command pair. Every time this object receives a *note-on*, it generates a *note-off* after a configurable 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 (which is, of course, the amount of time to pass before the *note-off* command is sent). It has two outlets, one for MIDI note number and the other for velocity.

There are two arguments associated with **makenote**, a velocity and a duration in milliseconds. In the patch, we have set velocity equal to 1 (a placeholder, as we will see), and duration equal to 500 milliseconds (or half a second). When the object receives a *note-on*, it will send the pitch and velocity of the *note-on* directly to its outlets, where, after the prescribed duration (500 milliseconds in this example) a *note-off* is also sent. Note that the velocity sent by the **kslider** (which in the example shown in the figure is a value of 103) overrides the value of 1 that had been provided as an argument. We don't need to use the first argument, and so we will simply provide a placeholder for its value. The second argument, the duration, can also be modified by sending a value to the right inlet, which will replace the value originally specified in the second argument.
Try playing some notes and changing the duration: observe how velocity values first reflect the value generated by **kslider**, and then, after the time specified by the duration parameter, they change to 0.

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

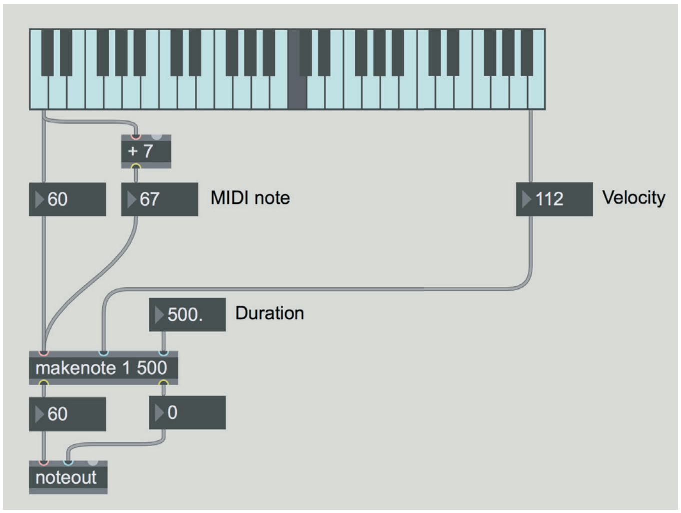


Fig. IB.3 Transposing MIDI notes

This patch is similar to the one in the file IA_01_transposition.maxpat, which we examined in the first section of Interlude A. In this case, every key pressed on the keyboard generates two notes simultaneously, separated by a distance of 7 semitones: the interval of a fifth. Every time that a **kslider** key is pressed, in fact, the value of the MIDI note number is sent to the **makenote** object and also to the addition object that adds 7 to the first note number and then sends its output to **makenote**. To create these note pairs, we didn't need to repeat velocity and duration values, since these values are sent to "cold" inlets of the **makenote** object, which are stored as internal variables (the contents of internal variables are re-used every time that a new value arrives at the "hot" inlet). In the figure, for example, both notes (the middle C and the G above it) will have a velocity of 112 and a duration of 500 milliseconds.

From this example, you can see that Max's rule about "hot" inlets being on the left is complementary with right-to-left execution, as discussed in Section 1.7. 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 set up.

(...)

other sections in this chapter:

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 multi-channel 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 multi-channel 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).

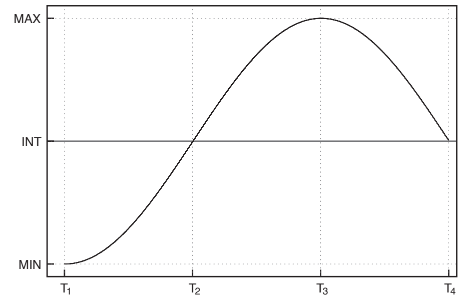


Fig. 4.1 An LFO for controlling the position of a sound in space

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:

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

- To learn how to move a sound within a stereo field
- To learn how to implement vibrato
- To learn how to simulate instruments whose frequency is controlled, such as a theremin
- To learn how to implement tremolo
- To learn how to build pulse width modulation algorithms
- To learn how to vary cutoff frequency, center frequency, and Q of filters using oscillating control signals
- To learn how to use pseudo-random signal generators for control
- To learn how to 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
- Positioning and moving sound in stereo and multi-channel systems
- Modulating control oscillators with pseudo-random LFOs

ACTIVITIES

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

TESTING

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

SUPPORTING MATERIALS

- List of Max objects
- Attributes for specific Max objects
- Glossary

4.1 CONTROL SIGNALS: STEREO PANNING

You can use the output of a normal **cycle~** 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 **cycle~** object, in this case, should be low enough to be below the threshold of human hearing. You learned how to parameterize stereo position in Section 1.6,[1] and you can begin your work here by extracting the algorithm for positioning a sound in a stereo field from the file 01_18_pan_function.maxpat. (You'll wind up with the parts of the original patch that were connected to the **line~** object, as shown in Figure 4.1.)

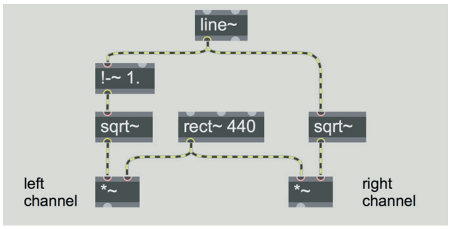


Fig. 4.1 A panning algorithm

You need to replace the **line~** object from the original patch (which modulated the position of the sound in space by using line segments) with a sine wave generator oscillating between 0 and 1. (A value of 0 will pan the sound left, while a value of 1 will pan the sound right.) The **cycle~** object, however, generates a sine wave that oscillates between -1 and 1. You could modify this oscillation interval by using the pair of simple calculations that you learned about in the theory chapter, but this will be the subject of the next section. For now, employ the **scale~** object, introduced in Section IB.8, to rescale the signal, completing the patch as shown in Figure 4.2.

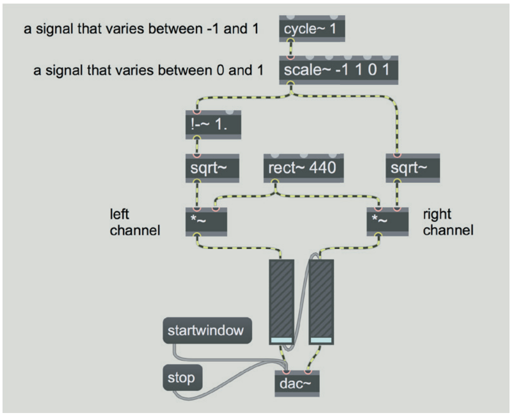


Fig. 4.2 Stereo panning controlled by an LFO

[1] If you don't remember how to do this, refresh your memory by rereading the relevant sections of both theory and practice.

In this patch, the **line~** object has been replaced by the **scale~** object, and a **cycle~** object has been connected to it. As you know, the **scale~** object takes four arguments, the first two specifying the range of the input signal, and the last two specifying the desired range of the output signal. In our case, the arguments [-1 1 0 1] indicate that we will be feeding the object an input signal that ranges from -1 to 1, and that we want to rescale this input to fit a signal that ranges from 0 to 1. The **cycle~** object itself is set to generate a control signal of 1 Hz, which will make the sound travel from the left to the right and back again over the period of one second; by connecting a float number box to the **cycle~** object, you can change the oscillation frequency. Try the patch with various frequencies, but stay below 20 Hz; higher frequencies will generate interesting audible modulation anomalies that we will take up in Chapter 10.

At this point, you can simplify the patch by using the **vs.pan~** object from the Virtual Sound Macros library, an object that implements a stereo panning algorithm; the object takes the sound to be positioned on its left inlet, and the positioning control signal on its right inlet. (See Figure 4.3 for the simplified patch.)

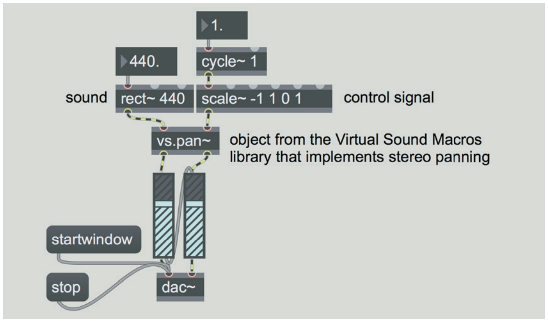


Fig. 4.3 Stereo panning using the **vs.pan~** object

You can see that the **vs.pan~** object performs the same function as the algorithm in Figure 4.1. We are using it simply to free up room in the graphical display of our patch.
Try this patch, substituting control signals made with other waveforms, such as the square wave shown in Figure 4.4.

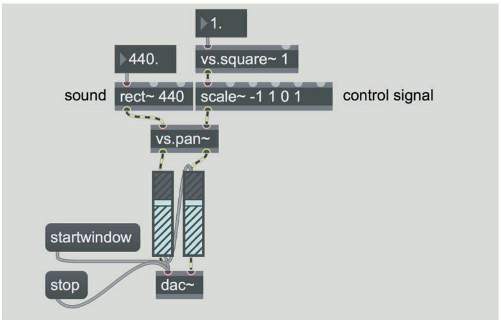


Fig. 4.4 Controlling panning with a square wave LFO

Under the control of a square wave, 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 lowpass filter, which smooths the sharp corners of the square wave. (See Figure 4.5 for this modification.)

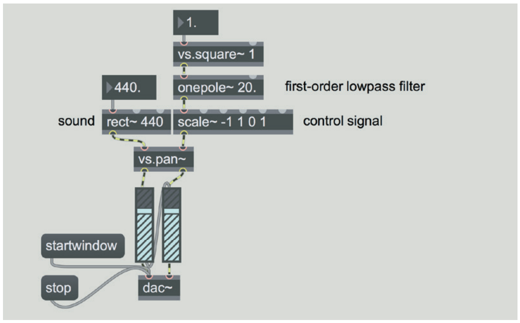


Fig. 4.5 Filtering an LFO

In this patch, we have set a cutoff frequency of 20 Hz, which means that the control signal can't 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.

(...)

other sections in this chapter:

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

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

SUPPORTING MATERIALS
- List of Max/MSP objects
- Commands, attributes, and parameters for specific Max/MSP objects

- Glossary

Alessandro Cipriani • Maurizio Giri

Electronic Music and Sound Design

Theory and Practice with Max 7 • 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 - Max and MSP Programming Techniques

"This book is one of the first courses on electronic sound that explicitly integrates perception, theory, and practice using examples of real-time sound synthesis you can manipulate and experience for yourself. In my opinion, Cipriani and Giri have done a masterful job of allowing experiential and theoretical knowledge to reinforce each other. This book will work either as a textbook or as a vehicle for the independent learner. As a bonus, the book includes a thorough introduction to digital signal processing with Max and serves as a wonderful introduction to the programming concepts in that software. I hope you will take advantage of the excellent Max examples the authors have created. They are simultaneously fun and enlightening, and they sound good enough to use on stage. They are also worth examining as models for your own Max patches, or for extending in new ways. As you will see, the theoretical chapters are the "T" chapters, while practical and experiential knowledge is imparted by the "P" chapters. These chapters alternate, in the form of a ladder, refining the concepts at ever higher levels of sophistication. I want to wish you good luck on this new adventure, and also thank my two Italian friends for creating such a comprehensive resource for learning about digital music – the one I wish existed when I was a student!"
(from the Foreword by David Zicarelli, publisher of Max).

This is the first of a series of three volumes dedicated to digital synthesis and sound design. It is ideal for those who are newcomers to the field, but it will also prove to be an indispensable resource for those who wish to deepen existing skills in sound design, in electronic music, or in Max.

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 Perugia 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 company that develops music applications and plug-ins for mobile devices and computers. He was artist in residence at the Cité Internationale des Artes in Paris, and at GRAME in Lyon. He collaborated with members of the Nicod Institute, a research center of the École Normale Supérieure in Paris, on a project about the philosophy of sound.

ISBN 978-88-99212-02-5

9 788899 212025

www.contemponet.com