Alessandro Cipriani • Maurizio Giri

Electronic Music and Sound Design

Theory and Practice with Max and MSP • volume 2

ConTempoNet

This is a demo version of

ELECTRONIC MUSIC AND SOUND DESIGN
Theory and Practice with Max and MSP - volume 2
by Alessandro Cipriani e Maurizio Giri

Alessandro Cipriani • Maurizio Giri

ELECTRONIC MUSIC AND SOUND DESIGN

Theory and Practice with Max and MSP - Volume 2

Cipriani, Alessandro. Giri, Maurizio.
Electronic Music and Sound Design : theory and practice with Max and MSP. Vol. 2.
/ Alessandro Cipriani, Maurizio Giri.
Includes bibliographical references and index.
ISBN 978-88-905484-4-4
1. Computer Music - Instruction and study. 2. Computer composition.

Original Title: Musica Elettronica e Sound Design - Teoria e Pratica con Max e MSP

Translation by Richard Dudas

First edition 2014

Audio and Interactive Examples: Vincenzo Core
Index: Salvatore Mudanò

ConTempoNet s.a.s., Rome (Italy)
e-mail **posta@contemponet.com**
posta@virtual-sound.com
URL: **www.contemponet.com**
www.virtual-sound.com
facebook **www.facebook.com/electronic.music.and.sound.design**

CONTENTS

FOREWORD TO THE SECOND VOLUME

by Richard Boulanger

With their *Electronic Music and Sound Design: Theory and Practice with Max and MSP* the master teachers and composers – Alessandro Cipriani and Maurizio Giri have produced a series of "interactive and enhanced books" that present the student of computer music with the finest and most comprehensive electroacoustic curriculum in the world. By "illustrating" the text with a wealth of figures and clearly explained equations, they take the reader "under the hood" and reveal the algorithms that make our computing machines "sing". By using David Zicarelli's incredibly powerful and intuitive media-toolkit – *Max* to create hundreds of synthesis, signal processing, algorithmic composition, interactive performance, and audio analysis software examples, Cipriani and Giri have provided the means for students to learn by hearing, by touching, by modifying, by designing, by creating, and by composing. On page after page, and with Max patch after Max patch, they brilliantly guide the student to a deeper knowledge and understanding that is guaranteed to release their musical creativity in new and profound ways.

As we all know, digital cameras are so "smart" today that it is virtually impossible to take a bad picture. But how to frame and freeze a moment in time, and then to have that frozen moment "speak" through time – no camera can do that. A "photographer" does that. And it takes a great teacher, a great mentor to help a student "see" what is right before their eyes. How does a great teacher do this? They practice what they preach, and they teach by example. This is exactly what Cipriani and Giri do in this series. *Electronic Music and Sound Design* is filled to overflowing with working and teaching examples, models, and code. It is a treasure chest of riches that will enlighten and inspire the 21st century musician, audio artist, and designer to make the most of their "instrument" – the computer itself. They are teaching the next generation how to play it!

Today, brilliant design provides us with intuitive tools and systems that "anyone" can make work; but understanding how they actually work, and understanding how one might actually work with them – that is the challenge. Innovation doesn't spring from accidents and good luck. For sure, turning knobs can produce some crazy sounds, but a collection of crazy sounds is far from musical. As Varese would say, "music is organized sound". I would humbly expand this by saying that "music is structured sound", "music is sculpted sound"; music is the manifestation and articulation of "thought forms" that we resonate with and share, "mind models" that spring fourth from "sound understanding". The masterpieces of tomorrow's Audio Art will reveal a vision that comes into focus as today's students grow in their appreciation and understanding of how things work and how to work with them, and Cipriani and Giri are paving the way for an age of audio enlightenment.

I firmly believe that this series by Cipriani and Giri, these "interactive and enhanced books", in which definition and design, in which theory and practice, in which compositional advice supported by an analysis of historical master-

works are all so tightly coupled with audio examples and editable and working Max and Max for Live patches, set the stage for the next generation of innovators. This book is essential for young and creative computer musician who have the tools and want to know how they work and how to work with them. In *Electronic Music and Sound Design*, Cipriani and Giri feed the hands, they feed the ears, and they feed the minds of the students in ways and to a degree that no computer music textbook has ever done.

Volume 1 moved from basic audio to software synthesis, filtering, spatialization, and some MIDI, with a great introduction to Max and a good assortment of Max tricks. Along the way, they introduced and covered a lot of synthesis and signal processing theory. The book is structured with a "theoretical and practical" chapter to be studied in parallel. A unique collection of Max patches is provided, in "presentation" mode, so that the theoretical concepts can be "explored". There is audio ear training, chapter tests, activities, suggested projects and modifications, and a unique glossary of terms, at the end of each chapter. In fact, each chapter begins with a set of "learning objectives" and "competencies", and a list of prerequisites (usually the contents of the previous chapters). The chapters are filled with exercises, activities, assignments and end with a quiz. It is a great curricular model. I am a particular fan of Chapter 3 on Noise, Filters, and Subtractive Synthesis as it is a great balance of practical, mathematical and theoretical. The "practical" chapters feature full Max Patches ready to be modified, repaired, expanded, and explored. In addition to the theoretical and practical chapters, there are two "Interludes" that focus on Max programming. In addition to all the patches featured in the "text", Cipriani and Giri provide a huge library of abstractions (they call them "macros") that make programming and design even more efficient. There are many solutions, optimizations, and tricks revealed in this collection too, and it is worth some serious study as well. It is truly amazing how much computer music you are learning and how much Max you are learning – at the same time!

Volume 2 is structured much like Volume 1 – starting each with chapter objectives and outcomes and ending with a quiz and a chapter-specific glossary of terms. In fact, it picks up where the first volume left off – starting with chapter 5! In general it features a more in depth coverage of topics and builds on what was learned in Volume 1. By this point, the student is more advanced in their understanding and skills and so more depth is presented and more difficult challenges are assigned. As might be expected, in this volume three "Interludes" take the reader even deeper into Max with a focus on time and sequencing, advanced preset, data, time, polyphony, and score management and the idiosyncrasies of working with bpatchers, concluding with a major interlude focusing on Max for Live and the Live API that helps the reader to move all their studies into a rich and robust production and performance environment. As in Volume 1, the chapters are again organized in pairs with a theoretical chapter supporting a practical chapter.

Chapter 5 focuses on Digital Audio and Sampling and features some really exciting "sample-cutters" and "scrubbers". Chapter 6 focuses on Delay Lines

and associated effects such as comb-filtering and pitch shifting and culminating in delay-line based synthesis – the famous Karplus-Strong plucked string algorithm. Lots of great sounds here. Chapter 7 focuses on Dynamic Processors, Envelope Followers, Compressors, Limiters, and their creative use – such as side-chaining. There are a lot of practical and useful performance tools here.

The game changer in the series, and the masterpiece of this book is what is covered in Chapter 8. Here Cipriani and Giri begin to teach the reader about the world of computer music and how to "speak" the language with some fluency – how to compose Audio Art. It is titled: "The Art of Organizing Sound: Motion Processes". In it Cipriani and Giri present and analyze a number of masterworks, link to them on their website, and showcase some of the processes that define their uniqueness. Further, the compositional approach, and the aesthetic ideas of a number of innovative composers is cited. This chapter is not only filled with musical models, but also filled with some wonderful role models – including the inspiring compositions of Cipriani and Giri themselves. This chapter is so important at this point in the "course" because it establishes context and sets the stage for more expanded compositional work with the techniques that have been learned and the systems that have been built. This is where Cipriani and Giri teach the student to "see".

Finally, Chapter 9 focuses on MIDI and gives us a deeper and more complete review of the MIDI spec and shows the ways that this knowledge and these messages can be applied in Max. This chapter sets the stage for the final Interlude that focuses on the incredibly important Max for Live application of the work thus far.

And so…

Volume 1 was fantastic. Volume 2 raises the bar and brings insights into the compositional process, new ideas on working with "time-forms", and new ways to integrate signal processing and synthesis algorithms into a powerful performance and production tool via Max For Live. I can't wait for Volume 3! Until then, I will close by saying that I am deeply honored to be associated with this great pedagogical milestone and to write the foreword for Volume 2. Moreover, I am so happy for all the students around the world who will learn so much from working their way through the text, the examples, the music, the quizzes, the projects – all under the guidance of these great teachers – Alessandro Cipriani and Maurizio Giri.

Max is the brilliant and inspired artistic creation of David Zicarelli and Miller Puckette. This software has revolutionized the field of computer music and made it possible for "musicians" to write software; for "musicians" to develop their own custom interactive systems. As such Max has liberated the artist, and revolutionized the field of computer music, and made possible the most incredible, diverse, and profound musical creations and performances.
For many years now, the international community of Max users, developers, and teachers has grown. Their numbers are vast, and the work that they have

created and shared has been so inspiring; but to date, there has never been a full synthesis, signal processing, composition and production curriculum built on top of Max – not until now. The series of "interactive and enhanced books" under the title *Electronic Music and Sound Design: Theory and Practice with Max and MSP* clearly establishes Alessandro Cipriani and Maurizio Giri as two of the greatest and most important teachers of computer music in the world.

Dr. Richard Boulanger
Professor of Electronic Production and Design, Berklee College of Music
Author and Editor of The Csound Book & The Audio Programming Book – MIT Press

TRANSLATOR'S NOTE

by Richard Dudas

Back in the days when Max had just a few dozen objects and a relatively restricted range of what it was capable of doing compared with the program today, it nonetheless still seemed like a limitless environment. Its basic object set could be employed, arranged and rearranged in countless ways to build whatever one's imagination desired. It was an ideal tinkerer's toolkit – much like the popular crystal radio sets and erector sets of the 50s and 60s, the more modern construction sets made of interlocking plastic pieces which first appeared in the 60s and 70s, or the basic microcomputer systems from the 80s. When Max came along at the dawn of 1990s, its "do it yourself" paradigm was perfectly suited to the creative and eager musician in the MIDI-based home studio in an era when much of the available commercial software had highly limited functionality. Max offered musicians and sound artists the ability to create their own software to "go outside the box" without needing to learn the intricacies of a textual programming language nor the mundane specifics of interfacing with the computer's operating system.

Since that time, the Max environment has continued to grow and evolve from a program geared toward interacting with MIDI and simple media to one that encompasses audio and video processing and connections to external software and hardware. But in getting bigger, the sheer magnitude of features available within it has caused it to become rather daunting for many musicians, even though they may actually be keen to discover what it has to offer. Furthermore, the instruments and audio effects that have become prevalent since digital audio workstations moved from the studio to the home studio have become increasingly more complex, so understanding their inner workings has consequently also oftentimes become mystifying. That is where this series of books by Alessandro Cipriani and Maurizio Giri comes to the rescue.

This series of books provides a straightforward, musically-oriented framework to help new users get into the program and at the same time effortlessly learn the theory behind each of the topics they are studying. It also helps intermediate and advanced level students and professionals better grasp concepts they may already be acquainted with. In addition to presenting a progressive series of compelling musical tools and explaining their theoretical underpinning, it also supplies useful pre-fabricated high-level modules in instances where none readily exist in Max. Since these modules are provided as patches, they can be taken apart, analyzed, modified and learned from, or simply used as-is, depending on the user's level of familiarity with the program. Most importantly, the books in this series do not attempt to teach every esoteric detail and object that is available in the environment – that is a good thing! – they concentrate on shedding light on those fundamental notions and tools (and some more advanced ones, too) that are immediately necessary to help users understand what they are doing and get started using the program creatively and practically.

I have been a devoted and passionate user of Max nearly since its inception, was fortunate enough to be a beta-tester of early versions of the program, and later also worked as one of the developers of the software at Cycling '74. Max has been an important part of my personal creative musical output (and sometimes also my free time on the weekends!), and has additionally been central to my work as educator in the field of computer music. Now, while working in the rôle of translator for this second volume, I have discovered (via both Cipriani and Giri's admirable text as well as the Max program itself) that there are still new ideas to contemplate, new information to absorb, new techniques to amass and many alternate ways to design and improve commonly used algorithms for sound processing and synthesis. For me, this is one of the most amazing aspects of Max, and indeed of music and the arts, in general. From my perspective as an educator, this book is everything I would hope for, and more – its very strength is that it offers the reader technical knowledge alongside compelling artistic and creative motivations for using open-ended software such as Max instead of encouraging blind reliance on commonplace off-the-shelf tools, however seductive their sound may initially seem. Thus, I am both happy and proud to have been able to play a part in bringing this excellent series of books, written from a decidedly *musical* perspective, to a wider audience.

Richard Dudas
Assistant Professor of Composition and Computer Music, Hanyang University School of Music

INTRODUCTION TO THE SECOND VOLUME

This book is the second in a series of three volumes dedicated to digital synthesis and sound processing. The first volume of the series covers a variety of topics including additive synthesis, noise generators, filters, subtractive synthesis and control signals. The third volume will cover reverberation and spatialization, various techniques for non-linear synthesis (such as AM, FM, waveshaping and sound distortion techniques), granular synthesis, analysis and resynthesis, physical models, procedural sound design and a second chapter dedicated to the organization of sound.

PREREQUISITES

All three volumes consist of chapters containing theoretical background material interleaved with chapters that help guide the user's practice of that theory via practical computing techniques. Each pair of chapters (theory and practice) work together as a unit and therefore should be studied alongside one another. This second volume has been designed for users with various levels of knowledge and experience, although they should already fully understand the concepts and use of Max which have been outlined in the first volume. The contents of this volume have been designed to be studied either by oneself or under the guidance of an instructor.

SOUND EXAMPLES AND INTERACTIVE EXAMPLES

The theoretical chapters of this book are meant to be accompanied throughout by numerous sound examples and interactive examples that can be downloaded from the Virtual Sound website at: **************. By referring to these examples, the user can immediately listen to the sound being discussed (in the case of sound examples), or discover and experiment with sound creation and processing techniques (with the interactive examples), without having to spend intervening time on the practical task of programming. In this way, the study of theory is always concretely connected to our experience and perception of both the sounds themselves, and the many possible ways they can be processed and modified.

MAX

The practical chapters of this book are based on the software Max 6, although Max 5 users can still use this text. We have made sure that all the patches and activities that are presented here can be realized with both versions. The sole object specific to Max 6 that we have used is scale~. For Max 5 users, we have included an abstraction that reproduces the functionality of scale~ on our support page at: **************. The patches, sound files, library extensions and other supporting material for this volume's practical activities can also be found on that page.

MAX FOR LIVE

The final chapter, or rather "interlude," of this book deals with Max for Live – an application that lets users create plug-ins for Ableton Live using Max. This is decidedly substantial chapter, in which all the knowledge learned over the

course of the first two volumes will be put to use to create devices (the term used for plug-ins in the Live environment). Special emphasis has been given to the discussion and study of the "Live API" which allows users to create devices which can be used to control other plug-ins or even the Live application itself.

TEACHING APPROACH AND METHOD OF THIS BOOK

As with the first volume, this second volume should be studied by reading each theory chapter in alternation with its corresponding practice chapter, in addition to carrying out the recommended computer-based activities. Nonetheless, one major difference, compared to the first volume, is in the type of practice activities which are suggested: the final activities of correcting, analyzing and completing algorithms, as well as substituting parts of algorithms for one another, are no longer present in this volume. Here, throughout each practice chapter, a copious selection of activities is presented to help the reader both test and deepen the various skills and knowledge that he has acquired thus far, in addition to suggesting ways of using them creatively. Throughout this volume the analysis of algorithms and patches is still carried out in detail (as it was in the first volume) when new techniques are being illustrated. However, where older, familiar processes and techniques are concerned, analysis is now left up to the reader. In other words, we have catered the second volume to a different type of reader. When writing the first volume, we were aware that our target reader was someone who, although thoroughly interested in the subject matter, could have been completely devoid of prior experience within the realm of electronic music. In this volume we can now presume that the reader is at an "intermediate" level – someone who has already made sounds with Max and/or with other software, and who knows the basics of synthesis and sound processing: in short, a reader who has already "digested" the material presented in the previous volume. Even those who have not yet read the first volume but possess the aforementioned skills will still be able to greatly benefit from this book, although we should point out that many of the concepts, objects and algorithms presented in the first volume will be referred to throughout the course of this text.

We would also like to point out the presence of a chapter in this volume titled "The Art of Organizing Sound: Motion Processes" (chapter 8 in both theory and practice). This chapter was designed to give the reader an opportunity to develop his own individual versions of the proposed activities in a more complex and creative way than in the first volume. This means that the reader will be encouraged to use his perception, analysis and critical thinking, in addition to his own experience and ingenuity. The importance of such a section dedicated to the creative use of one's knowledge and skills should neither be overlooked nor underestimated. Even though the software that we use may continue to evolve and change over time, the skills that we obtain through active personal practice and creation act as a flexible tool which can be applied in different technological contexts. It is our firm belief that a passive and "bookish" approach to learning is sterile and devoid of meaning, therefore our aim is to enable the reader to associate and interconnect his knowledge, skills, perception, analytic ability, ability to ask the right questions and to solve problems, and ability to create original musical forms with sound,

in a natural, inventive and personal way. The goal of this section is thus to impart how to work within one's own area of competence.

In order to be able to say one is competent in the field of electronic and computer music it is not enough simply to know how to create an LFO, for example, but more importantly how to use it and what to do with it in specific creative contexts. Indeed, simply knowing how to follow a series of steps is not an indication of competence; the expert also needs to know how to interpret those steps. Essentially, knowing how to provide the proper settings for a patch, or how to modify an object's parameters, not simply as an abstract task, but with the aim of achieving a certain goal for a sound's motion or evolution over time (or within in the listening space) is an essential ability for sound artists, sound designers and composers.

The reverse-engineering exercises in the first volume hinted at the possibility that the starting point for being able to use any given synthesis and sound processing system is not so much the theory behind it, as the actual context for which it will be used. In the case of the reverse-engineering exercises, the starting point was a pre-existing sound, specifically selected for the exercise at hand, whose properties and characteristics needed to be recognized in order to be able to simulate its spectrum, envelope, etc.…

In chapter 8 of this second volume, however, the theoretical knowledge and practical abilities that we have thus far developed will be put to use and further strengthened by focusing on the reader's own original sound processing skills and ability to construct motion processes. Nonetheless, the basic compositional activities that we propose in this chapter should be limited to sound forms not exceeding one minute in duration, and should be designed outside the bounds of a wider formal scope and context, such as that of a larger compositional project. That having been said, exactly what kind of sound creation is being proposed, here?

The scope of sound creation practices is both immense and diverse. It ranges from algorithmic composition to "laptop orchestras", from live electronics with human-machine interaction to acousmatic compositions. There are also soundscape compositions, sound installations, audio-visual installations, sound art, sound design work – the list is seemingly endless.
Not surprisingly, an infinite number of schools of thought have emerged, each with its own unique formal approach, ranging from narrative to abstract, or even in other directions such as forms of an ambient nature, etc.… It is therefore our desire to supply just a few pertinent tools to allow the reader to sharpen his own personal skills. We will also try to avoid providing rules and regulations as much as possible, but rather to try to propose a personalized experience for sonic discovery. Consequently, we have decided to reinterpret and adapt some ideas about spectromorphology, as proposed by Denis Smalley in some of his articles, for creative endeavors. Thus we are introducing the categories of simple motion, complex motion and compound motion. For the interaction between theory and practice, we suggest that each student interpret the type of motion being described, based on both the technical

information provided and the specific purpose for which it will be used, in order to be able to make use of it within his personal sounds.

SUPPORTING MATERIAL

All the material referenced throughout the course of this book can be downloaded from the Virtual Sound website's support page: **************.
In order to begin working with this text, you will first need to download all of the Sound Examples and Interactive Examples located on the support page. Bear in mind that you should constantly refer to these examples while reading through the theory chapters.

In order to work interactively with the practice chapters of this book, you will first need to install the Max program, which can be obtained at the site: www.cycling74.com. Once Max has been installed, you will also need to download and install the Virtual Sound Macros library from the support page mentioned above. The support page includes detailed instructions concerning the correct installation procedure for the library. Last but not least, the support page also includes the necessary patches (Max programs) related to the practice chapters of this book.

BIBLIOGRAPHY

As in the previous volume, the final pages of this book include a list of the most absolutely essential reference works, in addition to the bibliographical references cited throughout the course of the text itself.

COMMENTS AND CORRECTIONS

Corrections and comments are always welcome. Please contact the authors by e-mail at: a.cipriani@edisonstudio.it and maurizio@giri.it

ACKNOWLEDGEMENTS

The authors would like to thank Vincenzo Core and Salvatore Mudanò for their patience and long hours of work, Lorenzo Seno for his advice about digital audio, and Richard Boulanger, Marco Massimi and David Zicarelli for their generosity.
We particularly wish to thank Richard Dudas, whose invaluable work on this book went far beyond simply translating it. His constant feedback provided us with some very useful insights.

DEDICATIONS

This volume is dedicated to Arianna Giri, Sara Mascherpa and Gian Marco Sandri.

5T
DIGITAL AUDIO AND SAMPLED SOUNDS

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The Contents of Volume 1 (Theory and Practice)

OBJECTIVES

Knowledge

- To know the underlying principles of analog-to-digital and digital-to-analog sound conversion
- To know the measurable characteristics of audio interfaces and sound conversion
- To know the fundamentals of data compression
- To know the causes and effects of foldover and quantization noise
- To know various techniques for editing and organizing sounds inside a sampler
- To know several methods of segmentation and pitch manipulation of sampled sounds

Skills

- To be able to aurally distinguish and clearly describe the key differences between decimation and sample rate reduction
- To be able to aurally distinguish and clearly describe the differences between a given sound and the same sound played in reverse
- To be able to aurally distinguish the main differences between a sound processed with the Blocks Technique and one processed via slicing

CONTENTS

- Audio Interfaces and A-to-D / D-to-A Sound Conversion
- The Nyquist Theorem and Foldover
- Quantization Noise and Dithering
- The Organization of Sounds in a Sampler
- Segmentation of Sampled Sounds: The Blocks Technique and Slicing
- Pitch Modulation in Sampled Sounds
- Audio Data Compression
- Audio Data Transmission and Jitter

ACTIVITIES

- Sound Examples - Interactive Examples

TESTING

- Questions with Short Answers
- Listening and Analysis

SUPPORTING MATERIALS

Glossary

5.1 DIGITAL SOUND

In section 1.2T we already established that (a) sound is a mechanical phenomenon coming from a disturbance through a medium of transmission (generally the air) that has characteristics that can be perceived by the human ear. An acoustic sound can be amplified, reproduced and modified; but in order to do so, it must first be transformed into a signal capable of being measured, recorded, reproduced and modified in simple ways. Let's suppose we have a flautist friend who is playing; to transform his acoustic sound we can use a microphone. The microphone will operate as an electro-acoustic transducer, simply meaning that it acts as a constant gauge of variations in air pressure. At the same time the microphone will generate an electrical signal corresponding to the original, in the sense that its outgoing flow of electric tension corresponds to – in other words is analogous to – that of the input sound wave. For this reason the signal coming from the microphone is called an *analog signal*. However, be aware that the analog signal is never exactly identical to the original, but contains a certain amount of distortion (however minimal it may be) in addition to the introduction of noise (see figure 5.1).

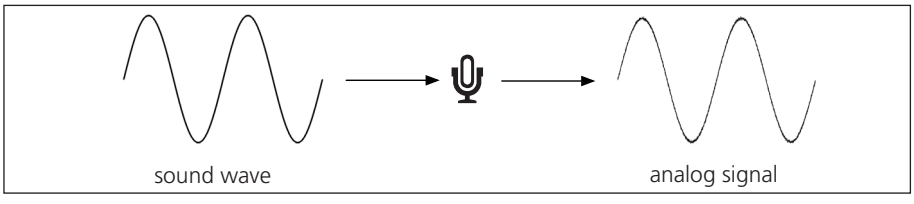

Fig. 5.1 An analog signal

In digital signals, however, the signal is represented by a series of numbers (remember that a digit is a base component of a numerical value). Each of the numbers in a digital signal represents the value of the instantaneous pressure, that is to say the value of the sound pressure at a given instant.
In order to generate a digital signal, the amplitude of the sound is measured at regular intervals (figure 5.2). This process is called sampling and is entirely analog.[1]

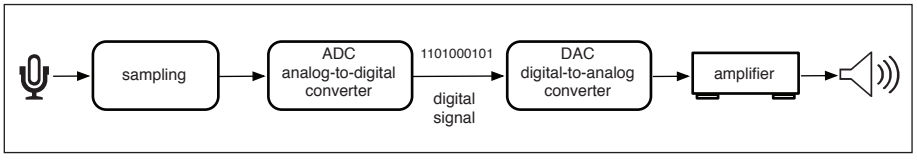

Fig. 5.2 Sampling and conversion

[1] The sampling system that is most often used is Pulse Code Modulation (PCM), based on the acquisition of the analog signal at regular time intervals. This sampling process is achieved by product modulation, which consists of multiplying the signal by a series of impulses. The result of this multiplication is a spectrum containing frequencies equal to the sums and differences of the two multiplied signals. In other words, this produces ring modulation – a technique which we will cover in more detail in the section of the third volume dedicated to non-linear synthesis.

Once the individual analog samples are arranged in time so each one assumes the amplitude value of the signal at that particular instant, they become subsequently converted into a flow of (binary) numerical data. This process is called *analog-to-digital* conversion. In order to be able to listen to the digitally converted signal again, a process of *digital-to-analog conversion* is necessary. This is the process by which the digital signal becomes converted into an analog signal once again, so it can be sent to an amplifier and subsequently to the speakers.

The audio interfaces in our computers (or the external audio interfaces attached to them) generally include both an analog-to-digital converter (or ADC) and a digital-to-analog converter (or DAC).

ANALOG TO DIGITAL CONVERSION

As we just stated, analog to digital conversion entails "translating" a signal composed of variations in electric tension into a numerical signal by defining its electric tension in (binary) numerical terms at regular intervals. In reality, however, transforming a sound from an analog signal to a digital one requires several different steps, as we will see at the end of this section where the subject is covered in more detail. For now it is enough for you to simply understand that the ADC contains both analog sampling and digital conversion.

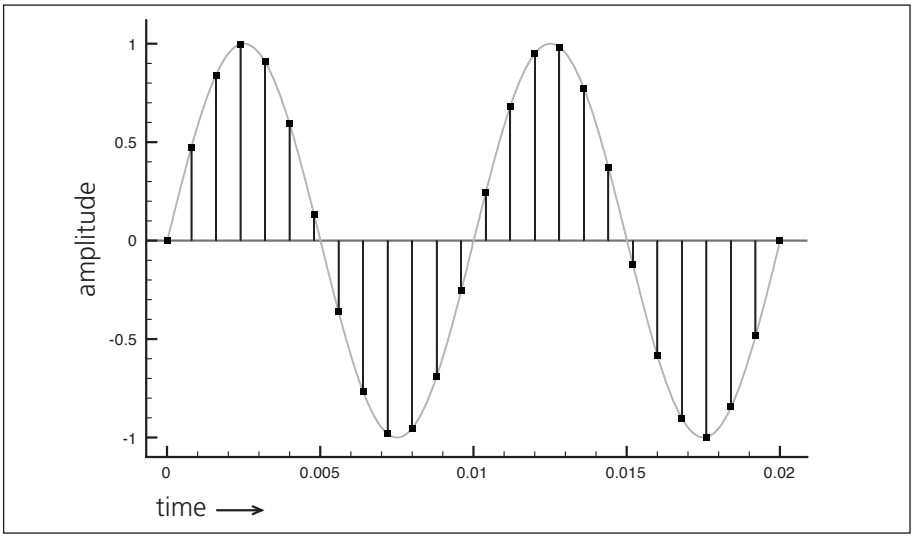

Fig. 5.3 A sampled signal

In figure 5.3, the continuous line represents the flow of an analog signal (i.e., voltage or electric tension), while the superimposed points represent the value of the sampled signal after being converted into numerical values. The sampling and the conversion of the signal's first amplitude value (the value 0) takes place at time 0. After a given amount of time (at time 0.001) a second sampling and a conversion takes place. In the interval between the two sampling times, the analog signal has evolved in its instantaneous amplitude value, and now has

an amplitude value equal to about 0.5. The next sampling takes place at time 0.002, and the converted amplitude value is a little above 0.8. As you can see, the process of sampling records only a few out of an infinite number of values which make up the analog signal over time, so you might therefore think that the resulting digital signal contains errors, or is otherwise not faithful to the original signal. However, we will soon discover that as long as the analog signal contains only frequencies which are less than half the sampling rate, it is possible to reconstruct the original signal from the digital samples without any ambiguity.

The time interval that passes between one sample and the next is called the *sampling period*, and is measured in seconds or in a subdivision thereof (milliseconds or microseconds). In figure 5.4 it is indicated with the label *sp*.

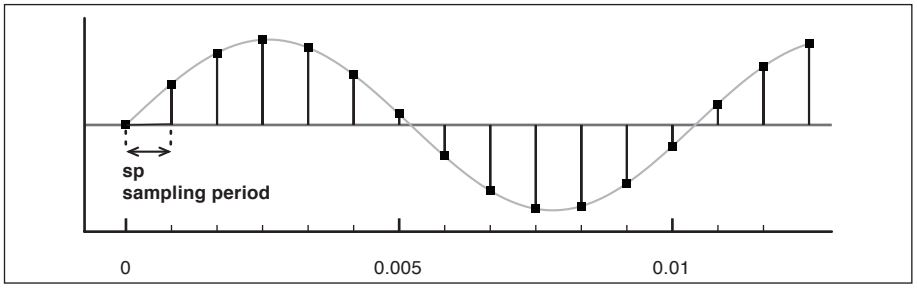

Fig. 5.4 The sampling period

The inverse of the sampling period is known as the *sampling rate* (it is also referred to as the *sampling frequency*, or *sample rate*), which was briefly described in section 1.5 of the first volume of this series. It is commonly abbreviated *sr*, as it is throughout this text, and is measured in Hertz (Hz). For example, if we use a sampling rate of 48000 Hz to sample a sound, it means that in one second of time its amplitude will be measured 48000 times.

We can therefore define the following relationships:

sr = 1/ sp

sp = 1/sr

In other words, the sampling rate (the number of samples per second) is equal to the inverse of the sampling period, and vice-versa.

To find out how to calculate a sampling rate *sr* that will allow a sound to be correctly sampled, we need to refer to the *Nyquist theorem* (also called the *sampling theorem*), which states that the sampling rate must be greater than twice the maximum frequency contained in the signal. Therefore, if *fmax* is the maximum frequency contained in a given signal, the minumum sampling rate *sr* that we would need to use to faithfully reproduce it would be:

sr >= 2 · fmax

This naturally implies that the maximum representable frequency will be less than half the sampling rate. The frequency equal to half the sampling rate is called the *Nyquist frequency*.

Before carrying out the process of sampling on a sound, any frequencies above the Nyquist frequency that are contained in the analog signal must first be eliminated, otherwise they would create an undesirable effect known as foldover, which we will discuss shortly. For this reason, conversion systems incorporate some important measures, including the application of an analog lowpass filter before sampling (called an *anti-aliasing filter*), in order to eliminate frequencies above the Nyquist.

Bear in mind, however, that it is impossible to create an ideal analog filter with an infinitely steep slope at its cutoff frequency, and it is even technically extraordinarily difficult to design one with a very sharp slope. Consequently, there is always a risk that some frequencies above the Nyquist frequency will remain in the sound after filtering. To resolve this problem, sampling systems use a technique of *oversampling* after the analog filtering step (using a filter whose slope is not steep). When using oversampling, the sound is sampled at a higher sampling rate than the one that was initially chosen, in order to create a new Nyquist frequency far above the cutoff frequency of the analog filter. Subsequently, a digital lowpass filter, with a steep slope, is applied to the oversampled signal. This eliminates any frequencies above our original (higher) Nyquist frequency (the one before the oversampling). The sound can now finally be resampled at the chosen sampling rate, using a process of *downsampling*. By using this method we can be certain that there will be no undesired effects resulting from foldover when sounds are sampled.

The best results (above and beyond the stated techniques) can be obtained using sampling rates above 44.1 kHz, in order to move undesired spectral images to higher frequencies. It is precisely for this reason that higher sampling rates are generally used. So, which sampling rates are more frequently used for audio? In the case of compact discs, a *sr* of 44.1 kHz was adopted (permitting reproduction of sounds up to 22050 Hz), whereas, the *sr* of DVD and Blu-ray Discs can go up to 192 KHz (therefore permitting reproduction of frequencies up to 96000 Hz, well above the maximum audible frequency).

FOLDOVER IN DIGITALLY GENERATED SOUNDS

What would happen if the sampling system did not include an anti-aliasing filter, and thus allowed frequencies above the Nyquist to pass through to the sampling stage? Or what would happen if, instead of sampling a sound, we digitally generated a signal inside the computer whose frequency was above the Nyquist frequency? In both of these scenarios, we would obtain an effect known as foldover. Foldover is the phenomenon by which frequency components that exceed half the *sr* become *reflected* back under it. For example, a frequency component of 11000 Hz converted using a sampling rate of 20000 Hz, will produce a foldover component of 9000 Hz, as we will see in detail shortly.

Let's imagine that we want to generate a sine wave with a frequency of 12000 Hz. If we use a sampling rate of 18000 Hz, we will have a Nyquist frequency equal to 9000 Hz. The sound that we want to generate will thus *exceed* the Nyquist frequency by 3000 Hz. Consequently, that sound will not have its original frequency (in this case defined as 12000 Hz), but instead will appear 3000 Hz *below* the Nyquist frequency with its sign inverted[2] (in other words 6000 Hz, but with a negative sign).
Therefore, in the case where an oscillator's frequency is set above the Nyquist, the actual output frequency due to foldover can be calculated with the following formula, where *sr* is the sampling rate, f_C is the frequency to convert and f_O is the output frequency:

$$\mathbf{f_O = f_C - sr}$$

Let's now apply that formula to our previous example:

12000 - 18000 = -6000

Note that this formula is only valid in the case where frequency f_C is between half the sampling rate (i.e., the Nyquist frequency) and 1.5 times the sampling rate *sr*. In the section dedicated to *aliasing* we will learn a general formula that works for all frequencies.

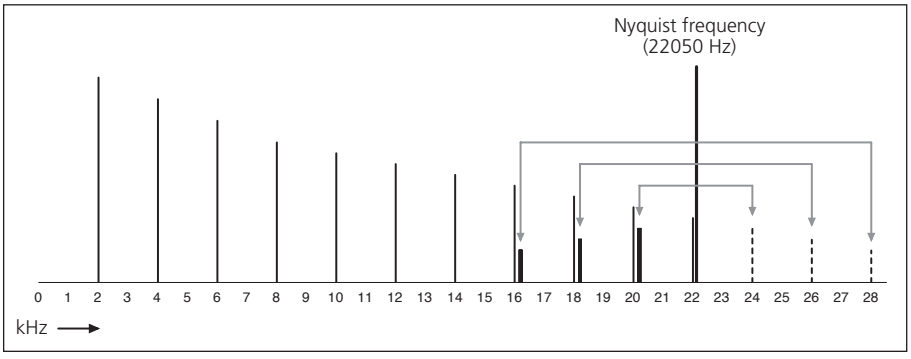

Fig. 5.5 Foldover

Naturally, this phenomenon occurs not only in relation to sinusoidal signals but for any component of a complex signal. Let's take a look at figure 5.5, which shows a signal made up of various harmonic partials whose fundamental frequency is equal to 2000 Hz. The solid lines represent components actually present in the signal after conversion, whereas the dashed lines represent the original components of the analog signal that are no longer present in the converted

[2] The behavior of each partial, when its sign is inverted from positive to negative (or from negative to positive), will depend on its waveform and phase. For example, when the sign of the frequency of a sinusoid is inverted, its phase is inverted, but when we invert the sign of the frequency of a cosine it remains the same. From this perspective, we can see that it not easy to calculate the effect of foldover upon the output spectrum.

signal. The bold solid lines represent these latter components which have become "reflected" back under the Nyquist frequency (which in this example is equal to 22050 Hz). Let's now take a closer look at the frequency at 24000 Hz (that is, the 12th harmonic of the 2000 Hz fundamental in this example): since it is above the Nyquist frequency, it is subject to foldover, and therefore becomes:

frequency to convert - sampling rate = output frequency
24000 - 44100 = -20100

Similarly, the frequency at 28000 Hz (the 14th harmonic of the 2000 Hz fundamental) becomes:

28000 - 44100 = 16100

In the interactive examples which follow, we can hear three different sound events, each based on a sampling rate of 22050 Hz and therefore having a Nyquist at 11025 Hz. From this we can conclude that any sound above 11025 Hz will be subjected to the effect of foldover.

- in the first example we will sweep the frequency of a sinusoidal sound from 20 to 10000 Hz; there is no foldover in this scenario so we will therefore hear a simple ascending glissando

- in the second example the glissando will extend from 20 to 20000 Hz. At the moment the sound passes above 11025 Hz, we will hear the phenomenon of foldover. Once it goes above the threshold of the Nyquist frequency, the upward glissando will become a descending one, since the more the frequency ascends, the more the reflected frequency will descend due to foldover. The sound will continue to sweep downward until it stops at 2050 Hz. According to the above formula we can calculate:

  f_C - sr = f_O
  20000 Hz - 22050 = -2050 Hz

- In the third example the sound will sweep from 20 to 30000 Hz. In this case we can audibly discern a double foldover. Let's take a look at it in detail:

1) in the initial stage from 20 to 11025 Hz, the output frequency corresponds to the frequency we have specified.
2) at the moment it exceeds 11025 Hz the first foldover occurs, making the resulting frequency glissando downward until it arrives at the 0 (in other words, it does this as the frequency goes upward from 11025 to 22050 Hz).

   f_C - sr = f_O
   22050 Hz - 22050 - 0

3) our generated frequency now continues beyond 22050 until it arrives at 30000 Hz. At the instant when the reflected sound goes below the threshold at 0 (into the negative), the frequency will once again begin to ascend, thereby creating another foldover taking place when the frequency of the signal is less than zero. In general, frequencies less than zero reappear with their sign inverted, in other words mirrored into the positive (-200 Hz becomes 200 Hz, -300 Hz becomes 300 Hz, etc.).

As a result of this second foldover, the output frequency rises again until 7950 Hz. So, how do we obtain this final frequency? 7950 Hz is equal to 30000 Hz (the final destination frequency of the signal we are converting) minus 22050 Hz (the sampling rate). Note that because it is caused by a second foldover, the final frequency has a second inversion of its sign, back into the positive.

To summarize what has happened, when the frequency we specify exceeds the Nyquist Frequency (in this case 11025 Hz), a first foldover occurs. When the specified frequency goes beyond the sampling rate (in this case 22050 Hz) a second foldover takes place (see figure 5.6, below).

f_C - sr = f_O
30000 - 22050 = 7950

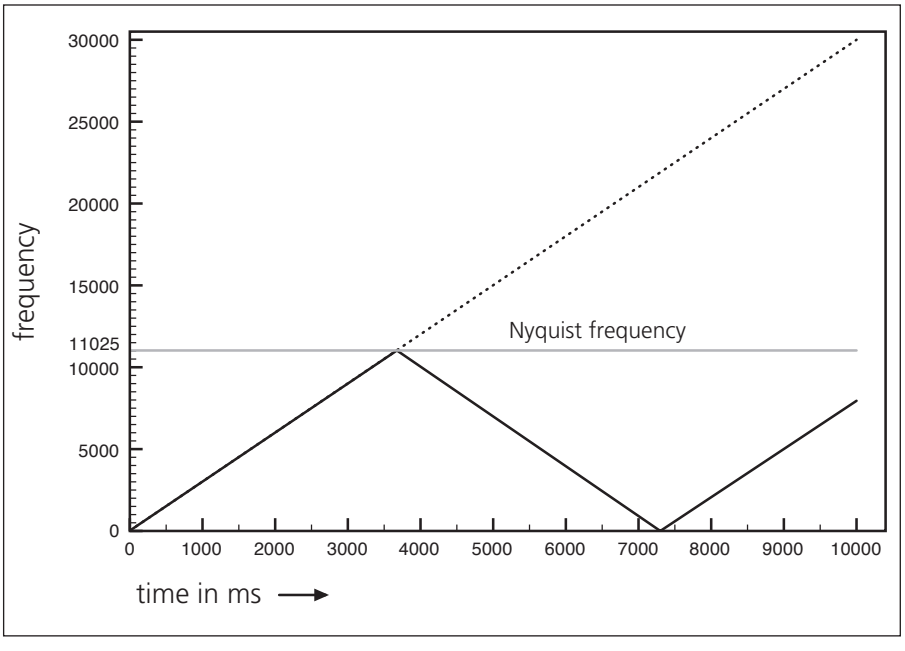

Fig. 5.6 Double foldover

SOUND EXAMPLES 5A • Foldover

5A.1 Simple ascending glissando from 20 Hz to 10000 Hz; *sr* = 22050
5A.2 Ascending glissando from 20 Hz to 20000 Hz with foldover; *sr* = 22050
5A.3 Ascending glissando from 20 Hz to 30000 Hz with double foldover; *sr* = 22050

ALIASING: THE CAUSES

Although we have already explained the effects resulting from foldover, if we want to better understand its causes we will need to explain the Nyquist theorem in more detail. In order to have a more precise notion of the limitations imposed by the Nyquist theorem, let's imagine a hypothetical sampling system without anti-aliasing filters that uses a given sampling rate, **sr**, to sample a sinusoidal waveform whose frequency, **f**, is less than the Nyquist frequency. With this system we obtain is a series of samples that numerically represent the sine wave. Now, if with the same sampling rate sr we sample a sine wave whose frequency is (**f + sr**) – in other words, a sine wave whose frequency is equal to the sum of the previous sine wave's frequency and the sampling rate (and therefore greater than the Nyquist frequency) – we obtain exactly the same series of samples as we do when sampling the sine wave with frequency f. Furthermore, we would also obtain the exact same series of values when sampling sine waves with frequencies (**f + 2 · sr**), (**f + 3 · sr**), (**f + 4 · sr**), etc., ad infinitum. We can therefore generalize that given a sampling rate sr, all of the sine waves at frequencies of f + an integer multiple of the sampling rate will be converted into the same series of samples.
To provide a more concrete example, if our sampling rate sr is 5000 Hz, and we sample a sine wave whose frequency f is 1000 Hz, we will obtain a series of sampled values. If we then sample a sine wave whose frequency is 6000 Hz – i.e., a frequency equal to f (1000 Hz) plus sr (5000 Hz) – we will get an identical series of values.

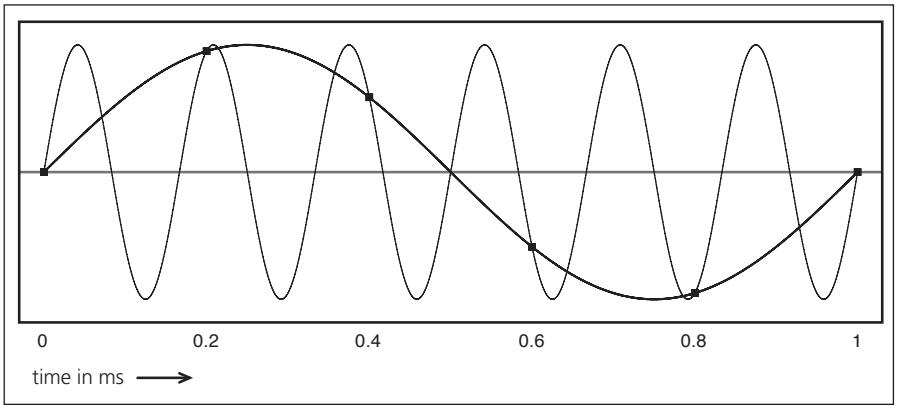

Fig. 5.7 Two sine waves with frequencies of 1000 Hz and 6000 Hz sampled at a rate of 5000 Hz.

Figure 5.7 illustrates how two sine waves with different frequencies – 1000 Hz and 6000 Hz, respectively – sampled at the same sampling rate of 5000 Hz, yield identical sample values (identified by the square-shaped points in the graph). This same series of values also can be obtained by sampling sinusoids at 11000 Hz (**f + 2 · sr**), 16000 Hz (**f + 3 · sr**), etc.

This mathematical equivalence also holds true when using *negative integer multiples* of the sampling rate; sampled sine waves with frequencies of (**f - sr**), (**f - 2 · sr**), (**f - 3 · sr**), (**f - 4 · sr**), etc., will all produce the same series of samples. To continue from our previous example, a sine wave with frequency -4000 Hz – the sum of the frequency 1000 Hz (f) and sampling rate -5000 Hz (-sr) – will generate the same amplitude values as the sine wave with frequency 1000 Hz. Note that a sine wave of -4000 Hz is equal to a 4000 Hz sine wave with its sign inverted (see figure 5.8).

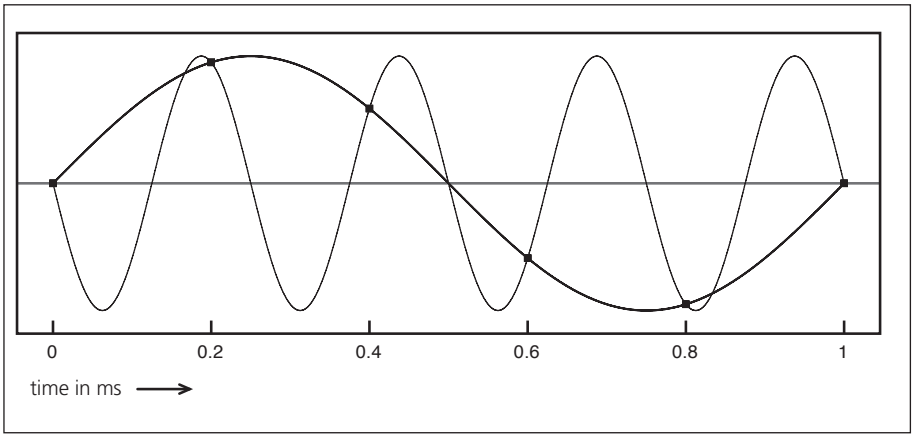

Fig. 5.8 Two sine waves with frequencies of 1000 Hz and -4000 Hz sampled at a rate of 5000 Hz.

To summarize, we can say that: *given a sampling rate sr and any integer k, positive or negative, we can not distinguish between the sampled values of a sine wave of frequency f Hz and that of a sine wave with the frequency (**f + k · sr**) Hz.*

Returning to our previous example, here are the frequencies of the sine waves that we find inside the audio spectrum, and which generate identical sample values:

frequency (f+k · sr)
1000 Hz = 1000 + (0 · 5000)
6000 Hz = 1000 + (1 · 5000)
11000 Hz = 1000 + (2 · 5000)
16000 Hz = 1000 + (3 · 5000)
-4000 Hz = 1000 + (-1 · 5000)
-9000 Hz = 1000 + (-2 · 5000)
-14000 Hz = 1000 + (-3 · 5000)

This situation also holds true when converting a sound that is more complex than a sine wave: the mathematical relation (**f + k · sr**), in fact, is valid for any and all spectral components in a sampled sound.
From this relationship we can derive a formula that will allow us to calculate the output frequency of any arbitrary frequency to be sampled. We will indicate the frequency that we are sampling with f_C, the sampling rate with the usual sr, the output frequency with f_O, and the whole number multiple of sr that is closest to f_C using N, in order to obtain the formula:

$$\mathbf{f_O = f_C - N \cdot sr}$$

As we can see, this formula is very similar to the simplified one that we used earlier in the section dedicated to foldover.

Here are some examples:

1) $\mathbf{f_C = 6000, sr = 10000, N = 6000/10000 = 0.6 = 1}$ (closest whole number value)

therefore: **6000 - 1 · 10000 = -4000**

2) $\mathbf{f_C = 13000, sr = 10000, N = 13000/10000 = 1.3 = 1}$ (closest whole number value)

therefore: **13000 - 1 · 10000 = 3000**

3) $\mathbf{f_C = 21000, sr = 10000, N = 21000/10000 = 2.1 = 2}$ (closest whole number value)

therefore: **21000 - 2 · 10000 = 1000**

4) $\mathbf{f_C = 2500, sr = 10000, N = 2500/10000 = 0.25 = 0}$ (closest whole number value)

therefore: **2500 - 0 · 10000 = 2500** (in this case there is no foldover because $\mathbf{f_C < sr/2}$)

Let's now take a look at what happens in the frequency domain:

In a hypothetical sampling system without *antialiasing* filters, the components (due to the foldover effects we have just described) will make more images of the same spectrum called *aliases*, replicated periodically around multiples of the sampling rate. More precisely, we will obtain a periodic spectrum that, in an ideal sampling system, will repeat along the frequency axis to infinity. In figure 5.9 we see how the various copies are positioned in the frequency domain in the case of a sine wave of 1000 Hz sampled with sr = 5000. The frequencies shown in the image are the same as in the preceding table 1000, 6000, 11000, 16000, -4000, -9000 and -14000.

As we have already learned, the negative frequencies reflect into the positive frequency range, therefore in the image components are shown at 1000 (the sampled frequency, which we will call f), 4000 (**sr-f**, which is 5000-1000), 6000

(**sr+f**), 9000 (**sr · 2-f**), 11000 (**sr · 2+f**), 14000 (**sr · 3-f**), 16000 (**sr · 3+f**), etc. In figure 5.9, the spectrum is represented as symmetrical around 0, and the successive copies are mathematically translated so they also appear symmetrical around integer multiples of the sampling rate.

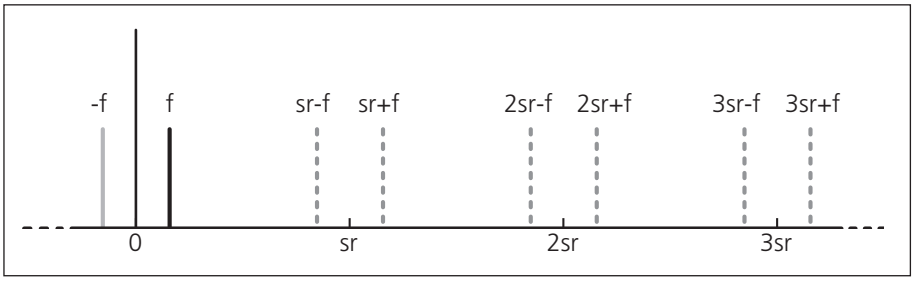

Fig. 5.9 *Aliasing*

Naturally, in real-world sampling systems frequencies above the Nyquist are eliminated before sampling precisely in order to cancel out the phenomena that we have been discussing. As we will see later, in the case of decimation, when undersampling a digital signal, frequencies resulting from *aliasing* will become present and therefore audible and should be filtered beforehand with an appropriate lowpass digital filter.

So, what happens if instead of sampling a sine wave we sample a sound composed of several partials? In this case, each and every one of the sound's components will be replicated in this same way. Consequently the aliased sonic image will contain copies of the complex spectrum, as shown in figure 5.10.

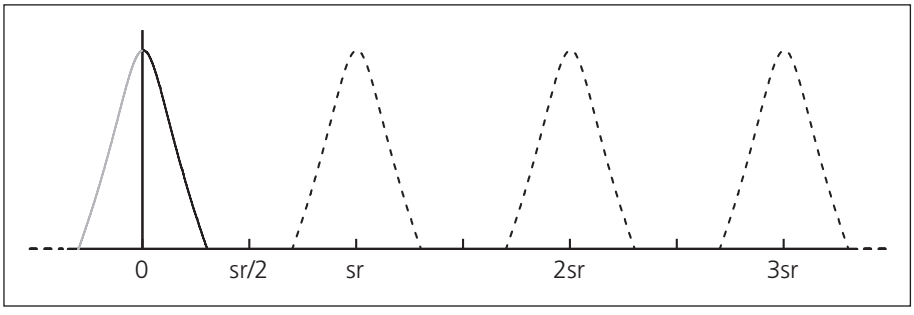

Fig. 5.10 Aliasing of a complex sound.

DIGITAL-TO-ANALOG CONVERSION

Let's now take a look at the reverse process: *digital-to-analog conversion*. When converting from a digital signal to an analog one, it becomes a piecewise (or stepped) signal in which, after each sample and until the next, a *sample-and-hold* mechanism (see section 4.7T and footnote 14, chapter 3.4P) is applied in order to sustain the electric tension value (i.e., voltage) of the analog output until the subsequent sample (remember that each sample represents a single

point and there are no other values in-between samples). Using this method, a signal composed of discrete samples can be converted into a continuous (albeit slightly jagged) analog signal. Since this process introduces angular steps not present in the original (smooth) analog signal, we have inevitably modified the signal by altering its waveform, creating components not present in the original. These components, called *aliases*, form other images of the original spectrum (basically, harmonics of it) around the oversampling frequency, each one decreasing in amplitude as frequency increases.

In figure 5.11 we can see how the *sample-and-hold* mechanism reduces the components at high frequencies. More precisely, in the upper part of figure (a) we can see an ideal sampling system made with impulses of an infinitesimal duration: the resulting spectrum extends to infinity without losing the amplitudes of the components. In the lower part (b) we can see a sampling system realized by means of *sample-and-hold* which as we previously stated, is comprised of progressively quieter copies of the spectrum.[3]

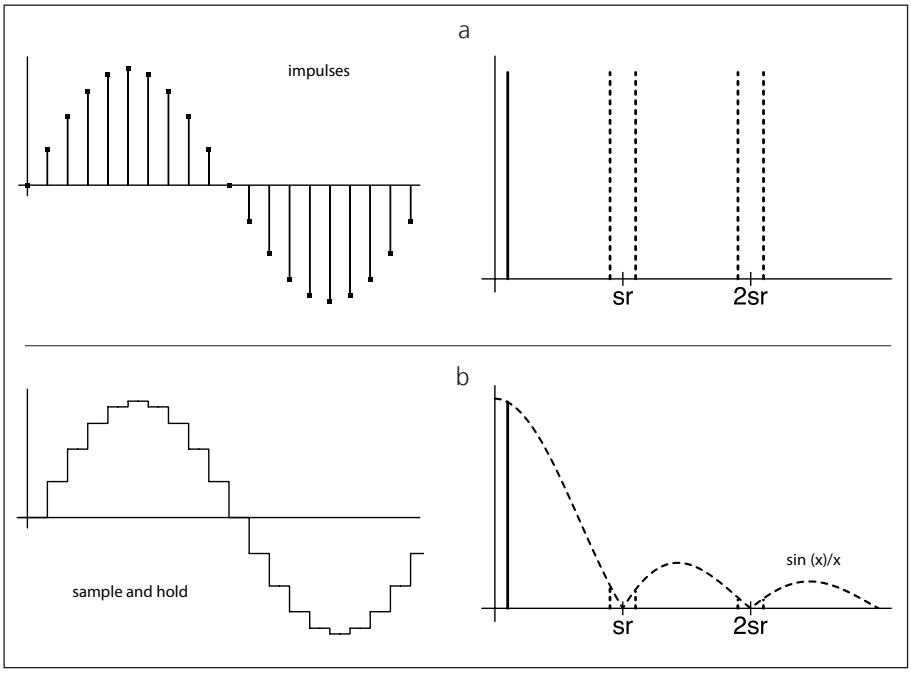

Fig. 5.11 Sampling and aliasing

Naturally, if the spectrum in this case is composed of several partials we will also have copies of the spectrum decreasing in amplitude as the frequency increases.
(...)

[3] The copies of the spectrum in a sample and hold -based sampling system will decrease according to the function sin(x)/x (see the image in the lower right hand corner of figure 5.11)

other sections in this chapter:

5P
DIGITAL AUDIO AND SAMPLED SOUNDS

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The Contents of Volume 1 (Theory and Practice) + Chapter 5T

LEARNING OBJECTIVES

Skills

- To know how to manage the global audio settings for MSP
- To know how to record a sound
- To know how to use sampled sounds and modify their amplitude, frequency and bit resolution
- To know how to control the playback of a sound in reverse
- To know how to creatively control looping of sampled sounds
- To know how to import a sampled sound into a buffer and use this to generate sound
- To know how to assemble a simple sampler
- To know how to creatively control the degradation of a sound by means of bit reduction and decimation

Competence

- To be able to realize a short etude based on sampled sounds, using looping, reverse playback, reading from different starting and ending points in the file, envelopes, glissandi, etc.

CONTENTS

- Audio Settings in MSP
- Foldover
- Bit Reduction and Decimation
- Quantization Noise and Dithering
- Sampled Sound Acquisition and Playback Methods
- The Construction of a Sampler
- The Blocks Technique
- Slicing
- Audio Scrubbing and Designing a Random Scrubber

ACTIVITIES

- Building and Modifying Algorithms

SUPPORTING MATERIALS

- List of Max Objects - List of Attributes and Messages for specific Max objects - Glossary

5.1 DIGITAL SOUND

GLOBAL AUDIO SETTINGS IN MSP

In order to manage the global audio settings and choose the audio interface that MSP will communicate with, you will need to open the Audio Status window[1] which is found in the Options Menu (figure 5.1).

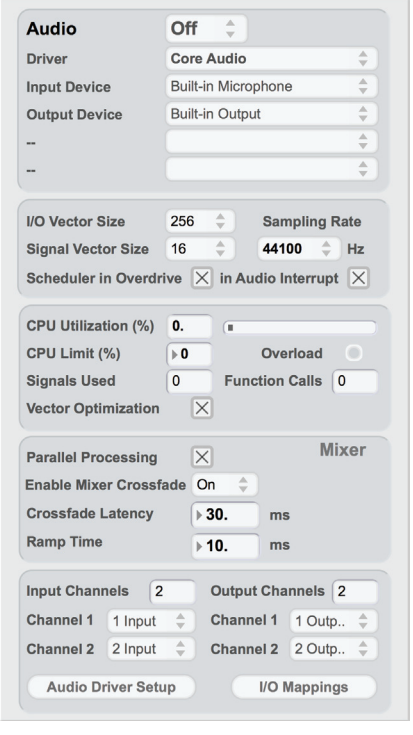

Fig. 5.1 The Audio Status window

We will now take a look at the main features of this important window. It is not necessary to memorize all the information that is described here; you can always refer back to this section each time you need information about the Audio Status window.

The window is divided into five boxed sections, each of which contains a group of related parameters. The first section primarily deals with the audio interface. Let's look at some of its main features:

Audio: this can be used to start or stop the DSP engine (this has the same function as sending the start and stop messages to the object `adc~` or `dac~`).

[1] This name was adopted beginning with Max 6, in older versions this window was called DSP Status.

Driver: this chooses the audio driver used by MSP. It can be used either to select the driver for an audio interface connected to the computer (in the figure above, the "CoreAudio" driver of a Macintosh computer is selected), or to send the signal to an application compatible with the ReWire protocol[2], by selecting the "ad_rewire" menu option. The "NonRealTime" option can also be selected to generate a signal out of real-time, which can be very useful if the computation algorithm is too complex to be able to be executed in real-time by the computer. In this case the signal will naturally be saved to disk so you can listen to it when processing is finished. Finally, by selecting the option "Live," users of Max For Live[3] can send the audio directly to the program Ableton Live.

Input Device, Output Device: these two parameters allow us to specify the input and output devices. Their function depends on the type of driver and audio interface used.

The second boxed section allows us to regulate the relationship between efficiency and latency for the audio processing, as well as to regulate the temporal precision of Max messages.

I/O Vector Size: digital audio signals do not pass between MSP and the audio interface one sample at a time, but rather in groups of samples called *vectors*. The size in samples of the input/output vector can be set here. The smaller the vector is, the less latency (i.e., delay) there will be between input and output. On the other hand, processing each vector has a certain computational cost – this means that, when using a very small vector size, more vectors will need to be computed per second than with larger vectors, and this will increase the percentage of CPU needed to compute the signal. What's more, a vector that is too large can create problems, in so much as MSP may not be able to calculate the signal in the time available, and this could create a click in the audio being output. We recommend using a setting no greater than 256 samples, even though the range of possible values for this setting will depend on the audio interface (some interfaces, for example, could have a minimum I/O latency of 512 samples).

Signal Vector Size: this parameter indicates the number of samples that the MSP *patch* itself will process at a time. Also in this case, the larger the vector is, the less the computational cost will be. The difference between this and the former vector setting is that the signal vector size will have no effect on latency, and cannot be set to values larger than that the I/O Vector Size. For certain objects (we will see which ones at the appropriate moment) there can nonetheless be useful minimum values. We recommend using a value between 16 and 128 samples.

Sampling Rate: here we can specify the sampling rate. The list of sampling rates available will vary depending on the audio interface used.

[2] For more about the ReWire protocol, see http://www.propellerheads.se.

[3] We will talk about Max For Live at the end of this Volume.

Scheduler in Overdrive: this option can also be set using the Options menu. When Max is in overdrive, it gives priority to timed events (for example, bangs sent by the **`metro`** object) and MIDI messages that it receives, over other tasks of secondary importance, such as the refreshing the patch's graphics, or responding to mouse or computer keyboard input. This means that Max will be rhythmically more precise, but may run the risk of no longer responding to keyboard or mouse input if there are too many timed events to deal with. Nonetheless, MSP signals always have priority over Max messages.

in Audio Interrupt: this option is available only when Max is in overdrive mode. When this option is set to *yes*, timed events are always processed immediately before calculating each signal vector. This allows us to more precisely synchronize audio with Max messages[4]. However, when using this option we should select a very small Signal Vector size (less than 64 samples) otherwise the Max messages are likely to be sent at times significantly different from those expected. This happens because Max must "wait" for MSP to compute its signal vector before being able to generate its timed message. For example, if the signal vector were 1024 samples long using a sampling rate of 44100 Hz, the vectors would be calculated roughly every 23 milliseconds. This means that a sequence of timed Max events would be reproduced in successive bursts every 23 milliseconds. With a signal vector of 16 samples, on the other hand, the time interval between two vectors would be less than a half a millisecond, and therefore would only create very small delays in the timing of Max events and thus absolutely imperceptible to the listener.

The third boxed section presents information about the current signal processing status:

CPU Utilization: indicates the percentage of the computer's processor that MSP occupies in order to perform the active patch's audio algorithm. Obviously, the same patch will use different percentages on computers with different processing power. When a patch requires a percentage equal to or above 95% the computer will become difficult to control and will respond extremely slowly to commands; it is therefore a parameter to keep an eye on.

CPU Limit: this allows you set limits on the percentage of CPU that MSP is able to use (a value equal to zero means "no limit"). This can be useful to keep the program from "taking over" all of the computer's resources.

Signals Used: represents the number of internal *buffers* used to connect MSP objects in the active patch(es).

[4] This option is very useful in patches where the Max messages activate the production of sound in MSP, such as in the patch IB_04_sequence.maxpat which was discussed in Interlude B of the first volume: we recommend that you also keep it activated for the patches which will be described in successive sections of this chapter.

Function Calls: reports how many MSP functions are used in the active signal processing network. An MSP function roughly corresponds to an MSP object, although some objects may have multiple internal functions that get added to the overall signal processing network. In general you could think of this number as an estimate of how many calculations are needed to generate a sample in the active patch. Note that the lower the value of these last two fields, the greater the patch's efficiency will be.

The fourth boxed section (present as of Max version 6) allows users to configure the parameters of the *Mixer Engine*, the system that controls the modification of signals in the Max patch. Before Max 6, all signal processing took place inside a single process or *DSP chain*, even when multiple patches were running simultaneously. This meant, among other things, that the global audio would be interrupted every time a patch was modified, because the DSP chain needed to be reconstructed each time. As of Max 6 there is a distinct DSP chain for every active process, and this permits us to work on a patch without interrupting the audio of the other active patches.

Let's take a look at the parameters for the *Mixer Engine*:

Parallel Processing: this parameter lets us assign each DSP chain to a different processor, when running on multiprocessor systems.

Enable Mixer Crossfade: when an active patch (i.e., one that is producing sound) is modified, it is possible to make a crossfade between the sound produced by the patch before and after making the changes. Of course, during the crossfade, the program uses twice the CPU, because there are, in fact, two active copies of the patch running simultaneously. This parameter can be turned *Off* or *On*, or set to automatic (*Auto*), in which case the latency needed to make the crossfade (see below) is added only when the patch is being edited while the DSP engine is on.

Crossfade Latency: the amount of time defined by this parameter is used to rebuild the new DSP chain after a modification to the patch, and to realize the subsequent crossfade. If the time specified is less than the time necessary to rebuild the DSP chain, or if the crossfade is not activated, the audio of the old patch will fade out and that of the modified patch will fade in afterward.

Ramp Time: this is the actual duration of the crossfade. This parameter should have a value less than the amount of time of the Crossfade Latency, so the crossfade will happen correctly.

In addition to using the *Audio Status* window to set the parameters of the *Mixer Engine*, they can also be set in the Max Preferences window, which can be opened using either the *Max* menu (on Macintosh) or the *Options* menu (on Windows). Furthermore it is possible to set some of the *Mixer Engine* parameters by clicking on the last icon (the Mixer icon) in the *Patcher Window Toolbar* located in the lower part of every *Patcher Window*.

Input Channels, Output Channels: these are the number of input and output channels used by the audio interface.
The four menus provided can be used to set the first two of the audio interface's input and output channels. In order to set other channels you need to click on the lower right-hand button marked "I/O Mappings." The button on the lower left, labeled "Audio Driver Setup" can be used to access the preferences for the audio interface you are using.

FOLDOVER

Referring to section 5.1 of the theory chapter, we can see what happens when we move the frequency of a sine-wave oscillator above the Nyquist frequency. Rebuild the patch shown in figure 5.2.

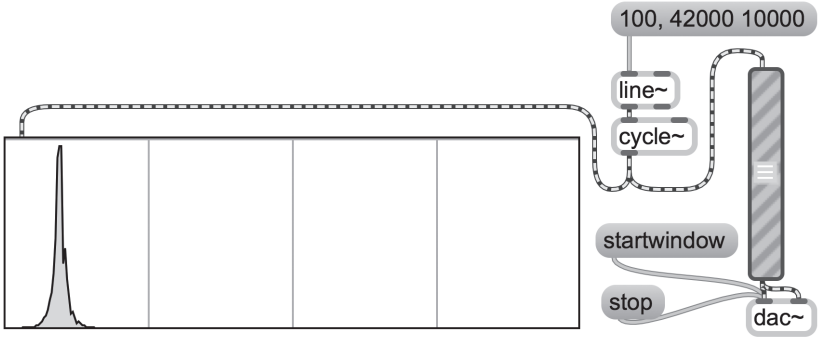

Fig. 5.2 Glissando beyond the Nyquist frequency.

The `cycle~` oscillator object is connected to the `spectroscope~` object which, as we have already seen, can be used to visualize a signal's spectrum. Clicking on the *message box* connected to the `line~` object, triggers a glissando (from 100 to 42000 Hz over 10 seconds) that will go beyond the Nyquist frequency.

The destination frequency indicated in the figure (42000 Hz) assumes that we are using a sampling rate (*sr*) of 44100 Hz. If your audio interface is set to a different sampling rate (you can check this in the *Audio Status* window) you should change it to 44100 Hz, or else use a different destination frequency. The destination frequency should be about 2000 Hz less than the sampling rate being used – for example, for an *sr* of 48000 Hz, the destination frequency of the oscillator should be set to 46000 Hz, whereas for a *sr* of 96000 Hz, the destination should be around 94000 Hz, and so forth. When running the patch, we can see (in the spectroscope) how the sinusoid rises until the Nyquist frequency and then "bounces" back in the opposite direction, stopping at the reflected frequency 2100 Hz (in reality -2100 Hz), which corresponds to the formula that we saw in section 5.2 of the theory chapter:

fc	**- sr**	**=**	**output frequency**
42000 Hz	**- 44100 Hz**	**=**	**-2100 Hz**

Now let's replace the sine wave with a sawtooth oscillator (figure 5.3).
We will use the `phasor~` object as an oscillator. Although the `phasor~` object outputs a ramp from 0 to 1, by including a few simple math operations, we can convert this into a ramp from - 1 to 1 (as we have already done in the first chapter).

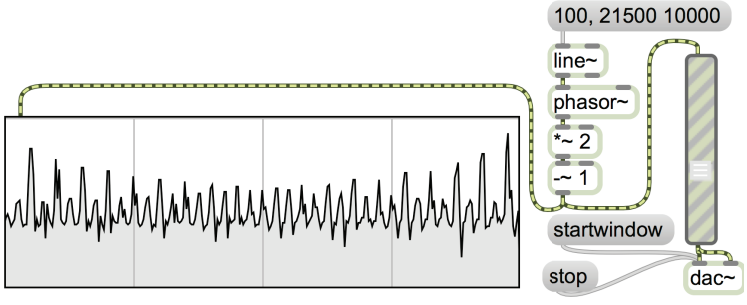

Fig. 5.3 Making a glissando with a non-bandlimited waveform

You will notice that we have lowered the destination frequency to 21500 Hz. This is because we do not need to increase the fundamental beyond the Nyquist since the sawtooth is already rich in harmonics which will pass the Nyquist frequency (as always, you can modify the destination of the ramp in accordance with the sampling rate of your audio interface). If we turn on the patch and trigger the upward glissando, we will hear a shower of harmonics bouncing off the "wall" at the Nyquist frequency. These reflected harmonics quickly descend to 0 Hz, and rebound upward once again, only to be reflected back down, back up, and so forth, creating a constant zigzag throughout the spectrum during the glissando. After 10 seconds, when the ramp is finished, the spectrum will settle into an inharmonic arrangement of partials.

The spectrum of this waveform is extremely rich in harmonics, because it is very close to an ideal sawtooth waveform (which, as we already know, contains an infinite number of harmonic components). This means that it is practically impossible for the `phasor~` object to output a signal without generating *foldover*[5].

Could the partials that exceed the Nyquist be eliminated by applying a lowpass filter (for example with a cutoff at 20000 Hz) to our non-bandlimited signal generator? Unfortunately, no, because the partials reflected due to foldover inside the reproducible audio band are completely indistinguishable from a sound generated at that frequency, and the filtering would only happen after the reflections. For example, if we use a sampling frequency of 48000 Hz, and we employ the `phasor~` object to generate a non-bandlimited signal with a fundamental of 10000 Hz, we will obtain the following series of harmonics:

[5] In reality, when the fundamental frequency is very low, only the very highest harmonics will exceed the Nyquist. As we have seen in section 2.1T, these harmonics are extremely weak, so the foldover is negligible.

Fundamental: 10000 Hz
2nd partial: 20000 Hz
3rd partial: 30000 Hz = 24000 - (30000 - 24000) = 18000 Hz
4th partial: 40000 Hz = 24000 - (40000 - 24000) = 8000 Hz
etc.

Since the third and fourth partials exceed the Nyquist frequency, they are reflected to 18000 and 8000 Hz, respectively. However, these reflected sounds are *actually* already at 18000 and 8000 Hz at the output of the `phasor~` object, so filtering out frequencies above 20000 Hz would have no effect. To filter the unwanted reflections in this scenario, we would need to use a lowpass filter with a cutoff below 8000 Hz, but by doing this we would also eliminate the fundamental and second partial! To fully understand this, try to add the lowpass filter `vs.butterlp~` to the patch shown in figure 5.3, and you will realize that the signal is filtered only *after* the foldover has already taken place.

In chapter 2.1P of the first volume, we have already seen how it is possible to create approximations of ideal waveforms (such as a sawtooth wave) that contain a limited (i.e., not infinite) number of harmonics, using the `vs.buf.gen10` object. Using the method shown in that chapter, we can approximate a sawtooth waveform using only 20 harmonics, and therefore, supposing a sampling frequency of 44100 Hz, we can create an oscillator that can safely go up to about 1102 Hz without incurring foldover (at 1102 Hz, the 20th harmonic will be at 22040 Hz, just slightly under the Nyquist) – above this frequency limit, components within the waveform will begin to be reflected downward.

Fortunately, in MSP there is a group of *bandlimited* oscillators (which we have already seen in section 2.1 of the practice chapter), which generate "classic" waveforms (sawtooth, triangle and square waves), and which allow us to generate the waveform at any frequency under the Nyquist without the effects of foldover. In reality, these waveforms are not written into a predetermined wavetable, but are generated by an algorithm that limits the number of harmonics based on the fundamental frequency of the oscillator: if this is 100 Hz, for example, the waveform will have around 200 harmonics, if it is 1000 Hz it will have only about 20, etc. (assuming we are using a sampling rate of 44100 Hz – for other sampling rates the number of harmonics will vary proportionately).
Let's now replace the `phasor~` and math operations in our patch with a `saw~` object, which outputs such a bandlimited sawtooth waveform (see figure 5.4).

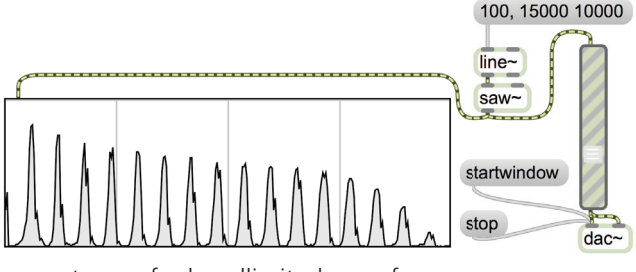

Fig. 5.4 The spectrum of a bandlimited waveform

If we run the patch and trigger the glissando, we can see in the spectroscope that the upper harmonics vanish little by little as they approach the Nyquist frequency. At the end of the ramp, when the fundamental of the oscillator passes 11025 Hz (one quarter of the sampling rate, or half the Nyquist), there is only a single component – the fundamental – left in the spectrum, since at this frequency the second harmonic would create foldover (because it would be greater than 11025 • 2, and thus above the Nyquist frequency).

5.2 QUANTIZATION AND DECIMATION

BIT REDUCTION, QUANTIZATION NOISE AND DITHERING

Within MSP, samples are represented as floating-point numbers that have a length of 64 bits (as of Max version 6). Apart from the bit used for the number's sign, there are 11 bits used for the mantissa and 52 bits for the fractional part of the number. There are therefore 2^{53} levels of quantization (for more on floating-point number theory, scaling factor, etc., see Chapter 5.2T).

Let's look at a practical example of how we can reduce the number of quantization levels of an MSP audio signal. In addition to serving as a pedagogical example, this technique can additionally be used to produce a somewhat clichéd Lo-Fi (low fidelity) sound which has already acquired a musical aesthetic of its own. Recreate the patch shown in figure 5.5.

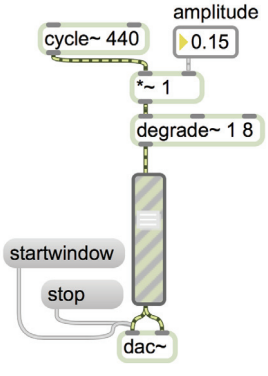

Fig. 5.5 Generating an 8-bit sound

The **degrade~** object can be used to (virtually) reduce both the number of bits used for the samples in a digital signal, as well as its sampling rate. This object has two arguments that can also be changed by using the object's second and third inlets. The first argument indicates the reduction factor of the sampling rate[6] and the second the bit depth of the signal. The example in the above figure therefore shows degradation to an 8-bit signal, which has only 256 levels of quantization.

[6] A value of 1 indicates a sampling rate which is unchanged compared to that used by the sound card: further details below.

When the amplitude of the signal (a sine wave oscillator) is at maximum we are using all quantization levels available to us, but even so, quantization noise is still clearly audible. By lowering the value of the number box which controls the sine wave's amplitude to around 0.15 we can hear a much more pronounced effect because the signal uses even fewer degrees of quantization.

Now let's take a look at a patch that will let us apply any desired number of bits and calculate the relative levels of quantization: open the file **05_01_quantize.maxpat** (figure 5.6).

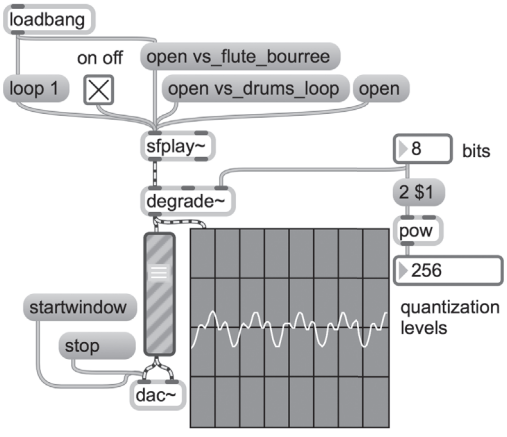

Fig. 5.6 File **05_01_quantize.maxpat**

The right hand part of the patch contains the algorithm that allows us to calculate the number of levels of quantization based on a given number of bits. Since quantization levels are a power of two raised to the number of bits (and therefore always powers of two), we have used the `pow` object to correctly calculate this value. The `pow` object receives a list of 2 values from a message box: the first (the base) is set to 2 and the second (the exponent) is the replaceable argument $1 that can be changed to any value between 1 and 24, thereby allowing us to calculate the number of quantization levels anywhere between 1 and 24 bits.

In this example patch we are also using the `degrade~` object to quantize the input sound to a desired number of bits by sending a bit depth reduction value to the third inlet.
Upon opening the patch, the `loadbang` object triggers two message boxes that tell the `sfplay~` object to open the sound file **vs_flute_bouree.wav**[7] and to

[7] This file (and indeed all the other audio files that we will use) is found inside the sound library Virtual Sound Macros, in the folder "soundfiles" Note that in Max it is not strictly necessary to add the file extension to the name of a sound file (in this case .wav). However, if there are multiple sound files in the same folder with the same name but a different extension, Max will load the file whose extension comes first, alphabetically. For example, if there are two files vs_flute_bouree.wav and vs_flute_bouree.aif, Max would load the file with the extension .aif.

activate loop mode for playback. In order to actually trigger the `sfplay~` object to play the sound file, we need to click on the `toggle` connected to it (after having clicked on the "startwindow" message, naturally), and at this point it is possible to hear the result of reducing the number of bits and consequently altering the sound's quantization. You can try to load other sounds and hear how they become degraded by lowering the samples' bit length. Remember that every time you load/open a new sound file you will need to initiate playback once again by reactivating the `toggle` connected to `sfplay~`.
In order to improve the sound quality of our purposely-degraded audio and eliminate as much distortion as possible, we can simulate dithering: open the file **05_02_dither.maxpat** (fig 5.7).

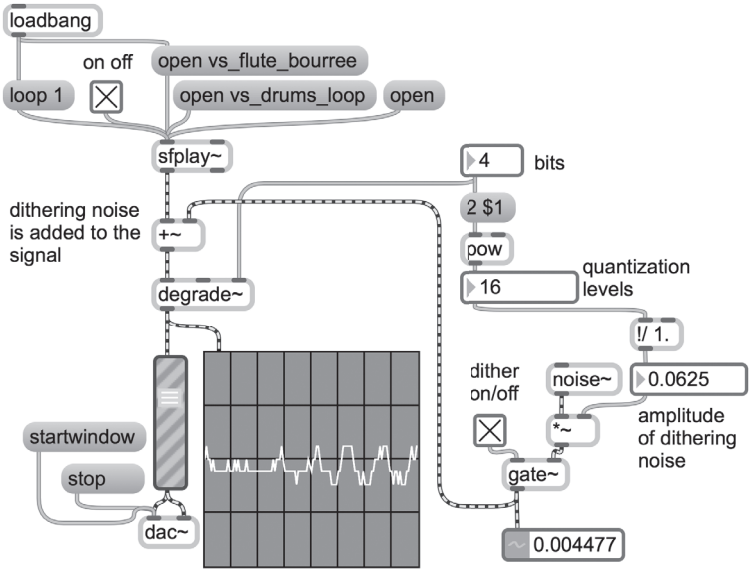

Fig. 5.7 The file **05_02_dither.maxpat**

In the right-hand part of the patch there is a `noise~` object which produces white noise that is scaled down to match the quantization span for the desired bit depth. Using the operator `!/` with an argument of "1." will generate a number representing the amplitude of a single level of quantization: for example if the level of quantization is at 4 bits, as it is in the figure, the amplitude of a single level is equal to 1/16 or 0.0625. This amplitude serves as our multiplication factor for the white noise.

The scaled noise is added to the signal (see the left-hand side of the patch), and the resulting summed signal is sent to the input of the `degrade~` object. The dithering can be turned on and off using the `toggle` in the lower-right side of the patch. This `toggle` is connected to a `gate~` object which lets an input signal in the right inlet pass through to the output if the value sent to the left inlet is 1, or will block the signal if the value sent to the left inlet is 0. When the number of bits is low (under 12) the noise added by the dithering is

very apparent, but nevertheless contributes to eliminate harmonics produced because of distortion. Try to use the percussion sound **vs_drums_loop.aif** with a very low number of bits (for example 4): although the result is very noisy, dithering does allow the drum sound to be acceptably reconstructed (especially considering we are only using 4-bit numbers as samples!); when turning off dithering the sound becomes extremely distorted.
To reduce the dithering noise we can use filters: open the file **05_03_dither_filter.maxpat** (figure 5.8).

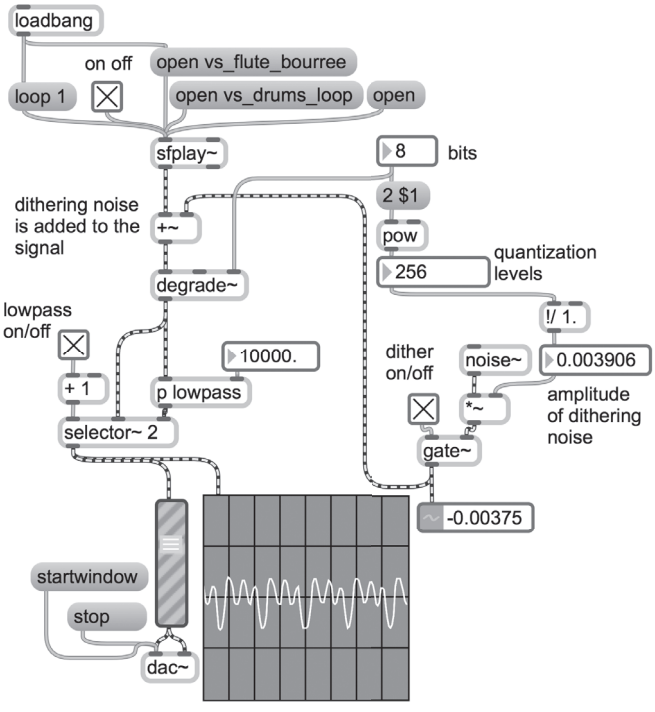

Fig. 5.8 The file **05_03_dither_filter.maxpat**

Here, we have added a lowpass filter with a steep slope (the subpatch [p lowpass] actually contains an 8th order *Butterworth* lowpass filter). It is possible to turn the lowpass filtering on and off via the `selector~` object which we have already seen in chapter 1.2P. Try to adjust the cutoff frequency of the filter so it eliminates as much of the dithering noise as possible, without excessively darkening the resulting sound.

(...)

other sections in this chapter:

Interlude C
MANAGING TIME, POLYPHONY, ATTRIBUTES AND ARGUMENTS

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1 and Chapter 5 (theory and practice) in this volume.

SKILLS

- To know how to control various types of musical time values: duration, tempo, timing, as well as the global time management system in the Max environment
- To know how to build and control a step sequencer
- To know how to manage polyphony in Max
- To know how to control attributes and arguments in abstractions

CONTENTS

- Duration, metrical tempos and timing values in Max
- The global time management system in Max
- Arguments and attributes
- Algorithms for a step sequencer
- Polyphonic patches
- Abstractions and arguments

TESTING

- Activites at the computer

SUPPORTING MATERIALS

- List of Max objects - List of attributes, messages and graphical elements for specific Max Objects - Glossary

IC.1 THE PASSAGE OF TIME (IN MAX)

For this chapter we suggest you activate both the "Scheduler in Overdrive" and "in Audio Interrupt" options in the Audio Status window, and set the Signal Vector Size between 16 and 64 samples (preferably 16).

The units we have been using to measure time until now have either been milliseconds (for instance when indicating the interval between successive bang messages output by the `metro` object) or samples (for example when indicating the delay time of an audio signal input into the `delay~` object)[1]. It is also possible to use other time measurement units within Max, and above all possible to synchronize several objects together using a ***master clock*** (a system of global time management in the Max environment) controlled by the **`transport`** object, which lets the user, among other things, activate and deactivate the global passage of time.

Rebuild the patch shown in figure IC.1.

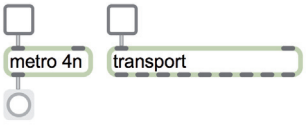

Fig. IC.1 The `transport` object

Notice that the argument for the `metro` object is not a value in milliseconds, but a symbol: 4n (we will see what this means shortly). This causes the `metro` object's timing to be linked to the master clock.
First, turn on the `toggle` above the `metro` object. Contrary to what you were probably expecting, the `metro` does not output anything. Now, turn on the `toggle` above the `transport` object and the `metro` will start to produce bangs. The `transport` object has activated the master clock, which, in turn, has activated the `metro` object. In this scenario, the `metro` object will only "run" when the master clock is active. But, what is our metronome's tempo? The symbol 4n that we provided as an argument to `metro` is a *tempo-relative time value*, and more precisely a *note value* which indicates that our tempo corresponds to one quarter note (crochet). The `metro` object will therefore output a bang every quarter note beat.

However, what is the duration of a quarter note? Its duration depends on the transport object, or more specifically on the object's "**tempo**" attribute. This attribute is used to express the number of beats per minute[2] (abbreviated bpm) and by default its value is set to 120 bpm (120 beats per minute, or one beat every half second).

[1] This object has already been discussed in section 3.5P of the first volume.
[2] This is therefore the same tempo that is used for metronome markings.

The metronome's tempo can be changed by sending the `transport` object the "tempo" message followed by a value in bpm. Modify the preceding patch so it resembles the one shown in figure IC.2

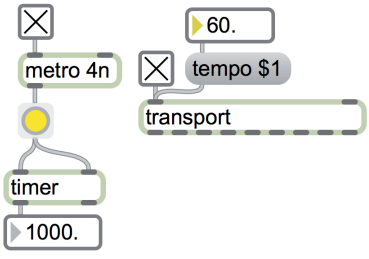

Fig. IC.2 Modifying the metronome tempo

In addition to adding the message to modify the global tempo in bpm, we have also added a `timer` object that lets us calculate the interval between successive bangs (in milliseconds). As you can see, by changing the tempo to 60 bpm, the `metro` will output one bang every second (1000 milliseconds). Try changing the tempo in bpm and see how the millisecond tempo calculated by the `timer` object changes (note that in the patch, the button above the timer is connected to both of its inlets).

The **`timer`** object is a "stopwatch" that starts when a bang is sent to its left inlet, and outputs the time that has elapsed since it started whenever it receives a bang in its right inlet. You should therefore take note that, unlike the majority of Max objects, the `timer` object's "hot" inlet is its right inlet.

ACTIVITY

Create a new patch to try out the `timer` object, sending bangs separately to its left and right inlets, paying careful attention to the values output by `timer`. Afterward, explain why connecting one `button` to both of timer's inlets, as shown in figure IC.2, allows us to calculate the time between successive bangs from a single `button` object.

As you may have guessed, there are several different symbols which are used to indicate the main note values. These are furthermore subdivided into minimal units called *ticks*. A quarter note (crochet), as we have already seen, is represented with the symbol 4n, and can be subdivided into 480 ticks. This means that an eighth note (whose symbol is 8n) corresponds to 240 ticks, and a sixteenth note (16n) to 120 ticks, and so on. Obviously the duration of the ticks depends on the metronome tempo of the `transport` object.

Here is a table of symbols and their corresponding fractional note name values, traditional note names, and values in ticks:

1nd dotted whole note (dotted semibreve) - 2880 *ticks*
1n whole note (semibreve) - 1920 *ticks*
1nt whole note triplet (semibreve triplet) - 1280 *ticks*
2nd dotted half note (dotted minim) - 1440 *ticks*
2n half note (minim) - 960 *ticks*
2nt half note triplet (minim triplet) - 640 *ticks*
4nd dotted quarter note (dotted crochet) - 720 *ticks*
4n quarter note (crochet) - 480 *ticks*
4nt quarter note triplet (crochet triplet) - 320 *ticks*
8nd dotted eighth note (dotted quaver) - 360 *ticks*
8n eighth note (quaver) - 240 *ticks*
8nt eighth note triplet (quaver triplet) - 160 *ticks*
16nd dotted sixteenth note (dotted semiquaver) - 180 *ticks*
16n sixteenth note (semiquaver) - 120 *ticks*
16nt sixteenth note triplet (semiquaver triplet) - 80 *ticks*
32nd dotted thirty-second note (dotted demisemiquaver) - 90 *ticks*
32n thirty-second note (demisemiquaver) - 60 *ticks*
32nt thirty-second note triplet (demisemiquaver triplet) - 40 *ticks*
64nd dotted sixty-fourth note (dotted hemidemisemiquaver) - 45 *ticks*
64n sixty-fourth note (hemidemisemiquaver) - 30 *ticks*
128n hundred twenty-eighth note (semihemidemisemiquaver) - 15 *ticks*

In the patch in figure IC.2, change the argument to the `metro` object from 4n to other note values, each time checking the tempo of the resulting beats with the `timer` object.

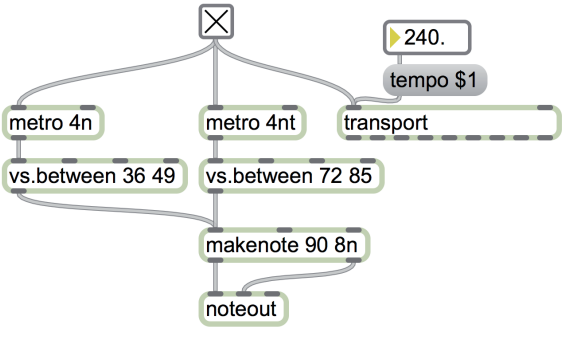

Fig. IC.3 Two against three

We have added a second `metro` object whose beat rate (tempo) is equal to a quarter note triplet (4nt). The two different tempos are used to generate two random MIDI note streams. In the time that it takes for two notes to be generated in the bass voice, three notes are generated in the treble register. Note that the second argument of the `makenote` object, corresponding to the note's duration, is 8n, which indicates an eighth note (quaver) duration – as you can

see, the `makenote` object (which we spoke about in section IB.1), can also use tempo-relative time values.

In this patch, a single `toggle` is being used to start the `transport` and two `metro` objects simultaneously – this ensures that the two note streams will start exactly at the same time. If you now change the `transport` object's tempo in bpm you will notice that the two note streams always remain synchronized, in the same 2-against-3 rhythm.
Warning: the tempo in bpm, as well as all the other commands sent to the `transport` are global! This means that they apply to the entire Max environment. If you have two patches open and they both use relative time values, starting and stopping the master clock or modifying the tempo in bpm in one of the two patches will also activate and deactivate the master clock and change the tempo of the other. It is also possible to entirely eliminate the `transport` object from both of the patches and instead use an independent window (known as the ***Global Transport***) which contains commands for the master clock (this window can be opened at any time by choosing the Global Transport option from the Extras menu). By using this window we can start and stop the master clock, change its tempo in bpm, and see the total elapsed time, among other things.

Let's return to figure IC.3. Try changing the argument of the first `metro` to 8n. Now, for every four notes in the bass we will hear 3 in the upper register, because four eighth notes (quavers) are equal in duration to a quarter note triplet.

ACTIVITY

The metrical rapport of beats between the two random note streams shown in figure IC.3 can be denoted as the ratio 2/3 (i.e., two against three). Try modifying the arguments of the two `metro` objects in order to obtain note streams with the following rhythmic ratios: 3/4, 3/8, 2/6, 4/9 (the latter may not be immediately obvious: you will need to use dotted note values for the numerator).

With the available note value symbols you can create only a certain number of metric relationships; they cannot be used to create a 4/5 beat ratio (four against five), just to give one example. In order to obtain these kinds of metric relationships it is necessary to use *ticks*.

In the case of a 4/5 beat ratio, since our quarter note is equal to 480 ticks, a quarter note quintuplet will equal 480 • 4 / 5 = 384 ticks. This value does not have a corresponding symbol in the list of note values that Max accepts. Unfortunately it is not possible to provide a value in ticks directly as an argument to `metro`, but this can still be done using setting the "**interval**" attribute, either with the object's inspector or by using a message followed by a value and a measurement unit. Modify the patch as shown in figure IC.4.

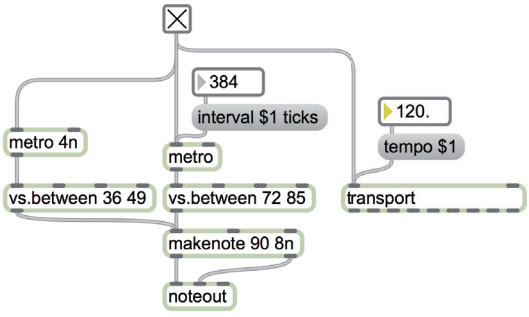

Fig. IC.4 Four against five

Here, the message "interval 384 ticks" is sent to the second `metro` using a message box. As we have already seen above, this corresponds to a quarter note quintuplet.

ACTIVITIES

- Create the following beat ratios, using the "interval" attribute for both metro objects: 3/10, 5/9, 15/8.

- Make Max do the math: modify the preceding patch so that all the user has to do is set the rhythmic ratio in two number boxes (for example the values 4 and 3 to represent 4/3), and let the patch calculate the exact value for the number of ticks. (Hint: you should only need to add one object for each `metro`.)

(...)

other sections in this chapter:

6T
DELAY LINES: ECHOES, LOOPING, FLANGER, CHORUS, COMB AND ALLPASS FILTERS, PHASER, PITCH SHIFTING, REVERSE, KARPLUS-STRONG ALGORITHM

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The contents of volume 1, chapter 5 (theory and practice) and Interlude C

OBJECTIVES

Knowledge

- To know how to use delay lines for various purposes
- To know how to simulate single and multiple echoes
- To know how to use loops, slapback, multitap delays and ping-pong delays
- To know the parameters and use of flangers, chorus and phasers
- To know the basic theory and some possible applications of comb and allpass filters
- To know how to use pitch shifting and reverse
- To know the basic theory behind the Karplus-Strong algorithm
- To know several applications for the simulation of plucked strings and percussion using the Karplus-Strong algorithm

Skills

- To be able to aurally distinguish and describe the different types of echoes
- To be able to aurally distinguish and describe the main differences between chorus, phaser and flanger
- To be able to aurally distinguish and describe the modification of the main parameters of the Karplus-Strong algorithm

CONTENTS

- Delay Lines
- Different types of Echoes
- Using delay lines for looping, chorus, flanger, phaser, pitch shifting and reverse
- Comb and Allpass Filters
- Synthesis using the Karplus-Strong algorithm

ACTIVITIES

- Sound examples and interactive examples

TESTING

- Questions with short answers
- Listening and analysis

SUPPORTING MATERIALS

Fundamental Concepts - Glossary - Discography

6.1 DELAY TIME: FROM FILTERS TO ECHOES

Delaying a signal is one of the most powerful and versatile tools we have at our disposal in the realm of computer music. A wide variety of synthesis and sound processing techniques, from subtractive synthesis to physical models, from reverb algorithms to the bulk of classic effects (which will be covered in this chapter) all have the use of longer or shorter delay lines in common as the basis of their design.

In section 3.6 we already learned that delay lines are a necessary component of digital filters. In this case the *delay time* between the input and output of the filter is extremely short, and calculated in single samples (for reference, the amount of time that separates one sample and the next, at a sampling rate of 48000 Hz is equal to 1/48000th of a second, or about 0.00002 seconds). Many different types of effects can be created using delays that range from this "microscopic" *delay time* up until a slightly larger span of just a few milliseconds.

In this chapter, in particular, we will take a look at the *flanger* (which has a constantly varying delay time between 1 and 20 ms.), *chorus* (typically using a variable delay time of 20-30 ms.), *slapback delay* – an effect giving a nearly simultaneous doubling of a sound (using a delay somewhere in the range of 10 to 120 ms), and different kinds of *echo* effects (whose delay can range between 100 ms. and several seconds). Furthermore, we will also take a look at the *comb filter* – a handy tool that can be used for different purposes, including the creation of multiple resonances that have a harmonic relationship.

Finally, we will also look at two different implementations of the *allpass filter*: the first can be used alongside a *comb filter* to create reverberation effects (which we will cover in more detail later on, in the third volume), and the second can be used to make a *phaser*. Figure 6.1 shows an illustrative and helpful graph with time values ranges that should give you an approximate idea of the delay times necessary to obtain different effects. The ranges mainly serve as rough guidelines that can be modified as necessary in order to appropriately adapt them to different kinds of input sounds.

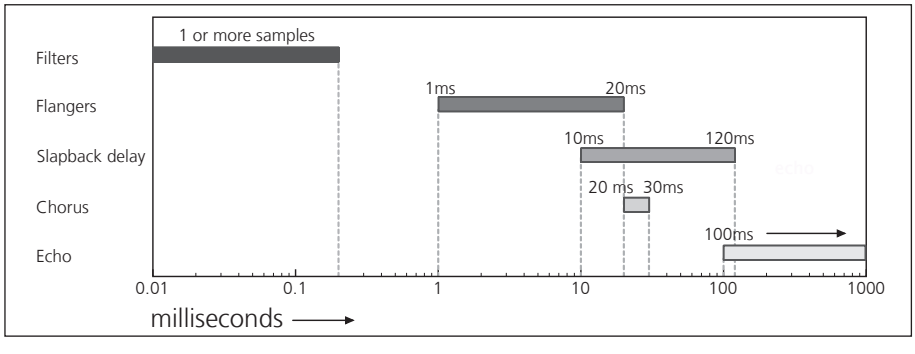

Fig. 6.1 Different delay times from filters to echoes

6.2 ECHOES

The *echo* effect – i.e., the repetition of a sound – can clearly be perceived if the repetition of the sound takes place in a time frame greater than what is known as the *Haas zone* (25-35 ms.). If, on the other hand, the repetition arrives to the listener in a shorter amount of time, the listener will have difficulty perceiving the second sound as a separate one. Moreover, in the case of sounds whose attack is slow, the sound will tend to be perceived not as a separate sound at all, but rather a sound fused together with the first.

To simulate a simple echo effect we need to create an algorithm with a delay, which takes an input audio signal and repeats it at its output after a given amount of time that can vary anywhere between a few milliseconds and several seconds. The delayed sound (defined as the "*wet*" sound) can optionally be added to the original sound (the "*dry*" sound). If we only want to hear the sound of the effect itself (the *wet* sound), we just need to give the *dry* sound an amplitude of zero. The control of the proportion of the dry and the wet sound is known, appropriately enough, as *balance*, and is often defined as a percentage where 100% indicates only the wet sound, 0% indicates only the dry sound, and 50% indicates equal amounts of *wet* and *dry*. Figure 6.2 shows a simple delay example.

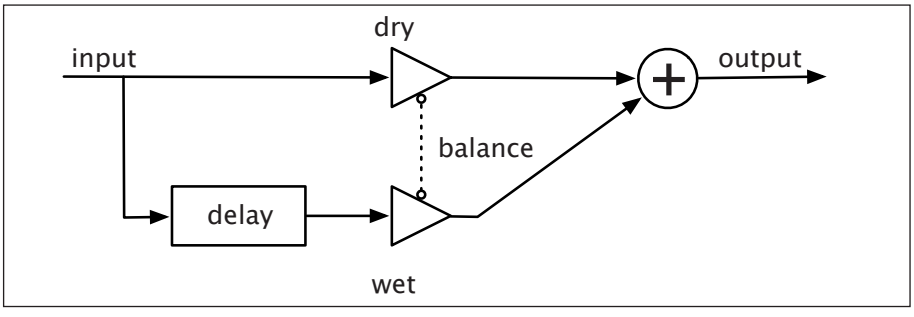

Fig. 6.2 A simple delay example

INTERACTIVE EXAMPLE 6A.1 • *Echo effects with delay times inside and outside the Haas zone.*

If there is more than one repetition of the input sound we effectively obtain a multiple echo. In order to create multiple echoes, we need to add *feedback* to our algorithm.

Feedback amounts to being able to add the output of a delay back into its input; an example of delay with feedback is shown in figure 6.3.

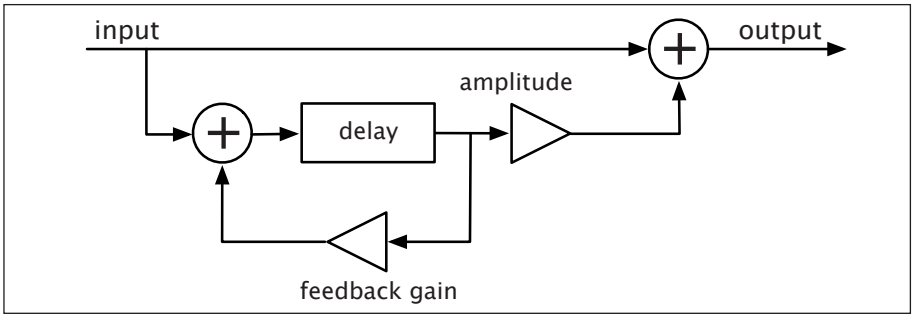

Fig. 6.3 Block diagram for delay with feedback

INTERACTIVE EXAMPLE 6A.2 • *Multiple Echoes*

In this manner, the delay effect can be repeated many times, by adjusting the *feedback* gain. If the gain is zero, there will be no feedback, and therefore the repeated sound will happen just once. By slowly increasing the gain we can get more and more repetitions of the sound.

Technically speaking, the feedback gain is just a multiplication factor for the signal (i.e., a scalar), whose value will typically vary between 0 and 1. With a gain of 50%, for example the feedback signal will be multiplied by 0.5 (in other words its amplitude will be reduced by half with each repetition of the sound). To give an example, if the multiplier is set to 0.5, and the initial amplitude is 1, the various attenuated repetitions will have amplitudes of 0.5, 0.25, 0.125, and so on.

When using a gain of 0% the signal will be multiplied by zero (and therefore silenced), whereas with a gain of 100% it is multiplied by 1 (and is therefore not modified). The percentage of gain actually represents the ratio between the amplitude of the output signal and that of the signal which is sent back to the input. Therefore, if the multiplier value is 0.99 (equal to 99% gain) the output sound will only be reduced by 1% each time the signal is sent back to the input. A multiplier of 1 (equal to 100% gain) is not advisable as it will result in a virtually infinite repetition of the same amplitude, and if other audio signals are meanwhile also sent to the input, distortion is likely to happen once the signal from the feedback loop has been added to it. Obviously, multipliers greater than 1 are also not recommended, for the same reason.

ACCUMULATION OF DC OFFSET

It is possible that a sound's waveform (figure 6.4) might contain a positive or negative *DC offset* (refer to section 5.3).

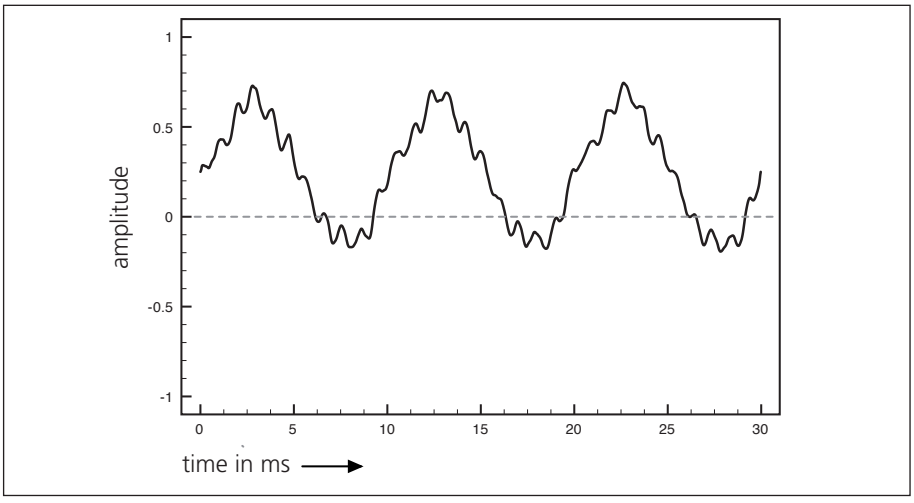

Fig. 6.4 A sampled sound with a significant *DC offset*.

This imbalance might not be problematic when directly playing back the sound. However, you should be aware that when creating a delay with feedback, the DC offset of the various repetitions will accumulate and can easily take the sound out of the amplitude range of the audio interface. To rectify this potential problem, you simply need to add a highpass filter with a very low cutoff frequency (for example 10 Hz) – since DC offset is simply a signal with a frequency of 0 (as already explained in section 5.3), it can be easily filtered out using such a filter.

SIMULATION OF TAPE DELAY AND ANALOG DELAYS

The term ***tape delay*** refers to old echo systems in which a sound was recorded to tape using one record head, and played back using another playback head, allowing the possibility of feedback. The delay time was dependent on the distance between the two heads and the speed of the tape. This type of echo could have a very long delay time (using two tape recorders placed far from one another), or have multiple echoes via the use of multiple playback heads. Some systems used magnetic discs similar to hard disks instead of tape. The repeated sound in these systems was slightly filtered each time it passed from the playback head to the record head, due to a loss of amplitude in the high frequency range that is typical of analog recording technology. Therefore, in addition to the delay itself, a digital simulation of tape delay requires using filters that will modify the spectral content of the sound each time it passes through the feedback loop. Alongside a lowpass filter, you can also add a highpass filter inside the feedback loop in order to eliminate the accumulation of low frequencies.

Such a system can also be used to simulate another type of analog delay from a later era than tape delay, in which delay lines were no longer recorded to magnetic media but to solid state components (the delay technology used inside analog guitar pedals).

INTERACTIVE EXAMPLE 6A.3 • *Tape Delay - Analog Delay*

(...)

other sections in this chapter:

6.7 ALLPASS FILTERS
Schroeder Allpass Model
Second-Order Allpass Filters

6.8 THE PHASER
The Parameters of the Phaser
Depth
Range
Feedback
Speed/rate
Q Factor

6.9 PITCH SHIFT, REVERSE AND VARIABLE DELAY
Variable Delay Without Transposition

6.10 THE KARPLUS-STRONG ALGORITHM

- Sound Examples - Interactive Examples
- Questions with Short Answers
- Listening and Analysis
- Fundamental Concepts - Glossary - Discography

6P
DELAY LINES: ECHOES, LOOPING, FLANGER, CHORUS, COMB AND ALLPASS FILTERS, PHASER, PITCH SHIFTING, REVERSE, KARPLUS-STRONG ALGORITHM

LEARNING AGENDA

PREREQUISITES FOR THIS CHAPTER

- The contents of volume 1, as well as chapter 5 (theory and practice), Interlude C and 6T

OBJECTIVES

Abilities

- To be able to distinguish and control different types of delay lines
- To know how to build delay line effects such as echo, echo with feedback, tape delay simulation, slapback delay, ping-pong delay, multitap delay, multiband-multitap delay
- To know how to build and control loops using delay lines
- To know how to build flanger and chorus algorithms
- To know how to program and use different types of comb and allpass filter algorithms, and to construct harmonic resonators and phaser effects.
- To know how to build delays that control pitch shifting, reverse and variable delay effects including glissandi and changes of delay time without transposition
- To know how to program the Karplus-Strong Algorithm to simulate the sound of plucked strings and other sounds
- To know how to use delay lines for Max messages

Skills

- To be able to create a brief composition based on the use of sampled sounds, using loops, reverse, different types of delay lines with various modifications.

ACTIVITIES

- Building and Modifying Algorithms

TESTING

- Activities at the computer

SUPPORTING MATERIALS

- List of Max objects - List of attributes, messages and arguments for specific Max objects

6.1 DELAY TIME: FROM FILTERS TO ECHOES

To create a delay line with Max we can use the `delay~` object, which delays a signal a certain number of samples, as we have already seen in section 3.6P of the first volume. Rebuild the patch shown in figure 6.1.

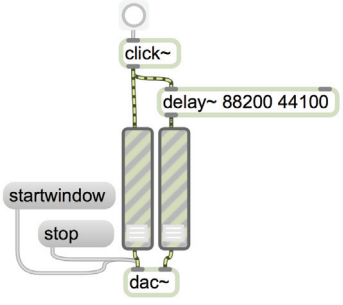

Fig. 6.1 The `delay~` object

The signal output by the `click~` object is delayed by 44100 samples on the right channel[1]; if the sampling rate of your audio interface is 44100 Hz, the delay would be equal to one second.
It is easy to understand that specifying the delay time like this – in samples – is rather inconvenient if we need to express a delay in seconds, since the number of samples representing one second could change depending on the sampling rate being used, and thus we could never be sure that the delay time would be the same in every situation. Generally, the `delay~` object is used in cases where we need to delay a signal by a specific number of samples (as we did for the filters in section 3.6P). Nonetheless, it is always possible to express the delay time in other formats by using the appropriate syntax for the different time units that we learned in section IC.1. Modify the previous patch as shown in figure 6.2.

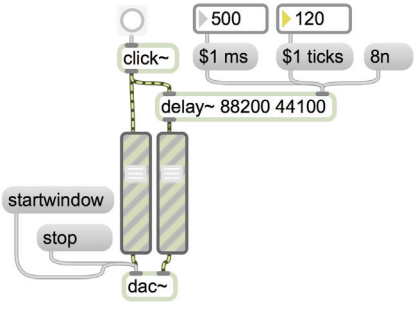

Fig. 6.2 Specifying the delay time with different time formats

[1] Remember that the two arguments to the object represent the amount of memory reserved for the delay (in other words the maximum possible obtainable delay) and the delay actually created by the object, respectively. Both values are expressed in samples by default.

The figure shows how to specify the delay time for the **delay~** object in three different formats: milliseconds, ticks and note value. The desired value should be sent to the second inlet of the object, thereby replacing the argument 44100. Please refer to section IC.1 (subsection "Time Values") for the complete list of available formats[2].

Later in this chapter, we will often be using delay times that vary over time. In this case, in order to avoid discontinuities, the delay time will be sent as a signal, not as a Max message. It is possible to vary the delay time of the **delay~** object using a signal, but this can only be used to define a time expressed in samples, bringing us right back to the "inconvenient" situation we found ourselves in, above.

If we need a continuously variable delay it is always preferable to use the pair of objects called **tapin~** and **tapout~**, which respectively write and read a signal (with a pre-defined delay) into a given allocated memory location. The delay time for these objects is expressed in milliseconds, and can be modified using a signal, or via Max messages. Furthermore, with a **tapin~** and **tapout~** pair, it is possible to create a feedback loop in the delay line. In other words, it is possible to send the delayed sound back into the input of the delay line – something that is not possible with the **delay~** object. One limitation inherent in a **tapin~** and **tapout~** pair is that you cannot create a delay time less than the Signal Vector Size (see section 5.1P), whereas with the **delay~** object, as we already have seen, we can even create a delay as small as one sample. The **tapin~** and **tapout~** objects will be very useful throughout the next sections, so you should be sure to understand and learn how to use them well. Reconstruct the patch shown in figure 6.3.

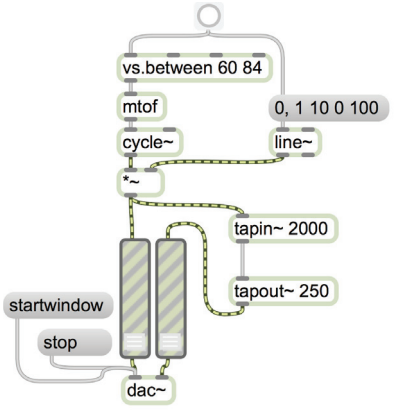

Fig. 6.3 The **tapin~** and **tapout~** objects

In the upper part of the patch we have a simple algorithm that creates a brief sound with a random frequency with each successive bang. This sound is

[2] Another possibility is to convert a value expressed in milliseconds into a value in samples using the **mstosamps~** object which we have already seen in section 2.4P of the first volume.

immediately sent to the left channel, and delayed with a `tapin~`/`tapout~` pair on the right channel. Let's take a closer look at how these objects work.

The `tapin~` object takes one argument which defines the maximum possible delay time in milliseconds (not samples). This object actually "takes over" (allocates) a chunk of memory whose length in samples is determined by dividing the argument by 1000 (to convert the duration to seconds) and then multiplied by the sampling rate. In the case shown here, the argument is 2000 (milliseconds), which equals 2 seconds when divided by 1000, so if the sampling rate is 44100 the amount of memory allocated by the object would be 88200 samples. This memory is used by `tapin~` to write (i.e., record) the signal it receives. Each time it arrives at the end of the allocated memory (after 2 seconds, in our case) it begins again from the memory's starting location, overwriting what had been recorded previously. Technically speaking, this is known as a circular buffer (see section 6.2 in the theory part of this chapter), which we could imagine as a loop of magnetic tape running past a record head (represented by `tapin~`). The `tapout~` object, on the other hand, functions as a playback head, and reads the same circular buffer at a certain distance (given by the argument) with respect to the record head of the `tapin~` object it is connected to, thereby creating a delay of the desired length. In the case of figure 6.3, the circular buffer is 2 seconds long (2000 milliseconds), as indicated by the argument to `tapin~`, and the current delay time is 250 milliseconds, as indicated by the argument to `tapout~`. We will see how to vary these arguments in the next section.

We would also like to point out that the cable connecting `tapin~` and `tapout~`, as you can see, is not an audio cable (since it is a plain grey line), but a connection that allows `tapin~` and `tapout~` to both share the same circular buffer. If we connect a `print` object to the outlet of `tapin~` and click on the "startwindow" message box, we can see the "tapconnect" message printed in the Max window. The `tapin~` object sends this message to the `tapout~` object each time the DSP engine is activated, and is used to allow both objects to share the same pre-allocated memory zone.

In order to create two read locations along one delay line, it is possible to connect several `tapout~` objects to one single `tapin~`. Alternatively, you could provide `tapout~` with multiple arguments, each one corresponding to a read location (in milliseconds).

6.2 ECHOES

To create an echo effect using the `tapin~` and `tapout~` objects, rebuild the patch illustrated in fig. 6.4.

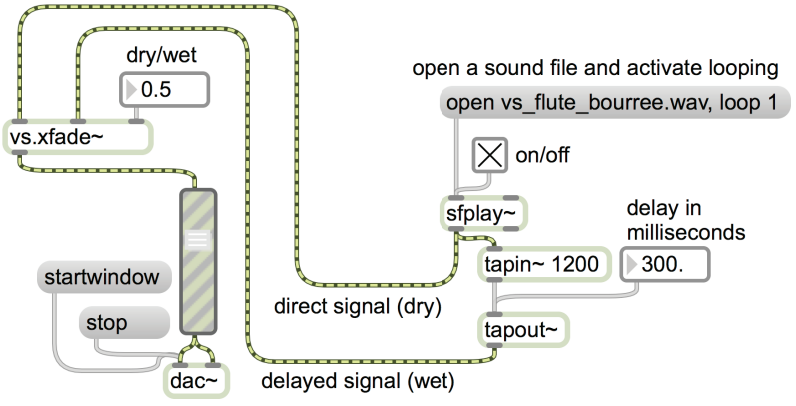

Fig. 6.4 The Echo effect

Note that the message box on the right has two messages separated by a comma: the first opens the sound file, whereas the second sets the loop mode for `sfplay~`. As we already know, the sound file vs_flute_bouree.wav is contained in the Virtual Sound library and Max should be able to find it without your needing to specify the entire file path, presuming you have installed this library correctly. If you want to load a different sound, you will need to send `sfplay~` the "open" message, which will open up a system dialog allowing you to select the sound file you want to use. If you want to test out the effect using an instrument or your voice, you will need to connect a microphone to your audio interface and replace the `sfplay~` object with `adc~`. You can do this the majority of the patches that we will talk about in this chapter, but just be careful not to create feedback between the speaker(s) and microphone! When starting out, we suggest using headphones, until you are confident about using the effects in a live situation.

The `vs.xfade~` object (which we already covered in section 3.5P of the first volume) located on the left lets you mix the direct (dry) signal coming from `sfplay~` with the delayed (wet) signal coming from `tapout~`. Remember that the number sent to the third inlet is used to adjust the proportion between the signals connected to the first two inlets, in other words it controls the *balance*: a value of 0 means only the first signal will be output, a value of 1 means only the second will be output, a value of 0.5 indicates an equal mix of both, and so forth.

We can set the delay time by sending a numerical message to `tapout~`. Click on the `toggle` connected to the `sfplay~` object and try changing the delay time to hear the different possible resulting echo effects. Remember that in

order to hear a distinct repetition of the sound, the delay time has to be above the Haas zone, which is in the range of 25-35 milliseconds. Using the sound file shown in the figure (that of a flute with a fairly soft attack) you really only sense a clear doubling of the sound when the delay is above 50 milliseconds. Since the bourée in the sound file is played at 100 bpm, try setting the delay time to 300 milliseconds (corresponding to a delay of an eighth note, or quaver) and multiples of it. Using the `tapin~/tapout~` pair you can also introduce feedback in order to create multiple echoes (something that is impossible to do with `delay~`). Modify the patch as shown in figure 6.5.

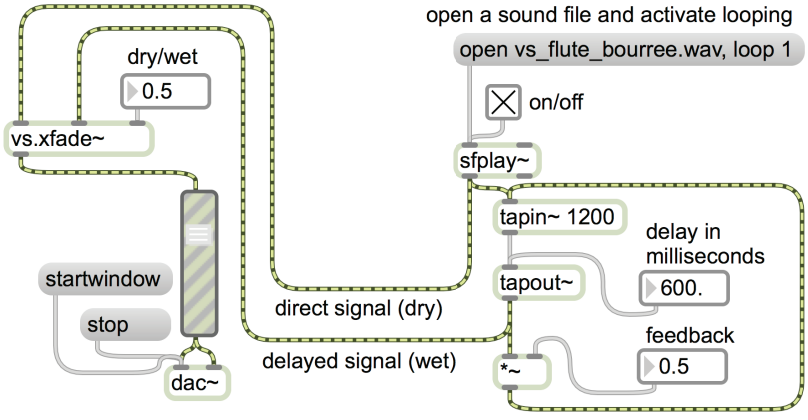

Fig. 6.5 Echo with Feedback

Here, the signal coming from `tapout~` is scaled using a signal multiply object controlled by a floating-point number box (under the label "feedback"). The scaled signal is sent back to the input of `tapin~`, where it is added to the new signal coming from `sfplay~`.

But be careful! In order to avoid distortion, you should set limits on the values output by the floating point number box that controls the feedback. To do this, open the object's inspector, go to the Value category and set the "Minimum" and "Maximum" attributes to 0 and 0.99, respectively. These attributes limit the range of values that can be output by the number box, so you can easily avoid setting a feedback value greater or equal to 1, and therefore be sure that the repetitions of the echo will gradually die out.

As we mentioned earlier, feedback can be created using a `tapin~/tapout~` object pair, but not with `delay~`. In fact, MSP does not actually allow *feedback loops*[3] within the flow of its signal chain.

[3] Creating a feedback loop, which causes an error in MSP, should not be confused with looping an audio file, which we discussed in Chapter 5..

By carefully observing the two loops shown in figure 6.6, we can immediately see the difference between a loop made with the `delay~` object (which will stop the DSP engine) and one made with `tapin~` and `tapout~`: on the left side, the audio signal flow is connected in a completely closed loop (all of the connections are made with yellow and black audio cables), whereas on the right the audio loop remains open, because the connection between `tapin~` and `tapout~` is not made with an audio cable.[4]

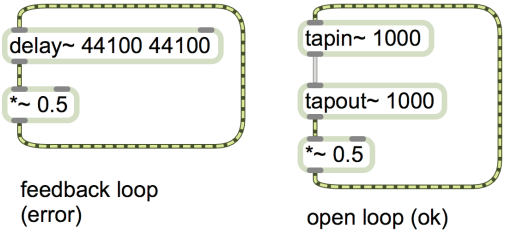

Fig. 6.6 A feedback loop

ACTIVITY

In order to avoid the accumulation of DC offset in the patch shown in figure 6.5, insert a first-order highpass filter (see section 3.4P in the first volume) between the `sfplay~` and the delay algorithm. The cutoff frequency should be set lower than the audible bandwidth (for example 10 Hz) in order not to eliminate any meaningful components present in the input signal.

[4] As already stated, with this connection, the `tapin~` object sends the "tapconnect" message to the `tapout~` object, allowing both objects to share the same circular buffer without an explicit audio connection.

SIMULATION OF TAPE DELAY

In order to simulate Tape Delay we need to insert a lowpass filter after the output of `tapout~` (see figure 6.7).

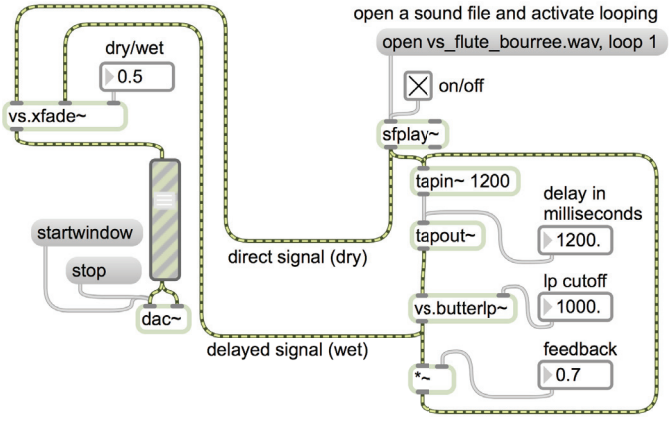

Fig. 6.7 "Tape Delay" with lowpass filter and feedback

Here, we used a Butterworth filter (`vs.butterlp~`, see chapter 3) to do this. Try to rebuild this patch by adding the parameters shown in the figure. You will hear that every copy of the delayed sound is slightly darker in timbre than the previous one. To hear just the filtering effect by itself, click on the "on/off" `toggle` to start the sound file and then, after about a second, click again to stop it.

To avoid continually accumulating low frequencies, as well as to help make the repetitions themselves clearer, we can also add a highpass filter such as `vs.butterhp~` (figure 6.8). As a welcome side effect, this filter also eliminates the accumulation of any DC offset that the input signal might possibly contain.

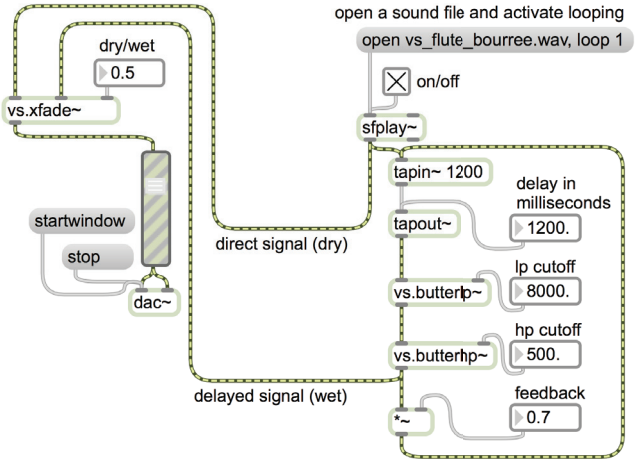

Fig. 6.8 Adding a highpass filter to the tape delay

Try using different cutoff frequencies; using a relatively steep cutoff for both filters in the medium high frequencies (4000 Hz for the lowpass and 1000 Hz for the highpass) lets you create a "telephone" sound effect, for example.

ACTIVITY

Starting with the patch in figure 6.7, use different types of filters, such as the bandpass filter `vs.butterbp~` or a filterbank using `fffb~` (see section 3.7P), in order to modify the spectral content of the delayed signal. In all cases, be very careful of the Q values you use! They should definitely not go under 0, nor be to high, for that matter. Always start using a feedback value of 0, and then raise it slowly and cautiously.

(...)

other sections in this chapter:

7T
DYNAMICS PROCESSORS

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5 and 6 (theory and practice) and Interlude C

OBJECTIVES

Knowledge

- To know the different types of dynamic processors
- To know the possible uses for envelope followers
- To know the uses and parameters of compressors, de-essers and limiters
- To know the use and parameters of expanders and (noise-)gates
- To know the uses and difference between:
  downward compression, downward expansion
  upward compression, upward expansion, parallel compression
  multi-zone compressors, multi-band compressors
  compressors with external time threshold, side chain and ducking,
  adaptive gate and ducker, triggering gates, gate sequencers
  feedback with controlled dynamics

CONTENTS

- Envelope followers
- Compressors and Limiters
- Expanders and Gates
- Downward, Upward and Parallel Compression
- Downward and upward expansion
- Side chain and ducking
- Technical and creative uses of dynamics processors

ACTIVITIES

- Sound Examples

TESTING

- Test with short answers
- Listening and analysis test

SUPPORTING MATERIALS

- Fundamental Concepts – Glossary – Discography

In Section 5.1 we spoke of the dynamic range as being the ratio between the maximum and minimum amplitudes that can be represented by a given piece of hardware, software or storage medium. This is a purely technical description, based on the number of bits used. Within this range, defined by the number of bits, we can make technical and expressive choices that could be different for each piece or each sound. Although we may be working with a system that technically has a dynamic range of 90 dB, we could decide to compose a work for television, whose dynamic range would therefore be reduced, say, to 15 dB. In this context, the use of dynamics processors does not have anything to do with the number of bits available (this is predetermined by the working environment settings when creating a new project and thus cannot be changed) but rather on technical and expressive choices that can be used within a given system in order to change the dynamic range of a sound or piece.

Dynamics Processors are devices that process a sound (or a series of sounds or even an entire piece) by transforming its dynamic range for a wide variety of purposes, either technical or creative. Remember that dynamic range is expressed as a ratio, or difference, between maximum and minimum amplitude, and is therefore a very different concept from absolute amplitude. For example, you could have a piece of music whose dynamic range is 20 dB with a maximum amplitude of 0 dB, and another with the same dynamic range whose maximum amplitude is -3 dB. We will soon see how to work with both of these parameters, but for the moment it is important just to remember that the main purpose of dynamics processors is to transform the dynamic range in different ways.

In this chapter we will look specifically at envelope followers / envelope shapers, different types of compressors and expanders, limiters and gates.

7.1 ENVELOPE FOLLOWER

The **envelope follower** (sometimes called a peak amplitude follower or envelope detector) performs the function of extracting the envelope of a sound by measuring the amplitude of its waveform's positive-valued peaks. The envelope follower produces a control signal based on a series of amplitude values extracted from a given sound A. This control envelope can then be imposed upon another sound B (by simply multiplying it by the extracted envelope), or used to control the center frequency of a filter or other effect parameters. There are an endless number of possible applications: percussive envelopes can be applied to continuous sounds, or the envelope of a recording of the waves of the sea could be applied to the sound of a chorus, as will be demonstrated in the examples that follow.

You could even remove the envelope from a sound by dividing the sound's signal by its own envelope. Doing this actually cancels out the sound's envelope by giving it a constant value of 1.[1] The resulting sound will thus have no

[1] Any number divided by itself is equal to 1.

changes in dynamics. This is useful, because once its dynamics have been flattened out, the amplitude envelope of an entirely different sound can be effectively applied to it, by multiplying the sound and envelope together.

You could also invert a sound's envelope, so that when the original sound is at its maximum peak amplitude, the inverted sound will be at its minimum. These types of operations are sometimes referred to as envelope shaping.

Another function we will look at is often called *balance* in some audio programming languages[2] (not to be confused with the same term which is commonly used to indicate control of stereo spatialization). This technique is used in cases where a filtered sound, for example, is significantly weaker (or louder) than the original, unfiltered sound. A typical example would be when the center frequency of a bandpass filter is not present or has a very low amplitude in the original sound. Using the balance algorithm, the envelope of the original sound can be applied to the resulting filtered sound.

SOUND EXAMPLE 7A.1

a) Envelope of a piano imposed on the sound of a flute
b) Envelope of a snare drum applied to a trumpet
c) Excerpt from Cipriani, A., *Aqua Sapientiae/Angelus Domini*: envelope of sounds of ocean waves applied to contrapuntal voices.

An envelope can be measured using different types of systems: one of these, described by Dodge and Jerse, uses a technique called *rectification*, which amounts to transforming the amplitude values of the samples that make up a sound into absolute values (i.e., without a + or - sign). This way all negative sample values will become positive. The "rectified" signal is then sent to a lowpass filter (with a sub-audio cutoff frequency) which is used to round out the sharp edges of the waveform. If the filtering is too excessive, the resulting curve will be too far from the original, but if the filtering is too light you will notice some bumps and a general "edginess" to the envelope. Therefore, it is important to know what kind of lowpass filter should be applied, based on both the complexity of the envelope in sound A, and the eventual use the envelope will be put to as a control signal. Nonetheless, it is always good to have higher definition when extracting complex envelopes.
Another system is based on calculating the average of the absolute values of the of the samples' amplitudes. In this case, the degree of definition of the envelope will be due to the number of samples uses to calculate the average: the more samples that are used, the less accurate will be the envelope's profile.

[2] Cfr. Dodge and Jerse 1997, p. 181

The control signal generated by an envelope follower can also be applied to filter parameters, as we mentioned earlier. For example, you could decide that the control signal from an amplitude follower (with appropriately scaled values) could be used to control the center frequency of a bandpass filter or the cutoff frequency of a lowpass filter. These filters can affect a second sound, or be used to modify the original sound itself. (Be careful not to confuse the rounding function of the first lowpass filter, which helps create the control signal output by the envelope follower, with the second lowpass filter that is itself controlled by the envelope follower's output, in order to affect another sound.) Using this system you can also obtain a result similar to that of a VCF of an analog synthesizer (described in Section 3.5T), in which the frequency of a filter depends on the amplitude envelope, following its profile, and resulting in a more brilliant sound when the amplitude is at its maximum and a darker sound as the amplitude decreases. This actually corresponds to the behavior of a wide range of acoustic instruments.

SOUND EXAMPLE 7A.2

a) Envelope follower controlling the filtering of a second sound (bandpass)
b) Envelope follower used to control the filtering of a second sound (lowpass)
c) Envelope follower used to control the filtering of the same sound whose envelope was extracted (bandpass)
d) Envelope follower controlling the filtering the sound whose envelope was extracted (lowpass).

7.2 COMPRESSORS AND DOWNWARD COMPRESSION

THE COMPRESSOR

A **compressor**[3] is a dynamics processor used to reduce the dynamic range of a sound. There are many uses for compressors – both technical and creative – so this remains an important device in the electroacoustic sound-processing chain. Before going into the many possible uses for this device, let's try to describe something similar to what happens inside a simple compressor. Imagine we have an amplifier with a single volume knob, and an input sound that varies in unpredictable ways. We want this sound to always maintain a

[3] Warning! The word compression can be ambiguous, since it is used both for data reduction (as we discussed in section 5.3) and compression intended to reduce the dynamic range of a sound (which is being covered in this section). For this reason, in Chapter 5 we referred to the reduction of data specifically as "data compression," leaving the simple term "compression" to indicate a reduction in dynamic range.

high output sound pressure level, but not above a certain intensity. How do we do this? As soon as we realize the sound has gone above a certain threshold, we can immediately reduce the gain (in other words, lower the volume), and when the sound goes below the threshold we can bring the gain back to its original position.

- What is the threshold above which we lower the volume?
- How much should we lower it?
- How fast should we turn the knob to lower the volume?
- How fast should we return the knob to its original position?

By answering these questions we get closer to understanding some of the basic concepts of compressor parameters:
Threshold: over what threshold in dB does a compressor begin to take effect?
Compression ratio (or slope): how is the amplitude scaled when it is above the threshold?
Attack time: for each increase in amplitude, how much time, starting from the moment when the input signal exceeds the threshold, does it take the compressor to reach the determined ratio via a reduction in gain?
Release time: for every decrease in amplitude, how much time does it take the compressor to reach the determined ratio via an increase in gain?

It is also important to mention here that a lot of literature on the subject insists on the false notion that the release happens only when the sound drops again below the threshold or that the attack happens only when it goes above the threshold. In reality, the release happens with every decrease in amplitude, even when the sound remains above the threshold. Only the final release that takes place will actually correspond to the return of the sound's amplitude to the level below the threshold.[4] In the same way, as long as the sound is above the threshold, an attack will happen each time there is an increase in amplitude.

Some of the possible uses for a compressor are:

- **to make the voice of a speaker more easily understandable.** In the case of something like a documentary film, there might be a section with both music and ambient sound where we additionally might want to superimpose the voice of a speaker without reducing the level of the other two signals very much, but be able to keep the voice clearly comprehensible. In such a scenario there could be some phonemes pronounced by the speaker that could be easily masked by the music, while others remained clear (i.e., having the right intensity). A compressor could be used to attenuate only the loudest parts, without affecting the soft parts, so that, once leveled-out, the entire signal could be globally increased to be able to be understood above the music.

[4] For more information, see Izhaki, R., 2012, pp. 280-1

- **to create musical effects:** the very heavily compressed sound of an electric guitar is well-known in the rock world – this practically makes the attack of each note disappear, eliminating the characteristic sound of the plucked string. Also, strong compression on a voice will compress vocal peaks, allowing all the less evident vocal sounds like breath and sounds produced by moisture in the mouth to come to the fore..

- to compress the dynamic range in order **to be able to increase the overall level of the signal**. If a given piece of music has some peaks at 0 dB but the vast majority of the sounds in the piece have a much lower intensity, it is impossible to increase the amplitude of the entire signal because the peaks, which are already at the maximum amplitude 0 dB, will become distorted. Therefore, if we apply a compressor to reduce all the peaks to -3 dB, but leaving the low intensity sounds untouched, we can then increase the overall level of the compressed signal by 3 dB, bringing the peaks back to 0 dB, but also increasing the level of all the other sounds present in the signal. If this function is applied to an entire audio file (for example to an entire piece of music during the mastering process) as well as to a sound or series of sounds, using a limiter (which we will talk about later in this chapter) is generally preferred.

- **to level-out unequal sound pressure of selected instruments:** for example with wind instruments which tend to produce greater intensity for high frequencies than they do for lower ones. In this case, as we will see, we could use a multi-band compressor.

- **to reduce sibilance in vocal sounds**. This is possible using a special combination of a lowpass filter and a compressor. For this purpose there exists a special compressor called a de-esser (useful when a speaker or singer has S sounds that are too sibilant), which we will discuss at the end of this section.

SOUND EXAMPLE 7B.1 • Examples of Compression (examples before and after compression)

a) Voice of a speaker rendered more understandable using compression
b) Compression as a means to change timbre (electric guitar example)
c) Use of compression to increase the overall level of a signal
d) Use of compression to level the unequal sound pressure of some instruments
e) Use of a de-esser to reduce sibilance of a voice.

COMPRESSOR PARAMETERS AND DOWNWARD COMPRESSION

There are two distinct types of compression: **downward compression**, which attenuates peaks above a threshold (this is the kind of compression we have been discussing up to now), and **upward compression**, which increases the level of low intensity zones (which we will discuss later).

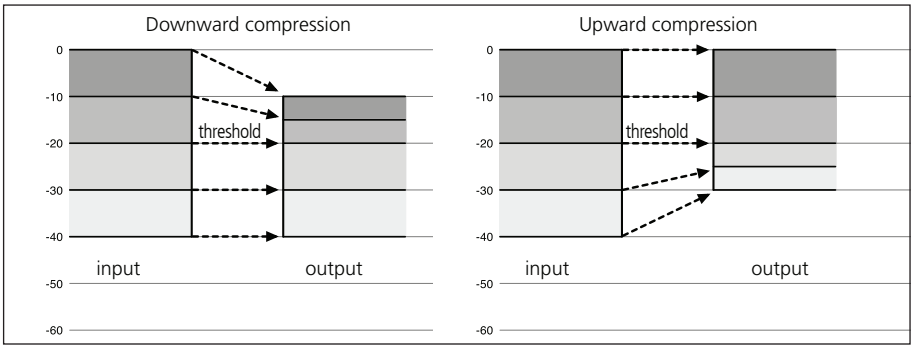

Fig. 7.1 Downward and upward compression[5]

We can't provide you with any hard and fast rules for regulating all the parameters of dynamics processors, because one situation can be vastly different from another, and there are often many different strategies to achieve the same results. Therefore, we will simply provide you with some rudimentary information along with a little bit of advice. In this section we will take a look at the various parameters of a compressor in detail and analyze them from the point of view of *downward compression* (i.e., attenuating peaks above a threshold).

(...)

[5] Image adapted from Izhaki, R., 2012, pag. 263

other sections in this chapter:

7P
DYNAMICS PROCESSORS

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5 and 6 (theory and practice), Interlude C and Chapter 7T

OBJECTIVES

Abilities

- To be able to apply the envelope of one sound to another sound or other parameters
- To be able to construct algorithms for envelope followers, downward and upward compression, parallel compression, de-esser, limiters and live normalizers
- To be able to apply appropriate parameter values to any kind of dynamics processor.
- To be able to build algorithms for gates, downward or upward expanders, side chain and ducking
- To understand how to use dynamics processors for both technical and creative purposes

CONTENTS

- Envelope Followers
- Compressors, Limiters, Live Normalizers and de-essers
- Expanders and Gates
- Creating Downward and Upward compression and expansion
- Creating parallel Compression
- Creating external Side chains and ducking
- Creating adaptive gates and duckers, triggering gates, gate-sequencers and feedback with controlled dynamics

ACTIVITIES

- Building and Modifying Algorithms

SUPPORTING MATERIALS

- List of Max objects - List of commands, attributes, and parameters for specific Max objects

7.1 ENVELOPE FOLLOWERS

There are different ways to create an envelope follower in Max. As we will discover, each of these methods has its own qualities that make it more or less useful for different possible applications. We will analyze the different ways an envelope follower can be designed, and consider the most effective uses for each of them.

The first object that we will learn is **`average~`**. Rebuild the simple patch shown in figure 7.1.

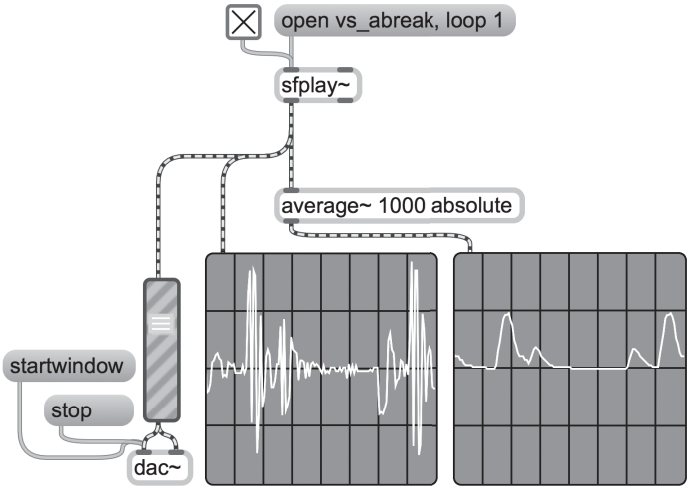

Fig. 7.1 Envelope following with `average~`

This object creates a signal representing an average of the input sample values. As you can see in the figure, its output signal follows the dynamic profile – i.e., the envelope – of the input signal. The first argument to `average~` indicates the number of input samples that will be taken into consideration when calculating the mean value, or average (this is 1000 in the figure). The second argument ("absolute" in the figure) specifies the mode that will be used to calculate the average. The three possible modes are:

1) *bipolar* (the default mode): this mode simply calculates the average of the input samples. More specifically, the sample values are summed and the result is divided by the number of samples used for the sum. For audio signals this average tends to be near zero, especially when using a large number of samples (at least as many as there are in one cycle of an input waveform), because the equivalent quantities of positive and negative values in a bipolar signal tend to cancel each other out.

2) *absolute*: here, the average is calculated using the absolute value of the input samples (i.e., they are always positive values). This is the mode shown in the figure.

3) *rms* (*root mean square*). In this mode, the input samples are first squared (consequently becoming all positive) before being averaged. The square root of this average is then calculated. Although this method is more expensive in terms of CPU usage, it provides the most accurate results. Nonetheless, the absolute mode is generally sufficient in many situations.

You can change the operational mode by sending the messages "bipolar," "absolute" or "rms" to the `average~` object, and additionally change the number of samples it uses to calculate the average by sending it an integer value. Note that this value cannot be greater than the numerical argument you provided.

So, just how many samples do we need to use in order to obtain adequate envelope following? Unfortunately, it is impossible to define a good value that would work in all situations. Nonetheless, you should be aware that as the number of samples increases, any accents in the input sound will tend to be smoothed-out in the resulting envelope-followed signal. Conversely, as the number of samples decreases, the envelope will begin to more and more faithfully follow the amplitude peaks in the original signal, until the point where one sample is used for the "average," thereby just reproducing the instantaneous sample values of the input waveform.

We can now apply the envelope obtained from `average~` to a signal with a constant amplitude. Modify the patch as shown in figure 7.2.

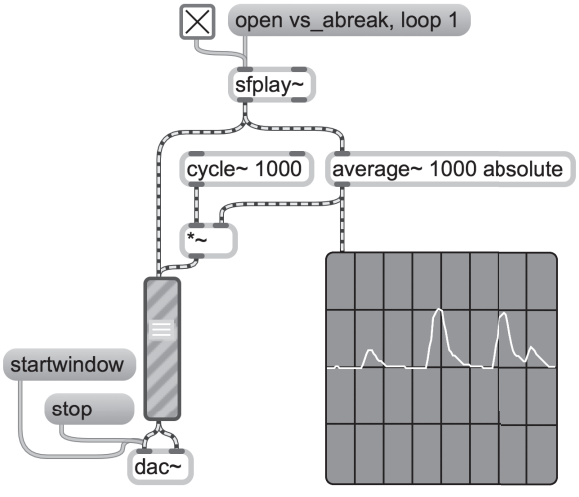

Fig. 7.2 Envelope applied to an oscillator

In this example, the envelope from a drum loop sound is applied to a sinusoidal oscillator at 1000 Hz. When you activate the patch you will probably notice that the envelope applied to the oscillator seems to be delayed with respect to the direct drum sound. This happens because each sample output by `average~`

results from averaging the preceding 1000 samples. A simple solution to correct this would be to delay the direct signal (the one going from `sfplay~` to the `gain~` object) until the two sounds seem to be in sync. Try this out by inserting a 200-300 sample delay.
Now, try modifying the number of samples that `average~` uses to calculate the mean by connecting a number box to the object. What happens when you use 1000 samples? What about with 10 samples? Or with just 1?

We can also use the envelope obtained from the envelope followed to control some sound processing parameters. Modify the patch as shown in figure 7.3.

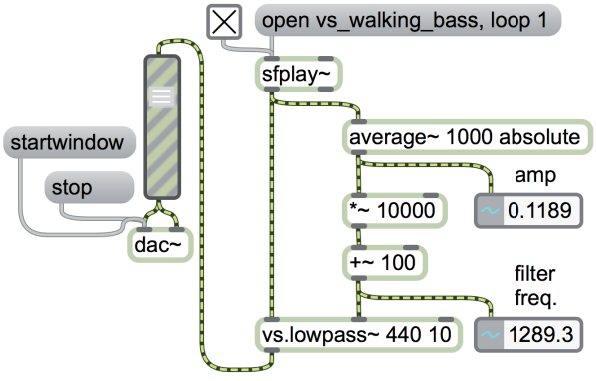

Fig. 7.3 Envelope controlling a filter

Here, the sound vs_walking_bass.wav has been loaded, and the envelope calculated by the average~ object is used to control the lowpass filter that is applied to this sound file. As you can see, the frequency of the filter follows the dynamic profile of the sound. The amplitude of the envelope is multiplied by 10000 and then the value 100 is added to it and the result is used as a cutoff frequency. Actually, in this example the signal generated by the envelope follower will rarely exceed 0.1, so the maximum cutoff frequency will be around 1100 Hz (10000 • 0.1 + 100). Try changing the values of the multiplier, addition, Q factor (second argument to the filter) and the filter type (for example, try using a highpass filter).

Let's now take a look at how we can control different parameters using the envelope follower; open the patch **07_01_envfollow.maxpat** (figure 7.4).

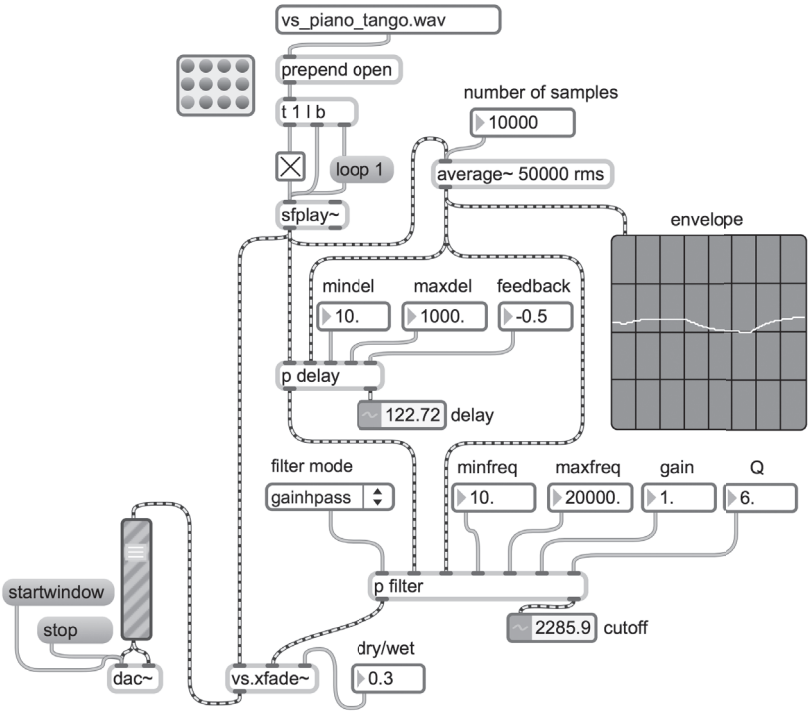

Fig. 7.4 The file **07_01_envfollow.maxpat**

In this patch, the envelope follower is used to control both the delay time and the cutoff frequency of a filter simultaneously. The signal output by `sfplay~` is sent to an `average~` object that creates the envelope (in rms mode) and sends it to two subpatches. The first subpatch, [`p` delay], takes the envelope in its second inlet and uses it to control the delay time of the sound sent to its left inlet. The delay time can vary between the given minimum and maximum values (via the two `flonum` objects labeled "mindel" and "maxdel"), and could optionally include feedback. The delayed signal is sent out the left outlet of the subpatch, while its right outlet outputs the actual delay time (displayed in the `number~` object). The contents of the subpatch are fairly simple (figure 7.5).

The signal outptut by `average~` ("env" inlet) is sent to a `scale~` object that modifies the 0-1 range of the envelope into the mindel-maxdel range set in the main patch.[1] The delay time is sent to the `tapin~/tapout~` object pair with feedback on the left side of the subpatch.

[1] Note that it is very unlikely that the envelope values output by `average~` will ever get close to the maximum value of 1, and therefore also unlikely that you will ever reach the maximum delay time. Therefore, this should be taken into account when setting the "maxdel" parameter.

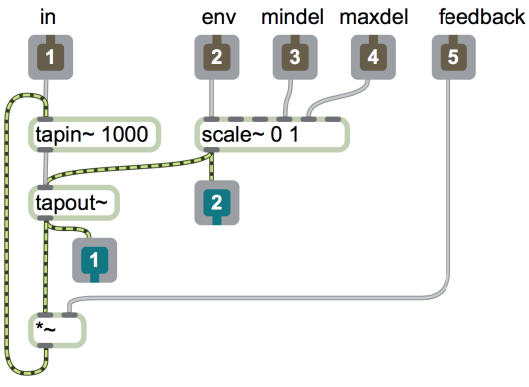

Fig. 7.5 The [p delay] subpatch

Next, the delayed signal in the main patch is sent to the [p filter] subpatch, which uses the envelope to modulate the cutoff frequency of a filter. The parameter here is also scaled between a minimum and maximum range ("minfreq" and "maxfreq"), and it is additionally possible to control the filtered signal's amplitude ("gain") and Q factor.

You can always change the filter type using the `umenu` object labeled "filter mode". There are three available filters: gainlpass, gainhpass and gainbpass – more specifically a lowpass filter, a highpass filter and a bandpass filter, all of which have optional gain control. Let's now take a look at the contents of this subpatch (figure 7.6).

(...)

other sections in this chapter:

Interlude D
ADVANCED PRESET MANAGEMENT, BPATCHER, VARIABLE ARGUMENTS, DATA AND SCORE MANAGEMENT

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6 and 7 (theory and practice) and Interlude C

OBJECTIVES

Abilities

- To know how to manage complex data storage systems
- To know how to control the partial or total visualization of abstractions and subpatches inside a patch
- To know how to use variable and local arguments in abstractions and subpatches
- To know how to manage data sets and create algorithms for controlling "scores"

CONTENTS

- Advanced systems for managing presets and interpolating between them
- Systems for visualizing abstractions and subpatches inside a patch
- Data management in Max

SUPPORTING MATERIALS

- List of Max objects - List of attributes, arguments, messages and commands for specific Max objects - Glossary

ID.1 ADVANCED PRESET MANAGEMENT

Up until now we have been using the `preset` object to control the parameters present in a patch. This allows us to quickly and easily store and recall the values contained in interface objects such as number boxes or multisliders. Figure ID.1 shows a brief résumé of this object's features.

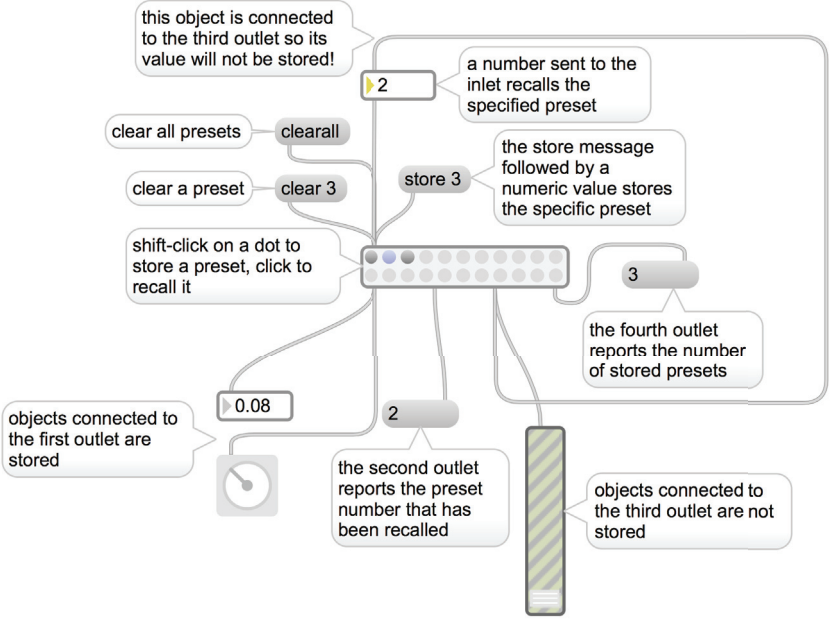

Fig. ID.1 The `preset` object

It is possible to include or exclude specific objects from the preset storage using the first and third outlets. Let's take a look at how this works.

- When no objects are connected to the `preset` object's first and third outlets, it will store the values of all the interface objects present in the patch

- When one or more objects are connected to the first outlet, only the values of those objects will be stored, and all others will be ignored

- When one or more objects are connected to the third outlet (but nothing connected to the first), the values of these objects will be ignored and the others will be stored (we have often used the third outlet to exclude storing the values of objects used to control audio output volume, such as `gain~` or `live.gain~`, because the settings for these are dependent on the listening setup being used).

- If there are objects connected to both the first and third outlets, these object will be stored or ignored, respectively, and any other objects in the patch (which are not connected to the `preset` object) will be ignored.

As we previously learned, in order to store a preset you just need to shift-click on one of the **preset** object's dots with the mouse. A simple click is used to recall one of the stored presets. You can also recall a preset by sending an integer value to the **preset** object's inlet – a value of 1 corresponds to the first dot, 2 to the second, and so on. To save a preset you can also send the "store" message to the inlet followed by a preset number. The "clear" message, followed by a number, can be used to delete a specific preset and the "clearall" message to delete all stored presets.

Whenever a preset is recalled, the corresponding number is sent out the second outlet, and when a preset is stored, the number is sent out the fourth outlet. Note that in the patch shown in the figure, the number box connected to the inlet has been excluded from preset memorization – can you figure out why?

Although simple and functional, **preset** is fairly limited where parameter management is concerned. For example, it cannot store the state of interface objects contained in subpatches, nor can it interpolate values between two presets. In the next section we will take a look at a more refined (though, needless to say, more complex) system for preset control and management.

THE PATTR OBJECT

The **pattr** object is a universal data container (that is, it can store numerical values, lists or symbols) which is capable of sharing its contents with Max's interface objects. Furthermore, the object can also be used to help us take our first steps toward a more advanced system for preset management. Let's look at some of the features shown in the patches presented in figure ID.2.

Even though these example patches are very simple, we still recommend that you recreate all six of them shown in the figure, because some features of the **pattr** object work quite differently from the standard Max object behaviour that you have learned so far.

In patch number 1, we can see that a **pattr** object accepts any kind of message (numbers, symbols or lists), and that these messages are immediately passed through to the object's left outlet. A bang can be used to re-send the last message received. It is also possible to "bind" an interface object to a **pattr** by connecting the second outlet of **pattr** to the interface object, as shown in patch number 2. Any value output by the interface object will also be stored in **pattr** and sent to its left outlet. And, as you can see in patch 3, a value sent to **pattr** will be forwarded to the "bound" interface object in addition to being sent out the left outlet.

The first argument to **pattr** defines its *scripting name*. Every **pattr** object is *required* to have its own scripting name, so if no argument is provided, a default scripting name will be assigned by Max (see patch number 4). The second argument to **pattr** specifies the scripting name of an interface object, allowing it

to be bound to the pattr without a connection. When you rebuild patch number 5, you will need to assign the scripting name "guitar" to the floating-point number box using the inspector. In order to make sure the object is properly bound to the `pattr`, you will need to first assign its scripting name and then create the [`pattr` blue guitar] object afterward. Note that in this case we now have two scripting names: one for the `pattr` object and one for the `flonum`. It is not possible to assign the same name to both the `pattr` and the `flonum` because each object's scripting name must be unique[1].

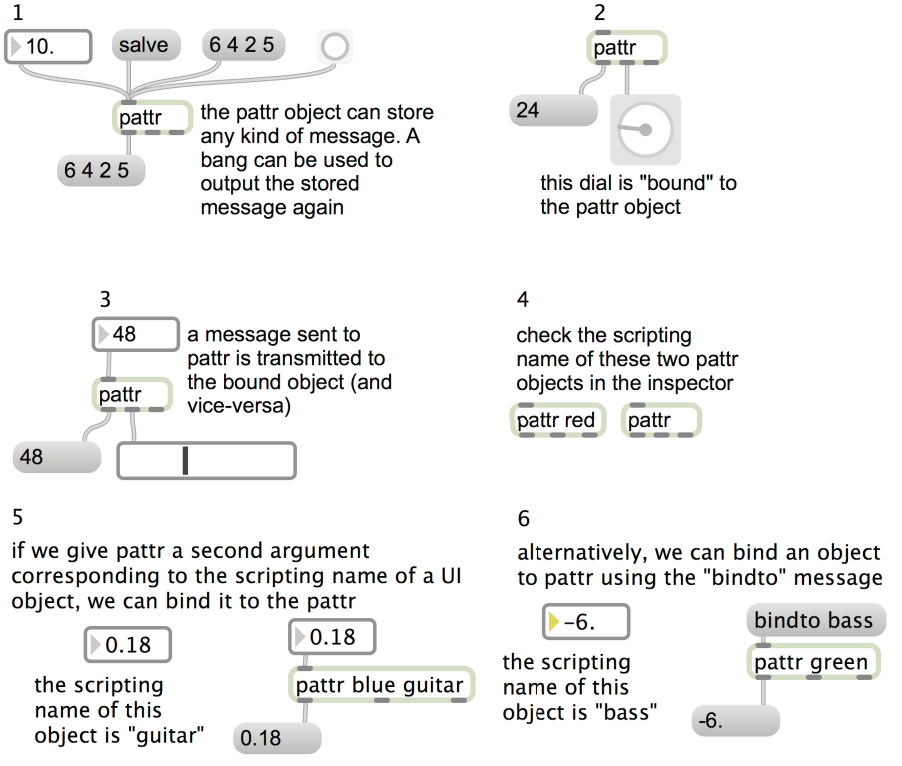

Fig. ID.2 The `pattr` object

We can also bind an interface object to a `pattr` "on the fly" by sending `pattr` the "bindto" message followed by the scripting name of an interface object (shown in patch 6).

When a patch containing `pattr` is saved, the data contained in the object will be saved and recalled each time the patch is opened. The use of this feature can be set with the "autorestore" attribute, which is enabled by default. To

[1] We have already learned that some objects, such as `buffer~` and `groove~`, can share the same name. However, in these cases it is not the objects' scripting name that is shared, but rather a named memory space that contains the shared data.

test this out, set the value of the slider in patch 3 to halfway along the slider and then save and reload the patch: the slider's cursor will appear in the same position it was when the patch was saved and not at 0, as it would be for an "unbound" object.

THE PATTRSTORAGE OBJECT

Apart from the autorestore function, the `pattr` object doesn't yet seem to offer any particularly new features that we couldn't also make ourselves using a pair of `send` and `receive` objects and the `pvar` object. However, by using the `pattr` object together with the **`pattrstorage`** object we can do some new and interesting things. Open the patch **ID_01_pattrstorage.maxpat** (figure ID.3).

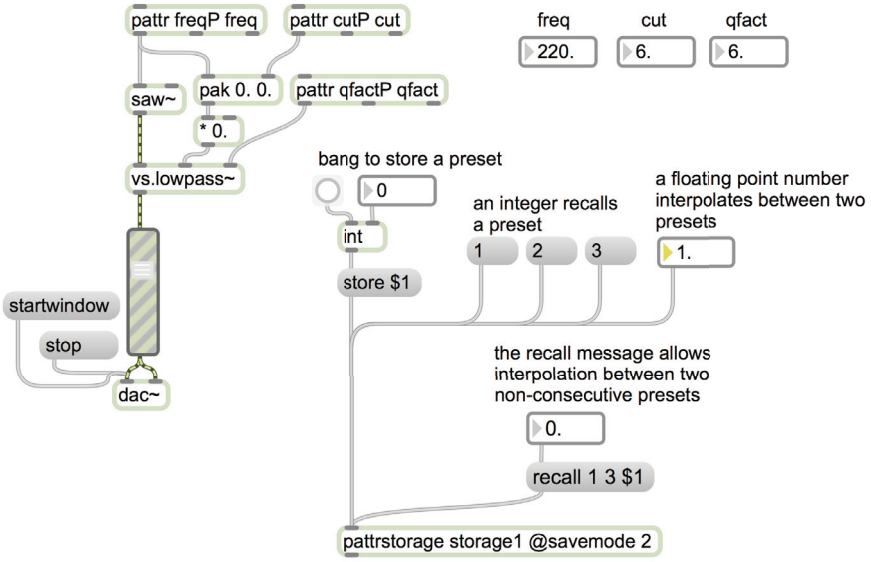

Fig. ID.3 The file **ID_01_pattrstorage.maxpat**

The `pattrstorage` object is the heart of this built-in preset management system, since it lets us save and recall the values contained in all `pattr` objects.

The three `flonum` objects at the top of the patch have been given the scripting names "freq", "cut", and "qfact", and are consequently bound to the three `pattr` objects visible to their left. Note that each of the three `pattr` objects has been given two arguments: the first is the scripting name of the `pattr` itself, and the second is the scripting name of one of the three number boxes to the right. Activate the DSP engine and click on the three message boxes with the values 1, 2 and 3 at the center of the patch – each of these can be used to select one the three presets saved in the `pattrstorage` object.

(...)

other sections in this chapter:

8T
THE ART OF ORGANIZING SOUND: MOTION PROCESSES

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6 and 7 (theory and practice), Interlude C and D

OBJECTIVES

Knowledge

- To know the various possible modalities of sound motion
- To know the inter-relationship between different types of motion
- To know the types of motion that can bring out a perceptual transition between one parameter and another
- To know some different possibilities for motion inside Timbre
- To know the limits and ambiguities concerning the categorization of motion

CONTENTS

- Simple, complex and compound motion
- Motion Sequences
- Types of motion within a timbre

ACTIVITIES

- Sound Examples and Interactive Examples

TESTING

- Listening and analysis test with short answers (maximum 30 words)

SUPPORTING MATERIALS

- Fundamental Concepts - Glossary

The domain where music takes form is a heavily spatialized temporality. But to be clear: music does not thereby become visual; it exists and persists wholly within the ear. Nonetheless, its organization and logic-based connections come into our minds via the visual world – the world of space. [...] We thus find ourselves confronting the very breath of matter itself.
(Salvatore Sciarrino, 1998)

Our principal metaphor for musical composition must change from one of architecture to one of chemistry. [...] This shift in emphasis is as radical as possible – from a finite set of carefully chosen archetypal properties governed by traditional "architectural" principles, to a continuum of unique sound events and the possibility to stretch, mould and transform this continuum in any way we choose, to build new worlds of musical connectedness.
(Trevor Wishart, 2004)

– to compose no longer with notes but with sounds;
– to no longer compose only sounds but the differences that distinguish them from one another;
– to act on these differences – namely, to control the evolution (or the non-evolution) of the sound and the speed of this evolution;
(Gerard Grisey, 2001)

8.1 WHAT ARE MOTION PROCESSES?

INTRODUCTION

Up until now we have been creating single sounds or even sequences of sounds without any particular inclination to explore how they can be artfully organized into larger forms. From this point onward we will begin to work on the sounds' various possible articulations, as well as their motion. This chapter (alongside the corresponding practice chapter) is designed to further enhance the theoretical knowledge and practical skills you have learned thus far (both in terms of sound analysis and listening as well as in terms of programming), and to bring out your creativity and capacity to build motion processes. As already mentioned in the introduction to this volume, we will limit ourselves to sound articulations under one minute in duration, exploring an intermediate area between the micro-form inherent in individual sounds, and the macro-form of an entire sound composition.[1] In this chapter we will therefore discuss the motion processes of sound independently of a more extensive formal scope and context, namely that of an actual sound composition. This does not necessarily mean that your creative process needs to follow this sequence (from the creation of individual sounds, to motion processes followed by the construction

[1] The term "composition" is being used here to mean an activity and experience that goes beyond just musical composition in the literal sense, but also encompasses works of sound art, sound design, audio-visual works, soundtracks, soundscapes, etc.

of an overall compositional form), but nonetheless this progressive order works well from a purely pedagogical perspective.

In practice, every sound artist follows his or her own way of building a sonic form in time and/or space, whether that means starting from the project as a whole, working on the detailed specification of its sound components at a later stage, working at all structural levels simultaneously or actually entrusting the computer to make some of the formal choices.

For the creation of interactive installations (even those that exist online), the compositional forms used may not take on specific predetermined shapes, so the purpose of the composer/artist in such a context is to design an interactive environment where the form is constantly recalculated, based on the choices of users inside that environment.

Finally, a sound designer who wants to create sounds destined for moving image, interactive games, and the like, will have to make formal decisions that relate to environments created by others, or developed by third parties.

All of the topics covered in this chapter should be beneficial for any type of creative sound work regardless of its eventual intended purpose. When composing, constructing and processing timbres right down to the minutest detail, it is actually very important to know how to create and manage the various structural levels (from the micro to the macro) relative to the goals of the creative project at hand. Therefore it is also important to know how to listen to and evaluate the results, by "zooming" in and out from one level to another. Horacio Vaggione speaks of this as "an action/perception feedback loop".[2] "As a painter who works directly on a canvas must step back some distance to perceive the result of his or her action, validating it in a variety of spatial perspectives, so must the composer dealing with different time scales.[3] This being so, a new category must be added to the action/perception feedback loop, a kind of 'shifting hearing' allowing the results of operations to be checked at many different time scales. Some of these time scales are not audible directly and need to be validated perceptually by their effects over other (higher) time scales."[4]

One such shift in hearing is going from listening to individual sounds as independent units, to listening to the motion inherent in the sounds themselves. By listening in this way, we are able to notice the effects of sounds by "observing"

---

2 *"The meaning of any compositional technique, or any chunk of musical knowledge, arises from its function in support of a specific musical action, which in turn has a strong bearing on the question of how this action is perceived. Action and perception lie at the heart of musical processes, as these musical processes are created by successive operations of concretization having as a tuning tool – as a principle of reality – an action/perception feedback loop."* (Vaggione, 2001, p.61)

3 We could also extend this idea further by adding: "and in space."

4 *(Vaggione, 2001, p.60)*

them from a slightly higher structural level, but not so high that we "look at" them in the context of an entire creative work. We will confront some higher formal and temporal levels in the next volume, and begin to define ideas and/or projects in terms of their form. Nonetheless, both the theory and practice sections of this chapter are designed to help you develop a "mobile" sense of listening and the ability to set an action/perception feedback loop into motion within the context of your own knowledge, skills and creativity.

MOTION PROCESSES

Motion Processes are evolutions within sound that happen via variations in spectrum or space or which can occur through interactions between these and other parameters.[5]

There can be multiple types of motion (even contrasting or ambiguous ones) within a single sound sequence, so it is very important for anyone working with sound to learn how to control and create complex motion sequences that evolve from an organic combination of simple types of motion, and lead to the creation of rich sonic environments. In this chapter (and in the corresponding practice chapter) we will offer the reader some different technical approaches that may be taken in order to create the kinds of sounds exemplified by the types of motion being described.

[5] This term was adapted from an article by Denis Smalley (Smalley 1986), and is fundamental to understanding the compositional process in electronic (and other kinds of) music. For Smalley, motion processes are processes of evolution within sound that can happen through variations in spectrum or space, or by the interaction of these and other parameters. The point of view from which Smalley observes motion in sound is not a technological one – he does not even touch upon electroacoustic processes and techniques used for the production of music. His emphasis focuses on concentrated listening and the aspects of sound that emerge from such listening. The terminology that Smalley uses to describe the different types of sonic motion can also be useful to help us better understand and listen to pieces of electroacoustic music with greater awareness, even though they may not have a printed score. (Naturally this also applies to composing such music, for those who have the skill and desire to do so.) However, there is one important difference between Smalley's notion and ours: from our perspective, motion processes are not merely considered from a listening point of view, but also from a technical-creative one. The upshot of this is that the idea of something like unidirectional motion, from our standpoint, does not simply represent an audibly recognizable increase or decrease in frequency, for example, but rather a general increase from lower to higher values (or vice-versa) in different perceptually relevant domains, such as frequency, duration, amplitude, etc. In this way, unidirectional motion can be understood in the broadest possible terms: transversal movement in relation to its parameters. The task of separating all the parameters in a sound, although extremely useful when listening to a piece analytically, must give way to expectations of complexity and interrelationships of sounds within the field of sound creation; these ideas will be developed further in the chapter dedicated to sound creation in the next volume. The technical and compositional purposes of this book lead us to work within an essentially different process than that of Smalley's concept of "spectromorphology". Therefore, our purpose in listening to sounds is not to arrive at a definition of their characteristics and possible evolution in time, but rather to begin with definitions and listening in tandem with one another, in order to work toward the creation of sonic motion.

Nonetheless, you should be aware that you may find yourself confronted with examples whose motion may have a very different sense when defined from a technical perspective than it would when observed from a perceptual standpoint. This should not actually pose any problems, per se, but rather offer an opportunity to contemplate how one is constantly in contact with contradictions and ambiguities while in the act of organizing sound[6], and that getting to know and accepting these elements will provide richness and multiple levels of meaning to your work.

However, our aim here is not to build a regular and coherent theory, but to take an open, interactive educational approach (even sometimes a problematic one) where the reader can genuinely and practically begin to take his/her first steps and eventually venture out to make his/her own choices and discoveries.

Naturally, it is important to understand that, from a creative perspective, our goals, methods and ways of looking at the parameters and properties of sound are also sometimes very different from those when dealing with sound from a physical point of view.

For example, during the process of creating a musical work we are able to pass from the time domain to the frequency domain with ease. Furthermore, where perception is concerned (and hence also where music is concerned), we tend to mix parameters of highly differing natures, passing through different notions of time and space and exploring the boundaries between one property of sound and another. Therefore, in order to be able to talk about the art of organizing sound, we need to mix different approaches in a way that would be irreconcilable for a scientist (because the necessary leaps in logic could neither be formulated nor verified by any kind of scientific method) so we will need to review some of the concepts that we have learned in previous chapters from a broader perspective (or at least less rigorously from a technical stance), but one which is closer to the approach which needs to be taken by composers, sound artists and sound designers. In particular, we will consider frequency, amplitude and duration to be basic parameters whose organization in time gives way to other, more complex sound properties such as timbre, rhythm and spatialization. Often, we will find ourselves moving from one to the other smoothly without a break, since this chapter has been designed to provide an experiential path to be followed. Even the categories that will be introduced here should only serve as points of reference to help you better understand their limits and boundaries.

6 This is the broadest term we can use to describe creating and giving structure to sound using new technologies. It was coined by Leigh Landy in his influential book (Landy 2007), and also relates to the concept of "organized sound" proposed by Edgar Varèse.

MOTION CATEGORIES

We have chosen to use just a few categories, organized in increasing order of complexity:

Simple Motion
- motion resulting from changes in the values of just one parameter of one sound

Complex Motion (which contains simple motions)
- motion resulting from changes in the values of several of a sound's parameters

Compound Motion (which could also contain simple and/or complex motions)
- motion characterized by changes in the values of the parameters of several sounds

Motion Sequences (which could contain any type of motion)
- a series of motions which relate to one another

These categories are not absolute and unconditional – they are simply based on conventions which we have adopted, and many "gray-zones" exist in-between them. Let's take a look at a couple of examples of these kinds of ambiguous areas.

1) **Ambiguity arising from the distinction between motion in one sound (simple and complex motion)** and motion in several sounds (compound motion). What, precisely, is a single sound? This entirely depends on our perception of that sound – some sounds (the sound of the sea, for example) can either be considered a single sonic element or a composite sound made up of several simultaneous autonomous units (such as the sound of waves). The intentions of the person working with the sound should also be taken into consideration: a sound with many components can move as a single object, or the parameters controlling its various components can be moved separately.

A sound recorded to an audio file or originating from an algorithm could also contain compound motion but if we deal with the sound as a single unit, such as changing its overall intensity, this motion could be considered to be simple because it acts on just one of the sound's parameters. On the other hand, if we separately affect the various individual sounds that make up the composite sound, we would be creating compound motion.[7]

2) **Ambiguity between simple and complex motion within a system that it is in and of itself complex.** Let's take the use of filters as an example. If we add a filter with an upward-moving cutoff frequency to a sound, the movement of this filter is an example of simple motion, since just one parameter is being changed. Obviously, when the user creates this kind of simple motion, what goes on *inside* the filter is something much more complex.

[7] The concept of the "objet sonore" (sound object), theorized by Pierre Schaeffer, could also give rise to this same ambiguity, so we prefer not to introduce this concept at the present time.

The same thing happens in a stereo spatialization system: the motion from one extreme to the other, using a panning parameter from 0 to 1 is simple in that we use just one simple line segment to move the sound from one side to the other. However, it is simple *only* if we do not take into consideration what is happening inside the panner, which is actually a double control over the intensities of the left and right channels, inversely proportional to one another. For practical and logical reasons these motions are included in the first category.

Vaggione observes that "some types of representation that are valid on one level cannot always retain their pertinence when transposed to another level. Thus, multi-level operations do not exclude fractures, distortions, and mismatches between the levels. To face these mismatches, a multi-syntactical strategy is 'composed.'"[8]

For this reason, paradoxically, we have divided the types of motion into categories in order to be able to highlight the gray-zones – the areas of overlap – that we run into every step of the way when learning and practicing electronic music and sound design. The notions that follow should therefore not be taken as law, nor are they intended to be exhaustive (something virtually impossible in the field of sound research). They are simply intended to provide a journey into sound and its different kinds of motion in order to help you acquire a deeper awareness of them – this is a prerequisite for those who want to venture into the art of sound organization.

Now that we have briefly described the most thorny aspects and limitations relating to the formalization of the structural levels, we can finally begin to describe the various motion processes themselves.

(...)

[8] Vaggione, 2001.

other sections in this chapter:

8.4 EXPLORING MOTION WITHIN TIMBRE
- From intervallic pitch to noise
- The importance of attack on the perception of timbre
- Timbral motion using resonant filters
- Increasing spectral complexity
- Progressive saturation in spectral complexity with an ascending and descending glissando
- Increasing the amount of randomization in a waveform
- Distribution curves for spectral components
- Distribution of components in white and pink noise
- From noise to note

8.5 COMPOUND MOTION
- Compound centrifugal or centripetal frequential and spatial motion
- Synchronized centrifugal/centripetal oscillatory motion
- Opposing motion: continuous increase and decrease in pitch using shepard tones
- Oscillatory motion of a bandpass filter applied to centrifugal motion
- "Skidding" centrifugal or centripetal motion: random walk applied to frequency
- "Skidding" compound motion: irregular but synchronized rhythm, pitch and spatialization
- Centrifugal and centripetal compound motion: space and pitch
- Compound motion based on accumulation and rarefaction

8.6 ALGORITHMIC CONTROL OF MOTION
- From pitch to timbre
- Algorithmic control of rhythm
- Algorithmic control of spatialization

8.7 INTRODUCTION TO MOTION SEQUENCES
- Contrasts
  - Superimposed contrasts
  - Alternating dynamic contrasts
  - Contrasts leading to fusion: "little bang"
- Gestures and textures
- Regularity in rhythmic sequences and their function
- Polyrhythms and rhythmic irregularity
- The relationship between figure and ground
- Processi di moltiplicazione

- SOUND EXAMPLES AND INTERACTIVE EXAMPLES
- LISTENING AND ANALYSIS TEST WITH SHORT ANSWERS (MAXIMUM 30 WORDS)
- FUNDAMENTAL CONCEPTS - GLOSSARY

8P
THE ART OF ORGANIZING SOUND: MOTION PROCESSES

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6 and 7 (theory and practice), Interlude C and D, and Chapter 8T

OBJECTIVES

Abilities

- To know how to envision and create different kinds of motion within sounds and directions of motion within their timbre.
- To know how to build algorithms to interrelate different types of motion and motion sequences.
- To know how to apply appropriate parameter values in order to cause the listener to perceive a transition from one parameter to another within a processed sound.

CONTENTS

- Practicing simple, complex and compound motion processes
- Practicing motion processes within the timbre of a sound
- Practicing building motion sequences

ACTIVITIES

- Building and Modifying Algorithms
- Reverse Engineering Activities

SUPPORTING MATERIALS

- List of Max Objects

8.1 MOTION PROCESSES

Since motion processes have already been described in detail in chapter 8T, this practice chapter will consist mostly of a series of activities alongside some analysis and "reverse engineering" exercises that may be realized in parallel to reading the theory chapter. Note that for our purposes, the aim of reverse engineering sounds is not to faithfully reconstruct them, but rather to be able to redesign the motion that has been applied to the sound.
Naturally, when creating the small electronic music and sound design études proposed throughout this chapter, you may rely on any of the techniques you have learned in the previous chapters. Although this chapter is short, it will nonetheless require a fair amount of work. Let's start off with the first of our various motion typologies.

8.2 SIMPLE MOTION

UNIDIRECTIONAL MOTION – FREQUENCY

Applying simple motion to frequency essentially amounts to creating a glissando. Naturally, the main objects that could be used to achieve this are `line~` and `curve~`. Since we are already well acquainted with a wide variety of techniques making use of these objects, we will not dwell on them here for very long.

Creating simple stepwise motion (in other words non-continuous) can be done in several different ways. Rebuild the patch shown in figure 8.1.

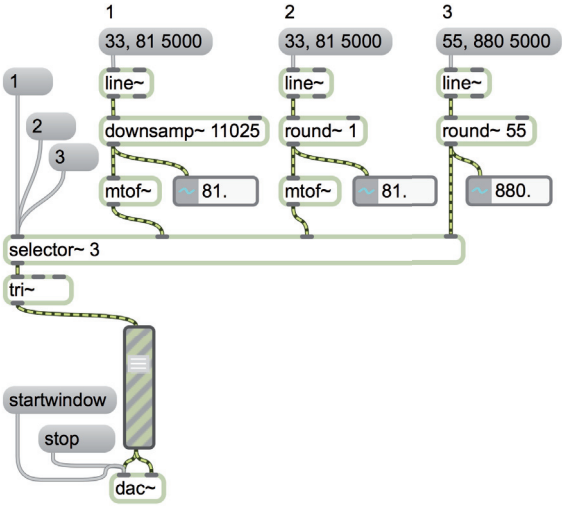

Fig. 8.1 Three different ways to quantize a glissando

All three methods shown in the figure express simple unidirectional motion from A0 to A4. The first "quantized glissando" was created with the `downsamp~` object, which downsamples the input signal using a sample and hold technique. Its argument indicates the sampling period (expressed in samples). Therefore,

the object in the figure will sample (and hold) one value every 11025 samples – this amounts to a quarter of a second, presuming we are using a sampling rate of 44100 Hz. The glissando is done with MIDI values, which are converted to frequency values after the sample and hold.

In the second example, the [**round~** 1] object is used to round the signal values to the nearest multiple of a given interval value. The argument indicates the interval that will be used – if it is 1 the signal value will be rounded to the closest whole number value, if it is 2 it will round it to the closest multiple of 2, 3 to the closest multiple of 3, and so on. It is also possible to round values to multiples of floating-point numbers: using an argument of 0.5 would produce output values that are multiples of 0.5. In the example shown in the figure, this quantization produces a series of ascending semitones.

In the third example, a glissando is created using frequency values directly, instead of via MIDI note numbers. Here, the values are rounded to multiples of 55, thereby creating a glissando across the harmonic series based on the note A (55 Hz).

ACTIVITIES

- Try to discover at least three other techniques for quantizing a glissando.
- Reverse-engineer sound example 8A.1, using a sampled sound of your choice.

UNIDIRECTIONAL MOTION – DURATION AND RHYTHM

To create this type of motion, we can use the technique illustrated in the patch 05_13_blocks_tech_accel.maxpat (see section 5.4P). The **vs.randmetro** object (see section IC.3) could additionally be used in order to introduce a little bit of irregularity into the rhythm.

ACTIVITIES

- Reverse-engineer sound example 8A.2, using a sampled sound of your choice.
- Create a unidirectional motion in rhythm that makes a transition from a regular beat to an irregular one.
- Create a rhythmic motion where the sound progressively shortens even though the generation of the rhythm remains fixed.
- Create a series of accelerandi and rallentandi using multiple delays (you can take inspiration from the patch 06_02_multitap2.maxpat) using at least 32 delay taps.

(...)

other sections in this chapter:

9T
MIDI

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6, 7 and 8 (theory and practice), Interlude C and D

OBJECTIVES

Knowledge

- To know the MIDI protocol
- To know the structure of and be able to use channel and system messages
- To know the basic uses of MIDI controllers

CONTENTS

- The MIDI Protocol: Connections and Messages
- Transmitter and Receiver modules: the flow of MIDI data.
- The Structure and Use of Channel Voice Messages and Channel Mode Messages
- The Structure and Use of System Real Time Messages
- MIDI Controllers: from Quasi-Instrumental Interfaces to Control Surfaces
- Advanced MIDI Controllers: from Using a MIDI Data Glove to Gesture Mapping

TESTING

- Test with short answers (maximum 30 words)

SUSSIDI DIDATTICI

- Fundamental Concepts - Glossary

9.1 THE MIDI STANDARD

The exchange of information between electronic musical instruments, controller systems and computers often makes use of the *MIDI protocol*, a standard created in the early 1980s, and still widely in use today. The term MIDI is an acronym for Musical Instrument Digital Interface. The MIDI protocol is used by a large number of applications, often very different from one another. In this chapter we will cover only those parts of MIDI which are essential for our purposes, alongside some important basic information.

There are two types of MIDI devices:
- *controllers*: these are devices that output MIDI messages
- *sound modules*: these are devices that use the MIDI messages they receive to generate or modify sounds.

Furthermore, sound modules themselves can be divided into two categories:
- instruments, such as synthesizers and samplers, which output sound
- sound processors, such as delays, reverberators and other effects, which are used to modify external sound sources.

Sometimes, one device, such as a computer, can behave as either transmitter or receiver, depending on how it is being used. Note that a computer by itself cannot communicate with external devices using MIDI unless it has MIDI interface connected to it (or a built-in MIDI Interface Card). Today, the majority of MIDI instruments being manufactured have a built-in digital connection (such as USB) that can connect directly to a computer, allowing it to receive and transmit MIDI data to the computer without the need for a dedicated MIDI interface. Figure 9.1 shows a generic back panel of a MIDI interface with two MIDI Out ports, two MIDI In ports and a USB port which connects to the computer.

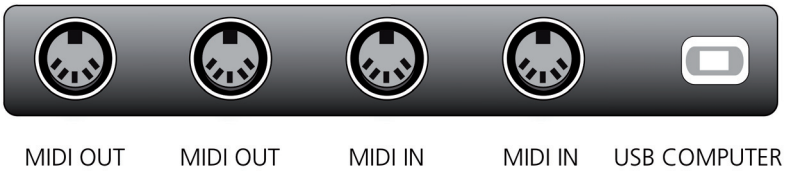

Fig. 9.1 Back panel of a generic MIDI interface

Today MIDI is used in a wide variety of applications, many of which cross over into the realm of professional audio. Furthermore, MIDI is a continually evolving protocol, and is used by computers, the internet, mobile devices (such as mobile phones), complex sound control systems, light boards, multimedia systems, and countless other devices. There exist several other communication protocols which are even faster and more flexible than MIDI (such as OSC, for example), although they are not very commonly found on electronic musical instruments such as keyboards, MIDI guitars, etc. As such, MIDI remains universally utilized due to both its ease of use and strong presence in the realm of commercial musical instrument production.

9.2 MIDI MESSAGES

Before delving into the details of MIDI messages, we feel it is necessary to first point out that MIDI messages themselves do not contain any sound, but instead define the way in which a particular sound, residing elsewhere in memory, will be played. In other words, they simply express what note will be played, when that note will begin and end, what dynamic it will have, and so forth. Therefore, it is very important to differentiate a sound file (containing a sound's waveform, along with its duration, envelope, timbre, frequency, etc.) from a MIDI message, which simply provides information relating to the way a given sound will be played, and not its waveform. The waveform itself is usually stored somewhere in memory (or generated by an algorithm which is stored in memory) in a separate location from the MIDI message. In other words, a keyboard sends MIDI data, and this data is used to trigger the sound-generating module receiving it. This module could have waveforms stored in memory, or could use some other type of sound-generating algorithm, and would produce sound only in reaction to MIDI messages coming from the keyboard. In order to be completely clear: the keyboard only sends instructions to the sound module via MIDI, and the module receiving these instructions outputs the sound via its own audio interface. One reason that there is sometimes confusion about this is that there are two types of MIDI keyboards:

1) a mute keyboard, or *Master Keyboard*, which exclusively performs the function of a transmitter – in other words, a basic *MIDI Keyboard Controller*. This kind of keyboard generates only MIDI data, which can be sent to external MIDI receivers using via its MIDI Out port.

2) a keyboard that works as a sampler or synthesizer – that is, a keyboard which also contains its own internal sound generation module in addition to generating MIDI data. This type of keyboard can act as both transmitter (via MIDI out) and receiver (via MIDI in), just like a computer, but it has an additional internal connection that sends the MIDI messages it transmits to an internal sound generation module. When we play a key on this kind of keyboard, we create MIDI messages that can be sent to either an external module (via MIDI out) or to the internal module that outputs sound from the keyboard's audio outputs (or to both places). Nonetheless, even in this situation the MIDI data itself (either that output by the keyboard or sent to the internal sound module) *does not contain any sound information*: the sound is always generated by an external or built-in sound module.

As we will learn in the next section, MIDI controllers come in all shapes and sizes. In addition to keyboards, there are also MIDI guitars, MIDI string and wind instruments, MIDI percussion and pedals (even Key Pedalboards, usually one or two octaves, and generally used for low-pitched sounds like organ pedals), just to name a few. These are all controllers whose shape and use is similar to that of the acoustic instruments they emulate, but which, as we have said before, are only used to send MIDI messages to a sound module – either external or built-in.

This module could contain sounds that match the interface (such as the sound of a guitar played by a MIDI guitar controller) or not (for example the sound of a flute or automobile horn being activated by a MIDI clarinet).

Now that we have hopefully cleared-up the general concept of MIDI and what types of information are *not* included in MIDI messages, let's move on to learn about the actual information that *can* be contained in them. The most basic use of MIDI is a sound module responding to a message resulting from a key being played (Note On) and the same key being released (Note Off) on a mute keyboard, along with pressure associated with playing that key. This pressure is called Key Velocity, because it is actually measured as the speed with which the key was played (not actually its pressure). It corresponds to the dynamic that note would have if it were played on a piano.

Because everything in MIDI is expressed in terms of numbers, we need to know the convention that has been adopted. This convention (as we learned in section 1.4T) means that each note (or better yet, each key) corresponds to a particular number (Key Number, in MIDI terminology): middle C is equal to 60, C# equal to 61, D to 62, D# to 63, etc.

The possible note values range from 0 to 127, even if the MIDI instruments themselves are not capable of producing every available note. A MIDI keyboard with the same range as the piano would only be able to produce note values between 21 and 108.[1]

Analogously, each dynamic corresponds to a number ranging between 0 and 127. The variation in dynamics within the sound generating module is generally on a logarithmic scale, in order to better match the ear's perception of dynamics and thus make the step between any two values have the same audible change in dynamics.

The MIDI protocol, to a limited extent, also allows the performance of notes outside the tempered tuning system[2], either by using instruments that are set up to handle tunings other than equal temperament (this is heavily dependent on the instrument being used – not all instruments are designed to handle alternate tunings), or by sending MIDI information that determines a variation in pitch for each note (with respect to its "normal" pitch), before the note is played.

[1] If you happen to have a controller with fewer octaves than the sounds available in the sound module you are using, you can always change the position of the sound by associating it with the keys an octave higher or lower, so that it falls within the instrument's actual playable range.
[2] See section 1.4T in the first volume for information about tuning systems.

This information is called *Pitch Bend*, and has the same effect as the physical *Pitch Bend Wheel* (or *Pitch Bender*) controller found on practically every MIDI instrument. This controller is generally some kind of wheel or slider that lets you momentarily alter the intonation of the notes being played.

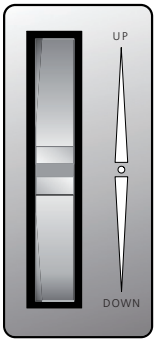

Fig. 9.2 A generic *Pitch Bend Wheel*

Furthermore, it is possible to select between different types of sounds (timbres or programs) stored in the instrument itself. There is a dedicated MIDI message, called *Program Change*, that lets you select a sound from among the different ones available in the instrument.

Let's now take an in-depth look at the two types of MIDI messages:

- Channel Messages
- System Messages

CHANNEL MESSAGES

If one MIDI transmitter (a computer, for example) is connected to several instruments, it is important to be able to establish a way of sending MIDI messages to just one particular instrument, otherwise all the instruments would simultaneously play the same "part." For this reason, each MIDI message is associated with a specific *channel number*, so each instrument can be set to handle only the messages that are relevant to it – in other words, only those messages which have the same channel number that the user has defined for the instrument itself. Channel Messages, are therefore addressed to a specific channel, and can be received and handled only by instruments that are set to receive on that particular channel. There are two types of Channel Messages: Channel Voice Messages, which deal with performance modes and execution times, and Channel Mode Messages which indicate the behavior of the receiving device. Let's begin by taking a look at the main Channel Voice Messages: Note On, After Touch, Polyphonic Key Pressure, Note Off, Program Change, Pitch Bend and Control Change.

NOTE ON

The Note On message corresponds to playing a given key on a MIDI keyboard or otherwise activating a note using one of the previously mentioned MIDI controllers. In addition to the note number (or Key Number) and Key Velocity,

this message also contains the MIDI channel number. In this instance the *Key Velocity* could be more aptly called *Attack Velocity* since it relates to the speed with which the key was played.

Just to give an example, a MIDI message that plays middle C is composed of three pieces of information: 144 60 120. The first number (called the Status Byte) is 144 and this value represents a "Note On" message, indicating a key has been pressed. The second number (called the First Data Byte) is 60, and this represents the note, or Key Number (middle C in this case). The third number (called the Second Data Byte[3]) is 120 and corresponds to the Attack Velocity (the maximum value for which is 127). So, what about the MIDI channel? This information is included in the first number. A Note On can actually be activated using any number between 144 and 159 as the Status Byte. If the first number is 144 (as it is here) the note is sent on MIDI channel 1, if the number were 145 it would be sent on midi channel 2, and so on, until number 159, which indicates transmission on channel 16.

NOTE ON MESSAGE			
	Status Byte	**1st Status Byte**	**2nd Status Byte**
Message Sent	Note ON + MIDI Channel	Note Number	Attack Velocity
Range of Values Transmitted	144 – 159	0 – 127	0 – 127
Message Contents	144 = Note ON MIDI ch. 1 145 = Note ON MIDI ch. 2 etc..... .....up until 159 = Note ON MIDI ch. 16	0 = C, first octave [4] 1 = C#, first octave etc..... .....up until 127= G, last octave	0 = no attack 1 = minimum amplitude etc..... .....up until 127= maximum amplitude

(...)

[3] You should be aware that sometimes the *Status Byte* is referred to as the 1st byte, the first *Data Byte* as the 2nd byte and the second *Data Byte* as the 3rd byte. Regardless of the terminology used, the content of the three bytes is the same, whether you split them into two groups (Status Byte and Data Bytes), or if the bytes are simply numbered according to the order in which they are sent and received.

[4] According to the MIDI standard, there are different ways of numbering octaves. Here, we are defining the lowest octave that can be represented by MIDI as octave -2, and the highest octave as octave 8. Taking into consideration that the highest octave only goes as high as G because of the 128-note limitation inherent in MIDI, there are a total of 10 and a half octaves from -2 to 8. With this definition, middle C on a piano keyboard falls in the third octave, and can therefore be referred to as C3. The standard 440 Hz A is thus called A3.

other sections in this chapter:

Channel pressure (or channel after touch)
Polyphonic key pressure (or polyphonic after touch)
Note off
Program change
Pitch bend
Control change
System messages

9.3 MIDI CONTROLLERS
Simple controllers
Control surfaces
Midi sensors, midi data glove and motion tracking

- Test with short answers (maximum 30 words)
- Fundamental concepts
- Glossary

9P
MIDI AND REAL-TIME CONTROL

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6, 7 and 8 (theory and practice), Interlude C and D and chapter 9T

OBJECTIVES

Abilities

- To be able to use Max to manage MIDI data flow within a system of interconnected virtual devices.
- To be able to use Max to manage MIDI data flow (including polyphonic data) between MIDI hardware devices and software.

CONTENTS

- MIDI objects in Max and how they are used to manage messages
- Advanced MIDI polyphony management in between Max and external MIDI hardware.

SUPPORTING MATERIALS

- List of Max objects - List of attributes for specific Max objects

9.1 MIDI AND MAX

Up to now we have only lightly touched on the subject of MIDI in the Practice chapters of this series. In this chapter we will finally delve a little deeper into some aspects of the relationship between Max and MIDI in order to provide some essential information about their combined use.

In addition to note messages, which were already covered in section IB.1, other types of MIDI messages may also be sent and used to modify the parameters of the instrument (or effect) receiving them. Thus far, we have only been using MIDI to communicate with the built-in synthesizer of the computer's operating system, but it is nonetheless also possible to interface with any kind of MIDI device.

In Max's Options menu you can open up the MIDI Setup dialog. This window contains a list of all the MIDI devices – both physical and virtual – which are connected to the computer (see figure 9.1).

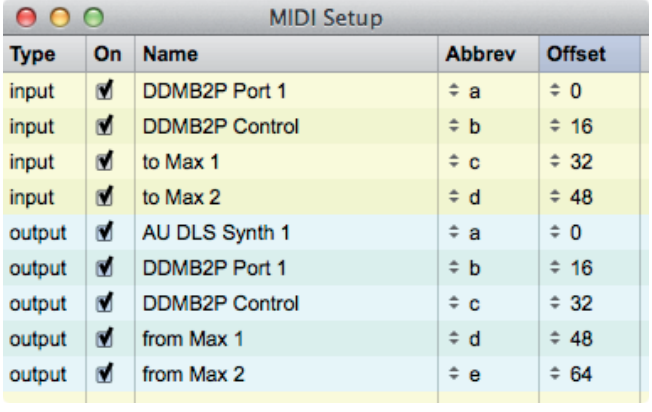

Fig. 9.1 The MIDI Setup window

The window includes both a list of input devices (those that can send MIDI messages to Max) as well as output devices (which can receive MIDI messages from Max). As you can see, some of the devices shown may be used for both input and output. This would be the case for a device such as a physical MIDI interface (not a virtual one) connected to the computer which has ports for both MIDI IN and MIDI OUT connections (like the DDMB2P device in the first line of the list shown in the figure).

Each device can usually handle 16 communication channels of MIDI data. However, one device could actually be able to control several instruments simultaneously, and some of these instruments could even be polytimbral – that is, capable of producing sounds with different timbres simultaneously. In this scenario, different MIDI channels would be used to address the different instruments, or different sections of a polytimbral instrument. For example, if we connect a digital piano, a polytimbral synthesizer and a reverb unit, we

could decide to send MIDI messages to the digital piano on channel 1, to the polytimbral syntheziser on channels 2, 3, 4 and 5 (to which we have assigned four different timbres), and to the reverb on channel 6. The `noteout` object, which has been used extensively throughout these first two volumes, has three inlets: the first for the MIDI note value, the second for the velocity and the third for the MIDI channel.

Each device has a name and we can additionally define an abbreviation and a channel offset.[1] using the MIDI Setup window. Each of the MIDI objects in Max can refer to a particular device using one of these parameters (generally given as arguments to the object).

In the MIDI Setup window, any device can be activated or deactivated by clicking on its checkbox (located in the "On" column), as well have its abbreviation or channel offset modified. Note the "AU DLS Synth 1" device which is the virtual synthesizer in Mac OSX (or the analogous device in Windows, which is called Microsoft DirectMusic DLS Synth): this is the synthesizer that we have been using up to now for all the MIDI examples.

Finally, some information about the devices called "to Max 1," "to Max 2," "from Max 1" and "from Max 2": these are virtual connections between Max and other programs running on the same computer, and are only available on Mac OSX. If you launch a program that uses MIDI (such as a sequencer) while Max is running, you will also see these virtual Max devices among the list of interfaces recognized by the program. This way you can send MIDI messages to Max using the "to Max 1" (or 2) port, as well as receive MIDI messages from Max using the "from Max 1" (or 2) port. In order to do the same thing on Windows, you need to use third-party software such as LoopBe1 (http://www.nerds.de) or LoopMIDI (http://www.tobias-erichsen.de).

[1] As we have already mentioned, there are 16 MIDI channels numbered from 1 to 16. However, because of the channel offset it is possible to use higher values, since the offset is added to the channel number. For example, you can see in the figure that device b has an offset of 16. This means that its channels (1-16) will be seen by Max as channel 17-32. Therefore when we send (or receive) a note message on channel 17, we are actually sending it to channel 1 on device b. The arguments to the MIDI objects let us specify the name of the device and channel, or only the channel (with a possible offset). In other words, with the settings shown in figure 9.1, the objects [`noteout` b 2] and [`noteout` 18] will both send messages to the same device and channel: the second channel on device b.

9.2 MIDI MESSAGE MANAGEMENT

In addition to the **noteout** object, used to send MIDI note messages, there is also a notein object that receives note messages from an external device, such as an actual (hardware) keyboard. Moreover, Max has a variety of other built-in objects capable of handling MIDI messages; some of these are shown in figure 9.2.

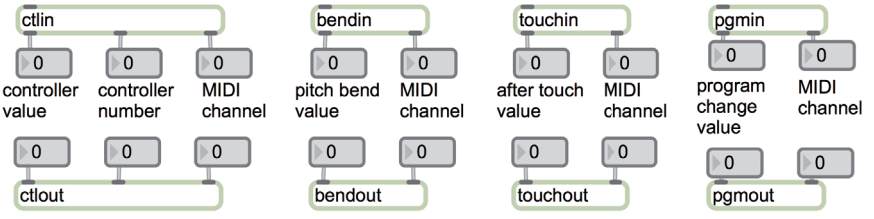

Fig. 9.2 Some other MIDI objects

The **ctlin** and **ctlout** objects receive and send control change messages. These messages are generally sent by controllers capable of transmitting a continuous stream of numerical values, such as the modulation wheel found on countless MIDI keyboards (and often used to control the a sound's vibrato), or pedals, faders, etc. (see chapter 9.3T). There are 128 controllers (numbered from 0 to 127) available on each MIDI channel, and each controller can be used to transmit (or receive) values between 0 and 127. Control change messages are generally used to modify some characteristic of an instrument's sound (such as adding vibrato to a sound, or changing the cutoff frequency of a filter) or some parameter of an effect (such as controlling the amount of distortion applied to the sound of an electric guitar being sent through a distortion effect).

The **bendin** and **bendout** objects receive and send pitch bend messages. These values (between 0 and 127) determine the modification of a note's pitch, used to simulate small glissandi on string instruments such as a guitar or violin, or on winds like clarinet or sax. The control mechanism used for this on MIDI keyboards is generally some kind of wheel.

The **touchin** and **touchout** objects receive and send after touch messages. These messages are output by MIDI keyboards based on the pressure exerted on the entire keyboard while keys are being held down. This is often used to modify some parameter of the sound, such as its volume or the cutoff frequency of a filter, and is therefore used similarly to a control change. Once again, the value range is between 0 and 127. The after touch message is global for the entire keyboard – the same message will be sent regardless of the key or keys that are being played. There is another MIDI message, polyphonic key pressure (not shown in the figure), that can send an independent pressure value for each key that is depressed, however this feature is rarely found on keyboards with the exception of perhaps some of the most expensive ones. That having been said, most keyboards are able to send global after touch messages.

The **pgmin** and **pgmout** objects receive and send program change messages, which are the equivalent of changing presets. Sending a program change message (yet again a value between 0 and 127) to a synthesizer (either real or virtual) tells it to change the timbre that it will be used to play subsequent notes. Note that in the Max environment, program change values are between 1 and 128, and are converted into values between 0 and 127 (the values are simply decremented by 1) before being sent to the device.

MIDI objects in Max can have the name of the device[2], the MIDI channel number or both as arguments (see footnote 1). For objects that receive MIDI messages, if the channel number is given as an argument, the associated outlet will disappear; see figure 9.3.

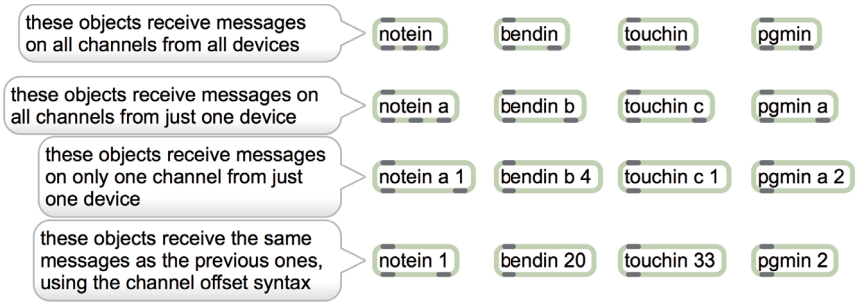

Fig. 9.3 Arguments to objects that receive MIDI messages

This figure provides an example for each possible combination of arguments. Note that the objects in the last two rows have one less outlet – they are missing the outlet corresponding to the MIDI channel, because this has been provided as an argument. Taking into consideration the device abbreviation and channel offset settings shown in the MIDI Setup window in figure 9.1, the objects in the last row will receive the messages on the same channel and from the same device as those in the row just above it.

The **ctlin** and **ctlout** objects deal with arguments in a slightly different way. There are 128 controllers per MIDI channel (numbered 0 to 127), and each of these can send values between 0 and 127. So, if these objects have just one argument, it is considered to be the controller number. This argument can be preceded by the device name, followed by the channel number, or both. If we want to specify a MIDI channel without defining a specific controller, we can use the value -1 as a controller number (see figure 9.4).

[2] The device name can be its abbreviation, which consists of a single letter, or the full name of the device as it appears in the "Name" column in the MIDI Setup window. When the full name is made up of several words separated by spaces, the name should be provided in quotes. Referring to the devices shown in figure 9.1, if we wanted to use the first device, we could either type the abbreviation a, or else use the full name "DDMB2P Port 1".

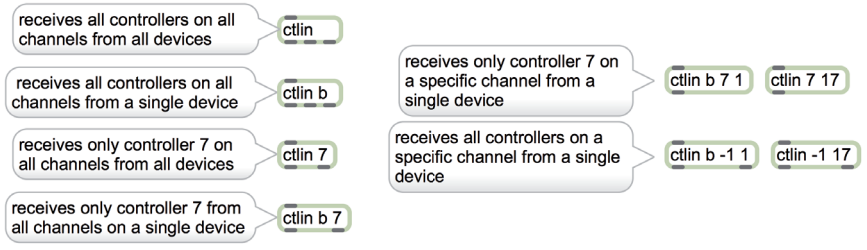

Fig. 9.4 Arguments for the `ctlin` object

The first outlet always corresponds to the value output by the controller. In the event that the object only has two outlets, the second outlet will correspond either to the MIDI channel number, as in the case of [`ctlin` 7] and [`ctlin` b 7], or the controller number, as in the case of [`ctlin` b -1 1] and [`ctlin` -1 17]. Take a moment to think carefully about this and be able to explain why.

If we double-click on a MIDI object while the patch is in performance mode, a contextual menu will appear, allowing one of the available devices to be selected. The selected device would be used in place of any that may have been provided as an argument to the object.

Now let's take a look at a patch that lets us modify the sound of a software or hardware synthesizer connected to the computer via MIDI. Open the file **09_01_MIDI_synth.maxpat**, shown in figure 9.5.

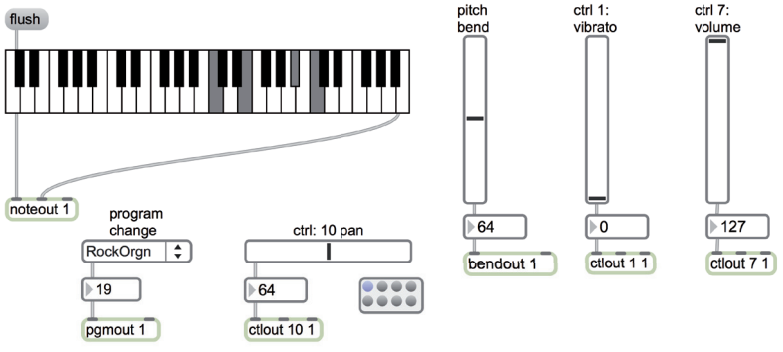

Fig. 9.5 The file **9_01_MIDI_synth.maxpat**

By default, this patch connects to the computer's internal synthesizer, but it is always possible to use another device by changing the settings in the MIDI Setup window.

(...)

other sections in this chapter:

Interlude E
MAX FOR LIVE

from "Electronic Music and Sound Design" Vol. 2 by Alessandro Cipriani and Maurizio Giri

LEARNING AGENDA

PREREQUISITES FOR THE CHAPTER

- The contents of Volume 1, chapters 5, 6, 7, 8 and 9 (theory and practice) and Interludes C and D
- Knowledge of the main functions of the Ableton Live program

OBJECTIVES

Abilities

- To know how to create a Max for Live device
- To know how to control the Live environment using the Live API

CONTENTS

- Building Audio and MIDI devices with Max for Live
- Building virtual instruments with Max for Live
- Using the Live API
- The hierarchical structure of the Live Object Model

ACTIVITIES

- Building and Modifying algorithms

SUPPORTING MATERIALS

- List of Max objects - List of attributes, messages and actions for specific Max objects - Glossary

IE.1 AN INTRODUCTION TO MAX FOR LIVE

In order to be able to follow along with this Interlude, you need to have a user license for both Max for Live and Ableton Live.[1] Since the information contained here is almost exclusively geared toward the use of Max for Live, it will not be a prerequisite for understanding future chapters in this series. Those who do not own both of the licenses mentioned above can therefore skip this Interlude without any negative impact on their understanding of the material presented in later chapters.

WHAT IS ABLETON LIVE?

Ableton Live (or more simply, Live) is a DAW (Digital Audio Workstation) application – in other words software designed for recording, manipulating and playing back audio tracks. In addition to this it can also handle MIDI sequences which can be used to control either external hardware or virtual instruments internal to the application itself. Live's most interesting feature is the ability to handle both audio and MIDI tracks non-linearly. Although the vast majority of DAW systems and sequencers actually put the sequences along a timeline that (needless to say) runs linearly through time, Live is actually able to trigger sequences independently of one another, and therefore create a kind of real-time arrangement of them that can be modified at will during performance.

Since this text is not intended to be a manual for Live, we will assume that you already have some knowledge of the program's main functions. In particular, this implies a general familiarity with the overall structure of the Live set and its sections, including the Help section and Live lessons, the use of audio clips and MIDI in "Session View" and "Arrangement View" modes, the use of standard Live *devices*, the use of the Group function to create *Audio Racks* containing multiple *devices*, the use of different *device* chains within an *Audio Rack*, the use of automation and the use of envelopes inside *clips*.

WHAT IS MAX FOR LIVE?

Max for Live is an extension of Live that allows Max patching language to be used to create new plug-ins (called *devices* in Live jargon)[2] which can be used inside the Live application itself. Max for Live can be used to control different operations in Live, such as changing the volume or panning of a track, stopping or starting a *clip*, or even modifying the parameters of other *devices*.

You do not need a full Max license in order to use Max for Live (henceforth referred to as M4L)[3] – you only need an M4L users' license in addition to your

[1] As of Live version 9, Max for Live is included in the packages available with Live Suite.

[2] We already mentioned plug-ins in section 3.8T in the first volume. They are basically software components that are "hosted" inside another program and used to further augment its functionality. Some typical examples of audio plug-ins are effects like compressors, delays and equalizers or virtual instruments such as synthesizers and samplers.

[3] M4L is the abbreviation commonly used for "Max for Live" by its user community.

Live license. Even if you own M4L but not Max itself, you will still be able to use the patches contained in this and the previous volume. However, be aware that in this case Max will not be capable of handling audio and MIDI input and output autonomously. This means that all audio signals and MIDI messages must inevitably be routed through Live.

IE.2 BASICS – CREATING AN AUDIO EFFECT WITH M4L

First of all, we strongly suggest that you read all of the sections of this chapter in order! In other words, even if you just want to use M4L to create an Instrument and not an Audio Effect, you shouldn't skip over that section because it also contains information that is essential to the understanding of subsequent sections.

Fig. IE.1 The "Max Chorus" device

Before proceeding any further, make sure that you have installed Max for Live on your computer, in addition to all of the M4L packages available in your account at the www.ableton.com website.

In the first column of the *Live Device Browser* (the area located to the left of the tracks in Live's main window), select the *Max for Live* category. Three folders should then appear in the second column of the *Browser*: *Max Audio Effect*, *Max Instrument* and *Max MIDI Effect*. These three folders are used, respectively, to group together audio effects, virtual instruments and MIDI effects created with M4L. Now, open the Max Audio Effect folder, find the "Max Chorus" device (the list is in alphabetical order) and drag it to an audio track (figure IE.1).

The device, visible in the lower part of the figure, was actually written in Max. To see the associated patch, you will need to click on the first of the three circular icons located on the upper right side of the device's title bar (figure IE.2).

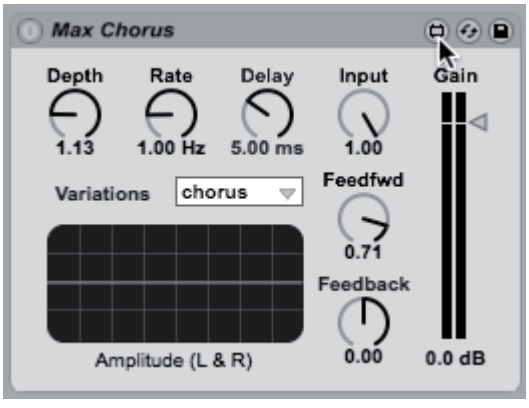

Fig. IE.2 Opening the patch of an M4L device

This will cause the Max application to be launched allowing you to be able to see and modify the patch. The patch initially opens in presentation mode, and by clicking on the familiar icon that enables and disables this mode (the little blackboard in the lower part of the patcher window), the patch can be switched to patching mode.

The patch associated with the "Max Chorus" device is shown in figure IE.3. Since this is a very simple patch, we won't bother to describe it. Note, however the pair of **teeth~** objects in the lower half of the patch – these are simply comb filters whose feed-forward delay and feedback delay times can be adjusted independently (unlike the comb~ object).

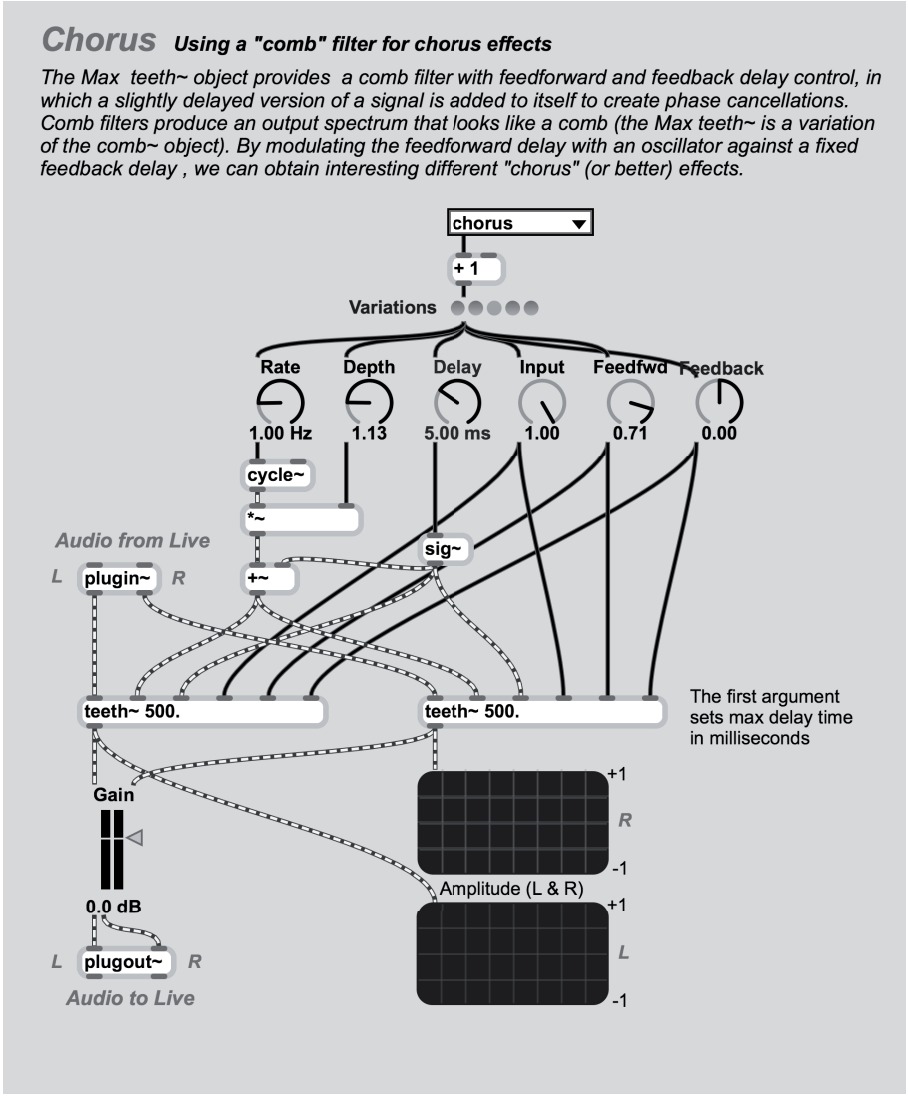

Fig. IE.3 The patch for the "Max Chorus" device

Furthermore, notice the title in the patcher window: "Max Chorus.amxd". The suffix for a patch used as a Live device is .amxd and not .maxpat. Try adding an audio clip to the track to test out the device. You can then try out other M4L audio effect devices, and open their associated patches to see how they work. Have Fun!

Now let's take a look at how to create a new M4L device. To begin with, notice that the first device in the list inside the *Max Audio Effect* folder is simply called "Max Audio Effect" and that it has a different icon from the other devices. This is actually a *template* that we can use as the basis for creating our new device. Drag it to an audio track, or alternatively double-click on its icon, and the default device will appear on the selected track (figure IE.4).

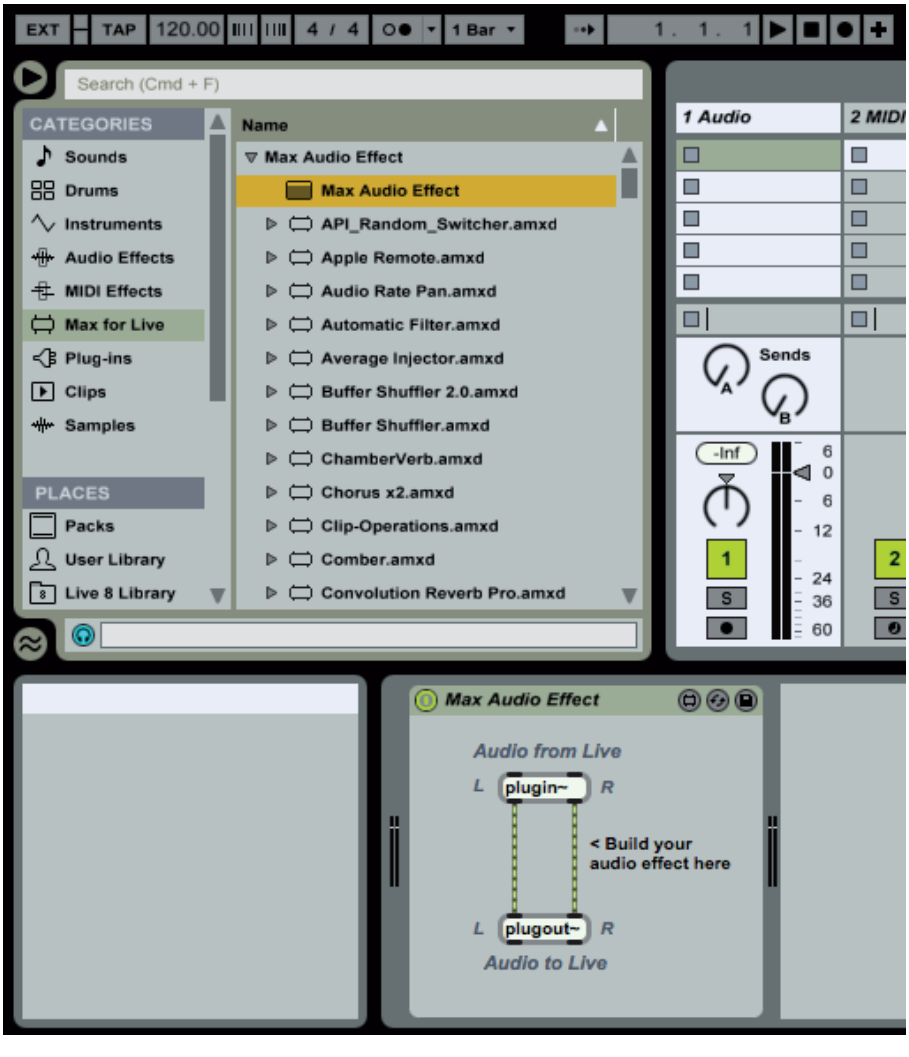

Fig. IE.4 The default device for audio effects

When you open this device you can modify it and save it under a new name. By default, M4L will ask you to save it inside the Max Audio Effect folder located in your "User Library." We suggest that you do not change this path, but rather optionally create a sub-folder where you can collect your own devices.

The default device contains only two objects: **plugin~** and **plugout~**. The former receives audio signals from the track (or from a previous device if there is one) and the latter sends audio to the track's output (or to a subsequent device, if there is one). These two objects basically replace the **adc~** and **dac~** objects used in normal Max patches.

Let's try building a *slapback delay* (see chapter 6.2P). Modify the device as shown in figure IE.5, and save the patch under the name "My Slapback Delay. amxd".

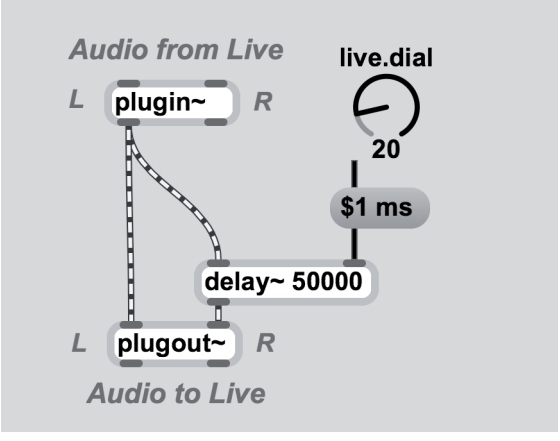

Fig. IE.5 Our first device: a *slapback delay*

Notice that a horizontal line labeled "Device Vertical Limit" will appear when the patch is in edit mode. Only objects located above this line will be visible in the device loaded by Live. As you can see, our device is not in presentation mode; a device can be automatically opened in presentation mode, just as a normal Max patch can, by activating the "Open in Presentation" attribute in the patcher inspector (see Interlude D, section ID.2). Naturally, in this case, all of the objects that we want to use must also be included in presentation mode.

The circular object visible on the upper right is a **live.dial** – this is the "Live version" of the standard **dial** object that we are already familiar with.[4] This object handles numerical values between 0 and 127 by default, but we will shortly see some common features of the "live.*" object set that allow this range to be customized.[5] Load an audio clip into the track and try out the device.

[4] This object is located in the Live category in the Object Explorer. If you cannot find it, you can always create it by typing the "l" (lower-case "L") key when the patch is in edit mode. This will cause a generic object box to appear with a text completion menu containing all the names of the objects in the Live category.

[5] We are using the shorthand term "live.*" here to mean any Max object in the Live category whose name begins with the characters "live.".

THE PARAMETERS OF THE "LIVE.*" OBJECTS

Let's pause for a moment here, and look at the `live.dial` object in depth, in order to present some important features common to all of the objects in the "live.*" object set.

As we have already said, the object's default output values are between 0 and 127. Naturally, the object's number format and its value range can be changed. If you open the object's inspector you will notice that there are a series of attributes in the "Parameter" category which are features of the "live.*" objects. Locate the following attributes in the list: "Type", "Range/Enum", "Unit Style" and "Steps" (figure IE.6).

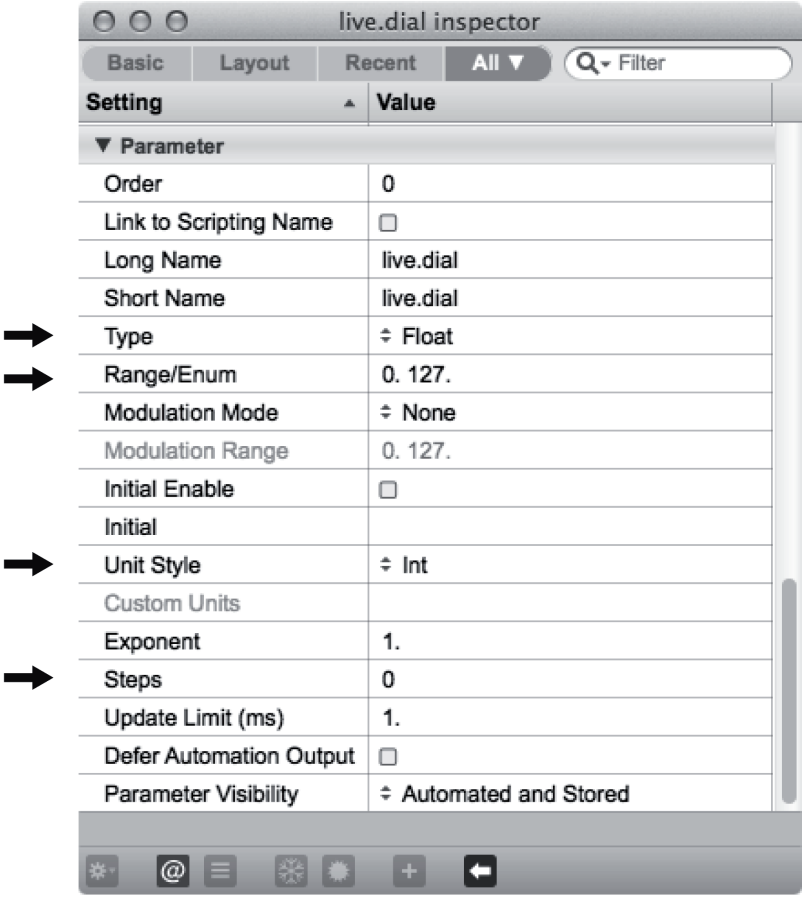

Fig. IE.6 The `live.dial` inspector

(...)

other sections in this chapter:

Alessandro Cipriani • Maurizio Giri

Electronic Music and Sound Design

Theory and Practice with Max and MSP • volume 2

Topics

Digital Audio and Sampled Sounds: decimation, blocks technique, slicing, scrubbing, timing and polyphony - Delay Lines: echoes, looping, flanger, chorus, comb and allpass filters, phaser, pitch shifting, reverse, variable delay, Karplus-Strong algorithm - Creative Uses of Dynamics Processors: envelope followers, compressors, limiters, live normalizers, expanders, gates, side chains, ducking - The Art of Organizing Sound: simple, complex and compound motion processes, motion within timbre, algorithmic control of motion, motion sequences - MIDI and Handling MIDI Messages in Max - Max for Live: audio effects, virtual instruments, MIDI effects, Live API and Live Object Model.

With their *Electronic Music and Sound Design: Theory and Practice with Max and MSP* (...) Alessandro Cipriani and Maurizio Giri have produced a series of "interactive and enhanced books" that present the student of computer music with the finest and most comprehensive electroacoustic curriculum in the world. By "illustrating" the text with a wealth of figures and clearly explained equations, they take the reader "under the hood" and reveal the algorithms that make our computing machines "sing". By using David Zicarelli's incredibly powerful and intuitive media-toolkit – *Max* to create hundreds of synthesis, signal processing, algorithmic composition, interactive performance, and audio analysis software examples, Cipriani and Giri have provided the means for students to learn by hearing, by touching, by modifying, by designing, by creating, and by composing. (...)
I firmly believe that this series by Cipriani and Giri, these "interactive and enhanced books", (...) set the stage for the next generation of innovators. (...)
In *Electronic Music and Sound Design*, Cipriani and Giri feed the hands, they feed the ears, and they feed the minds of the students in ways and to a degree that no computer music textbook has ever done.
(from the Foreword by **Richard Boulanger**, Berklee College of Music)

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

ALESSANDRO CIPRIANI is a composer and tenured professor in electronic music at the Conservatory of Frosinone. His compositions have been performed at major festivals (such as the Venice Biennale and the International Computer Music Conference), and have been released on CDs and DVDs issued by Computer Music Journal (MIT Press), among others. He has given seminars at many European and American universities, including the University of California, Santa Barbara and the Sibelius Academy in Helsinki.

MAURIZIO GIRI is a conservatory professor of composition and Max programming. He is an instrumental and electroacoustic composer and developer of Max for Live devices. He was artist in residence at the Cité Internationale des Arts in Paris, and at GRAME in Lyon. He is also an associate member of the Nicod Institute at the École Normale Supérieure in Paris, where he is collaborating on projects relating to the philosophy of sound.

ISBN 978-88-905484-4-4

9 788890 548444

www.contemponet.com