





# Loadbang

Programming Electronic Music in Pd

Johannes Kreidler

wolke

Second Edition 2013  
© Johannes Kreidler  
All rights reserved by the publisher Wolke Verlag, Hofheim  
Translation: Mark Barden  
Cover design: Friedwalt Donner and pd-graz  
Typesetting: michon, Hofheim  
Printing: Fuldaer Verlagsanstalt  
ISBN 978-3-95593-055-4

## Table of Contents

Preface .....	13
Introduction to this book's methodology .....	14
1. Introduction to Pd .....	15
1.1 General remarks .....	15
1.2 Installing and setting up Pd .....	18
2. Programming with Pd for the first time .....	19
2.1 Introduction .....	19
2.1.1 A simple example .....	19
2.1.2 Surface elements in Pd .....	24
2.1.3 Summary .....	26
2.1.4 Appendix .....	27
2.1.4.1 List of all objects .....	27
2.1.4.2 Help file .....	27
2.1.4.3 Duplication .....	28
2.1.4.4 Short cuts .....	28
2.1.4.5 Comments .....	28
2.1.5 For those especially interested: Atoms .....	28
2.2 The control level .....	30
2.2.1 Mathematical operations and order .....	30
2.2.1.1 Theory .....	30
2.2.1.1.1 Basic mathematical functions .....	30
2.2.1.1.2 Order .....	31
2.2.1.1.3 Expression .....	33
2.2.1.1.4 Other mathematical operations .....	33
2.2.1.1.5 Float and counter .....	35
2.2.1.1.6 Summary .....	36
2.2.1.2 Applications .....	36
2.2.1.2.1 Two frequencies—two volume levels .....	36
2.2.1.2.2 An interval .....	37
2.2.1.2.3 Random melody .....	37
2.2.1.2.4 Rounding .....	37
2.2.1.2.5 How long is this score? .....	37
2.2.1.2.6 Counting in series .....	38
2.2.1.2.7 Random without repetitions .....	38
2.2.1.2.8 More exercises .....	38
2.2.1.3 Appendix .....	39
2.2.1.3.1 Input for bang .....	39
2.2.1.3.2 How numbers are represented .....	39
2.2.1.3.3 More on trigger .....	39

2.2.1.4 For those especially interested . . . . .	40
2.2.1.4.1 About series . . . . .	40
2.2.1.4.2 Regarding float . . . . .	40
2.2.2 Different types of data . . . . .	40
2.2.2.1 Theory . . . . .	40
2.2.2.1.1 Bang—a GUI object . . . . .	40
2.2.2.1.2 Messages . . . . .	40
2.2.2.1.3 Lists . . . . .	41
2.2.2.1.4 Messages with variables . . . . .	42
2.2.2.1.5 Messages: Set . . . . .	43
2.2.2.1.6 Makefilename . . . . .	43
2.2.2.1.7 Openpanel . . . . .	44
2.2.2.1.8 Simple data storage . . . . .	44
2.2.2.1.9 Route . . . . .	44
2.2.2.1.10 Demultiplex . . . . .	45
2.2.2.1.11 Spigot . . . . .	46
2.2.2.1.12 Toggle . . . . .	46
2.2.2.2 Applications . . . . .	47
2.2.2.2.1 Using lists with pitches and dynamics . . . . .	47
2.2.2.2.2 On/off switch . . . . .	47
2.2.2.2.3 Pitches with names . . . . .	47
2.2.2.2.4 A simple sequence . . . . .	47
2.2.2.2.5 A limited counter . . . . .	48
2.2.2.2.6 More exercises . . . . .	49
2.2.2.3 Appendix . . . . .	49
2.2.2.3.1 Symbol boxes . . . . .	49
2.2.2.3.2 Slider . . . . .	49
2.2.2.3.3 Radio . . . . .	50
2.2.2.3.4 Using Slider and Radio . . . . .	50
2.2.2.4 For those especially interested: Other type specifications and more about boxes . . . . .	50
2.2.3 Time operations . . . . .	51
2.2.3.1 Theory . . . . .	51
2.2.3.1.1 Metro . . . . .	51
2.2.3.1.2 Delay . . . . .	51
2.2.3.1.3 Pipe . . . . .	51
2.2.3.1.4 Line . . . . .	52
2.2.3.1.5 Timer . . . . .	53
2.2.3.2 Applications . . . . .	54
2.2.3.2.1 Automatic random melody . . . . .	54
2.2.3.2.2 Glissando . . . . .	54
2.2.3.2.3 Glissando melody . . . . .	55
2.2.3.2.4 Irregular random rhythms . . . . .	55
2.2.3.2.5 Canons . . . . .	56
2.2.3.2.6 Rests . . . . .	57
2.2.3.2.7 Crescendo/Decrescendo . . . . .	58

2.2.3.2.8 Metronome .....	58
2.2.3.2.9 More exercises.....	59
2.2.3.3 Appendix .....	60
2.2.3.3.1 Distributing lists .....	60
2.2.3.3.2 Time resolution for control data .....	60
2.2.4 Miscellaneous .....	61
2.2.4.1 Sending and receiving .....	61
2.2.4.1.1 Send/Receive .....	61
2.2.4.1.2 Sending with lists .....	63
2.2.4.1.3 A series of lists .....	63
2.2.4.1.4 Value .....	64
2.2.4.2 Loadbang .....	64
2.2.4.3 GUI options.....	64
2.2.4.3.1 Number and symbol box .....	65
2.2.4.3.2 Bang .....	65
2.2.4.3.3 Toggle .....	66
2.2.4.3.4 Slider .....	66
2.2.4.3.5 Radio .....	67
2.2.4.3.6 Canvas .....	67
2.2.4.3.7 Examples of altered GUI objects .....	67
2.2.4.3.8 Change font size .....	68
2.2.4.3.9 Tidy up .....	68
2.2.4.4 Subpatches .....	69
2.2.4.4.1 Space .....	69
2.2.4.4.2 Modularization .....	71
3. Audio .....	73
3.1 Basics .....	73
3.1.1 Pitch .....	73
3.1.1.1 Theory .....	73
3.1.1.1.1 Controlling speakers digitally .....	73
3.1.1.1.2 Waves .....	75
3.1.1.1.3 Measurement .....	76
3.1.1.1.4 Sample rate .....	79
3.1.1.1.5 Samples—milliseconds .....	80
3.1.1.2 Applications .....	80
3.1.1.2.1 Tempered—random .....	80
3.1.1.2.2 More exercises .....	81
3.1.1.3 Appendix .....	81
3.1.1.3.1 Nyquist Theorem .....	81
3.1.1.3.2 DSP .....	82
3.1.1.4 For especially interested .....	82
3.1.1.4.1 da- / ad- conversion .....	82
3.1.1.4.2 Sound waves .....	83
3.1.1.4.3 Converting MIDI numbers into frequencies .....	83
3.1.1.4.4 Noise periodicity .....	84

3.1.2 Volume .....	85
3.1.2.1 Theory .....	85
3.1.2.1.1 Measurement .....	85
3.1.2.1.2 Problems .....	89
3.1.2.1.3 Phase .....	90
3.1.2.1.4 Sound waves are additive .....	91
3.1.2.2 Applications .....	93
3.1.2.2.1 Chord .....	93
3.1.2.2.2 Glissandi .....	93
3.1.2.2.3 Processing adc-input .....	94
3.1.2.2.4 Oscillator concert .....	94
3.1.2.2.5 More exercises .....	96
3.1.2.3 Appendix .....	96
3.1.2.3.1 Other tilde objects .....	96
3.1.2.3.2 Bit depth .....	97
3.1.2.4 For those especially interested .....	97
3.1.2.4.1 Sound pressure vs. sound intensity .....	97
3.1.2.4.2 Control data vs. signals .....	97
3.2 Additive Synthesis .....	99
3.2.1 Theory .....	99
3.2.1.1 The harmonic series .....	99
3.2.2 Applications .....	102
3.2.2.1 A random klangfarbe (German: sound color) .....	102
3.2.2.2 Changing one klangfarbe into another .....	102
3.2.2.3 Natural vs. equal-tempered .....	103
3.2.2.4 More exercises .....	103
3.2.3 Appendix .....	104
3.2.3.1 Pd's limitations .....	104
3.2.4 For those especially interested .....	104
3.2.4.1 Studie II .....	104
3.2.4.2 Composing with spectra .....	104
3.3 Subtractive synthesis .....	105
3.3.1 Theory .....	105
3.3.1.1 White noise .....	105
3.3.1.2 Filters .....	105
3.3.2 Applications .....	107
3.3.2.1 Filter colors .....	107
3.3.2.2 Telephone filters .....	108
3.3.2.3 More exercises .....	108
3.3.3 Appendix .....	108
3.3.3.1 White noise and clicks .....	108
3.3.3.2 Pink noise .....	109
3.3.3.3 DC offset .....	109
3.3.4 For those especially interested .....	110
3.3.4.1 How digital filters work .....	110
3.4 Sampling .....	112

3.4.1 Theory . . . . .	112
3.4.1.1 Storing sound . . . . .	112
3.4.1.1.1 Sound files . . . . .	112
3.4.1.1.2 Buffers . . . . .	112
3.4.1.2 Playback of saved sound . . . . .	120
3.4.1.3 Audio delay . . . . .	123
3.4.2 Applications . . . . .	125
3.4.2.1 A simple sampler . . . . .	125
3.4.2.2 With variable speed . . . . .	125
3.4.2.3 Any position . . . . .	126
3.4.2.4 Sampler-player . . . . .	127
3.4.2.5 Loop generator . . . . .	130
3.4.2.6 Reverb . . . . .	132
3.4.2.7 Texture . . . . .	133
3.4.2.8 Comb filter . . . . .	133
3.4.2.9 Octave doubler . . . . .	134
3.4.2.10 Karplus-Strong algorithm . . . . .	136
3.4.2.11 More exercises . . . . .	138
3.4.3 Appendix . . . . .	138
3.4.3.1 Array oscillator . . . . .	138
3.4.3.2 Array playback . . . . .	138
3.4.3.3 Playing back an array in a block . . . . .	139
3.4.3.4 Glissandi of samples . . . . .	139
3.4.3.5 Additive synthesis with array . . . . .	142
3.4.3.6 Latency . . . . .	143
3.4.4 For especially interested . . . . .	144
3.4.4.1 4-point-interpolation . . . . .	144
3.4.4.2 Sample-wise delay . . . . .	145
3.5 Wave shaping . . . . .	146
3.5.1 Theory . . . . .	146
3.5.1.1 Waveforms . . . . .	146
3.5.1.2 Transfer functions . . . . .	151
3.5.1.3 (Controlled) Random waveforms . . . . .	152
3.5.1.4 Wave stealing . . . . .	157
3.5.2 Applications . . . . .	158
3.5.2.1 Singing waveforms . . . . .	158
3.5.2.2 Transfers . . . . .	159
3.5.2.3 Even / odd partials . . . . .	159
3.5.2.4 More exercises . . . . .	160
3.5.3 Appendix . . . . .	160
3.5.3.1 Foldover . . . . .	160
3.5.4 For those especially interested . . . . .	162
3.5.4.1 GENDY . . . . .	162
3.6 Modulation synthesis . . . . .	163
3.6.1 Theory . . . . .	163
3.6.1.1 Ring modulation . . . . .	163

3.6.1.2 Frequency modulation .....	165
3.6.2 Applications .....	167
3.6.2.1 More sonically complex ring modulation.....	167
3.6.2.2 Live ring modulation .....	167
3.6.2.3 Live frequency modulation.....	167
3.6.2.4 More exercises.....	168
3.6.3 Appendix.....	168
3.6.3.1 Phase modulation .....	168
3.7 Granular synthesis.....	170
3.7.1 Theory.....	170
3.7.1.1 Theory of granular synthesis.....	170
3.7.2 Applications.....	175
3.7.2.1 Live granular synthesis.....	175
3.7.2.2 Live with feedback .....	177
3.7.2.3 More exercises.....	177
3.7.3 Appendix .....	178
3.7.3.1 Granular technique as a synthesizer .....	178
3.8 Fourier analysis .....	179
3.8.1 Theory.....	179
3.8.1.1 Analyzing partials.....	179
3.8.1.2 Analyze whatever signal you want.....	182
3.8.2 Applications .....	184
3.8.2.1 Filters .....	184
3.8.2.2 Folding .....	185
3.8.2.3 Compressor .....	186
3.8.2.4 Spectral delay .....	187
3.8.3 Appendix.....	188
3.8.3.1 Fiddle~ .....	188
3.8.3.2 Tuner.....	190
3.8.3.3 Octave doubler #2.....	191
3.8.3.4 Pitch follower.....	191
3.8.3.5 More exercises.....	191
3.9 Amplitude corrections .....	192
3.9.1 Theory.....	192
3.9.1.1 Limiter.....	192
3.9.1.2 Compressor .....	194
3.9.2 Applications.....	194
3.9.2.1 Larsen tones .....	194
3.9.2.2 More exercises.....	195
3.9.3 Appendix .....	195
3.9.3.1 Movements in space .....	195
3.9.4 For those especially interested .....	197
3.9.4.1 Other windows .....	197

4. Controlling sound .....	200
4.1 Algorithms .....	200
4.1.1 Theory.....	200
4.1.1.1 What are algorithms?.....	200
4.1.2 Applications.....	201
4.1.2.1 Stochastics .....	201
4.1.2.2 Recursive systems.....	204
4.1.2.3 More exercises.....	205
4.1.3 Appendix .....	205
4.1.3.1 DSP loop.....	205
4.1.4 For those especially interested .....	206
4.1.4.1 Algorithmic composition .....	206
4.2 Sequencer .....	207
4.2.1 Theory .....	207
4.2.1.1 Text file .....	207
4.2.1.2 Qlist.....	209
4.2.2 Applications .....	210
4.2.2.1 Score for a patch .....	210
4.2.2.2 More exercises.....	212
4.2.3 Appendix.....	212
4.2.3.1 Modifying qlist .....	212
4.2.4 For those especially interested .....	214
4.2.4.1 Creating lists externally: Lisp .....	214
4.3 HIDs.....	215
4.3.1 Theory.....	215
4.3.1.1 Keyboard and mouse .....	215
4.3.1.2 MIDI.....	216
4.3.1.3 Using signals to control sound .....	218
4.3.2 Applications .....	218
4.3.2.1 Playing patches live .....	218
4.3.2.2 More exercises.....	219
4.3.3 Appendix .....	220
4.3.3.1 Other HIDs .....	220
4.3.3.2 Video input.....	220
4.3.4 For those especially interested .....	220
4.3.4.1 Instrument design .....	220
4.4 Network.....	221
4.4.1 NetSend / NetReceive .....	221
4.4.2 OSC.....	221
5. Miscellaneous .....	223
5.1 Streamlining .....	223
5.1.1 Theory.....	223
5.1.1.1 Subpatches .....	223
5.1.1.2 Abstractions.....	226
5.1.1.3 Expanding Pd .....	230

5.1.2 Applications .....	231
5.1.2.1 Customize your Pd .....	231
5.1.2.2 More exercises .....	232
5.1.3 Appendix .....	232
5.1.3.1 Creating a patch automatically .....	232
5.1.4 For those especially interested .....	235
5.1.4.1 Writing your own objects .....	235
5.2 Visuals .....	236
5.2.1 Theory .....	236
5.2.1.1 Pd is visual and this can be programmed .....	236
5.2.2 Applications .....	237
5.2.2.1 Main patch window and subpatches .....	237
5.2.2.2 Canvasses as display .....	237
5.2.2.3 Canvasses as expanded GUI .....	239
5.2.2.4 More exercises .....	240
5.2.3 Appendix .....	240
5.2.3.1 Data structures .....	240
5.2.4 For those especially interested .....	247
5.2.4.1 GEM .....	247
Afterword .....	248
Appendix A. Solutions .....	249
2.2.1.2.8 .....	249
2.2.2.2.6 .....	250
2.2.3.2.9 .....	251
3.1.1.2.2 .....	253
3.1.2.2.5 .....	254
3.3.2.3 .....	255
3.4.2.11 .....	256
3.5.2.4 .....	261
3.7.2.3 .....	262
3.8.3.5 .....	263
3.9.2.2 .....	264
4.1.2.3 .....	265
4.2.2.2 .....	267
5.1.2.2 .....	268
5.2.2.4 .....	269
Index .....	273

## Preface

This book is the result of my experience of teaching electronic music. Through the teaching process, I became familiar with the most common stumbling blocks students encounter—especially when the student's native language is not the language in which lessons are conducted.

Pd (Pure Data) is a professional, high-performance programming language for electronic sound processing. It is *open source*, i.e. available for free on the Internet. One disadvantage of this is that Pd is only discussed in certain institutions or Internet forums. The complicated technical terminology usually found there is enormously difficult for beginners to understand. This book will help first-time users to clear those first few hurdles when learning Pd.

Pd's main designer, Miller Puckette, is also writing a book about the theory and technology of electronic music processing with Pd. Surely there is no better teacher of a programming language than the person who designed it; his primarily scientific approach certainly does cover all the material in a thorough, systematic fashion. However, his method of teaching can be difficult to comprehend. My pedagogical experience has been that Puckette's text demands a large amount of mathematical, computer science, and terminological knowledge from its readers.

This book is designed for self-study, principally for composers. It begins with explanations of basic programming and acoustic principles before gradually building up to the most advanced electronic music processing techniques. Some knowledge of physics is assumed and explanations of basic physics concepts have been intentionally omitted. My book's teaching approach is focused primarily on hearing, which I regard as a faster and more enjoyable way to absorb new concepts than through abstract formulas. In terms of mathematics, I explain only what is absolutely necessary to comprehend a given processing concept. I explain the various techniques from a compositional perspective, rather than attempting a computer science-, math-, or physics-based discussion of processing phenomena or structures. Therefore, the decisions and comments I have made are purely subjective and are open to debate.

This book would not have been possible without the support of Prof. Mathias Spahlinger, the expert supervision of Prof. Orm Finnendahl, suggestions and patches from the Pd community, the manuscript editing and DocBook-XML coding efforts of Esther Kochte. I would also like to thank Mark Barden for the English translation and the Musikhochschule Freiburg and the state of Baden-Württemberg for financing the project. This book will hopefully increase interest in electronic music, thereby indirectly enriching the aesthetic discourse of New Music.

Johannes Kreidler, January 2008

## Introduction to this book's methodology

The following material begins with basic computer knowledge. The first steps are therefore described in meticulous detail.

Pd can run on different platforms (like Linux, OS X, or Windows) and this book is not platform-specific. Problems relating to the operating system will not be discussed, as they are simply beyond the scope of this tutorial (and it is also quite likely that changes—updates, bug fixes, etc.—will occur in the near future). It is therefore assumed that Pd has been correctly installed and has been integrated with the hardware environment (consult an Internet forum to resolve any of these sorts of problems, e.g. “Pd-list”).

How to use this book: Each lesson is comprised of a theory part, a practice part and an appendix, as well as individual aspects that are explained in greater detail at the end of each section. This in-depth information is aimed at advanced users and is not essential to acquire a basic working knowledge of Pd. I recommend working through the whole book without consulting these additional details first, then going back to learn them later.

Now and again, some fundamental concepts of acoustics are discussed. The exercises contain not only specific compositional questions, but also applications that are useful for musicians' everyday needs—e.g. tools like the metronome or tuning device. In this respect, the tutorial could be used by interpreters as well as composers.

# Chapter 1

## Introduction to Pd

### 1.1 General remarks

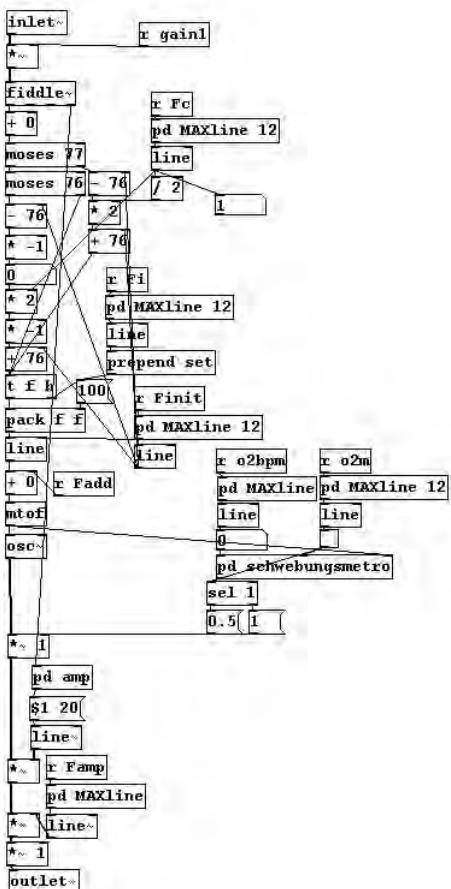
Pd (Pure Data) is a programming language for electronic music. Creating music on a computer is technically referred to as DSP (digital signal processing). “Digital” means that information is represented by digits—computers, as you may know, work only with numbers. “Signal” is the technical term for a special mode of computer operation that deals with sound. “Processing” refers to functions executed by the computer.

Pd was initiated by American software engineer Miller Puckette, who previous co-developed the well known and similarly structured software Max/Msp. Pd is not commercial software; i.e., it was not developed by a corporation and is not for sale. Instead, it is “open source”: its source code can be viewed by anyone. This source code is also not the (patented) property of a corporation, but is rather freely available to all. This also means that, provided sufficient knowledge, anyone can change the program. Today, many other programmers, musicians, acoustic engineers, and composers have joined Miller Puckette to continue Pd’s development. As a result of this, there is no final, definitive version of Pd; the program is under constant development. In addition to the huge advantage of free availability on the Internet, it is also “democratically” expanded and optimized on a professional level. One drawback to this is that a detailed operating manual for users who lack programming experience has not existed until now. In contrast to a corporation, which has a monetary interest in ensuring that first-time users can easily operate new software, the open source movement lacks such a driving force to make itself accessible. This book is an attempt to fill that gap.

In precise terms, Pd is a “real-time graphical programming environment for audio processing”. Traditionally, programmers work with text-based programming languages. They create what is called “code”, which is processed by a computer to produce a result. To carry out its programming functions, Pd uses visual objects that the user places and alters on the screen. These visual objects—small boxes that can be connected to each other—are a throwback to analogue studios that were used to produce electronic music before the advent of computers: various devices—today symbolized by our little boxes—are connected to each other using lines that—like cables—symbolize physical connections between the boxes. (Due to this type of connection, Pd is referred to as a datastream-oriented programming language.)



An analog studio—devices are connected with cables.



Pd boxes are connected to each other.

One major advantage of Pd is the aspect of „real-time“. This means that, in contrast to most programming environments where a text is first entered that must be separately processed by the computer before obtaining a result, changes in Pd can be made during performance. Like on a classical instrument, the user hears the result instantaneously and can change it immediately. This makes Pd especially well suited for use in live performance.

Pd has become much more than a programming language for electronic music. Since users across the globe can participate in the project, there are user-programmed modules for what are called „externals“: video, Internet connection, joystick integration, etc. Whole libraries of these modules even exist („external libraries“). Some of these externals have been integrated into the regular version of Pd.

## 1.2 Installing and setting up Pd

Readers of this book should have Pd installed on their computer so they are able to try out the processes described. Without this simultaneous practical experience, this tutorial will be difficult to understand.

First you need a computer with at least 128 MB main memory, a 500 MHz processor and ca. 500 MB hard disk space (these are the absolute minimum requirements!). Pd works with the following operating systems: Linux, OS X, and Windows.

Then you need to download the newest version of Pd-extended from the Internet. Enter “Pd-extended” into an Internet search engine. Since the address for the download portal may change in the future, no link to the site will be provided here. Pd-extended is a version of the original software (also called “Pd vanilla”) that has been expanded with numerous libraries. Most of the exercises described here work with the original version of Pd, but not all of them. The extra objects in Pd-extended make the program much more practical in general. This tutorial assumes Pd-extended version 0.39 or higher.

Once Pd has been installed, we open it from the directory Pd/bin/. A window appears. This is the main control center, so to speak. Here you can test whether Pd is functioning properly: In the main menu, click on **Media** → **Test Audio and MIDI**. Under “TEST SIGNAL”, click first on the box next to “-40”, then on the box next to “-20”. You should hear a sine tone coming out of the computer’s loudspeaker (A4). If you do not, then you need to adjust your hardware settings (under **Media** → **Audio settings**). More information regarding problems that arise at this stage cannot be given here. For help resolving any problems, please consult the “Pd-list”, a forum of Pd users on the Internet. If a microphone is connected, the digits in at least the leftmost two boxes under “AUDIO INPUT” should change in response to sound picked up by the microphone. As long as the test tone is working, you can work with the program without a microphone. (By Chapter 3 at the latest, however, you will sometimes need a microphone.)

## Chapter 2

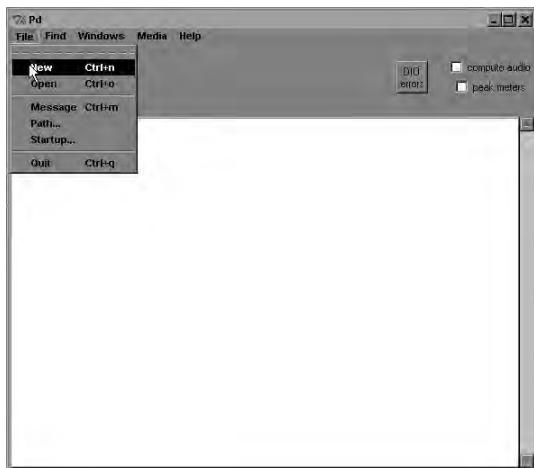
### Programming with Pd for the first time

In this chapter, we won't focus on producing music yet, but rather on understanding the way that computers and Pd handle data. We will be working with practical listening examples as often as possible to avoid unnecessarily abstract and dry technical explanations. However, the precise way in which computers produce sound will not be explained until Chapter 3. You should build the sample patches yourself in Pd. This first-hand experience will help solidify the concepts presented. Starting in Chapter 3, larger patches can be found at [www.kreidler-net.de/pd/patches/patches.zip](http://www.kreidler-net.de/pd/patches/patches.zip).

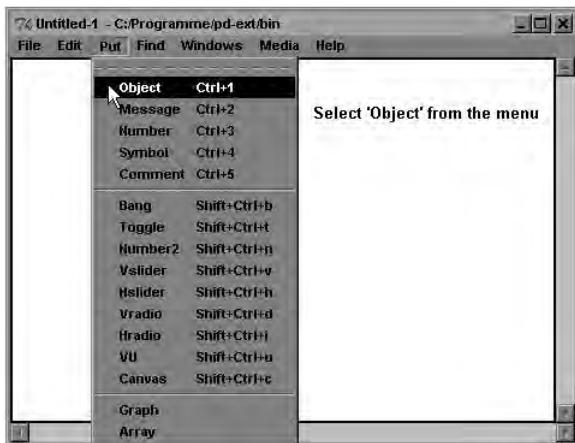
#### 2.1 Introduction

##### ◆ 2.1.1 A SIMPLE EXAMPLE

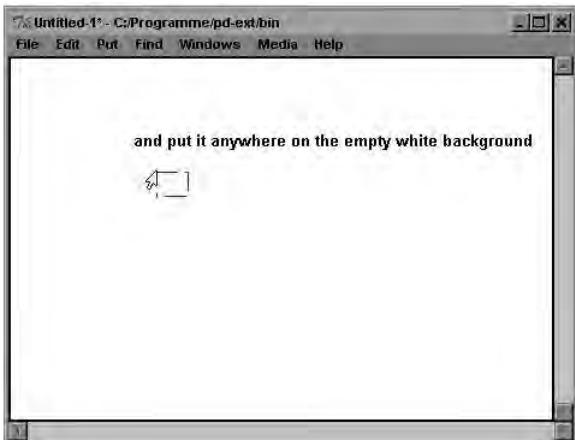
Once you have started Pd, the main Pd window appears on the monitor. Open a new programming window by clicking the **File** menu and then **New**.



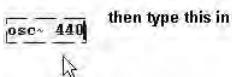
A new window opens. Add an object box under **Put** → **Object**, or with the keypad, using the listed key command: **Ctrl-1** (this is for Windows; other platforms may have different key commands).



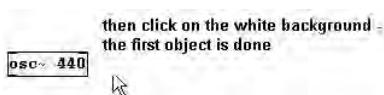
... you should see a blue box attached to the mouse cursor ...



Then click somewhere on the blank white surface in the new window to decouple the mouse from the object box. Type this into the box: "osc~ 440".

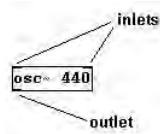


To accept what you have typed into the box, click anywhere outside the box on the white surface:



(The sign “~” means “tilde”; you’ll need to use this often in Pd.)

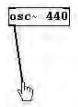
You now see a rectangular box with little black rectangles in the top and bottom corners. The upper rectangles are called “inlets”, the bottom rectangle is an “outlet”.



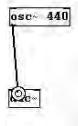
If you place the cursor onto the outlet rectangle, it changes to a circle (which resembles an open socket for a cable).



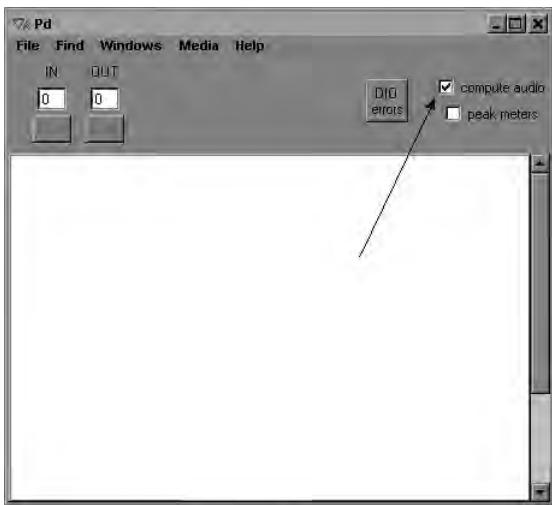
Now click on the rectangle and move the mouse while keeping the mouse button pressed. This draws a line that can be thought of as a cable.



But because you haven’t created an object to which you can attach this cable yet, the cable vanishes when you release the mouse button. Make another object just as you did the previous one and call it “dac~”. Position it below the “osc~” object by clicking it once so that it turns blue and moving the mouse with the mouse button held. Then start a cable from the outlet on “osc~” and connect it to the left inlet on “dac~”. The cursor changes into a circle when it is over the inlet.

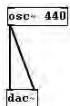


Release the mouse button. The “cable” now connects “osc~ 440” to “dac~”. You should hear a tone. If not, verify in the Pd main window that there is a check next to “compute audio” (in Linux: check if the field is red). If not, check the box with a mouse click:



(The “compute audio” function allows you to program in Pd without generating sound. This can save the computer much unnecessary processing power—though this is probably a non-issue with most computers these days.)

We hear a tone. To be specific, it is A4 (a' in the German system), also called the A440, the standard concert tuning pitch that has a frequency of 440 Hertz (the meanings of “Frequency” and “Hertz” will be explained later). Now connect the outlet from “osc~ 440” with the right inlet of “dac~” as well.



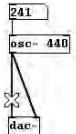
You should hear sound from both of the computer’s speakers. Now create a number box (**Put** → **Number** or with key command **Ctrl-3**) and attach its outlet to the inlet on the object named “osc~”. Then you need to change into what is called “Execute mode” (**Edit** → **Edit mode**, or with key command **Ctrl-E**; the cursor turns into an arrow). Click on the number box, hold the mouse button, and move the mouse up and down:



This changes the numbers and the pitch. The value should be at least 100; this range can be more finely adjusted by holding SHIFT while clicking and moving the mouse as described above.

Another way to enter values into the number box is to click on the number box, enter a value on the keyboard, and press ENTER.

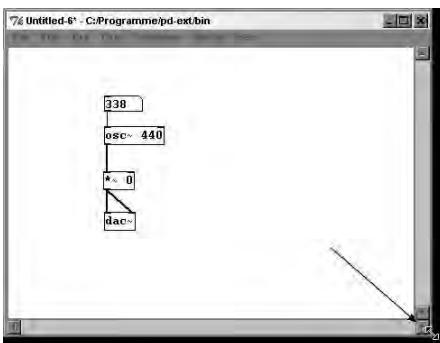
Now change back to the other mode, the “Edit mode” (**Edit** → **Edit mode**, or key command **Ctrl-E**). Move the cursor, which should have changed back to a hand, over the connection between “osc~” and “dac~”. The cursor becomes an X. Click on it, which will turn the cable blue.



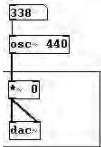
Then go to **Edit** → **Cut** or simply press BACKSPACE. This terminates the connection. Cut the other connection to “dac~” as well. Now create a new object where the cables used to be: “\*~ 0” and connect it to the other objects as shown:



Let's make some more room: Enlarge the window by clicking on its lower right corner, holding the mouse button, and pulling it down and to the right.

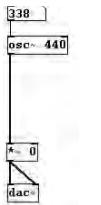


Then click on the lower right part of the white background near the “dac~” object, hold the mouse button, and draw a rectangle that includes the “dac~” and the “\*~” objects.



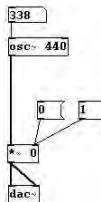
This is how you select a part of a patch. (You can also delete boxes this way. After selecting a portion of the patch, go to **Edit** → **Cut** or simply press BACKSPACE.)

When you release the mouse button, both objects appear in blue. Click on one of these selected objects, hold the mouse button, and pull them down to free up more space.



To deselect these objects, just click anywhere on the white background.

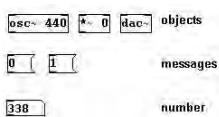
Then create two “Message” boxes (**Put** → **Message** or **Ctrl-2**) as below and enter “0” in one and “1” in the other.



Change back to execute mode (**Edit** → **Edit mode** or **Ctrl-E**) and click on the two message boxes in turn: clicking 1 turns the sound on, clicking 0 turns it off.

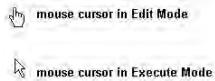
#### ◆ 2.1.2 SURFACE ELEMENTS IN Pd

The previous example covers most of the elements in Pd. Let’s take a closer look at them—we used three different kinds of boxes: *Object*, *Message*, and *Number*.

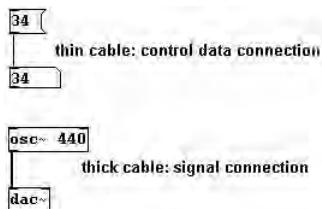


Object boxes are rectangular, message boxes have an indentation on the right side, and number boxes have a flat upper right corner.

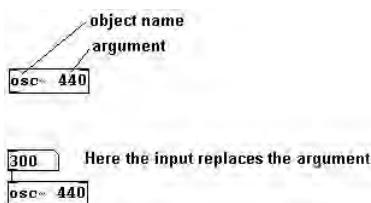
All of these boxes have inlets and outlets. The inlets are always on top, the outlets on bottom. You can always connect an outlet to an inlet (in this order). There is an edit mode and an execute mode. Edit mode is for programming and execute mode is for running the program. You can tell which mode you're in by looking at the cursor:



Let's take a closer look: There are two kinds of "cables", thick and thin. A thin cable connects the number box to the "osc~" object and a thick cable runs out of the "osc~" object. Thick cables transmit *signals*, while thin cables transmit only *control data*. With "compute audio" in the Pd main window, we determine whether the signals should be sent by marking or removing the checkmark. Moreover, all objects that produce signals or that work with signals as an input (input = that which goes into an inlet; output = that which comes out of an outlet), have a tilde ("~") after their name; other objects don't have this! These two levels are called the "control level" (where only control data flows, also called the "message domain") and the "signal level" (where signals flow, also called the "signal domain").



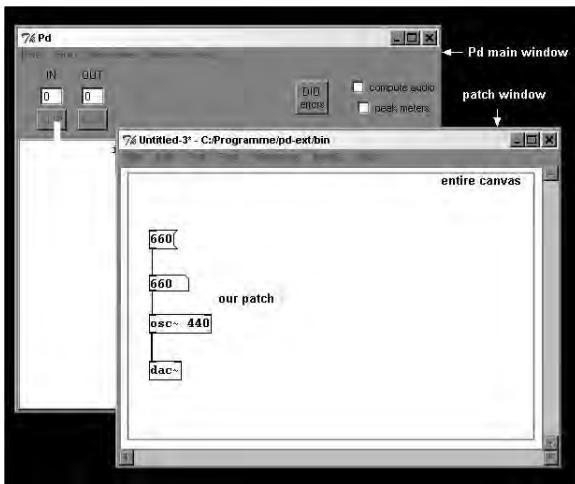
The first object you created was "osc~ 440", which is an "oscillator", and you heard a sine tone at 440 Hertz (the meaning of "Hertz" will be explained later). Then you made a number box and entered new values there, which caused the frequency of the tone you heard to change. That is the basic structure in Pd: an object has a name (if it produces signals, a tilde follows the name), then there is a space, and then one or several *arguments* follow (in this case, the initial 'argument' was "440"). With most objects, the arguments can be replaced with new values that are connected to the inputs (unlike with the "osc~" object here, the changed value usually goes into the far right inlet).



If new values are entered this way, the argument written in the object box is ignored (in this example, 300 instead of 440).

We can enter information in number boxes or message boxes. Message boxes also allow letters, which are called *symbols*. All of this information is referred to as *atoms*. An atom appears in a message box or in a number box (for more on atoms, see 2.1.5).

Another important term: The program that you write is called a *patch*. A patch first appears as a blank white background on which you write a program. This white background is also called a *canvas*.



#### ◆ 2.1.3 SUMMARY

- There are two modes: Edit mode and execute mode (you switch between them with **Ctrl-E** or under **Edit → Edit mode**). You program all the parts of a patch in edit mode and start all operations and sounds in execute mode.
- Within a patch, there is the control level and the signal level (control objects do not have a tilde after their names and are connected with thin cables; signal objects have a tilde at the end and are connected with thick cables). The signal level is only active if “compute audio” has been activated in the Pd main window.
- The elements of a patch are *objects*, *messages*, and *numbers*.
- An object often has one or several arguments (a.k.a. “creation arguments”), which can be changed using an input.
- A message is a fixed value in execute mode and is stored with the patch. When a message box is clicked, its contents are sent to all objects connected to its outlet. In contrast, number boxes can be altered in execute mode and their values are not saved.

#### ◆ 2.1.4 APPENDIX

A few additional things that can make your work in Pd easier:

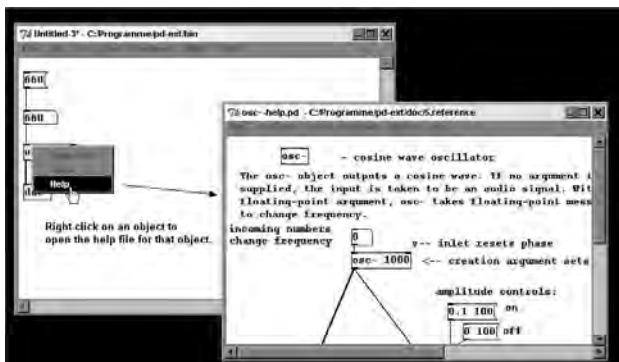
#### ► 2.1.4.1 List of all objects

If you click on the white surface (“Canvas”) with the right mouse button and open the **Help** menu, a list with all Pd objects appears.



#### ► 2.1.4.2 Help file

If you right-click on an object, a pull-down menu opens where you can select the help file for that object for a detailed explanation.



#### ► 2.1.4.3 Duplication

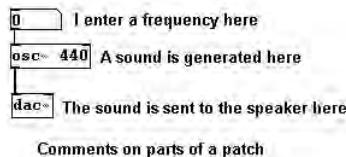
You will soon find it quite helpful to duplicate parts of patches. To do this, select an area so that the selected boxes appear in blue (as described under 2.1.1 in the context of making more space) and go to **Edit** → **Duplicate** or **Ctrl-D**. This duplicates the selected area and the copy appears as a selected area that you can move (click on a selected box, hold, move to desired location with the mouse, release).

#### ► 2.1.4.4 Short cuts

It is much faster and more comfortable to work if you use “keyboard shortcuts”. Many functions that you can select in the pull-down menus are also available as keyboard shortcuts. These key commands appear next to the function in the pull-down menu.

#### ► 2.1.4.5 Comments

Programming can get complicated fast. To help remind you of the meaning of a certain patch, it is recommended that you *add comments* to your patch. Comments can be added under **Put** → **Comment** (or with **Ctrl-5**). Here you can write whatever you like to explain your patch.



If you've understood everything thus far, then you understand the essential fundamentals of Pd's user interface. Now we can get into the structure of programming itself.

### ◆ 2.1.5 FOR THOSE ESPECIALLY INTERESTED: ATOMS

A message for an object has two parts: a method designation (selector) and zero, one, or several values (arguments). For example, if the message is “5”, then the actual message is “float 5” and is comprised of the atoms “float” and “5”. The message “bang” is comprised only of the selector “bang” and contains no arguments. The message “1 2 3 4 5” is actually the message “list 1 2 3 4 5”.

There are three kinds of atoms: a number (programming language = “float”) with a 32-bit value, a symbol, which is a string of letters, or a pointer, which is a kind of address (this will be covered in Chapter 5.2.3).

The message “float 5” is composed of the two type designations symbol and float. The type symbol has a value of “float” (a string) and the type float has a value of “5”.

The selector is always a symbol. Since objects can react differently to different messages, the selector first makes a more precise preliminary determination.

## 2.2 The control level

First we have to work through the basics of the control level in Pd. As already mentioned, Pure Data works only with data, i.e., with numbers (and with the help of letters). (In the examples, however, we will be working with processed sound as soon as possible.)

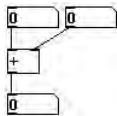
### ◆ 2.2.1 MATHEMATICAL OPERATIONS AND ORDER

#### ► 2.2.1.1 Theory

##### ▷ 2.2.1.1.1 Basic mathematical functions

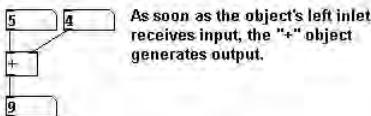
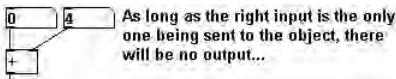
As previously mentioned, computers only work with numbers. Pd works with both numbers and “symbols”, in other words letters. But numbers are of even greater significance; in the first example we saw how important parameters like the pitch or volume of a sound are not determined in Pd using the traditional musical indications like *C4* for a pitch or *pianissimo* for a dynamic, but rather exclusively using numbers. For this reason, we’re going to spend some time learning the basics about how Pd processes numbers in the control level:

You can enter numbers in both number boxes or message boxes. Some objects allow mathematical operations to be performed using these numbers. Create the object “+” and connect number boxes to its right and left inlets as well as one to its outlet:

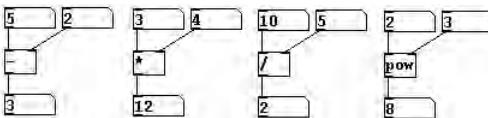


Enter a 4 in the right upper number box (in Execute Mode: click the number box once, type the number, press ENTER) and 5 in the left upper box. The number 9—the sum of 4 and 5—appears in the lower box. The “+” object has two inlets in which we can enter numbers and one outlet in which the result processed by the object (in this case a process of addition) appears.

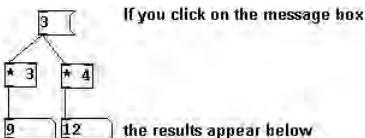
This example illustrates an important rule in Pd: with control objects that have several inputs, you have to enter data into the inlets from right to left. In other words, an object receives input. It will only create output based on this input when it receives input from the far left inlet. (One distinguishes between “cold” inlets, which do not cause an immediate change, and “hot” inlets, which trigger an immediate visible change when data is entered into them.) We will encounter this rule constantly.



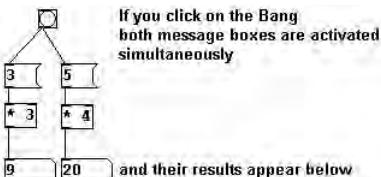
The other basic mathematical operations—subtraction, multiplication, division, and powers—follow the same principle:



If you want to perform several operations using one number, e.g.,  $3 * 3$  and  $3 * 4$  then just connect the number or message box to several inlets (for the sake of simplicity we will use arguments (“\*3” and “\*4”) instead of input for the multipliers. In the previous examples, we used inputs instead of arguments. If we enter an object without an argument, Pd assumes a value of “0” for the argument):

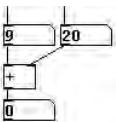


If you want to perform two different calculations at the same time, you have to transform one mouse click into several using a “bang” (**Put → Bang** or **Shift-Ctrl-B**). You can click the bang in Execute Mode.

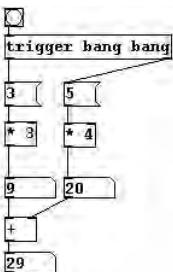


#### ▷ 2.2.1.1.2 Order

If you then want to add these two results, you have to make sure that they enter the “+” object in the correct order, i.e., from right to left.

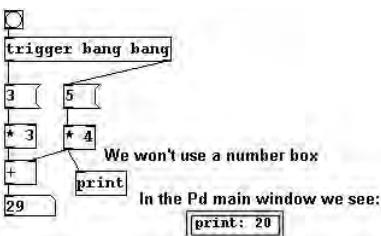


To ensure this, there is what's called a “trigger” object:



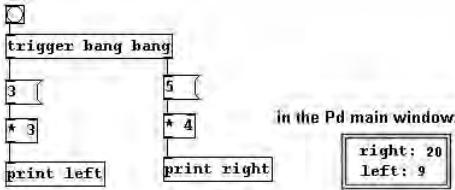
“trigger” can receive as input a bang, a number, a symbol, a pointer or a list (more about pointers and lists later). Once started in this way, “trigger” gives this input or transforms it into a bang as output *from right to left*. The output from a “trigger” object is determined by its arguments (bang, float, symbol, pointer, list). In this case, the arguments are two bangs and two outlets are created (an outlet is created for every argument you enter).

To save space, you can omit the number boxes from the first operations for the results you want to add, simply using the outputs from above directly as inputs below. If you ever want to know what value is being sent, just attach a “print” object to the output.



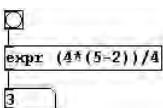
A “print” object’s input will appear in the Pd main window. All errors that occur also appear in this window. For example, if we try to create the non-existent object “zzzg-ghhh”, it will not be created and an error message (“zzzgghhh ... couldn’t create”) appears in the Pd main window.

You can use this window to clarify the way “trigger” works by creating numerous “print” objects and giving them different arguments. The results appear in the Pd main window under each other, i.e. one after the other chronologically (for more on order of operations, cf. 2.2.1.4):



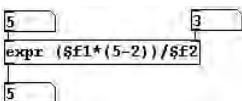
#### ▷ 2.2.1.1.3 Expression

Larger mathematical expressions can be programmed using the “expr” object. The argument in this case is the expression itself (using parentheses where necessary, just like back in math class!):

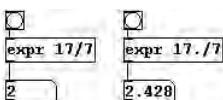


To generate the result, you have to give it a bang.

You could also use “variables”; they are called \$f1, \$f2, \$f3 etc. (counting begins with 1). This creates inlets from left to right in which values for the variables are entered (as always, the output is generated only once the leftmost inlet receives a value. So you have to ensure that all other values have been entered before the leftmost one).



N.B.: If you want an “expr” operation (without input) to generate a ,float number’ (i.e., a decimal value, not a whole number), then you have to include a decimal point in one of the values in the operation (for more on floats, see 2.2.1.4).

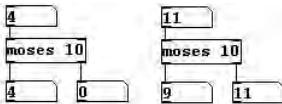


Exponential functions (a.k.a. ,raise to power’ operations) follow this syntax: “expr pow ([Basis], [Exponent])”. For example, to raise to 2 to the 3rd power: “expr pow (2, 3)”.

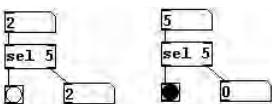
#### ▷ 2.2.1.1.4 Other mathematical operations

“moses”: The input is a number; “moses” decides, by evaluating whether this input is smaller than / larger than/equal to the argument, which outlet will receive it. If you give

“moses” an argument of 10 and give it an input that is smaller than 10, this input comes out of the left outlet. If the input is 10 or greater, it is sent to the right outlet.

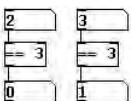


“select” (usually abbreviated to “sel”): Input is a number, output is a bang only when the input is the same as the argument. Any other numbers received as input come out the bottom right outlet.



#### *Relational tests*

“==”: If the left input is the same as the argument or the right input, the output is 1, otherwise 0:



“>=”: If the left input is larger than or equal to the argument or the right input, the output is 1, otherwise 0.

“>”: If the left input is larger than the argument or the right input, the output is 1, otherwise 0.

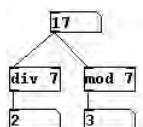
“!=”: If the left input is not equal to the argument or the right input, the output is 1, otherwise 0.

“<”: If the left input is smaller than the argument or the right input, the output is 1, otherwise 0.

“<=”: If the left input is smaller than or equal to the argument or the right input, the output is 1, otherwise 0.

#### *Two more mathematical modules:*

The result of a division operation (the quotient) can be expressed in decimal form ( $17 / 7 = 2.428$ ) or in the form of a ‘remainder’:  $17 / 7 = 2$  remainder 3. A quotient with a ‘remainder’ can be achieved in Pd with “div” and “mod”:



Then there are also other important mathematical operations (for more specific information on these functions, please consult a high school mathematics textbook):

“sin” = Sine

“cos” = Cosine

“tan” = Tangent

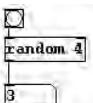
“log” = (natural) Logarithm

“abs” = Absolute value

“sqrt” = Square root

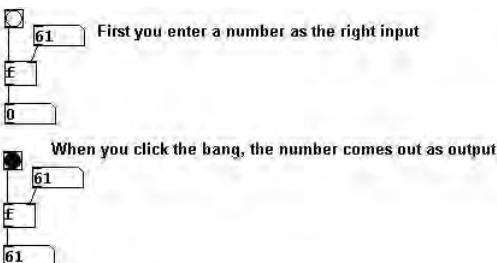
Finally, there is an algorithm (algorithms are mathematical operations that the computer calculates using entered values):

“Random” creates a random number within a given range. The lower limit has a default value of 0, the upper limit is entered as an argument (whole numbers only). The upper limit is exclusive; i.e., if you enter “random 4”, every time the object receives a bang as input it will randomly select an output of 0, 1, 2, or 3.



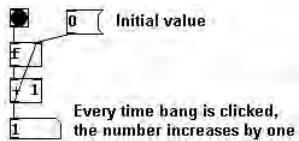
#### ▷ 2.2.1.1.5 *Float and counter*

Another important object in the context of number operations is the “float” object (abbreviated: “f”). This object is used to store numbers. When you enter a number into the right input, it is saved in the object for later use. If you send a bang to the left inlet, the number stored in the object is sent as output (for more on “float”, cf. 2.2.1.4).



You can also send a number directly into the left input. This causes it to be sent as output immediately. The number is also stored in the object for later use and can be resent using a bang.

Often in Pd, you'll want to use a "counter" that counts in whole numbers starting from a given input value. Here's an example:



Explanation:

First you give the "f" object a starting value of "0". The first time you click on the bang in the upper left, the "f" sends a 0 to the "+ 1" object. This object then generates  $0 + 1 = 1$ . This 1 then goes into the right inlet of the "f" object. The next time you send a bang, this 1 is sent as output to the "+ 1" object, which in turn generates a 2.

#### ► 2.2.1.1.6 Summary

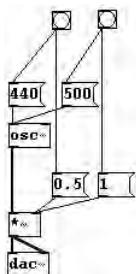
- The objects for mathematical operations demonstrate clearly an important rule in Pd: the inputs for a control object should always be entered from right to left. To ensure this is the case, we often need to employ a "trigger" object, which sends outputs from right to left one after the other.
- A "bang" is like a mouse click, that can be sent or received.
- The "print" object displays in the Pd main window outputs generated when running your patch. Outputs sent one after another in time appear underneath each other in the list; i.e. the output at the bottom of the list is the most recent.

#### ► 2.2.1.2 Applications

Now let's take a look at how to apply these concepts (everything dealing with sound will be explained later):

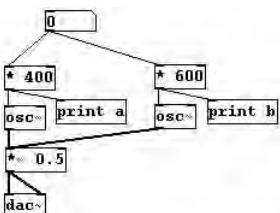
##### ► 2.2.1.2.1 Two frequencies—two volume levels

If you want to switch between two frequencies—a low quiet tone and a high loud one—you could use the following patch. Switch between tones by clicking on their respective bangs:



#### ▷ 2.2.1.2.2 An interval

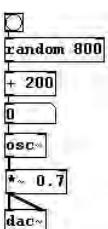
To produce a dyad, you'll need two “osc~” objects. In the following patch, moving the values in the number box up and down will produce a vertical interval (here, a perfect fifth) at various pitches:



Because “print” objects are present, the frequencies of these two tones will be displayed in the Pd main window.

#### ▷ 2.2.1.2.3 Random melody

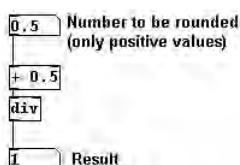
Indeterminacy!



Every bang you send will generate a pitch between 200 and 1000 Hertz—a random melody.

Now a few more examples of mathematical operations:

#### ▷ 2.2.1.2.4 Rounding



#### ▷ 2.2.1.2.5 How long is this score?

A value that composers need to calculate again and again: you've written a piece with 324 quarter notes at a tempo of quarter = 72. How long is the piece in seconds?

```

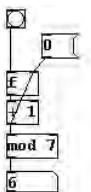
324
72
expr $(f1/f2)*60
270

```

Result: 270 seconds or 4 minutes 30 seconds.

#### ▷ 2.2.1.2.6 Counting in a series

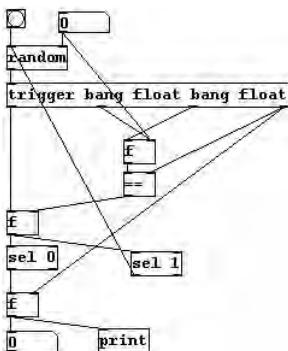
This counter counts only from 0 to 6; after the 6, it starts again at 0.



#### ▷ 2.2.1.2.7 Random without repetitions

If you've understood everything thus far, you should be able to handle the following challenge—but be warned, it's not easy:

Create a patch that generates random numbers where the same number never occurs twice in a row (unlike the normal “random” object). When you've finished, compare your patch to the solution. Good luck!



#### ▷ 2.2.1.2.8 More exercises

- Create two random melodies that run simultaneously.
- Create a patch where two bangs select two different intervals of your choosing (like the two bangs/two frequencies example).
- Use “expr” to calculate exponential functions, e.g.  $y = x^2$ ,  $y = x^{(2+x)}$ ; or  $y = 1 - (2^x)$ .

### ► 2.2.1.3 Appendix

#### ► 2.2.1.3.1 Input for bang

A bang is like a mouse click. You can click it and have it pass on this click; i.e., it can receive a click as input and then in turn sent a click as output. However, this input doesn't have to be a click. The "bang" object converts any control input that it receives into a bang. For example, you could use a number:



#### ► 2.2.1.3.2 How numbers are represented

Numbers with many decimal places cannot be read in their entirety in a normal number box. You can enlarge the number box, however, by right-clicking on it, going to "Properties", entering a larger value for "width", and then clicking on "Ok".

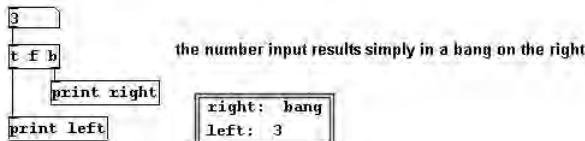
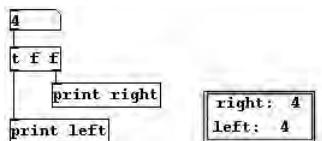
Another important aspect relates to numbers larger than 999999. They are represented in a simplified form, namely as a product (with max. two decimal places) of 1000000. The number 1000000 is represented as "e+006".

1784444 E.g., if you enter "1784444" you get: 1.78e+006

The same applies to numbers smaller than -999999 and for those between 1 and -1 with more than four decimal places.

#### ► 2.2.1.3.3 More on trigger

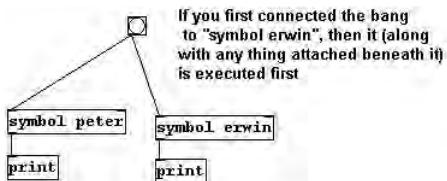
The "trigger" object is capable of distributing not only bangs but also numbers (later we'll learn even more possibilities). It is usually abbreviated as "t" and instead of typing out the arguments "bang" and "float", you can use just "b" and "f":



#### ► 2.2.1.4 For those especially interested

##### ▷ 2.2.1.4.1 About series

By default, objects and connections are (currently) carried out in the sequence (in time) in which they are created:



Of course, this cannot be seen and should be avoided for just that reason!

##### ▷ 2.2.1.4.2 Regarding float

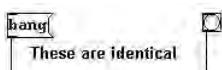
"f" stands for "floating point". In precise terms, this indicates a number that has decimal places and not a whole number. If you want to work only with whole numbers, you can always use "int" (abbrev. "i") instead of "float" in Pd. In contrast to Max/MSP, Pd works with floating points by default.

#### ◆ 2.2.2 DIFFERENT TYPES OF DATA

#### ► 2.2.2.1 Theory

##### ▷ 2.2.2.1.1 Bang—a GUI object

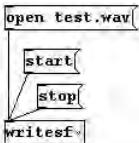
A "bang", like a mouse click, stands for the letter combination b-a-n-g. Letter combinations as symbols are the second form of data (besides numbers) that Pd uses. Some objects recognize certain words and work with their input. Many objects react to the symbol "bang". Since it occurs so frequently, there is a special graphic representation for "bang", a circle that flashes when active (**Put → Bang**). This is called a "GUI" object (GUI = graphical user interface, i.e., a graphic representation of something and/or a graphic that can be changed to produce and send new values).



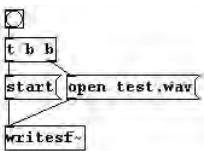
##### ▷ 2.2.2.1.2 Messages

In this context, let's have a look at the "writesf~" object (an audio object will be introduced here due to the fact that symbols are usually used in this context; the object itself will be further explained in the audio chapter). This object saves sound as WAV files. It works like this: first we allocate a file in the message box to which the sound is to be stored in WAV format: "open [file name]". If, for example, the file is to be called "test.

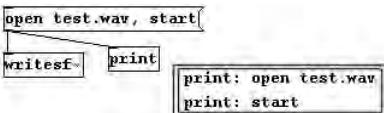
wav”, then you would enter “open test.wav”. Then we use the messages “start” and “stop” to start and stop the recording.



Usually you select a name and then start recording. Order is important, of course—before the “writesf~” object can begin recording, it has to know what the file is called that it is supposed to save to. This could be solved by using “trigger”:



But messages can also be sent one after another by writing them into the same message box, separated by a comma:



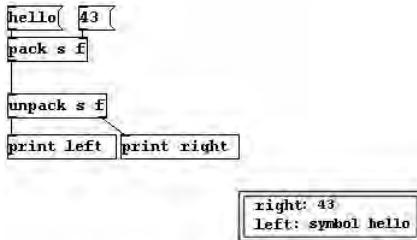
#### ▷ 2.2.2.1.3 Lists

The message “open test.wav” is a connection between two symbols (because it consists of two words separated by a space). This sort of succession of two or more symbols (or numbers) is called a “list”. The “pack” object can create a list from several “elements”. For the arguments, enter indications that specify what kind of elements the list should contain. A number, as with “trigger”, is expressed with “float”\* (or “f”), a symbol with “symbol” (or “s”). If you want to create a list that contains the messages “hello” and “43”, use the “pack” object as shown:



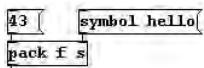
Once again: only when the leftmost inlet receives input is an (accurate) output generated. (If you click on “hello” first without having entered the right input for “pack”, only “list hello 0” will result.) The output for the “pack” object can at this point only be seen with the “print” object. This then displays: “list hello 43”. The elements of this list can, how-

ever, also be revealed using the inverse object (Pd has a lot of inverse objects) “unpack”, which works according to the same principle as “pack”, except that what appears here as output is the input for the “pack” object.

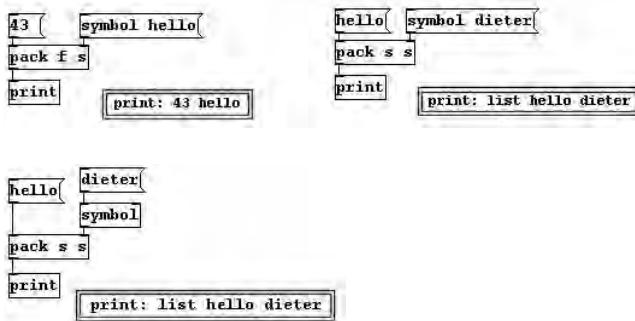


“print” now displays “43” and “symbol hello”. Everything that is not a number is preceded by an indication (called a “selector”) of its data type.

Also worthy of note: if you use a symbol anywhere but in the leftmost input for a “pack” object, it must appear like this:



One problem with “pack s s”: the first input is the only one that doesn’t have to be specifically labeled as a symbol. The second symbol must either be preceded by the word “symbol” in the message box or the message has to be converted using a “symbol” object:

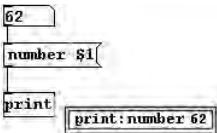


A final point on lists: a list that begins with a number needs not explicitly be labeled a list; if it begins with a symbol, however, the word “list” must be used.

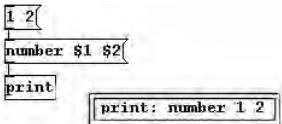
#### ▷ 2.2.2.1.4 Messages with variables

Let’s take a closer look at message boxes:

Variables can be integrated in a message box’s contents. This is done in a similar fashion as with “expr”, but not quite the same: first, the variables are called simply “\$1, \$2”, etc. If you enter a number as input for the message “number \$1”, the output from the message box will be the complete expression with this number.



The use of several variables—e.g., “number \$1 \$2”—does not create a corresponding number of inlets (as it does with “expr”), but instead there remains just one inlet. You need to enter a list of numbers into this:

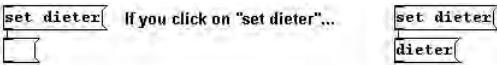


Symbols must be identified as such:



#### ▷ 2.2.2.1.5 *Messages: Set*

You can also completely redefine a message box’s contents by preceding them with the symbol “set”:

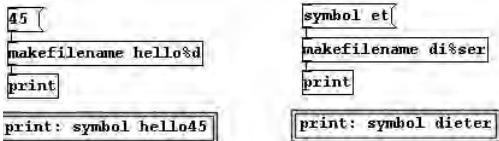


This changes the contents of a message box in execute mode (cf. the last point in 2.1.3). Using a variable could, for example, turn the output of a number box into a message:



#### ▷ 2.2.2.1.6 *Makefilename*

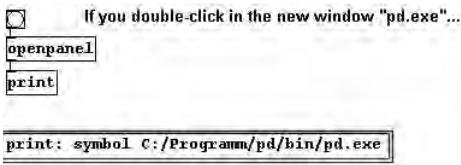
Ordinarily it is not possible to include a variable without a space to separate it. The “makefilename” object, however, makes this possible. Variables that can be included in arguments are “%d” for digits and “%s” for symbols:



#### ▷ 2.2.2.1.7 Openpanel

The object “readsf~” plays an existing sound file, e.g., one that is saved on the hard disk. It needs the message “open [name of sound file]”. “Name of sound file” refers to the place where the file is stored on a data storage device. If you want to use “readsf~” in a patch that is saved in the directory c:/Pd/Pd-patches/ to play a sound file named “hallo.wav” that is also saved in the same directory, you only have to enter “open hallo.wav”. If “hallo.wav” is saved in the directory c:/Pd/, however, you have to enter: “..//hallo.wav” or if it’s saved in c:/Pd/Pd-patches/soundfiles/, then “/soundfiles/hallo.wav”. If it is in c:/soundfiles/: “open ..//soundfiles/hallo.wav”. Or if it is on another drive, e.g., d:/soundfiles, then you have to enter “open d:/soundfiles/hallo.wav”.

These sometimes complicated directory path names can be more easily expressed using “openpanel”. When it receives a bang, it opens a window with the available contents for all of a computer’s drives. When you double-click on a file, “openpanel” enters the entire path for the file (as a symbol) in Pd:



If a patch hasn’t yet been saved, Pd (in Windows) assumes the path pd/bin/ .

#### ▷ 2.2.2.1.8 Simple data storage

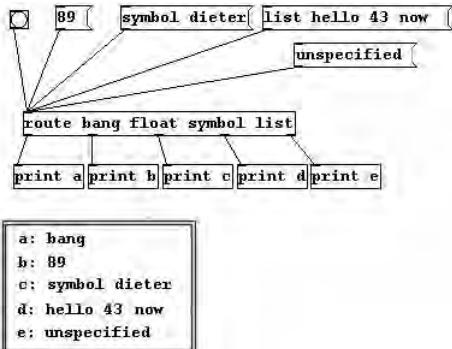
As already explained with the “float” object (2.2.1.1.5), data can be saved within a patch with the objects “float”, “symbol”, and “lister” (but it’s lost when you close the patch). “float” and “lister” are usually abbreviated to “f” and “l”.

The right inlet receives a number, a symbol, or a list that is to be stored in the object. This stored data is sent as output when the object receives a bang in the left inlet.

A number, symbol, or list can also be sent directly into the left inlet; they are then immediately sent out as output (and are also saved in the object itself).

#### ▷ 2.2.2.1.9 Route

An object that can be used to sort various types of data is “route”. It can also allocate the data type (number, symbol, list, bang)...



(Everything that cannot be allocated is sent out the right outlet.)

... as well as order lists according to names you have defined:

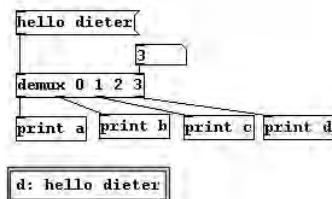


Numbers and symbols cannot be combined here. For example, “route 22 dieter” will not work.

#### ▷ 2.2.2.1.10 Demultiplex

“route” distributes an input to various outputs according to prefix. “demultiplex” (or “demux”, both in Pd-extended version) distributes an input to various outputs according to the input of another inlet. First “demux” receives the numbers of the outlets as an argument, starting with 0: “demux 0 1 2 3”.

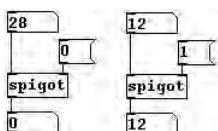
In this example, there are two inlets (“demux” always has only two inlets) and four outlets (one for each of the four arguments). Enter a number in the right inlet that corresponds to the number of an outlet. Now whatever you enter (number, symbol, or list) in the left inlet comes out the third outlet:



Note that Pd often begins counting not with 1, but with 0.

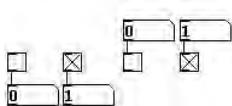
#### ▷ 2.2.2.1.11 Spigot

Another important object is “spigot”. Depending on whether its right input is a 0 or a 1, “spigot” either sends an input through or not—like a gate that is either open or closed.



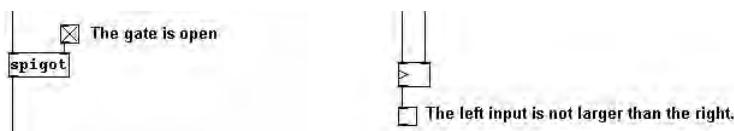
#### ▷ 2.2.2.1.12 Toggle

As you’ve seen with “spigot”, “==”, and other relational tests, 0 and 1 occur frequently in Pd. Due to this frequency—similar to “bang” for a mouse click—there is a graphic object for changing between 0 and 1 called “toggle” (**Put** → **Toggle** or **Shift-Ctrl-T**).



Toggle looks like an on/off switch and can often be thought of as such. But you should always remember that the computer always interprets it as simply a change between 0 and 1.

By attaching a toggle to a “spigot” you can more clearly see if the “gate” is open or closed. Or to see if a relational test delivers a positive or negative result:

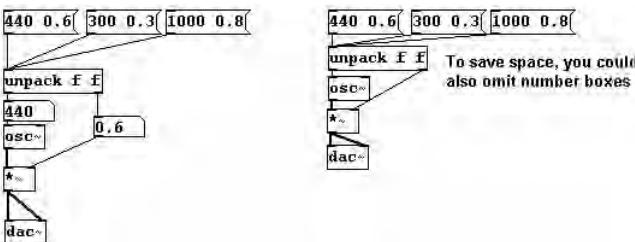


### ► 2.2.2.2 Applications

Let's see how these concepts work in practice:

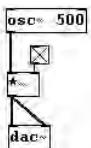
#### ► 2.2.2.2.1 Using lists with pitches and dynamics

Using a list to assign pitches to an oscillator coupled with dynamics:



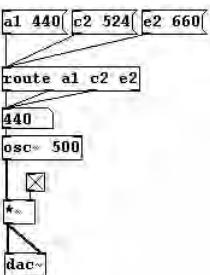
#### ► 2.2.2.2.2 On/off switch

In the first example, we saw that a tone could be turned on and off using “1” and “0”. You could use a toggle for this as shown:



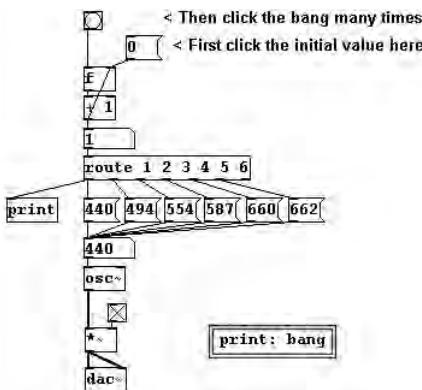
#### ► 2.2.2.2.3 Pitches with names

To assign pitches with (freely chosen) names to an oscillator:



#### ► 2.2.2.2.4 A simple sequence

Here's a counter that sends a particular pitch to the oscillator every time it receives a bang:

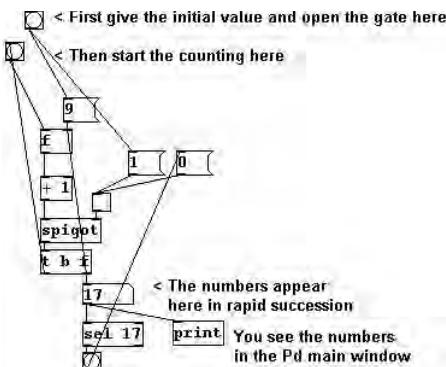


When, instead of a list, “route” receives an input value that is equivalent to one of its arguments, it sends a bang out of the corresponding outlet. In this example, “route” functions as a combination of several selectors (another possibility would be to attach a series of “sel-” objects to the counter: “sel 1”, “sel 2”, etc.).

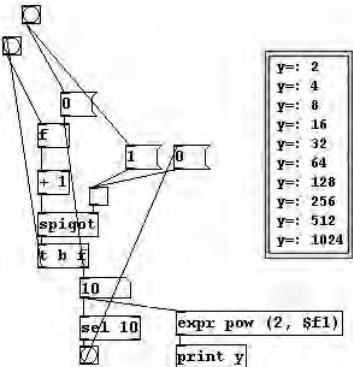
The “route” object’s rightmost outlet doesn’t need to be connected to anything as long as the input always corresponds to the “route” object’s arguments.

#### ▷ 2.2.2.2.5 A limited counter

Here’s a counter that counts upward starting at 10 and stops at 17:



This could be useful to, say, quickly calculate values for a mathematical function within a given range. This example shows the simple quadratic function  $y = 2^x$  for the range of 1 to 10:



For recursions (where an output is fed back as an input) such as these, you have to be very careful to avoid an infinite loop. If we restart this patch after it has already run once without reentering the initial value but instead just opening the gate and starting the calculation, it will start above ten and keep counting forever (as the sel 10 object that stops the calculation will never occur).

#### ► 2.2.2.2.6 More exercises

- Create a sequence of lists with pitches and dynamics.
- Create a patch that allows you to use a list of two numbers, which represent the first and last values in a range for x, to calculate values for y in an equation—e.g., values for the function  $y = 3^x$  from  $x = -2$  to  $x = 4$ .

#### ► 2.2.2.3 Appendix

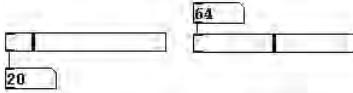
##### ► 2.2.2.3.1 Symbol boxes

Symbol boxes function analogously to number boxes (but are seldom used in Pd). For example, “sel” could also be used with symbols:



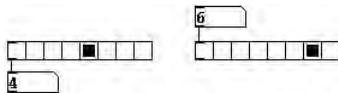
##### ► 2.2.2.3.2 Slider

There are two other GUI objects on the control level: the slider and the radio. The slider (**Put → HSlider** or **VSlider** or their shortcuts) is a graphic representation of a number box. It is, however, restricted to a range (with a default setting of 0 to 127):



#### ► 2.2.2.3.3 Radio

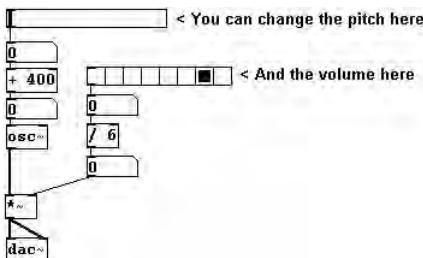
The radio (**Put** → **Hradio** or **Vradio**) is also a graphic representation of a number box, but extremely limited: only a few numbers (by default from 0 to 7) can be sent out, which is accomplished by simply clicking on a box.



Sliders and radios can be horizontal or vertical; this is only a difference in appearance and doesn't affect their function.

#### ► 2.2.2.3.4 Using slider and radio

Various pitches can be selected with a slider and various dynamic levels with a radio:



This creates a visually clear interface for changing a patch's parameters. It is especially helpful for use in live on-stage performance.

#### ► 2.2.2.4 For those especially interested: Other type specifications and more about boxes

A “float” specification can (e.g., with “trigger”) often be expressed with a number instead of with an “f” in Pd (the value of which can sometimes play a role, but not always—e.g., it could be valid with “f” or “pack” but not with “t”):

`t 10 34` is like `t f f`

`20` is like `f`

However, as this is certainly detrimental to clarity, the use of numbers is not recommended.

A few general observations regarding boxes: 1. Strictly speaking, all boxes are objects that can send and receive messages as well as react to these messages according to their (the boxes') characteristics. 2. The connections show which object sends messages to which other objects. If an object's outlet is connected to inlets of several other objects, then all of these objects receive the message. Order is (intentionally) not defined. 3. There are GUI objects that create and send messages based on user interaction. Examples of GUI objects: bang, toggle, slider, and canvas.

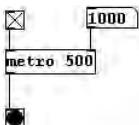
#### ◆ 2.2.3 TIME OPERATIONS

Music, as is commonly known, takes place in time. Therefore, it is essential that an audio programming language has the capability to control the chronological sequence (i.e., that durations/rhythms and sequences of events can be created).

##### ► 2.2.3.1 Theory

###### ▷ 2.2.3.1.1 Metro

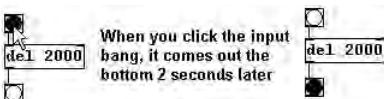
The first basic object for controlling the chronological sequence is called “metro”. As the name implies, this is a metronome. When you turn it on or off (using 1/0 in the left input or with toggle), bangs occur at the regular interval that is determined by the argument or the right input.



The tempo is set in **milliseconds** (ms), which are thousands of a second. If you want to send a bang once per second, enter “metro 1000”, “metro 2000” for a bang every two seconds, “metro 500” for a bang every 1/2 second (equal to quarter note = 120).

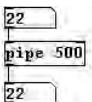
###### ▷ 2.2.3.1.2 Delay

“delay” (“del”) delays an incoming bang by the number of milliseconds in the argument or the right input:

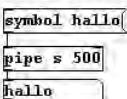


###### ▷ 2.2.3.1.3 Pipe

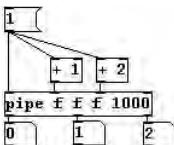
“pipe” achieves the same thing with numbers and symbols as “delay” does with bangs. The duration of the delay is entered as an argument. By default, “pipe” expects a number as input.



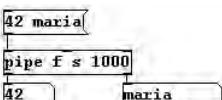
If you want to send a symbol through, this must first be entered as an argument (with "s", as with "route"). Second (or as the right input), you determine the duration:



Also, "pipe"—like "pack"/"unpack"—can have several inlets and outlets:



"pipe" handles lists like "route":

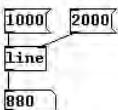


If an input is ,waiting' in a "del" or "pipe" object, it can be deleted before being sent with the message "clear" (pipe) or "stop" (delay):

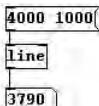


#### ▷ 2.2.3.1.4 Line

With "line", you can create a series of numbers in time. In other words, you can command the program to start counting within a restricted range, the start and end values of which you determine. "line" normally contains no argument. The right input is the duration of the series of numbers (by default 0). The left input is the target value (by default 0; this can be entered differently in the argument).

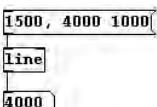


If you enter a new target value on the left, Pd jumps right to this value. This is because the value on the right was automatically reset to 0 and has to be reentered (this is an exception in Pd; usually Pd objects save their “cold” inlets until they are reset). Alternatively, you could enter both values (target value and duration) as a list:



If you click on the message box, nothing happens because “line” has arrived at 4000 and has remained there. If you enter a new target value into the list—e.g., 50—“line” counts from 4000 to 50 (in 1000 milliseconds).

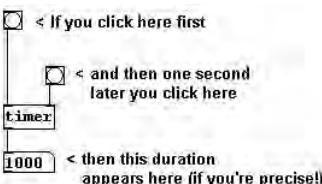
If you want to begin with a particular number and count to another number within a certain time frame, you have to first enter a value in the left input (without having entered anything in the right input). This will cause “line” to jump to this starting value (with one single message box); then you can enter the list. As previously mentioned in 2.2.2.1.2, you can include several messages in a single message box, provided you separate them with commas, like this:



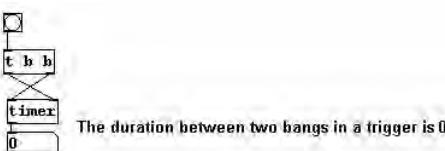
In this example, every time the message box is clicked, “line” counts from 1500 to 4000 in 1000 milliseconds.

#### ▷ 2.2.3.1.5 Timer

“timer” is like a stopwatch. Connect bangs to both inputs. The time measured is always the time between the left bang (which, of course, must be given first) and the right one (in milliseconds):



Here you can see that trigger operations do not result in any time expenditure for the computer, even though they occur one after another:



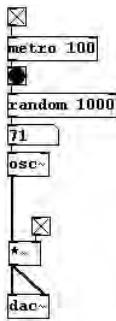
(cf. 2.2.2.2.5)

“timer” is (somewhat unnecessarily) an exception to the Pd rule that inputs must always occur from right to left.

#### ► 2.2.3.2 Applications

##### ▷ 2.2.3.2.1 Automatic random melody

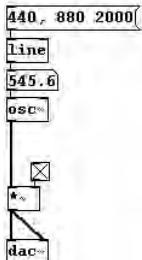
Now we can realize quite complicated musical configurations. For example, a quick random melody that runs automatically:



Try out various metronome speeds with the above example (a number box as right input into “metro”)!

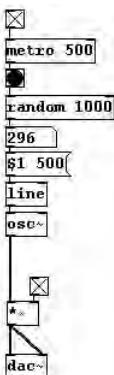
##### ▷ 2.2.3.2.2 Glissando

You can also create a glissando:



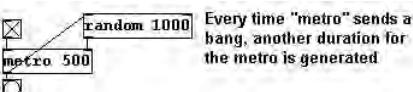
#### ▷ 2.2.3.2.3 *Glissando melody*

Or combine these last two patches to create a random glissando melody:

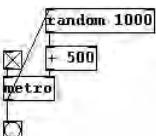


#### ▷ 2.2.3.2.4 *Irregular random rhythms*

You can also create irregular rhythms on the basis of random selection:



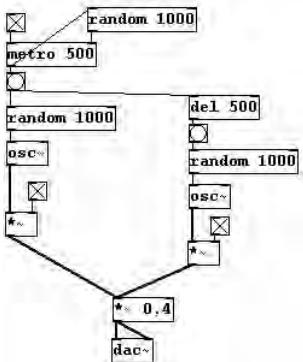
In this example, the “metro” object’s bangs are sent at an interval of between 0 and 999 milliseconds (selected randomly). If you want, say, durations of between 500 and 1500 milliseconds, you just have to use simple addition:



An addition to a mathematical operation such as this one (here “+ 500”) is called an “offset”.

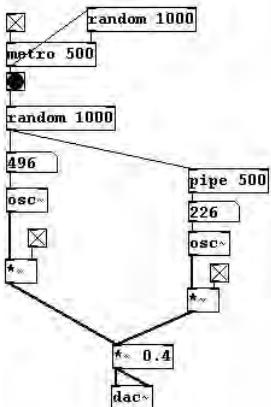
#### ▷ 2.2.3.2.5 Canons

These rhythms can then be connected to the random generator and transferred to another oscillator to make a rhythmical canon:



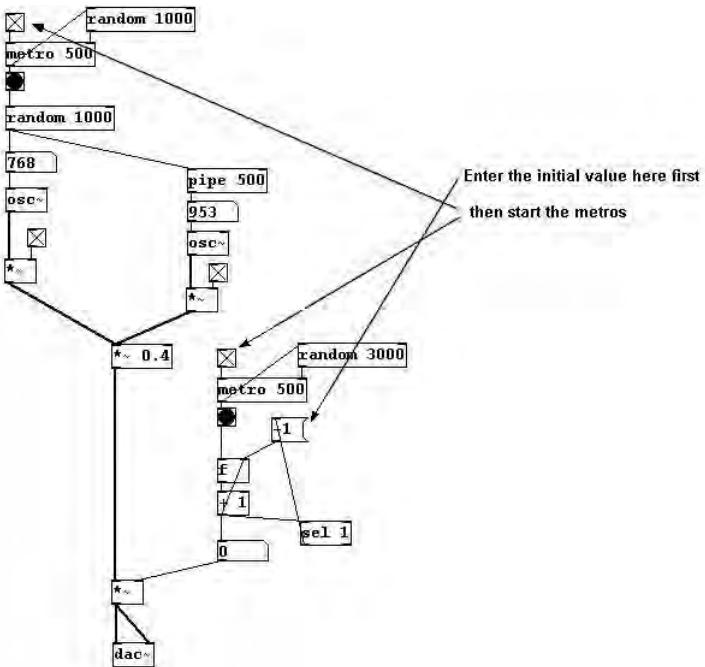
(The “ $*~ 0.4$ ” will be explained later.)

Or you can just make a true canon:



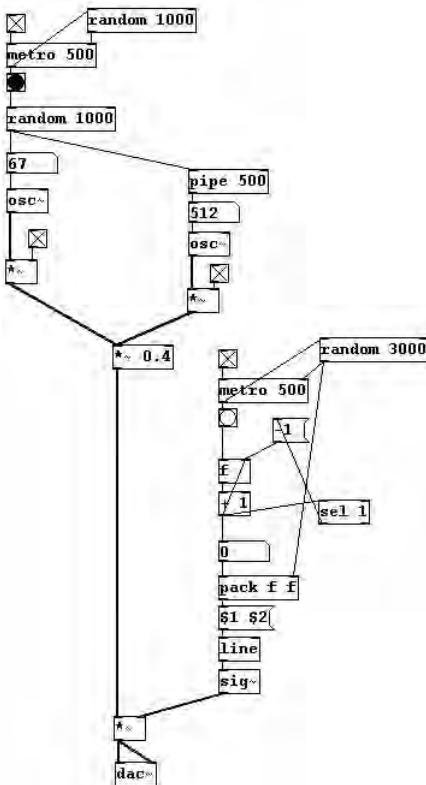
#### ▷ 2.2.3.2.6 Rests

You can also include automated rests:



#### ▷ 2.2.3.2.7 Crescendo/Decrescendo

Or as crescendi and decrescendi (“sig~” will also be explained later):



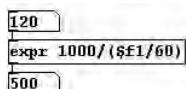
Here again you can see that, for the time being, Pd uses only numbers for calculations. A crescendo is as much a series of numbers as is a glissando. You could also say: a crescendo is a dynamic glissando.

#### ▷ 2.2.3.2.8 Metronome

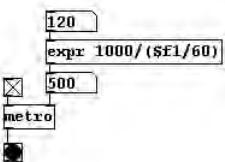
You could build a metronome like this:

First, let's make a working visual model, i.e. such that the metronome signal creates a bang that can be seen. Metronome markings are given as beats per minute, just like in a musical score: quarter note = 60, quarter note = 100 etc.

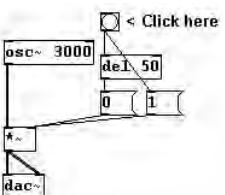
So you have to convert bpm (beats per minute) into milliseconds:



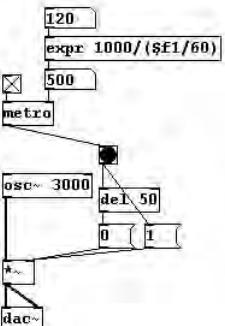
Now use this result as the input for a metronome.



Now we don't want to just see these impulses as a bang, but also hear them. Let's use the sound patch from earlier and set it so that a short tone is heard with each bang. You can create a short tone like this:



All together:



Once these connections are made, the metronome is finished. Later you'll learn how to incorporate an alternative sound signal.

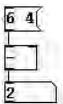
#### ▷ 2.2.3.2.9 More exercises

- Create a random melody that jumps to the next tone twice per second (alternatively: with a glissando).
- Create a metronome with irregular random rhythms (with an adjustable average tempo).
- Create a metronome that beats five times in tempo Quarter = 60 and five times in tempo Quarter = 100.
- Create a random melody that changes every two seconds from a fairly high register to a fairly low one.

### ► 2.2.3.3 Appendix

#### ► 2.2.3.3.1 Distributing lists

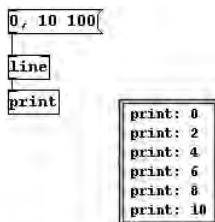
As you saw with “line”, in Pd you can enter a list in the leftmost inlet of an object that has several inlets instead of connecting something to all the object’s inlets (however, there are objects for which this will not work). The elements in the list are then distributed to the inlets from right to left:



#### ► 2.2.3.3.2 Time resolution for control data

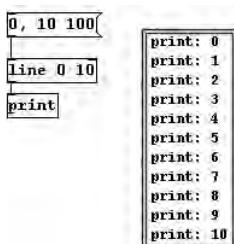
The time resolution for tasks on the control level is in milliseconds.

However, this is often not preset. You can imagine that a calculation in milliseconds requires a lot of processing power (also called CPU\*). For “line”, for example, the preset is that steps occur at intervals of 20 milliseconds:



If you want to count from 0 to 10 in 100 milliseconds, the computer executes a step every 20 milliseconds; this is why the output numbers have gaps.

This interval (in milliseconds) can be adjusted in the “line” object, as a second argument (the first gives the primary target value for the counting process, which is eventually replaced by the input):



---

\* CPU = central processing unit. There are often also other processors, e.g., in the graphics card, where special graphic operations are calculated.

You should be aware that the result will only be “clean” as long as the computer’s processing power is high enough. Otherwise, there will be errors.

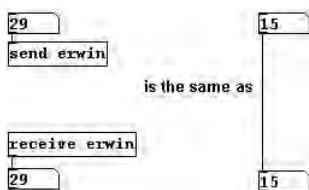
#### ◆ 2.2.4 MISCELLANEOUS

To improve Pd’s “handling”, there are several additional options.

##### ► 2.2.4.1 Sending and receiving

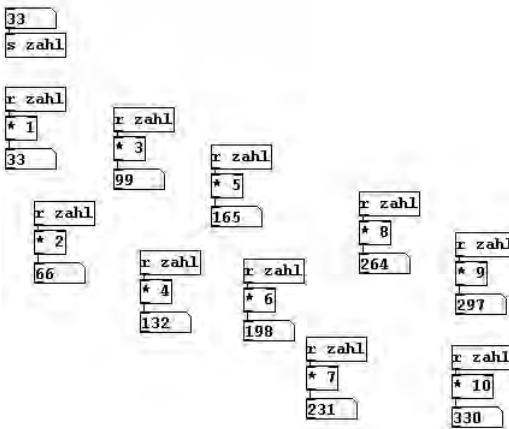
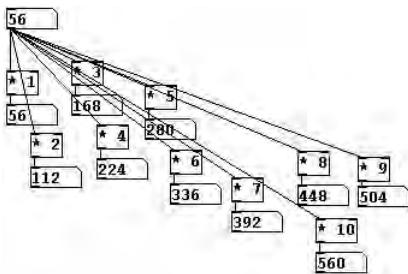
###### ▷ 2.2.4.1.1 Send/Receive

To avoid needing to connect all boxes with ‘cables,’ it is possible to use the objects “send” and “receive” to send and receive things.



The argument for a “send” object can be any name. A “receive” object having the same name as its argument receives input from the “send” object and sends it further.

“send”/“receive” (or “s” and “r”) are practical when you need data in the form of a number to be sent to many different locations (though this may make the patch more difficult to understand).



Such freely chosen names (we'll return to this later) must always be entered without spaces between letters; individual numbers are not allowed.

```

33
s_zahlen This name works
r_zahlen
33

```

```

0
s_5 but just a number as a name doesn't work
r_5
0

```

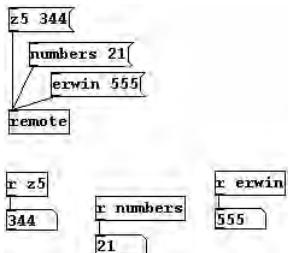
```

0
s_z5 Numbers combined with letters works again
r_z5
0

```

#### ▷ 2.2.4.1.2 Sending with lists

If you have different receivers (with different names), you can use “remote” to allocate messages from a central “distribution point” (similar to “route”). You give this object a list whose first element is the name of the receiver and whose second element is the message itself. “remote” is part of Pd-extended.

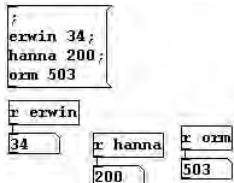


Another possibility is to precede the list with a semicolon in the message box. In this case, you need only click on the message box to send the message.



#### ▷ 2.2.4.1.3 A series of send lists

You can also send many different messages in a message box (with one click):



In message boxes, two punctuation marks have a special importance: commas (a series of many messages) and semicolon (which represent the sender).

N.B.: in message boxes, Pd automatically skips a line after a semicolon. If you write the following:

```
;hello 466
```

and then copy the entry (**Ctrl-D**) or close and reopen the patch, you will see this:

```
;  
hello 466
```

This is also the case for a comment (2.1.4.5.).

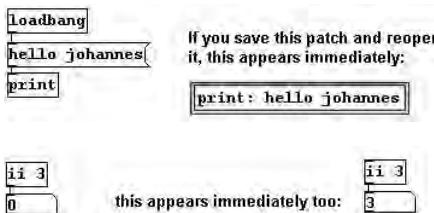
#### ► 2.2.4.1.4 Value

Another way to send a value is to determine it globally, i.e., for the entire patch. This is achieved with “value”. Give any name as the argument and enter the value as input. In other parts of the patch, the value can be retrieved with the same object and argument using a bang:



#### ► 2.2.4.2 Loadbang

You might like to keep several values for the next time you open a patch or there might be a particular value that you want to receive a bang right at the start. To achieve either of these, you could use the “loadbang” object (sends a bang as soon as a patch is opened) and “init” (abbreviation: “ii”), which sends a number or symbol (or a list of numbers, of symbols, or of numbers and symbols) as output (Pd-extended).



#### ► 2.2.4.3 GUI options

GUI stands for “graphical user interface” and refers to all special graphic objects in Pd. GUI objects are number and symbol boxes, bang, toggle, slider, radio, canvas, as well as array and VU, both of which will be explained later. All GUI objects have extended functions. To access these, right-click on the object and choose “Properties” from the pull-down menu with the left mouse button.



The following can be set here:

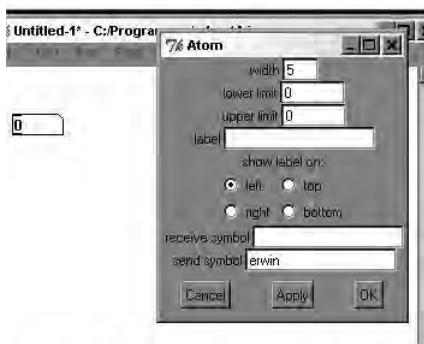
#### ▷ 2.2.4.3.1 Number and symbol box

*width* refers to the width of the box. This can be useful in conjunction with very large numbers or numbers with many decimal places.

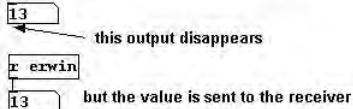
Use the *lower and upper limit* to adjust the range of values that the box will accept (for example, for numbers you might use a range of 0 to 1000).

With *label* you can assign the box a name for it to display (including the position where the name will appear).

With *receive/send*, you can build a send/receive function into the box. For example, if you enter “post1” in ,send,’ and have a receiver somewhere called “post1”, this receiver will receive all entries made in the number box. The same is true for the “receive” function.



If I name the number box's internal send function



As you can see in the graphic, the inlet and outlets disappear when the internal send or receive functions are activated.

The changes take effect when you click on “apply” or on “ok”.

#### ▷ 2.2.4.3.2 Bang

*size* refers to the changeable size (in pixels).

*intrprt/hold* indicates how long the bang lights up (in milliseconds).

*init* means that the value (in this case, a bang) will be sent as output as soon as the patch is opened (as with ,loadbang’).

The *send symbol / receive symbol* is like the internal “send-” / “receive” function in a number box.

With *name* you can create a “label”, as with number boxes. The position is determined with x and y values. In addition, you can define the font style and size as well as the colors for the background, foreground, and the name. First choose the element you want to change, ...



... then select a color from the given color options:



Under “compose color” you can also generate your own color.

“backgd” refers to the background, i.e., the entire area of the bang. “front” refers to the foreground, i.e., the color that briefly lights up when the bang is active (due to an input or by a mouse click).

All changes take effect when you click “apply” or “ok”.

#### ▷ 2.2.4.3.3 Toggle

This works analogously to a bang, except for *value*: by default, ‘toggle’ alternates between 0 and 1. Here, you can enter another value for the 0 (the 1 is always a 1).

#### ▷ 2.2.4.3.4 Slider

*width*: width in pixels

*height*: height in pixels

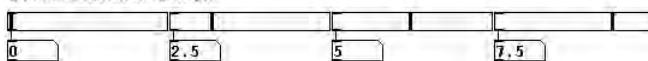
*bottom*: value of the slider when it's all the way down

*top*: value of the slider when it's all the way up

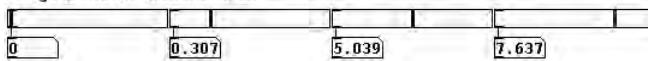
*lin*: Either linear or logarithmic addition/subtraction within the range. If you click on it, “log” appears in the field. The current setting is always the one that you can currently read.

N.B.: With “log”, ‘0’ cannot be chosen as a benchmark figure.

A linear slider from 0 to 10:



A logarithmic slider from 0 to 10:



*init:* The lower value in the range is sent as output when the patch is opened.

*steady on click:* The slider is moved by moving the mouse with the button held. If you click on “steady on click”, then “jump on click” appears, which causes the slider to move immediately to wherever you click within the slider GUI object.

The rest works as with ,bang.’

#### ▷ 2.2.4.3.5 Radio

With ,radio,’ everything works the same way as for all the others except for *number*: the number of boxes.

#### ▷ 2.2.4.3.6 Canvas

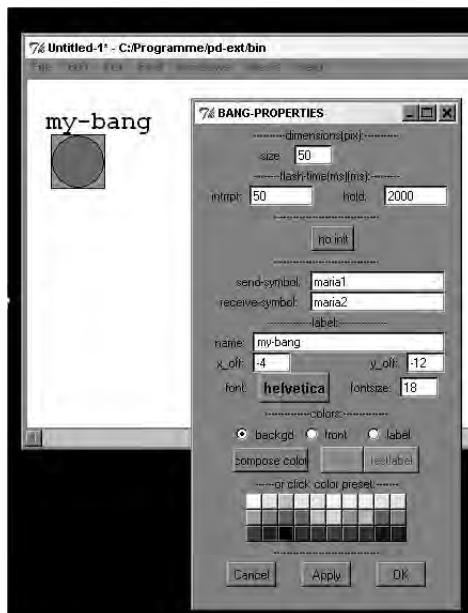
At the end of 2.1.2. you learned that the white surface on which objects are placed is called the “canvas”. We could also add other colored surfaces (**Put → Canvas**). They have no function apart from being colored surfaces.

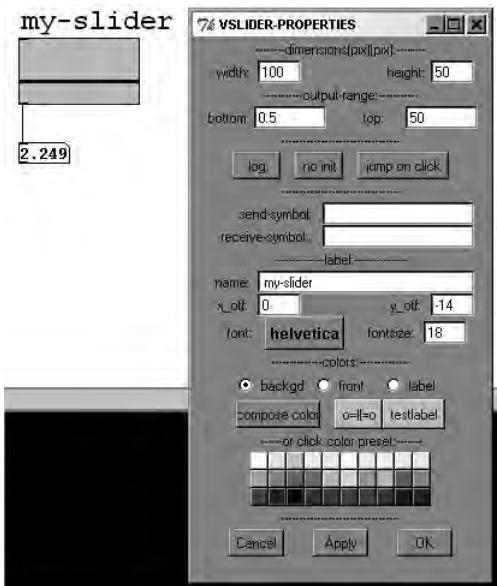
The surface contains a blue square in the upper left—that is the actual object. The entire surface is a product of this object, its output so to speak. It covers all the elements that were there before you created it and lies underneath everything you create thereafter.



The properties are the same as for the other GUI objects.

#### ▷ 2.2.4.3.7 Examples of altered GUI objects



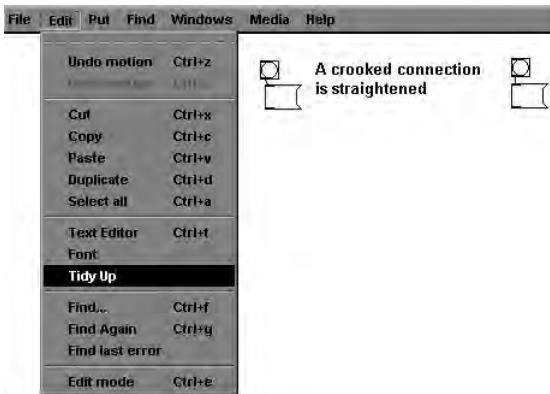


▷ **2.2.4.3.8 Change font size**

You can change the font setting for all boxes under **Edit → Font**.

▷ **2.2.4.3.9 Tidy up**

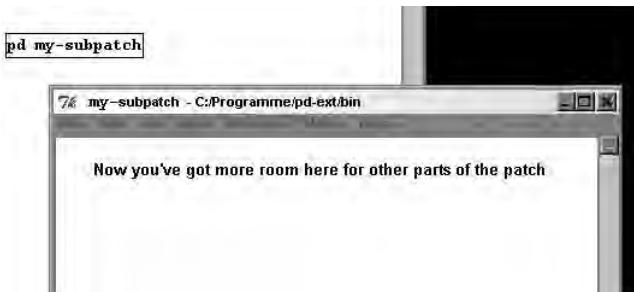
To straighten out slanted cable connections, you can select them and then go to **Edit → Tidy up**. This often doesn't work, however.



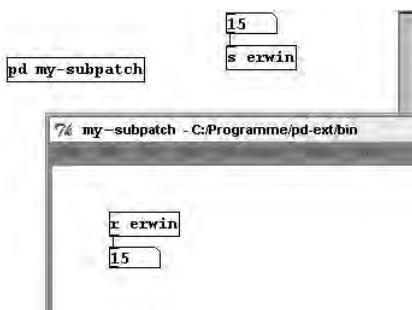
#### ► 2.2.4.4 Subpatches

##### ► 2.2.4.4.1 Space

In the course of programming, you will probably find that you run out of room at some point. For this reason, you can store parts of your patch in what are called “Subpatches”. If you create a “pd” object and enter a name (without spaces) as its argument—e.g., “pd my-subpatch”—a new window opens. (Once you’ve closed this window, you can reopen it in execute mode by clicking on the object “pd my-subpatch” once.) Now you have room for new parts of your patch.

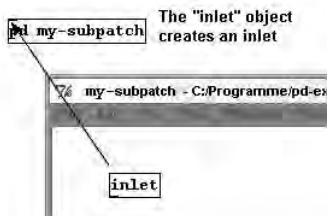


There are two ways to connect this subpatch to the first window in your patch: “send” and “receive” work both within the same window and between different ones:

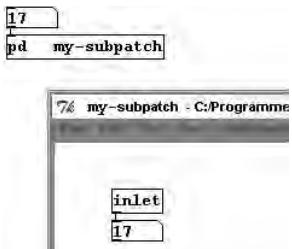


Here you can also see how you can switch between execute mode and edit mode independently in different windows.

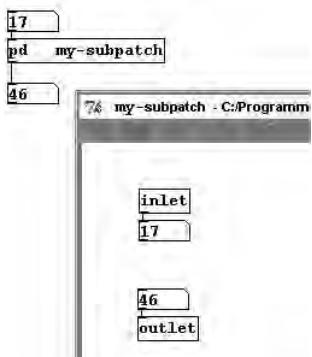
The other way to create a connection is by using the inlet and outlet objects. If you create an “inlet” object in the subpatch, the object for the subpatch (which is located in the first window and is called “pd my-subpatch”) now has a visible inlet.



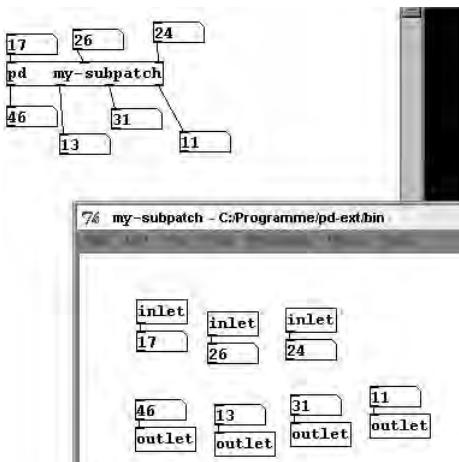
If you enter a number as input in the main window, this will appear in the subpatch.



The "outlet" object works in a similar way:



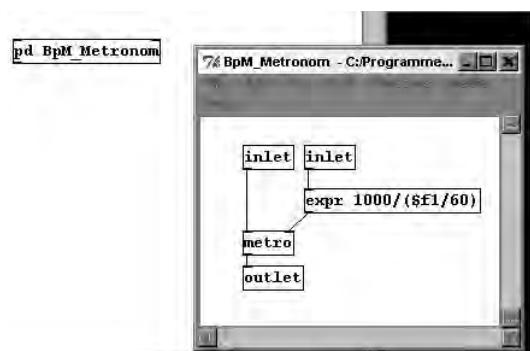
Several "inlet" or "outlet" objects are placed next to one another in the subpatch analogously to their arrangement in the subpatch object in the main window:

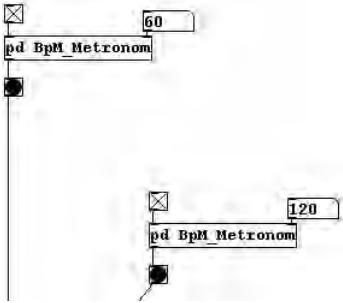


When you close the subpatch window, the subpatch remains active (i.e., still executes processing tasks) as long as the main patch window is open.

#### ▷ 2.2.4.4.2 Modularization

Subpatches solve not only spatial limitations, but also help you structure your patch more clearly. Parts of a patch that complete a certain task can be given their own subpatch, so that this little ‘machine’ is always available. A part like this is called a “module”. Here’s a sample module: a metronome that allows you to enter beats per minute instead of milliseconds:





Names you choose yourself cannot contain spaces. Instead you can use hyphens or underscores, as in this example.

## Chapter 3

### Audio

#### 3.1 Basics

From here on out, large patches are pre-assembled in additional files (<http://www.kreidler-net.de/pd/patches/patches.zip>).

Many of the functions described in Chapter 2 will not be used in the rest of the text—e.g., the “send” and “receive” objects—although they are certainly often used in practice. The patches you’ll encounter here have been reduced to the bare essentials. However, when you actually use the techniques presented in a composition or performance, it will be necessary to store them as subpatches and connect them with “inlets”/“outlets”, etc. You can see one example of this here: 3.4.2.4.

##### ◆ 3.1.1 PITCH

Let’s return to our first example. You heard a tone with a frequency of 440 Hertz (later with other frequencies, other pitches). You could turn it on and off—i.e., adjust the dynamic from loud to inaudibly quiet.

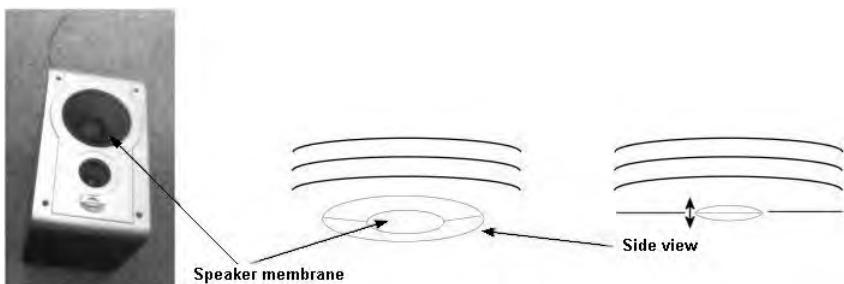
About pitch: Pd works with two kinds of pitches—Hertz or MIDI numbers. The traditional signs: A, B-flat, G-sharp, etc. are not used at all in Pd. Instead all chromatic pitches have MIDI numbers: A4 is the number 69, B-flat4 is 70, etc. The other way to describe pitch in Pd is in Hertz. To understand this, we need to understand a bit about musical acoustics.

##### ► 3.1.1.1 Theory

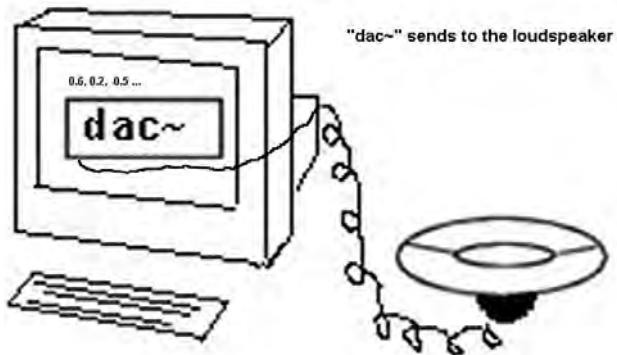
###### ▷ 3.1.1.1 *Controlling speakers digitally*

Sound is air in vibration. Traditional instruments are used to vibrate air at specific frequencies. You can do this, e.g., with strings (violin), lips (trumpet), or membranes (timpani). We even have a membrane in our ears—the eardrum—that vibrates sympathetically with vibrations in the air. Our brain transforms these vibrations into a different form, which is what we would call sound.

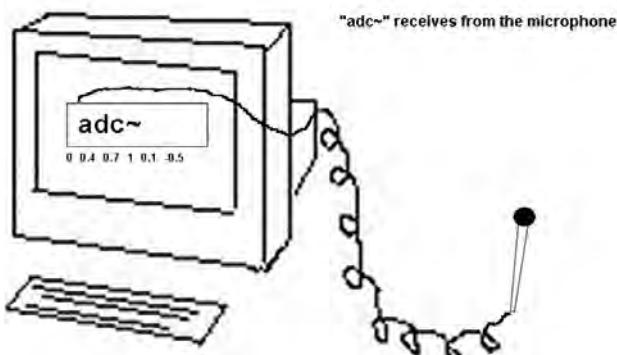
In electronic music, we use speakers to generate sound. These also have a membrane (or several) that vibrates back and forth, which causes the air to vibrate.



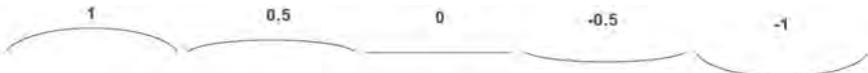
The vibrations of this membrane are controlled by the computer. In Pd, the “dac~” object (digital audio converter) handles this. Here’s how it works: sound is a physical phenomenon—vibrations in the air to be precise—and, as such, it is analog. **Computers, however, work only with numbers**, which means they are digital. The “dac~” object turns numbers into sound by converting numbers into fluctuations of electrical current that—once amplified—cause the speaker membrane(s) to vibrate accordingly.



The reverse of this process would be to connect a microphone to a computer. A microphone also has a membrane that responds to vibrations in the air and converts these vibrations into fluctuations in electric current, which it sends to the computer where they are then converted into numbers. In Pd, this input can be received with the “adc~” object.



Let's go back to speakers for a moment. A speaker's membrane can move back and forth. The outermost position (the most convex) is understood by the computer as position 1. The innermost position (the most concave) is position -1. When the membrane is precisely in the middle, as when at rest, this is position 0. All other positions are values of between -1 and 1.



In reality, these movements are so small and so fast that they almost cannot be observed with the naked eye.

#### ▷ 3.1.1.2 Waves

Let's imagine that a membrane moves from one extreme limit to the next (most convex, most concave) at a constant tempo:



Let's mark the individual stages:

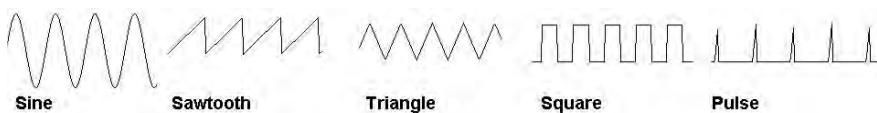


In an abstracted form with membrane position on the y-axis and time on the x-axis, we could represent such motion like this:



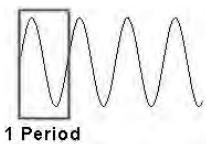
In physics terminology, this is called a *wave*. Here you can clearly see the *waveform*—a triangle.

There are different waveforms for different kinds of membrane movement. Their names reflect their visual resemblance:

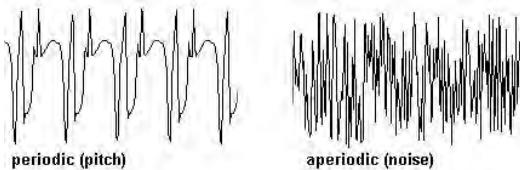


The Pd object “`osc~`” creates a sine wave.

One important thing to remember with regard to waveforms: they repeat constantly without changing their motion characteristics. Vibrations that exhibit this quality are said to be *periodic*. A period is one complete cycle of a vibration that constantly repeats.

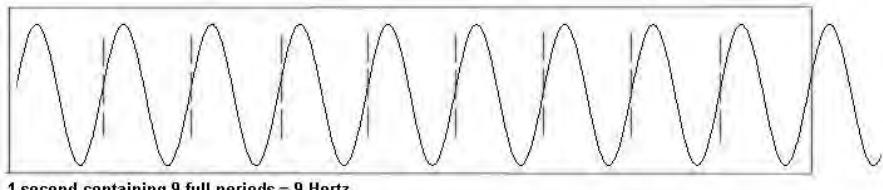


What makes periodic vibrations special is that we hear them as clear tones with definite pitch. In contrast, noises are aperiodic vibrations.



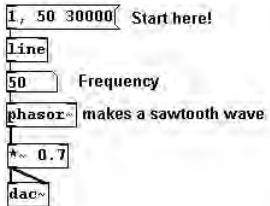
#### ▷ 3.1.1.3 Measurement

Let's first discuss periodic vibrations. It is possible to simply count the number of periods in a second. This number is a vibration's frequency and is measured in "Hertz" (Hz); frequency in this context always means how often something repeats in one second (expressed mathematically: 1/second).



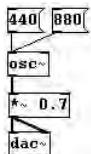
A tone's frequency determines its pitch. A440 (also called A4, the standard pitch that orchestras use for tuning) means that the air vibrates periodically at a rate of 440 times per second; C5 vibrates 523 times per second; the low G on a cello vibrates about 100 times per second.

Here you can already see: the slower the frequency, the lower the pitch appears to our ear. In fact, humans—depending on age—hear pitches between 20 Hz and 15000 Hz. Children can hear up to 20000 Hz; elderly people can often only hear up to 10000 Hz. Dogs and bats can hear well over 20000 Hz. This range is referred to as ultrasonic. In contrast to this is the infrasonic range, which is lower than the bottom of the audible threshold—i.e., between 0 and 20 Hz. This range is perceived by us as rhythm. You can use Pd to experience this for yourself with the following experiment:

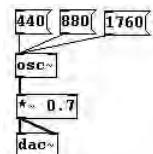


You hear a rhythm of clicks (that's the sound of a sawtooth wave) that gradually gets faster. After a certain speed (over 20 clicks per second), our perception 'shifts gears' and begins to hear a low pitch. For the air (and for the computer), this is still a "rhythm". But for the human ear (ca. 20 Hz) it's a pitch! The faster this rhythm becomes, the higher the pitch we hear.

Another defining characteristic of the human ear is that it hears pitches *logarithmically*. This means when a given frequency is doubled, we perceive this as an octave leap. If you change from A4 (440 Hz) to its double (880 Hz), you hear A5, which is exactly one octave higher:

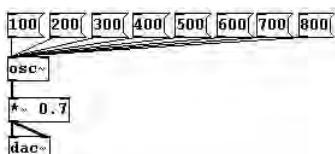


If we want to hear an octave above 880 Hz, we have to double it again.  $880 + 880 = 1760$ :



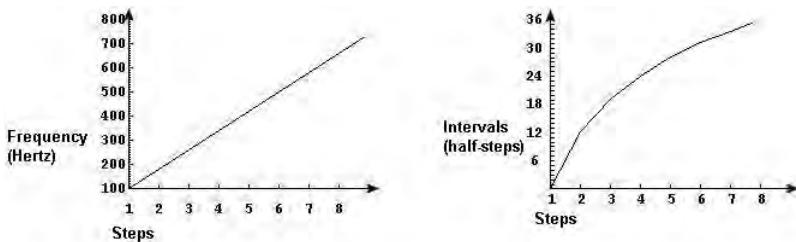
Just to be clear: from 30 Hz to 60 Hz, we hear an octave but from 1030 Hz to 1060 Hz, we hear just a small step. In fact, the jump from 10000 Hz to 20000 Hz is only an octave!

Another important concept: let's add the same amount to a fundamental frequency, say 100 Hz—which is roughly the frequency of the open G-string on a cello—to which we'll add 100 Hz successively:



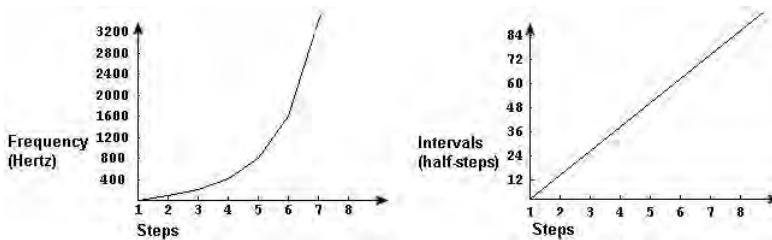
You hear an octave from 100 to 200, a fifth from 200 to 300, a fourth from 300 to 400, etc. In mathematics, this is an additive process in which the same amount is added each

time. Our ears, however, perceive that this amount gets smaller and smaller with every step:



The graph on the left shows the mathematical function—a linear function. The right side shows what we hear—a logarithmic function.

If you want to hear a linear progression—i.e., a process by which the same interval is added, for example the octave—the mathematical function has to be exponential:



The conversion from linear to logarithmic progressions in Pd is accomplished by using MIDI numbers and frequencies. MIDI numbers reflect the way we hear in that the intervals we hear correspond to an equivalent interval in MIDI numbers: one whole number per half-step. You can convert entries in frequencies and MIDI numbers in Pd:

```
440  Frequency
fton "fton" = "frequency to midi"
69  MIDI-No.
```

```
69  MIDI-No.
mtof "mtof" = "midi to frequency"
440  Frequency
```

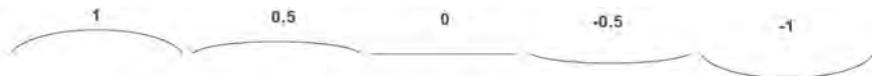
A small table of MIDI numbers, frequencies, and their traditional names:

Frequency (Hz)	MIDI	Note name
65.4	36	C2
100	43	G2
130.8	48	C3
261.6	60	C4
277.1	61	C#4
293.6	62	D4
311.1	63	D#4
329.6	64	E4
349.2	65	F4
370	66	F#4
392	67	G4
415.3	68	G#4
440	69	A4
466.1	70	A#4
493.8	71	B4
523.3	72	C5
1000	83	B5
4186	108	C8

N.B.: Oscillators like “osc~” or “phasor~” have to receive their input in Hertz.

#### ▷ 3.1.1.4 Sample rate

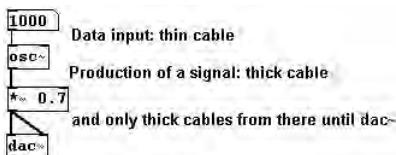
One has to remember that for Pd, sound is only numbers. Positions of a speaker’s membrane are numbers between -1 and 1.



Objects like “osc~” generate a very fast sequence of numbers between -1 and 1 that is sent to the speaker by the “dac~” object. To be specific, 44100 numbers per second are generated and sent. The loudspeaker makes 44100 tiny movements between -1 and 1 within one second. This number, 44100, is called the *sample rate*.

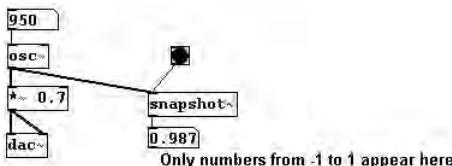
Every sound in Pd is produced using numbers between -1 and 1 at a rate of 44100 numbers per second (sample rate). A single individual number is called a *sample*.

All Pd objects that generate or process data at this speed have a tilde “~” in the object box. These objects are connected to each other with thick cables. We call these series of numbers *signals*.

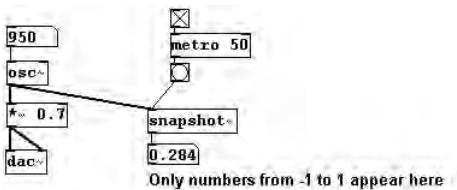


Whenever you want, you can give the “osc~” object a new frequency as input. The cable for this connection is thin, because the input is not in constant transmission. The “osc~” object’s outlet, however, is *constantly* sending signals, i.e., numbers between -1 and 1, 44100 per second (*per second* means: *Hertz*).

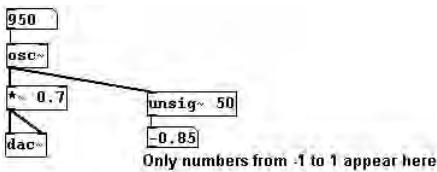
You cannot connect a number box to the “osc~” object’s outlet if you want to see the numbers. Number boxes can only be used for control connections, not signal connections. Signal connections are too fast: you wouldn’t be able to see 44100 different numbers per second. You can, however, show selected numbers from a signal with the “snapshot~” object. As inputs, it receives the sound signal and a bang that, when clicked, displays the current number when clicked. To see this number, connect either a number box or a “print” object to its output:



If you want a constant stream of these numbers, you could attach a (fast) metronome:



In Pd-extended, you could also use “`unsig~`”, which automatically connects a metronome. Enter the metronome value as the argument:



You can also use “`sig~`” to convert numbers on the control level into numbers on the signal level. You enter a value once into its inlet that is sent out its outlet 44100 times per second.

#### ► 3.1.1.5 Samples—milliseconds

As with frequency (and with amplitude as discussed in the next chapter), there are *two* different units in Pd for measuring time: samples and milliseconds. Samples are usually used for counting signals while milliseconds are used for control data.

Converting duration in milliseconds to duration in samples:

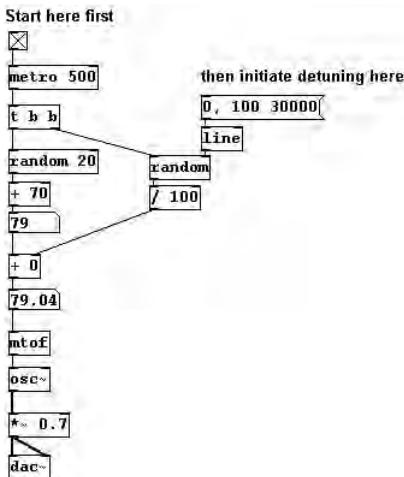


#### ► 3.1.1.2 Applications

##### ► 3.1.1.2.1 Tempered—Random

Random MIDI values are gradually offset. (Transition from equal tempered tuning to random tuning):

[patches/3-1-1-2-1-random-offset.pd](#)



#### ► 3.1.1.2.2 More exercises

- Create a glissando that we hear as linear and one that we hear as logarithmic from C3 to C6.
- Create a quarter-tone scale.

#### ► 3.1.1.3 Appendix

##### ► 3.1.1.3.1 Nyquist Theorem

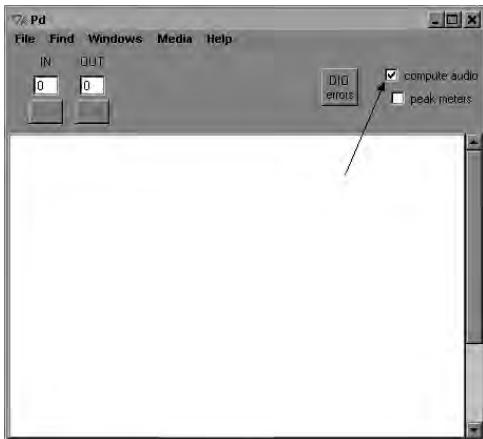
The number 44100 was chosen for a good reason. As previously mentioned, humans can hear up to 20000 Hertz at most. In 1928, US physicist Harry Nyquist (1889-1976) proposed a theory stating that a frequency of at least twice the signal frequency was necessary to accurately represent a sound signal digitally (“Nyquist-Shannon sampling theorem”). Concretely, this means that one needs the maximum and minimum values for each period to accurately represent a waveform’s basic shape, i.e., two points per period:



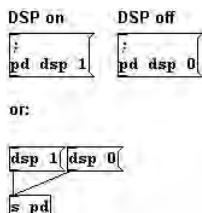
For a wave with 20000 Hz, which equals 20000 periods per second, we need at least 40000 points per second to accurately represent it. To ensure that the entire spectrum of sounds audible to humans was included, a sample rate of 44100 was chosen for audio CDs. This means that waves of up to 22050 Hz could be captured. For computers, a wide selection of frequency bandwidths exist, all the way down to 8000 Hertz for system sounds. High-quality audio recordings work with sample rates of 48000 Hz (48 kHz = kiloHertz, where kilo = thousand), 96 kHz, or even 192 kHz.

#### ► 3.1.1.3.2 DSP

It has become clear that simultaneous processing of numerous signals is very taxing on the computer. Imagine working with 100 “osc~” objects. Each one generates 44100 numbers per second and these have to be synchronized with each other. That's why Pd offers you the option of turning off DSP (digital signal processing) in the main window. This will spare your processor unnecessary work.



You can also send this as a command; the recipient “pd” is in this case the program itself:



With regard to computer music, the faster the processor, the higher the performance.

Pd lightens its workload by working with samples in blocks rather than individually. This greatly improves performance. The standard block size is 64 samples, but this setting can be changed. More on this at 3.8.1.1.

#### ► 3.1.1.4 For especially interested

##### ► 3.1.1.4.1 da- / ad- conversion

It was previously stated that “dac~” sends the numbers generated by Pd to the speaker membrane (3.1.1.1), but this is of course a bit oversimplified. Strictly speaking, the computer's sound card converts the numbers into an electrical current with variable voltage (“digital-analog-conversion” or “da-conversion”); the membrane position is in

turn determined by the amount of voltage. Going the other way, membrane fluctuations in a microphone are converted into a variable current, which is then digitized by the computer's sound card.

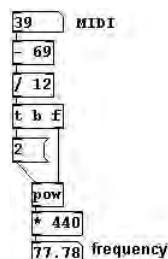
▷ **3.1.1.4.2 Sound waves**

Sound waves, in contrast to water waves, are longitudinal. Longitudinal waves, also called compression waves, are characterized by the fact that they vibrate along their direction of movement. (Transverse waves, on the other hand, vibrate along an axis perpendicular to the direction of movement.) For further explanation, please consult a high school physics textbook.

▷ **3.1.1.4.3 Converting MIDI numbers into frequencies**

The “mtof” object converts MIDI numbers to frequencies. The formula for this calculation is:

$$f = 440 \cdot 2^{(m-69)/12}$$



In one expression:

```
39 [MIDI]  
expr 440*(pow (2, ((\$f1-69)/12)))  
77.78 [frequency]
```

To calculate the frequency of a pitch in equal temperament that is a certain distance away from a given frequency, use this formula:

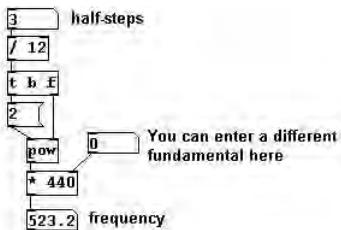
$$f = g \cdot 2^{\frac{a}{12}}$$

'f' is the frequency you want to know, 'g' the frequency of the given pitch, 'a' the interval in half-steps.

For instance, if you want to calculate the frequency of C5 and know that A4 has a frequency of 440 Hz:

$$f = 440 \cdot 2^{\frac{3}{12}} = 523.2$$

In Pd:



In one expression:

```
3
expr $f2*(pow (2, ($f1/12)))
523.2
```

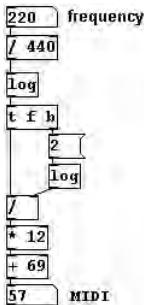
For the inverse operation—converting a frequency into MIDI—the formula is:

$$m = 69 + 12 \cdot \log_2(f/440)$$

However, in Pd we have only the natural logarithm based on Euler's number (the mathematic constant 'e'); so we need this formula as well:

$$\log_a(b) = \frac{\log_e(b)}{\log_e(a)}$$

Programmed in Pd:



In one expression:

```
220 frequency
expr (((log($f1/440))/(log(2)))*12)+69
57 MIDI
```

#### ▷ 3.1.1.4.4 Noise periodicity

We've covered the fact that noises are not periodic. You could, however, imagine a noise that lasts 10 seconds and then repeats precisely as before. Such a noise would theoretically have a periodic frequency of 0.1 Hz. So a noise can be more precisely defined as a

sound that is aperiodic or has a period of less than 20 Hz. Furthermore, one could also say that the frequencies of noise may have a common fundamental tone that is lower than 20 Hz.

Many exciting experiments have been conducted in the field of acoustics, for example involving the Doppler effect or calculating the length of sound waves. Please consult leading acoustics textbooks for more information.

#### ◆ 3.1.2 VOLUME

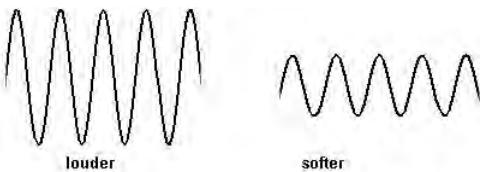
##### ► 3.1.2.1 Theory

###### ▷ 3.1.2.1.1 Measurement

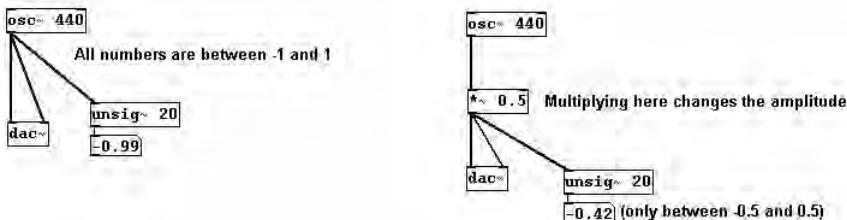
The next parameter of a sound we'll look at is its volume. Traditionally, volume in music is notated using dynamic markings like pianissimo, piano, mezzoforte, etc. Their use is subjective and variable depending on the instrument. In physics, which is what Pd uses as a model for this parameter, volume is represented with objective values in deciBel or root mean square values. Both units are comparable to MIDI numbers and frequencies for pitches. DeciBel (dB) reflect what we hear, where an ‘octave’ in volume corresponds to 6 dB. The scale ranges from 0 to 130 dB—where a value of between 15 and 20 dB is absolute silence and anything over 120 dB is capable of causing serious hearing damage. After 130 dB we perceive sound only as pain. Root mean square values (rms), like frequencies, do not correspond to what we hear, but are logarithmic values between 0 and 1, where 0 corresponds to 0 dB and 1 to 100 dB. rms refers to the geometric mean calculated from a series of amplitude values. These numbers are first squared, then the average is taken (by adding all values and dividing by the number of elements), and then the square root of this average is taken. The rms value for an audio signal is first calculated using a portion of the audio signal that lasts specific duration; for a heavily fluctuating signal like a pitch frequency, it gives you an idea of the average signal amplitude. The following objects can be used to convert from one to the other in Pd:



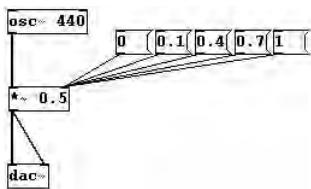
The volume of a vibration is determined by its amplitude, which is the degree to which the membrane is displaced outwards or inwards with respect to the neutral position at rest (the zero position). The greater the membrane's movement, the louder we perceive a sound to be. A representation on an axis looks like this:



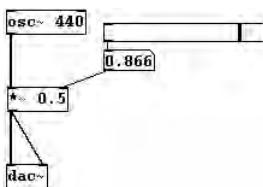
It cannot be emphasized enough: until this is sent to the speaker with the help of the “dac~” object, Pd works only with numbers. If a sound is quiet, this means that the numbers do not span the full range from -1 to 1, but are instead confined to a more restricted range around the zero position, say, between -0.5 and 0.5. This can be accomplished in a patch by multiplying the numbers generated by the “osc~” object by a certain factor:



You can use this method to set the volume to any level from absolute silence to as loud as possible (which depends on the speakers and the amplifier you’re using, of course).

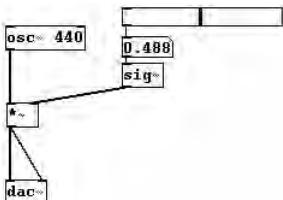


You could also attach a slider using HSlider (**Put** → **HSlider**) and setting its range from 0 to 1 (cf. Chapter 2.2.2.3.2 and 2.2.4.3.4):



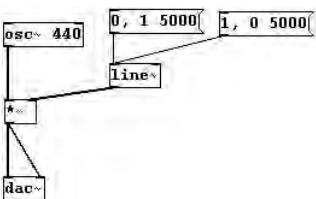
However, moving this slider quickly will result in disruptive sounds. This is because a signal (calculated in samples) clashes with control processing (calculated in milliseconds). If this is only a matter of a few numbers as previously with the factors 0, 0.1, 0.4, 0.7, and 1, this is irrelevant, but beyond a certain speed this can play a significant role.

To avoid this problem, you have to replace the control connection with a signal pendant. Use the “sig~” object to convert:

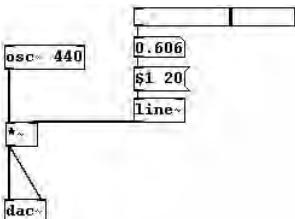


This ensures that the numbers generated by the oscillator (44100 numbers/sec) and those generated by the factor (“sig~” converts them into exactly 44100 numbers/sec) are synchronized. N.B.: if a signal is attached to the “\*~” object’s right inlet, the object must not have an argument: if you were to enter an argument (like 0.5, as used previously) the object would assume that its right input was control data.

To create a crescendo or a decrescendo, you have to use “line~”:

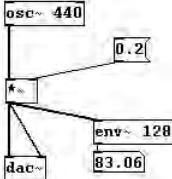


Here’s an elegant way to use a slider as a volume regulator (“Potentiometer”):

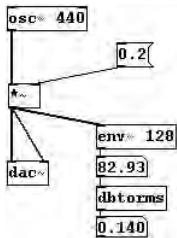


This executes a small crescendo/decrescendo between every step. Filling in steps with intermediate values in this way is called “interpolation” (as already seen with pitches in Chapter 2.2.3.2.3).

You can calculate the volume of a given sound using “env~”, which gives the volume in dB as output. You must always define a span of time in which this average value is to be calculated; its argument is given in samples (this number is usually a power of 2):

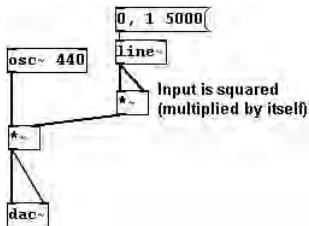


The conversion into rms ...



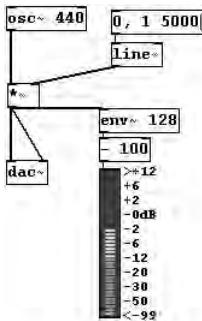
... makes it clear that factors between 0 and 1 are not to be confused with rms values between 0 and 1.

As already mentioned, humans' aural perceptions of volume and pitch do not correspond with the measurements in physics (as observed in the paired diagrams for pitches by frequency and interval). A simple trick for creating a more linear crescendo or decrescendo is to square the values:



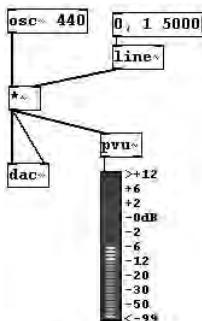
One should try out all the various possibilities, however. In the end, the way that the volume increases or decreases is a compositional decision. What exactly constitutes a "volume octave" cannot be objectified in the same way as pitch.

There is a GUI-object in Pd for visualizing amplitude: the VU meter (**Put** → **VU**). It takes a dB value as input. However, it works like a traditional mixing board: 100 dB is shown as 0 dB and deviations above or below this are shown in the positive or negative range, respectively. You have to take this into account when entering the input. Simply subtract from the "env~" object's output:



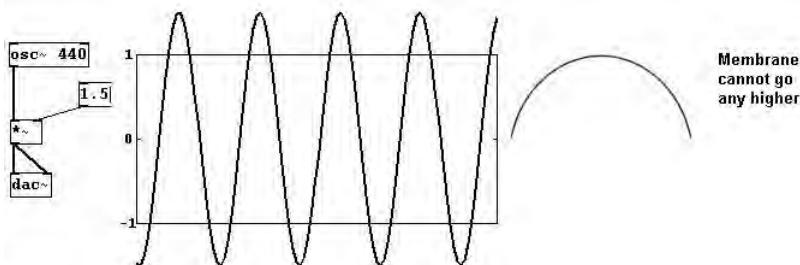
Then the VU shows changes in volume graphically. (VU is short for “volume”).

In Pd-extended, you can also use the “pvu~” object for the VU meter conversion:

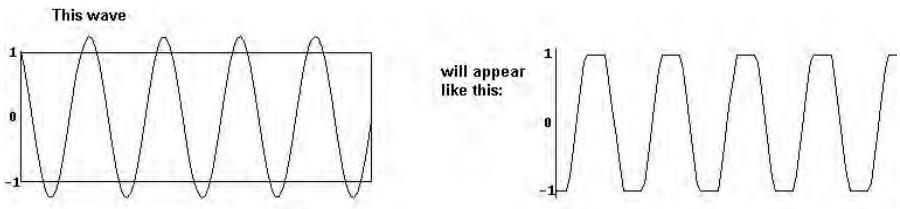


#### ▷ 3.1.2.1.2 Problems

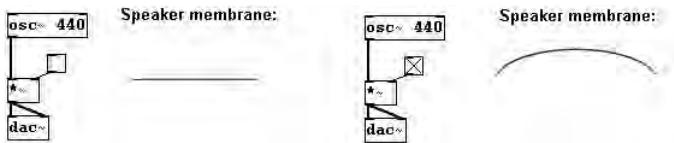
Another important thing: amplitudes above 1 and below -1 will be ‘clipped’. If “dac~” sends the speaker a value outside the range of 1 to -1, the membrane simply stays at the furthest extreme.



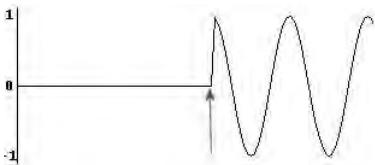
Increasing the volume of a sound to the point of ‘clipping’ results in an effect called *overdrive*.



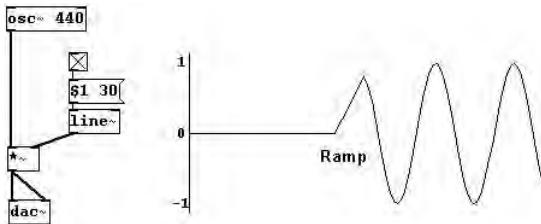
Another problem occurs when the speaker membrane has to span a large interval suddenly (e.g., when you turn on a sound); the result is a “click”:



This is especially noticeable when the sound itself exhibits very smooth membrane movement, as with a sine tone. The “jolt” is easy to see in this illustration:

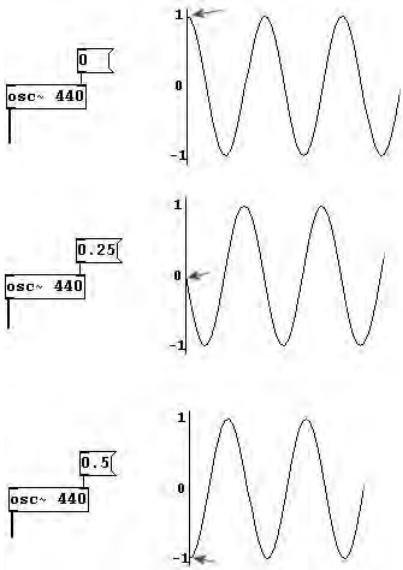


A “jolt” is usually a movement that is faster than 30 ms. To avoid this click, therefore, you need to build what’s called a “ramp”, i.e., a very fast crescendo at the beginning and end:



#### ▷ 3.1.2.1.3 Phase

In Pd, you can also set membrane position for a sound wave where it should begin (or where it should jump to). This is called the *phase* of a wave. You can set the phase in Pd in the right inlet of the “osc~” object with numbers between 0 and 1:

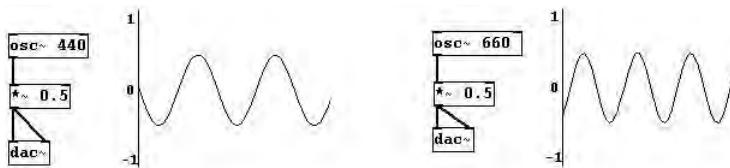


A wave's entire period is encompassed by the range from 0 to 1. However, it is often spoken of in terms of degrees, where the entire period has 360 degrees. One speaks, for example, of a "90 degree phase shift". In Pd, the input for the phase would be 0.25.

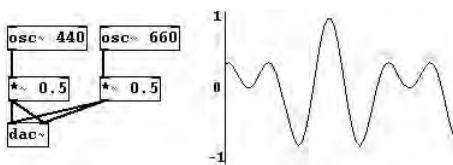
A phase shift doesn't have much effect on what we hear. We'll return to this concept later, however.

#### ▷ 3.1.2.1.4 Sound waves are additive

Let's say you have these two oscillators:

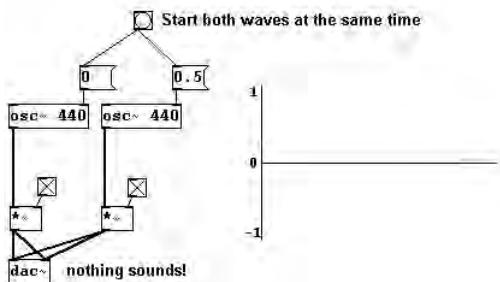


... and you connect them to "dac~". You'd get this:

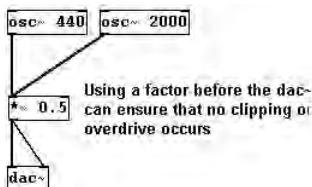


Due to the multiplicative factor, the individual waves only go from -0.5 to 0.5. Taken together, however, they cover a range from -1 to 1 and have a more complex form. This

is because sound waves are *additive*. Simply stated: all vibrations occur in the same air. This additive quality also entails cancellations. Opposed waves, where one is “moving backwards” while the other is “moving forwards” cancel one another out. This is what happens when vibrations that have the same frequency are 180 degrees out of phase:



When many sound sources are involved, we usually have to multiply the total sound by a suitable factor to avoid exceeding the limits of 1 and -1:



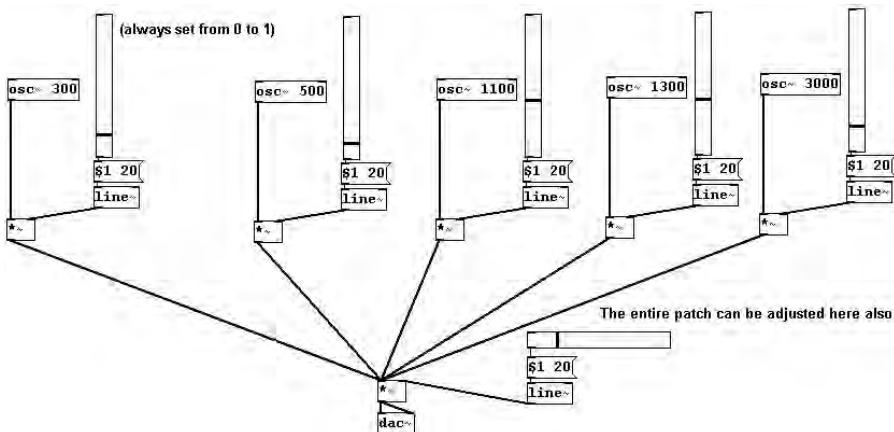
In this case, both oscillators are simply attached to a multiplication object. This automatically adds them (whenever several signals are given to an object as input, they are first added, then processed according to the object) before carrying out the multiplication.

## ► 3.1.2.2 Applications

### ► 3.1.2.2.1 Chord

To create a chord with variable volume for every tone in the chord:

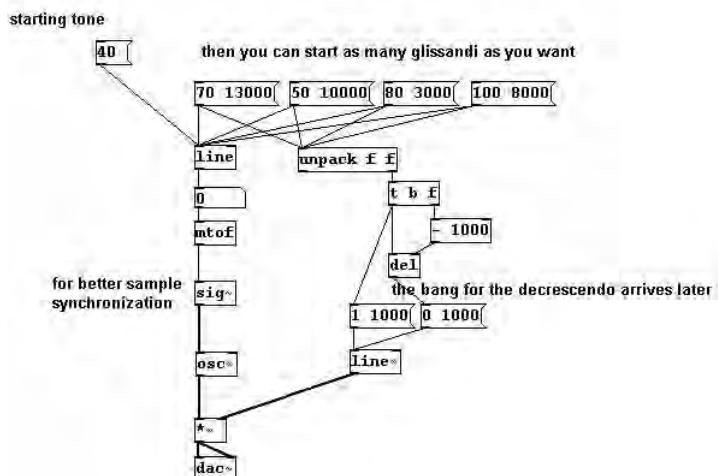
3-1-2-2-1-chord.pd



### ► 3.1.2.2.2 Glissandi

Glissandi that fade in and out smoothly at the beginning and end:

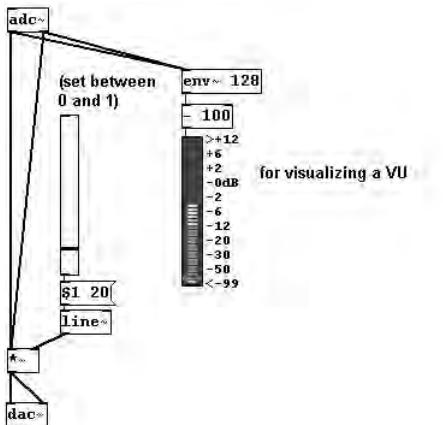
patches/3-1-2-2-2-glissandi-dim.pd



▷ **3.1.2.2.3 Processing adc-input**

Say something into a microphone and play it back at a changed volume:

patches/3-1-2-2-3-edit-input.pd

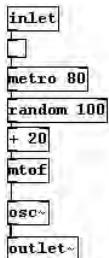


▷ **3.1.2.2.4 Oscillator concert**

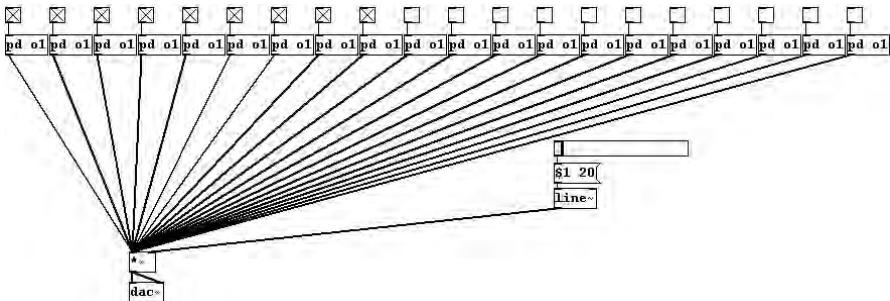
Let's get 'symphonic': why not use 20 oscillators at once?

patches/3-1-2-2-4-oscillatorconcert1.pd

First make the subpatch "o1":



...make multiple copies...

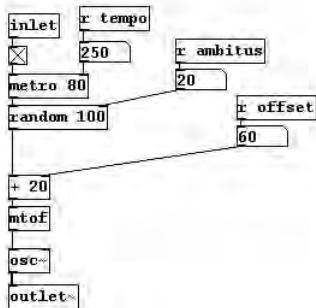


...then turn them all on!

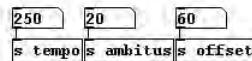
Of course, the parameters for each oscillator can be adjusted—and you've really got something to play with:

[patches/3-1-2-2-4-oscillatorconcert2.pd](#)

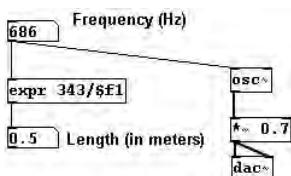
in every subpatch:



in the main patch:



The speed of sound at 20 degrees Celsius (68 degrees Fahrenheit) is about 343 meters per second. You can calculate the length of a period in space and then check the result immediately...



...by moving your head half a meter back and forth while listening to a frequency of 686 Hertz: you can clearly hear the wave's peak and its trough.

#### ► 3.1.2.2.5 More exercises

- Create (random) glissando chords that also have random volume changes for each individual tone.
- Create a patch in which the volume from a microphone input controls an oscillator's pitch (then use several, each with a different offset)!

#### ► 3.1.2.3 Appendix

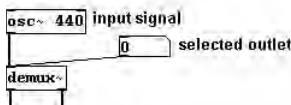
##### ► 3.1.2.3.1 Other tilde objects

Several of the objects covered in Chapter 2 also have a version with a tilde. They work the same way, except that they work with signals instead of control data:

**mathematic operations:**



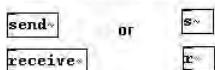
**demultiplex:**



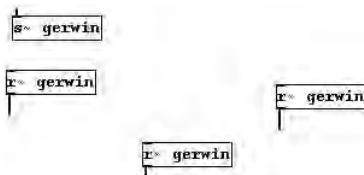
**signals in subpatches:**



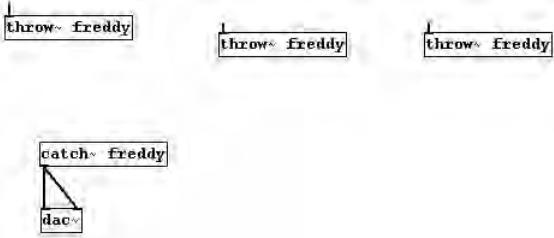
**sending/receiving signals:**



You can use “send~” with as many “receive~” objects as you like; however, you can only use one “send~” object:



You could also channel many different signals to one central location (for example, to the “dac~”) by using “throw~” and “catch~”:



► **3.1.2.3.2 Bit depth**

Bit depth is also an important concept in Pd. The computer's processor only works with binary code, i.e., with 0 and 1. The bit number shows how many places are used for zeroes or ones. If you have only two places, you could make  $2^2$ , that is, 4 different combinations:

```
0 0
0 1
1 1
1 0
```

The more places there are, the more detailed something can be processed. For Pd, which uses numbers to calculate frequencies, amplitude, etc., this means that the numbers can be processed more precisely, i.e., more decimal places can be used. Pd normally works with 32 bit. 32 bit means  $2^{32} = 131,072$  possible values for each sample.

► **3.1.2.4 For those especially interested**

► **3.1.2.4.1 Sound pressure vs. sound intensity**

Volume and more importantly increments of volume are—in both objective and subjective terms—heavily influenced by factors like architectural characteristics of the room, age of the listener, etc. There is no single, precise form of measurement for volume, though there are theories of sound pressure and sound intensity. For more information, it is strongly recommended that you consult a book about acoustics.

► **3.1.2.4.2 Control data vs. signals**

You may have noticed that for significant parts of sound production in Pd two different units are used: frequency and MIDI numbers for pitch, root mean square and deciBel for amplitude, and milliseconds and samples for time.

For the last of these, the example of using “line~” to create a crescendo/decrescendo given under section 3.1.2.1.1. should be explained further:

If you were to use a “line” object (without a tilde) for this, it would likely result in undesired popping or clipping sounds. This would require two different units of time

measurement to be combined; the problem is they are not synchronized. The different numeric intervals will likely not match up, which would cause irregularities in the form of short delays or even popping sounds to occur.

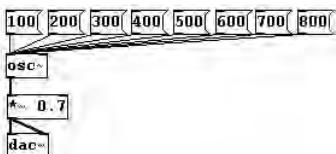
As described in Chapter 2.2.3.3.2, “line” generates a value every 20 milliseconds. That means it may not coincide with the samples. Though a new sample comes every 0.02 milliseconds, a “line” value may not coincide with a more or less simultaneous sample, which could lead to complications. A “line~” object (with tilde), however, generates a signal with 44100 values per second. These 44100 values are generated at precisely the same time as any other tilde object; they are always synchronized. The computer always processes 44100 samples per second synchronously regardless of their position in the patch.

## 3.2 Additive Synthesis

### ◆ 3.2.1 THEORY

#### ► 3.2.1.1 The harmonic series

The additive series of frequencies (i.e., the series that results from simply adding the same Hertz value repeatedly), which results in a string of intervals of decreasing size, is called the harmonic series:

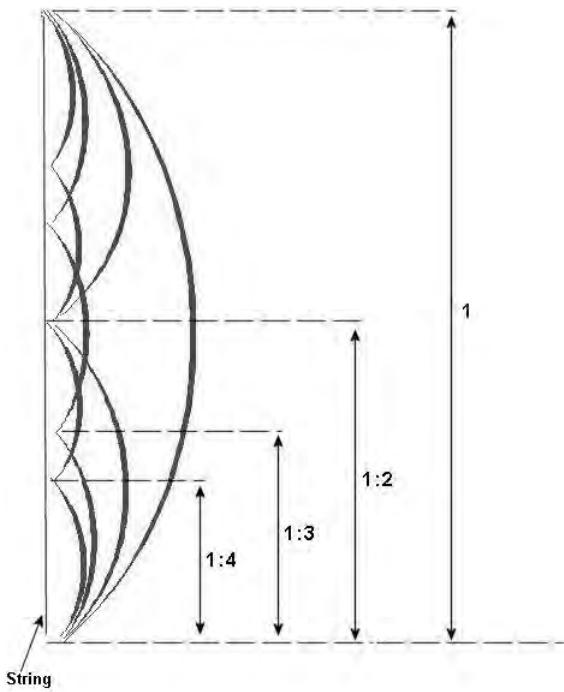


You can also derive the series by repeating an experiment devised by Pythagoras (ca. 570-510 BCE) in which a string is divided into various proportions:



The ratios describe the length of the two parts of the string in relation to each other.

When a string is bowed, it doesn't just vibrate as a whole, but also in every whole number proportion:

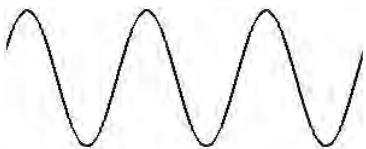


Here the ratios describe the length of the vibrating section in relation to the length of the entire string.

All of these partial vibrations (called ‘partials’ or ‘harmonics’) result in sound as well, so every sound made on a string is in fact already a chord!

The special thing about this chord is that all of its pitches melt together, at least when their relative volumes decrease as the pitches get higher. Every natural sound has overtones. Due to characteristics inherent to the human ear, we hear all of these pitches as just one tone.

In contrast, the upper partials themselves (i.e., the partials above the fundamental) do not have any overtones. An isolated sound without overtones does not exist in nature, but such a thing can be created using electronic means. These are called sine tones, a name that stems from the shape of their waveform:



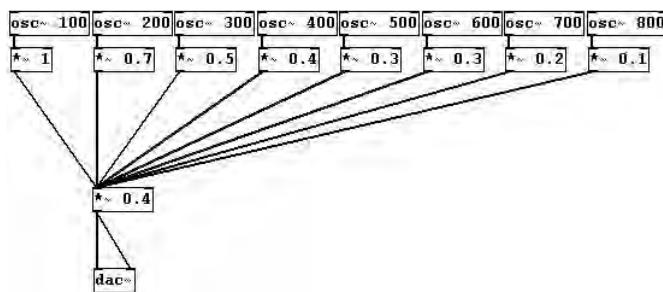
Physicist Jean Baptiste Joseph Fourier (1768-1830) discovered that every periodic sound can be represented using only sine tones (of different frequency, amplitude, and phase),

the sum of which is then identical with the original. Such an analysis and the corresponding mathematical process is called a Fourier analysis and Fourier transformation.

Using this principle, it is possible to create every periodic sound by layering many sine tones, a process called “additive synthesis”.

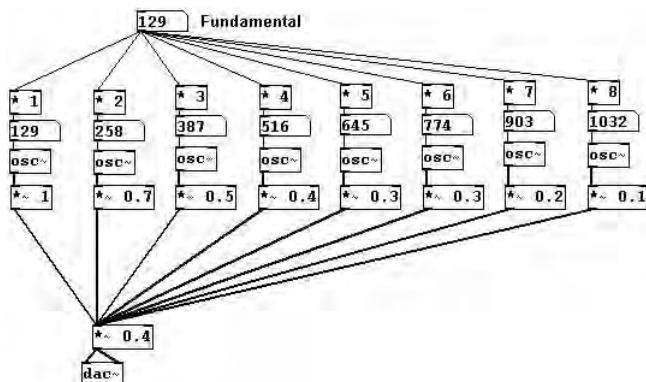
In Pd, as already mentioned, “`osc~`” can be used to generate a sine tone. Sine tones are a very characteristic sound of electronic music, as they are produced and can only be produced using electronic means.

Using a number of “`osc~`” objects, whose frequencies form an additive series, you can create a chord based on the overtone series:



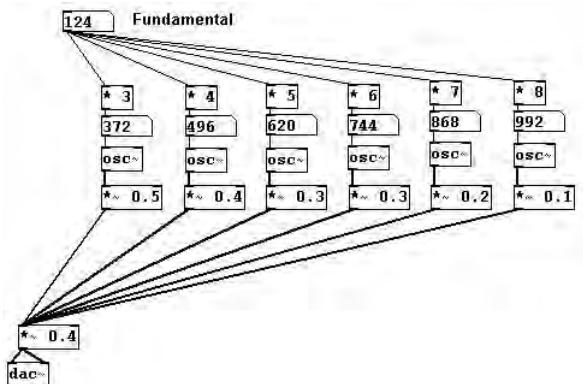
Typically, amplitudes become smaller as the frequencies get larger in order for the chord to blend better (though for some instruments, it is characteristic for certain partials to be louder than those on either side of them, e.g., the clarinet). The arrangement and relative volumes of overtones determine a sound’s *color*. You can also speak of its *spectrum*.

The fact that our ears blend the overtones together becomes clear when you change the fundamental frequency:



We'll just use the first eight partials here. (N.B. The term 'partial' includes the fundamental whereas the term 'overtone' does not. In other words, the 1st partial = the fundamental frequency, 2nd partial = 1st overtone, 3rd partial = 2nd overtone, etc.)

Even if you leave out the lower partials, you hear the fundamental frequency as the fundamental when you change it:



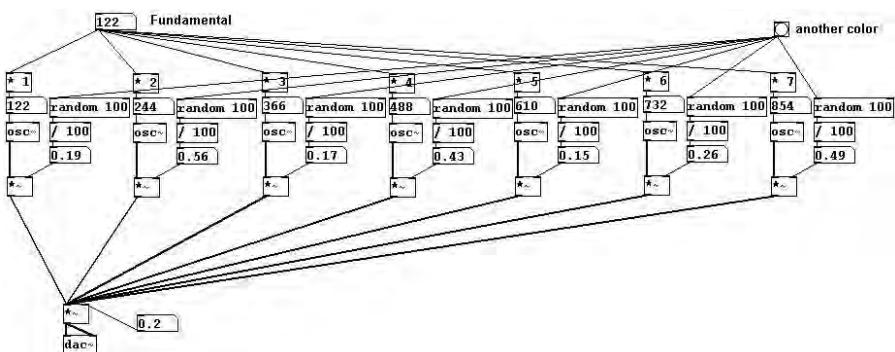
Our brain calculates the fundamental based on the remaining spectrum. This non-existent tone is called a *residual tone*.

### ◆ 3.2.2 APPLICATIONS

#### ► 3.2.2.1 A random klangfarbe (German: sound color)

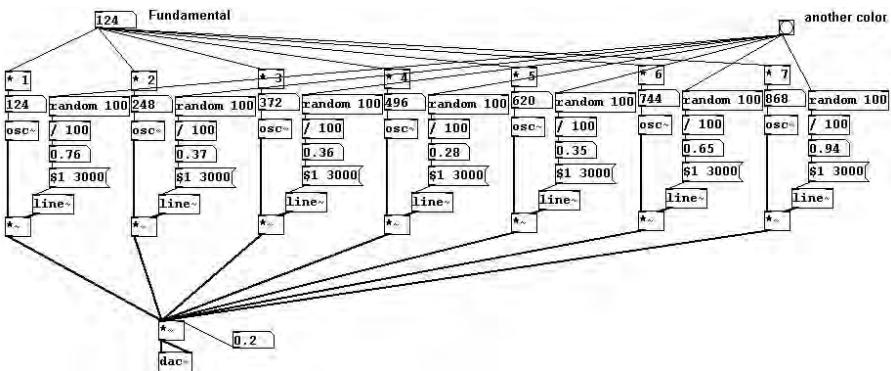
[patches/3-2-2-1-random-color.pd](#)

For the sake of space, this example has been limited to just the first seven partials:



### ► 3.2.2.2 Changing one klangfarbe into another

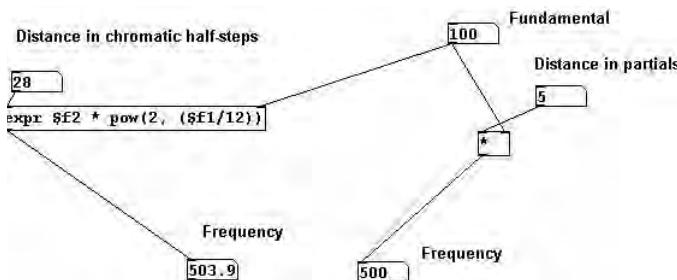
patches/3-2-2-2-colorchange.pd



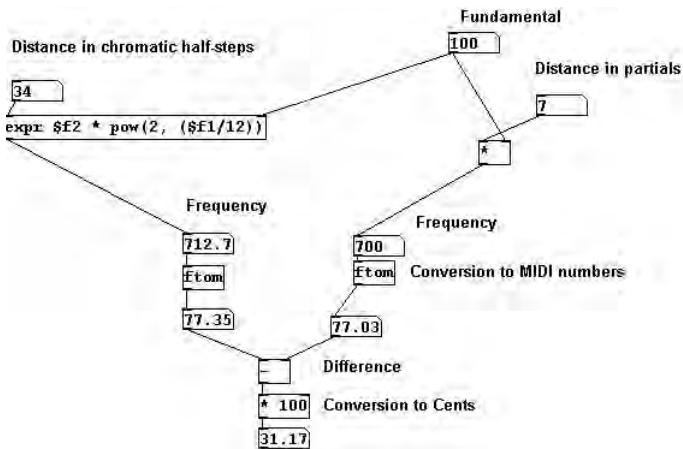
### ► 3.2.2.3 Natural vs. equal-tempered

Let's look at the difference between natural and equal-tempered intervals (first enter the fundamental frequency!):

patches/3-2-2-3-natural-tempered.pd



Showing the difference between natural and equal-tempered tuning in cents (hundredths of a half-step):



You can see here: the 7th partial is 31 cents flatter than the equal-tempered seventh.

#### ► 3.2.2.4 More exercises

Create an overtone chord with manipulated overtones, i.e., with imprecise overtones.

#### ◆ 3.2.3 APPENDIX

##### ► 3.2.3.1 Pd's limitations

The previous example of random klangfarbe reveals one of Pd's limitations: you can't randomly determine the number of oscillators. You have to at least determine the maximum first.

#### ◆ 3.2.4 FOR THOSE ESPECIALLY INTERESTED

##### ► 3.2.4.1 Studie II

One of the pioneering pieces in the history of electronic music is ‘Studie II’ by Karlheinz Stockhausen, written in 1954. This work uses only sine tones and mixtures thereof in non-tempered intervals. The author strongly recommends you analyze this piece!

##### ► 3.2.4.2 Composing with spectra

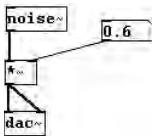
In the fourth chapter of his book “Audible Design”, composer and theorist Trevor Wishart describes many possibilities for composing with spectra.

### 3.3 Subtractive synthesis

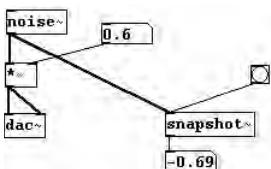
#### ◆ 3.3.1 THEORY

##### ► 3.3.1.1 White noise

Claude Debussy once responded to the question of how he composed by saying he starts by taking all pitches and then leaves out the ones he doesn't like. He foresaw the idea of filtering. In contrast to additive synthesis—which uses what might be considered the 'atom' of sound, the sine tone, as a starting point—subtractive synthesis begins with all sound and reduces it. It is actually possible to produce all sound. Causing a speaker membrane to vibrate completely chaotically and randomly will produce all audible frequencies simultaneously. The Pd object used to accomplish this is called “noise~”:



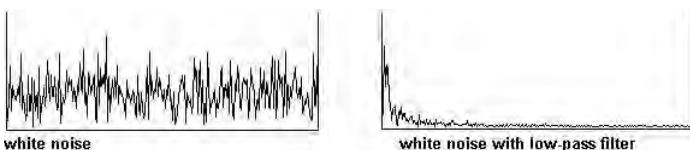
Technically speaking, it would be more accurate if “noise~” were named “random~” instead, because it produces 44100 random numbers per second. These numbers occur in a range of -1 to 1, i.e., membrane positions.



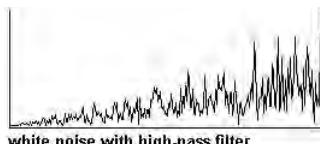
##### ► 3.3.1.2 Filters

Like light, noise that contains all audible frequencies is called “white noise”. Normal white light contains all light frequencies while, say, red or blue light can be derived from it using filters.

Pd also has filters such as “lowpass”, which allows only the low frequencies to pass through while suppressing the high frequencies. This is represented in the following diagram; the x-axis represents frequency and the y-axis amplitude:

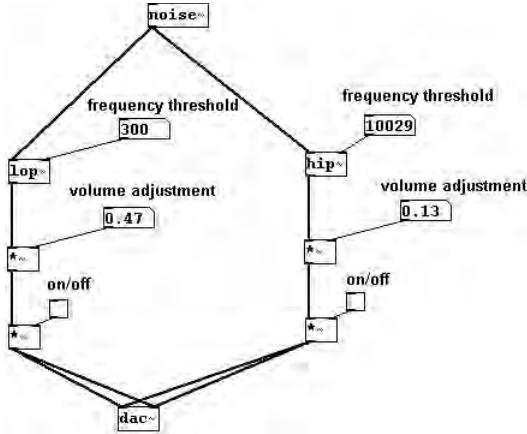


There is also a “highpass” filter, which only allows high frequencies to pass through:

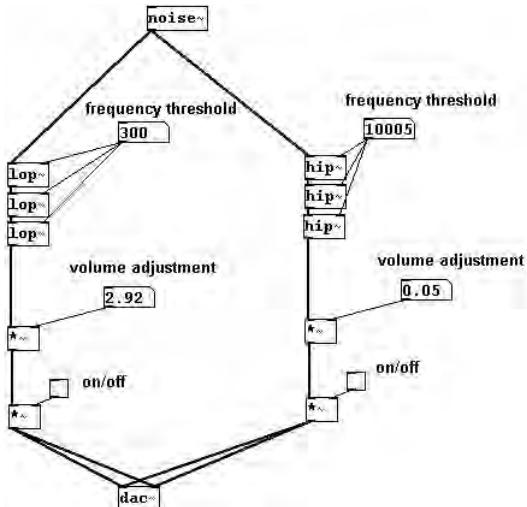


white noise with high-pass filter

The Pd objects for these filters are called “`hip~`” and “`lop~`”. Their argument or right input is the frequency from which the sound should be filtered.

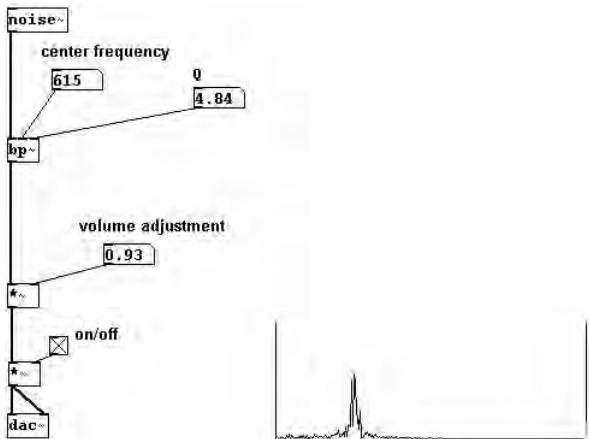


As you can see in the preceding diagrams, the filters are not particularly ‘steep’. However, you can intensify their effect by using several filters one after another (cascade):

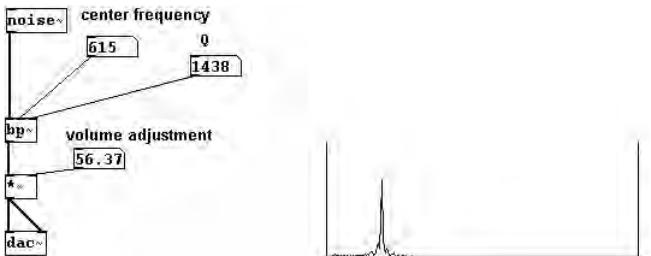


The volume has to be readjusted for each filter, as they reduce the sound's intensity. (Though they sometimes strengthen other things.)

Another kind of filter is called a "band-pass". This allows only a small portion of sound surrounding a central frequency to pass through, like a 'band' of frequencies. As arguments/inlets it receives the central frequency and the width of the band, called "q".



Theoretically, if the band gets small enough, you should end up with just a single sine tone:



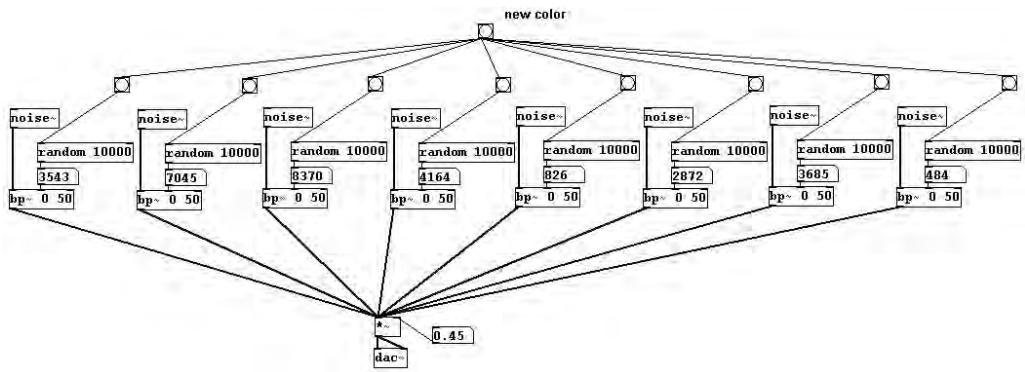
As you can easily hear, however, this is not the case. A certain noise element is always leftover with a band-pass filter.

#### ◆ 3.3.2 APPLICATIONS

##### ► 3.3.2.1 Filter colors

Just as an example of how filters might be used, here is a random distribution of band-pass filters:

[patches/3-3-2-1-filtercolors.pd](#)



### ► 3.3.2.2 Telephone filters

For transmitting telephone conversations, it was determined that frequencies from just 300 to 3000 Hz suffice for comprehending speech. You can simulate that:

`patches/3-3-2-2-telephonefilter.pd`



### ► 3.3.2.3 More exercises

Experiment with filtering the “glissando orchestra” (3.1.2.2.4).

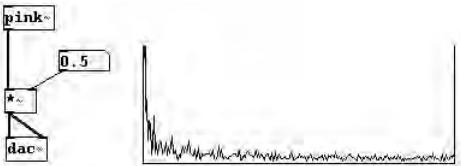
## ◆ 3.3.3 Appendix

### ► 3.3.3.1 White noise and clicks

The “noise~” object causes the speaker membrane to vibrate randomly. When you turn this sound on or off you don’t hear a click; this is because noise is composed solely of clicks of varying intensity. Therefore, you don’t need a “ramp” (cf. 3.1.2.1.2).

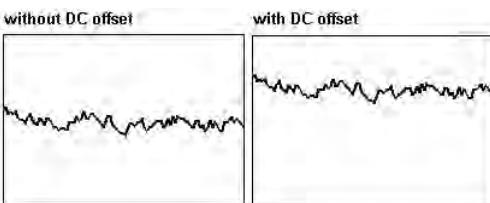
### ► 3.3.3.2 Pink noise

In addition to white noise, there is also “pink” noise. The human ear does not hear all frequency ranges at the same volume. It hears best around 2000 Hz, that’s why white noise sounds somewhat high. We hear considerably worse in the low and very high frequency ranges. If you want to create a noise that humans will perceive more as an evenly distribution of all frequencies, you have to adapt it to the way we hear, i.e., the low frequencies have to be significantly louder than the middle frequencies. This distribution is provided by pink noise and can be generated in Pd using the “pink~” object:

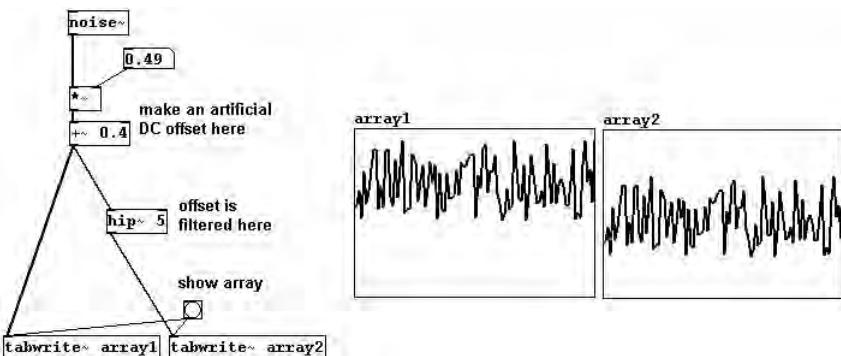


### ► 3.3.3.3 DC offset

When using a microphone, the signal will often exhibit a ground current. This is called “DC offset”. The result is this waveform:



This offset amounts to an infinitely slow vibration with a frequency that approaches 0. Because it is so low, it can be filtered out with a high-pass filter that is set extremely low:

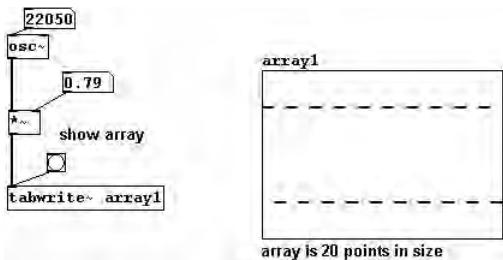


◆ 3.3.4 FOR THOSE ESPECIALLY INTERESTED

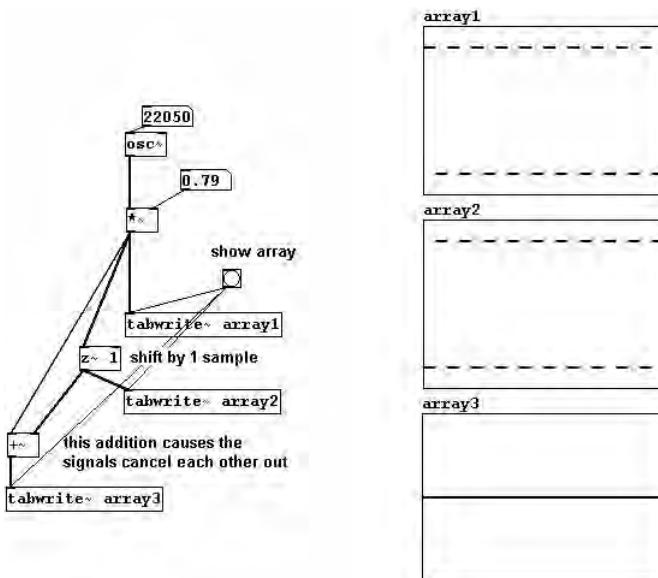
► 3.3.4.1 How digital filters work

The ‘inner life’ of digital filters is complicated. One should, however, have some sort of an idea about how precisely they work: As described in 3.1.1.3.1, a sample rate of 44100 Hz is capable of representing a wave with a maximum of 22050 Hz. This wave would have only two points per period:

patches/3-3-4-1-filterwork.pd



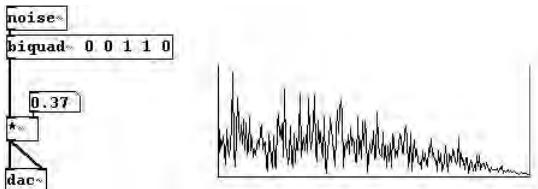
If you move this wave just one position, i.e., by one sample forwards or backwards, and then add it to the original wave, the result will be that the two waves cancel each other out completely. This shift can be accomplished (in Pd-extended) with “z~”.



Digital filters employ this method of delaying a wave by one sample and then adding it to the original wave to effect cancellation. The “biquad~” object can be used to adjust

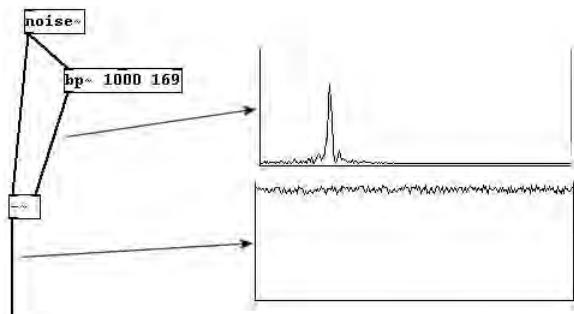
this by hand. It executes the following differential equation:  $y(n) = ff1 * w(n) + ff2 * w(n - 1) + ff3 * w(n - 2)$  with  $w[n] = x[n] + fb1 * x[n - 1] + fb2 * x[n - 2]$ .

'n' is the sample position and ff1, ff2, ff3, fb1, and fb2 are freely defined factors. In Pd, "biquad~" then requires five arguments for ff1, ff2, ff3, fb1, and fb2. The syntax is as follows: "biquad~" [fb1] [fb2] [ff1] [ff2] [ff3]. For the case of the wave of 22050 Hz discussed at the beginning of this section, you could also write "biquad~ 0 0 1 1 0". This suppresses high frequencies, especially waves with a frequency of 22050 Hz, which are completely cancelled out. Here is a low-pass filter using "biquad~":



You can use the biquad formula to create many other kinds of filters. For example, the arguments 1.41407 -0.9998 1 -1.41421 1 will make a "band-reject filter". This is the inverse of a band-pass filter; it rejects—i.e., blocks—a certain band of frequencies around a central frequency, in this case 5512.5 Hz. The explanation for this type of calculation would fill an entire book. In Pd-extended there are objects ("band-pass", "equalizer", "highpass", "highshelf", "hlshelf", "lowpass", "lowlshelf", "notch") that carry out these calculations. The advantage of the biquad filter is that considerably steeper filter profiles are possible than with, say, "lop~", "hip~", or "bp~". The drawback is that it not only suppresses certain frequencies but also significantly intensifies others to the point of "explosion" (you can see in the formula, that the filter works recursively).

With biquad processing you can also see that filters employ phase shifts. That's why, e.g., a "bp~" object isn't simply the inversion of a band-reject filter:



## 3.4 Sampling

### ◆ 3.4.1 THEORY

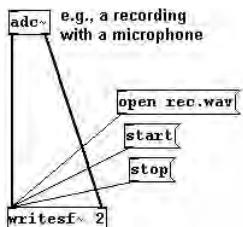
Let's clear up a potentially problematic issue to avoid any confusion: you've already learned that a sample is the smallest unit used for measuring and generating sound in a computer. The word 'sample' unfortunately also has another, quite different meaning in electronic music; it means a smallish section (usually a couple seconds long) of recorded sound. This chapter deals with the processing of short recorded bits of sound. First you'll have to learn how "array" works in Pd, which requires a fair amount of explanation.

#### ► 3.4.1.1 Storing sound

##### ▷ 3.4.1.1.1 Sound files

There are various locations in the computer where files can be saved: main memory or hard disks. Access to main memory is very fast in comparison to the hard disk; however, it has much less available space.

In Pd you can save sound to both locations. Saving to the hard disk means that you are saving a fixed sound file. WAV or AIFF formats are typically used. Use the "writesf~" object to write a sound file to disk. The argument is the number of channels; this creates a corresponding number of inlets to which you attach the sounds you want to record. First you have to use the message "open [name]" to choose the name of the file you want to create. Start recording using "start" and stop it using "stop".

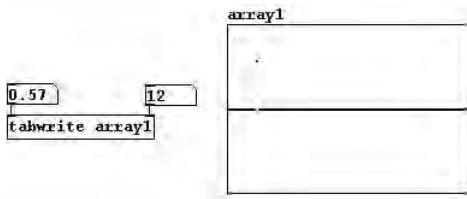


##### ▷ 3.4.1.1.2 Buffers

The other possible location is the main memory. Create one place for one sound using "array" (**Put** → **Array** then click "ok"). It is also a visualization of the sound.

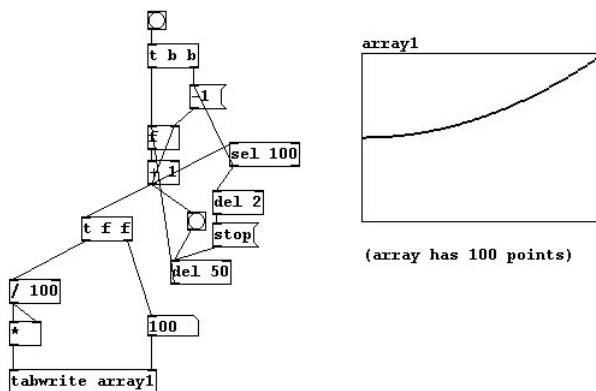
Let's first think of an array simply as storage for numbers. An array has a limited number of storage places. You can set this number using by right-clicking the array and going to "Properties". This opens two windows: one labeled "array" and the other labeled "canvas". In the "array" window, you can set the size. This number means the number of storage places. One number can be stored in each storage place.

You can allocate these cells using “tabwrite”. The right argument determines the position; the left determines the value you want to save (as always: from right to left). In the array, the x-axis shows position and the y-axis shows the value:



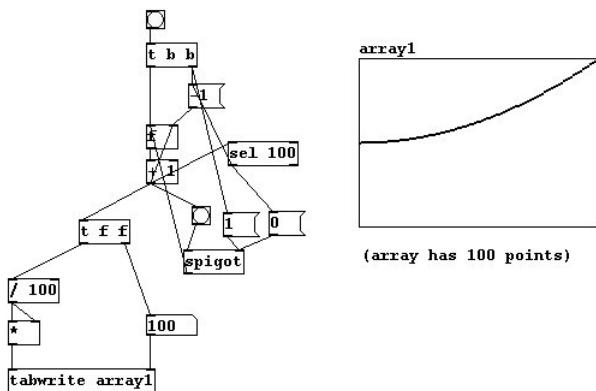
You can use “array” to represent functions:

`patches/3-4-1-1-2-function1.pd`

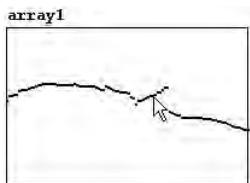


Or without temporal stretching (if the example shown causes a stack overflow, inserting “`del 0`” between “spigot” and “`f`” will help):

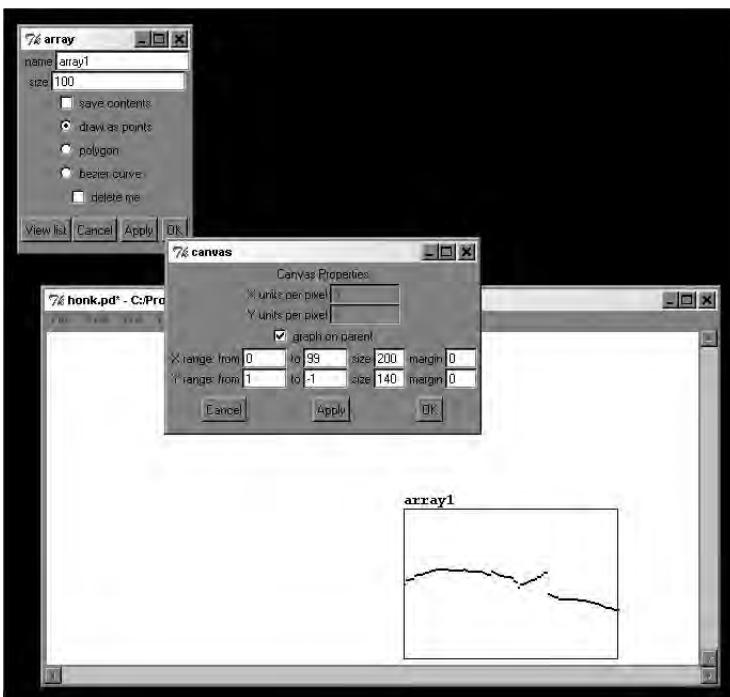
`patches/3-4-1-1-2-function2.pd`



You could also draw in the array itself with the mouse, ‘by hand’ so to speak. If you move the mouse to a value in the array, the cursor (shaped like an arrow) will change its direction and you can draw by moving the mouse with the mouse button held.



Under ‘Properties’ (right-click on the array), you can set the following:



#### In the “array” window:

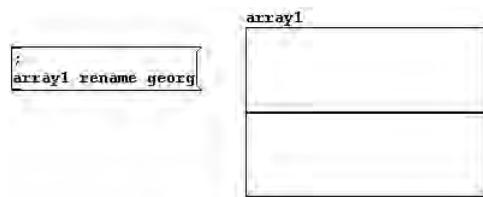
- Name: Like all names in Pd, use alphanumeric characters without spaces, not only numbers.
- Size: As described above.
- “save contents”: When checked, all values in an array will be saved. If you use large arrays or a large number of arrays, this could cause your patch to load very slowly.
- “draw as points” / “polygon” / “bezier curve”: Different kinds of visualization.
- “delete me”: When checked, this deletes the array! The empty box remains, however, and must also be deleted.
- “view list”: This displays all values in a list.

### In the “canvas” window:

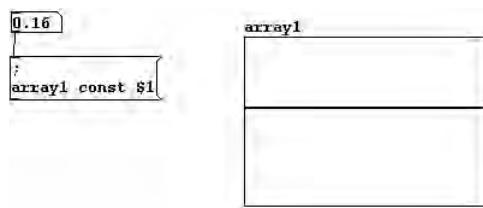
- “graph on parent”: This will be discussed later in 3.1.1.2.
- X range: You can set the range for the x-axis here.
- Y range: You can set the range for the y-axis here. Values that fall outside this range will also be saved, but this will enlarge the entire window so that these values can be seen.
- “size”: Visual size in the patch.

An array can also receive messages and “sends”.

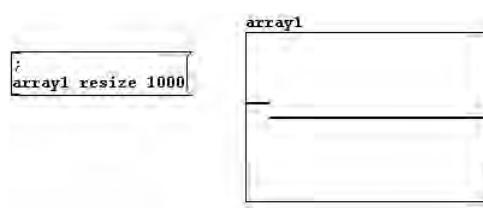
Renaming:



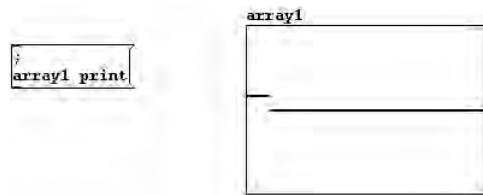
Making all values equal:



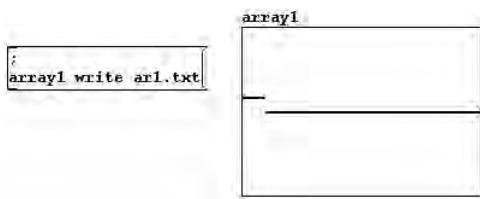
Changing the size:



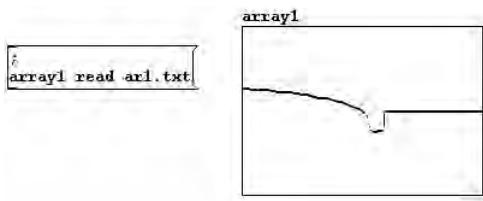
The size can also be printed out:



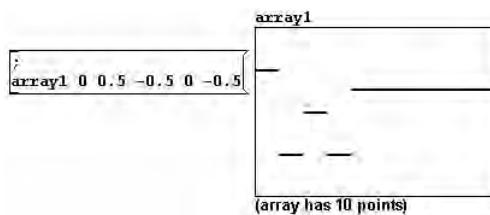
Writing the contents to a text file:



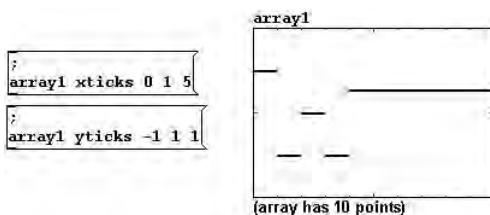
Reading a text file:



You can also enter values like this. The first number determines which storage place to start with; all other values are for the positions thereafter:

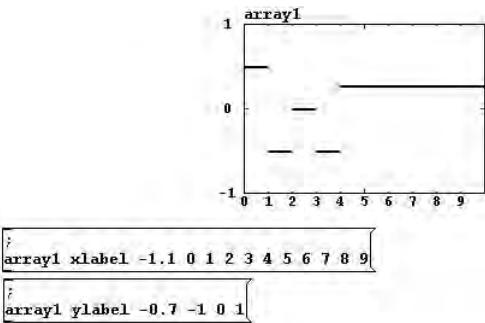


You can also create hash marks on the axes and label them:



The first argument is the starting position; the second is the distance between the lines; the third is the distance between the longer lines.

You can also number them:

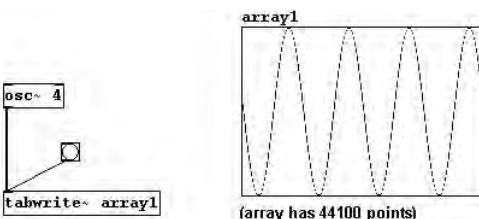


First, the position is labeled; second, the numbers you want to display.

N.B.: Lines and labels are not stored in the patch, so you have to reenter them every time you open the patch.

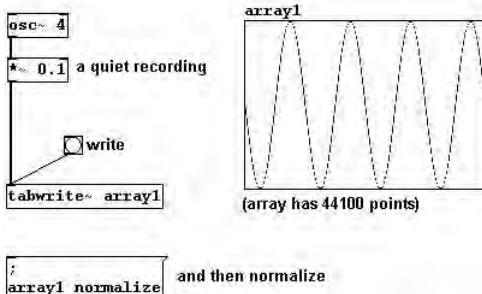
You can, of course, also store sound in an array. For a computer, sound is—as has been often mentioned—nothing but a string of numbers, 44100 numbers to be precise. You could store one second of sound in an array; you'd just need to use 44100 storage places. Here's where the “tabwrite~” object comes into play. It receives the sound input and otherwise just a bang. Unlike with “tabwrite” (without a tilde) where you entered the storage place in the right inlet ‘by hand’ and the corresponding value on the left, when “tabwrite~” receives a “bang”, it automatically starts with the first storage place and then proceeds at sample speed (44100 samples/sec). At the same time, every place is allocated a value received by the left inlet from the current sound, resulting in a total of 44100 stored numbers. Any sound that follows is not stored. If you want to stop prematurely, you can send a “stop” message.

patches/3-4-1-1-2-normalize.pd



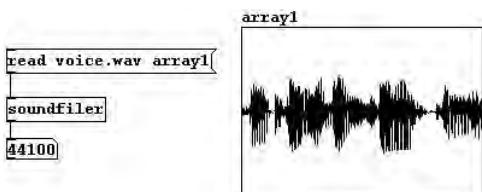
If “tabwrite~” receives a float message, this number is interpreted as a sample offset. In other words, the sample that corresponds to this float number will be the starting point for the array.

One useful function is that it's possible to raise the overall volume after the fact. If, for example, the original recording is too quiet (i.e., the membrane of the microphone didn't vibrate especially strongly, which resulted in fairly small values), you can amplify it. This is called “normalizing”. For this, you can use the following message:



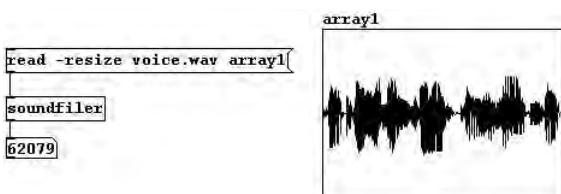
You can also make a connection between sound files located in the main memory and those located on the hard disk in array. This is achieved with the “soundfiler” object. This object allows you to load a sound file stored on the hard disk into an array, or to save the contents of an array on the hard disk as a sound file. The command “read” is used to load a sound file. The arguments for the command are the name of the file (with the path if necessary) and then the name of the array to which you want to write.

[patches/3-4-1-1-2-load-soundfile.pd](#)

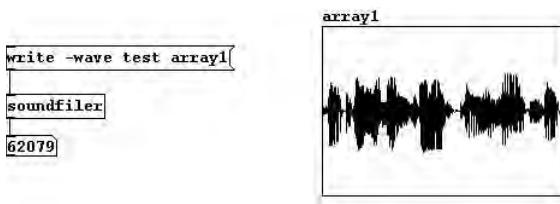


Once the file has successfully loaded, the size of the sound file in samples will be sent out the “soundfiler” object’s outlet.

You can also include other commands (called “flags”) in the message:



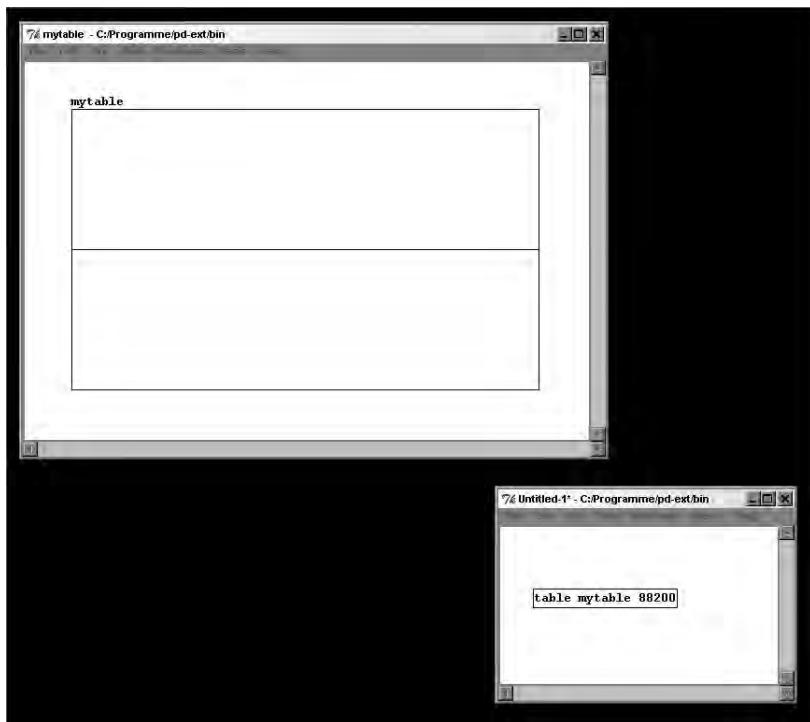
The resize command changes the array’s size to match that of the sample (this is limited to 4000000 samples—about 90 seconds, though this can be changed with “maxsize”). Conversely, the command “write” saves the contents of an array as a sound file to the hard disk. In this case, the format (WAV or AIFF) must be given as a flag, then the name (with path designation if necessary) of the file you want to create, and then the name of the array.



### Other important “flags” in this context are:

- normalize: Optimizes a file’s amplitude levels, as explained previously.
- rate: Used to set the sample rate for a file.

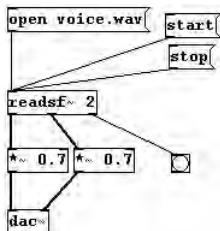
As an alternative to array, you could also use “table”. Create a “table” object; enter the name for the first argument and the size in samples for the second. This will create an array in a subpatch (click on the object in execute mode) that is treated like a normal array. This approach has the following advantage: the graphics for a normal array can be very complex. You can notice this when you move a big array around on the canvas: it moves very slowly. But if the graphic representation of an array is in a subpatch, the object itself can be moved much more easily.



### ► 3.4.1.2 Playback of saved sound

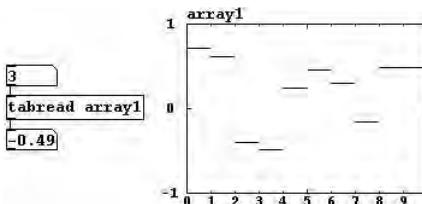
Sound files that are on an external storage device like a hard disk can be read—that is, played back—in Pd with “readsf~”. As with “writesf~”, you use the messages “start” and “stop” (you could also use “1” and “0”). Enter the number of channels as the argument. The rightmost output sends a bang when the end of a file is reached.

patches/3-4-1-2-play-file.pd



Let's look at the control level for ‘array’: let's say you have an array with 10 storage places. You can use “tabread” to read every one of these places:

patches/3-4-1-2-read-array1.pd

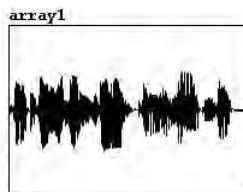


The principle is basically the same for signals, except that you have to receive the saved values at a rate of 44100 numbers per second. That's why there is “tabread~”. If you want to, say, play a sound stored in an array that lasts 1.5 seconds (= 66150 samples), you have to read the array values from 0 to 66149 at a rate of 44100 values a second. You can do this with “line~”:

patches/3-4-1-2-read-array2.pd

```
read voice.wav array1 load sound  
soundfiler
```

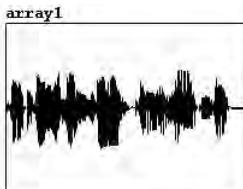
```
0, 66100 1500 play back  
line~  
tabread~ array1  
*~ 0.7  
dac~
```



“tabread~” receives the array name as an argument. You could also set the array you want to read with the message “set [arrayname]”.

Now you can start to play with these values. For example, you could play it back in twice the time:

```
0, 66100 3000 play back  
line~  
tabread~ array1  
*~ 0.7  
dac~
```

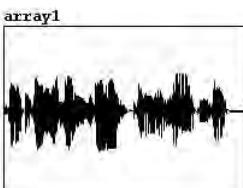


This cuts the playback speed in half. This causes everything to sound an octave lower because the time stretching makes all the soundwaves twice as long, which means their frequencies will be cut in half and thus sound an octave lower.

This leads the problem that every single sample you use needs to be played twice or else there will be gaps. To avoid this problem, there is a modified form of the “tabread~” object called “tabread4~”, which interpolates intermediate values that it generates using information from the values that ultimately precede and follow it. (More specific information on this function is available at 3.4.4.) In most cases, “tabread4~” is more suitable for reading arrays. This requires a readout spectrum from 1 to n—2, where ‘n’ is the size of the array you want to read.

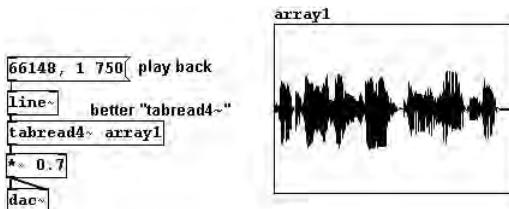
Of course, you could also play something back faster, which would raise the frequency:

```
1, 66148 750 play back  
line~ better "tabread4~"  
tabread4~ array1  
*~ 0.7  
dac~
```

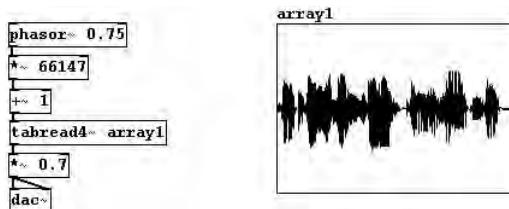


Later, when granular synthesis is explained, you'll learn how to alter tempo and pitch independently.

Playing a sample backwards is naturally also a possibility:



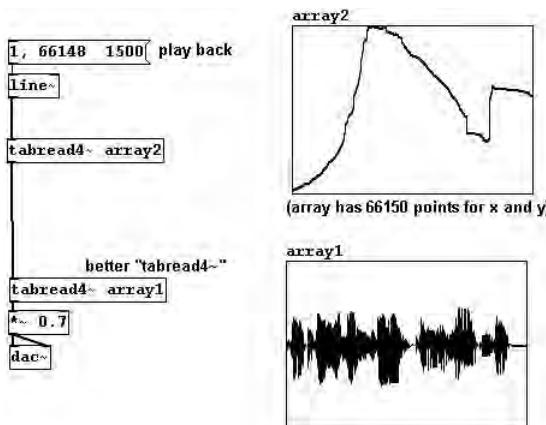
You could also use the “phasor~” object:



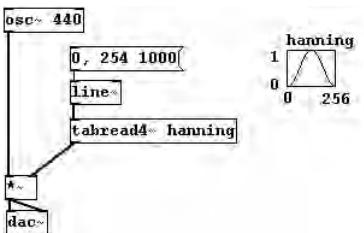
“phasor~” generates a series of numbers between 0 and 1 as a signal. If you multiply these values by 66148, you get a series of numbers from 0 to 66148. Enter 0.75 for the frequency so that the series occurs in exactly 1.5 seconds.

Another possibility would be to create your own array for the arrangement of the read-out and use it to play back the first array:

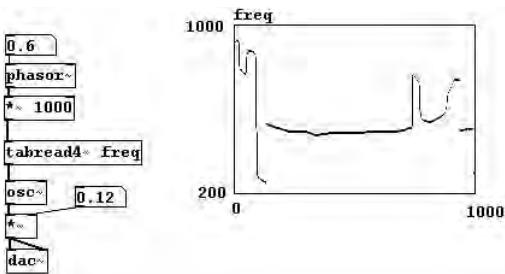
patches/3-4-1-2-read-array3.pd



Or you could use an array to control the amplitude:



Or to control the frequency:



Once again you can see: we're using only numbers to control various parameters.

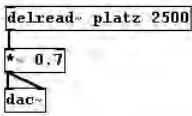
You can use as many “tabread~” objects as you like to read from the same array. However, you should never use two arrays with the same name, as it will almost certainly lead to errors.

#### ► 3.4.1.3 Audio delay

In Chapter 2.2.3.1.2 we mentioned how numbers or series of numbers can be delayed. You can also do this with signals. This is done by creating a buffer into which signals are written and out of which signals are read following a certain delay. To create this buffer, you use a “delwrite~” object. The first argument is a freely chosen name; the second is the size in milliseconds. As input, give it the signal you want it to write in the buffer. Once the buffer is full, it is written over again from the beginning. If the buffer is 1000 milliseconds long, the last 1000 milliseconds of the incoming signal are stored in the buffer.

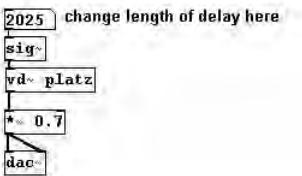
Use “delread~” to read from the buffer. The first argument is again the buffer name; the second is the delay (in milliseconds; can be changed using a control data entry in the input):

patches/3-4-1-3-delay.pd



Logically the amount of delay in “delread~” must be smaller than or equal to the buffer size. If you have a delay of 2000 milliseconds but the buffer holds only 1000 milliseconds, it clearly won’t work. Using a negative number for the delay interval is also impossible, as even Pd can’t see into the future. You can use as many “delread~” objects as you like to read simultaneously from a delay buffer. You cannot look into the wave patterns in the buffer.

While you can change the delay interval in “delread~”, you have to use a control data entry and there is a certain probability of error once you exceed a certain speed (this is again a conflict between control data and signals). For this reason, there is a special object for variable readings of delay buffers called “vd~” (short for “variable delay”). You give the delay interval (in milliseconds) as a signal as input and can change it however you like (though, again, you can’t use negative numbers or exceed the buffer size):

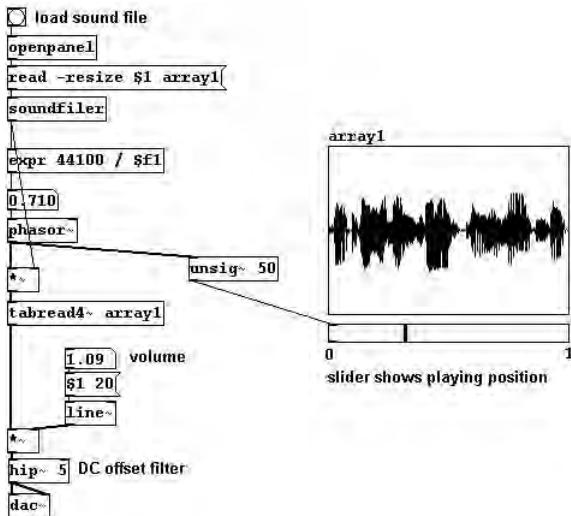


“vd~”, like “readsf4~”, creates an interpolation.

#### ◆ 3.4.2 APPLICATIONS

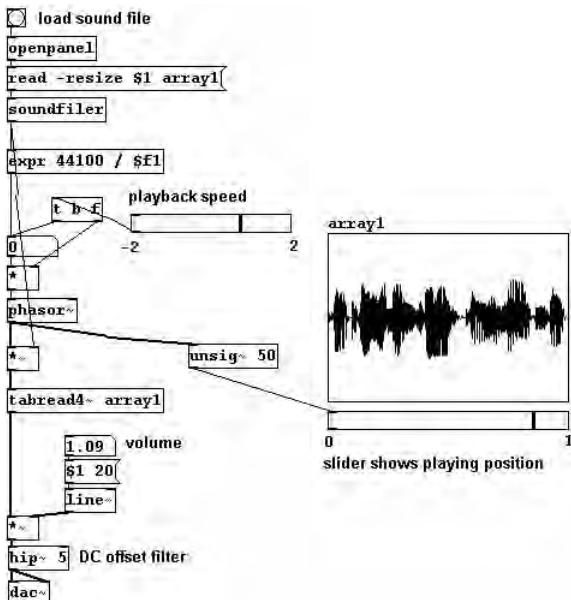
##### ► 3.4.2.1 A simple sampler

patches/3-4-2-1-simple-sampler.pd



##### ► 3.4.2.2 With variable speed

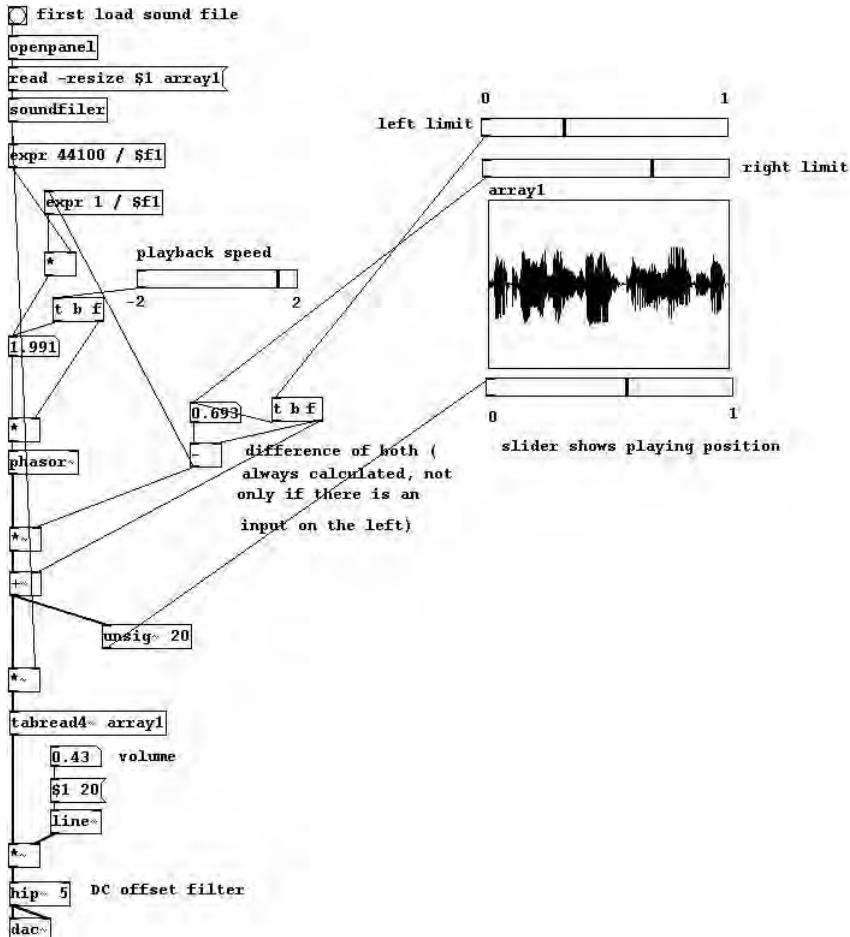
patches/3-4-2-2-sampler2.pd



### ► 3.4.2.3 Any position

Here's one way to pick out any position from a sample that you want:

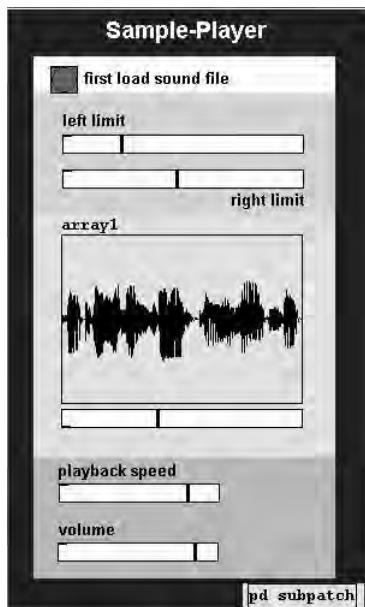
patches/3-4-2-3-sampler3.pd



#### ► 3.4.2.4 Sampler-player

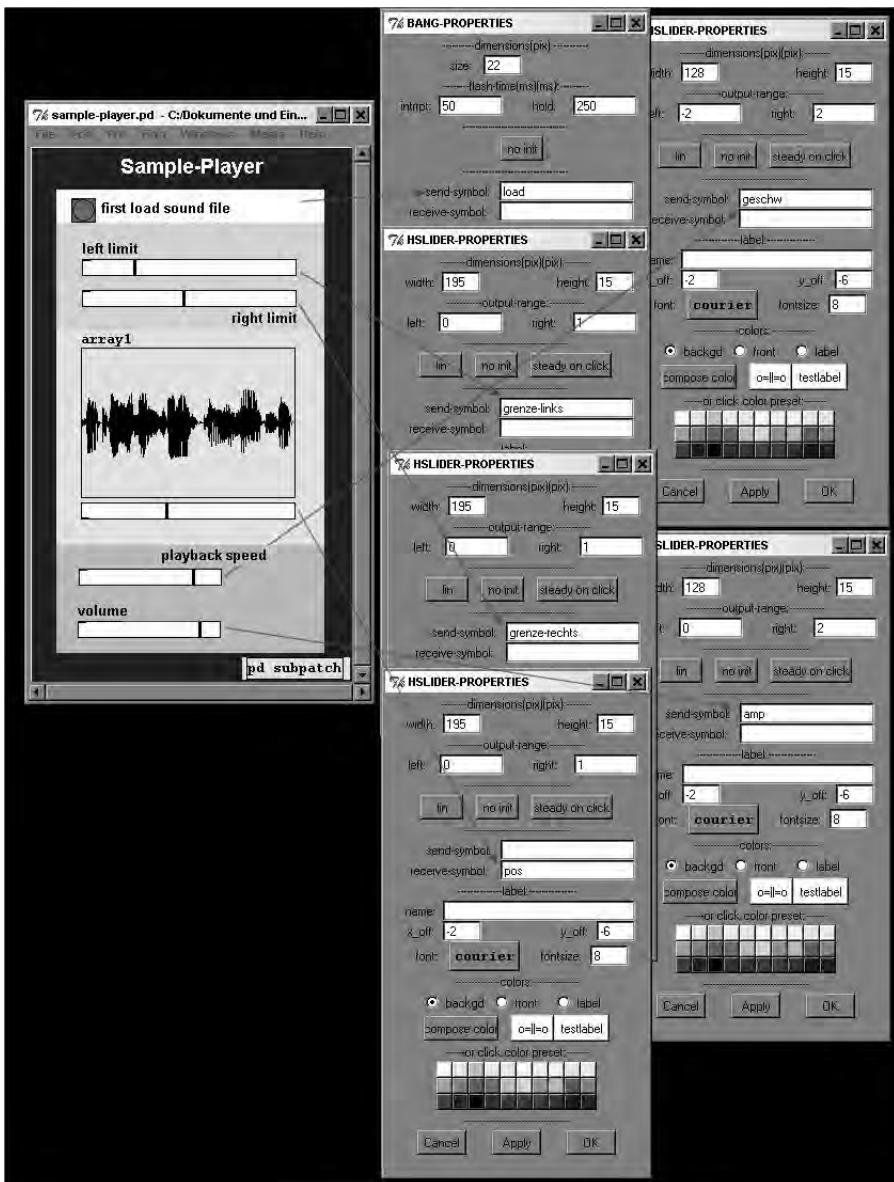
If you change the graphic representation as described previously (2.2.4), then your patch could look like this:

patches/3-4-2-4-sampler-big.pd

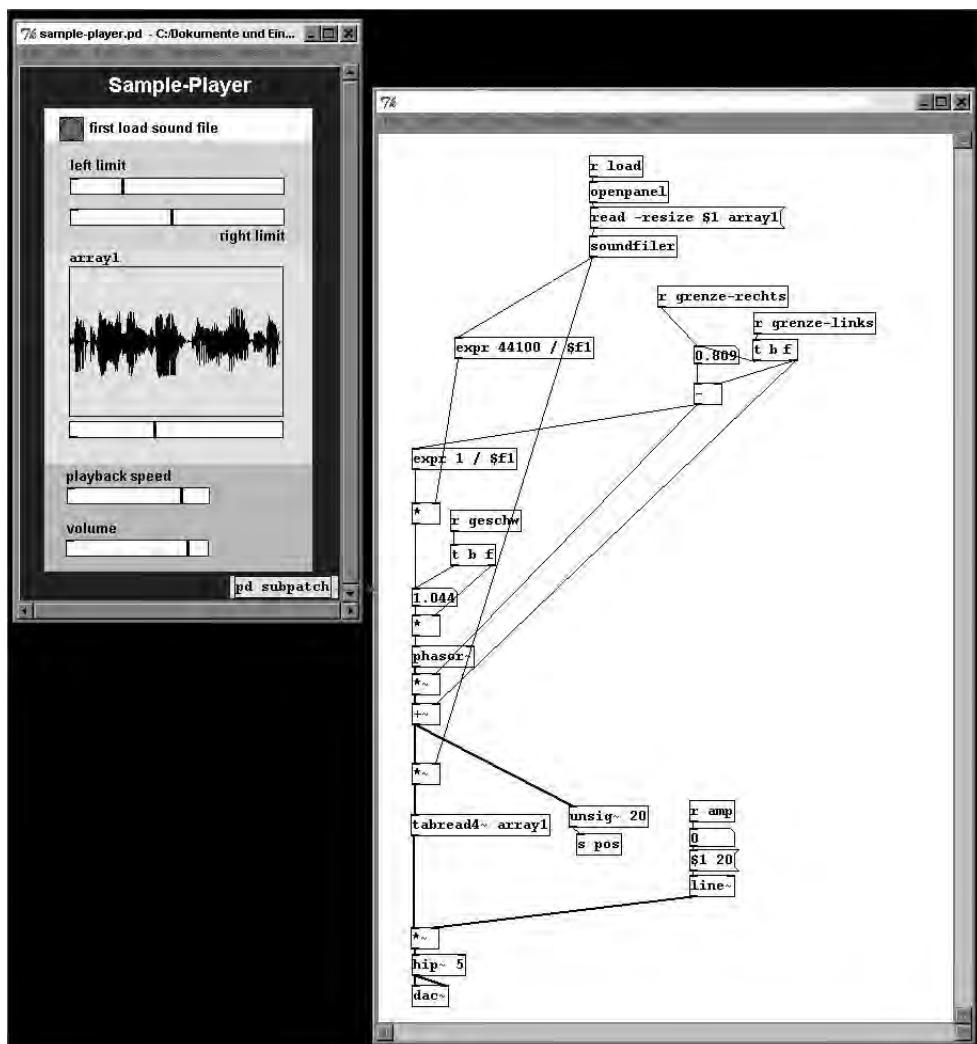


Four canvas objects make up the colorful background of the sliders and array. N.B.: The graphics that were created last always appear on top of the other ones. Let's say you have "array1" and then you make a colored canvas object; you have to create "array1" once again (just copy and then delete the old one) so that it appears on top of the canvas object.

To explain exactly how this was done:

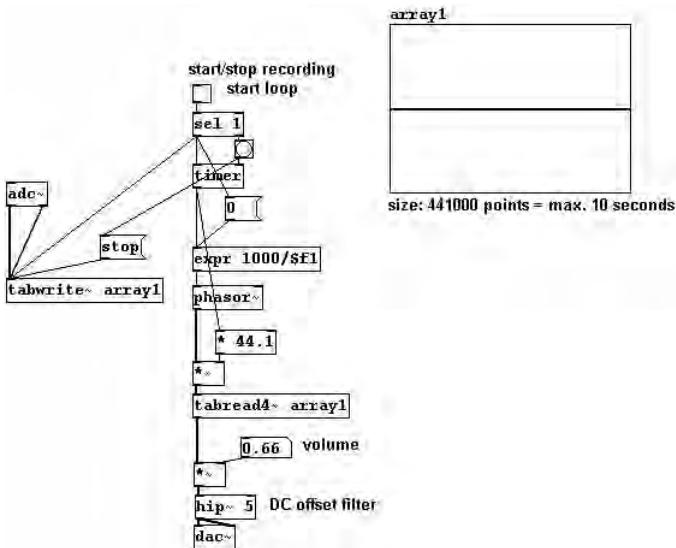


And in the subpatch:

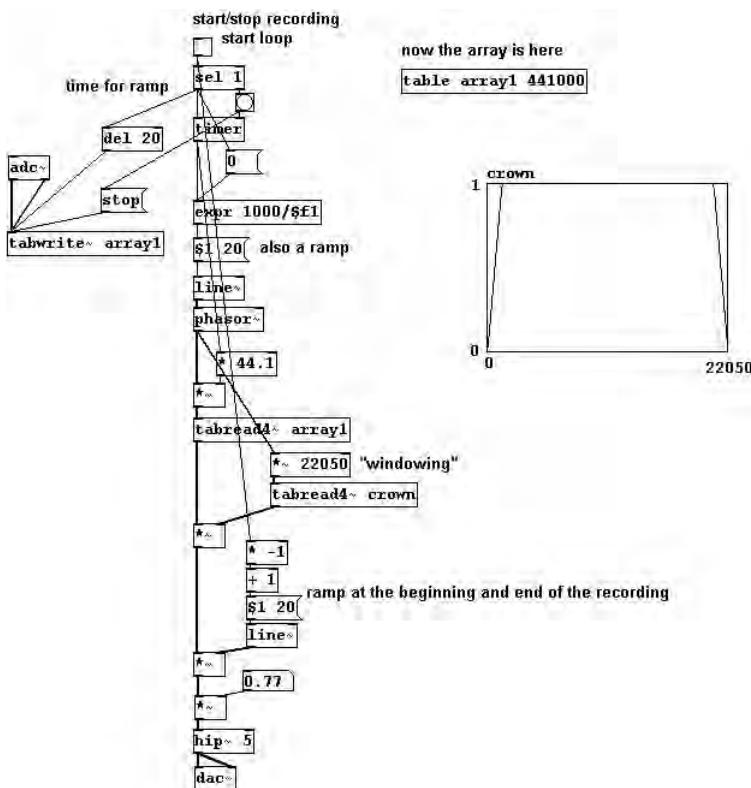


### ► 3.4.2.5 Loop generator

patches/3-4-2-5-loop-generator1.pd

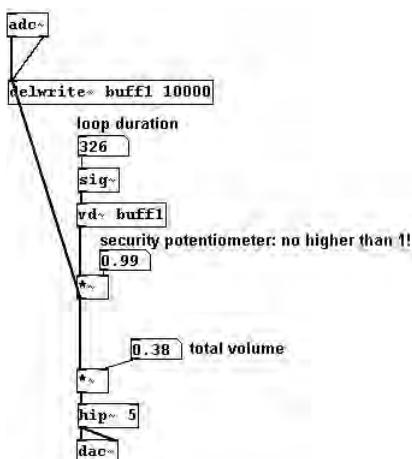


But with this loop generator, clicks occur. First, it is highly recommended that you put the array in a subpatch, because the graphic requires a lot of processing power. Second, on the loop's ends you should briefly go to 0 so that there is no sudden jump in value (which would cause a click). For this, you need to program "windowing". In sync with the readout of the array, this is determined in the amplitude by another array (here, "crown") that controls the dynamic envelope. This envelope has a value of zero at the beginning and end to ensure that there is no sudden change to the value when the loop repeats. Instead of the "crown" window, you could also use a "Hanning" window, which uses a part of the sine function (this will be covered later). Of course, the "crown" array should also be in another window as well, but it has been left this way for the sake of clarity.



A simpler version of the loop can be created using feedback:

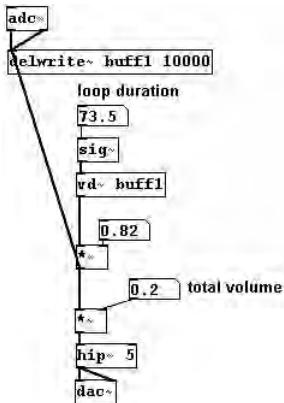
patches/3-4-2-5-loop-generator2.pd



A drawback to this is that there is a maximum loop duration; here it is 10000 milliseconds.

#### ► 3.4.2.6 Reverb

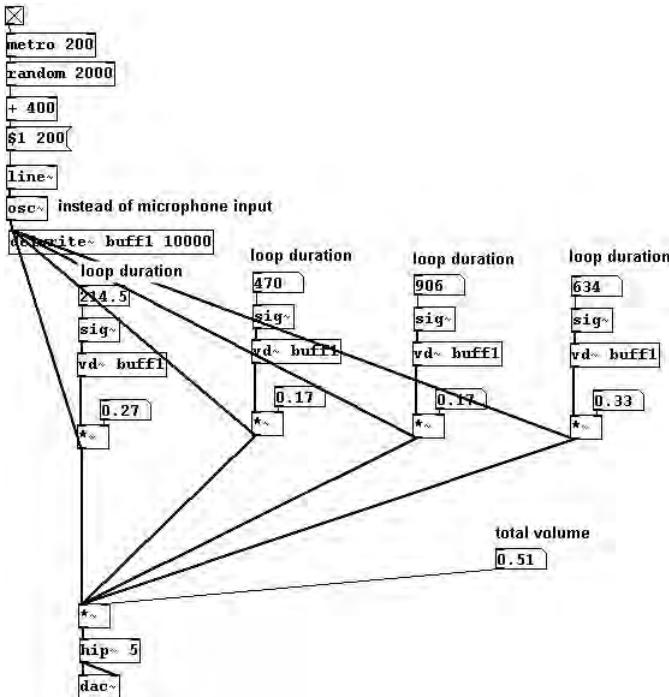
You can simulate a “reverb” effect if the signal feedback gets quieter and quieter:



#### ► 3.4.2.7 Texture

Or you can create a texture:

patches/3-4-2-7-texture.pd

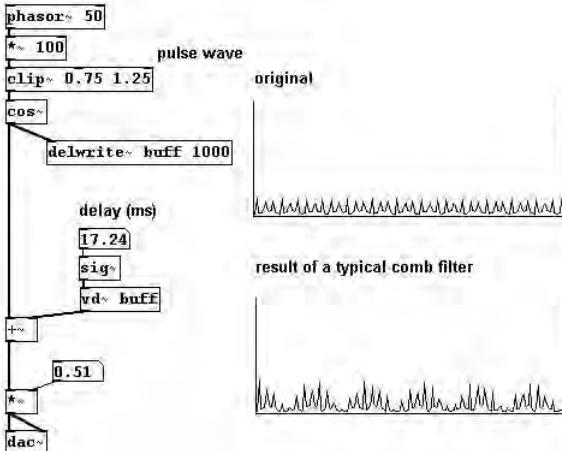


You have to be careful that the feedback doesn't 'explode', i.e., that its volume doesn't increase exponentially.

#### ► 3.4.2.8 Comb filter

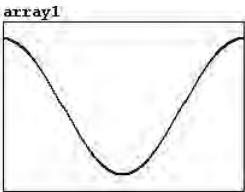
You can build a comb filter using audio delay. The idea is that you add the delay to the original signal. This results in amplifications and cancellations at regular intervals, which gives the spectrum the appearance of a comb:

patches/3-4-2-8-combfilter.pd

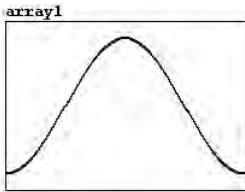


#### ► 3.4.2.9 Octave doubler

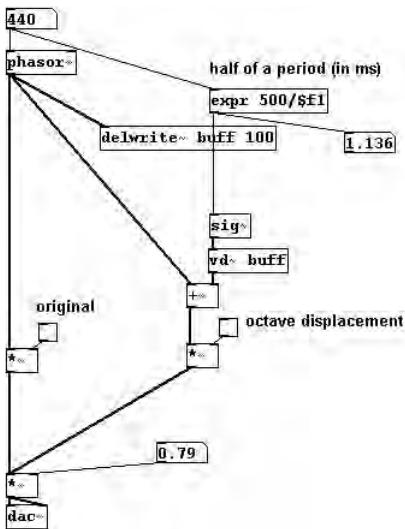
If you know the frequency of a signal's fundamental, you can construct an octave doubler as follows: Let's take a wave...



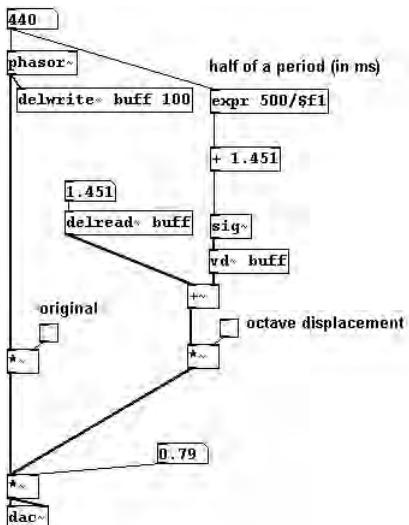
...and this signal delayed by half the length of one period...



...adding them together gives you 0 (= cancellation). If you delay a periodic signal by the half the duration of one period and add it to the original, the fundamental tone (and all odd partials) is cancelled out. That would look like this:



But it doesn't quite work like that. You have to remember that Pd processes all audio data in blocks of 64 samples (unless you change the setting), because it is more efficient than individually processing each sample (cf. 3.1.1.3.2). With the above patch, you'd get a delay of 1,136 milliseconds, or 50 samples. You could alleviate this problem by using a buffer with a one-block delay (64 samples = 1,451 ms) to read the original; the same goes for the delay offset:

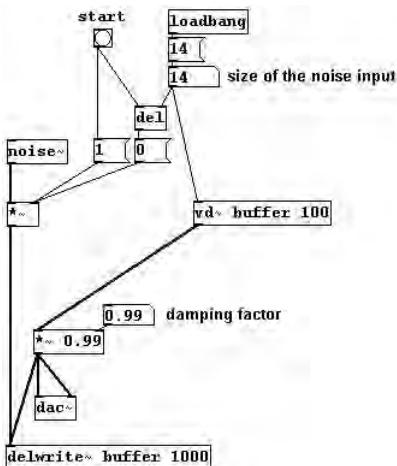


patches/3-4-2-9-oktavedoubler.pd

#### ► 3.4.2.10 Karplus-Strong algorithm

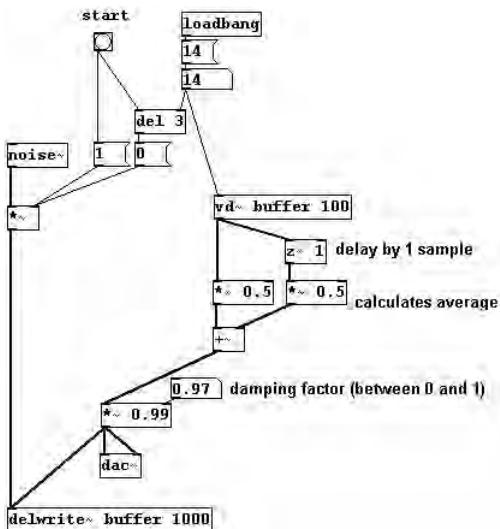
A special use of looping is the Karplus-Strong algorithm. It is one of the first examples of physical modeling synthesis, a process that attempts to replicate what occurs when a physical material vibrates. In our example, the physical model is a plucked string. When it is plucked, a string first vibrates chaotically then adjusts itself to the length of the string. It also loses energy, i.e., the vibration dies away. This can be reconstructed mathematically by taking an excerpt of white noise and playing it back periodically again and again by writing it to and reading it from a buffer:

patches/3-4-2-10-karplus-strong1.pd

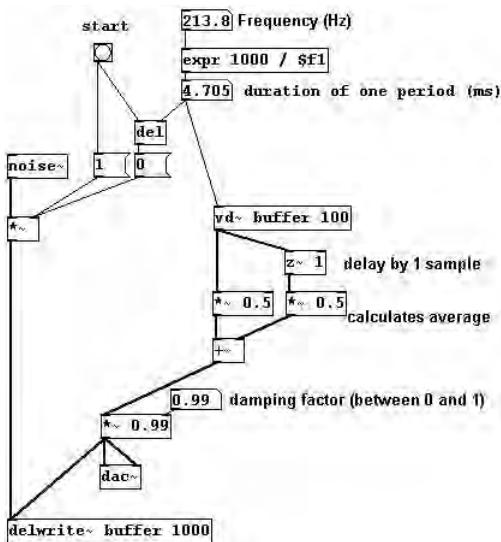


The string effect can be enhanced if the material you start with vibrates more and more ‘softly’. This works with by calculating the average: the average of every two samples is taken and this result is written to the buffer in place of the original values. The vibration becomes less and less ‘angular’. Use the object “z~” (Pd-extended) to set the delay to one sample; enter the number of samples as the argument:

patches/3-4-2-10-karplus-strong2.pd



The tone is different every time. This is because “noise~” produces random numbers, which are naturally different every time. We could add the calculation for the resulting frequencies:



patches/3-4-2-10-karplus-strong3.pd

### ► 3.4.2.11 More exercises

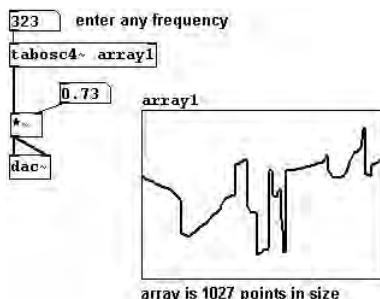
- a) Build a record function into the sample player.
- b) Create a patch for reverb or a texture with different delay times for the input signal, e.g., with multiples of the Fibonacci series (in which the next number is always the sum of the previous two: 0 1 1 2 3 5 8 13).
- c) Use different Karplus-Strong sounds to make textures of varying densities.
- d) Apply a comb filter to patches presented in the previous sections.

### ◆ 3.4.3 APPENDIX

#### ► 3.4.3.1 Array oscillator

One way to simplify the combination of “tabread~” and a multiplied “phasor~” signal is “tabosc4~”. This reads out an array for the frequency you enter. One limitation of this method is that the size of the array must be a power of two (e.g., 128, 512, 1024) plus three points (here, 1027 = 1024 + 3).

patches/3-4-3-1-arrayoscillator.pd

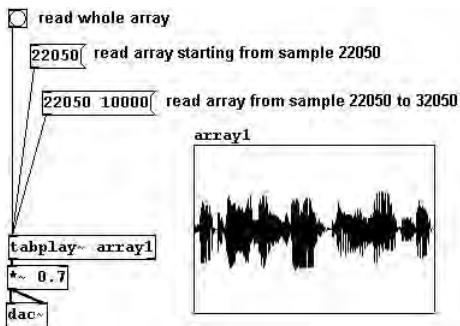


This touches on ”wave shaping”, a topic that will be further explained in chapter 3.5. You can draw any wave into an array using the mouse.

#### ► 3.4.3.2 Array playback

Yet another simplification is “tabplay~”; it simply plays an array back at the original speed (when banged). Conveniently, you can set the start and end points for playback (starting point and duration in samples):

patches/3-4-3-2-simply-play-array.pd

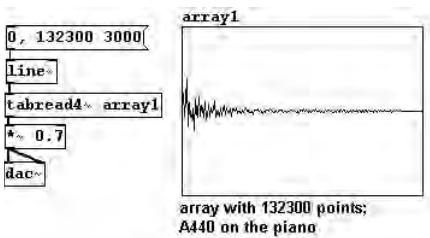


#### ► 3.4.3.3 Playing back an array in a block

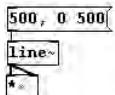
The “tabwrite~” and “tabread~” objects also have another special form: “tabsend~” and “tabreceive~”. They write/read an array in sync with the “blocks” (3.1.1.3.2). “tabwrite~” writes each block in an array (of course, these must be the size of a block, which is set to 64 samples by default in Pd). “tabreceive~” reads the array in every block. We will return to this later in the chapter on FFT (3.8).

#### ► 3.4.3.4 Glissandi of samples

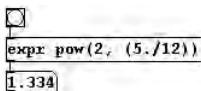
You know that you can play back an array at normal pitch or an octave higher, etc. But what if you want a glissando from the octave to the original pitch? For this, you'll need to subdivide into “main indicator” and “addition”. The “main indicator” run at normal speed over the array. Let's use an array with 132300 points as an example, which equals 3000 milliseconds:



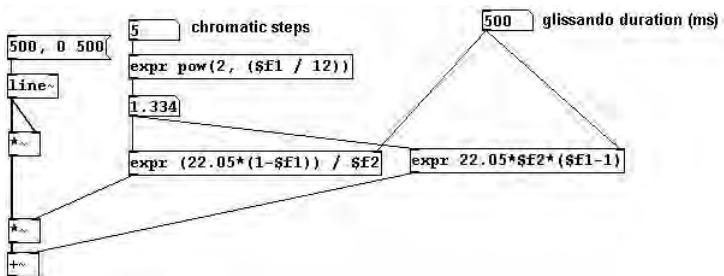
Then comes the “addition”, which is what makes the glissando. Let's use a glissando that begins five chromatic steps above the original pitch and returns to the original pitch in 500 milliseconds. You need to make “line~” of this in reverse, then square the values:



Above this, you have to determine the factor for the frequencies of the five chromatic steps (cf. 3.1.1.4.3)...

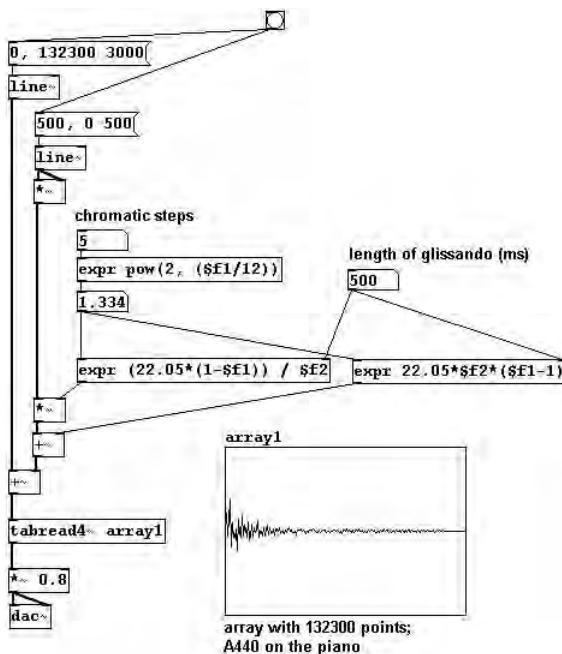


...and finally conduct the following calculation:



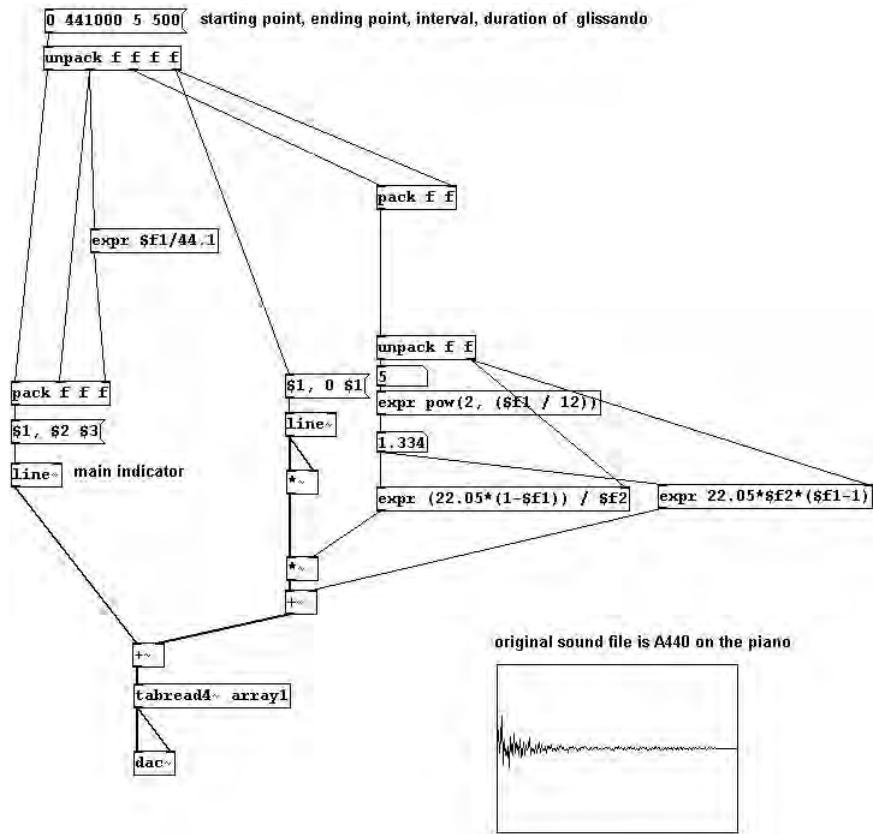
This is the “addition” that is added to the “main indicator”:

patches/3-4-3-4-sample-glissando1.pd



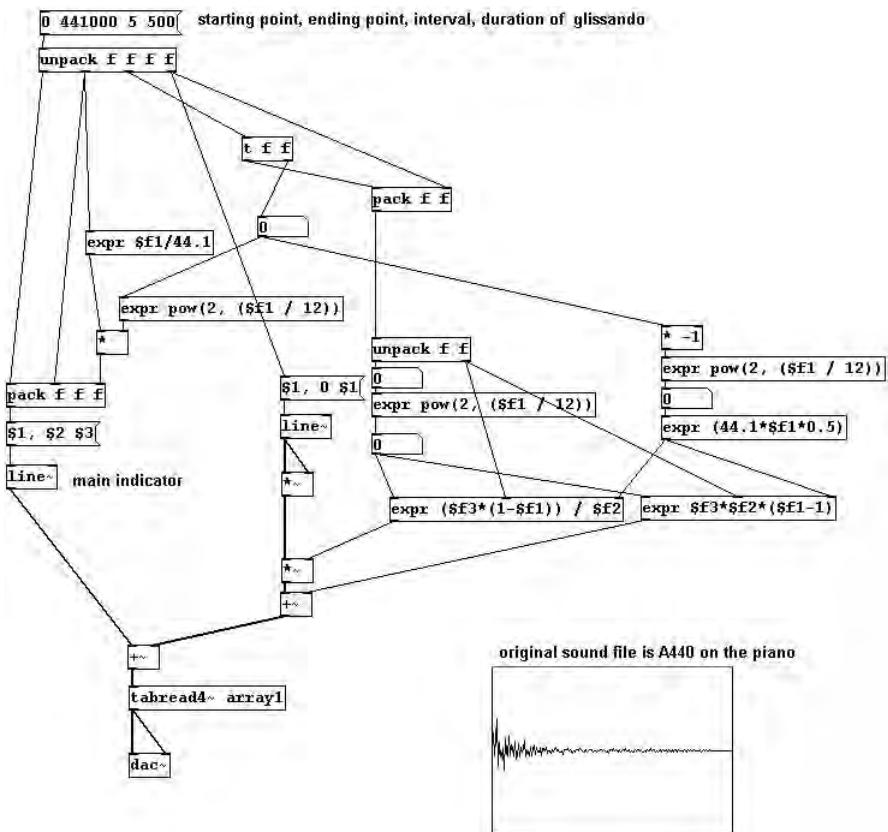
You can use any glissandi to the target pitch that you want; even negative values are possible:

patches/3-4-3-4-sample-glissando2.pd



Conversely, to move away from the original tone:

patches/3-4-3-4-sample-glissando3.pd



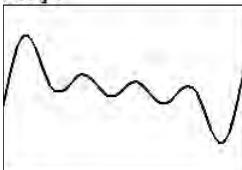
#### ► 3.4.3.5 Additive synthesis with array

As a special function in Pd, you can create a sum of sine tones in an array—i.e., additive synthesis as described in Chapter 3.2. This is accomplished using the message “sinesum”. The first argument is the (new) array size (should be a power of 2; three points will be added to this number automatically to ensure optimum connection of the beginning and ending of a phase) and also the volume factors for any number of partials:

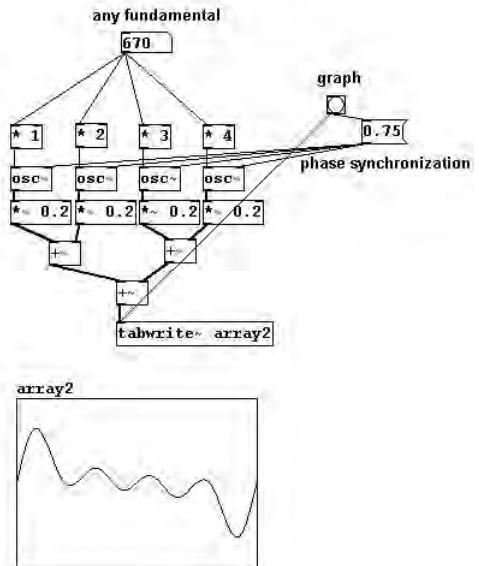
`patches/3-4-3-5-sinesum.pd`

```
; array1 sinesum 64 0.2 0.2 0.2 0.2
```

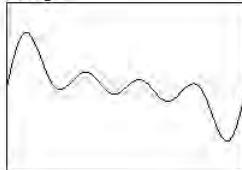
array1



corresponds to:



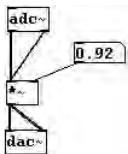
array2



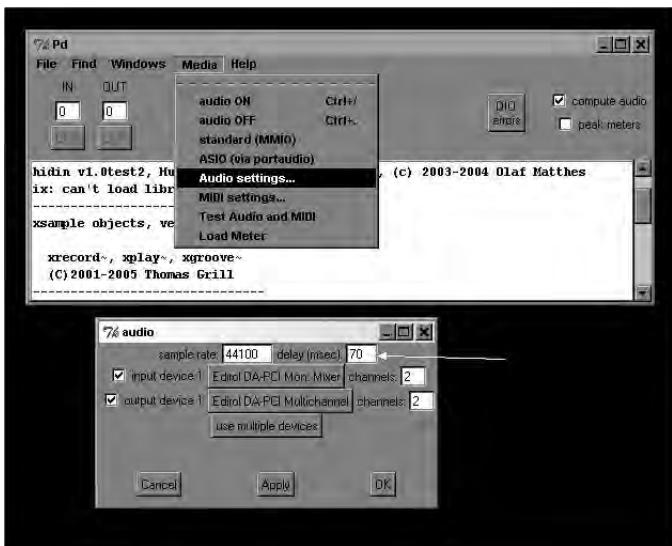
Instead of sine waves, you could also use “cosinesum” to work with cosine waves.

#### ► 3.4.3.6 Latency

Audio delay sometimes occurs when you don't want it to. You can even hear it when you have a microphone connected to a speaker and make an extremely short noise into the microphone, snapping your fingers, for example:



The sound card and especially the operating system determine the length of this “latency”. Ideally the latency is so short (under 5 ms) that the human ear cannot perceive the delay. This requires a fast computer processor, a good sound card, and an appropriate operating system. You can set the latency under **Media** → **Audio settings**:



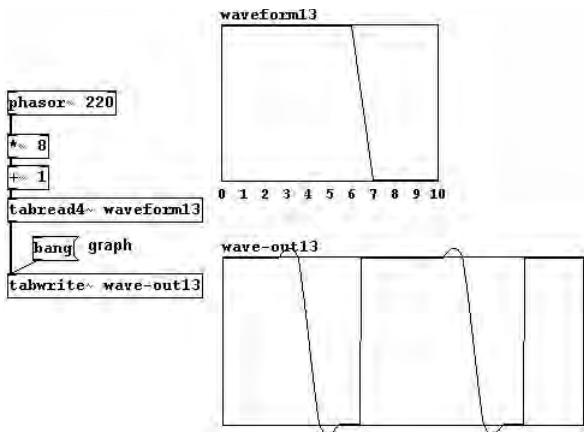
In Microsoft Windows, you cannot at the present time (June 2008) achieve a latency of less than 50 ms without causing errors.

◆ 3.4.4 For especially interested

► 3.4.4.1 4-point interpolation

In this example you can see how “tabread4~” interpolation works:

[patches/3-4-4-1-four-point-interpolation.pd](#)

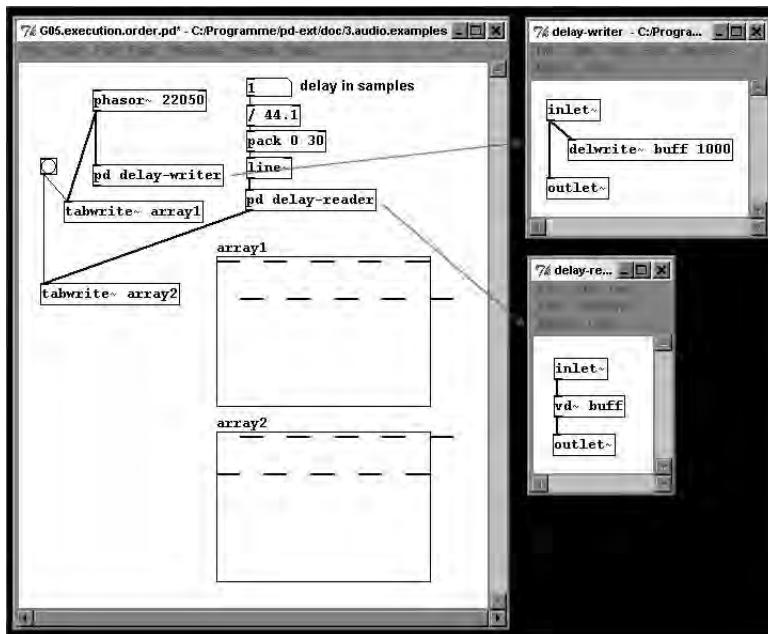


The jump from 1 to -1 is ‘softened’ by a kind of sinusoidal interpolation. As the name implies, four points are used and altered: namely the two directly in front of and the two directly behind the interval that you want to interpolate.

#### ► 3.4.4.2 Sample-wise delay

One way to delay something by a certain number of samples with “delread~” and “vd~” is by using a subpatch (otherwise the problem of block size, described previously in relation to octave displacement):

patches/3-4-4-2-samplewise-delay.pd



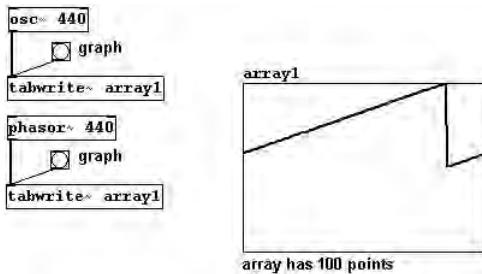
## 3.5 Wave shaping

### ◆ 3.5.1 THEORY

#### ► 3.5.1.1 Waveforms

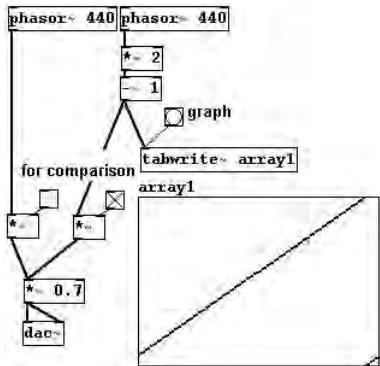
In 3.1.1.1.2 you learned about different waveforms (sine, sawtooth, triangle, square, and pulse). Pd has objects for two of these, namely “`osc~`” for sine waves and “`phasor~`” for sawtooth. You can use an array to display the waveforms:

patches/3-5-1-1-waveform-graph.pd



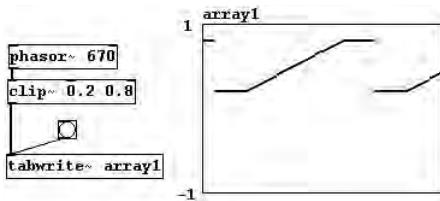
N.B.: The sawtooth of the “`phasor~`” object always goes from 0 to 1; it never goes into the negative range. You could make it stronger, however, by performing a small calculation:

patches/3-5-1-1-strong-phasor.pd



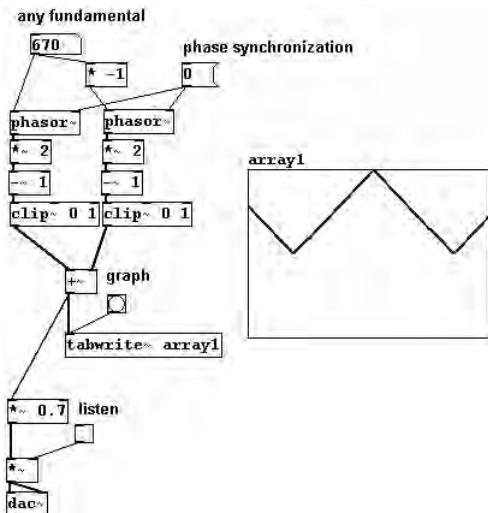
You can also create other waveforms by attaching a few operations to “`phasor~`”. To accomplish this, you’ll need to use a new object: “`clip~`”, which cuts off everything outside the indicated range. As the arguments, enter two numbers for the lower and upper limits; numbers outside of these limits will be ‘clipped’:

patches/3-5-1-1-other-waveforms.pd



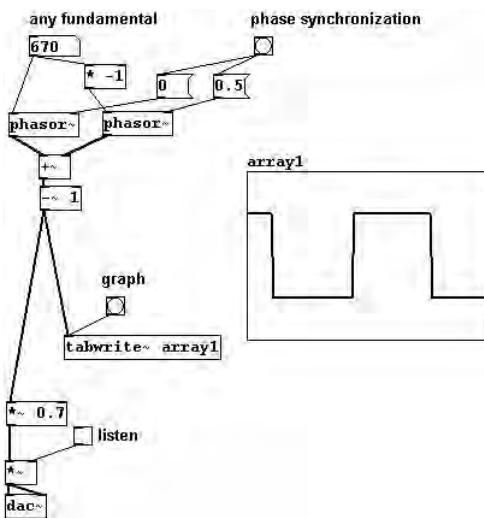
Now on to the Triangle wave:

patches/3-5-1-1-triangel.pd



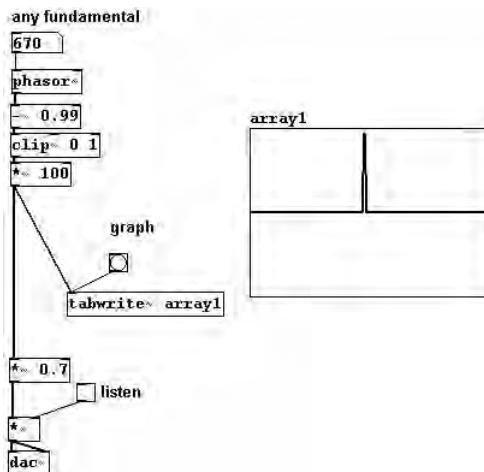
Square wave:

patches/3-5-1-1-square.pd



Pulse:

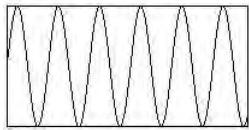
patches/3-5-1-1-pulse.pd



These are all standard waveforms that exhibit certain characteristics:

Sine: a single ton without any overtones

wave:

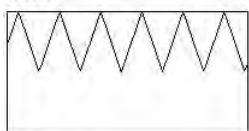


spectrum:



Triangle: like a sine wave, except with the odd partials as well

wave:

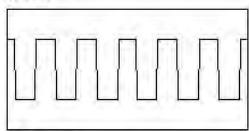


spectrum:

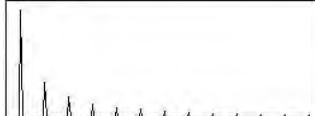


Square: only the odd partials

wave:

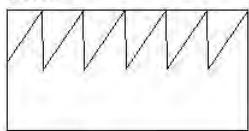


spectrum:



Sawtooth: all partials

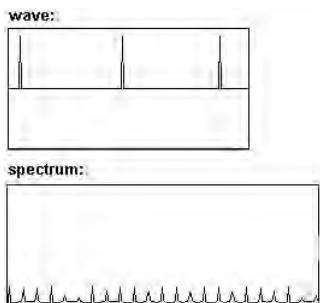
wave:



spectrum:

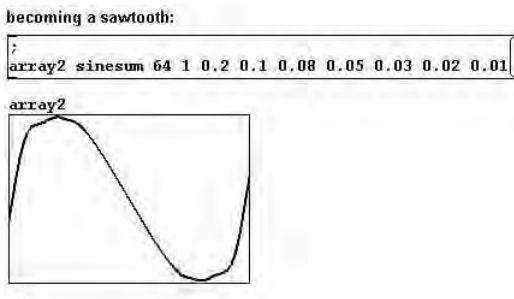
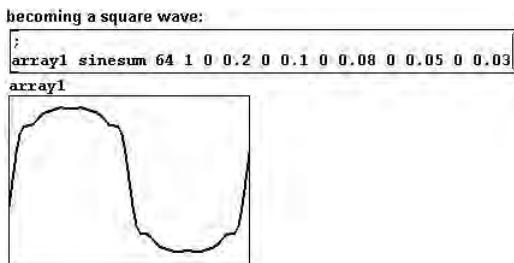


Pulse: all partials present at nearly equal intensities



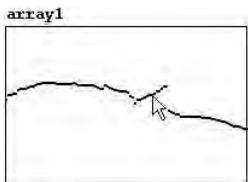
You can observe that symmetrical waveforms exhibit only the odd partials (within each period, exactly two periods of each subsequent odd partial fit; that's what is meant by symmetrical), while asymmetrical waveforms always exhibit the even partials as well. These waveforms can also be approximated using additive synthesis:

patches/3-5-1-1-waveform-fourier.pd



etc.

There is unlimited room for experimentation here and that's one way to arrive at new sounds. You can also draw your own waveforms directly in an array. To do this, you must be in execute mode and move the mouse to a line in the array. The cursor arrow changes direction:

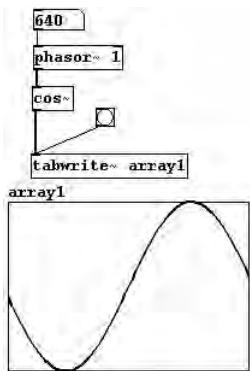


While holding the mouse button, move the mouse to draw your own waveform.

This is, however, somewhat tedious and ‘inelegant’. Let’s examine then the theory of ‘wave shaping’:

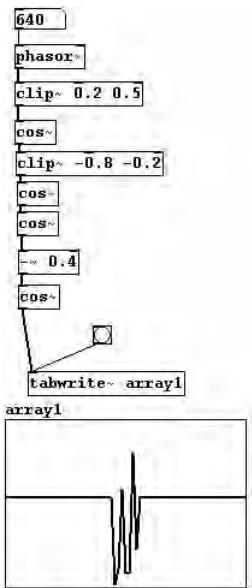
#### ► 3.5.1.2 Transfer functions

A linear function undergoes what’s called a “transfer function”. For linear functions, you can use the “phasor~”, which always goes from 0 to 1. You could make a cosine wave, for example, using the “cos~” object, which calculates a cosine function:



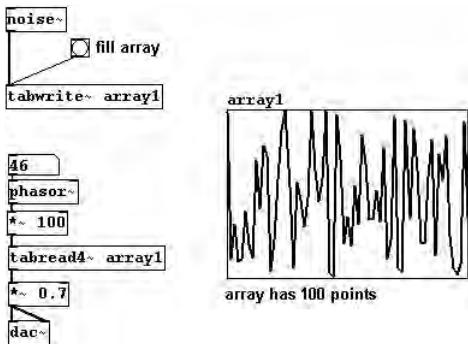
You can make all kinds of transfers this way; here’s another example:

[patches/3-5-1-2-transferfunction.pd](#)



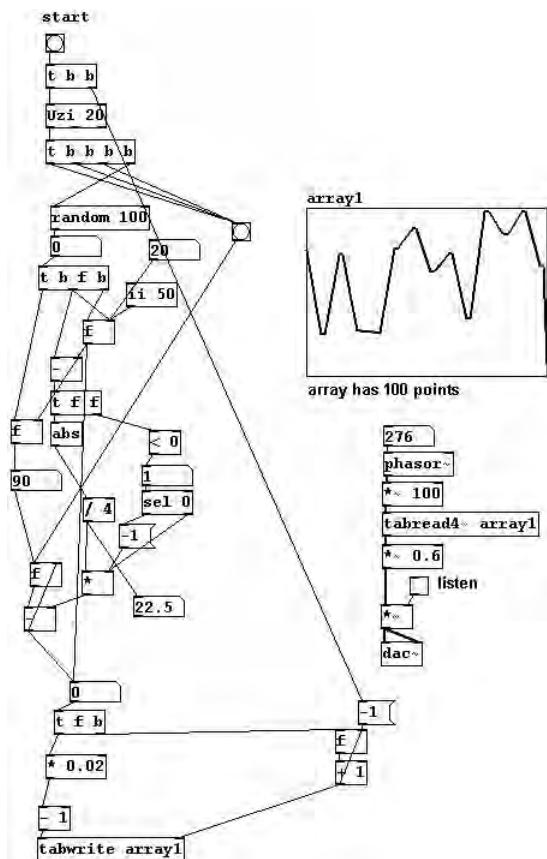
### ► 3.5.1.3 (Controlled) Random waveforms

You could also record noise into an array and then read it out periodically—i.e., read the same thing out again and again (cf. Karplus-Strong). If you do this more than 20 times a second, you'll hear a pitch:



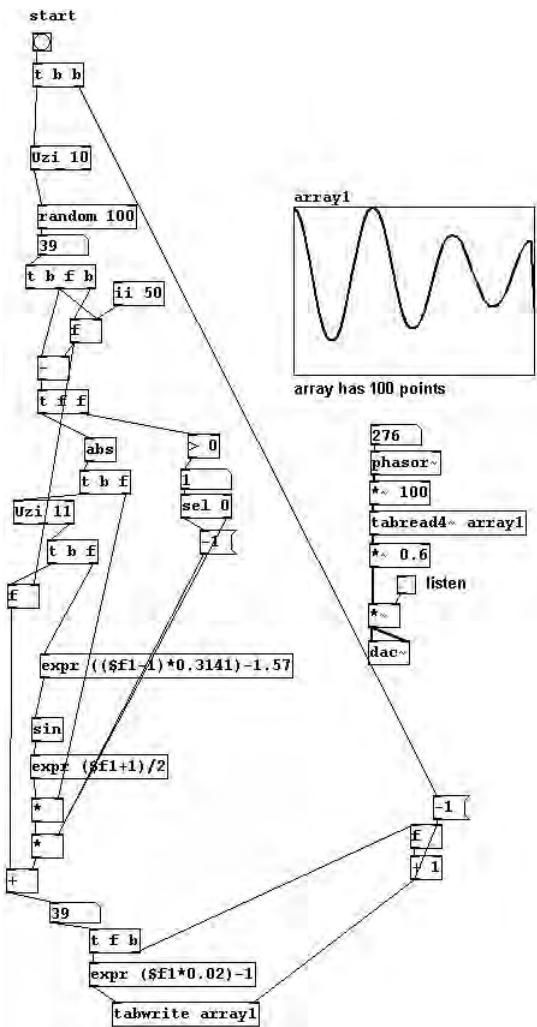
The spectrum of this sound is naturally somewhat unpredictable. Every time you fill the array with random numbers with the “noise~” object, you get a new wave with new characteristics.

But there is still somewhat of a system at work. You could, for example, interpolate all the clicks (large jumps in the waveform) to achieve a smoother resultant waveform. Let's generate random points using linear interpolation (“Uzi” (Pd-extended) generates the number of bangs specified in its argument and sends them as fast as possible):



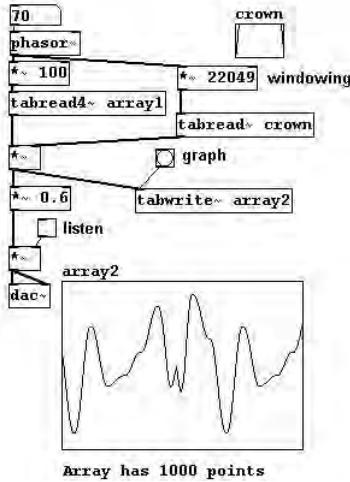
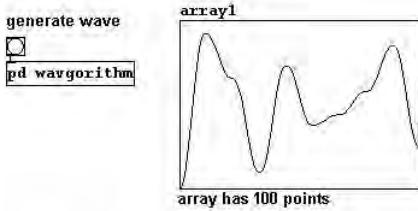
In this example, four points are always used for interpolations.

The result will be even softer if you use a sinusoid interpolation instead of a linear one. Here, ten points are used for interpolations:



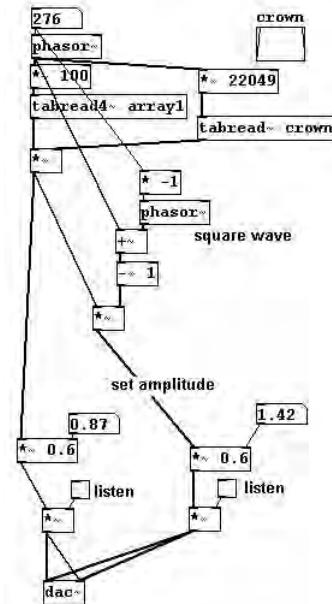
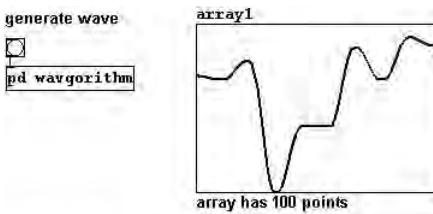
Now the connection from the end of the period to its beginning has to be interpolated as well. We'll use a windowing for this and program the calculation in a subpatch.

[patches/3-5-1-3-wavgorithm+sin+fenster.pd](#)



This way, the membrane is at 0 at the beginning and end of each period. The result could be even smoother if it were “windowed” with the Hanning window (3.9.4.1).

In addition, transfer functions using known waveforms—e.g., a square wave, which would intensify the odd partials—are also possible.

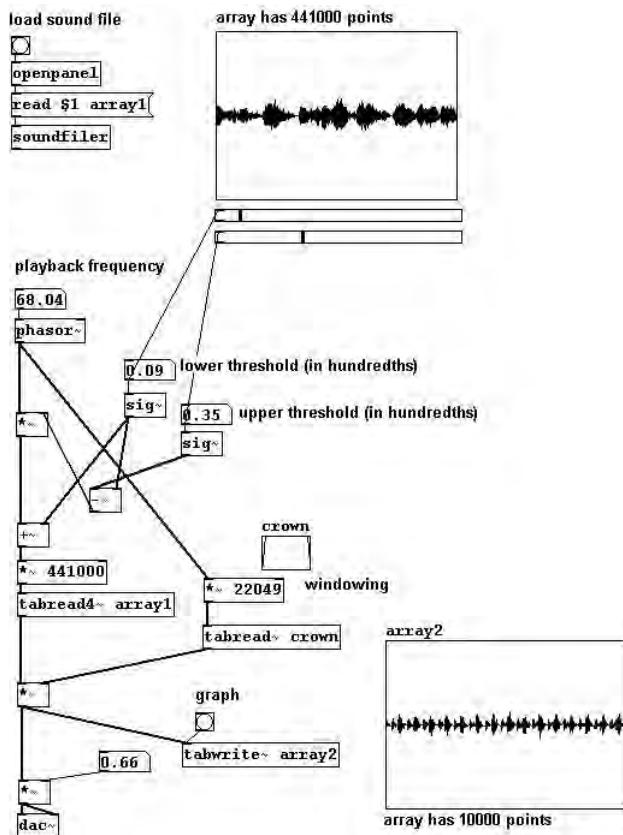


And so on. In these last few examples, the wave has to be created on the control level; in contrast to the first examples that used transfer functions, we cannot change these “live”.

#### ► 3.5.1.4 Wave stealing

A final technique that might be considered wave-shaping synthesis is “wave-stealing”. This involves taking a small section of known pieces of music...

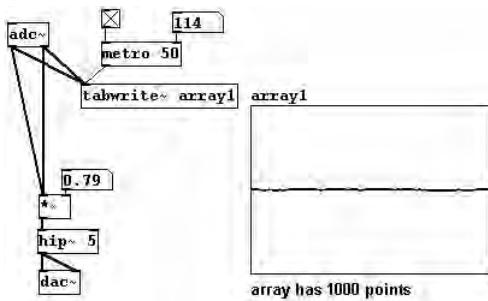
patches/3-5-1-4-wavestealing.pd



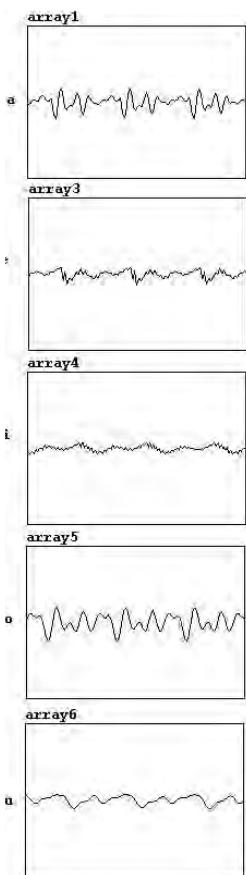
◆ 3.5.2 APPLICATIONS

► 3.5.2.1 Singing waveforms

With the following patch, it is possible to record waveforms with a microphone to sing waveforms.

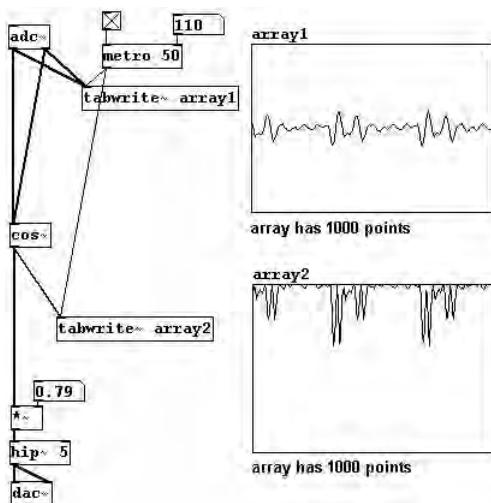


The vowels (German pronunciation) look something like this:



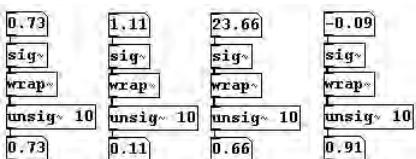
### ► 3.5.2.2 Transfers

And this input signal could naturally also be sent through a transfer function:



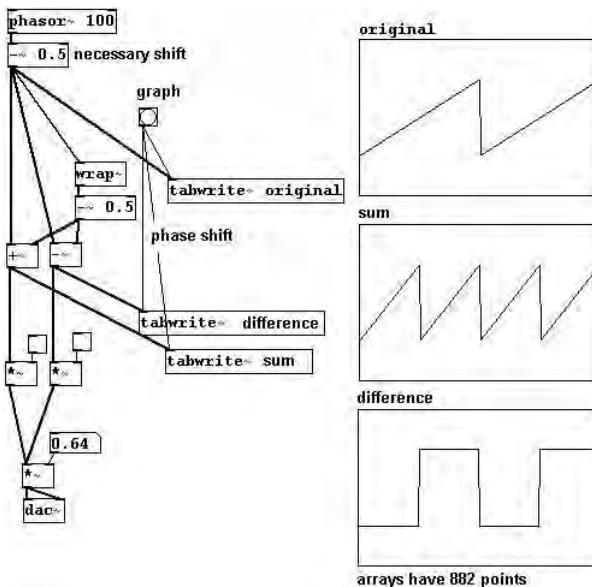
### ► 3.5.2.3 Even / odd partials

You can also divide a sawtooth wave into even and odd partials. To accomplish this, you'll need to use “wrap~”. It calculates the difference between the input number and the nearest integer below it (the absolute value is taken, so that the result is always positive). A few short examples:



And now for the sawtooth division: the “wrap~” object is used to phase shift the sawtooth wave; this is then added to and subtracted from the original signal, resulting in a sawtooth wave with twice the frequency and a square wave.

[patches/3-5-2-3-even-odd-partials.pd](#)



#### ► 3.5.2.4 More exercises

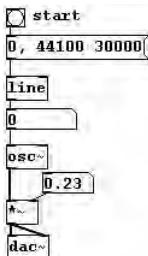
- Create a wave that changes constantly.
- Create a patch in which the interpolation and array points used for (controlled) random waveforms are variable in number.

#### ◆ 3.5.3 APPENDIX

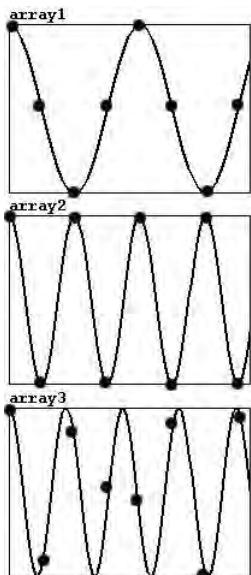
##### ► 3.5.3.1 Foldover

At this point, a particularly thorny problem in digital sound processing must be addressed: foldover. Let's first examine this situation:

patches/3-5-3-1-foldover1.pd



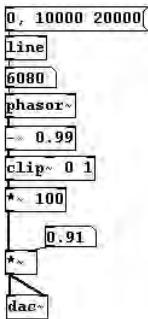
What happens? After 22050 Hz, the direction changes until it reaches 44100 Hz, at which point you get a pitch with a frequency of 0 Hz (after this, the pitch would go up again). The reason for this is that a sample rate with of 44100 Hz can produce a wave of 22050 Hz maximum (cf. 3.1.1.3.1). Moreover, there are some typical reading errors. Let's take a look at three waves with different frequencies: the top has a frequency of 11025 Hz, the middle 22050, and the bottom somewhat more than 22050. The markings stand for reference points (for the samples), which have a constant speed of 44100 per second.



Each period in a wave with 11025 Hz can be represented with four points (of course, the characteristic sine wave shape is lost). 22050 Hz is the highest frequency that can be correctly represented, since Nyquist's Theorem requires at least two points per period. Errors will occur with frequencies higher than this; not every period will be captured and the points of measurement will actually record a lower frequency instead of a higher one.

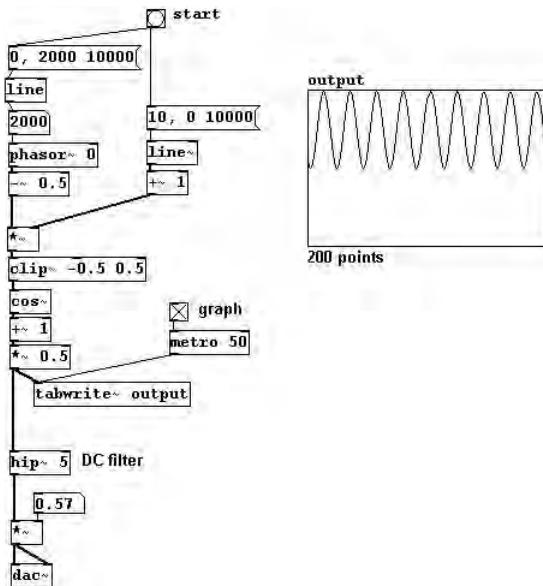
The problem is much more pronounced for waveforms that exhibit overtones, e.g., with a pulse:

[patches/3-5-3-1-foldover2.pd](#)



Here the effect can be noticed much earlier: after 700 Hz some overtones are inaccurately captured. This is because the pulse waveform practically consists of just a single line, which is quickly ‘missed’. A solution to this problem is to begin with a pulse waveform, but to broaden the wave as the frequency increases, so that you end up with a sine wave at the end:

`patches/3-5-3-1-foldover3.pd`



#### ◆ 3.5.4 FOR THOSE ESPECIALLY INTERESTED

##### ► 3.5.4.1 GENDY

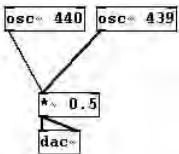
Iannis Xenakis in his later years developed a process for wave generation called GENDY. He ‘composed’ nothing but waveforms and derivatives of them, e.g., in his tape piece “Gendy 3”.

## 3.6 Modulation synthesis

### ◆ 3.6.1 THEORY

#### ► 3.6.1.1 Ring modulation

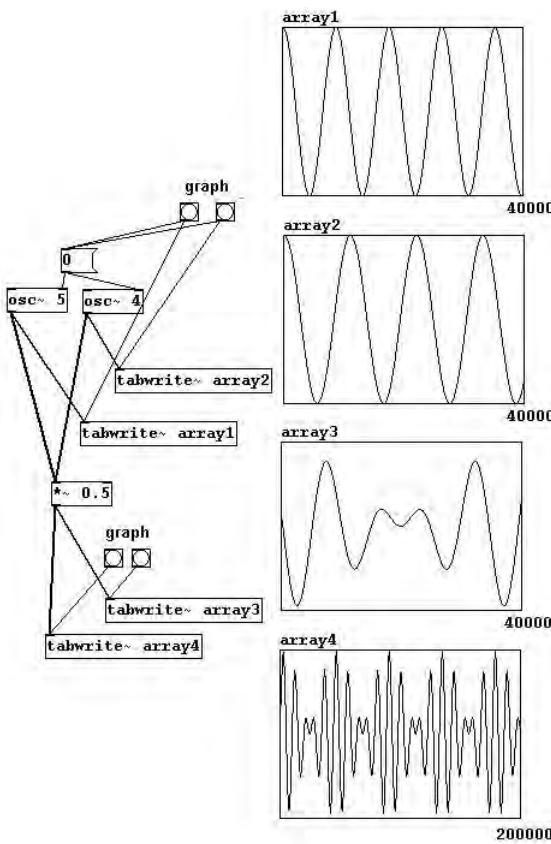
Let's first look at this phenomenon:



If you listen to two sine tones very close to each other, you hear fluctuating wave cancellations. This is due to the interaction of two almost (but only almost!) identical waves. This phenomenon is called *beating*. The speed, or rhythm, of the beating is exactly equal to the difference between the two frequencies—in this example:  $440 - 439 = 1$  Hz.

Let's take a look at a simplified example using oscillators with 4 and 5 Hz:

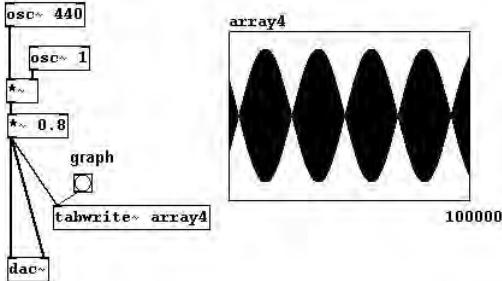
[patches/3-6-1-1-ringmodulation1.pd](#)



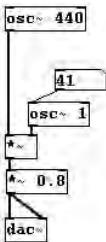
The waves alternate between summations (never forget: waves add themselves together!) and cancellations. In array1 and array2 you can see a part of the original wave, in array3 the summation, and in array4 the summation again over a longer period of time. The result is a pulsing rise and fall in volume. (For the human ear to perceive two different tones, their frequencies differ by about five cents.)

You can also first determine, precisely, the rhythm of these amplitude fluctuations. As you saw in array4 in the previous diagram, the amplitude has a sinusoid shape. You could therefore simply use an oscillator to determine the amplitude:

[patches/3-6-1-1-ringmodulation2.pd](#)



(The reason this array is so dark in comparison to the previous one is because it uses much higher frequencies.) The resulting wave corresponds to the summation of two waves. If you raise the frequency of the modulating amplitude higher and higher...



...you hear two frequencies that move away from each other symmetrically—one up, one down—by an interval equal to the distance of the amplitude from the middle axis, i.e., the initial amplitude frequency. This process is an example of amplitude modulation, called ‘ring modulation’ because of its symmetrical nature. If the initial frequency is 440 Hz and the amplitude frequency is 100 Hz, you’ll hear two tones: one at 340 Hz and one at 540 Hz.

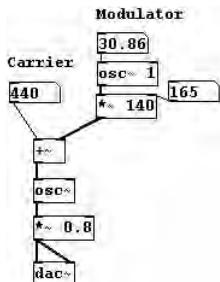
### ► 3.6.1.2 Frequency modulation

You could also use an oscillator to modulate a sine wave’s frequency. This is called frequency modulation:



One oscillator is “carrier” and the other is “modulator”. Using a low frequency with the “modulator” will result in vibrato. Beginning at 20 Hz, increasing this frequency will result in a more and more complex multiphonic:

`patches/3-6-1-2-frequencymodulation.pd`



The resulting wave is the summation of many different sine waves; the carrier frequency lies in the middle while the other tones lie above and below it at distances determined by the modulation frequency.



When the modulation amplitude rises, the amplitudes of the additional frequencies also rise. However, this increase is difficult to formulate mathematically.

A special situation arises when the modulation frequency is a whole number multiple of the carrier frequency (i.e., 1x, 2x, 3x, 4x, 5x, 6x, etc.). The other tones above the carrier frequency would also be whole number multiples of the carrier frequency—i.e., its overtones.

Furthermore negative frequencies are mirrored above in the positive range. In the special situation mentioned in the previous paragraph, these are covered by “normal” frequencies. Let’s say you have a carrier frequency of 200 Hz and a modulator frequency of 100 Hz; coverings occur starting with the third undertone (which is also 100 Hz and the following 200, 300, etc.), which result in amplifications and suppressions according to phase length.

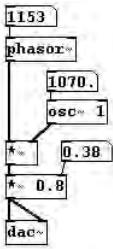
The advantage of FM synthesis over additive synthesis (the simple addition of sine waves) is that you only need two oscillators to make a rich and complex sound (you just have to change the frequency and especially the amplitude of the modulator!). A typical FM synthesis sound is a ‘disharmonic spectrum’, i.e., a quasi-spectrum above the fundamental with distorted overtones that are not whole number multiples of the fundamental. Some metallic instruments, like bells and gongs, exhibit similar spectra; sounds made using FM synthesis often have a ‘metallic’ timbre for this reason.

## ◆ 3.6.2 APPLICATIONS

### ► 3.6.2.1 More sonically complex ring modulation

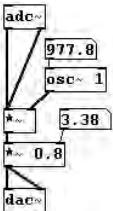
Ring modulations used with overtone-rich sounds are naturally much more complex and rich:

patches/3-6-2-1-ringmodulation3.pd



### ► 3.6.2.2 Live ring modulation

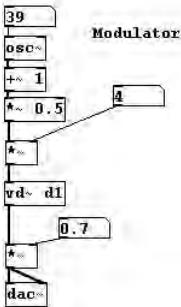
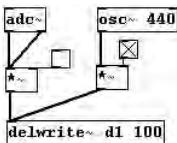
patches/3-6-2-2-ringmodulation-live.pd



### ► 3.6.2.3 Live frequency modulation

To use frequency modulation in live settings, you have to use variable delay in order to be able to change the frequency:

patches/3-6-2-3-frequencymodulation-live.pd



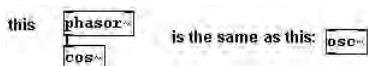
#### ► 3.6.2.4 More exercises

Combine everything you've learned up to now.

#### ◆ 3.6.3 APPENDIX

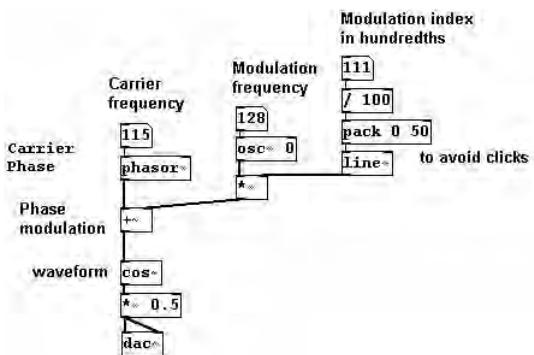
##### ► 3.6.3.1 Phase modulation

Frequency modulation is also called phase modulation and can also be programmed in this form. To accomplish this, the carrier oscillator must be divided into phase processing and waveform processing. Here's how that works:



And the phase modulation looks like this:

[patches/3-6-3-1-phasemodulation.pd](#)



## 3.7 Granular synthesis

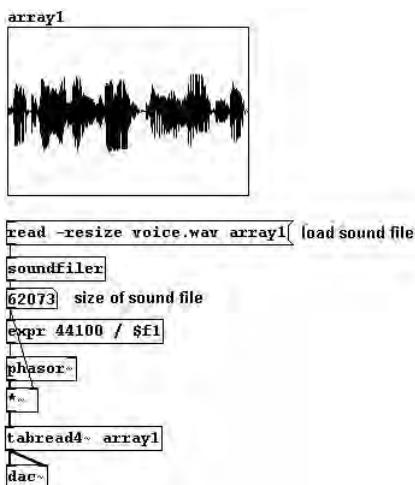
### ◆ 3.7.1 THEORY

#### ► 3.7.1.1 Theory of granular synthesis

With regard to sampling (3.3) you learned how to change the speed of an existing sound in an array, but this also resulted in a change of pitch. One way to decouple these parameters, is by using granular synthesis. The idea of granular synthesis is that a sound is sampled at the original speed, but it is played at a different speed from each sample point.

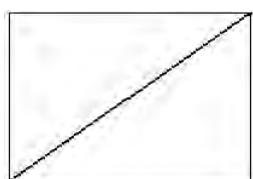
You have an “indicator” that moves across the array at normal speed:

patches/3-7-1-1-granular-theory1.pd

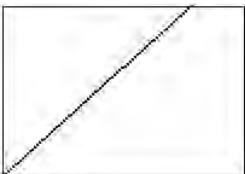


Only at certain intervals do we get information about the indicator's present position; when this information is received, the array is played back from that point, albeit at a different speed.

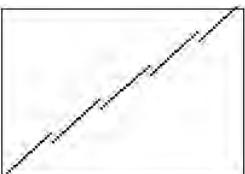
To understand this better, let's say this is the normal playback speed:



...and this is a speed that is ‘too fast’:

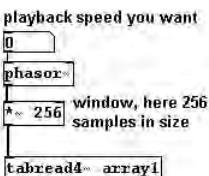


...then granular synthesis does this:



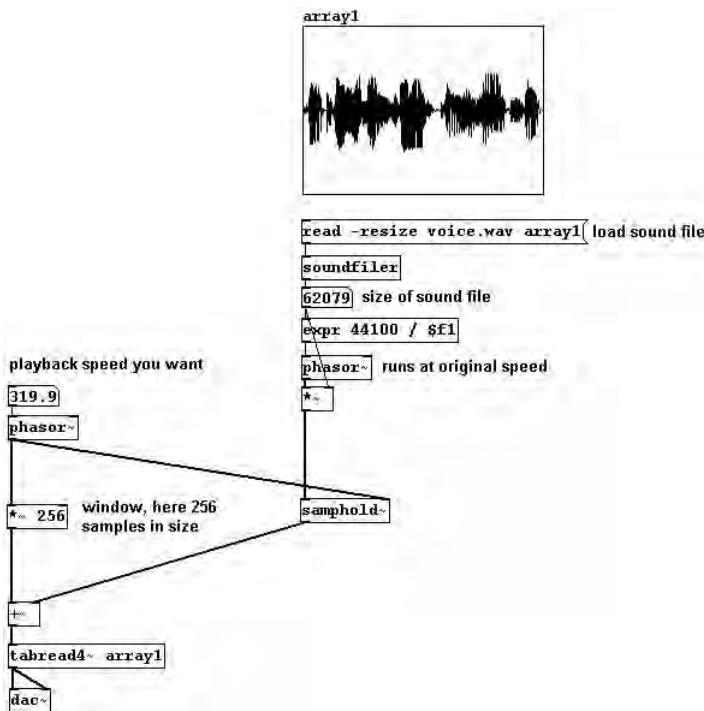
Though playback is ‘too fast’ (or ‘too slow’), it always begins at a point that corresponds to the initial speed. These individual chunks are called “grains”; their size is referred to either as “grain size” or “window size”. These “grains” are so tiny and used in such large quantities, that they are not heard individually, but rather as a continuous whole. That’s the magic behind granular synthesis.

Every individual “grain” is played back like this:



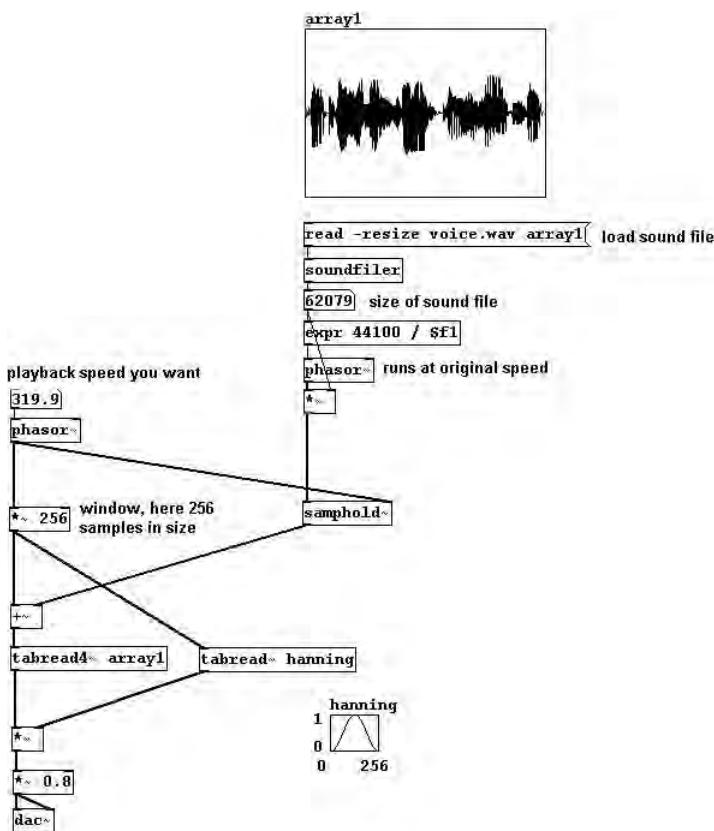
After a grain is played, there is a jump to the next position; this position is taken from the current position of the “main indicator”. There is a special object to accomplish this: “samphold~”. It works like “spigot”, only on the signal level. Both the left and right inlets receive a signal. When there is descending step in the right inlet, “samphold~” immediately sends the sample currently in the left inlet and repeats this until the value in the right inlet is lower than the preceding one. This somewhat strange setting makes sense if the right input is a “phasor~”. It receives only once—right at the end of a period—a descending step. A grain could be read out this way and the offset could be added to the end of it:

[patches/3-7-1-1-granular-theory2.pd](#)



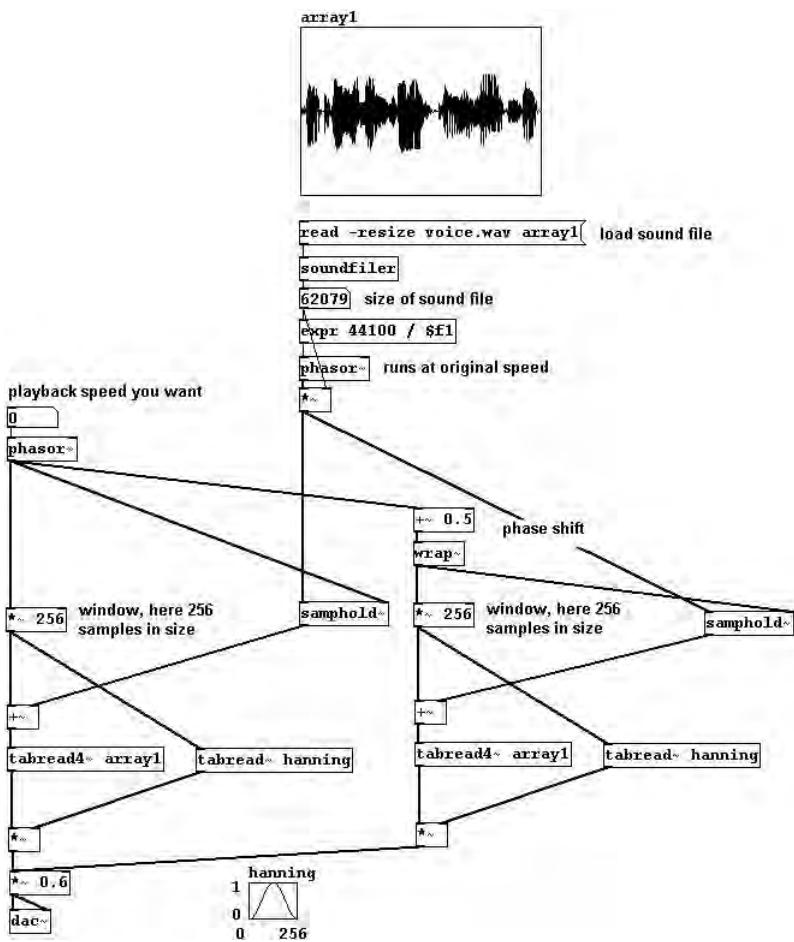
This way, the whole thing sounds higher, but with the same duration as the original. If you ‘look’ closely, it’s clear that this could lead to complications. If playback from one sample to the next is faster than the indicator’s speed (which runs at the original speed), then we’ll overshoot a sample and then return to it (thus repeating it) the next time “samphold~” is triggered. Conversely, if the “grains” are played back more slowly than the indicator moves, then some parts will be omitted. But as long as the original and the playback speeds do not diverge too dramatically, this is not (terribly) noticeable. To rectify this, some improvements can be made. First, you could use a Hanning window to suppress the clicks that result with every jump to a new value:

[patches/3-7-1-1-granular-theory3.pd](#)



The resultant gaps can be filled by using a second grain-reader, shifted by half a period:

[patches/3-7-1-1-granular-theory4.pd](#)

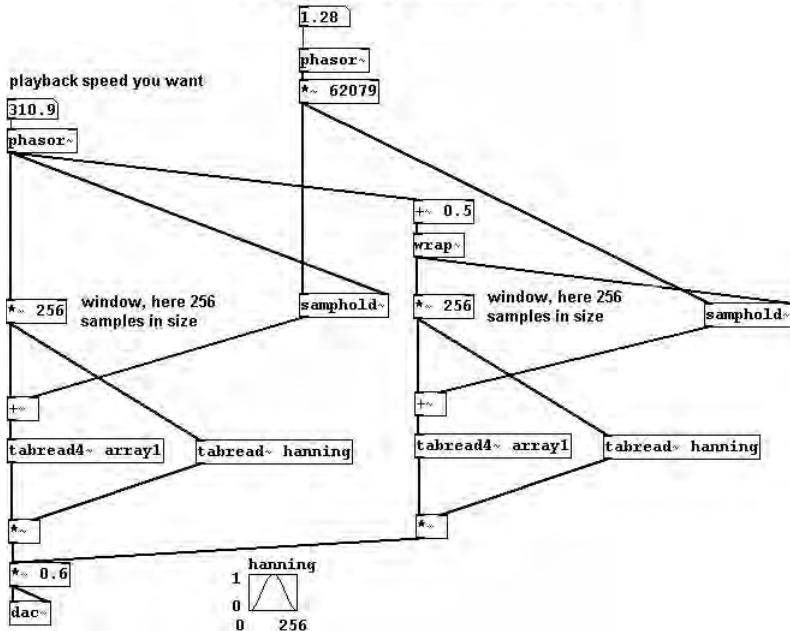


The nice thing about granular synthesis is that, in addition to the ability to change pitch without affecting speed, you can change speed without affecting pitch:

[patches/3-7-1-1-granular-theory5.pd](#)



here you can once again choose your own settings

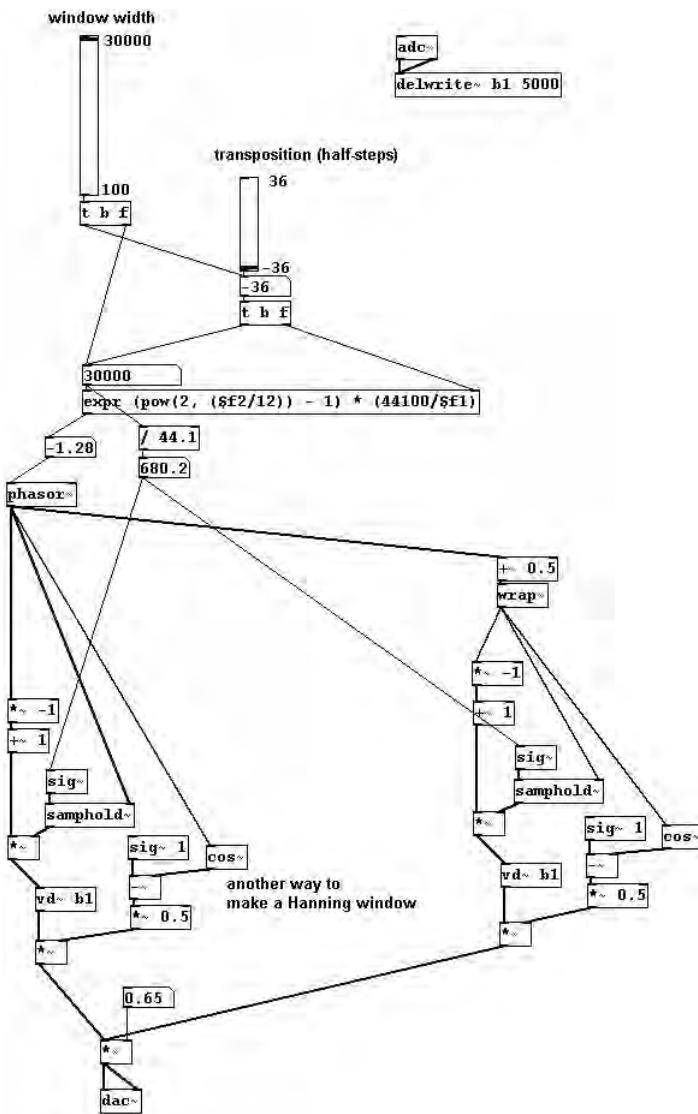


### ◆ 3.7.2 APPLICATIONS

#### ► 3.7.2.1 Live granular synthesis

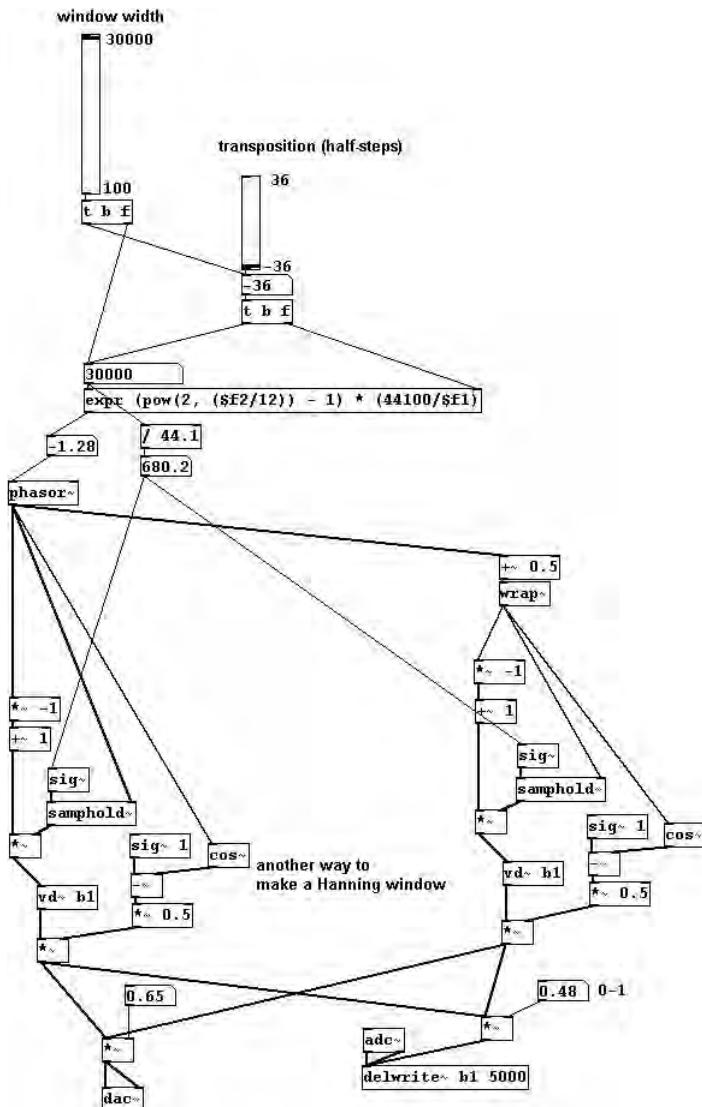
For use in live performance, you'll again need to use variable delays:

[patches/3-7-2-1-granular-live.pd](#)



### ► 3.7.2.2 Live with feedback

patches/3-7-2-2-granular-live-feedback.pd



### ► 3.7.2.3 More exercises

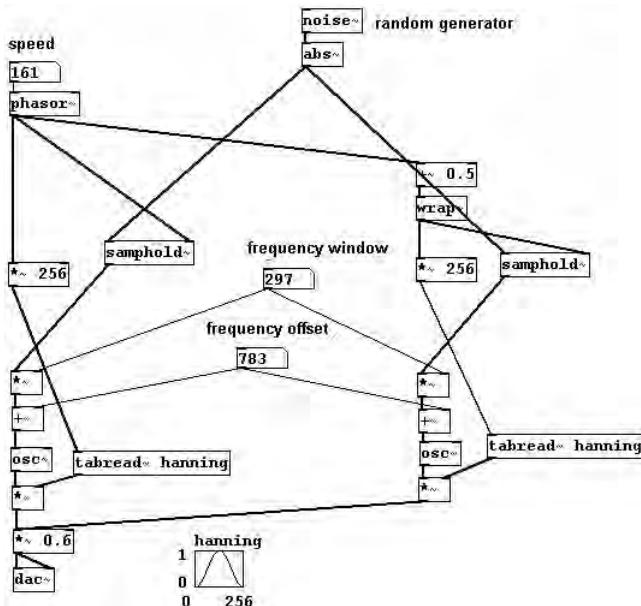
Create four readers each with a variable window size. Experiment!

◆ 3.7.3 Appendix

► 3.7.3.1 Granular technique as a synthesizer

Granular synthesis can also be used as a synthesizer for pitch clouds, most conveniently using a random generator:

patches/3-7-3-1-granularsynthesizer.pd



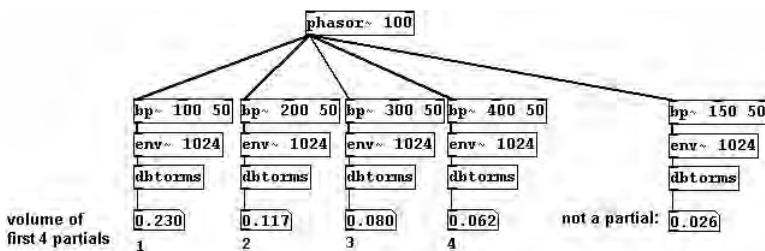
## 3.8 Fourier analysis

### ◆ 3.8.1 THEORY

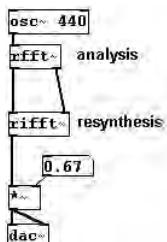
#### ► 3.8.1.1 Analyzing partials

Let's return to a basic concept of additive synthesis: a sound comprises partials. If you want to find out what the component parts of a sound are, you could employ a set of band-pass filters for every partial:

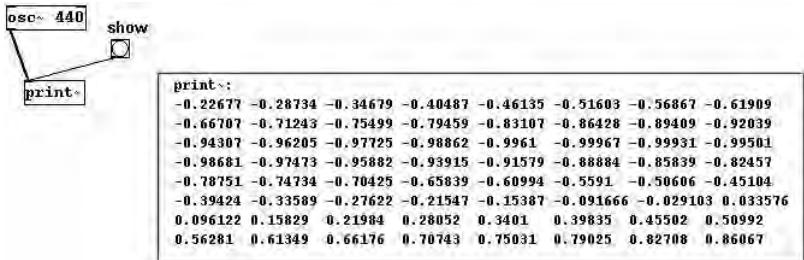
patches/3-8-1-1-analyze-partial.pd



This process performs what is called Fourier transformation. It divides the entire frequency spectrum into parts of equal size and determines the amplitude and phase for each part. One could in turn reconstruct the original signal from these values. The derivation of the component parts is called *analysis*; the reconstruction is called *resynthesis*. You can realize this using the objects “rfft~” and “irfft~”:

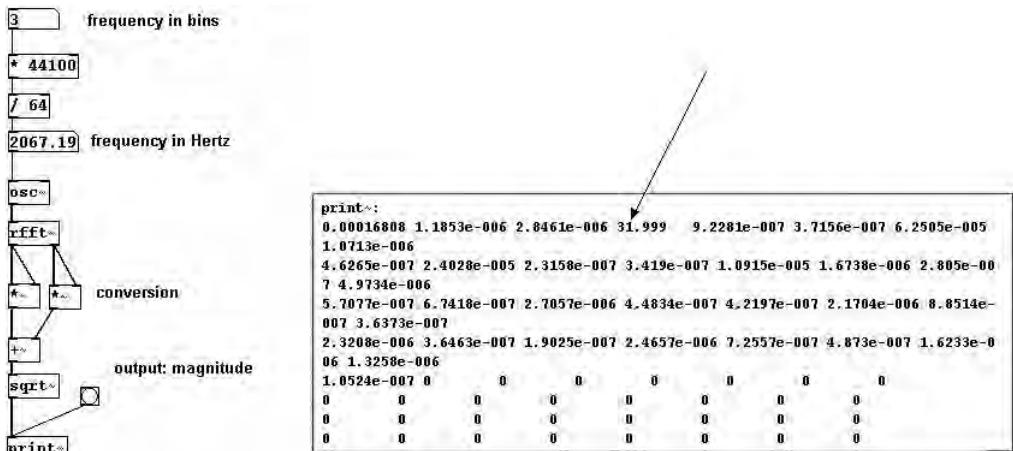


The size of the individual sections, called bins, is given by the block size. As discussed in Chapter 3.1.1.3.2, Pd always processes all tasks in blocks. Normally the block size in Pd is 64 samples. Using “print~” shows you all the values in a given block:



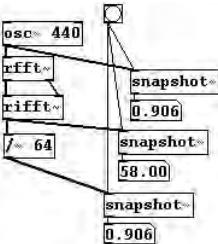
As with “snapshot~” or “unsig”, you can see the amplitude values produced. With “print~” you can actually see ALL of the values generated, limited in number to one DSP block. Let’s first stick to 64 samples; i.e., the entire spectrum up to 44100 Hertz is divided into bins with a size of  $44100/64 = 689$  Hz. The next thing we have to consider is that the amplitude and phase data with “FFT” is not represented in the customary format; they appear as sine and cosine values. For now, let’s not pursue this particular facet in further detail; you can transform the data into a more readily comprehensible form as follows:

patches/3-8-1-1-rfft1.pd



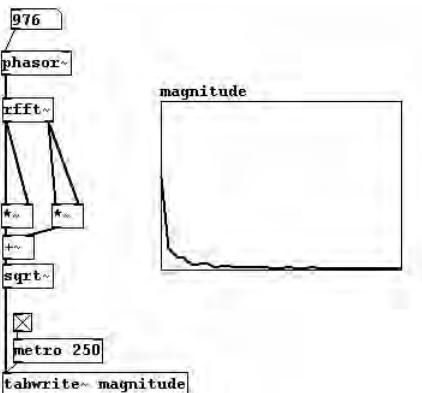
As you can see, “print~” generates 64 values for amplitude. The amplitude is given here as magnitude, always a positive value (because it was squared). Let’s take a closer look: except for the third bin, which has a value of (ca.) 32, we have nothing but very small values. There is no calculation for numbers above the Nyquist frequency.

Usually a normalization process is conducted after a FFT process, because the amplitude values become fairly high. First, this is the block size:



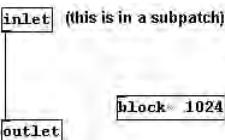
You could present the FFT analysis in an array:

patches/3-8-1-1-rfft-array.pd



This way you can see the spectrum of a signal. N.B.: FFT turns information that occurs in time into information in frequencies; these are updated in every new block. One speaks of the *time domain* and the *frequency domain*.

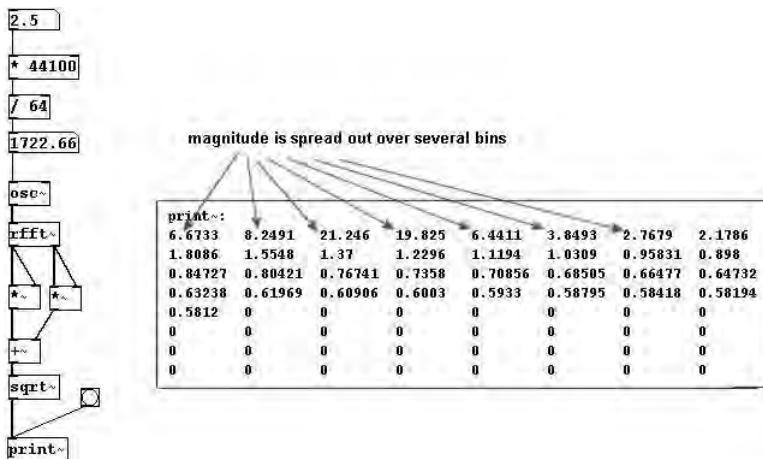
In Pd, the block size can only be changed in a subpatch. This is achieved using “block~”:



When choosing the block size, be sure to consider that a larger block size allows you to work with lower frequencies. For example: with a size of 1024 samples, every bin is  $44100/1024 = \sim 43$  Hz in size, so you have a finer resolution. The downside is that the process takes longer.

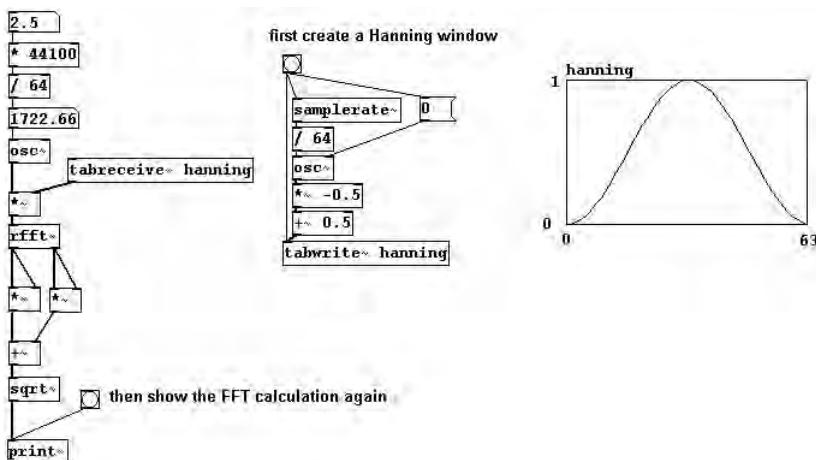
### ► 3.8.1.2 Analyze whatever signal you want

Let's stick with a block size of 64 samples, which we can use to analyze the spectrum of a fundamental frequency of 689 Hz. But what if other frequencies occur in between?

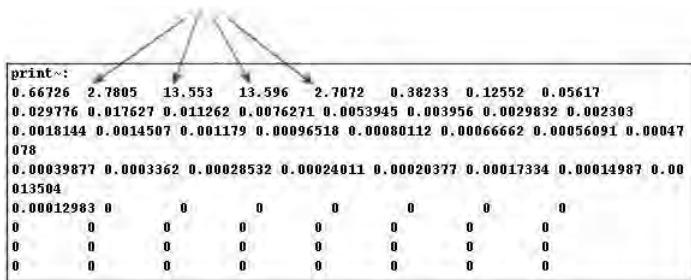


Then the information is divided among several bins and the phase changes with every analysis. This problem cannot be completely solved; you have to trick it a little bit. The normal way to solve the problem is to use overlapping windows as in granular synthesis; you create a windowed version of the original. You can use “tabreceive~” to achieve this, an object that always reads the given array in block size with a Hanning window—here with 64 samples.

patches/3-8-1-2-rfft3.pd



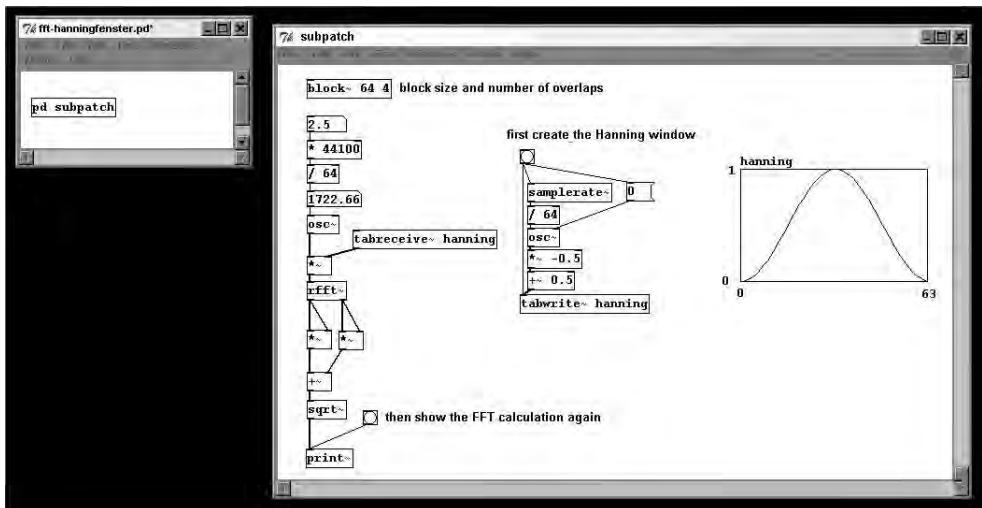
This way, the magnitude values aren't so "spread out".



0.66726	2.7805	13.553	13.596	2.7072	0.38233	0.12552	0.05617
0.029776	0.017627	0.011262	0.0076271	0.0053945	0.003956	0.0029832	0.002303
0.0018144	0.0014507	0.001179	0.00096518	0.00080112	0.00066662	0.00056091	0.00047
0.00039877	0.0003362	0.00028532	0.00024011	0.00020377	0.00017334	0.00014987	0.00
0.00012983	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

In addition to windowing, the windows need to overlap each other. This is very easy to do in Pd: simply enter the number of windows (usually 4) as the second argument of "block~". The result at the end also has to be windowed. The appropriate normalization for 4 overlapping windows is  $(3 * \text{block size}) / 2$ . Because you're using "block~", all of this has to fit in a subpatch.

patches/3-8-1-2-fft-subpatch.pd



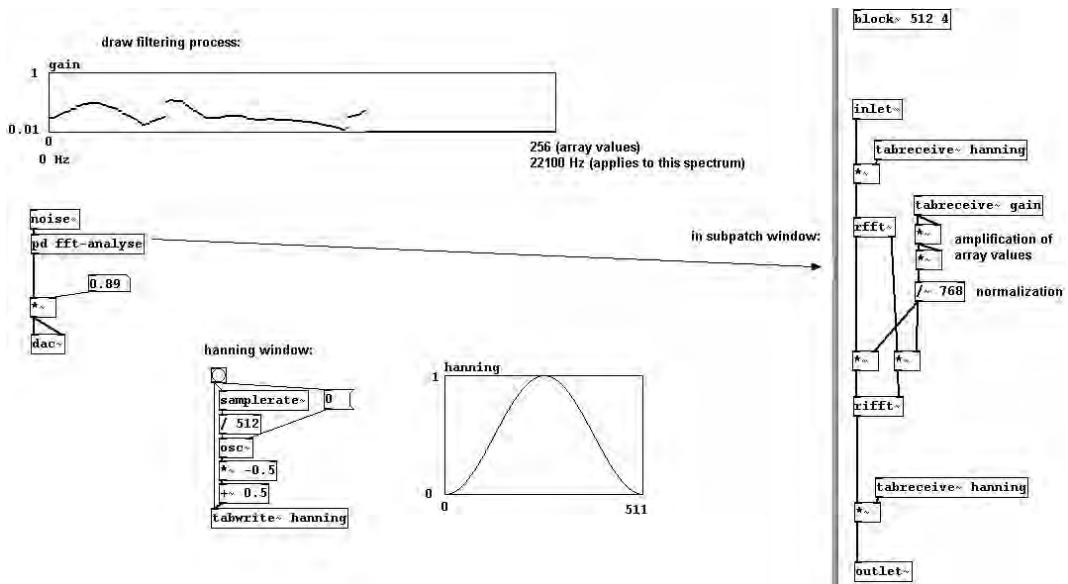
By overlapping and windowing, the chances are good that a signal will be correctly analyzed.

## ◆ 3.8.2 APPLICATIONS

### ► 3.8.2.1 Filters

What's useful about FFT, of course, is that the values it determines can be changed before you resynthesize the components into a sounding result. For example, you could set certain bins to be louder or quieter; you could build filters like high-pass, low-pass, etc., or 'draw' one yourself.

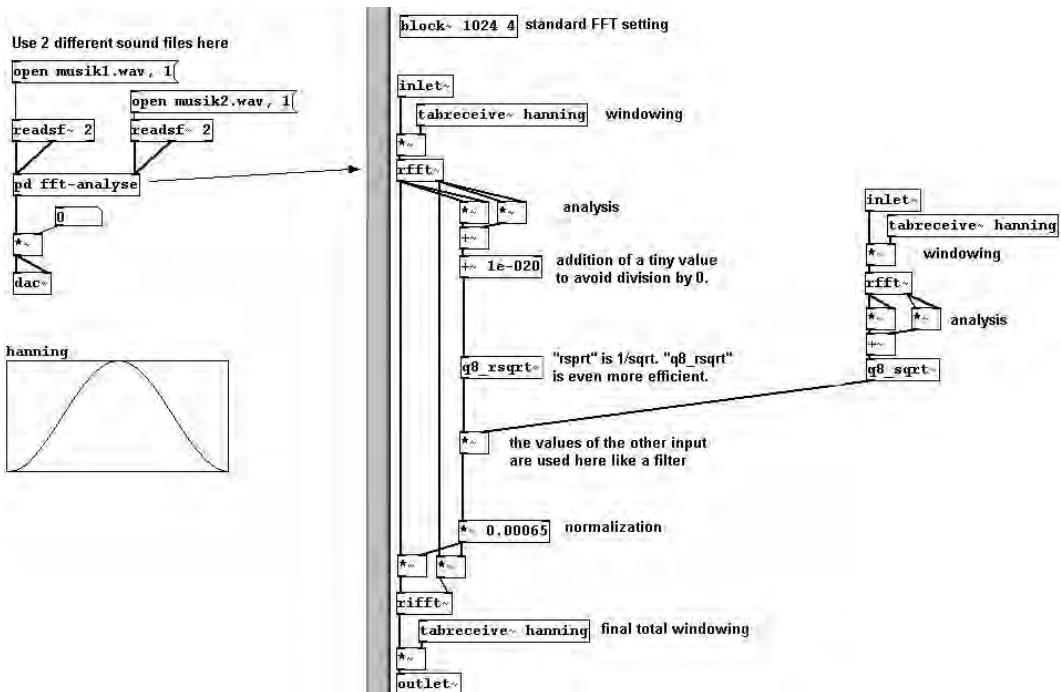
[patches/3-8-2-1-fft-filter.pd](#)



### ► 3.8.2.2 Folding

Convolution is a celebrated effect—folding a signal together with another; i.e., playing the average of their amplitudes. The Hanning calculation should be familiar to you by now. A block size of 1024 samples and four overlaps is standard.

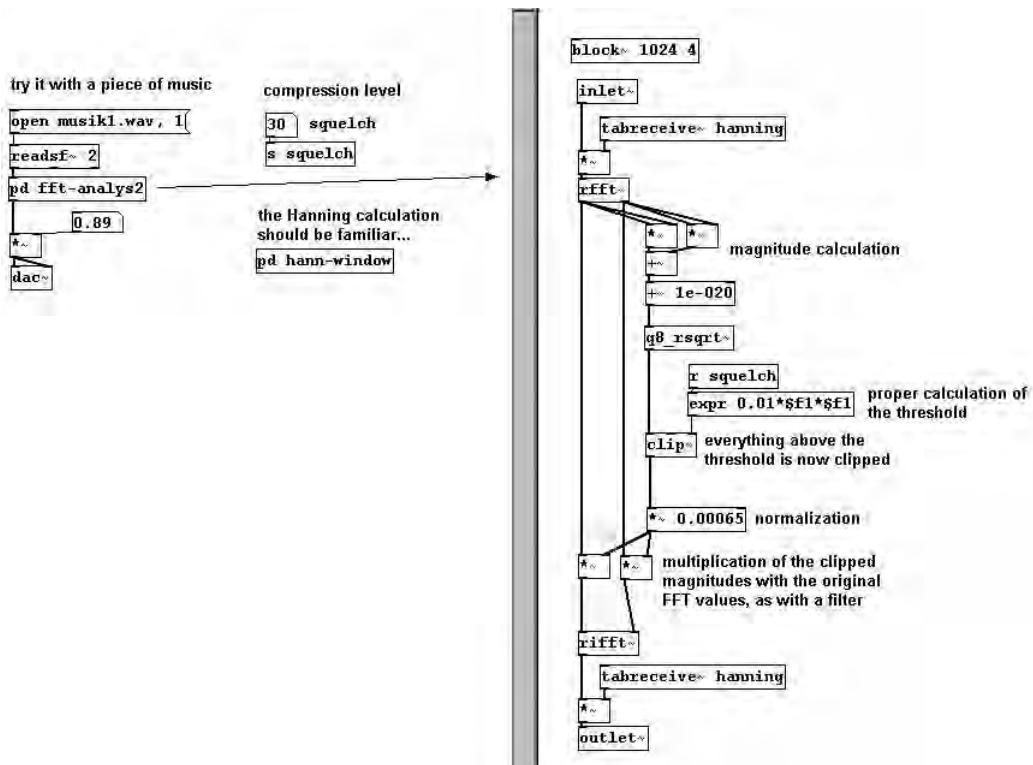
patches/3-8-2-2-convolution.pd



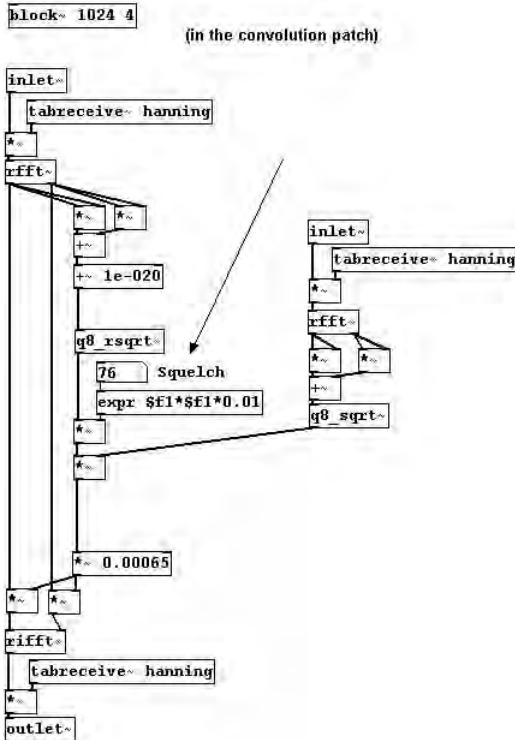
### ► 3.8.2.3 Compressor

You could also build a compressor. This means that weaker volumes will be amplified a bit to bring them closer to the louder volumes. Simply use the magnitude values as factors for the outputs of “rfft”, though be aware that values that exceed a certain threshold (“squench”) will simply be cut off at that point:

patches/3-8-2-3-compressor.pd



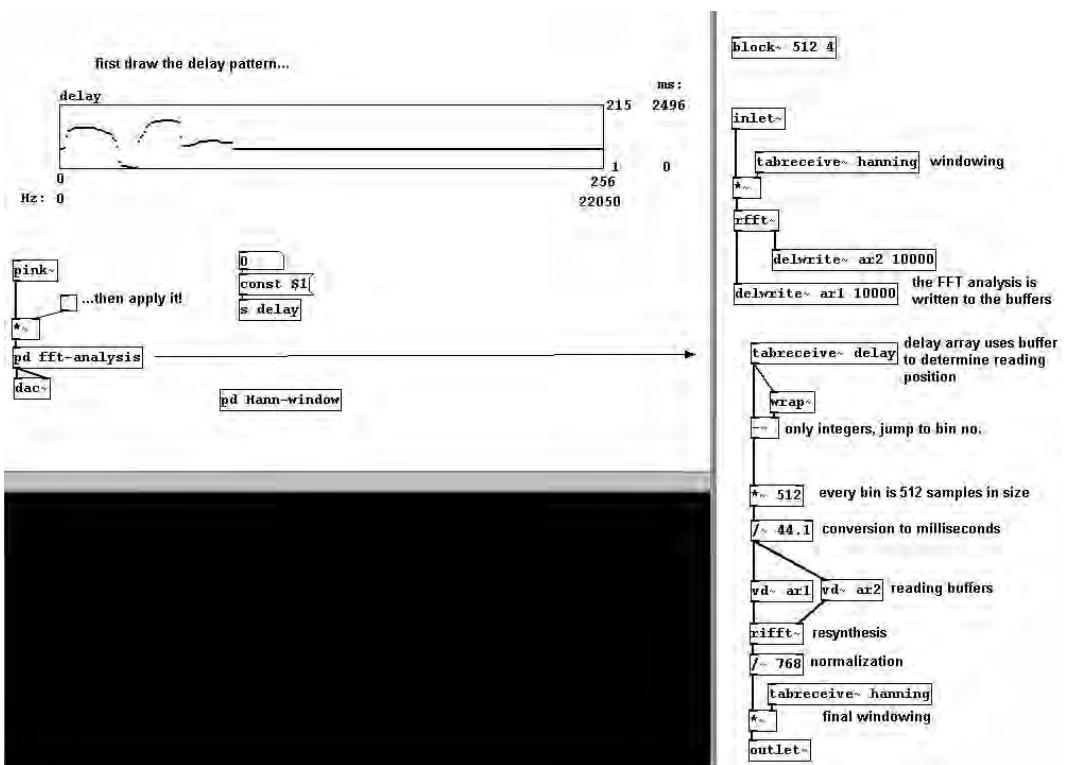
If you implement this in the folding of one of the two analyses, you get a richer convolution effect:



#### ► 3.8.2.4 Spectral delay

You can also play back certain bins with different amounts of delay to achieve what's called a "spectral delay". The FFT analysis is written to two different buffers. Using an array, you determine the delay for each bin. The maximum delay time is ca. 2500 milliseconds, as you have a buffer of 10000 milliseconds but 4 overlaps ("block~"), which means  $10000/4 = 2500$ . To be precise, it's actually 2496 milliseconds:  $2496 * 44.1 = \text{ca. } 110080$  samples, which is  $110080 / 512 = 215$  possible bin positions. Since the input signal usually doesn't fit in the bin size, the values of the analysis are divided among several neighboring bins (cf. 3.8.1.2). If these neighboring bins occur at different times, there can be reductions in volume.

[patches/3-8-2-4-spectral-delay.pd](#)

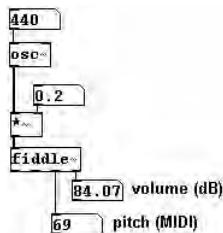


Try this out with a particularly eventful piece of music!

### ◆ 3.8.3 APPENDIX

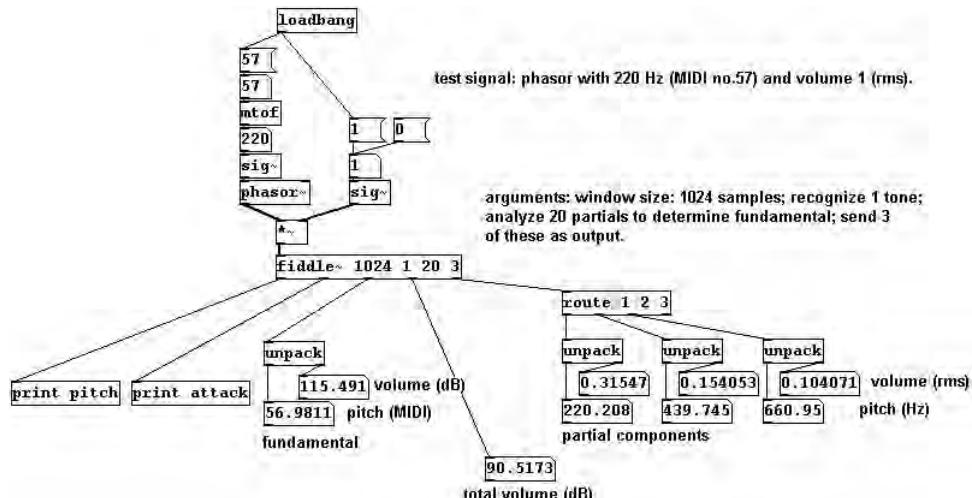
#### ► 3.8.3.1 fiddle~

There is an object in Pd that is based on the FFT algorithm that performs an analysis of both volume AND pitch. It is called “fiddle~”. It also determines the volumes and partials of the input signal.

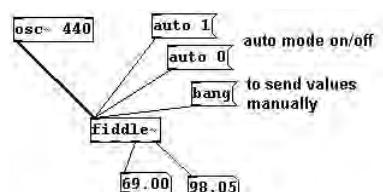


The arguments it receives are: 1. Window size (in samples), 2. Number of tones to be recognized simultaneously (max. three different tones), 3. Number of peaks to find, and 4. Number of peaks to output. The default settings are: 1. 1024, 2. 1, 3. 20, 4. 0. As outputs (from left to right): 1. Pitch in MIDI (only when there is a change), 2. Volume in dB (only when there is an extreme change ("attack")), 3. Pitch and volume of the fundamental (as a list), 4. Total volume, and 5. Individual partials with their respective volumes (in Hertz / rms!—also as a list).

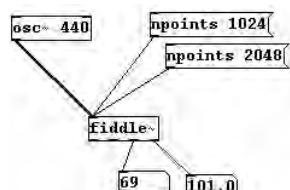
Patches/3-8-3-1-fiddle.pd



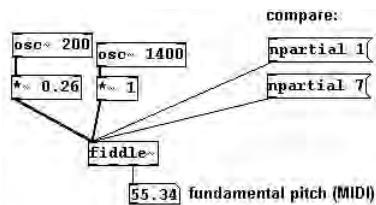
Messages for "fiddle~": to avoid constant data processing, you can turn off "auto mode" and activate "poll mode" instead; this only issues numbers when it gets a 'bang' message:



You can determine the window size (multiples of two):



Higher partials are not analyzed as intensively for determining the fundamental. You can change this, however, by instructing the object to analyze a certain partial at least half as intensely as the fundamental:



We use 200 Hz as the fundamental. The fundamental is played quietly, the 7th partial (1400 Hz) loud. Fiddle first identifies the louder tone as the fundamental until we tell it that the 7th partial is especially loud. It then concludes that the louder tone must be the 7th partial and correctly identifies the fundamental.

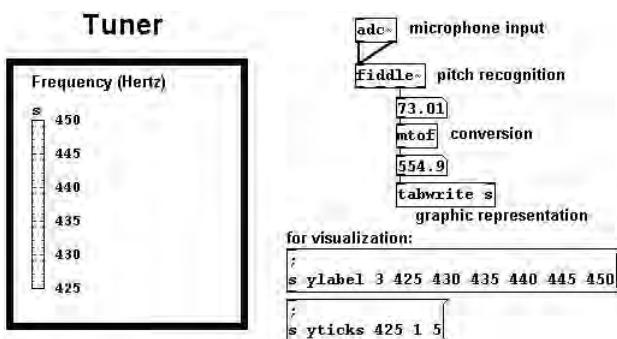
This is helpful if you know that certain partials of the input signal are especially strong (e.g., the third partial on a clarinet).

N.B.: The input signal is analyzed every half window size, i.e., if the window size is 1024, then 512 samples, which equals every 11.6 milliseconds. The smallest frequency that “fiddle~” can recognize is  $(44100 / \text{window size}) * 2.5$ ; for a window size of 1024 samples, this is ca. 108 Hertz.

#### ► 3.8.3.2 Tuner

Here's one way to build a tuner:

[patches/3-8-3-2-tuner.pd](#)

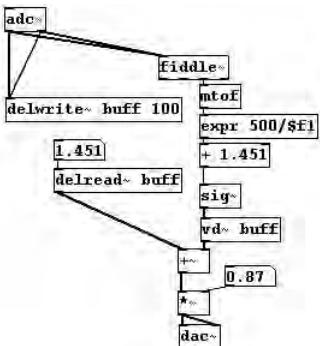


For this visualization, an array with only one storage place was used.

### ► 3.8.3.3 Octave doubler #2

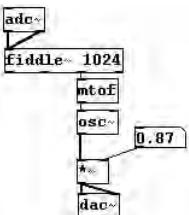
For the octave doubler described in 3.4.2.9, you can now use a microphone input as long as this fundamental can be used to conduct calculations (i.e., as long as the input signal is periodic and can be understood by “fiddle~”):

patches/3-8-3-3-oktavedoubler-fiddle.pd



### ► 3.8.3.4 Pitch follower

Many interesting applications can be imagined that use the “fiddle~” object in this way. A prototypical example would be a microphone input, like a singing voice, and to ‘trace’ the voice’s melodic contour like a laser pointer:



The following dilemma arises: there is always a delay when using “fiddle~”. The smaller the window size, the shorter this is. However, the smaller the window size, the higher the lowest range of pitches that can be recognized. Moreover, the result of “fiddle~” is always a bit chaotic. You can learn how to minimize this under 4.3.1.3.

### ► 3.8.3.5 More exercises

Instead of simply ‘tracing’ the microphone input, create a parallel voice a perfect fifth away or even a whole parallel chord.

## 3.9 Amplitude corrections

### ◆ 3.9.1 THEORY

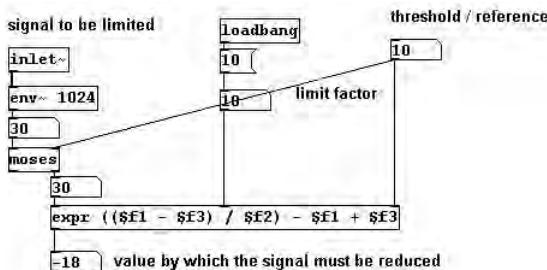
At the end of this large chapter devoted to audio techniques in Pd, we'll now take a look at amplitude processing.

#### ► 3.9.1.1 Limiter

As you learned in 3.1.2.1.2, the loudspeaker membrane can only vibrate up to a certain point; thereafter it is simply “clipped”. You can, however, build an automatic device that reduces parts of a signal that are too loud before they clip. In acoustic engineering, a device that accomplishes this task is called a “limiter”.

With a limiter, there is only one input signal whose volume has to be measured. If it exceeds the set upper limit, its volume will be reduced according to a set factor until it reaches the reference point. (In the following section, volume is calculated in dB.)

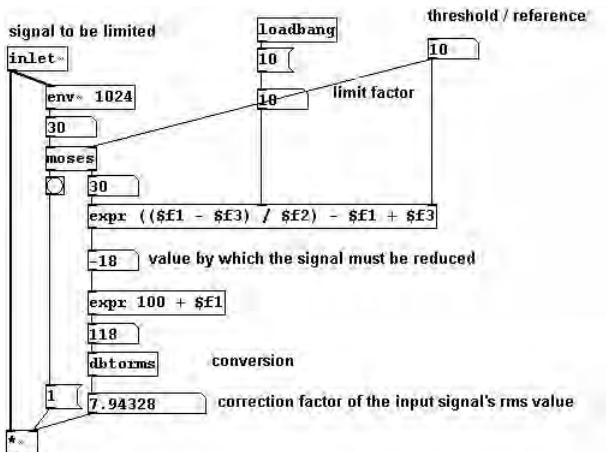
patches/3-9-1-1-limiter1.pd



In the first example, the threshold is 10 dB, the factor 10, and the input signal 30 dB. The difference between the input signal and the reference point is 20 dB. The factor determines that it should be reduced (by a factor of 10) to 2 dB; i.e. the input signal must be reduced by -18 dB to a value of 12 dB.

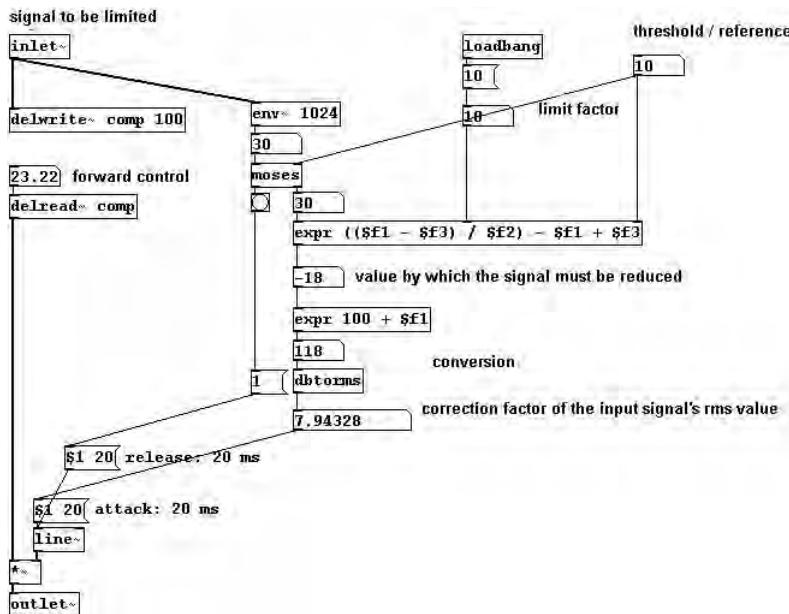
Now we'll apply the limiting factor to the original signal whenever it exceeds the threshold; if it stays below the threshold, the factor simply remains 1:

patches/3-9-1-1-limiter2.pd



There are two aspects at work: "env~" takes the average within the given sample, here always in a window of 1024 samples. That results in a delay of  $1024 / 44.1 = 23.22$  milliseconds that you can set. But you could also perform the reduction before the signal exceeds the threshold, which would delay the original signal even more. In signal processing, the term for this is "feedforward control action". If the delay of the original signal is shorter and the delay of the correction longer, one speaks of "reverse control action". In this case, the threshold is exceeded briefly before any correction is made. Some variables that have to be defined here are the speed at which the correction occurs and also the speed with which the signal returns to its original volume once it falls below the threshold again; these are called "attack" and "release" times.

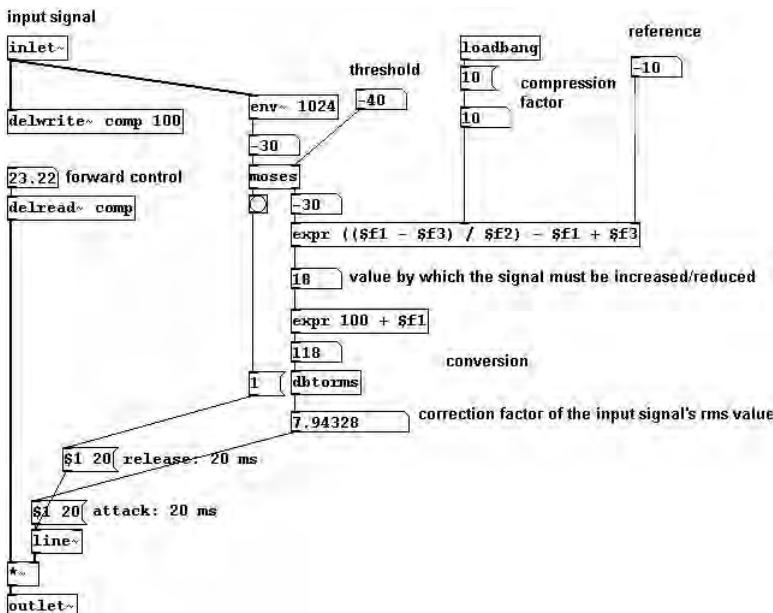
patches/3-9-1-1-limiter3.pd



### ► 3.9.1.2 Compressor

Once you've established the upper threshold (after which correction occurs) and the reference point (the point that corrections approach), you can also adjust the settings so that volumes below the reference point are amplified and those above it are made quieter. A device that accomplishes this task is called a compressor:

patches/3-9-1-2-compressor.pd



### ◆ 3.9.2 APPLICATIONS

#### ► 3.9.2.1 Larsen tones

If you connect a microphone input to a speaker—e.g., the “adc~” object directly to the “dac~”—and hold the microphone to the speaker, you'll soon hear a tone (and you'd best point the microphone away fast!). This is because air always contains some noise; the microphone picks it up and sends it to the speaker; the speaker plays it and the microphone picks up this amplified signal; etc. Depending on the distance between the microphone and the speaker, the signal is amplified each time. Depending on the room, cable length, and latency in the computer, this results in different high periodic tones, also called “Larsen Tones”. At the same time, volume increases dramatically, as the signal is constantly being amplified. This is the classic case of feedback—a circuit, a recursive system. Wherever microphones and speakers are used, there is a danger of feedback. A limiter used between the microphone and the speaker could help allay this danger.

### ► 3.9.2.2 More exercises

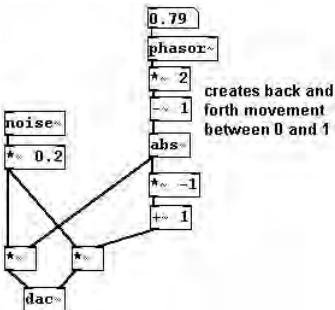
- a) Create an “expander”: turn small differences in amplitude into large differences!
- b) A volume inversion: change quiet into loud and loud into quiet.

### ◆ 3.9.3 APPENDIX

#### ► 3.9.3.1 Movements in space

Volume can be used to simulate movement in space. Normally we have a pair of stereo speakers and therefore two inputs for “dac~”. If you change the relative volumes of the speakers gradually—provided you are directly between the two speakers—you can experience how sound ‘travels’ back and forth between the two speakers. This is called a ‘phantom sound source’.

patches/3-9-3-1-spatial-stereo.pd

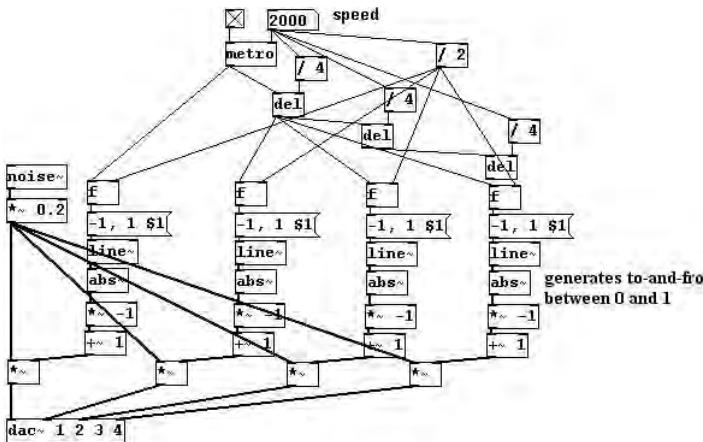


Speaker set-up:



If you have, say, four speakers set up in a square, you can create circular movement through space (naturally, this requires a sound card with four separate outputs; as arguments, you can give the inputs to the “dac~”):

patches/3-9-3-1-spatial-quadro.pd



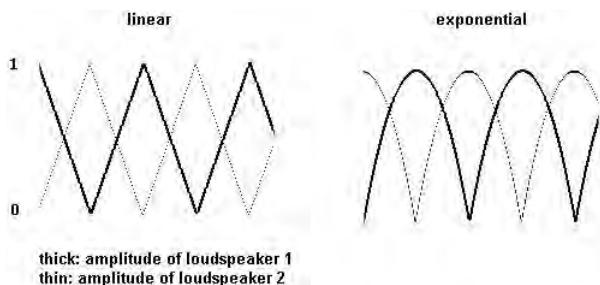
## Set-up:



■ sweet spot



In this case as well, the effect only works correctly if you are precisely in the middle (in the “sweet spot”). In the above example, you can also clearly hear a ‘hole’ in the volume between the two highest volume levels for each speaker. You should experiment to see whether the overlapping of the volume should be linear or exponential (cf. the window types in 3.9.4). The composer has to use her ear and decide for herself.

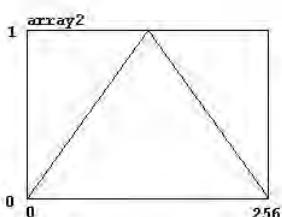
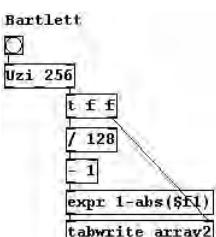
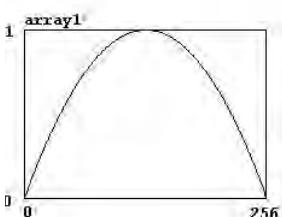
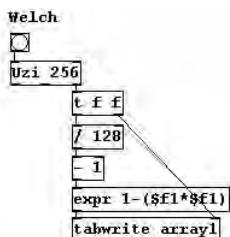


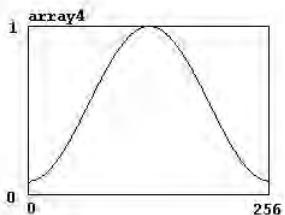
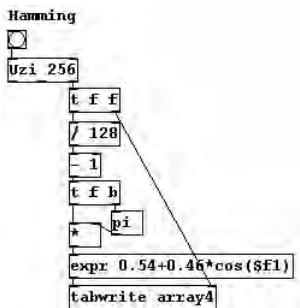
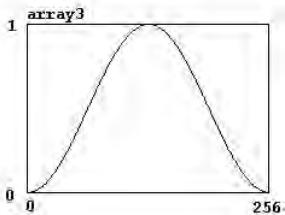
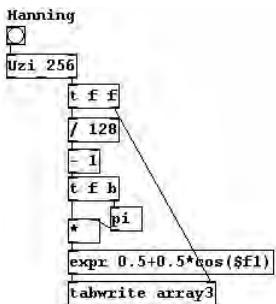
◆ 3.9.4 FOR THOSE ESPECIALLY INTERESTED

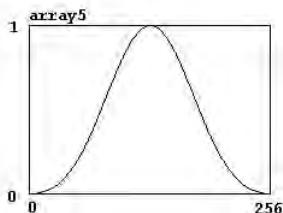
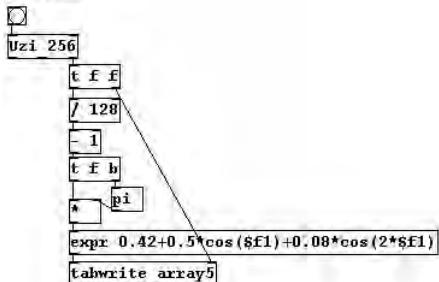
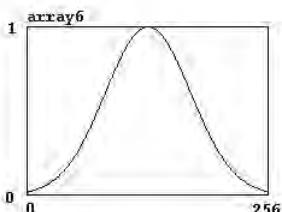
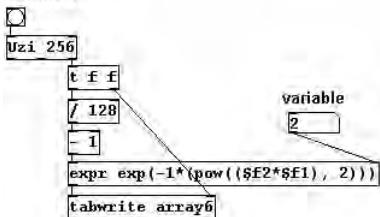
► 3.9.4.1 Other windows

In previous chapters, “Hanning” windows (which correspond to part of a cosine function) were often used to avoid clicks. But there are also other types of windows that you could experiment with:

patches/3-9-4-1-windowing.pd





**Blackman****Gaussian**

## Chapter 4

### Controlling sound

Music occurs in time and a composer would naturally like it for the music to change in time as well. In the previous chapter, we covered the basics for generating sound. Now we'll take a look at how you can use Pd to control these generated sounds—or control the control of these sounds—in time.

#### 4.1 Algorithms

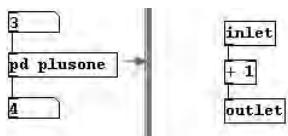
##### ◆ 4.1.1 THEORY

###### ► 4.1.1.1 What are algorithms?

Algorithm is the technical term for the description of a sequence of steps in a procedure that a computer program executes.

If you have a subpatch that adds 1 to an entered number, you could already consider this a (very simple) algorithm: this subpatch's algorithm is the addition of 1.

patches/4-1-1-1-plus-one-algorithm.pd



In essence, every object in Pd executes an algorithm. What used to require the use of a device called a noise generator is accomplished today with the algorithm contained in the “noise~” object.

In this chapter, we are particularly interested in developing algorithms that, once we initiate processing, the computer can execute completely on its own and that fulfill the purpose of changing sound in time. Some examples of this have been named already, like in 2.2.3.2.7.

#### ◆ 4.1.2 APPLICATIONS

##### ► 4.1.2.1 Stochastics

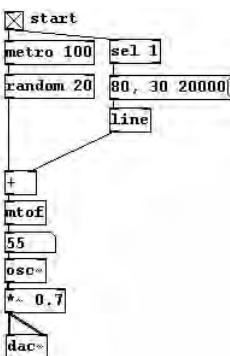
A very simple but abundant way to get the computer to operate on its own is by using a random generator.

patches/4-1-2-1-random.pd

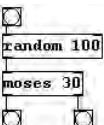


You can apply limits to this random selection that themselves change:

patches/4-1-2-1-random-limits.pd

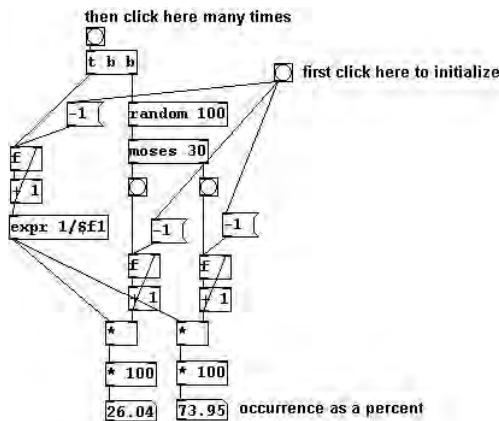


The result of a random generator follows the laws of stochastics, that is, of probability. With “random 6”, every number from 0 to 5 has a probability of  $1/6$ . Though highly improbable, it is possible that one of the numbers would never appear or wouldn’t appear for a very long time. This probability can also be directly controlled:



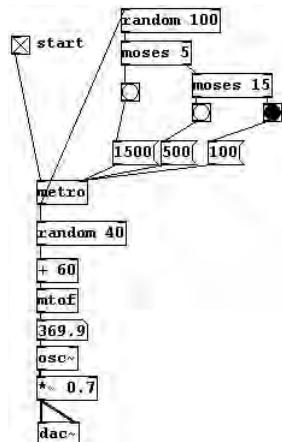
Here, the probability that a bang occurs on the left is 30 percent and 70 percent on the right. You can test this as shown here:

patches/4-1-2-1-probability.pd

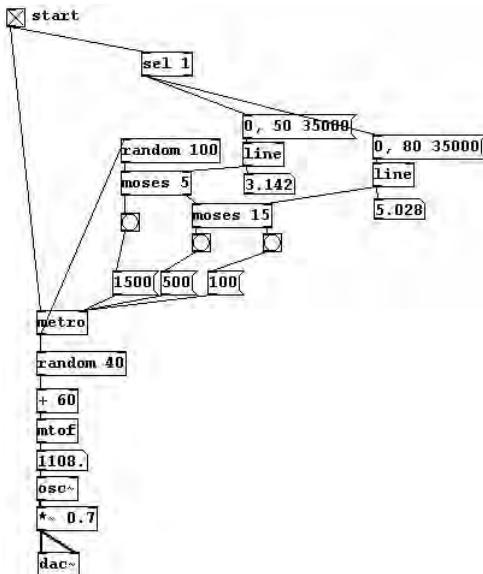


You can use this principle to select different durations for certain sound events: short occur very frequently, medium once in a while, and long rarely.

patches/4-1-2-1-probability-examples.pd

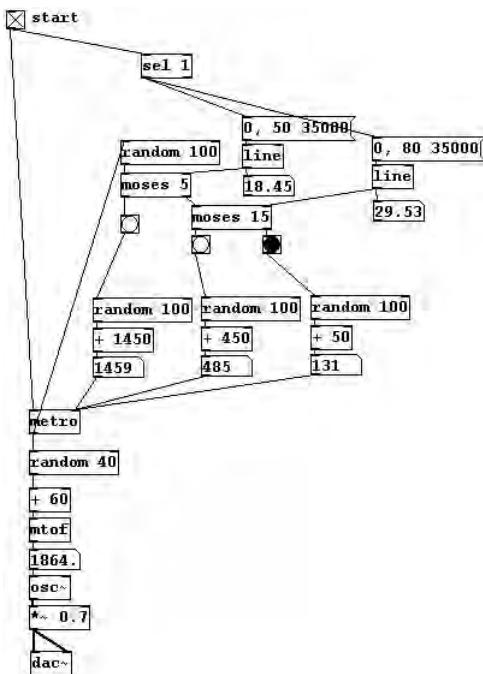


You could also invert this distribution over time...



At the beginning there are only short durations, at the end mainly long ones (which naturally require a lot of time).

Also, a little bit of variation can be introduced to the different durations:



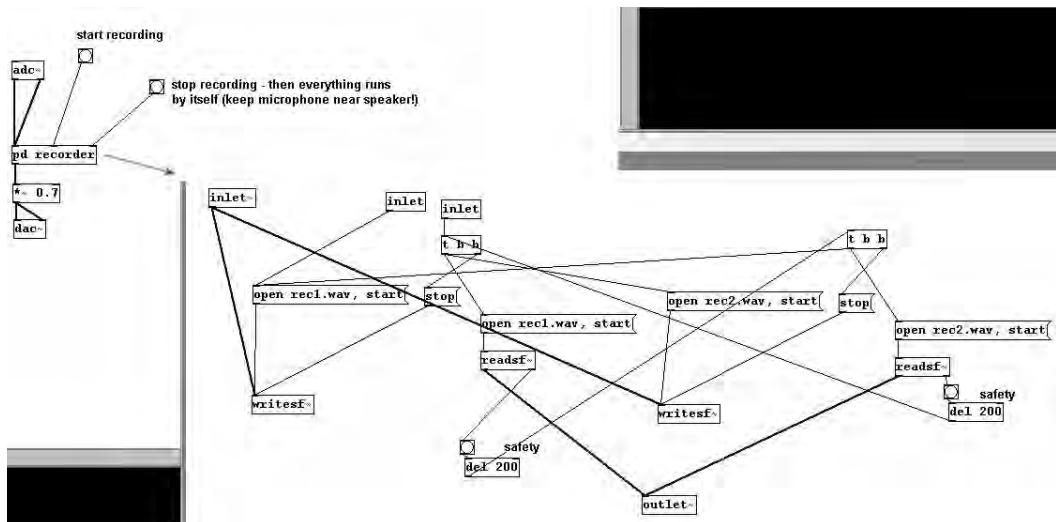
You can keep going with this, of course, ‘randomizing’ more and more parameters in your patch as you see fit.

#### ► 4.1.2.2 Recursive systems

There is a piece by Alvin Lucier based on a relatively simple idea: someone sits in a room and speaks into a microphone. This speech is recorded and then played back in the room. This playback is in turn recorded, played back, re-recorded, and so forth. Each time, the quality of the recording is worse, more and more information is lost. More specifically, the frequencies that the loudspeaker, microphone, and room can represent well are propagated while the others are gradually filtered out.

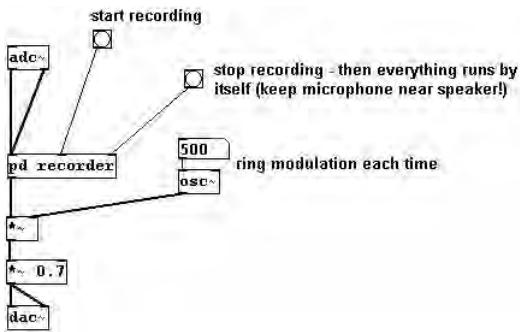
This can be easily programmed in Pd:

[patches/4-1-2-2-lucier.pd](#)



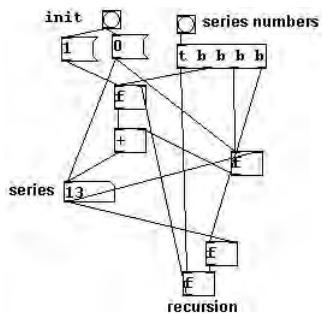
You could say that the algorithm is recording and playback. In this case, the result is fed back into the algorithm again and again. A process that runs automatically in this way is called a recursion. Recursions have already been mentioned in 3.4.2.9 and 3.4.2.10.

In the next example, we will also use alteration and re-recording. A recursive ring modulation:



Recursions that work purely with numbers can also be interesting. One of the most well known examples of this that frequently occurs in music is the Fibonacci series. The algorithm is that the last two numbers in a list are added together to produce a new final result in the list.

patches/4-1-2-3-fibonacci.pd



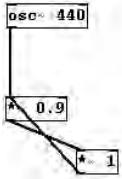
#### ► 4.1.2.3 More exercises

- Record a sample and play it back at the wrong speed. Record this ‘wrong’ playback, play it back, record, and so forth. Try this while (1) playing back the sample with the same ‘wrong’ speed and (2) with a different ‘wrong’ speed.
- Create a recursive wave shaping algorithm using delay (i.e. an algorithm in which the output is fed back as input).

#### ◆ 4.1.3 APPENDIX

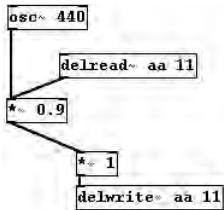
#### ► 4.1.3.1 DSP loop

Recursive algorithms used to distort sound have technical limits. If you do the following...



...the error message “DSP loop detected” appears and audio will no longer be sent through the loop. Without using time delay on the signal, you can’t create any (audio) recursions.

Here’s how you can avoid errors:



#### ◆ 4.1.4 FOR THOSE ESPECIALLY INTERESTED

##### ► 4.1.4.1 Algorithmic composition

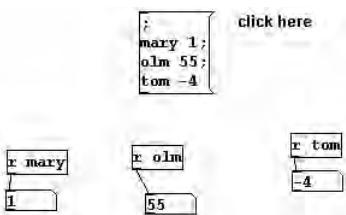
The use of algorithms in musical composition is a broad field. Algorithmic principles can be observed in the work of medieval composers and it has been a widely used area in music since the 20th century. Algorithmic compositions can be fascinating purely from a mathematical perspective. Nature is rich with examples of algorithms. For more information:

[http://en.wikipedia.org/wiki/Algorithmic\\_composition](http://en.wikipedia.org/wiki/Algorithmic_composition)

## 4.2 Sequencer

### ◆ 4.2.1 THEORY

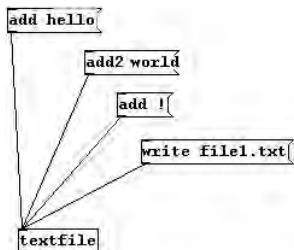
Instead of automatic processes, we can also write proper “scores” for a Pd patch. A simple example would be the use of many “send” commands, as described in 2.2.4.1.3:



But to be able to include much more information and determine the chronological sequence, the following section will cover ways to realize ‘scores’ in Pd.

### ► 4.2.1.1 Text file

You can retrieve numbers and symbols from a normal text file or, conversely, save numbers and symbols to a text file using “textfile”. Let’s first look at the saving function. Click the messages from top to bottom:

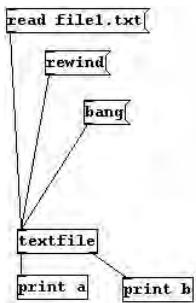


Now Pd has created a text file called “file1.txt” in the same directory as the patch. It contains:

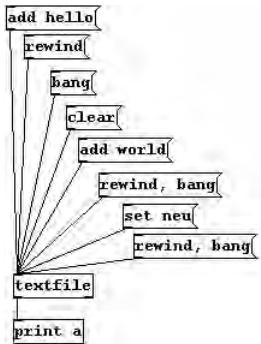
```
hello;  
world!;
```

“add” creates a symbol or number and follows it with a semicolon. “add2” doesn’t create a semicolon.

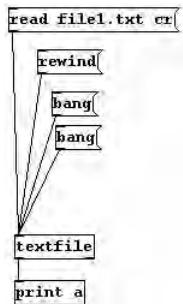
If you want to read what you’ve saved, load the file and use “rewind” to go all the way to the beginning. Now every time you hit ‘bang’, one line (up to the semicolon) will be sent through the left outlet. After the last line, a bang is sent out the right outlet.



You can also write something and read it with an object without ever saving it as a file. You can also use “clear” to delete everything. “set” first deletes everything and then begins a new line. Click from top to bottom:



You can also load a file so that the semicolons don't appear:

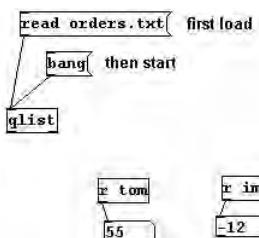


“write name.txt cr” also works in the same way.

#### ► 4.2.1.2 Qlist

A practical expansion to “textfile” is “qlist”. This can be used to send chronologically ordered messages with a text file to “receive” objects. The file “orders.txt” has these contents:

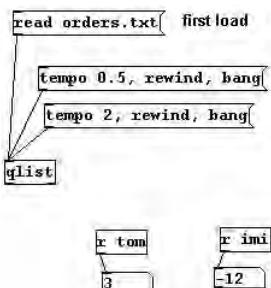
```
0 tom 55;  
1000 imi -12;  
4000 tom 3;  
2000 imi -2;
```



At the beginning, “tom” receives the number 55; one second later, “imi” receives -12; four seconds later “tom” receives 3; two seconds later “imi” receives -2. It works the same way with symbols.

Otherwise, “qlist” has the same functions as “textfile”: add, add2, rewind, clear.

You can also modify the tempo using “tempo” and a factor:

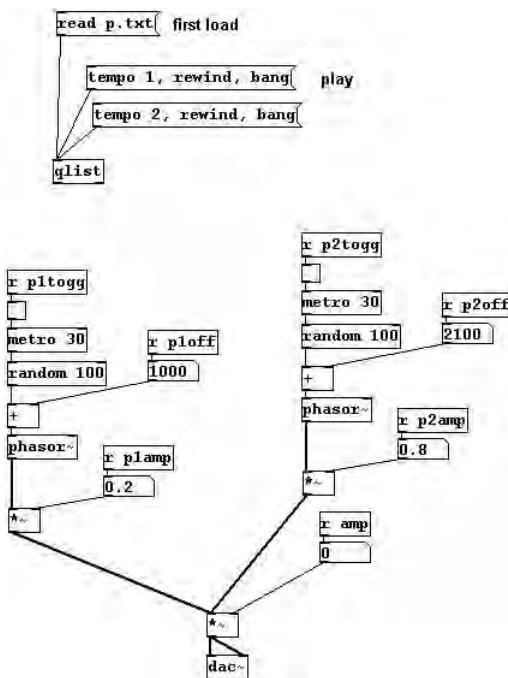


## ◆ 4.2.2 APPLICATIONS

### ► 4.2.2.1 Score for a patch

Provided the sounds have been assembled, you can now write a piece of music as a text file. Let's say you have this patch...

patches/4-2-2-1-score.pd



...and this "score" (patches/p.txt):

```
0 ploff 1000;
0 pltogg 1;
0 plamp 1;
0 amp 0.5;

3000 p2off 100;
0 p2togg 1;
0 p2amp 1;

2000 p2off 400;

3000 plamp 0.2;
```

```

3000 p2amp 0;

1000 p2off 2100;
0 p2amp 0.8;

```

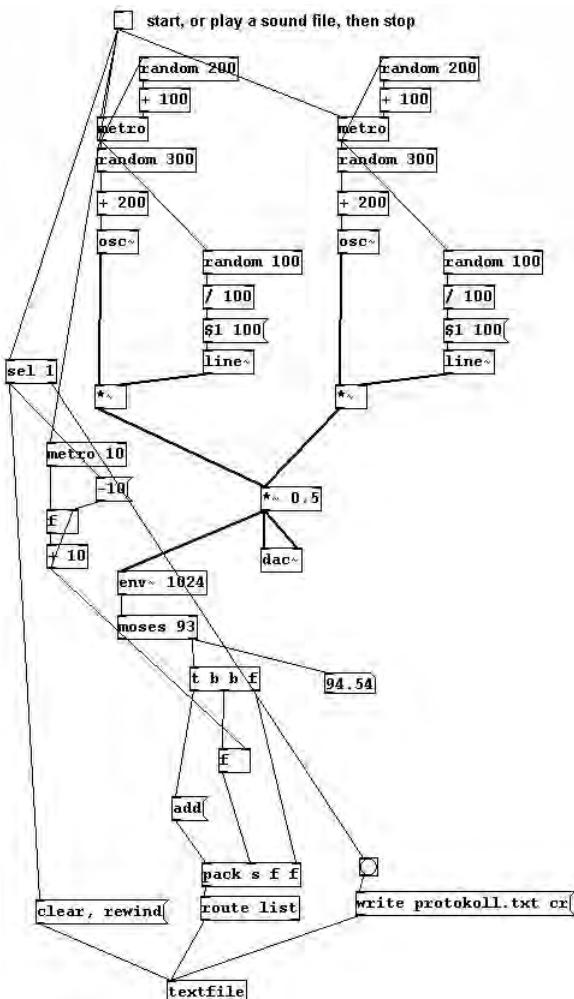
```

5000 amp 0;
0 p1togg 0;
0 p2togg 0;

```

You could also write information from sounds:

patches/4-2-2-1-write-score.pd



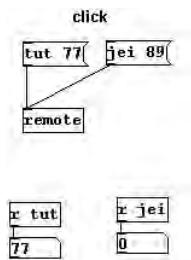
### ► 4.2.2.2 More exercises

Write stochastic algorithms into a text file that uses a “qlist” to play back the patch from 4.2.2.1 at different speeds.

### ◆ 4.2.3 APPENDIX

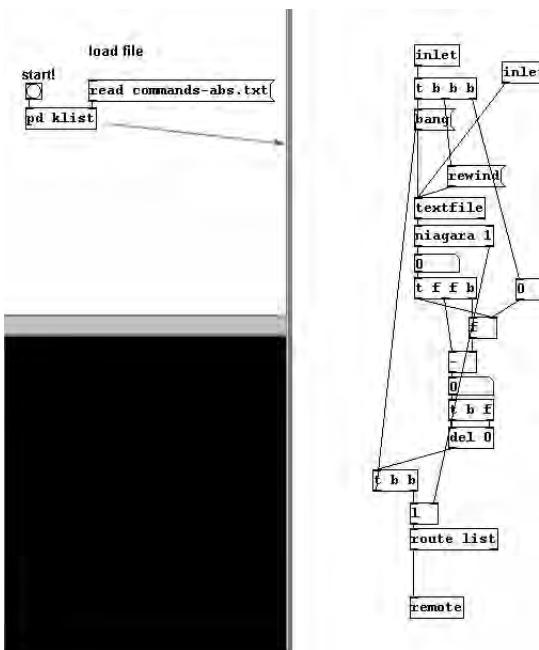
#### ► 4.2.3.1 Modifying qlist

The time for qlist are delta values, that is, they always describe the time interval from one event to the next. Sometimes it might be more practical to write text file with absolute time intervals instead. The “remote” object can be used to accomplish this. “remote” (not available in the original version of Pd; it’s part of Pd-extended) receives the name of a receive object as a list followed by the value that you want to send there. This saves you from having to use several “sends”:



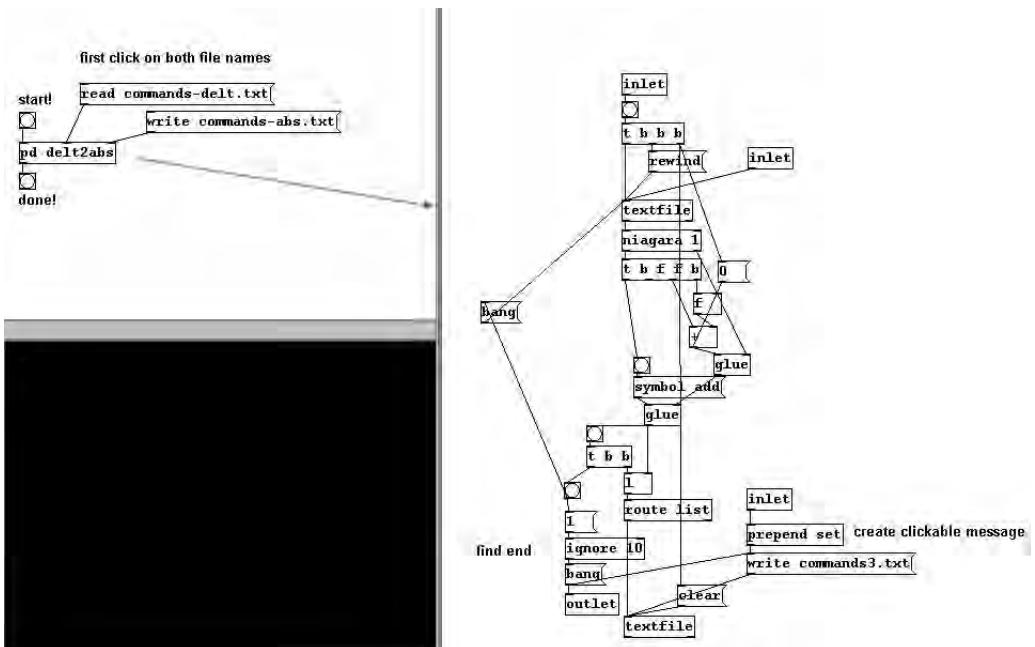
So you can build your own qlist using absolute values:

patches/4-2-3-1-qlist.pd



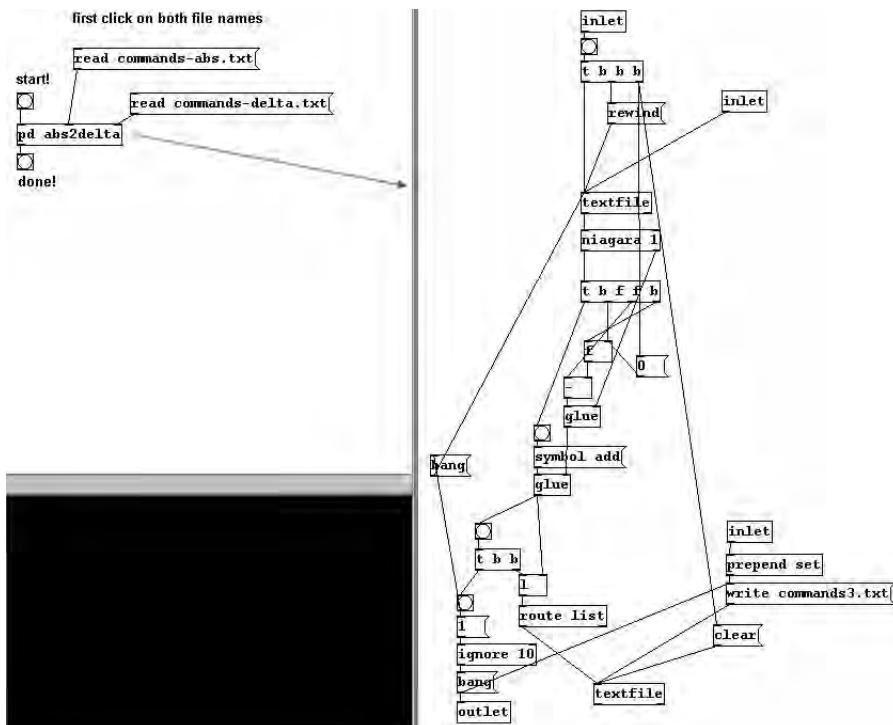
And here's how to change a list of delta values into absolute values:

patches/4-2-3-1-klist-convert1.pd



And conversely:

patches/4-2-3-1-klist-convert2.pd



◆ 4.2.4 FOR THOSE ESPECIALLY INTERESTED

► 4.2.4.1 Creating lists externally: Lisp

You can also take "textfile" to use text files that contain previously conducted algorithms. There are special programming languages that can accomplish this. One of these languages is LISP, which is especially well suited to the creation and processing of lists:

[http://en.wikipedia.org/wiki/Lisp\\_\(programming\\_language\)](http://en.wikipedia.org/wiki/Lisp_(programming_language))

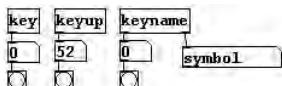
## 4.3 HIDs

### ◆ 4.3.1 THEORY

It's possible to play a patch live, just like you might play an instrument live on stage. While something is running in the patch, you can click on GUI objects, though using the mouse is fairly impractical if you want be precise with respect to time. For this reason, there are Human Interface Devices (HIDs), interfaces between man and machine. The mouse and keyboard are technically HIDs, but there are also many others, some of which are specially designed for use in music, for example to control a Pd patch.

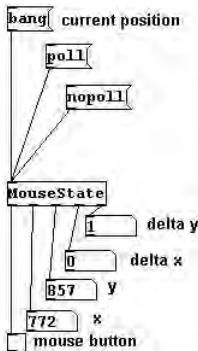
#### ► 4.3.1.1 Keyboard and mouse

If you click in a number box, you can enter a value with the keyboard. You could also use the keyboard to transmit other information directly:



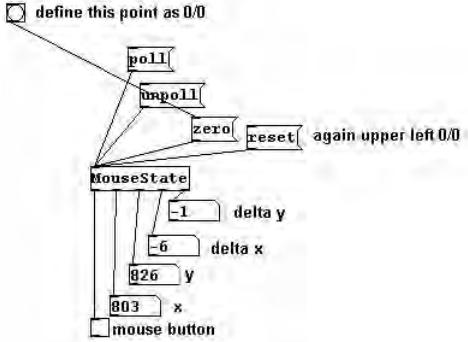
The keys are numbered (with some exceptions, the F1 and F12 keys, for example). “key” registers that a key has been pressed, “keyup” that it has been released. Each time, the key’s number is sent. “keyname” shows a key’s normal name.

Using the “MouseState” (case-sensitive!) external from Pd-extended, you can also use data from the mouse:



With “poll” and “unpoll” you can start/stop the display (in my version you have to click “unpoll” first and then “poll” for it to work). This displays absolute x/y-coordinates, delta values, and also whether the left mouse button is held.

Normally the coordinates 0/0 appear in the monitor’s upper left; with “zero” you can select another point as a reference:



#### ► 4.3.1.2 MIDI

At the beginning of the 80s, large electronic instrument manufacturers established a standard data transfer protocol for use with an array of input devices called Musical Instrument Digital Interface (MIDI). Now there are MIDI keyboards, MIDI mixers, MIDI gloves, etc. In Pd, there are objects for receiving and sending MIDI data. You can send this input to a device or patch where it is converted into sound. However, most computer soundcards contain MIDI sounds as well, so MIDI data can also be converted into sound there.

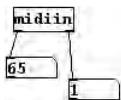
MIDI protocol itself doesn't contain any sounds, but comprises commands for controlling the patch or other instruments, e.g., "note-on", "velocity", and "note-off". In addition to these basic commands MIDI uses other more specialized commands that could be used to, say, load other sounds or to modify loaded sounds using control data produced using switches, buttons, or tuning knobs.

Every standard MIDI command (except for system-exclusive data, SysEx for short) carries a channel number in addition to its command ID and command data. The channel number is 4 bits long, which means  $2^4=16$  channels can be controlled. Depending on the software, the channels are numbered either 0 to 15 or 1 to 16, though the latter is more common.

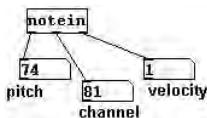
Since MIDI is a serialized protocol and the data rate of MIDI interfaces is fairly small by today's standards, there can often be timing problems when many notes are played at once, especially in conjunction with a sequencer program. Even just striking a chord comprising several notes can lead to an audible delay, because MIDI can never send notes simultaneously, only one at a time.

For the following examples, MIDI hardware is necessary. You can set these devices in Pd under **Media → MIDI settings**.

The most basic object is “midiin”. Every MIDI input is displayed there, the value on the left and the channel number on the right.

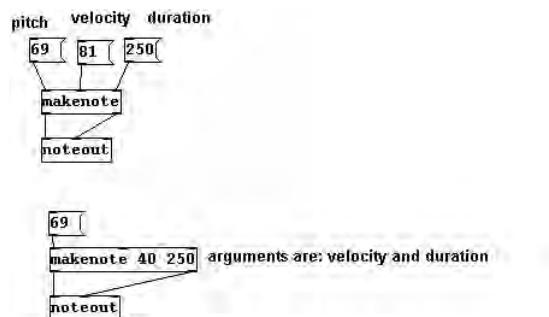


If you have a MIDI keyboard or other input device with definite pitches, with “notein” you get the following values:

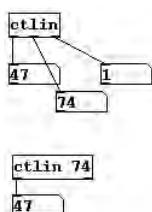


The MIDI number for the pitch appears on the left, the velocity (strength of the attack) in the middle, and the channel number on the right.

Conversely, you could send this information to an instrument. If only one instrument is connected, you don't have to enter a channel number. You'll need to use “noteout” and then “makenote”, the latter of which combines the entries in a manner similar to the “pack” object:



There are also control values, which are entered with “ctlin” and “ctlout”. Let's take a look at “ctlin”:

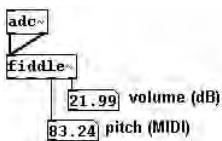


The left output is the value, the middle the number of the controller, and the channel on the right. You could also enter the middle value as an argument directly in the object to select the controller.

All other MIDI senders and receivers function like this as well. Among them are “pgmin”, “bendin”, “touchin”, and “sysexin”.

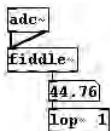
#### ► 4.3.1.3 Using signals to control sound

Sound input received through a microphone can not only be used purely as sound, but also as control data. As you learned in 3.8.3.1, you can use “fiddle~” to determine information regarding amplitude and frequency:



These numbers—again, Pure Data works only with numbers—can be used in conjunction with parameters in a patch.

One problem here is that data that comes from the “fiddle~” object is very chaotic. There is a trick you can use to filter it:



A low-pass filter can also receive control data input. In this case, only relatively slow changes are allowed to pass.

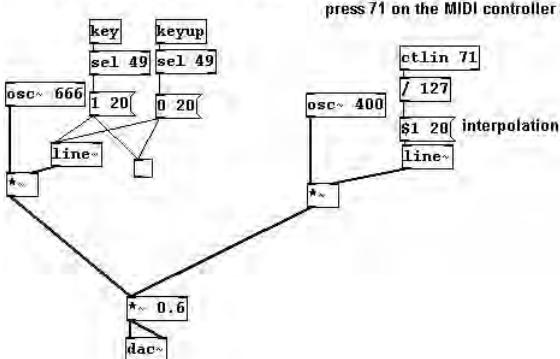
#### ◆ 4.3.2 APPLICATIONS

##### ► 4.3.2.1 Playing patches live

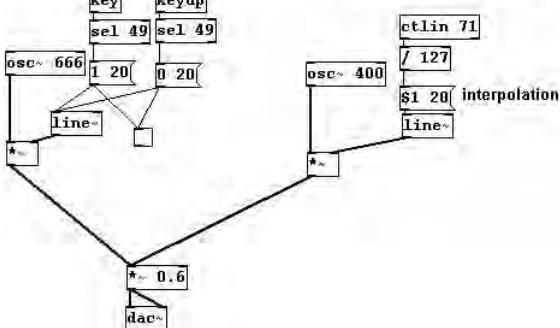
Using the input devices/methods described above, parameters in a patch can be changed externally:

patches/4-3-2-1-patch-play.pd

press "1" on the keyboard for this



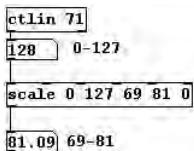
press 71 on the MIDI controller for this



It's clear that different devices have different functions: a button stands for on/off, a knob for gradual changes.

For controllers that have a row of numbers, like knobs or sliders, it is recommended that you use interpolation (especially because they have only 128 values in MIDI).

In Pd-extended (but only if GEM is not loaded), there is a very useful external you can use for this: “scale”. For example, if you change frequencies between 69 and 81 (MIDI numbers) and want to use a MIDI controller for this (which generates numbers from 0 to 127), you could write:



The last argument stands for linear (0) or exponential (1).

#### ► 4.3.2.2 More exercises

- Use external MIDI sounds instead of oscillators for the algorithms in 4.1.2.1.
- Use parameters from any of the patches in Chapter 3 with input devices.

◆ 4.3.3 APPENDIX

► 4.3.3.1 Other HIDs

In addition to the normal keyboard, mouse, and MIDI devices, the number of other input devices continues to rise: anything from joysticks for computer games, to tablets for drawing, to motion sensors. Though there is at present (June 2008) no single object in Pd that will work with all of these devices, the following externals should be mentioned:

“joystick”, which can receive information from joysticks; “wintablet” for Wacom tablets in Windows; and “hid” in Linux and Mac OSX, which works with many different input devices. A device called an Arduino Board has also been around for awhile; this device digitizes information it receives from analog instruments that are connected to it. With additional software, this information can also be received by Pd.

► 4.3.3.2 Video input

There is also a video component in Pd called GEM that can be used to extract numbers from a video signal—either previously recorded or live—that could, of course, then be used to control sound parameters.

◆ 4.3.4 FOR THOSE ESPECIALLY INTERESTED

► 4.3.4.1 Instrument design

I have also given a presentation (in German) in the Lagerhaus Lecture Series in Freiburg, Germany on the art of using external input devices in a composition. It can be viewed online at this address:

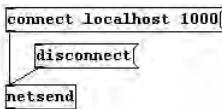
<http://www.kreidler-net.de/theorie/instrument-design.htm>

## 4.4 Network

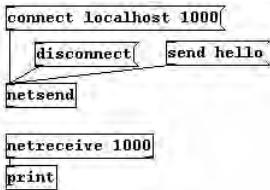
The following section deals with networking several computers that are to be used not only for sound processing, but also communication with other people at other computers in order to play a piece of music together.

### ◆ 4.4.1 NETSEND / NETRECEIVE

A Pd patch on one computer can exchange data with a Pd patch on another computer. First you connect the computers with a network cable, both of which are running Pd. Use “netsend” to connect your computer to another one. Enter the message “connect [name] [port number]”; instead of the name, you could also enter the IP address of the other computer. “disconnect” terminates the connection.



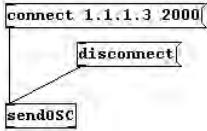
Once connected, use “send” followed by the symbols you want to send, to send messages to the other computer. It receives this data using “netreceive [port number]”.



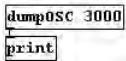
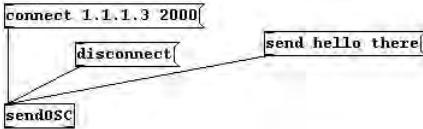
### ◆ 4.4.2 OSC

In Pd-extended, the OSC objects that are used by many other computer programs for data exchange are also available. OSC stands for *open sound control*. They work almost exactly like “netsend” and “netreceive”.

You have to be connected to another computer with a network cable and both must be running OSC. Use “sendOSC” (case-sensitive!) to connect to another computer. Give the message “connect” as well as the IP address and port number of the other computer. Terminate the connection using “disconnect”.



Once connected, use “send” followed by the symbols you want to send, to send messages to the other computer (which also must be running OSC). The other computer receives the data using “dumpOSC” with the port of the sender as the argument.



## Chapter 5

### Miscellaneous

#### 5.1 Streamlining

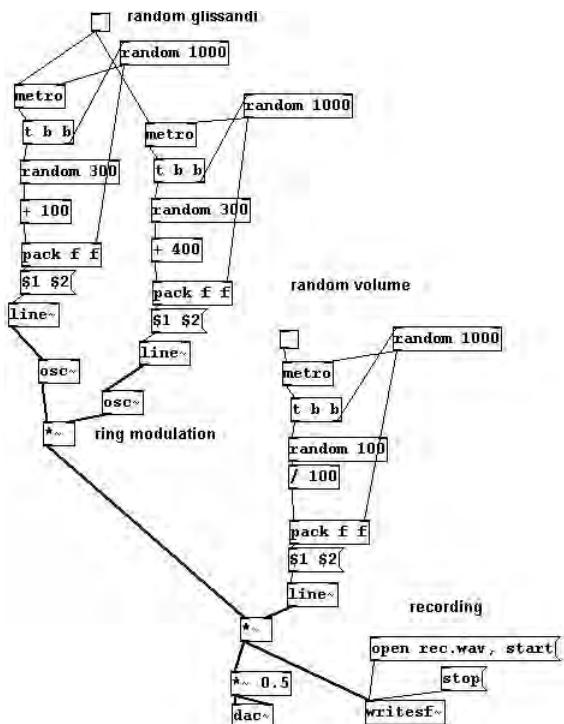
##### ◆ 5.1.1 THEORY

###### ► 5.1.1.1 Subpatches

You already learned how to create subpatches in Pd in 2.2.4.4. Now, you'll learn how to use them wisely.

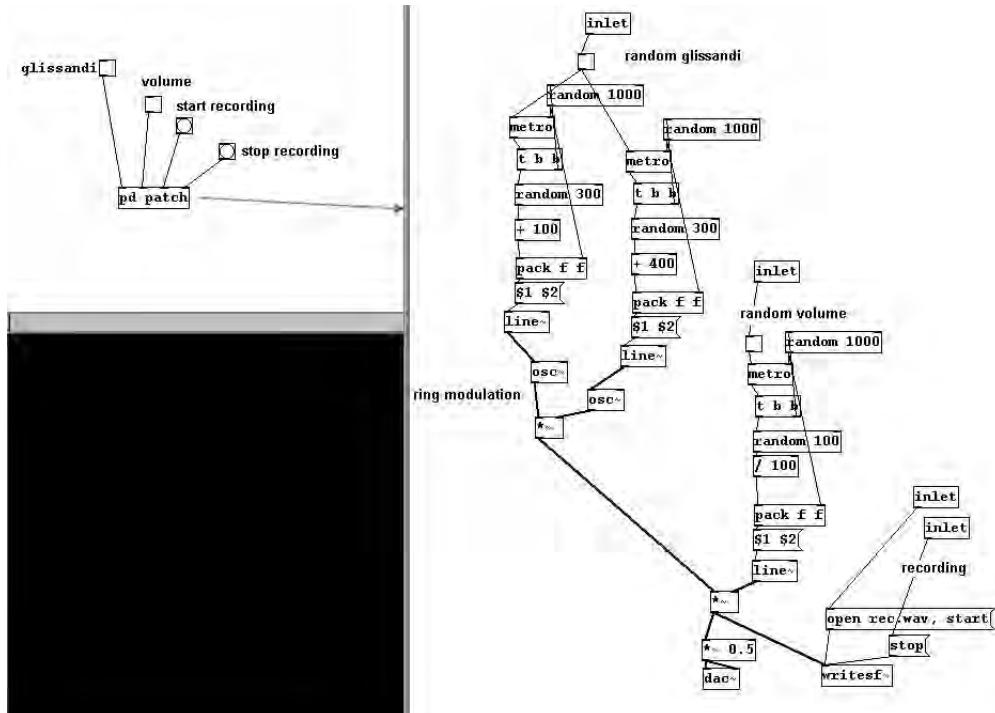
Let's say you have this patch:

patches/5-1-1-1-subpatch1.pd



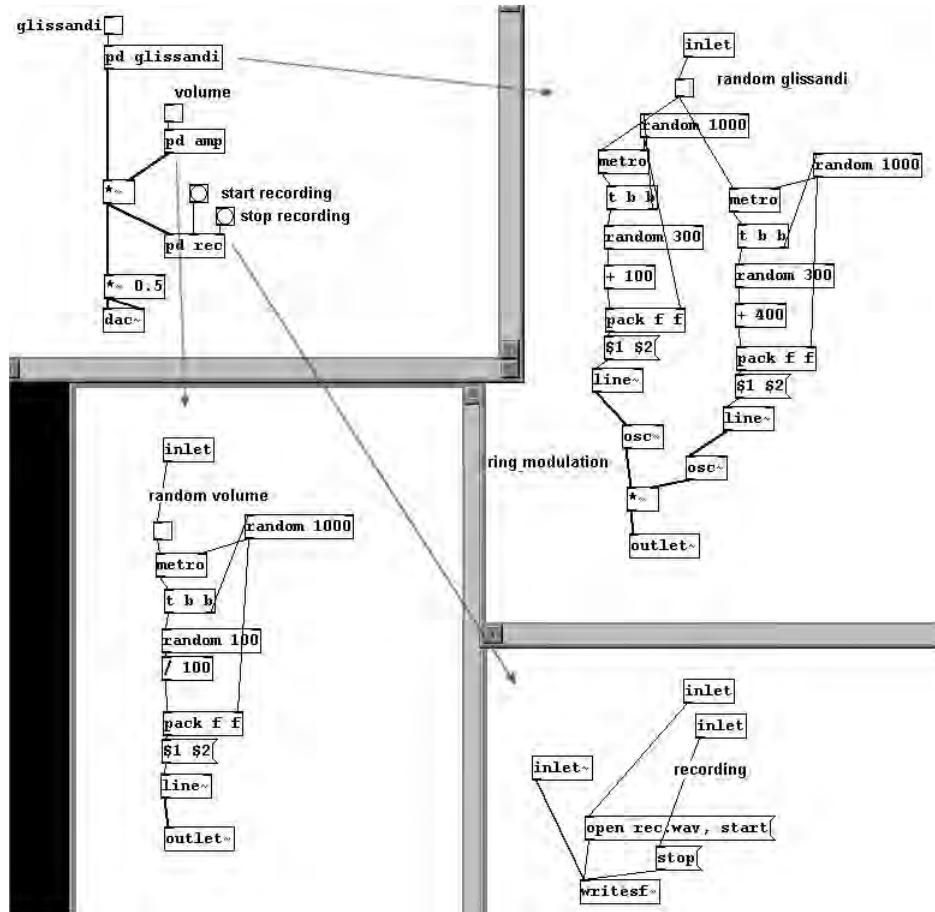
You could tidy it up by storing everything that doesn't require immediate access in a subpatch:

patches/5-1-1-1-subpatch2.pd



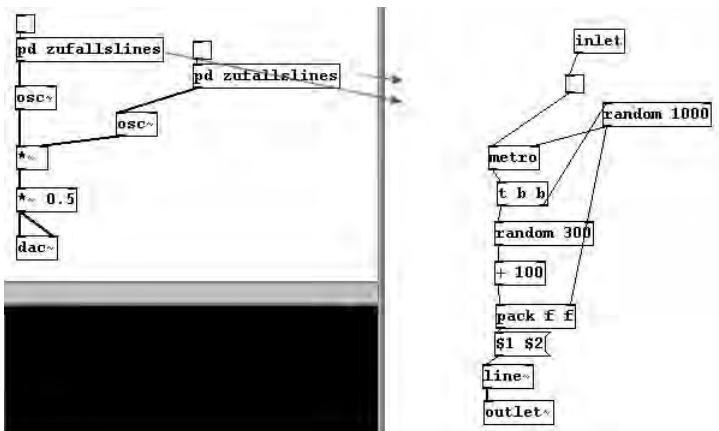
It also makes sense to store the contents in several subpatches, in case you want to edit a specific part of it later.

patches/5-1-1-1-subpatch3.pd



You could build a certain algorithm (here: random lines) for a patch that could be used in different places:

patches/5-1-1-1-module.pd

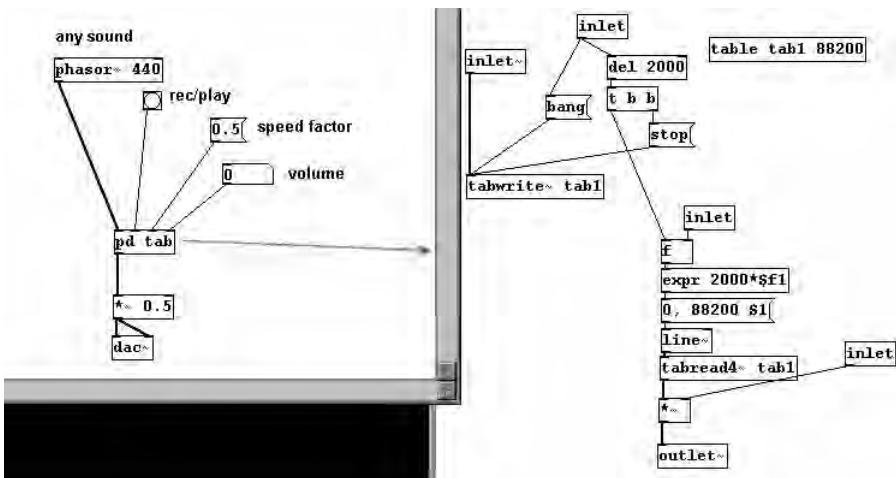


A subpatch like this is called a “module”.

#### ► 5.1.1.2 Abstractions

If you have a subpatch that can be used universally, like the following one, which records two seconds in an array, then plays back the sample at a certain speed with variable amplitude:

`patches/5-1-1-2-abstraction1.pd`



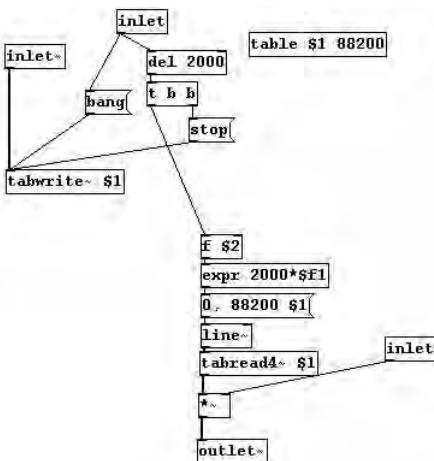
...and you want to use it in different locations, then a problem arises: once the subpatch has been duplicated, you have to rename the array and enter a new input (for the speed). Instead you could make an abstraction out of it; this is a patch that is stored as a separate file and can be retrieved with a multiple of variables.

Take the subpatch from before and save it as “record.pd”. Then open a new patch and save it as “main” in the same directory as “record.pd”. Then create an object called “record”:

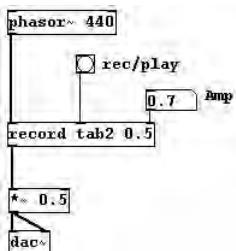
**record**

“record” does not actually exist as an object in Pd. But if a patch with this name (plus the suffix “.pd”) exists in the same directory, an object can be created that treats this patch like a subpatch.

An advantage of this lies in the variables. Reload the “record.pd” patch (**File → Open**). You can write variables in form of “\$1”, “\$2”, etc. inside objects. Let’s define variables for the array name and the speed:

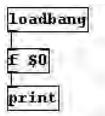


Now you can write the “record” object in the “main.pd” patch again, this time with two arguments for the two variables; the first argument is the array name and the second is the playback speed.

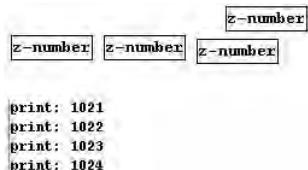


The volume, which we want to change in the main patch as well, is set as an inlet just as for a subpatch.

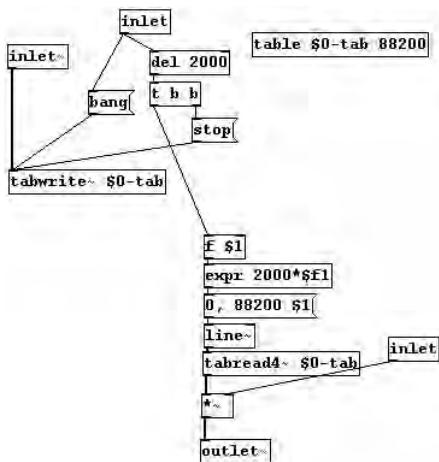
But we've still got the problem that the array has to be given a unique name every time we load "record" in the main patch. There is a special option in Pd to solve this that generates an individual random number that can be used as a name for a certain patch. This is generated by \$0. You could make the abstraction "z-number.pd":



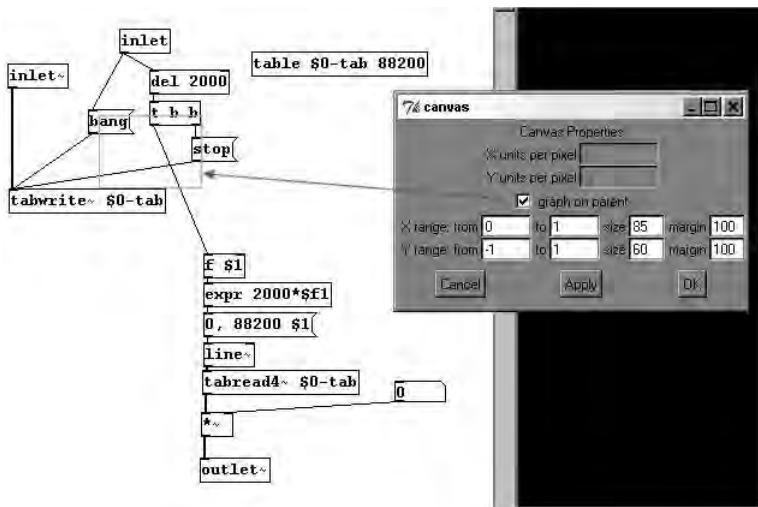
A random number with an offset of 1000 is generated, and it counts up from there. Now, every time you call up the abstraction another number will be generated:



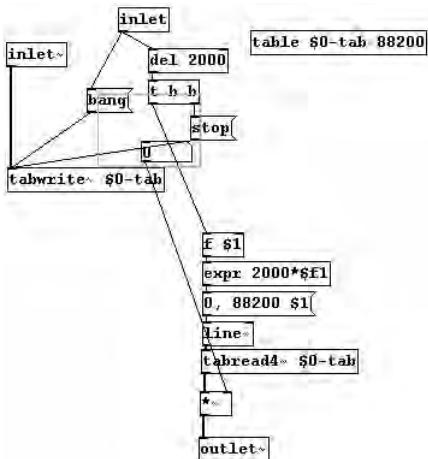
This is used to give the array a unique name. The conventional form is: \$0-[arrayname]. You still need to set a variable (\$1) for the speed. The volume should stay variable.



Finally, you can bring in graphic elements from an abstraction using the function "graph on parent". In the "record" patch, you can create a number box instead of an inlet for the volume, then right-click anywhere on the white surface → **Properties**. Check "graph on parent" and then click "apply":



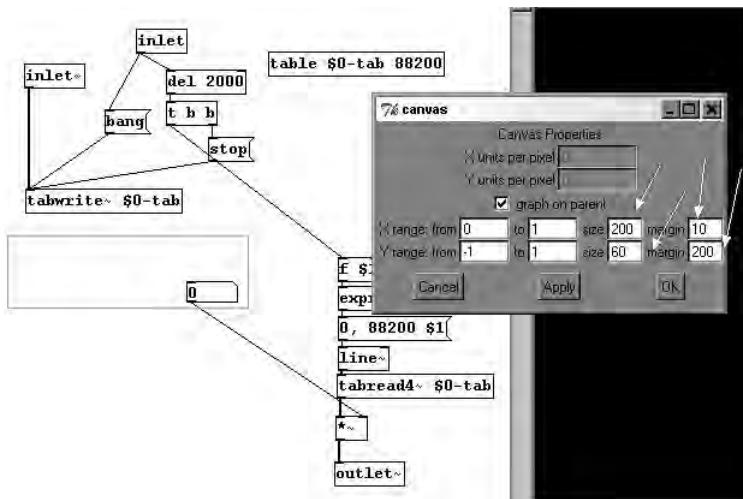
You should see a red rectangle. All GUI objects that are enclosed within this red rectangle will appear later in the object. Move the number box for the volume inside the red rectangle, save “record.pd”, and then create the object in the new patch:



In the new patch:

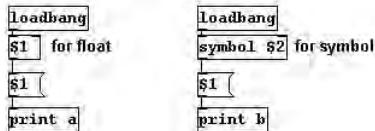


You can change the size and position of the red field here:



“Graph on Parent” also works with subpatches.

Two important points: you can view and change the contents of an abstraction in the main patch (simply click on the object in the main patch). However, changes you make here will not be saved! Only if you open the abstraction itself as a normal patch in Pd are the changes you make actually saved. The second point is that \$-variables can only be written in objects in an abstraction; \$1 in a message box would just have the normal meaning of an input in the message box (cf. 2.2.2.1.4). For example:



And, of course, if you copy the main patch and want to use it elsewhere (e.g., in another directory or on another computer) you have to copy the “record” patch as well. You make abstractions available at all times by saving them in the “extra” directory in the Pd directory. All patches found saved here are always available as abstractions. Similarly, you can create your own directory containing abstractions and configure it to load automatically. You can set a path for this under **File** → **Path**.

#### ► 5.1.1.3 Expanding Pd

Many useful objects not included in the original version of Pd have been developed by programmers around the world since its inception. These objects are called “externals”. Collections of externals are called “libraries”, e.g., the zexy library or the MaxLib library. You have to put them in the folder called “Extra” and also integrate them into

the startup process (**File → Startup**). A new version of Pd called “Pd-extended” already includes libraries that expand the original version.

Pd can also be expanded through the use of additional programs. GEM is the most well known and is used to process video data and video files in a manner similar to how Pd processes sound and sound files. Additionally, open sound control (OSC) data from other programs that also have an OSC connection can be used in Pd. The Pd versions of some programmers have been expanded to such an extreme degree that they are barely recognizable as versions of Pd.

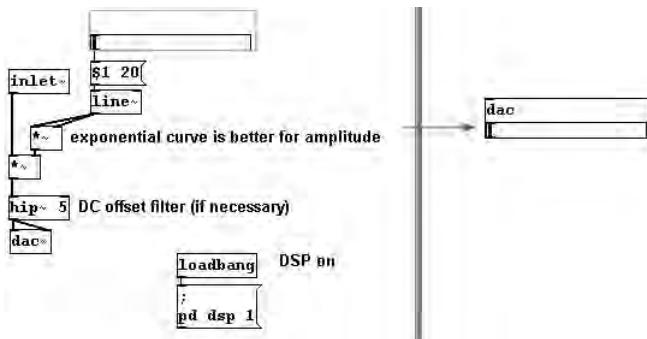
#### ◆ 5.1.2 APPLICATIONS

##### ► 5.1.2.1 Customize your Pd

As mentioned in the previous chapter, Pd can be customized to a large extent. Let’s create some useful abstractions.

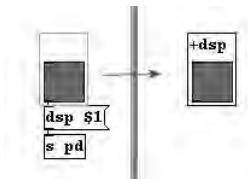
For normal programming needs, a “dac~” object with a built-in volume slider, now called “dac”:

patches/dac.pd



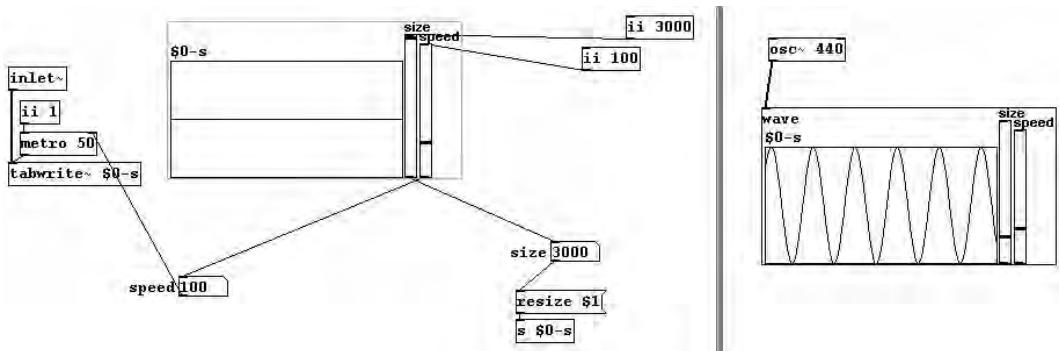
A DSP switch, called “+dsp”:

patches/+dsp.pd



Graphic representation of a waveform:

patches/wave.pd



### ► 5.1.2.2 More exercises

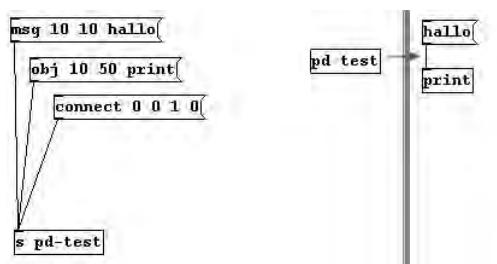
As abstractions:

- Integrate the DSP switch into the “dac” abstraction as well as a mute on/off button.
- Build a compressor.
- Build a graphic representation for an incoming spectrum.

### ◆ 5.1.3 APPENDIX

#### ► 5.1.3.1 Creating a patch automatically

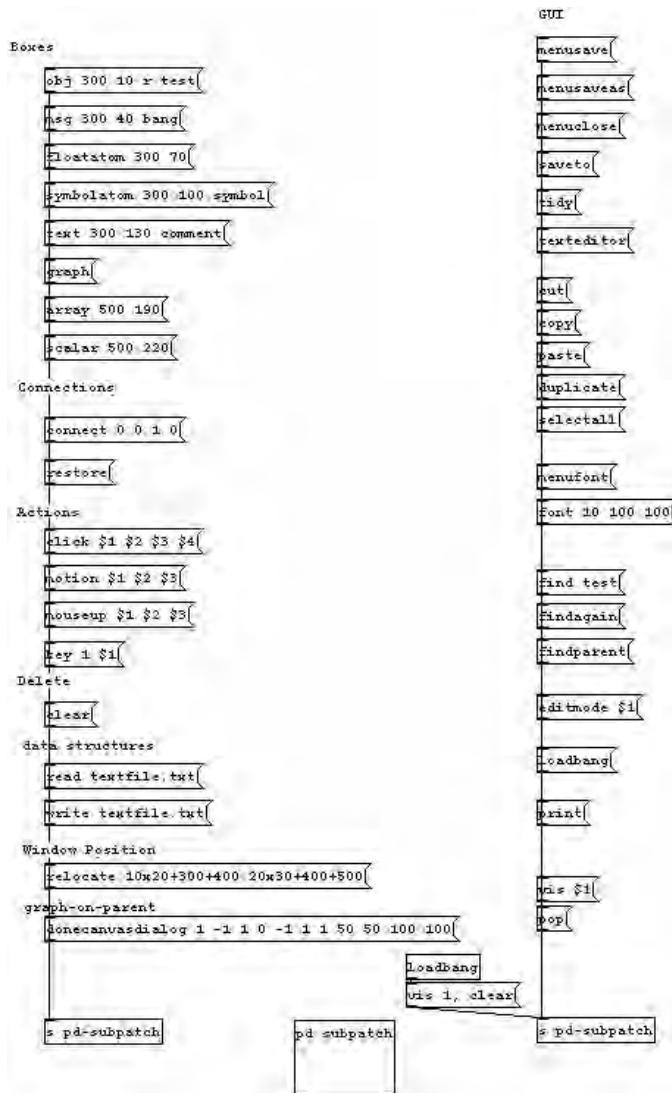
We have a patch with a subpatch called “test”. In the main patch, we send the following messages to the subpatch using “s pd-test” from top to bottom:



“obj” creates an object; you enter the x/y-coordinates and the name of the object with its arguments. The same goes for the message box. The connection works like this: “connect [object number] [outlet number] [object number] [inlet number]”. The objects are

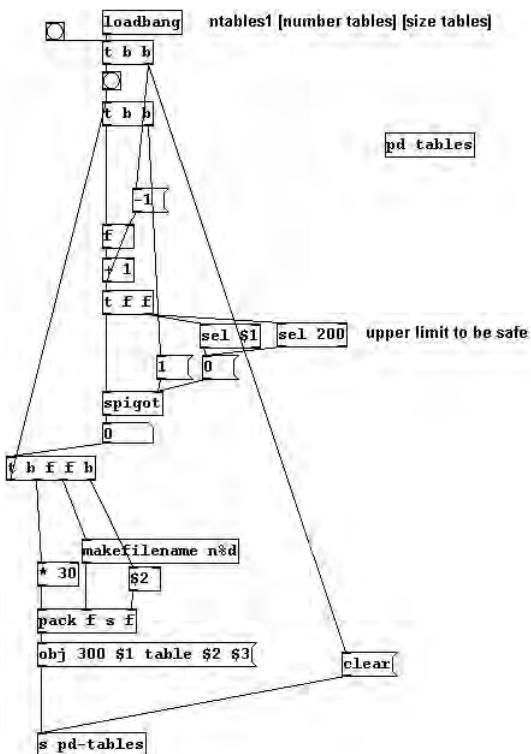
numbered in the order they are created, the outlets and inlets from left to right. Everything is initially set to 0. The message “clear” deletes the content of the subpatch.

Here is an overview of all the commands. At present (June 2008) some of these don’t yet work:



This makes it possible—albeit in a fairly complicated manner—to, say, create a certain number of arrays with an abstraction, which cannot be accomplished in Pd any other way (cf. 3.2.3.1):

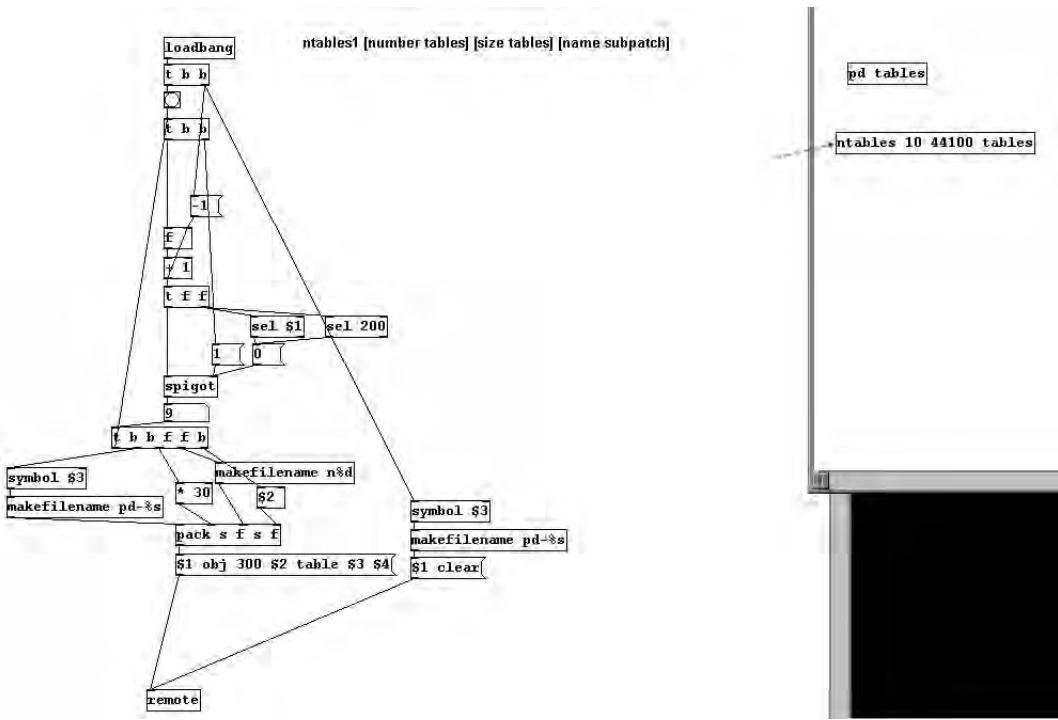
patches/ntables.pd



With this abstraction, you can create ten arrays in the subpatch with a size of 44100:

**ntables1 10 44100**

In this way, you could also create the subpatch in your main patch:



◆ 5.1.4 FOR THOSE ESPECIALLY INTERESTED

► 5.1.4.1 Writing your own objects

...is naturally also possible. Pd is merely a surface for a program in the programming language C, which allows you to write your own objects (“externals”). Here is a guide to help you do this: <http://iem.at/pd/externals-HOWTO/>

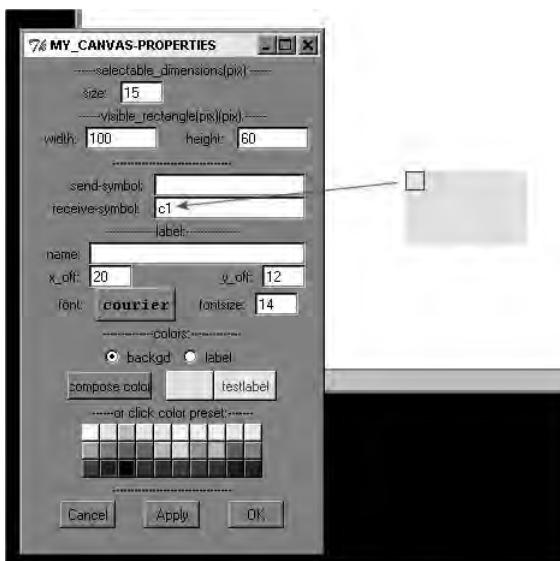
## 5.2 Visuals

### ◆ 5.2.1 THEORY

#### ► 5.2.1.1 Pd is visual and this can be programmed

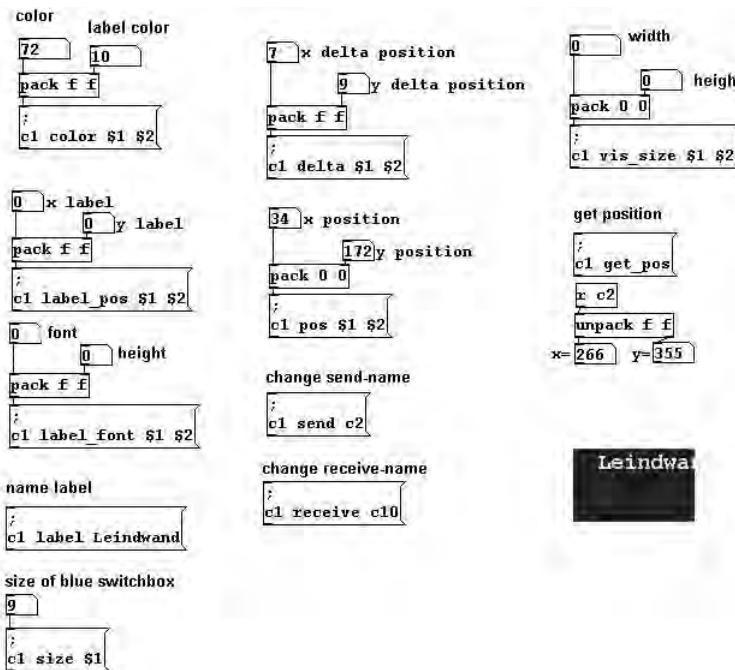
The tools for visual representation were already introduced in 2.2.4.3. Here, we'll examine how to control a canvas.

Create a canvas (**Put** → **Canvas**) and give it the receive symbol “c1” under properties (right-click on the box in the canvas's upper left):



Now you can send the canvas messages:

patches/5-2-1-1-canvas.pd



#### ◆ 5.2.2 APPLICATIONS

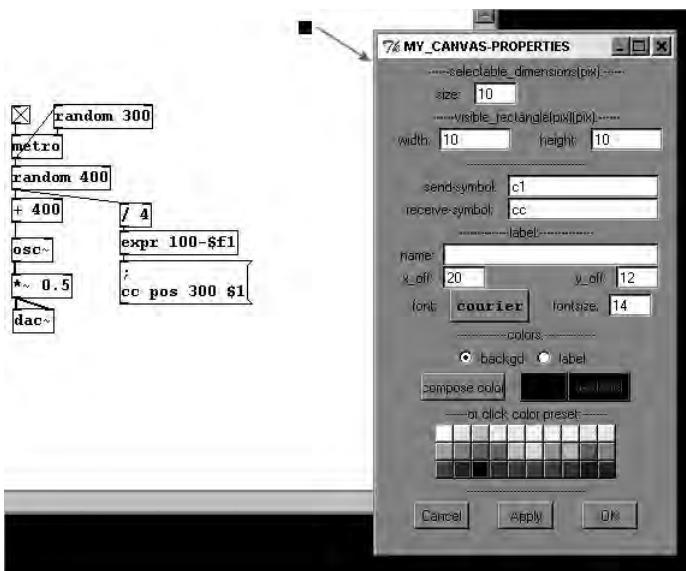
##### ► 5.2.2.1 Main patch window and subpatches

This should be clear from the previous material: a tidy patch comprises the main window and subpatches (cf. 3.4.2.4 and 5.1.1.1). In this manual, I've usually refrained from this sort of organization for the sake of clarity (it's easier to comprehend a graphic when all the required information is present in just one window).

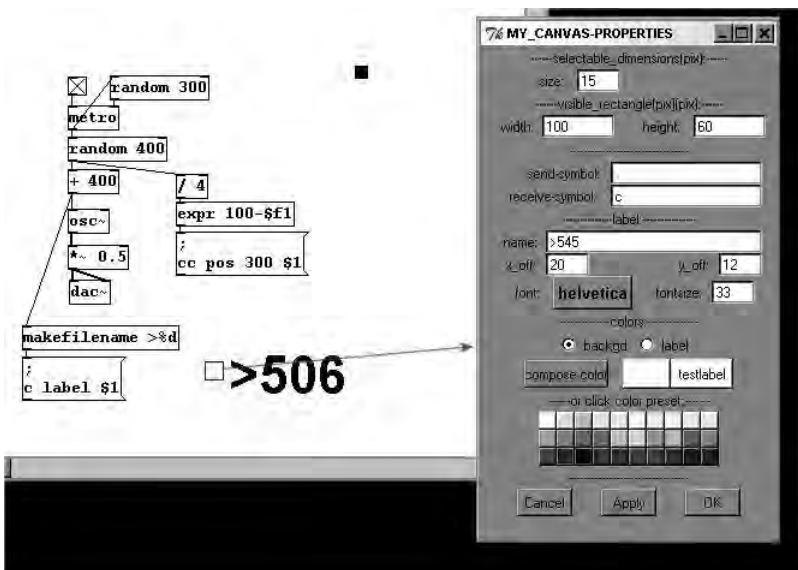
##### ► 5.2.2.2 Canvasses as display

The colorful elements in canvasses can greatly enhance the clarity of your patches (cf. 3.4.2.4). For displaying functions, they can also be used in variable manner. For example, you could display the results of a random generator like this:

patches/5-2-2-2-canvas-display.pd



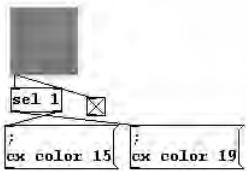
You could also display a number in a larger size (however, a symbol must always accompany the number; here, a “>” is used):



The possibilities are nearly endless; you could even produce complete graphic scores using canvasses.

### ► 5.2.2.3 Canvases as expanded GUI

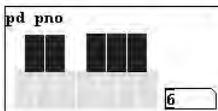
Canvases can be laid over other GUI objects; the order in which they are created is critical. You could make your own toggles:



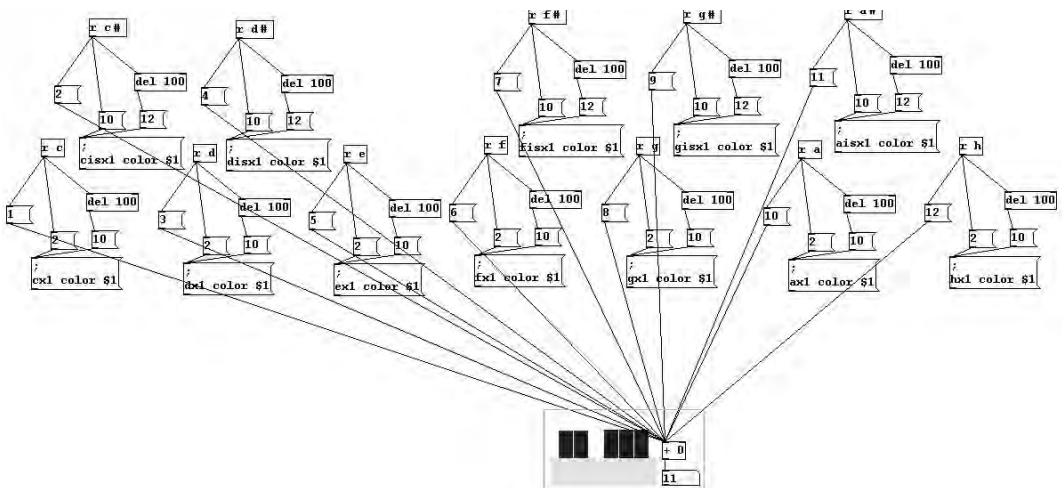
First, a toggle with a size of 55 was created and attached to "sel 1". Then, a canvas with a size of 55 and a receive symbol "cx" was created and superimposed directly on top of the toggle (the order of creation is essential; otherwise, the canvas disappears behind the toggle). The "sel 1" object and all the rest could be separated into a subpatch and an internal "send" object for the toggle could be used.

Here again, the possibilities are endless. To name just one example, you could replace an octave of piano keys like this (clearly, I am not the most talented Pd graphic designer):

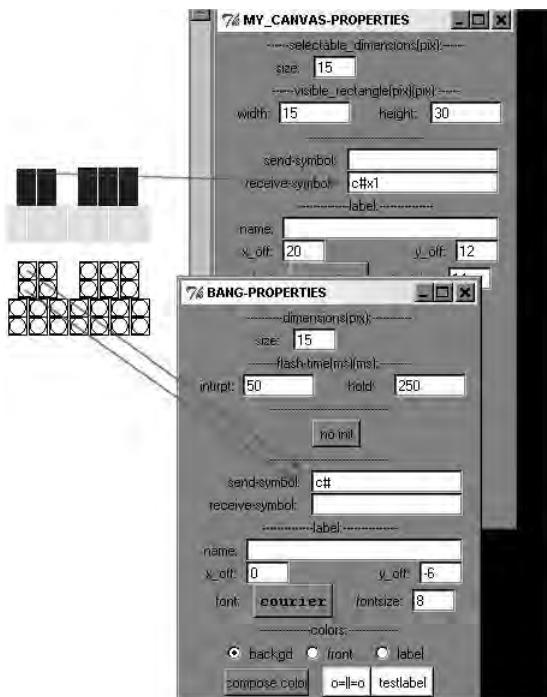
patches/5-2-2-3-piano-display.pd



The subpatch contains:



Behind the ‘piano keys’ is a series of bangs:



#### ► 5.2.2.4 More exercises

- Create a stopwatch.
- Create a graphic for a running 5/8 meter, i.e., a visual click track.

#### ◆ 5.2.3 APPENDIX

##### ► 5.2.3.1 Data structures

Data structures are an entire family of graphics in Pd.

You can place graphic elements in a subpatch. First you create the subpatch “graphic” and define variables and a graphic for this subpatch. This is called a “template”. It contains the variables with “struct”; for its argument, enter the template’s name (here “g1”) and then pairs comprising type and name—in this case, float x float y float q. “float” is the type (a decimal number); x, y, and q are freely chosen names.

```
pd graphic
```

```
struct g1 float x float y float q
```

The graphic can be defined with the objects “drawcurve”, “drawpolygon”, “filledcurve”, and “filledpolygon”. Let’s use “filledpolygon” for our first example. A polygon is a geometric shape with many sides. The arguments from left to right are interior color, perimeter color, perimeter width, pairs of coordinate points (starting from upper left and proceeding clockwise).

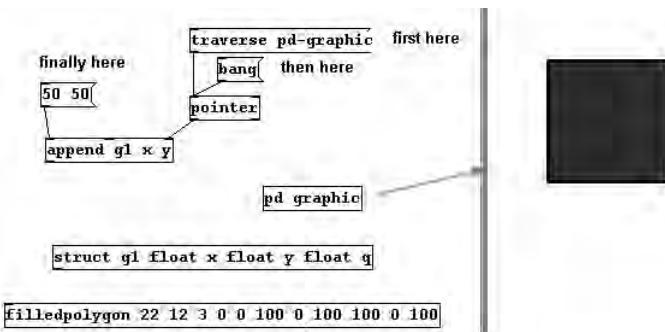
```
pd graphic
```

```
struct g1 float x float y float q
```

```
filledpolygon 22 12 3 0 0 100 0 100 100 0 100
```

To create a graphic, we need “append”. Its first argument is the template’s name and then possible variables—x and y should always be used. The right input is the place in the subpatch where the graphic is to be placed. You have to imagine that graphic elements in a subpatch are ordered one after another as in a list. “append” must first know the place in the list that the “pointer” object has named. Give “pointer” the Message “traverse pd-graphic”; this causes “pointer” to go to the beginning of the list. A “bang” message will send this place on the list (to “append”). Then give data for the variables into the “append” object’s left inlet.

patches/5-2-3-1-data-structures1.pd

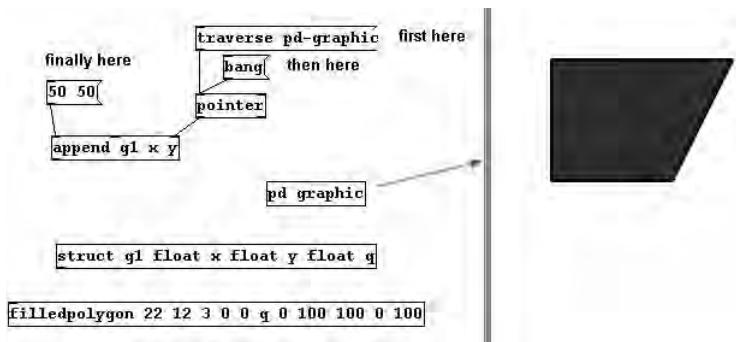


Just for the sake of clarity: “x” and “y” are special names for a graphic element used in data structures. It defines an absolute position, 50/50. You’ve now created a “filledpolygon” with the interior color 22, the perimeter color 12, the perimeter width 3, and with 4 points: upper left at a distance of 0/0 from the absolute position, upper right at a distance of 100/0 from the absolute position, lower right at 100/100, and lower left at 0/100. If you simply removed the last two numbers from “filledpolygon”, you’d be left with a triangle. You can apply this same method to create a polygon with any number of sides.

“traverse pd-graphic” basically takes you right to the beginning of the subpatch “graphic”; a “bang” sends this position to “append” and, when we send it data with the necessary variables, “append” creates the graphic at this position.

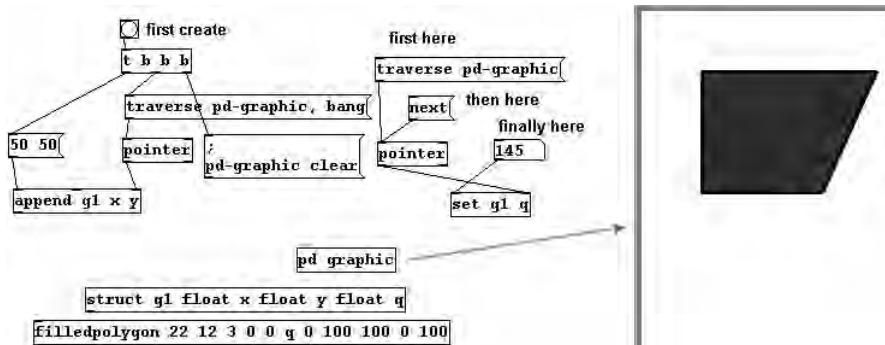
But you could also change something about this graphic. You defined the variable “q” (a float) for the template “g1”. Now enter the following for the “filledpolygon”: “filledpolygon 22 12 3 0 0 q 0 100 100 0 100”. One point has just one variable. The point disappears and you now have a triangle. This variable can be set using “append”:

patches/5-2-3-1-data-structures2.pd



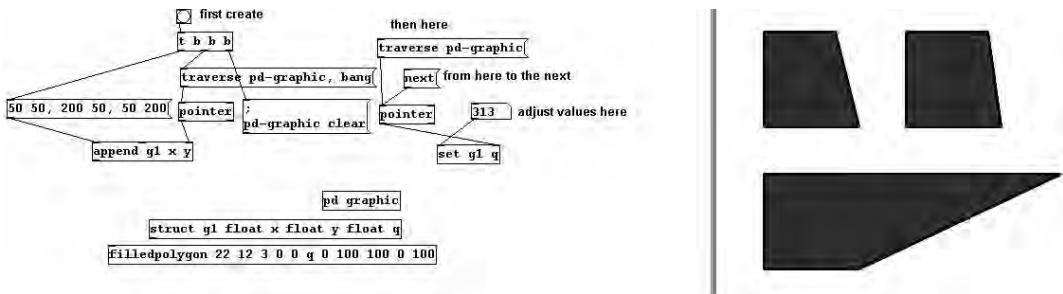
...but also by using “set”. This must also receive the information from the pointer regarding the position where something is to be changed in the subpatch. In this case, it is one step after the beginning. Use “traverse pd-graphic” to go all the way to the beginning. The beginning is empty; use “next” to go to the next graphic. Now you can enter a value for q in the left input.

patches/5-2-3-1-data-structures3.pd



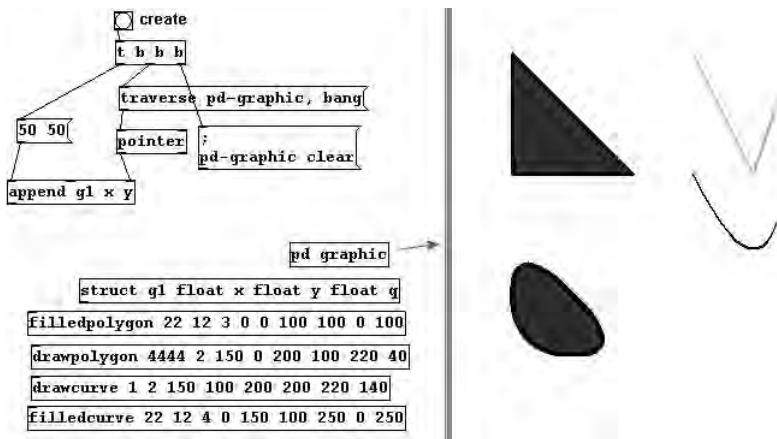
Many graphics can be made simply by sending messages to “append”. Use “next” to move from one graphic to the next.

patches/5-2-3-1-data-structures4.pd



Regarding the other graphic elements: “drawpolygon” is just a line with angles; the first argument with the interior color is omitted. The same is true of “drawcurve”, except that its angles are rounded. “filledcurve”, however, results in a closed shape and the first argument is again reserved for the interior color.

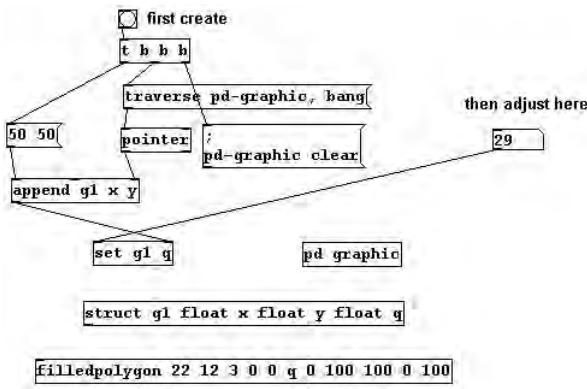
patches/5-2-3-1-data-structures5.pd



Another important point: the points (vortices) of a graphic that have variables can be changed with the mouse; the cursor changes its shape at the point, meaning you can now click and drag to modify the graphic's shape.

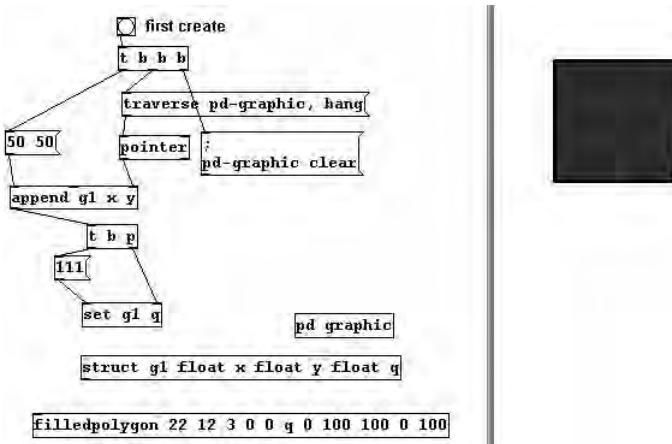
If you use “append” to design a graphic object, the output is a new pointer for this new object (as with “next”):

patches/5-2-3-1-data-structures6.pd



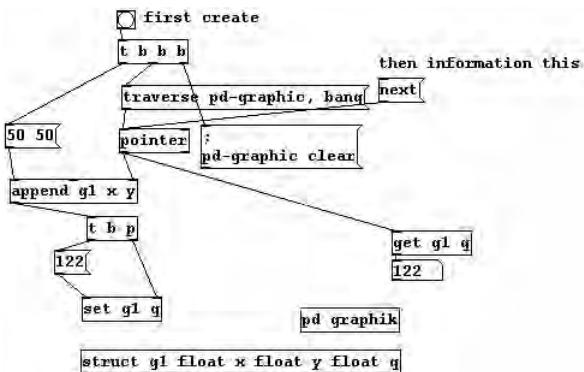
A pointer is a type of information, e.g., for “trigger”:

patches/5-2-3-1-data-structures7.pd



You can use “get” to receive information from graphic elements that are attached to pointers:

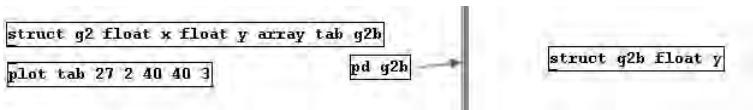
patches/5-2-3-1-data-structures8.pd



Last, you can create a (graphic) array using data structures. The array is defined in “struct” with a name and another allocated template. Use “plot” to define the color, width, starting point (x/y), and distance between points for this array.

```
struct g2 float x float y array tab g2b
plot tab 27 2 40 40 3
```

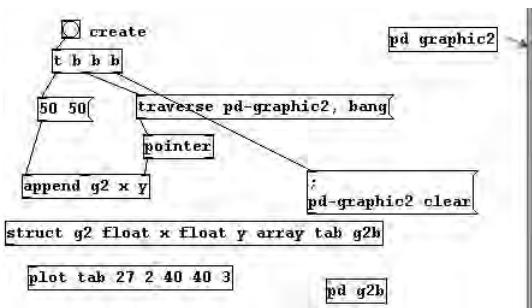
Another subpatch must contain the other template that determines the array’s variables:



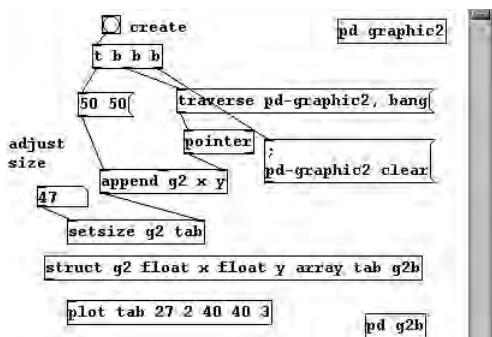
The variable “y” is automatically understood as the height of the array. This variable is necessary to create the array correctly.

Here is the created array:

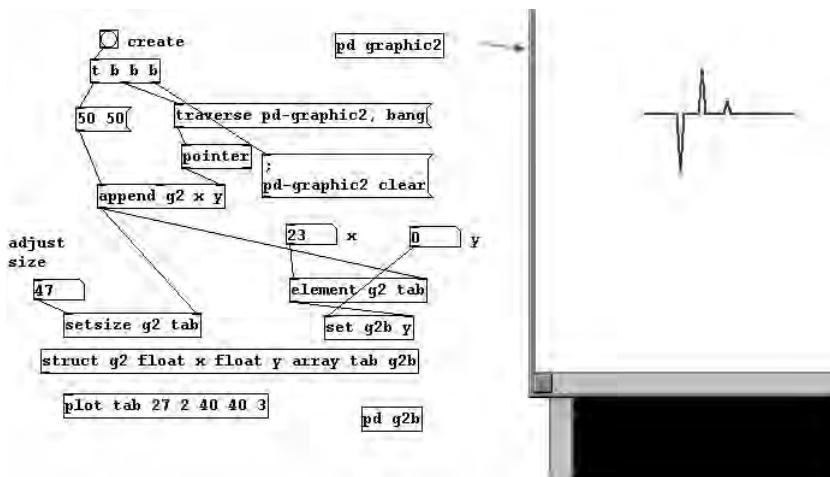
patches/5-2-3-1-data-structures9.pd



You can change the size with “setszie”:



The variable “y” has to be changed somewhat indirectly; use “element” to access it.



The form of data structures may change in the future (date of writing: June 2008). Furthermore, additional special functions for data structures can be found in the original Pd documentation.

◆ 5.2.4 FOR THOSE ESPECIALLY INTERESTED

► 5.2.4.1 GEM

As mentioned previously, there is an additional program for Pd called GEM that is used for video generation and editing.

<http://gem.iem.at>

## Afterword

Now that you've finished this tutorial and tried out all the techniques presented, hopefully you're eager to combine all the objects with each other and make your own discoveries—after all, that's exactly the intention behind Pd's design.

There is, of course, virtually no end to further study of the concepts presented here. Many other books about digital sound processing await your perusal—especially "Theory and Techniques of Electronic Music" by Pd's main designer Miller Puckette. It would certainly be advisable to further increase your proficiency with and knowledge of acoustics, studio recording techniques, and programming. You may also want to consider acquiring fluency with the basics of a sequencing program for programming sound. However, enthusiasm, artistic satisfaction, and aesthetic reflection are still the most important things.

For any further questions about Pd, I encourage you to contact the "Pd-list" Pd community. This book was written according to Pd version 0.39 from late 2007. Hopefully the information contained here will not be outdated too quickly.

Johannes Kreidler

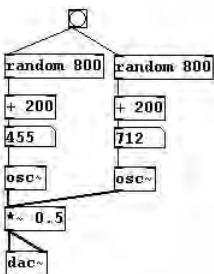
## Appendix A. Solutions

Here you'll find the suggested solutions to the "More exercises" sections found at the end of every chapter. These are by no means comprehensible; in most cases, other correct solutions are possible.

### ▷ 2.2.1.2.8

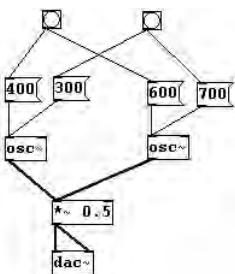
- a) Two simultaneous random melodies:

patches/a-1-two-randommelodies.pd



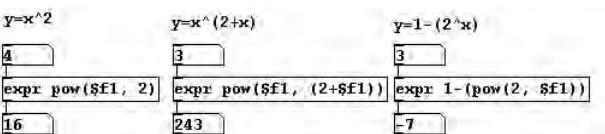
- b) Select (any) two different intervals using two bangs:

patches/a-2-two-intervals.pd



- c) Use "expr" to calculate exponential functions, e.g.,  $y = x^2$  or  $y = x^{(2+x)}$  or  $y = 1 - (2^x)$ :

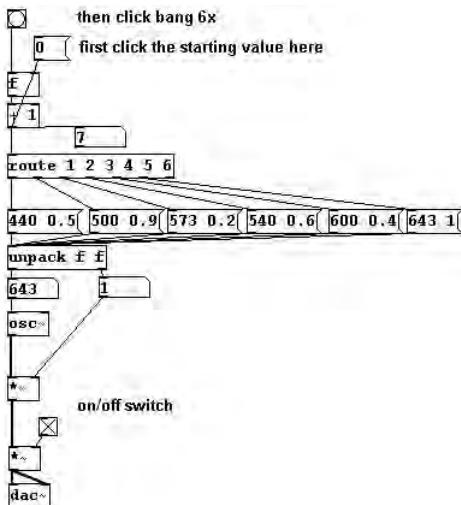
patches/a-3-exponentialfunctions.pd



▷ 2.2.2.2.6

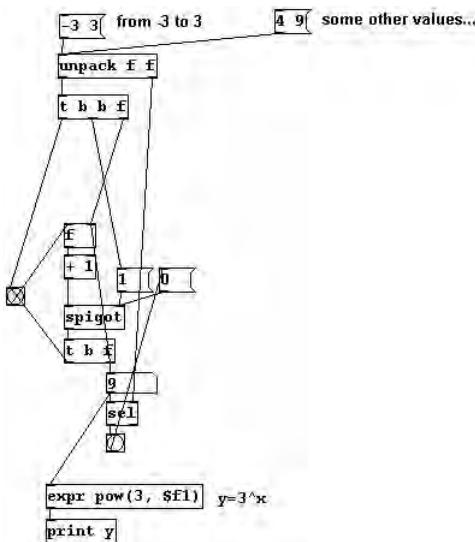
a) A sequence of lists with pitch and volume:

patches/a-4-listsequence.pd



b) A function that, given a list of two numbers (the start and end values for the x-domain), can calculate the y-values—e.g., to calculate the values of the function  $y = 3^x$  for the range from  $x = -2$  to  $x = 4$ :

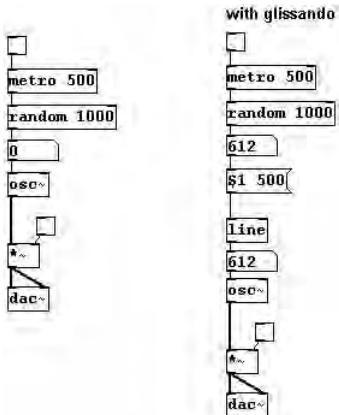
patches/a-5-functionpart.pd



▷ 2.2.3.2.9

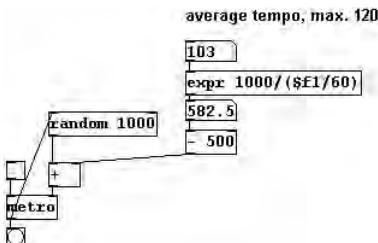
- a) A random melody that jumps to the next tone every 0.5 seconds (alternatively: as a glissando rather than a jump):

patches/a-6-randommelody500.pd



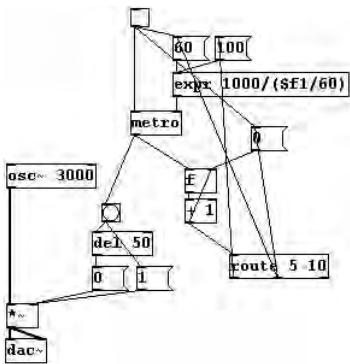
- b) A metronome with irregular random rhythms (for which you can set the average tempo):

patches/a-7-irregmetro.pd



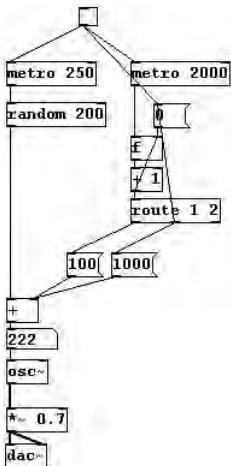
- c) A metronome, that beats five times at tempo quarter note = 60 and five times at quarter note = 100:

patches/a-8-twometro.pd



d) A melody that changes between very high and very low registers every two seconds:

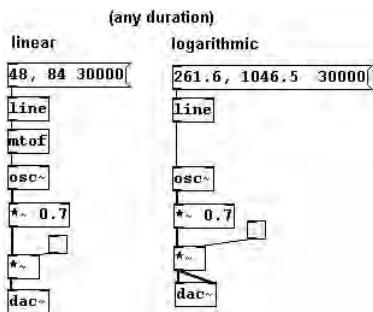
patches/a-9-highlowmelody.pd



▷ **3.1.1.2.2**

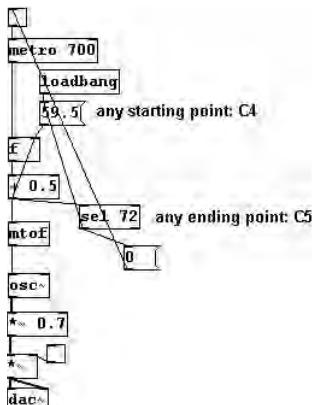
a) Two glissandi: one linear and one logarithmic (to the human ear) from C3 to C6.

patches/a-10-linloggliss.pd



b) A quarter-tone scale:

patches/a-11-quarterton.pd

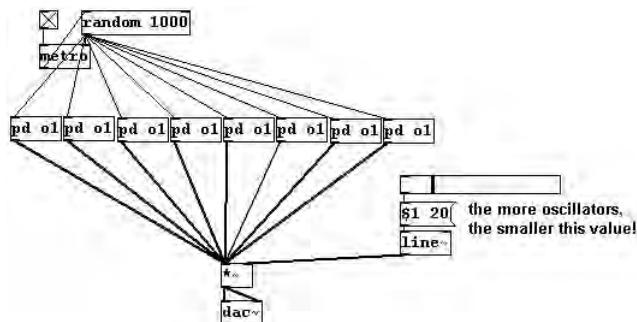


▷ **3.1.2.2.5**

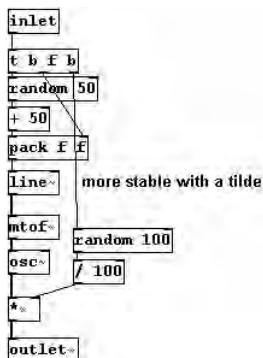
a) (random) glissando chords that also have random volume changes for each individual tone:

patches/a-12-randchord.pd

This main patch....:

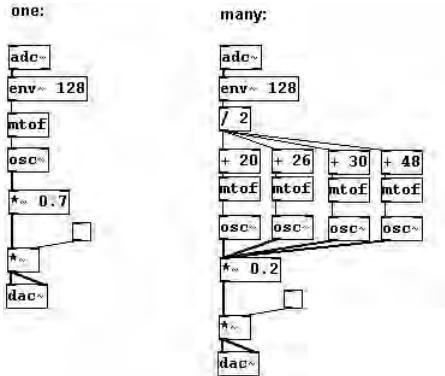


...with this subpatch "o1":



b) The volume of the microphone input controls an oscillator's pitch (alternatively: many oscillators' pitches having different frequency offsets):

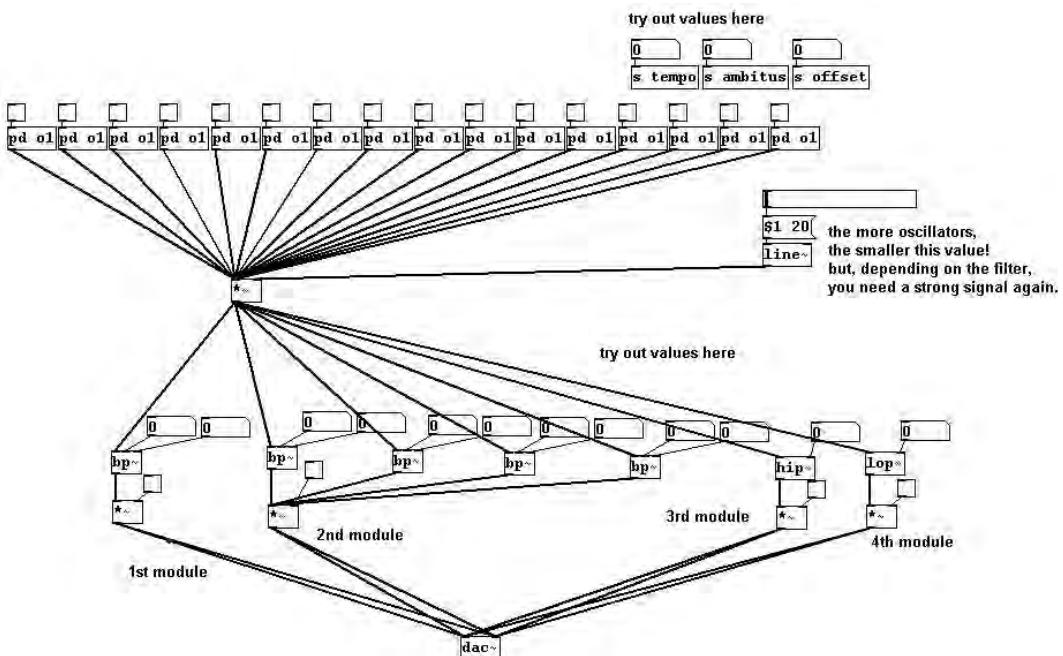
patches/a-13-adccampcont.pd



### ► 3.3.2.3

Experiment with filtering the “glissando orchestra” (3.1.2.2.4):

patches/a-14-orchestrafilter.pd

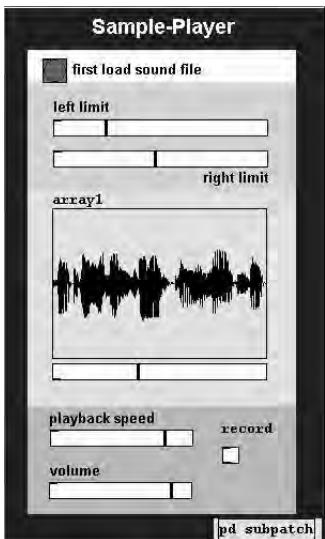


► 3.4.2.11

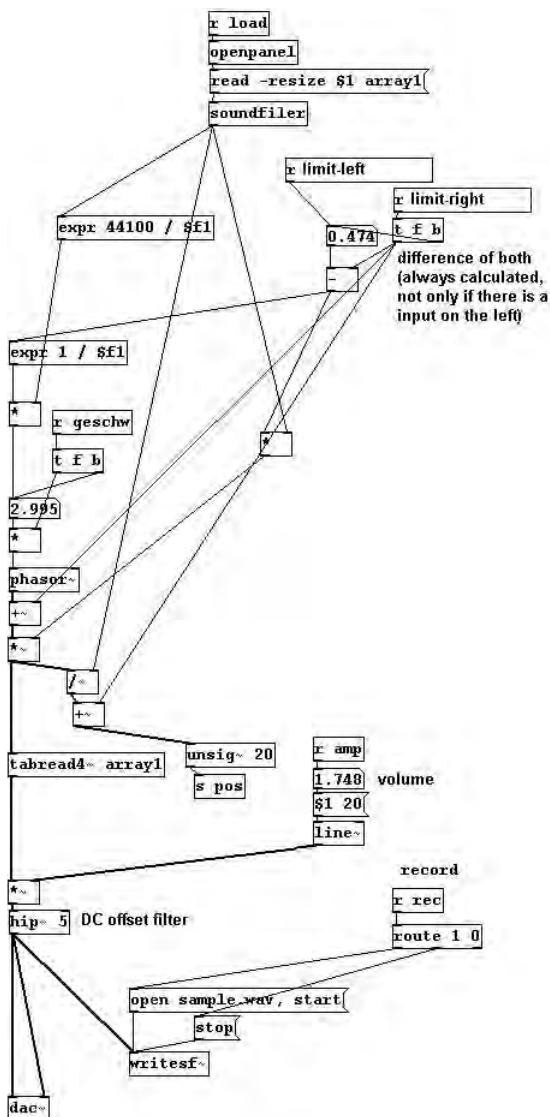
a) Build a record function into the sample player:

patches/a-15-recsampler.pd

A toggle is added to the main patch of 3.4.2.4 that sends to “rec”.

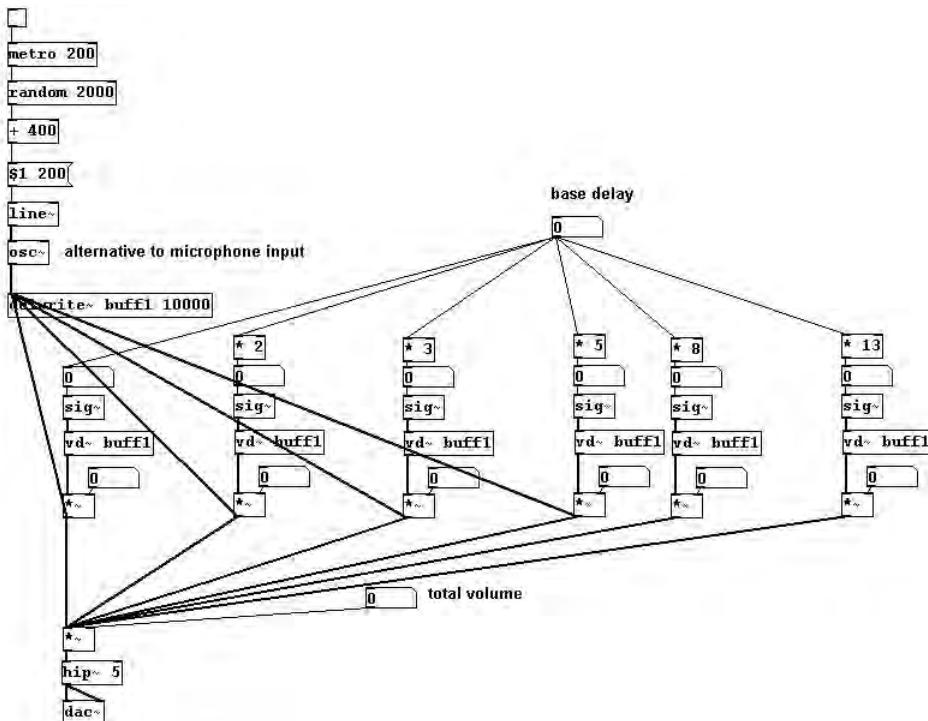


In the subpatch:



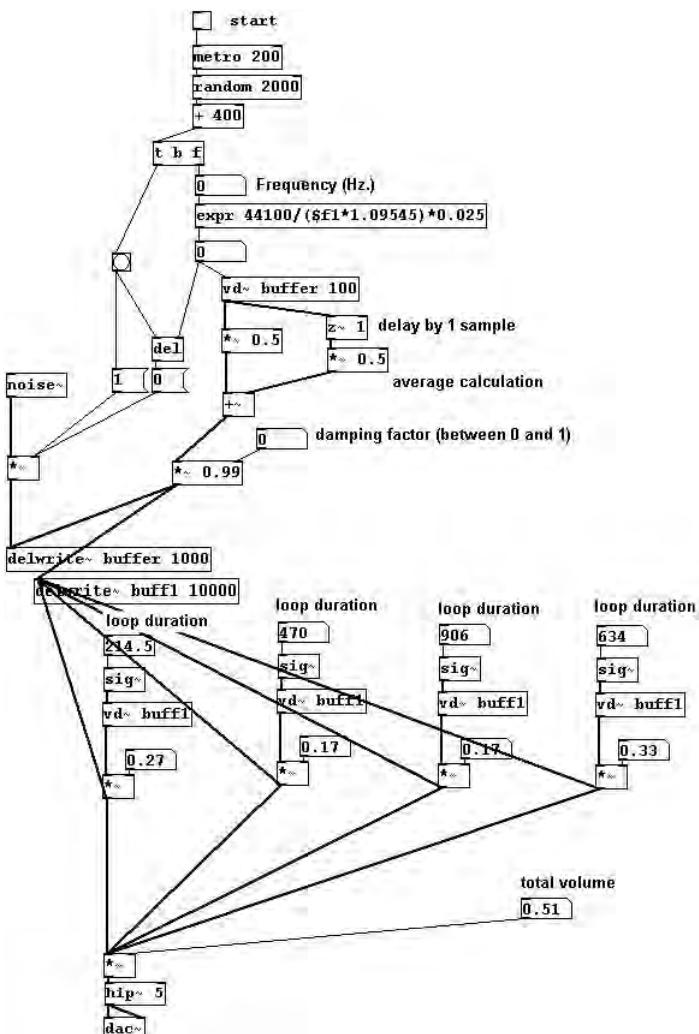
b) Create a patch for reverb or a texture with different delay times for the input signal, e.g., with multiples of the Fibonacci series (in which the next number is always the sum of the previous two: 0 1 1 2 3 5 8 13):

patches/a-16-fibodelay.pd



c) Use different Karplus-Strong sounds to make textures of varying densities:

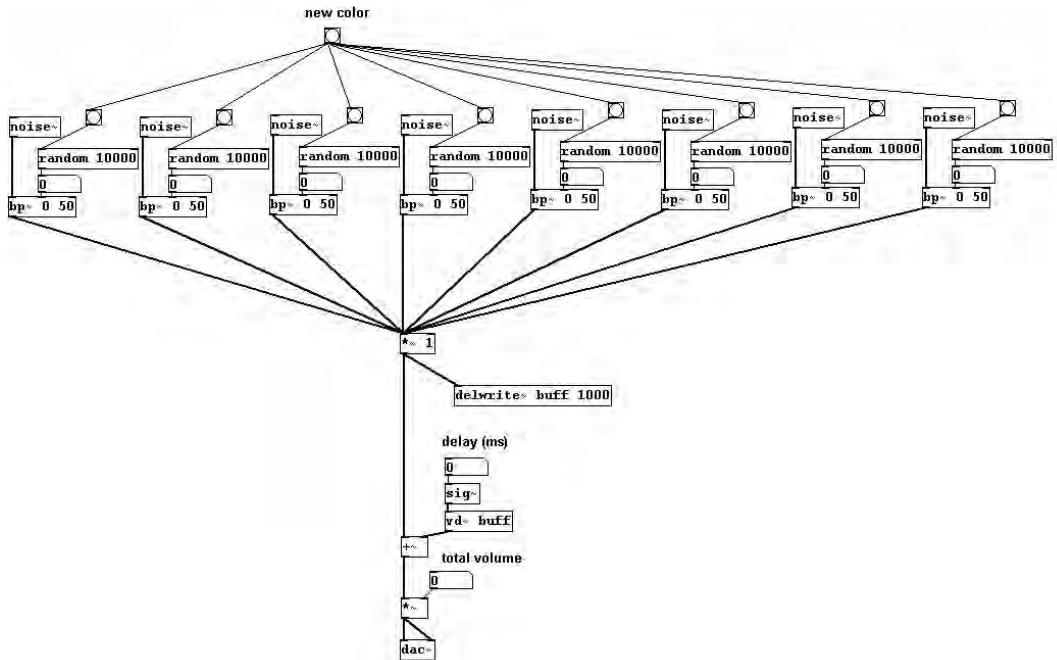
patches/a-17-karplus-text.pd



d) Apply a comb filter to patches presented in the previous sections:

patches/a-18-combfilteruse.pd

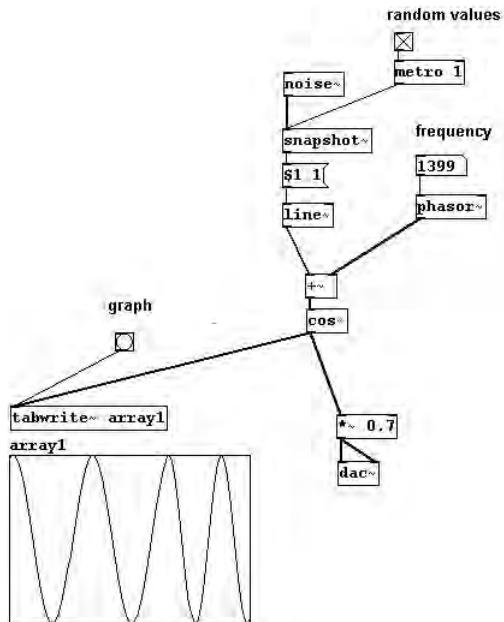
E.g., using the “filter colors” (3.3.2.1) (use a very short delay, e.g., 15 ms):



► 3.5.2.4

A wave that changes constantly:

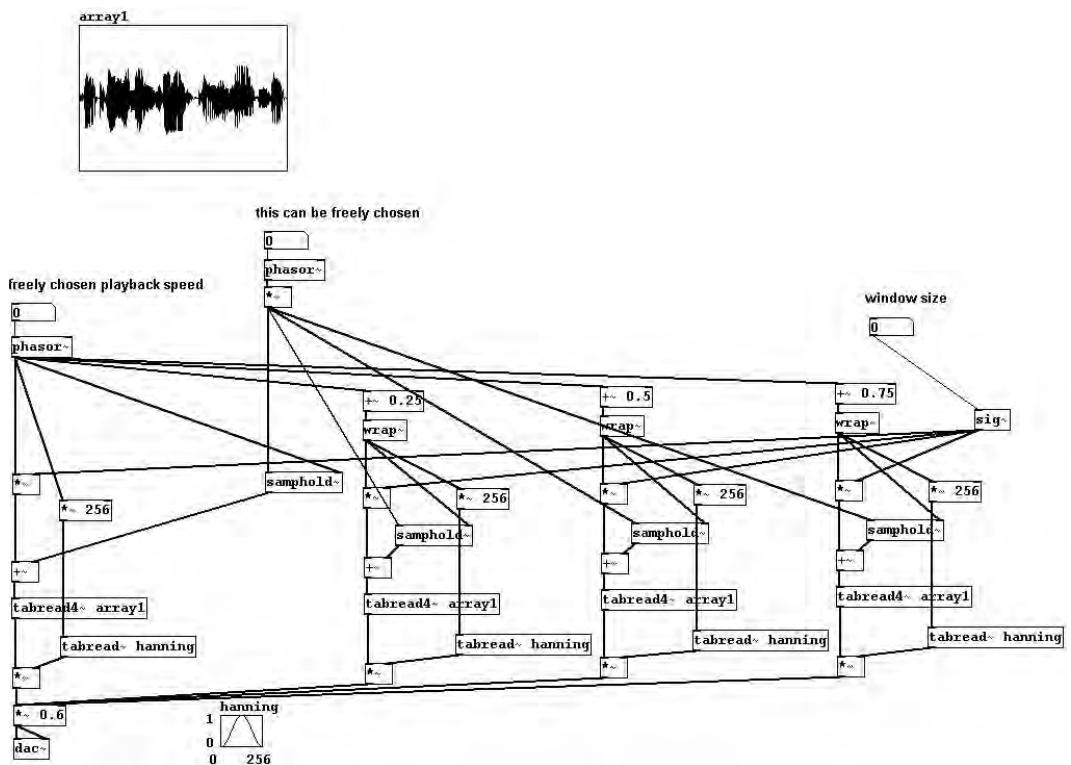
patches/a-19-wavechange.pd



### ► 3.7.2.3

Four readers, each with a variable window size. Experiment!

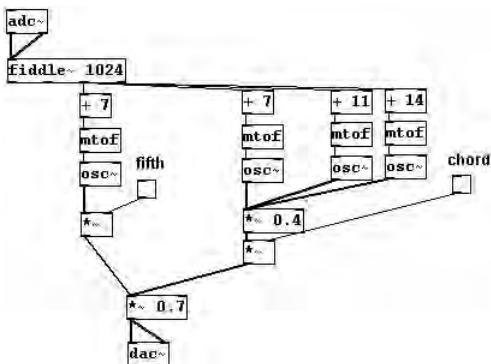
patches/a-20-four-reader.pd



### ► 3.8.3.5

Instead of simply ‘tracing’ the microphone input, a parallel voice a perfect fifth away or even a whole parallel chord:

patches/a-21-followers.pd

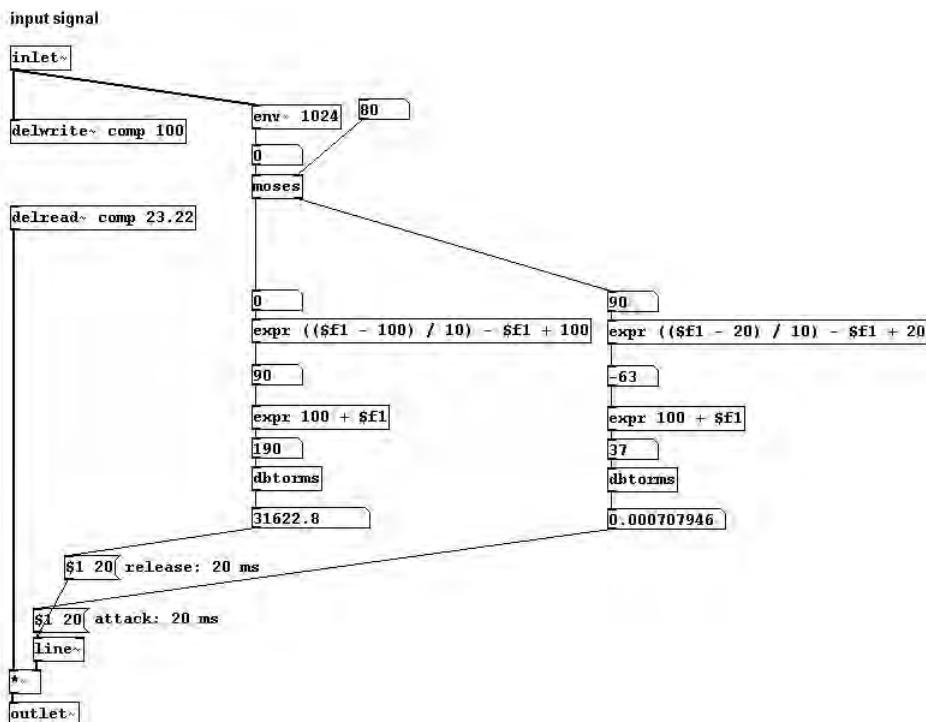


### ► 3.9.2.2

a) An “expander”—small differences in amplitude are turned into large differences: For this, you just need to use a value larger than the input signal as a reference in the compressor patch (3.9.1.2).

b) A volume inversion: quiet becomes loud and loud becomes quiet.

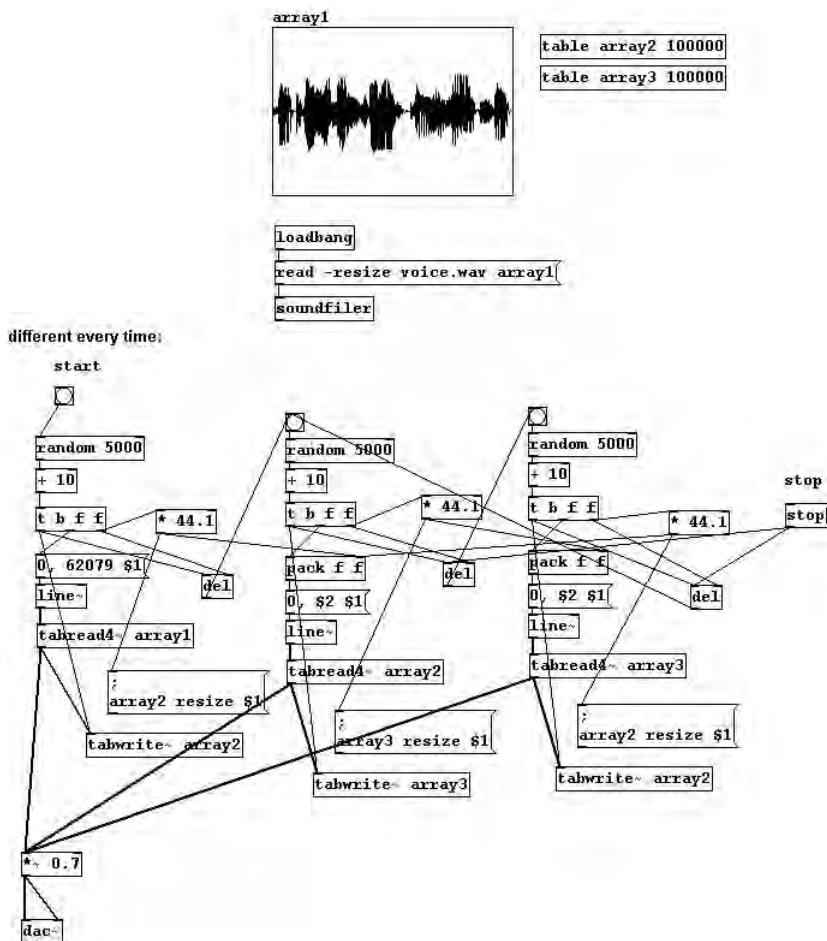
patches/a-22-ampinvert.pd



### ► 4.1.2.3

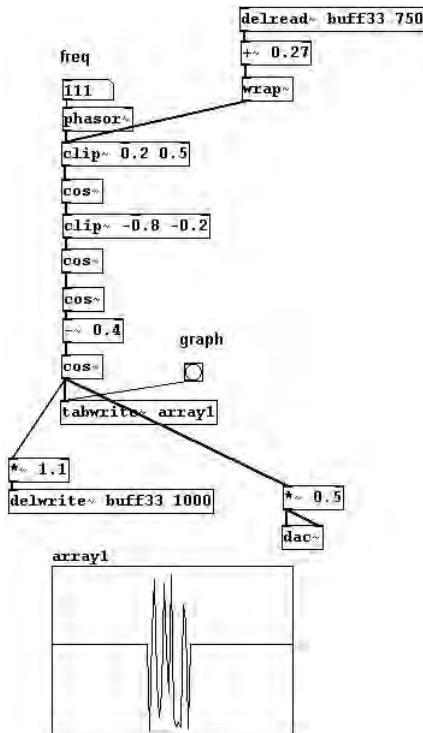
- a) Record a sample and play it back at the wrong speed. Record this 'wrong' playback, play it back, record, and so forth. Try this while (1) playing back the sample with the same 'wrong' speed and (2) with a different 'wrong' speed.

patches/a-23-sample-false.pd



b) A recursive wave shaping algorithm using delay (i.e. an algorithm in which the output is fed back as input):

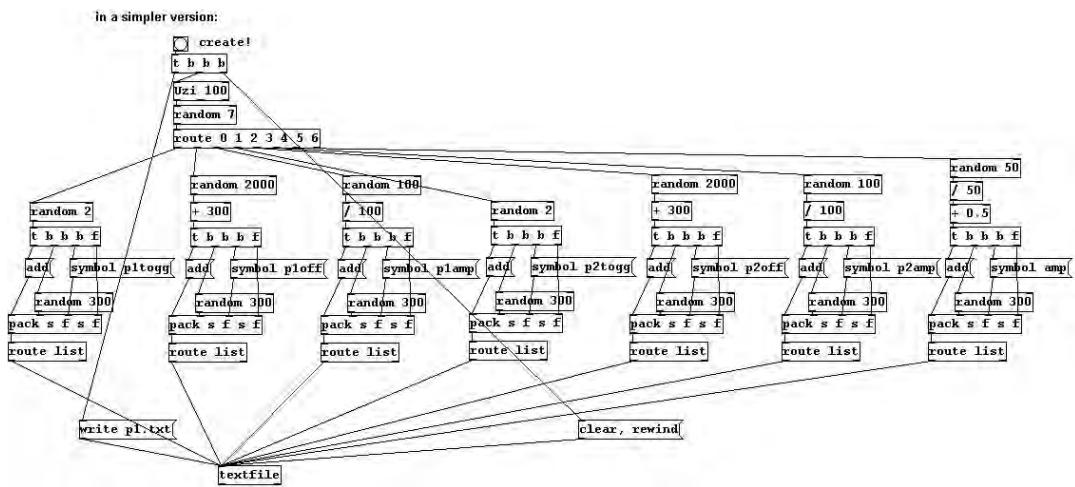
patches/a-24-waveshape-feedback.pd



### ► 4.2.2.2

Write stochastic algorithms into a text file that uses a “qlist” to play back the patch from 4.2.2.1 at different speeds:

patches/a-25-textcreate.pd

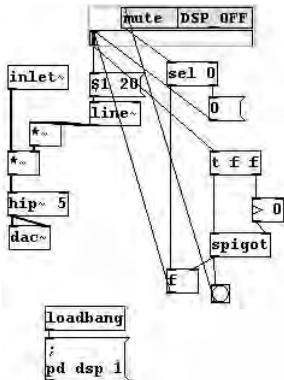


### ► 5.1.2.2

As abstractions:

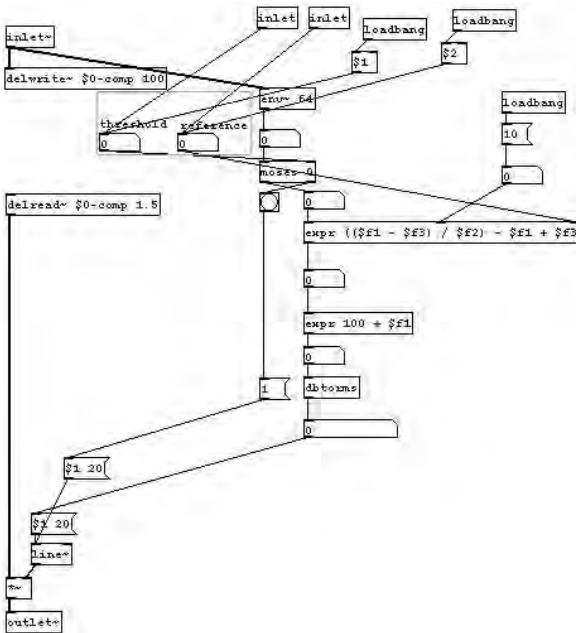
a) Integrate the DSP switch into the “dac” abstraction as well as a mute on/off button:

patches/a-26-dac-extended.pd



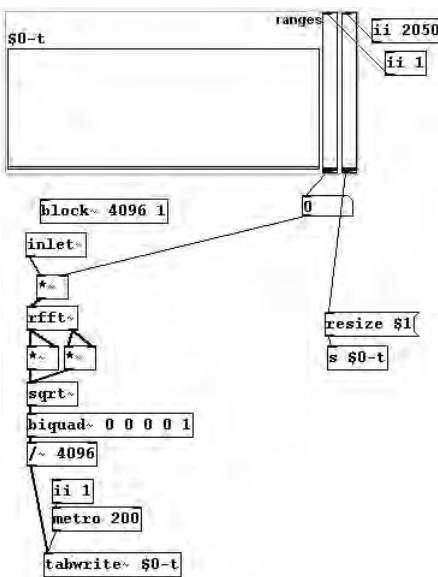
b) A compressor:

patches/a-27-compress~.pd



c) A graphic representation of an incoming spectrum:

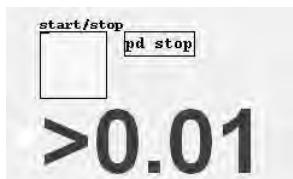
patches/a-28-spectrum.pd



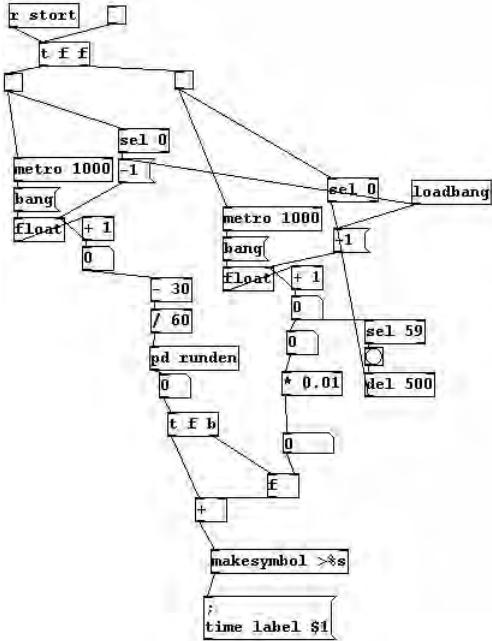
#### ► 5.2.2.4

a) A graphic representation of a stopwatch:

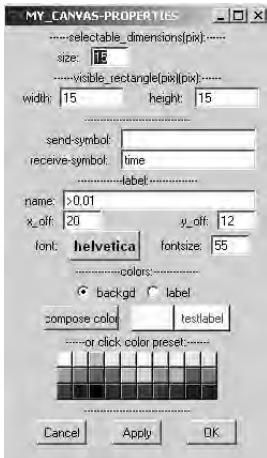
patches/a-29-stop.pd



...with the subpatch "stop":



...and the following canvas settings:



In the toggle, it is configured that it sends to “stort”.

b) A graphic for a running 5/8 meter, i.e., a visual click track. See patch file.

[patches/a-30-opt-track.pd](#)





## Index

Abstractions 226ff, 233ff, 268ff  
Adc~ 74, 82f, 94, 143, 218  
Additive Synthesis 91f, 99ff, 142f, 150  
Algorithm 35, 200ff, 206  
Amplitude corrections 192ff  
Append 241ff  
Argument 25f  
Array 112ff, 233ff  
Atoms 28

Bang 31, 36, 39, 40, 64, 65f  
Beating 163f  
Bp~ 107f  
Biquad~ 110f  
Bit depth 97  
Blocks 82, 135, 139, 179ff  
Block~ 181f, 183

Canon 56  
Canvas 26, 67, 127, 236ff  
Catch~ 96f  
Chord 93, 254  
Click 90, 108  
Clip~ 146f  
Clipping 89f  
Comb filter 133f, 260  
Comments 28, 63  
Compressor 186f, 194, 268  
Compute audio 21f, 82  
Controlling sound 200ff  
Convolution 185  
Cos~ 151  
Cosinesum 143  
Costumize Pd 231ff  
Counter 36, 38, 48, 250  
Crescendo 58, 87

Dac~ 22, 74, 82f, 231, 268  
Data structures 240ff  
Db 85  
DC Offset 109

Debussy, Claude 105  
Decrescendo 58, 87  
Del(ay) 51, 56, 110, 123f, 145, 187f  
Delread~ 123f  
Delwrite~ 123f  
Demu(ltiple)x 45f, 96  
Div(ision) 34  
Drawcurve 243  
Drawpolygon 243  
DSP 15, 82, 231, 268  
Duplication 28

Edit Mode 25f  
Env~ 87f, 192ff  
Execute Mode 25f  
Expander 264  
Expr(ession) 33, 249  
Externals 230f, 235

Feedback: see “Recursion”  
FFT: see “Fourier Analysis”  
Fiddle~ 188ff, 218, 263  
Filledcurve 243  
Filledpolygon 241ff  
Filters 105ff, 133f, 184, 255, 260  
F(loat) 35, 40, 44, 50  
Folding 185  
Foldover 160ff  
Font 68  
Fourier Analysis 100f, 179ff  
Fourier, Jean Baptiste Joseph 100f  
Frequency 76ff, 83f  
Frequency modulation 165ff, 167ff

GEM 231, 247  
Get 244f  
Glissando 54, 58, 93, 139ff, 253f  
Granular Synthesis 170ff, 262  
Graph on parent 228ff  
GUI 40, 49f, 51, 64ff, 88, 215, 229, 239f

Harmonic series 99ff  
Help file 27  
Hid 220  
HIDs 215ff  
Hip~ 106

Init 64, 69f  
Inlet 21, 30f, 53, 69ff, 73, 96  
Installing 18  
Instrument design 220  
Int 40

Joystick 220

Karplus-Strong algorithm 136ff, 259  
Key 215  
Keyname 215  
Keyup 215  
Klangfarbe 102f

Larsen tones 194  
Latency 143f  
Limitations of Pd 104  
Limiter 192f, 194  
Line 52f, 54f, 58, 60, 87, 97f  
Lisp 214  
L(ister) 44  
List of all Objects 27  
Lists 41f, 47, 60  
Loadbang 64  
Loop generator 130ff  
Lop~ 105f  
Lucier, Alvin 204

Makefilename 43f  
Mathematical operations 30f, 33ff  
Message Box 24ff, 28f, 31, 40f, 42f, 63  
Metro(nome) 51, 55, 58f, 71f, 251f, 270  
MIDI 216ff  
MIDI numbers 73, 78, 83  
Mod(olo) 34  
Modularization 71, 225f  
Moses 33f, 201f  
MouseState 215f  
Netreceive 221  
Netsend 221  
Network 221f  
Noise 76, 84f, 105ff, 108f  
Normalization 117f  
Number Box 22, 24ff, 39, 65  
Nyquist Theorem 81, 161

Object Box 24ff  
Octave doubler 134f, 191  
Offset 55, 80f  
Openpanel 44  
Open Source 13, 15  
Order 31ff, 40  
OSC 221f, 231  
Osc~ 22, 75, 79, 90f  
Outlet 21, 70f, 73, 96  
Overdrive 89f  
Overtone 100ff  
  
Qlist 209ff, 267  
  
Pack 41f  
Partial 100ff, 159f, 179  
Phase 90f, 92, 111, 168f  
Phase modulation 168f  
Phasor~ 77, 79, 122, 146f, 171f  
Physical modelling 136  
Pink~ 109  
Pipe 51f, 56  
Pitch 73, 76, 191  
Plot 245f  
Pointer 241ff  
Print 32, 36, 179f  
Puckette, Miller 13, 15, 248  
Pulse 75, 148, 150  
Pvu~ 89  
Pythagoras 99  
  
Radio 50, 67  
Ramp 90  
Random 35, 37, 38, 54, 55, 80f, 105, 152ff, 178, 201ff, 249, 251, 254  
Readsf~ 44, 120  
R(eceive) 61ff, 69, 73, 96, 207, 212  
Recursion (Feedback) 49, 111, 177, 204f, 265f  
Remote 63, 212f  
Relational Tests 34  
Residual Tone 102  
Reverb 132  
Rfft~ 179ff  
Rifft~ 179ff  
Ring modulation 163ff, 167, 204f  
Rms 85  
Rounding 37

Route 44, 47f, 250  
Samphold~ 171f  
Sampler 125ff, 256f  
Sample rate 79, 81, 98, 161  
Sampling 112ff  
Sawtooth 75, 77, 146, 149  
Sel(ect) 34, 48, 49  
S(end) 61ff, 69, 73, 96, 207  
Sequencer 207ff  
Set 242  
Setsize 246  
Short Cuts 28  
Sig~ 80, 87  
Signals 25, 80  
Sine 75, 100f, 148  
Sinesum 142f, 150  
Slider 49f, 66f  
Snapshot~ 79, 180  
Sound intensity 97  
Soundfiler 118f  
Sound pressure 97  
Space 195f  
Spectral delay 187f  
Speed of sound 95  
Spigot 46  
Square Wave 75, 147f, 149  
Stochastics 201ff, 267  
Stockhausen, Karlheinz 104  
Streamlining 223ff  
Struct 241ff  
Subpatch 69, 200, 223ff, 230  
Subtractive Analysis 105ff  
Symbol 26, 29, 42, 44, 49, 65  
  
Table 119  
Tabosc4~ 138  
Tabplay~ 138f  
Tabread 120ff, 144f  
Tabreceive~ 139, 182  
Tabsend~ 139  
Tabwrite 113, 117  
Textfile 207f  
Texture 133, 258f  
Throw~ 96f  
Tidy up 68

Time operations 51ff, 200ff  
Time resolution for control data 60  
Timer 53f  
Toggle 46, 47, 66  
Transfer functions 151f, 159  
Triangle Wave 75, 147, 149  
T(trigger) 32, 36, 39, 53  
Tuner 190

Unpack 42  
Unsig~ 80, 180  
Uzi~ 152f

Value 64  
Variables 33, 42f, 226ff  
Vd~ 124, 167f, 175ff  
Volume 85ff, 192ff, 264  
Vu meter 88f

Waves 75ff, 83, 95m 146ff, 232, 261  
Wave shaping 138, 146ff, 261, 266  
Wave stealing 157  
Windowing 130f, 154f, 172f, 182f, 197ff  
Wintablet 220  
Wishart, Trevor 104  
Wrap~ 159f  
Writesf~ 40f, 112  
Xenakis, Iannis 162  
Z~ 110, 136f





Pd-Graz (Hg.)  
**bang.**

**Pure Data (1. International PD-Convention  
Graz)**

PD (aka Pure Data) is a real-time graphical programming environment for audio, video, and graphical processing. It is one of the most lively free-software-developments in the last years that is mostly implemented in artistic contexts.

The core of Pd is written and maintained by Miller Puckette and includes the work of many developers, making the whole package very much a community effort.

In September 2004 artists from Graz initiated the 1st International PD-Convention to bring most of the central figures around PD together on one physical place to discuss current issues. These discussions are the starting points for the texts published in this compilation.

The authors are PD-developers, media/art theoreticians and artists.

Texts by: Miller Puckette, Marc Ries, Christian Scheib, Brian Jurish, Susanne Schmidt, Reinhard Braun, Andrea Mayr, Frank Barknecht, Günther Geiger, Franz Xaver, Thomas Grill, Hans-Christoph Steiner, Iohannes Zmölnig, James Tittle, Cyrille Henry, Alexandre Castonguay, and Winfried Ritsch, Michael Pinter, Reni Hofmüller, and others.

The book contains the first DVD issue realized by the label pd~.

edited by: pd-graz (Verein zur Förderung von Kunstproduktionen mit Pure Data)

176 p., DVD, pb., € 24.-  
ISBN 978-3-936000-37-5