

Computational Music Synthesis

A Set of Undergraduate Lecture Notes by

Sean Luke

Department of Computer Science
George Mason University

Zeroth Edition

Online Version 0.2
June, 2019



Copyright 2019 by Sean Luke.

Thanks to Carlotta Domeniconi, Bryan Hoyle, Lilas Dinh, Robbie Gillam, John Otten, Mark Snyder, Adonay Henriquez Iraheta, Alison Howland, Daniel McCloskey, Stephen Yee, Jabari Byrd, Riley Keesling, Benjamin Ngyen, Joseph Parrotta, Laura Pilkington, Pablo Turriago-Lopez, Eric Velosky, Joshua Weipert, Joshua Westhoven, Matt Collins, Kelian Li, Thomas Tyra, Talha Mirza, Curtis Roads, Julius Smith, Brad Ferguson, Zoran Duric, and Dan Lofaro.

Get the latest version of this document or suggest improvements here:

<http://cs.gmu.edu/~sean/book/synthesis/>

Cite this document as:

Sean Luke, 2019, *Computational Music Synthesis*, zeroth edition,
available for free at <http://cs.gmu.edu/~sean/book/synthesis/>

Always include the URL, as this book is primarily found online. Do *not* include the online version numbers unless you must, as Citeseer and Google Scholar may treat each (oft-changing) version as a different book.

BibTeX: @Book{ Luke2019Synthesis,

```
author = { Sean Luke },
title = { Computational Music Synthesis },
edition = { zeroth },
year = { 2019 },
note = { Available for free at http://cs.gmu.edu/\$\simsean/book/synthesis/ }
```

This document is licensed under the **Creative Commons Attribution-No Derivative Works 3.0 United States License**, except for those portions of the work licensed differently as described in the next section. To view a copy of this license, visit <http://creativecommons.org/licenses/by-nd/3.0/us/> or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA. A summary:

- You are free to redistribute this document.
- **You may not** modify, transform, translate, or build upon the document except for personal use.
- You must maintain the author's attribution with the document at all times.
- You may not use the attribution to imply that the author endorses you or your document use.

This summary is just informational: if there is a conflict between the summary and the actual license, the actual license always takes precedence. Contact me if you need to violate the license (like do a translation).

Certain art is not mine. Those images I did not create are marked with a special endnote like this: ^{©14} These refer to entries in the Figure Copyright Acknowledgements, starting on page 155, where I list the creator, original URL, and the license or public domain declaration under which the image may be used. In some cases there is no license: instead, the creators have kindly granted me permission to use the image in the book. In these cases you will need to contact the creators directly to obtain permission if you wish to reuse the image outside of this book.

Contents

List of Algorithms	4
0 Preface	5
0.1 Caveats	5
0.2 Algorithms	5
1 Introduction	7
1.1 A Very Brief History	7
1.2 The Synthesizer Player's Environment	9
1.3 A Typical Synthesizer	11
2 Representation of Sound	13
2.1 Units of Measure	16
2.2 Digitization of Sound Waves	17
3 The Fourier Transform	21
3.1 The Discrete Fourier Transform	22
3.2 Computing Amplitude and Phase	23
3.3 Real-Valued Fourier Transforms	24
3.4 The Fast Fourier Transform	25
3.5 Windows	28
3.6 Applications	29
4 Additive Synthesis	31
4.1 History	31
4.2 Approach	33
4.3 Implementation	35
4.4 Architecture Examples	39
5 Modulation	43
5.1 Low Frequency Oscillators	43
5.2 Envelopes	46
5.3 Step Sequencers and Drum Machines	50
5.4 Arpeggiators	52
5.5 Gate/CV and Modular Synthesizers	52
5.6 Modulation Matrices	53
5.7 Modulation via MIDI	54
6 Subtractive Synthesis	55
6.1 History	55
6.2 Implementation	62
6.3 Architecture Examples	63

7	Oscillators, Combiners, and Amplifiers	67
7.1	Oscillators	67
7.2	Antialiasing and the Nyquist Limit	70
7.3	Wave Shaping	73
7.4	Wave Folding	74
7.5	Phase Distortion	75
7.6	Combining	76
7.7	Amplification	79
8	Filters	81
8.1	Digital Filters	83
8.2	Building a Digital Filter	86
8.3	Transfer Functions in the Laplace Domain	87
8.4	Poles and Zeros in the Laplace Domain	88
8.5	Amplitude and Phase Response	88
8.6	Pole and Zero Placement in the Laplace Domain	89
8.7	The Z Domain and the Bilinear Transform	92
8.8	Pole and Zero Placement in the Z Domain	94
8.9	Basic Second-Order Butterworth Filters	96
8.10	Digital Second-Order Butterworth Filters	98
8.11	Formant Filters	101
9	Frequency Modulation Synthesis	103
9.1	Frequency and Phase Modulation	103
9.2	Sidebands, Bessel Functions, and Reflection	105
9.3	Operators and Algorithms	108
9.4	Implementation	110
10	Sampling	113
10.1	History	113
10.2	Pulse Code Modulation	114
10.3	Wavetable Synthesis	115
10.4	Granular Synthesis	116
10.5	Resampling	117
10.6	Basic Real-Time Interpolation	119
10.7	Windowed Sinc Interpolation	120
11	Effects and Physical Modeling	125
11.1	Delays	125
11.2	Flangers	126
11.3	Chorus	127
11.4	Reverb	129
11.5	Phasers	132
11.6	Physical Modeling Synthesis	132

12 Controllers and MIDI	135
12.1 History	135
12.2 MIDI	139
12.2.1 Routing	140
12.2.2 Messages	141
12.2.3 CC, RPN, and NRPN	144
12.2.4 Challenges	146
12.2.5 MPE	147
12.2.6 MIDI 2.0	149
Sources	151
Figure Copyright Acknowledgments	155
Index	159

List of Algorithms

0	Bubble Sort	6
1	The Discrete Fourier Transform	23
2	RecursiveFFT (Private Subfunction)	27
3	The Fast Fourier Transform	27
4	The Inverse Fast Fourier Transform	28
5	Multiply by a Window Function	29
6	Simple Monophonic Additive Synthesizer Architecture	36
7	Sine Table Initialization	38
8	Sine Approximation	38
9	Buffer Output	39
10	Simple Low Frequency Oscillator	44
11	Random Low Frequency Oscillator	45
12	Sample and Hold	46
13	Simple Linear Time-based ADSR	48
14	Simple Exponential Rate-based ADSR	49
15	Simple Parameter Step Sequencer	51
16	Simple Monophonic Subtractive Synthesizer Architecture	62
17	Pink Noise	69
18	Initialize a Digital Filter	85
19	Step a Digital Filter	86
20	Delay Line	125
21	Multi-Tap Delay Line	128
22	Karplus-Strong String Synthesis	132

0 Preface

This book was developed for a senior computer science course I taught in Spring of 2019. Its objective was to teach a computer science student with some music experience how to build a **digital music synthesizer in software** from the ground up. I hope it'll be useful to others.

The text assumes an undergraduate computer science background and some basic calculus and linear algebra. But it does not assume that the reader is particularly familiar with the history, use, or significance of music synthesizers. To provide some appreciation of these concepts, I've tried to include quite a bit of history and grounding in the text. The text also doesn't assume that the reader knows much about electrical engineering or digital signal processing (indeed it should be obvious to experts in these fields that I don't know much either!) and so tries to provide an introduction to these concepts in a relatively gentle manner.

One of the problems with writing a book on music topics is that reading about these topics is not enough: you have to *hear* the sounds being discussed, and *see* the instruments being manipulated, in order to gain an intuitive understanding for the concepts being presented. The mere pages here won't help with that.

0.1 Caveats

While I am a computer science professor, a musician, and a synthesizer tool builder on the side, I am by no means an expert in how to build music synthesizers. I am very familiar with certain subjects discussed here, but many others were entirely new to me at the start of developing this book and course. My knowledge of filters, resampling, and effects is particularly weak.

What this means is that you should take a lot of what's discussed here with a big grain of salt: there are likely to be a great many errors in the text, ranging from small typos to grand misconceptions. I would very much appreciate error reports: send them to sean@cs.gmu.edu. I may also be making significant modifications to the text over time, even rearranging entire sections as necessary. I have also tried very hard to cite my sources and give credit where it is due. If you feel I did not adequately cite or credit you, send me mail.

I refer to my own tools here and there. Hey, that's my prerogative! They are all open source:

- **Gizmo** is an Arduino-based MIDI manipulation tool with a step sequencer, arpeggiator, note recorder, and lots of other stuff. I refer to it in Section 5.
<http://cs.gmu.edu/~sean/projects/gizmo/>
- **Edisyn** is a synthesizer patch editor with very sophisticated tools designed to assist in exploring the space of patches. I refer to it in Section 5.
<http://cs.gmu.edu/~sean/projects/edisyn/>
- **Flow** is an unusual polyphonic additive modular software synthesizer. I refer to it in Section 4.
<http://cs.gmu.edu/~sean/projects/flow/>

0.2 Algorithms

Algorithms in this book are written peculiarly and relatively informally. If an algorithm takes parameters, they will appear first followed by a blank line. If there are no parameters, the algorithm

begins immediately. Sometimes certain shared, static global variables are defined which appear at the beginning and are labelled **global**. Here is an example of a simple algorithm:

Algorithm 0 *Bubble Sort*

```
1:  $\vec{v} \leftarrow \langle v_1, \dots, v_l \rangle$  vector to sort           ▷ User-provided parameters to the algorithm appear here  
2: repeat                                         ▷ Then a blank space  
3:   swapped  $\leftarrow$  false                         ▷ Algorithm begins here  
4:   for  $i$  from 1 to  $l - 1$  do          ▷  $\leftarrow$  always means "is set to"  
5:     if  $v_i > v_{i+1}$  then          ▷ Note that  $l$  is defined by  $v_l$  in Line 1  
6:       Swap  $v_i$  and  $v_{i+1}$   
7:     swapped  $\leftarrow$  true  
8:   until swapped = false          ▷ = means "is equal to"  
9: return  $\vec{v}$                          ▷ Some algorithms may return nothing, so there is no return statement
```

1 Introduction

A music synthesizer, or **synthesizer** (or just **synth**), is a programmable device which produces sounds in response to being played or controlled by a musician or composer. Synthesizers are omnipresent. They're in pop and rock songs, rap and hip hop, movie and television scores, sound cards and video games, and—unfortunately—cell phone ringtones. Music synthesizers are used for other purposes as well: for example, **R2-D2**'s sounds were generated on a music synthesizer,¹ as was **Deep Note**, the famous trademark sound played before movies to indicate the use of **THX**.² The classic "Ahhh" bootup sound on many Macintoshes in the 90s and 00s was produced on a **Korg Wavestation**, a popular commercial synthesizer from the late 90s.

Traditionally a music synthesizer generates sounds from scratch by creating and modifying waveforms. But that's not the only scenario. For example **samplers** will *sample* a sound, then edit it and store it to be played back later as individual notes. **Romplers**³ are similar, except that their playback samples are fixed in ROM, and so they cannot sample in the first place.

Synthesizers also differ based on their use. For example, while many synthesizers produce tonal notes for melody, **drum machines** produce synthesized or sampled drum sounds. **Vocoders** take in human speech via a microphone, then use this sound source to produce a synthesized version of the same, often creating a robot voice sound. **Effects** units take in sounds—vocals or instrumentals, say—then modify them and emit the result, adding delay or reverb, for example.

Synthesizers have often been criticized for notionally *replicating*, and ultimately *replacing*, real instruments. And indeed this is not an uncommon use case: a great many movie scores you probably thought were performed by orchestras were actually done with synthesizers, and much more cheaply. Certainly there are many stories in history of drum machines eliminating drummers from bands. But more and more synthesizers have come to be seen as instruments in their own right, with their own aesthetic and art.

1.1 A Very Brief History

We'll cover synthesizer history more in-depth in later sections: here we'll start with a very *very* brief history of the basics.

1960s Synthesizers have been around since the late 1800s in various guises, but with a few famous exceptions, they did not seriously impact the music scene until the 1960s with the rise of **modular synthesizers** manufactured by the likes of Robert Moog and Don Buchla. These devices consisted of a variety of modules which generated sounds, modified sounds, or emitted signals meant to change the parameters of other modules in real time. The units were connected via quarter-inch **patch cables** (the same as used by electric guitars), and so even today the term for a synthesizer program which defines a played sound is a **patch**.



Figure 0 Moog modular synthesizer.^{©1}

¹An ARP 2600. See Figure 46 in Section 6.

²Deep Note was computer-generated by a software synthesizer written in custom C code.

³This is a derogatory by common term. It's a mash-up of **ROM** and *sampler*.

Modular synthesizers had many failings. They were large and cumbersome, required manual connections with cabling, could only store one patch at a time (the one currently wired up!), and usually could only produce one note at a time (that is, they were **monophonic**). Modular synthesizer keyboards of the time offered limited control and expressivity. And modular synths were very, very expensive.

Modular synthesizers were also **analog**, meaning that their sounds were produced entirely via analog circuitry. Analog synthesizers would continue to dominate until the mid-1980s.

1970s This decade saw the introduction of compact, portable, all-in-one analog synthesizers which eliminated the wires of their predecessors, and thus could be used realistically by touring musicians. One prominent model was the **Moog Minimoog Model D**, shown in Figure 1. Other models allowed multiple notes to be played at the same time (they were **polyphonic**). The 1970s also saw the introduction of the first viable **drum machines**, synthesizers which produced only drum sounds following rhythm patterns rather than notes.



Figure 1 Moog Minimoog Model D.^{©2}



Figure 2 Roland CR78 drum machine.^{©3}

1980s and 1990s This period saw an explosion in synthesizer technology. Due to the introduction of **MIDI**, a simple communication standard, musicians could play synthesizers from remote keyboards, from computers, or from **sequencers** which stored note event data much like a computerized music box or player piano roll. Synthesizer hardware began to be separated from its means of musical control: one could purchase **controllers** (commonly keyboards) whose sole function was to manipulate synthesizers via MIDI, as well as **rackmount** or **desktop (tabletop) synthesizers** with no keyboard at all.

Synthesizers also benefitted from RAM and CPUs, enabling them to store and recall patches at the touch of a button. And critically, while nearly all previous synthesizers produced sound from analog electronic components, the CPU gave rise to **digital synthesizers** which produced discrete waveforms and enabled many new approaches to synthesis. The digital tsunami began with **frequency modulation** (or **FM**) synthesizers, spearheaded by the highly influential **Yamaha DX7**. Digital synthesizer approaches which followed included **wavetable synthesis**, **samplers** and **romplers**, digital **additive synthesis**, and **vector synthesis**. The onslaught of FM alone almost singlehandedly did away with the analog synthesizer industry. Digital synthesizers also migrated from music halls to more mundane uses: video games, sound cards on personal computers, and eventually (sigh) ringtones.



Figure 3 Yamaha DX7.^{©4}

2000s and Beyond As personal computers became increasingly powerful, the turn of the century saw the rise of **digital audio workstations** or **DAWs**: software which could handle much of the music production environment entirely inside a laptop. This included the use of **software synthesizers** rather than those in hardware. The early 2000s also saw the popularization of **virtual analog synthesizers**, which simulated classic analog approaches in digital form.

Analog synthesizers have since seen a renaissance as musicians yearned for the warm, physical devices of the past. At the extreme end of this trend, we have since seen the reintroduction of modular synthesizers as a popular format. What goes around comes around.



Figure 4 Propellerhead Reason DAW.^{©5}

1.2 The Synthesizer Player’s Environment

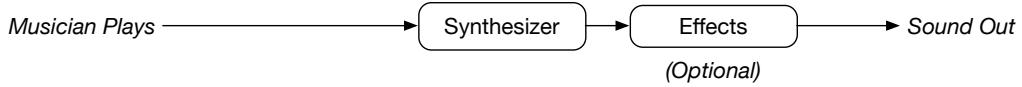
Music synthesizers are ubiquitous in the music and sound effects scene, and so appear in a wide range of of music performance and production scenarios. If you’re unfamiliar with the architecture of these scenarios, it’s useful to review a few common ones. The scenarios in question are shown in Figure 5.

Playing Around This is the obvious basic scenario: you own a synthesizer and want to play on it. This is sometimes called **noodling**. The important item here is the possible inclusion of an **effects unit**. Effects are manipulations of sound to add some kind of “texture” or “color” to it. For example, we might add a **delay** or **echo** to the synthesizer’s sound, or some **reverb** or **chorus**. Effects, particularly reverb, are often important to make a synthesizer’s dry sound become more realistic or interesting sounding. Because effects are so important an item at the end of the synthesizer’s audio chain, many modern synthesizers have effects built in as part of the synthesizer itself.

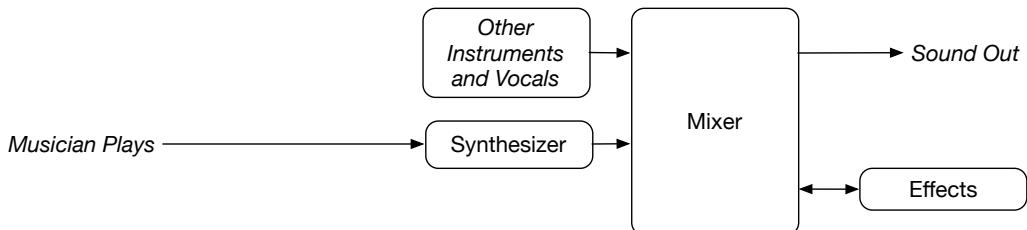
Performance In the next scenario, you are playing a synthesizer as a solo or group live performance involving sound reinforcement. To do this, you will need the inclusion of a **mixer**, a device which sums up the sounds from multiple inputs and produces one final sound to be broadcast. Mixers can be anything from small tabletop/desktop or rackmount devices to massive automated **mixing consoles**: but they all do basically the same thing. Here, the effects unit is added as an auxiliary module: the mixer will **send** a mixed sound to the effects unit, which then **returns** the resulting effects. The mixer then adds the effects to the final sound and outputs it. The amount of effects added to the final sound is known as how **wet** the effects are.

Production A classic synthesizer sound recording and production environment adds a **recorder** (historically a **multi-track tape recorder**) which receives sound from the mixer or sends it back to the mixer to be assessed and revised. The high-grade speakers used by recording engineers or musicians to assess how the music sounds during the mixing and recording process are known as **monitors**. Additionally, because synthesizers can be controlled remotely and also controlled

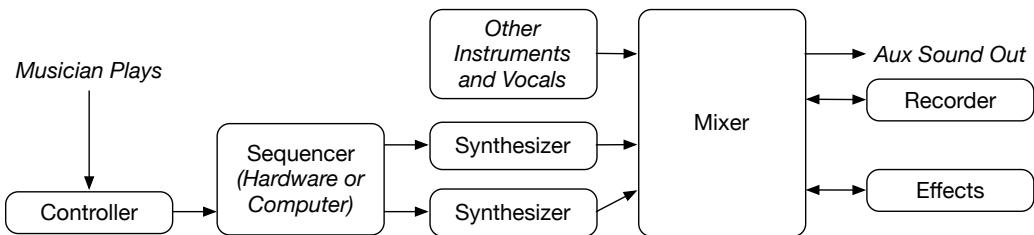
Playing Around



Performance



Production



In-The-Box Production

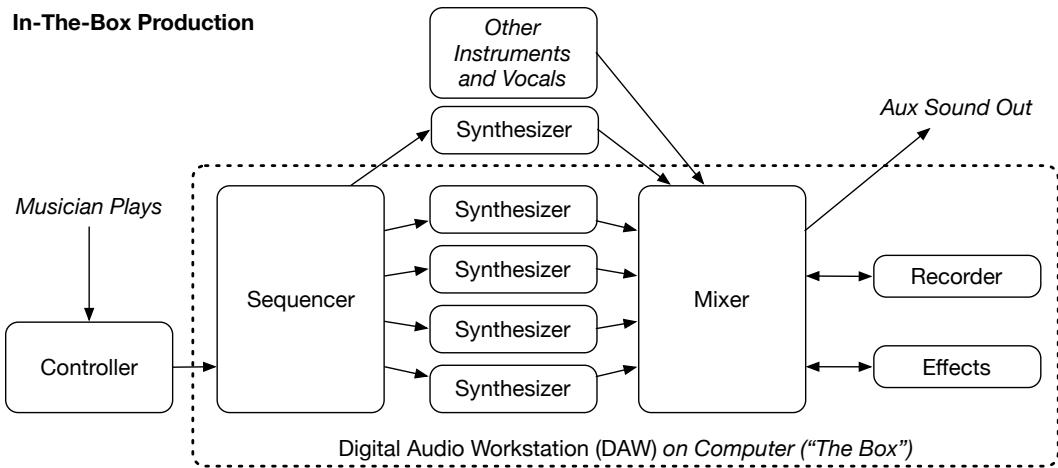


Figure 5 Four common scenarios in which music synthesizers are used.

via automated means, a musician might construct an entire song by playing multiple parts into a **sequencer**, which records the event data (when a note was played or released, etc.) and then can play multiple synthesizers simultaneously. Sequencers can be found both as computer software and as dedicated hardware. In this scenario, a musician wouldn't play a synthesizer directly, but rather would play a **controller**, often a keyboard, to issue event data to the sequencer or to one or more downstream synthesizers.

In-the-Box (ITB) Production The classic synthesizer production environment has given way to one in which most of these tasks are now done in software on a computer. This is known as **In The Box** or **ITB** production. The core software environment for ITB production is the **digital audio workstation** or **DAW**. This software is a combination of a sequencer, mixer, recorder, and effects unit. Most DAWs are also modular in design and can be extended via plug-ins, the most well-known being **Virtual Studio Technology** plugins (or **VSTs**), or **Audio Unit** (**AU**) plugins. These plugins can be used for many tasks, but are often used to add **software synthesizers** (or **softsynths**) as additional playable instruments directly in the DAW.

A DAW interacts with the musician in several ways. First, the musician can enter note event data directly via his controller to the computer through a **MIDI interface**. Second, the DAW's internal sequencer can also control external synthesizers via the same interface, playing them along with its software synthesizers. Third, the musician can record audio into the DAW from those synthesizers or from other audio sources (instruments, vocals) via an **audio interface**. Fourth, the computer can use this same audio interface to play audio on monitors or headphones to be assessed by the musician or studio engineer. Ultimately the DAW will be used to write out a final version of the song in digital form for a CD or MP3 file, etc.

1.3 A Typical Synthesizer

A musician interacts with a typical synthesizer in four basic ways:

- By playing it (of course).
- By changing one or two parameters in real time while playing as part of the performance (turning a knob, say).
- By editing all of its parameters offline to change how it sounds.
- By automating his performance, both playing and real-time parameter-modification.

Consider the **Dave Smith Instruments Prophet '08**, which came out in (what else?) 2008. This is a classic **analog subtractive** synthesizer, and so it is festooned with knobs and buttons (see Figure 6) to enable easy programming both in real time during performance and also offline.

The Prophet '08 is **polyphonic**, meaning that it can play multiple notes at a time: in this case, at most eight. The sound circuitry necessary to play a single note is called a **voice**; and thus the Prophet '08 has eight voices. Figure 7 shows the Prophet '08's voice card.



Figure 6 Dave Smith Instruments Prophet '08.^{©6}

Voices The architecture for a single Prophet '08 voice is very typical of a subtractive analog synthesizer. Each of its eight voices has two **oscillators**, which are modules that produce sound waves. These oscillators are then **combined** together to form a single sound wave, which is then fed into a **filter**. A filter is a device which modifies the tonal qualities of a sound: in this case, the Prophet '08 has a **low pass filter**, which can tamp down high frequencies in a sound wave, making it sound duller or more mellow. The filtered sound is then fed into an **amplifier** which changes its volume. All of the currently sounding voices are then finally added together and the result is emitted as sound.

The oscillators, combiner, filter, and amplifier all have many parameters. For example, the oscillators may be **detuned** relative to one another; the combiner can be set to weight one oscillator's volume more than another; the low-pass filter's **cutoff frequency** (the point beyond which it starts dampening sounds) may be adjusted, or the amplifier's volume scaling may be tweaked.

Modulation The synthesizer's many parameters can be manually set by the musician, or the musician can attach them to **modulation** devices which will change the parameters automatically over time as a note is played, to create richer sounds. For example, both the filter and the amplifier have dedicated **DADSR envelopes** to change their cutoff frequency and volume scaling respectively. A DADSR (Delay/Attack/Decay/Sustain/Release) envelope is a simple function which starts at 0 when a note is played, then *delays* for a certain amount of time, then rises (*attacks*) to some maximum value at a certain rate, then falls (*decays*) to a different (*sustain*) value at a certain rate. It then holds at that sustain value as long as the key is held down, and when the key is *released*, it dies back down to zero at a certain rate. This allows (for example) a sound to get suddenly loud or brash initially, then die back and finally decay slowly after it has been released.

In addition to its two dedicated envelopes, the Prophet '08 has an extra envelope which can be assigned to many different parameters; and it also has four **low frequency oscillators** or LFOs which can also be so assigned. An LFO is just a function which slowly oscillates between 0 and 1 (or between -1 and 1). When attached to the pitch of a voice an LFO would cause **vibrato**, for example. The Prophet '08 also has a basic **step sequencer** which can play notes or change parameters in a certain repeating, programmable pattern; and a simple **arpeggiator** which repeatedly plays notes held down by the musician in a certain repeating pattern as well. Modulation sources can be assigned to many different parameters via the Prophet '08's **modulation matrix**.

Patches and MIDI The parameters which collectively define the sound the Prophet '08 is making are called a **patch**. The Prophet '08 is a **stored patch synthesizer**, meaning that after you have programmed the synthesizer to produce a sound you like, you can save patches to memory; and you can recall them later. A patch can also be transferred to or from a computer program, or another Prophet '08, over a **MIDI** cable. MIDI can also be used to play or program a Prophet '08 remotely from another keyboard controller, computer, or synthesizer. Because you can play a Prophet '08 remotely via another keyboard, you don't need the Prophet '08's keyboard at all, and indeed there exists a keyboard-less standalone tabletop or **desktop module** version of the synthesizer.

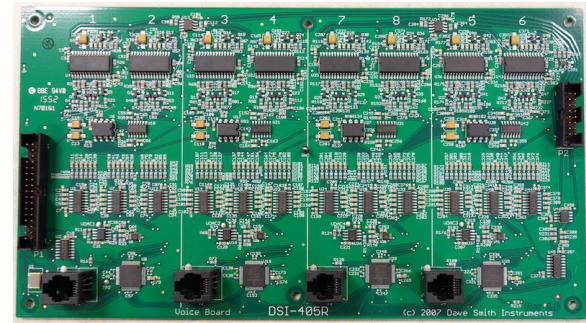


Figure 7 Eight-voice circuitry for the Prophet '08.

2 Representation of Sound

Sound waves represent changes in air or water pressure as a sound arrives to our ear and, in their simplest form, they are simple one-dimensional functions of time, that is $f(\text{time})$. Figure 8 at right shows a snippet of a sound wave. The x -axis is time and the y -axis is the wave's current **amplitude** at that time. Sound waves may come in pairs, perhaps resulting in **stereo** sound, or even larger numbers: for example **quadraphonic** sound consists of four waves.

But a function of time isn't the only way to view a sound wave. It's true that many music synthesizers and effects devices manipulate sound waves this way. And as humans we are accustomed to *think* of sound this way. But in fact this isn't a particularly good way to think of sound, because our brains don't receive a *sound wave* at all.

Instead, we can think of our brains as receiving, at any given time, an *array* of amplitudes for different **frequencies**. This array changes over time. If the incoming sound has loud high frequencies, for example, then the amplitudes corresponding to those frequency slots in the array will have large numbers. One way of viewing this is shown in Figure 9, in which the arrays are vertical slices out of the image (the x -axis is time). A graph of this type is known as a **spectrogram**.

How is this so? A critical fact about sound waves (and any time-variant wave) is that any sound wave can be described as the infinite sum of sine waves, each with its own frequency, amplitude, and phase. For example, consider Figure 10 at right. There are three dashed sine waves, each of a different frequency⁴ and a different amplitude. If you add up the waves, the result is the more complex wave shown in bold black. These sine waves are known as **partials**, since each of them is in some sense a *part* of the final sound wave.

If we disregarded phase, we could plot the three partials in Figure 10 by their amplitudes as a kind of bar chart, where the x -axis would be *frequency*, not time. This bar chart is shown in Figure 11. If we viewed the sound in this way, it's known as casting the sound in the **frequency domain** (because of the x -axis). If we viewed a sound in the classic way (Figure 10), where the x -axis is time, this is known as percieiving the sound in the **time domain**.

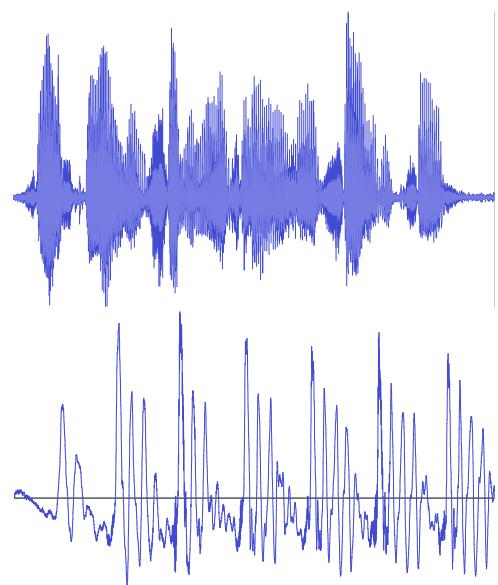


Figure 8 A sound wave and zoomed-in portion. The x -axis is time, the y -axis is amplitude.

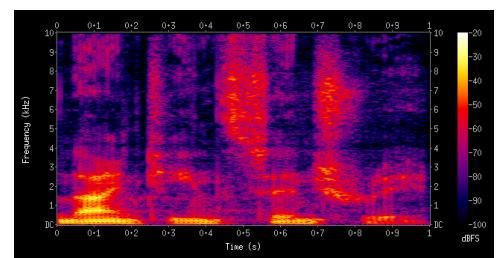


Figure 9 A spectrogram of the spoken phrase "nineteenth century". The x -axis is time, the y -axis is frequency, and the color is amplitude.^{©7}

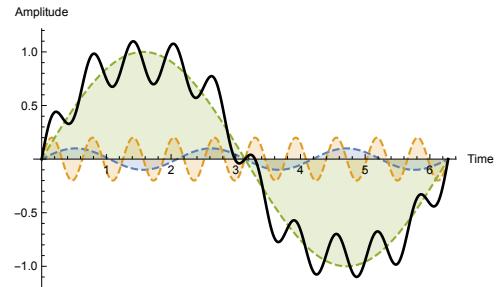


Figure 10 Three sine waves (colored) and the result of adding them up (in bold).

⁴In this example, they all have the same phase, since they're all 0 at time 0.

When a sound is arbitrarily long and complex, the number of sine waves required to describe it is effectively infinite and uncountable: so the frequency domain is no longer a bar chart, but is really a real-valued function of frequency. As an example, Figure 12 shows the time domain and the (real-valued) frequency domain of a note from a bass guitar.

There exists a straightforward mathematical transformation between the time domain and the frequency domain. It's called the **Fourier Transform** (and its inverse, not surprisingly named the **Inverse Fourier Transform**). The mathematics and algorithms to perform this transform are discussed in Section 3.

As mentioned before, we perceive things in the frequency domain: this is because of how the cochlea in our ear operates. The cochlea is essentially a long, curled up tube filled with liquid and lined with hair. Hairs near the start of the cochlea vibrate in sympathetic response to low-frequency sounds, and hairs further along the cochlea vibrate in response to higher frequency sounds. The louder the sound at a hair's frequency, the more the hair vibrates. The hairs are connected to nerves which send signals to the brain. Thus when a sound enters the cochlea, the hairs are essentially doing a kind of Fourier Transform, breaking the sound out into its separate frequency components, which are then passed to the brain.

Phase Phase is the point in time where the sine wave begins (starts or restarts from zero). Consider Figure 13, which shows three sine waves with identical frequency and amplitude, but which differ in phase. Along with amplitude and frequency, the phase of a partial plays a critical part in the sound. Thus the frequency domain should not be thought of as a single plot of frequency versus amplitude, but rather as *two separate plots*, one of frequency versus amplitude, and the other of frequency versus phase. Similarly, the partials that make up a sound have *two* components: amplitude and phase.

Phase is far less important to us than amplitude: humans can detect amplitude much better. In fact, while we can distinguish partials which are *changing* in phase, if we were presented with two sounds with identical partials except for different phases, we would *not be able to distinguish between them!* Because of this, some synthesis methods (such as **additive synthesis**) almost entirely disregard phase: though other ones (such as **frequency modulation synthesis**), which rely on *changes* in phase very heavily.

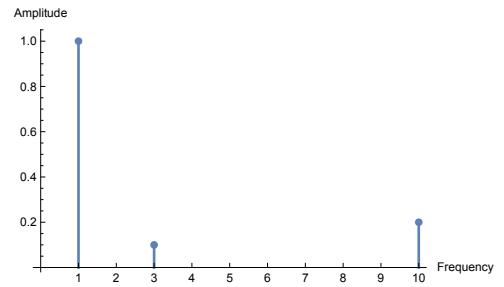


Figure 11 The three sine waves from Figure 10, plotted by amplitude and frequency.

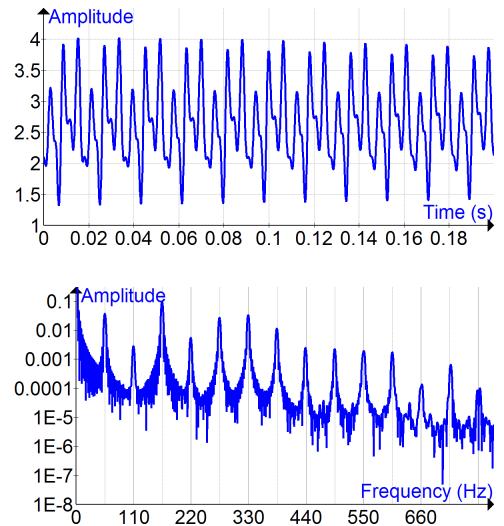


Figure 12 Time Domain (top) and Frequency Domain (bottom) of a bass guitar note.^{©8}

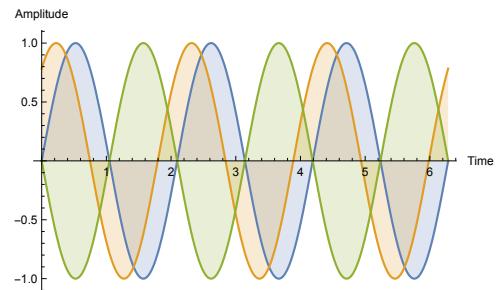


Figure 13 Three identical sine waves which differ only in phase. Note that the blue and green sine waves have entirely *opposite* phase.

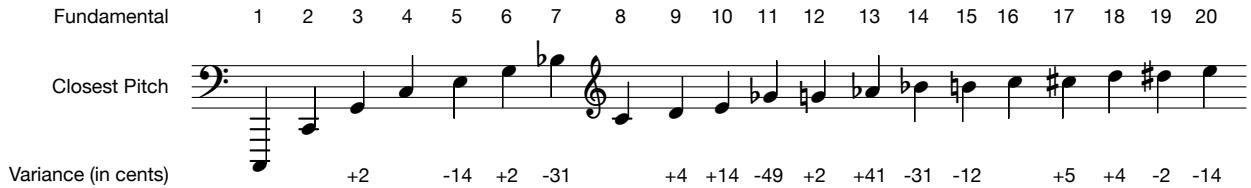


Figure 15 Harmonic series. The fundamental is shown as a low C, and successive harmonics are shown with their equivalent pitches. The numbers below the notes indicate the degree to which the harmonic deviates from traditional pitches (in cents).⁵⁹

Harmonics and Pitch Identification Most sounds which we perceive as “tonal” or “musical” have the large majority of their partials organized in very specific way. In these sounds, there is a specific partial called the **fundamental**. This is often the lowest significant partial, often the loudest partial, and often the partial whose frequency we typically would identify as the **pitch** of the sound—that is, its associated **note**. Partials other than the fundamental are called **overtones**. Furthermore, in these “tonal” sounds, most overtones have frequencies which are integer multiples of the fundamental. That is, most of the overtones will have frequencies of the form $i \times f$, where f is the fundamental frequency, and i is an integer 2, 3, When partials are organized this way, we call them **harmonics**.

A great many instruments have partials organized largely as harmonics. This includes woodwinds, brass, strings, you name it. The reason for this is that many instruments essentially fixed strings or tubes which can only vibrate according to certain **modes**. For example, consider Figure 14. Here the ends of a violin string are fixed at 0 and 2π respectively. There are only so many ways that a violin string can vibrate as long as those ends are fixed. Figure 14 shows the first four possibilities, and their frequencies correspond to the first four harmonics (f , $2f$, $3f$, and $4f$). A woodwind is similar: vibrations in the air in its tube are essentially “fixed” at the two ends.

Many harmonics are very close to the pitch of standard notes, and this has a strong effect on our perception of the tonality of instruments and the chords they produce. For example, the second harmonic, whose frequency is $2f$, is exactly an octave above f . The third harmonic is very nearly a fifth above that. The fourth harmonic is two octaves above the fundamental. Figure 15 shows a fundamental, various harmonics, the notes that they are closest to, and their degree of deviation from those notes. A few harmonics are quite off,⁵ but many are tantalizingly close.

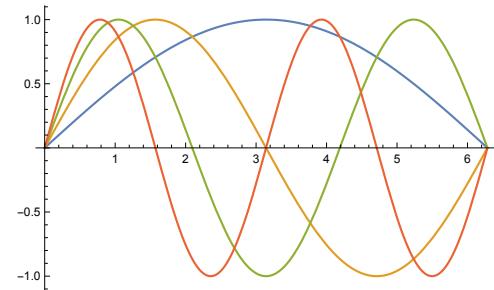


Figure 14 Modes corresponding to the **first**, **second**, **third**, and **fourth** harmonics in a stringed instrument.

⁵You might be wondering: *why* are they off? This is an interesting question. Classically notes have been tuned such that an octave corresponds to a doubling in frequency. That lines up nicely with harmonics since every harmonic that is the next power of 2 is an octave higher. But within an octave, how would one space the remaining notes? The classic approach has been to assume that there are 12 notes, and that one spaces them such that the ratio between the frequencies any two successive notes (A / A \flat , say, or F/E) is exactly the same. This is a fancy way of saying that the notes are laid out not linearly in frequency but logarithmically. This tuning strategy is known as *Equal Temperament*. The problem with this model is that these logarithmic frequencies don’t line up along integer multiple values like the harmonics do. Many of them are *close enough*, but some are pretty off. Because integers and logs don’t match up, temperament strategies to make notes sound more harmonious together have been a matter of debate for many centuries.

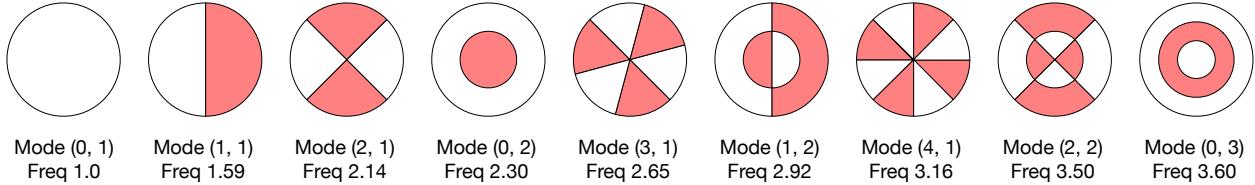


Figure 16 First nine drum modes, ordered by frequency. Red regions vibrate up when white regions vibrate down and vice versa. Drum vibration patterns may combine these modes. Underneath each mode is the frequency relative to the first mode (the fundamental).

Notice that when defining the term *fundamental* I used the word *often* three times: the fundamental is *often* the lowest harmonic, *often* the loudest, and *often* what determines the pitch. This is because sometimes one or more of those things isn't true. For example, organs often have one or two fairly loud partials an octave or two *below* the fundamental (an octave lower corresponds to half the frequency). Similarly, bells will usually have at least one large partial lower than the fundamental called the **hum tone**. In fact, the hum tone *is* in many ways *the* fundamental, but we usually identify the pitch of the bell (its so-called **strike tone**) with the second harmonic (the **prime**), which is also usually louder. Thus the prime is typically thought of as the fundamental.

Bells are bizarre. The next major partial up from the prime is usually the **tierce**,⁶ and it's just a *minor third* above the prime, or about 1.2 times the prime frequency.⁷ There are other inharmonic partials as well. And yet we perceive bells as, more or less, tonal. The specific amplitudes of the various partials in bells can cause us to associate the pitch with partials other than the prime. In fact, bells may cause us to associate the pitch with a partial that *doesn't even exist in the sound*.

Finally, drums have their own unique harmonic characteristics quite unlike strings or pipes. A drum is a 2D sheet, and so its harmonic vibration patterns (or modes) are *two dimensional*. This results in a complex series of partial frequencies, as shown in Figure 16, which are generally atonal.⁸

2.1 Units of Measure

Frequency The frequency of a partial is typically measured in *Hertz* (or Hz), which is the number of cycles of its sine wave per second. 1Hz means a sine wave whose full wave takes 1 second to complete. The **period** of a partial is the amount of time, in seconds, for its sine wave to complete one cycle. This is the inverse of frequency: thus if a cycle has a period of p , then it has a frequency of $f = \frac{1}{p}$ Hz.

Another measure of frequency we'll see is **angular frequency**, which is in *radians per second*. Angular frequency ω often appears in the imaginary portion of a complex number, and so you'll see it appearing in things like $i\omega$ or $e^{i\omega}$. It's closely related to Hertz: specifically, $2\pi\omega = 1$ Hz.

⁶Bells have classic names for their unusual partials: hum, prime, tierce, quint, nominal, deciem, unideciem, etc.

⁷One major consequence of this very loud minor third partial is that songs in major keys can sound horrible when played on a **carillon** (an instrument consisting of a large collection of bell-tower bells). Consider a major chord C, E, G. The C bell produces both the C prime and a very loud E♭ tierce which is sounded at the same time as the next note in the chord: E. The simultaneous E♭ and E sound pretty bad. Songs in minor keys sound only a little bit better than major keys: they too have this dissonance, just further up in the chord. Compositions specifically for the carillon are usually written in **fully diminished** scales and chords, that is, ones consisting only of minor thirds, such as C, E♭, G♭, A, C. That way the tierce of each bell lines up harmoniously with the prime of the next note in the chord.

⁸In a famous paper, Mark Kac asked whether the 2D mode patterns of a drum could be ascertained from the sound produced (Mark Kac, 1966, Can one hear the shape of a drum?, *American Mathematical Monthly*, 73(4)). It took 30 years to determine that the answer was no (Carolyn Gordon, David Webb, and S. Wolpert, 1992, Isospectral plane domains and surfaces via riemannian orbifolds, *Inventiones Mathematicae*, 110(1)).

Frequency is of course closely associated with **pitch**: what note the sound is being played at. Pitch goes up logarithmically with frequency. When a frequency is doubled, its perceived pitch has gone up an octave. More precisely, if a note is a certain frequency f , and we go up n semitones (half-steps) from that note, the new note is at frequency $g = 2^{n/12} \times f$. Similarly, if you have gone from frequency f to frequency g , then you have moved $n = 12 \log_2(g/f)$ semitones in pitch.

Amplitude We'll usually describe amplitude as the actual y value of the sound wave itself. However the amplitude of a signal is occasionally converted into \log_{10} and described in terms of **decibels** (dB): specifically, $1\text{dB} = 20 \log_{10} \times \text{change in amplitude}$. Doubling the amplitude is approximately an increase of 6dB. A doubling in perceived volume is often described as an increase of 10 dB. Decibels are a relative measure. Thus you will very often see *negative decibels* relative to some signal volume, to indicate sounds quieter than that signal.

Sine waves of course go both above 0 amplitude and below it: thus a sine wave may be described informally as having a "negative amplitude" at a certain point: though technically amplitude, like volume, is a magnitude measure and is only positive. Last, when we are multiplying an amplitude or volume to make it louder or softer, we are often said to be modifying the **gain** of the signal.

Phase Because we're talking about sine waves, the phase of a partial is an angular measure and so is typically expressed as a value from $0\dots 2\pi$ (or if you like, $-\pi\dots\pi$). In Figure 13 the green and blue sine waves are out of phase by π .

The Stereo Field When sounds are in stereo, they can appear to come from left of us, in center, or to the right of us. The angular position from which a sound appears to originate is known as the **pan** position of the sound.

2.2 Digitization of Sound Waves

Sound waves can be stored in (effectively) a real-valued form, on tapes or on records, etc. But modern sound is normally stored in digital form. To do this, the sound is *sampled* at uniform intervals of time, and the amplitude of each sample (positive or negative) is typically stored as an integer. Thus discretization occurs in two directions: (1) a discrete number of samples (2) each sample stored as a discrete integer.

A popular way to think of these samples is to lay them out on a grid, as shown in Figure 17 (left subfigure), where the x dimension is (discretized) time, one unit per sample, starting at 0 and increasing; and the y dimension is the (discretized) amplitude of each sample, going from -1 to $+1$. But it's dangerous to think of it this way, because it implies that when the wave is played, it takes the form of a blocky function with all horizontal and vertical lines (shown in red) and right angles. This is not really what happens.

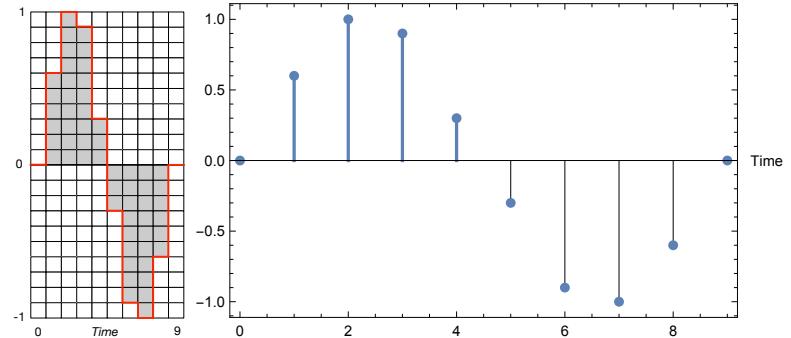


Figure 17 A sine wave discretized and shown in grid form (left) and as a lollipop graph (right).

Instead, to play a digital sound, it is first fed to a **Digital-Analog Converter** or **DAC**.⁹ This device changes its output voltage abruptly to match each new sample as it is being played: in this sense it resembles the blocky function. But then this voltage is fed into a **filter**, often a capacitor, which converts the abrupt voltage change into one which slides smoothly from one sample value to the next, producing a curvy, smooth output.

Thus it might be best to think of a digitized wave as a lollipop graph, as in Figure 17 (right subfigure). This helps remind you that the samples are not a bunch of blocky lines, but are in fact just numbers sampled from a real-valued function (the original sound), at very specific and precise times, and from which another real-valued function can be produced.

The Nyquist Limit and Aliasing The highest possible frequency that can be faithfully represented in a digitized sound of sampling rate n is exactly $n/2$. This is known as the **Nyquist limit**. However it is possible to draw digital waves which contain within them higher frequency partials than the Nyquist limit: these waves do not present themselves as proper partials of a given frequencies, but instead create unusual artifacts known as **aliasing** (or **foldover**).¹⁰ To prevent aliasing, audio devices apply a **low pass filter** to strip out frequencies higher than Nyquist before reducing a sound wave to a given sampling rate. For more on this (and it's pretty important), see Section 7.2.

Sampling Rates When a sound is sampled, the *speed* of the sampling is known as the **sampling rate**. A sampling rate of n kHz means that one sample is done every $1/(n \times 1000)$ of a second. One common sampling rate is 44.1 kHz (that is, one sample every $1/44,100$ of a second): this is the sampling rate of a compact disc, and is a common rate produced by many early digital synthesizers. Another popular rate is 48 kHz (one sample every $1/48,000$ of a second): this is a common rate in sound production: it was the sampling rate of Digital Audio Tape and had long been used in laboratory settings. A third popular rate in sound production is 96 kHz.

Why these values? 44.1 kHz was chosen by Sony in 1979 for the Compact Disc for a very specific reason. The maximum frequency that humans (typically teenagers) can hear is approximately 20 kHz. Thus a reasonable sampling rate for human-perceptible sound would be one which can accommodate at least 20 kHz. However to prevent aliasing, a recording application would need to apply a low-pass filter at 20 kHz. Low-pass filters cannot cut off frequencies precisely: they need some degree of wiggle-room in frequency. It turns out that 2.05 kHz is adequate wiggle-room. This means that the sampling rate would need to handle a grand total of 22.05 kHz. If you recall from the Nyquist Limit, the sampling rate is twice the maximum frequency: hence 44.1 kHz.

48 kHz seems more reasonable: it too is sufficient to cover 20 kHz, with even more wiggle room for the low-pass filter, and it's divisible by many different integers. 96 kHz is simply twice 48 kHz.

Bit Depth The sampling rate defines the x axis of the digitized signal: the bit depth defines the y axis. Bit depth is essentially the *resolution* of the amplitude of the wave. The most common bit

⁹A DAC outputs sound. What device would do sound input, or sampling? That would be, naturally, an **Analog-Digital Converter** or **ADC**.

¹⁰I learned this the hard way a long time ago as an undergraduate student. I had created a sound editor where you could draw the wave as a bitmap, and found that if you drew a perfect sawtooth wave (a 45-degree angle going up to the top, then a sharp vertical line going down, and repeating) it created strange artifacts. This was because it's possible to *store* a sawtooth wave as a digital representation, but this in fact was stuffing in some partials above the Nyquist limit which created bad aliasing problems.

depth is 16 bits: that is, each sample is a 16-bit unsigned integer.¹¹ This implies that a sample can be any one of $2^{16} = 65536$ possible values. The notional center position is half this (32768); this is the canonical 0-amplitude position. A sine wave would oscillate up above, then down below, the center position.

You'd think that small bit depths result in a "low resolution" sound in some sense, but this isn't the effect. Rather, bit depth largely defines the **dynamic range** of the sound: the distance in amplitude between the loudest possible sound representable and the quietest sound before the sound is overwhelmed by hiss. The point at which you can't hear quiet sounds any more because there's too much hiss is called the **noise floor**. This is also closely associated with the **signal to noise ratio** of a medium. A higher bit depth largely translates into more dynamic range. Since this is a difference in amplitudes, it's measured in dB: a bit depth of n yields a difference in dB of roughly $6n$.

Viewed this way, even analog recording media can be thought of as having an effective "bit depth" based on its dynamic range. A vinyl record has at most a "bit depth", so to speak, of 10–11 bits (that is, 60–72 dB). A typical cassette tape is between 6–9 bits. Some very high end reel-to-reel tapes might be able to achieve upwards of 13–14 bits. These are all quite inferior to CDs, at 16 bits. And DVDs support a bit depth of 24 bits!

Compression Schemes Compression won't come into play much in the development of a music synthesizer, but it's worth mentioning it. The human auditory system is rife with unusual characteristics which can be exploited to remove, modify, or simplify a sound without us being able to tell. One simple strategy used in many early (and current!) sound formats is **companding**. This capitalizes on the fact that humans can distinguish between different soft- or medium-volume sounds more easily than different high-volume sounds. Thus we might use the bits in our sample to encode logarithmically: quiet sounds get higher resolution than loud sounds. Early techniques which applied this often used either the **μ -law** or **a-law** companding algorithms.¹²

More famous nowadays are **lossy compression** schemes such as **MP3**, which take advantage of a variety of eccentricities in human hearing to strip away portions of a sound without being detected. For example, humans are bad at hearing sounds if there are other, louder sounds near them in frequency. MP3 will remove the quieter sound under the (usually correct) assumption that we wouldn't notice. MP3 generally has a fixed **bitrate**, meaning the number of bits MP3 uses up to record a second of audio. But if some sound has a lot of redundancy in it (as an extreme example: total silence), some compression schemes take advantage of this to compress different parts of a sound stream at different bitrates as necessary. This is known as a **variable bitrate scheme**.

Channels Another final factor in the size of audio is the number of **channels** it consumes. A channel is a single sound wave. Stereo audio will consist of two parallel sound waves, that is, two channels. **Quadraphonic** sound, which was designed to be played all around the listener, has four channels. Channels may serve different functions as well: for example in a movie theater one channel, largely for voice, is routed directly behind the screen, while two or more channels provide a stereo field on both sides of the viewer, and an additional channel underneath the viewer drives the subwoofer. Similar multi-channel formats have made their way into home theaters, such as **5.1 surround sound**, which requires six channels.

¹¹You could certainly use a 2's-complement signed representation with 0 at the center instead.

¹²If you think about it, these are in some sense a way of representing your sound as floating-point.

3 The Fourier Transform

As discussed earlier, any sound wave can be represented as a series of sine waves which differ in frequency, amplitude, and phase. In Section 4, we will see how to take advantage of this to produce sound through additive synthesis. Here we will consider the subject somewhat formally, and also discuss useful algorithms which automatically convert sound to and from the time and frequency domains. These algorithms are useful for many reasons, which we discuss later.

For any sound wave $s(t)$, where t is the time, we have some function $S(f)$ describing the frequency spectrum. This function, with some massaging, provides us with the amplitude and phase of each sine wave of frequency f participating in forming the sound wave $s(t)$. As it turns out, both $S(f)$ and $s(t)$ are functions which yield complex numbers, though when used for sounds, the imaginary portion of $s(t)$ is ignored (both the imaginary and real portions of $S(f)$ are used to compute phase and magnitude).

We can convert from $s(t)$ to $S(f)$ using the **Fourier Transform**.¹³ The **Inverse Fourier Transform** does the opposite: it converts $S(f)$ into $s(t)$. The two transform functions are so similar that, as we'll see, they're practically the same procedure. It's useful to first see those sines and cosines being constructed to form a sound wave. So let's look at the *Inverse Fourier Transform* initially, to get an intuitive feel for this:

$$s(t) = \int_{-\infty}^{\infty} S(i\omega) (\cos(\omega t) + i \sin(\omega t)) d\omega$$

Note that ω is the *angular frequency* of a sine wave. So what this is doing is, for every possible frequency (including negative ones!), we're computing the sine wave in both its real- and imaginary components, multiplied by our (complex) spectral value at that frequency, $S(i\omega)$, which includes both the amplitude and the phase of the sine wave in question. Add all of these sine waves up and you get the final wave.

This isn't the classic way to describe the Inverse Fourier Transform. Instead, we'd use **Euler's Formula**,¹⁴ $e^{i\theta} = \cos(\theta) + i \sin(\theta)$, to cast the cos and sin into an exponential. This results in:

$$\begin{aligned} s(t) &= \int_{-\infty}^{\infty} S(i\omega) (\cos(\omega t) + i \sin(\omega t)) d\omega \\ &= \int_{-\infty}^{\infty} S(i\omega) e^{i\omega t} d\omega \end{aligned}$$

¹³The Fourier Transform is named after Joseph Fourier, who in 1822 showed that arbitrary waves could be represented as a large sum of sine waves (the **Fourier Series**).

¹⁴There are lots of ways to intuitively explain why $e^{i\theta} = \cos(\theta) + i \sin(\theta)$ using rotation about the complex unit circle as shown in Figure 18. But maybe it's easier to just explain with **Taylor series**. Here are three classic Taylor series expansion identities:

$$\cos(\theta) = 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \dots \quad \sin(\theta) = \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \dots \quad e^\theta = 1 + \theta + \frac{\theta^2}{2!} + \frac{\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{\theta^5}{5!} + \frac{\theta^6}{6!} + \frac{\theta^7}{7!} \dots$$

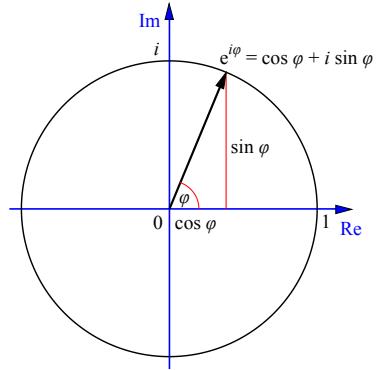


Figure 18 Euler's Formula on the complex plane. The X axis is the real axis, and Y is the imaginary axis.^{©10}

As it turns out, the (forward) Fourier Transform is eerily similar to the inverse. Note that the big difference is a minus sign:

$$\begin{aligned} S(i\omega) &= \int_{-\infty}^{\infty} s(t) (\cos(\omega t) - i \sin(\omega t)) dt \\ &= \int_{-\infty}^{\infty} s(t) e^{-i\omega t} dt \end{aligned}$$

Yes, that's going from negative infinity to positive infinity in *time*. The Fourier Transform reaches into the far past and the far future and sums all of it.

3.1 The Discrete Fourier Transform

The Fourier Transform above is continuous, which isn't going to happen in a computer. And it's also considering things like infinite positive and negative time and infinite positive and negative frequencies. We need a discrete version.

Specifically, we are often faced with the following situation. We have sampled some N evenly spaced samples of sound comprising a total amount of time T . The sampling rate—the amount of time between a pair of samples—is thus T/N . The samples will be called $t = 0, 1, 2, 3, \dots, N-1$. Not only are the number of samples discrete, but the number of *frequencies* will wind up being discrete as well. The angular frequencies for $i\omega$ are $f \times \omega \frac{2\pi}{T}$ for integer values of f from 0 to $N-1$ inclusive. Using our discretized t and f values, our two equations transform to:¹⁵

$$S(f) = \sum_{t=0}^{N-1} s(t) e^{-i2\pi f t \frac{1}{N}} \quad s(t) = \frac{1}{N} \sum_{f=0}^{N-1} S(f) e^{i2\pi f t \frac{1}{N}} \quad (t, f = 0, 1, \dots, N-1)$$

Note that because our sampled sound is no longer infinite in length, we now have a notion of a *maximal wavelength*: the biggest sine wave we can use in our sound is one whose period is T .

You could see it in sine/cosine form by applying Euler's Formula¹⁶ again: remember it's $e^{i\theta} = \cos(\theta) + i \sin(\theta)$. This yields:

This means that

$$\begin{aligned} e^{i\theta} &= 1 + i\theta + \frac{(i\theta)^2}{2!} + \frac{(i\theta)^3}{3!} + \frac{(i\theta)^4}{4!} + \frac{(i\theta)^5}{5!} + \frac{(i\theta)^6}{6!} + \frac{(i\theta)^7}{7!} + \dots = 1 + i\theta - \frac{\theta^2}{2!} - \frac{i\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{i\theta^5}{5!} - \frac{\theta^6}{6!} - \frac{i\theta^7}{7!} \dots \\ &= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \dots\right) + \left(i\theta - \frac{i\theta^3}{3!} + \frac{i\theta^5}{5!} - \frac{i\theta^7}{7!} + \dots\right) \\ &= \left(1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \dots\right) + i\left(\theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \dots\right) = \cos(\theta) + i \sin(\theta) \end{aligned}$$

¹⁵Some people instead prefer to split the $\frac{1}{N}$ among the two transformations as $\frac{1}{\sqrt{N}}$, which results in nearly identical equations:

$$S(f) = \frac{1}{\sqrt{N}} \sum_{t=0}^{N-1} s(t) e^{-i2\pi f t \frac{1}{N}} \quad s(t) = \frac{1}{\sqrt{N}} \sum_{f=0}^{N-1} S(f) e^{i2\pi f t \frac{1}{N}} \quad (t, f = 0, 1, \dots, N-1)$$

¹⁶By the way, a degenerate case of this formula is one of the most spectacular results in all of mathematics. Specifically, if we set $\theta = \pi$, then we have $e^{i\theta} = e^{i\pi} = \cos(\pi) + i \sin(\pi) = -1 + i(0) = -1$. From this we have $e^{\pi i} + 1 = 0$, an amazing equation containing exactly the five primary constants in mathematics

$$S(f) = \sum_{t=0}^{N-1} s(t) \left(\cos\left(-2\pi ft \frac{1}{N}\right) + i \sin\left(-2\pi ft \frac{1}{N}\right) \right)$$

$$s(t) = \frac{1}{N} \sum_{f=0}^{N-1} S(f) \left(\cos\left(2\pi ft \frac{1}{N}\right) + i \sin\left(2\pi ft \frac{1}{N}\right) \right) \quad (t, f = 0, 1, \dots, N-1)$$

A bit of trigonometry allows us to convert the first equation to:

$$S(f) = \sum_{t=0}^{N-1} s(t) \left(\cos\left(2\pi ft \frac{1}{N}\right) - i \sin\left(2\pi ft \frac{1}{N}\right) \right)$$

$$s(t) = \frac{1}{N} \sum_{f=0}^{N-1} S(f) \left(\cos\left(2\pi ft \frac{1}{N}\right) + i \sin\left(2\pi ft \frac{1}{N}\right) \right) \quad (t, f = 0, 1, \dots, N-1)$$

Notice that these two transforming equations are identical except for a minus sign and $1/N$. This allows us to create a unified algorithm for them called the **Discrete Fourier Transform** or **DFT** (and the **Inverse Discrete Fourier Transform** or **IDFT**).

Algorithm 1 The Discrete Fourier Transform

```

1:  $Xr \leftarrow \langle Xr_0 \dots Xr_{N-1} \rangle$  array of  $N$  elements representing the real values of the input
2:  $Xi \leftarrow \langle Xi_0 \dots Xi_{N-1} \rangle$  array of  $N$  elements representing the imaginary values of the input
3:  $forward \leftarrow$  Is this a forward (as opposed to inverse) transform?

4:  $Yr \leftarrow \langle Yr_0 \dots Yr_{N-1} \rangle$  array of  $N$  elements representing the real values of the output
5:  $Yi \leftarrow \langle Yi_0 \dots Yi_{N-1} \rangle$  array of  $N$  elements representing the imaginary values of the output
6: for  $n$  from 0 to  $N-1$  do
7:    $Yr_n \leftarrow 0$ 
8:    $Yi_n \leftarrow 0$ 
9:   for  $m$  from 0 to  $N-1$  do
10:    if  $forward$  then
11:       $z \leftarrow -2\pi mn \frac{1}{N}$                                  $\triangleright$  The only difference in the equations is the minus sign
12:    else
13:       $z \leftarrow 2\pi mn \frac{1}{N}$ 
14:     $Yr_n \leftarrow Yr_n + Xr_m \cos(z) - Xi_m \sin(z)$            $\triangleright$  This is just the  $e^{i\theta}$  stuff
15:     $Yi_n \leftarrow Yi_n + Xi_m \cos(z) - Xr_m \sin(z)$            $\triangleright$  and multiplying complex numbers
16:    if not  $forward$  then
17:       $Yr_n \leftarrow \frac{Yr_n}{N}$ 
18:       $Yi_n \leftarrow \frac{Yi_n}{N}$ 
19: return  $Yr$  and  $Yi$ 
```

3.2 Computing Amplitude and Phase

In the Forward DFT, each slot in the resulting frequency domain array is known as a **bin** in the Fourier transform world. For each bin $n = 0 \dots (N-1)$, the frequency value is a complex number Y_n consisting of real Yr_n and imaginary Yi_n components. From this we can extract:

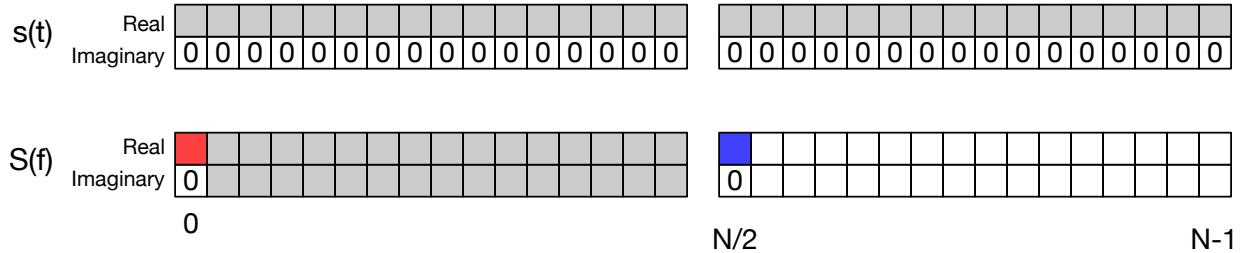


Figure 19 Values of interest in the Time domain $s(t)$ and Frequency domain $S(f)$ arrays for a Real-Valued DFT. Gray, Red, and Blue boxes show numerical values of interest. Boxes with 0 in them should (or will) be set to 0. The red box at 0 is the value of the DC Offset. The blue box at $N/2$ is the value of the Nyquist frequency bin, and is only important to retain if one ultimately needs to reverse the process via an inverse transform; otherwise it can be ignored. The blank white boxes are just reflected complex conjugates of the gray boxes in $S(f)$, and can be ignored since they are redundant.

- The **amplitude** of the bin is just the magnitude $|Y_n|$, that is, $\sqrt{Y_{r_n}^2 + Y_{i_n}^2}$. When Y_n is only real valued, the magnitude is simply its absolute value.
- The **phase** of the bin is $\tan^{-1}\left(\frac{Y_{i_n}}{Y_{r_n}}\right)$
- The **frequency** of the bin (for the first $N/2 + 1$ elements) is $n/N \times R$, where R is the sampling rate (44.1K for example). As we'll see in the next section, we're really only interested in the first $N/2 + 1$ elements.

3.3 Real-Valued Fourier Transforms

Now, our sound is not a bunch of complex numbers: it's a bunch of *real-valued* numbers. This means that when we apply the DFT to it, the imaginary portion X_i will be all zeros. This has an interesting effect: there will be a certain mirror symmetry among the outputs $Y_1 \dots Y_{N-1}$. Specifically, each output Y_n will be the complex conjugate of output Y_{N-n} starting at $n = 1$. That is, for all $n > 1$, $Y_{i_n} = -Y_{i_{N-n}}$.

However output Y_0 will not be part of this symmetry pattern: it will be real-valued only (that is, $Y_{i_0} = 0$). This is the "0 Frequency" bin, or **DC Offset Bin**: the vertical offset of the sound. Furthermore, if N is even—which is usually the case—then this implies that output $Y_{N/2}$ (the center point in the symmetry) will be equal to its own complex conjugate, which also implies that it must be real-valued ($Y_{i_{N/2}} = 0$). This is the **Nyquist Frequency Bin**, and it represents frequencies beyond what can properly be represented. If you are planning on doing a DFT, modifying the values, and then doing an inverse DFT to output the result in the time domain, you'll need to hold onto the Nyquist frequency bin value; otherwise you can ignore it.

Figure 19 shows this situation. This symmetry means that, when the time domain is real-valued, only slots $0 \dots N/2 - 1$ (and possibly $N/2$) matter in the frequency domain: the remaining slots are just reflective complex conjugates of something else. This should make sense: the **Nyquist limit** indicates that the largest possible frequency which can be embedded in a digital signal is half the sampling rate. So if we have a 1 second clip sampled at 44.1KHz, and thus have 44,100 samples, even though we get back 44,100 "frequency bins", in fact only the first 22,050 (+ 1) are relevant.

3.4 The Fast Fourier Transform

The problem with the DFT is that it is slow: its two for-loops means that it's obviously $O(N^2)$. But it turns out that with a few clever tricks we can come up with a version of the DFT which is only $O(N \lg N)$! This faster version is called the **Fast Fourier Transform** or **FFT**.¹⁷ The FFT uses a divide-and-conquer approach to recursively call smaller and smaller FFTs.

Recall that the forward DFT looks like this:

$$S(f) = \sum_{t=0}^{N-1} s(t)e^{-i2\pi ft \frac{1}{N}}$$

What if we divided the summing process into two parts: summing the *even* values of t and the *odd* values of t separately? We could write it this way:

$$\begin{aligned} S(f) &= \sum_{t=0}^{N-1} s(t)e^{-i2\pi ft \frac{1}{N}} \\ &= \sum_{t=0}^{N/2-1} s(2t)e^{-i2\pi f(t \times 2) \frac{1}{N}} + \sum_{t=0}^{N/2-1} s(2t+1)e^{-i2\pi f(t \times 2+1) \frac{1}{N}} \\ &= \sum_{t=0}^{M-1} s(2t)e^{-i2\pi f(t \times 2) \frac{1}{2M}} + \sum_{t=0}^{M-1} s(2t+1)e^{-i2\pi f(t \times 2+1) \frac{1}{2M}} \quad (M = N/2) \\ &= \sum_{t=0}^{M-1} s(2t)e^{-i2\pi f(t \times 2) \frac{1}{2M}} + \sum_{t=0}^{M-1} s(2t+1)e^{-i2\pi f(t \times 2) \frac{1}{2M}} \times e^{-i2\pi f \frac{1}{2M}} \\ &= \sum_{t=0}^{M-1} s(2t)e^{-i2\pi ft \frac{1}{M}} + e^{-i2\pi f \frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1)e^{-i2\pi ft \frac{1}{M}} \\ &= \sum_{t=0}^{M-1} s(2t)e^{-i2\pi ft \frac{1}{M}} + e^{-i2\pi f \frac{1}{N}} \times \sum_{t=0}^{M-1} s(2t+1)e^{-i2\pi ft \frac{1}{M}} \quad (M = N/2) \end{aligned}$$

Let's call those two splits $E(f)$ and $O(f)$ for *even* and *odd*:

$$S(f) = E(f) + e^{-i2\pi f \frac{1}{N}} \times O(f)$$

It turns out that if we use this equation to compute $S(f)$ for f from $0 \dots S(N/2 - 1)$, we can *reuse* our $E(f)$ and $O(f)$ to compute $S(f)$ for $N/2 \dots N - 1$. That's the divide-and-conquer bit. So let's assume that the above derivation is for just the first case. We'll derive a similar equation the second case, $S(f + N/2)$, otherwise known as $S(f + M)$.

To do this we take advantage of two identities. The first is that for any integer k , it's the case that $e^{-i2\pi k} = 1$. The second is that $e^{-i\pi} = 1$:

¹⁷The DFT has been around since 1828, reinvented in many guises. The FFT in its current form is known as the **Cooley-Tukey FFT**, by **James William Cooley** and **John Tukey** circa 1965. Tukey is famous for lots of things in statistics as well, not the least of which is the invention of the box plot. But interestingly, the FFT in fact *predates* the DFT: it was actually invented by (who else?) **Carl Friedrich Gauss**. Gauss developed it as part of his astronomical calculations in 1801, but did not publish the results. No one noticed even when his collected works were published in 1866.

$$\begin{aligned}
S(f + M) &= \sum_{t=0}^{N-1} s(t) e^{-i2\pi(f+M)t\frac{1}{N}} \\
&= \sum_{t=0}^{N/2-1} s(2t) e^{-i2\pi(f+M)(t\times 2)\frac{1}{N}} + \sum_{t=0}^{N/2-1} s(2t+1) e^{-i2\pi(f+M)(t\times 2+1)\frac{1}{N}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi(f+M)(t\times 2)\frac{1}{2M}} + \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi(f+M)(t\times 2+1)\frac{1}{2M}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi(f+M)(t\times 2)\frac{1}{2M}} + e^{-i2\pi(f+M)\frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi(f+M)(t\times 2)\frac{1}{2M}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi(f+M)t\frac{1}{M}} + e^{-i2\pi(f+M)\frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi(f+M)t\frac{1}{M}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi f t \frac{1}{M}} e^{-i2\pi M t \frac{1}{M}} + e^{-i2\pi(f+M)\frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi f t \frac{1}{M}} e^{-i2\pi M t \frac{1}{M}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi f t \frac{1}{M}} + e^{-i2\pi(f+M)\frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi f t \frac{1}{M}} \quad (\text{First Identity}) \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi f t \frac{1}{M}} + e^{-i2\pi f \frac{1}{2M}} e^{-i2\pi M \frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi f t \frac{1}{M}} \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi f t \frac{1}{M}} - e^{-i2\pi f \frac{1}{2M}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi f t \frac{1}{M}} \quad (\text{Second Identity}) \\
&= \sum_{t=0}^{M-1} s(2t) e^{-i2\pi f t \frac{1}{M}} - e^{-i2\pi f \frac{1}{N}} \times \sum_{t=0}^{M-1} s(2t+1) e^{-i2\pi f t \frac{1}{M}}
\end{aligned}$$

Notice that once again we have the same splits $E(f)$ and $O(f)$! So we can say:

$$S(f + N/2) = E(f) - e^{-i2\pi f \frac{1}{N}} \times O(f)$$

So if we wanted to compute $S(f)$ for all $f = 0 \dots N-1$, we could do it like this:

1. Compute $E(f)$ for all $f = 0, 2, 4, \dots, N-2$
2. Compute $O(f)$ for all $f = 1, 3, 5, \dots, N-1$
3. For all $f = 0 \dots N/2-1$, compute $S(f) = E(f) + e^{-i2\pi f \frac{1}{N}} \times O(f)$
4. For all $f = 0 \dots N/2-1$, compute $S(f + N/2) = E(f) - e^{-i2\pi f \frac{1}{N}} \times O(f)$

What this all means is that to compute $S(f)$, we just need to compute $O(f)$ and $E(f)$, and then use each of them *twice*. What are $O(f)$ and $E(f)$? They're themselves Fourier Transforms on $s(2t)$ and $s(2t+1)$ respectively, and since they only go from $0 \dots M-1$, they're half the size of $S(f)$! In short, to compute a Fourier Transform, we can compute two half-size Fourier Transforms, and then use them twice each. This is recursive: each of *them* will require two *quarter-size* Fourier Transforms, and so on until we get down to an array of just size 1.

Steps 3 and 4 together are N in length. Similarly when we're inside $O(f)$ or $E(f)$, steps 3 and 4 are $N/2$ in length: but there's two of them ($O(f)$ and $E(f)$). Continuing the recursion to the next level, steps 3 and 4 are $N/4$ in length, but there are 4 of them, and so on, all the way down to N individual computations of size 1. Thus at any level, we have $O(N)$ computations.

How many levels do we have? We start with 1 size- N computation, then 2 size- $N/2$ computations, then 4, then 8, ... until we get to N size-1 computations. The length of $\langle 1, 2, 4, 8, \dots, N \rangle$ is $\lg N$. So our total cost is $O(N \lg N)$.

This divide-by-2-and-conquer strategy assumes, of course, that N is a power of 2. If your sample count is not a power of 2, there are a number of options for handling this not discussed here. The FFT is thus:

Algorithm 2 *RecursiveFFT (Private Subfunction)*

```

1:  $Xi \leftarrow \langle X_{i_0} \dots X_{i_{N-1}} \rangle$  array of  $N$  elements representing the imaginary values of the input
2:  $Xr \leftarrow \langle X_{r_0} \dots X_{r_{N-1}} \rangle$  array of  $N$  elements representing the real values of the input

3:  $N \leftarrow$  length of  $Xi$ 
4: if  $N = 1$  then
5:   return  $Xr$  and  $Xi$ 
6: else
7:    $Yi \leftarrow \langle Y_{i_0} \dots Y_{i_{N-1}} \rangle$  array of  $N$  elements representing the imaginary values of the output
8:    $Yr \leftarrow \langle Y_{r_0} \dots Y_{r_{N-1}} \rangle$  array of  $N$  elements representing the real values of the output
9:    $M \leftarrow N/2$ 
10:   $Ei \leftarrow \langle E_{i_0} \dots E_{i_{M-1}} \rangle$  even-indexed elements from  $Xi$                                  $\triangleright \forall x : E_{i_x} = X_{i_{2x}}$ 
11:   $Er \leftarrow \langle E_{r_0} \dots E_{r_{M-1}} \rangle$  even-indexed elements from  $Xr$                                  $\triangleright \forall x : Er_x = X_{r_{2x}}$ 
12:   $Oi \leftarrow \langle O_{i_0} \dots O_{i_{M-1}} \rangle$  odd-indexed elements from  $Xi$                                  $\triangleright \forall x : O_{i_x} = X_{i_{2x+1}}$ 
13:   $Or \leftarrow \langle O_{r_0} \dots O_{r_{M-1}} \rangle$  odd-indexed elements from  $Xr$                                  $\triangleright \forall x : Or_x = X_{r_{2x+1}}$ 
14:   $Ei, Er \leftarrow \text{RecursiveFFT}(Ei, Er)$ 
15:   $Oi, Or \leftarrow \text{RecursiveFFT}(Oi, Or)$ 
16:  for  $n$  from 0 to  $M - 1$  do                                 $\triangleright e^{-i2\pi f/N} = \cos(2\pi f/N) - i \sin(2\pi f/N)$ 
17:     $\theta \leftarrow -2\pi n / N$ 
18:     $Yr_n \leftarrow Er_n + \cos(\theta)Or_n$ 
19:     $Yi_n \leftarrow Ei_n - \sin(\theta)Oi_n$ 
20:  for  $n$  from  $M$  to  $N - 1$  do
21:     $\theta \leftarrow -2\pi n / N$ 
22:     $Yr_n \leftarrow Er_n - \cos(\theta)Or_n$ 
23:     $Yi_n \leftarrow Ei_n + \sin(\theta)Oi_n$ 
24: return  $Yr$  and  $Yi$ 

```

Algorithm 3 *The Fast Fourier Transform*

```

1:  $Xi \leftarrow \langle X_{i_0} \dots X_{i_{N-1}} \rangle$  array of  $N$  elements representing the imaginary values of the input
2:  $Xr \leftarrow \langle X_{r_0} \dots X_{r_{N-1}} \rangle$  array of  $N$  elements representing the real values of the input

3:  $N \leftarrow$  length of  $Xi$ 
4:  $Yi, Yr \leftarrow \text{RecursiveFFT}(Xi, Xr)$ 
5: return  $Yr$  and  $Yi$ 

```

You can also easily do the FFT in an iterative rather than recursive form, that is, as a big loop largely relying on dynamic programming. It's a bit faster and doesn't use the stack, but it has no computational complexity advantage.

There of course exists an **Inverse Fast Fourier Transform** or IFFT. We could change some signs just like we did in the DFT: but instead let's show off an alternative approach. It turns out that the Inverse FFT is just the FFT on the complex conjugate of the data.¹⁸ That is:

$$\text{IFFT}(S) = \text{conj}(\text{FFT}(\text{conj}(S)))$$

...where $\text{conj}(C)$ applies the **complex conjugate** to every complex number $C_i \in C$. If you have forgotten, the conjugate of a complex number $a + bi$ is just $a - bi$. So we could write it like this:

Algorithm 4 The Inverse Fast Fourier Transform

- 1: $Xi \leftarrow \langle Xi_0 \dots Xi_{N-1} \rangle$ array of N elements representing the imaginary values of the input
- 2: $Xr \leftarrow \langle Xr_0 \dots Xr_{N-1} \rangle$ array of N elements representing the real values of the input

- 3: **for** n from 0 to $N - 1$ **do**
- 4: $Xi_n \leftarrow 0 - Xi_n$
- 5: $Yi, Yr \leftarrow \text{Fast Fourier Transform}(Xi, Xr)$
- 6: **for** n from 0 to $N - 1$ **do**
- 7: $Yi_n \leftarrow 0 - Yi_n$
- 8: **return** Yr and Yi

And that's all there is to it!

3.5 Windows

The Fourier Transform converts a sound of length N into amplitudes and phases for $N/2$ frequencies stored in $N/2$ bins. But those aren't necessarily all the frequencies in the sound: there's no reason we can't have a frequency that lies (say) half-way between two bins. Storing a frequency like this causes it to spread, or **leak**, into neighboring bins.

As a result, even a pure sine wave may not show up as a 1 in a certain bin and all 0 in the other bins. Rather, it might look something like Figure 20. In addition to the primary (nearest) bin, we see leakage out into other bins. The up/down pattern of leakage forms what are known as **sidelobes**. Often we'd like to reduce the sidelobe leakage as much as possible, and have the primary lobe to be as thin as possible, ideally fitting into a single bin.

We can't meet this ideal, but we have ways to rough it. The approach is to preprocess our sampled sound with a **window** before running it through the FFT. Using a window function $w(n)$ is very simple: you just multiply it against each of your samples s_0, \dots, s_{N-1} , resulting in $s_0 \times w(0), \dots, s_{N-1} \times w(N-1)$:

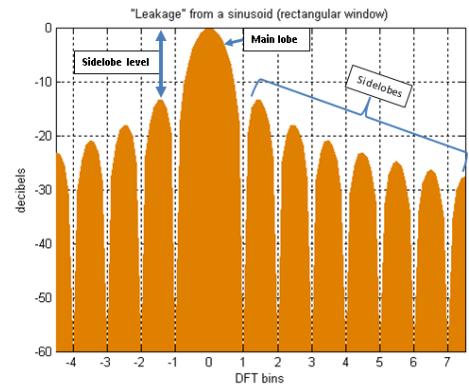


Figure 20 Sidelobes in an FFT (note that the Y axis is on a log scale).¹¹

¹⁸This works for the Inverse DFT too. And why not? They're effectively the same procedure.

Algorithm 5 Multiply by a Window Function

```

1:  $Xr \leftarrow \langle Xr_0 \dots Xr_{N-1} \rangle$  array of  $N$  elements representing sound samples
2:  $w(n, N) \leftarrow$  window function

3:  $Yr \leftarrow \langle Yr_0 \dots Yr_{N-1} \rangle$  array of  $N$  elements representing revised sound samples
4: for  $n$  from 0 to  $N - 1$  do
5:    $Yr_n \leftarrow Yr_n \times w(n, N)$ 
6: return  $Yr$ 
```

If you have no window function, then $w(n) = 1$ for all n . This is called the **rectangular window**. Most window functions are zero or near-zero at the ends and positive in the center. There are many window functions, depending on your particular needs. A very popular general-purpose window function is the **Hamming window**:

$$w(n, N) = 0.53836 - (1 - 0.53836) \times \cos\left(\frac{2\pi n}{N - 1}\right)$$

This is far from the only window option: there's Hann, Blackman, Triangular/Bartlett, Gaussian, Tukey, Planck-Taper, Slepian, Kaiser, Dolph-Chebyshev, Ultraspherical, Poisson, Lanczos,

3.6 Applications

The Fourier Transform isn't just for sounds; indeed this is a very minor use. It's used for signal processing of all kinds in everything from the study of electrical circuits to spectral analysis of stars. There are 2-D (or higher dimensional versions!) which operate on images and video. In fact, fastest known way to multiply two very large (like million-digit) numbers is to convert them with the FFT, perform a special operation on them, and convert them back!

Though the FFT has many applications outside of the audio realm, let's consider a few interesting cases in sound processing alone:

- **Visualization** This one is obvious: an FFT is great at visualization. You can easily analyze the amplitudes and phases at a variety of frequencies. If you do successive FFTs, perhaps one per tenth of a second, you could create a **spectrogram**, such as was shown in Figure 9 (another example is the bottom subfigure — the frequency domain — Figure 12 on page 14).
- **Filtering** You can accentuate, lower, or entirely strip out partials by converting the sound to the Fourier domain with an FFT, modifying (or zeroing out!) the amplitudes of interest, then converting back to the time domain with an IFFT. Similarly, you could modify the phases of various partials. In fact, it's often *much faster* to do an FFT, perform modifications in the frequency domain, then do an IFFT, than to just do the equivalent thing while in the time domain! Section 11.4 discusses an example of this in depth.

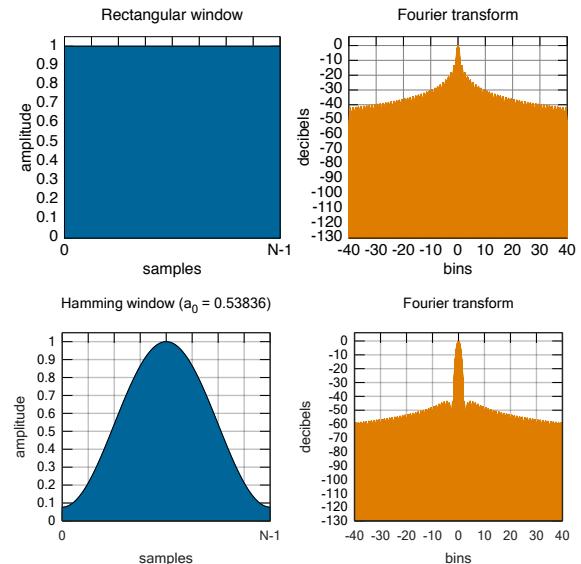


Figure 21 Rectangular and Hamming Windows and their effects on the frequency domain (note that the Y axis for the frequency domain is on a log scale).^{©12}

- **Pitch Scaling** It used to be that pitch shifting was done by recording at a very slow speed, then speeding it up: the Alvin and the Chipmunks effect. But the FFT can be used in a limited fashion to pitch shift (up or down) *without changing the speed*. This is known as **pitch scaling**. For example, to double frequency, just do an FFT, then just move each partial in the FFT array to the slot representing twice its frequency. Then do an IFFT to go back to the original sound.
- **Resynthesis** A sound is sampled and analyzed, and then recreated (more or less) using a synthesizer. One common use of resynthesis is a **vocoder**, which samples the human voice and then recreates it with a vocal synthesis method. Some resynthesis techniques work entirely in the time domain, but it's not uncommon to perform resynthesis by pushing the sound into the frequency domain where it's easier to manipulate and analyze.
- **Image Synthesis** Here you *start* with a spectrogram in the frequency domain, and allow the musician to modify it as if it were an image: indeed he might create an "image" from scratch rather than loading a sample in the first place. This is essentially tweaking the image much as one might do in Adobe Photoshop, then playing the result. Such tools are called **image synthesis** synthesizers. Many additive synthesizer tools, such as **Image-Line Software's Harmor**, also sport image-synthesis facilities.

Many of these techniques do not use a single FFT on the entire sound, but instead must break it into short samples, performing an FFT on each of them. These samples may have gaps between them, or may overlap with one another. This general approach is called the **Short Time Fourier Transform** (or **STFT**). For example, every column in Figure 22 is the result of an individual STFT. One challenge is that doing an STFT, then manipulating the partials, then performing an inverse STFT, results in a sound chunk which won't line up properly with its neighbors any more: there are various tricks applied in the STFT to compensate for this. See discussion of the STFT in Section 11.4.

There are alternatives. For example, it turns out that multiplying the amplitude or phase of partials in the frequency domain corresponds to a similar procedure in the time domain called **convolution**. Convolution is the basic tool used to develop filters solely in the time domain, which is the standard approach taken in Section 8. Similarly, there are various clever algorithmic tricks to approximate pitch scaling or pitch shifting in the time domain involving removal or interpolation of individual samples.

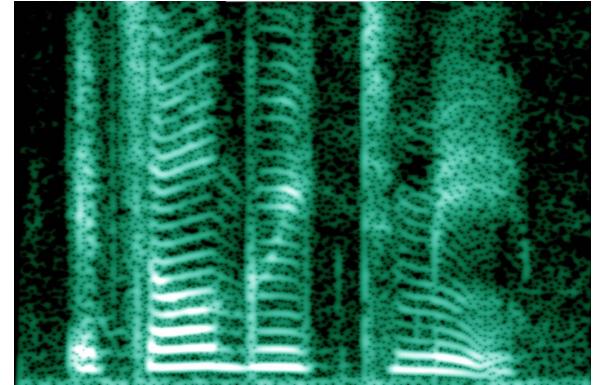


Figure 22 A Spectrogram (or Sonogram) of the human voice. The x axis is time, the y axis is frequency, and the brightness is amplitude.^{©13}

4 Additive Synthesis

An additive synthesizer builds a sound by producing and modifying a set of partials, then adding them up at the end to form the final sound wave. The partials could be added using an IFFT, or more commonly by adding up a bunch of sine waves. Additive synthesis is one of the most intuitive and straightforward ways of synthesizing sounds, and yet it is among the rarest due to its high number of parameters. It's not easy to develop an additive synthesizer that isn't tedious to use. The high computational cost of additive synthesizers has also restricted their availability compared to other techniques.

4.1 History

Additive synthesis is easily the oldest form of electronic music synthesis, and if we relaxed its definition to allow adding up waves beyond just sine waves, its history stretches back in time much further than that.

Organ makers have long understood the effect of playing multiple simultaneous pipes for a single note, each with its own set of partials, to produce a final mixed sound. Pipe organs are typically organized as sets of pipes (or **stops**), one per note, which produce notes with a certain timbre. As can be seen in Figure 23, stops are of many different shapes, and are made out of different materials, notably steel and wood. A full set of stops of a certain kind, one per note, is known as an **organ rank**.

Good organs may have many ranks. To cause an organ to play a stop from a rank when a note is played, a control called a **stop knob** or **drawknob** is pulled out. Organs can play many ranks from the same note at once by pulling out the appropriate stop knobs; in fact some ranks are even designed to play multiple pipes in the *same rank* in response to a single note (a concept called, in organ parlance, **mixtures**). If you wanted to go all-out, playing all the ranks at the same time, you would pull out all the stop knobs: hence the origin of the term "**to pull out all the stops**".

Early electronic synthesizer devices were largely additive, using **tonewheels** (also called **alternators**). A tonewheel, originally devised by Hermann von Helmholtz and later Rudolf Koenig (Figure 24), is a metal disk or drum with teeth (Figure 25). The tonewheel is spun, and an electromagnet is placed near it, and as the teeth on the tonewheel get closer or farther from the magnet, they induce a current in the magnet which produces an electronic wave.¹⁹ We can do a simple kind of additive synthesis by summing the sounds from multiple tonewheels at once.



Figure 23 (Left) Ranks of Organ Stops.^{©14} (Right) Organ Stop Knobs.^{©15}



Figure 24 Rudolf Koenig's synthesizer.^{©16}

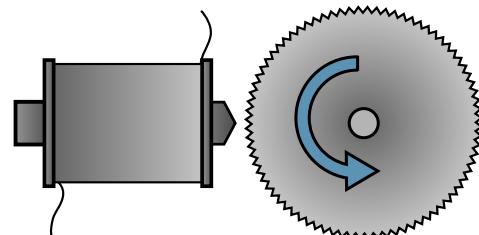


Figure 25 Diagram of a tonewheel. As the wheel spins, its teeth alternatively get closer to or farther from an electromagnet, causing the magnet to produce a wave.^{©17}

¹⁹This magnetic induction is essentially the same concept as an electric guitar pickup.

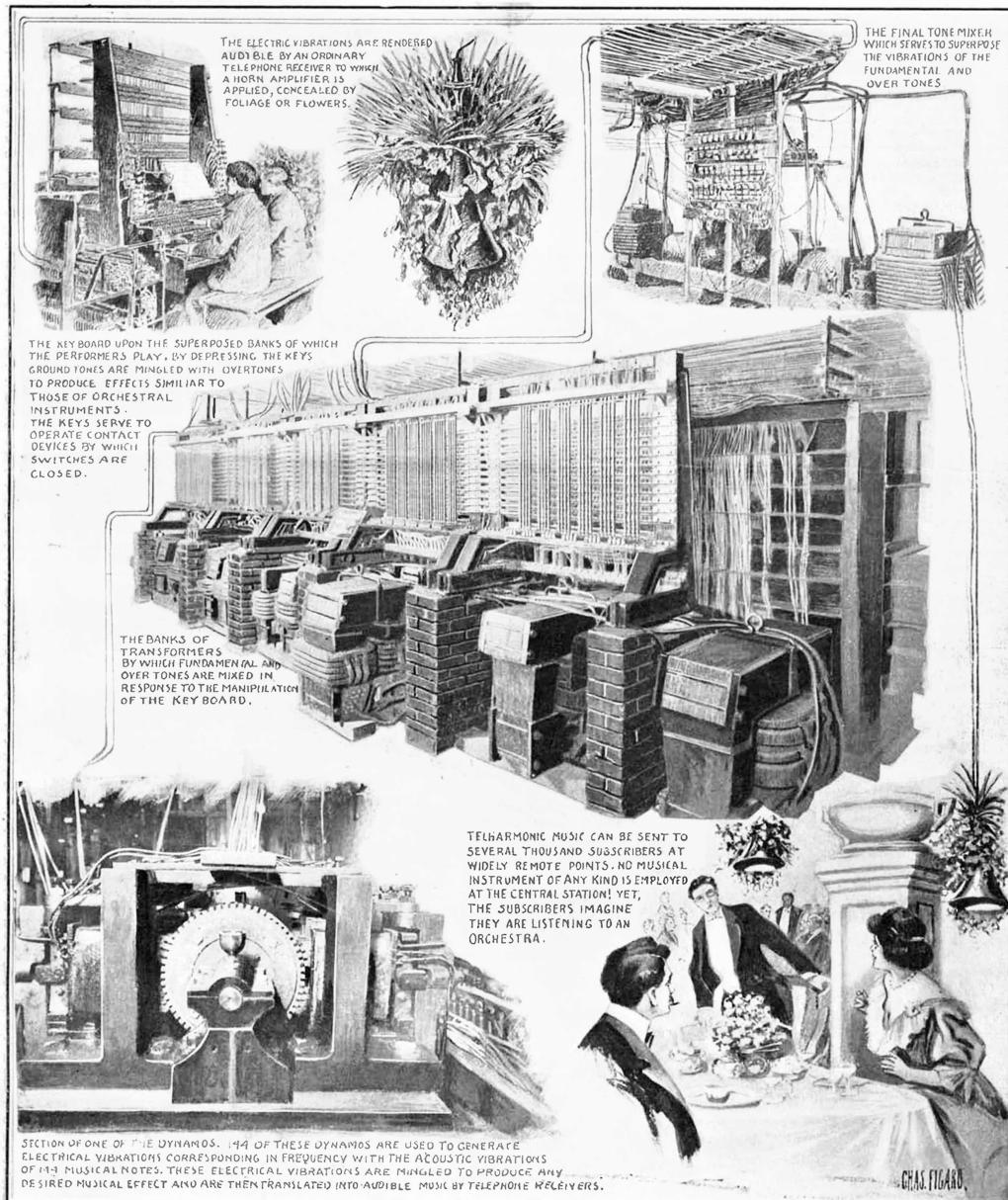
SCIENTIFIC AMERICAN

[Entered at the Post Office of New York, N. Y., as Second Class Matter. Copyright, 1907, by Munn & Co.]

Vol. XCVI.—No. 10
ESTABLISHED 1845.

NEW YORK, MARCH 9, 1907.

10 CENTS A COPY
\$8.00 A YEAR.



THE TELHARMONIUM—AN APPARATUS FOR THE ELECTRICAL GENERATION AND TRANSMISSION OF MUSIC.—[See page 210.]

Figure 26 The Telharmonium. Tonewheel ("dynamo") shown at bottom left.^{©18}

The first significant electronic music synthesizer in history, Thaddeus Cahill's massive **Telharmonium**, relied on summing tonewheels and was thus a kind of additive synthesizer. The idea behind the Telharmonium was that a single performer could produce a song electronically, which then could be broadcast over telephone lines to many remote sites at once. Figure 26 shows the Telharmonium in all its glory, including a tonewheel diagram at bottom left.

Tonewheels later formed the sound-generation mechanism (along with the famous **Leslie** rotating speaker) of the **Hammond Organ**: and it too worked using additive synthesis. The Hammond Organ sported nine **drawbars** which specified the amplitudes of nine specific partials ranging in frequency from one octave below the fundamental to three octaves above. These drawbars were linked to tonewheels which produced the final sound.

Most later attempts in additive synthesis were in the digital realm. In 1974 the Rocky Mount Instruments (or RMI) **Harmonic Synthesizer** was probably the first electronic music synthesizer to do additive synthesis using digital oscillators. The **Bell Labs Digital Synthesizer**, a highly influential experimental digital synthesizer, was also entirely additive. Fairlight's **Qasar M8** generated samples by manipulating partials, and then used an IFFT to produce the final sound. Finally (and importantly) the commercially successful, but quite expensive, New England Digital **Synclavier II** sported additive synthesis along with other synthesis modes (sampling, FM), putting additive synthesis within reach of professional music studios.

During the 1980s and 1990s, Kawai was the primary manufacturer to produce additive synthesizers. Kawai's **K3**, **K5**, and later its much improved **K5000** series brought additive synthesis to individual musicians. Since the 1990s, the method has not shown up much in commercial hardware synthesizers, but it features prominently in a number of software synthesizers, including AIR Music Technology's **Loom**, Native Instruments Inc.'s **Razor**, Image-Line Software's **Harmor** and **Harmless**, and Camel Audio (now Apple)'s **Alchemy**.

4.2 Approach

Each timestep an additive synthesizer produces and modifies an array of partials, and once the array is sufficiently modified, the synthesizer gives it to its sound generation facility, which uses the array to produce a single sample. The sound generation facility typically produces this sample by handing each partial in the array to a corresponding sine-wave generator, which runs it through a sine wave function to produce a single sample value for that partial. Then all the sample values are added up to produce the final sample.



Figure 27 Hammond B3 Organ.^{©19}



Figure 28 Kawai K5000s.^{©20}

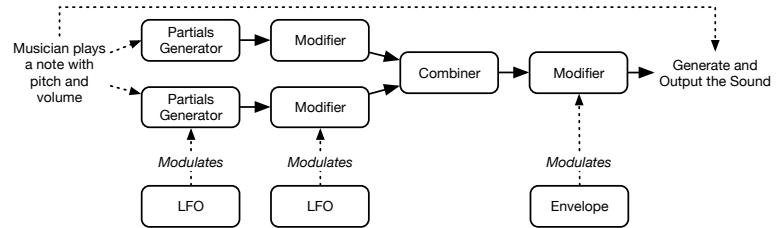


Figure 29 One possible additive pipeline (of many). Arrays of partials are passed from module to module.

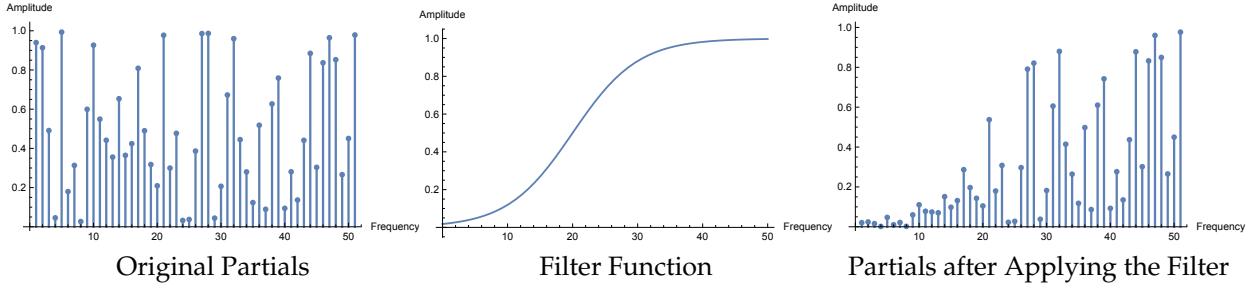


Figure 30 Effect of a filter on the amplitudes of partials in an additive synthesizer.

Figure 29 shows one possible pipeline for an additive synthesizer. This isn't the only possibility by far, but it serves as an example with many of the common elements:

- **Partials Generators** These are sources for arrays of partials. They could be anything. For example, a generator might output one of several preset arrays of partials designed to produce specific tones. A partials generator could also change the partials arrays it emits over time. For example, a partials generator could emit one of 128 different arrays of partials, and the particular array being emitted is specified by a parameter. This has a close relationship with a technique discussed later called **wavetable synthesis**.
- **Partials Modifiers** These take arrays of partials and modify them, emitting the result. A simple modifier might just **amplify** the partials by multiplying all of their amplitudes by a constant. Or perhaps a modifier might change the frequencies of certain partials.

Another common modifier is a **filter**, which shapes partials by multiplying their amplitudes against a filter function as shown in Figure 30. There are many possible filter function shapes, though certain ones are very common. For example, a **low pass filter** cuts off high frequencies after some point, whereas a **high pass filter** cuts off low frequencies. The filter in Figure 30 is an example of a high pass filter. There is also the **band pass filter**, which cuts off all frequencies except those in a certain range, and the **notch filter**, which does the opposite. Another common filter in additive synthesis is the **formant filter**, where the amplitudes of partials are shaped to simulate the effect of the human vocal tract (see Section 8.11).

In other forms of synthesis which work in the time domain rather than the frequency domain, filters can be tricky to implement: indeed all of Section 8 discusses these kinds of filters. But with an additive synthesizer we are fortunate, because in the frequency domain a filter is little more than a function which manipulates the arrays of partials.²⁰

- **Partials Combiners** These take two or more arrays of partials and merge them somehow to form a single array. If the partials are harmonics and both arrays contain the same

²⁰If you'd like a basic filter function for an additive synthesizer, try using the two-pole Butterworth filter amplitude response equations in Section 8.9. For example, if you have a desired cutoff frequency $\omega_0 > 0$ (in radians) and resonance $Q > 0$, then for each partial, given its frequency ω (in radians again), multiply its amplitude against $1/\sqrt{(1 - \omega^2/\omega_0^2)^2 + (\omega/(\omega_0 Q))^2}$ to get a basic low-pass filter. To convert a frequency from Hz to radians, just multiply by 2π . Also, resonance normally doesn't drop below $Q = \sqrt{1/2}$, which is generally considered the minimum "no resonance" position.

frequencies, then this could be as simple as adding together the amplitudes of the same-frequency harmonics from both arrays: the additive version of **mixing**. If the partials have arbitrary frequencies, and you need to produce a new array that is the same size as each of the previous arrays, then you'd have to use some clever approach to cut out partials: for example, you might throw all the partials together and then delete the highest frequency ones.

- **Modulation** All along the way, the parameters of the partials generators, modifiers, and combiners can be changed in real time via automated or musician-driven **modulation** procedures. A modulation signal typically varies from -1 to 1, or perhaps from 0 to 1. Modulators can be used not only to change the parameters of the aforementioned modules, etc., but also the parameters of *other modulators*. The two common kinds of automated modulators are:
 - **Low Frequency Oscillators** or **LFOs** simply cause the signal to go up and down at a certain rate specified by the musician.
 - **Envelopes** vary the signal over time after a key has been pressed. For example, in an **Attack-Decay-Sustain-Release** (or **ADSR**) **envelope**, when you press a note, the envelope begins sweeping from 0 to some **attack level** over the course of an **attack time**. When it reaches the attack level, it then starts sweeping back down to some **sustain level** over the course of a **decay time**. When it reaches the sustain level, it stays there until you release the key, at which point it starts sweeping back to zero over the course of a **release time**. The musician specifies these values.

Modulation is absolutely critical to making realistic sounds. Consider for a moment that when someone plays an instrument such as trumpet, we often first hear a loud and brash blare for a moment, which then fades to a mellow tone. There are two things that are happening here. First, the trumpet is starting loudly, then quickly dropping in volume. Second, the trumpet is starting with lots of high-frequency harmonics, giving it a brash and buzzy sound, and then quickly reduces to just low-frequency harmonics, resulting in a mellowing of tone. If we wished to simulate this, we'd use a modulation procedure which, when a note was played, made the sound louder and perhaps opened a low-pass filter to allow higher harmonics through, and then soon thereafter quieted the sound and closed much of the filter (cutting out the higher harmonics). This kind of modulation procedure calls for one or more envelopes. Similarly if we wished to add **tremolo** (rapidly moving the volume up and down) or **vibrato** (rapidly moving the pitch up and down) or another oscillating effect, we could use an LFO. Other modulation mechanisms include **Arpeggiators** and **Sequencers**. We'll cover all of these in more detail in Section 5.

4.3 Implementation

An additive synthesizer can be implemented straightforwardly as a set of modules which offer arrays of partials or individual modulation values to one another. Every so often the code would update all of the modules in order, allowing them to extract the latest information out of other modules so as to revise their own offered partials or modulation. A final module, *Out*, would extract and hold the latest partials. Every time tick (more rapidly than the modules update) the facility would grab the latest partials from *Out* and use them to update a sample, one sample per tick. Here's a basic monophonic additive synthesizer top-level architecture:

Algorithm 6 *Simple Monophonic Additive Synthesizer Architecture*

```

1:  $M \leftarrow \langle M_1, \dots, M_m \rangle$  modules
2: tick  $\leftarrow 0$ 
3: counter  $\leftarrow 0$ 
4:  $\delta \leftarrow 0$ 
5:  $\alpha \leftarrow$  interpolation factor
6: ticksPerUpdate  $\leftarrow$  number of ticks to wait between updates  $\triangleright$  ticksPerUpdate = 32 works well

7: procedure Tick
8:   if Note Released then
9:     for  $i$  from 1 to  $m$  do
10:    Released( $M_i$ , pitch)
11:   if Note Pressed then
12:     for  $i$  from 1 to  $m$  do
13:       Pressed( $M_i$ , pitch, volume)
14:   tick  $\leftarrow$  tick +1
15:   counter  $\leftarrow$  counter +1
16:    $\delta \leftarrow (1 - \alpha) \times \delta + \alpha$ 
17:   if counter  $\geq$  ticksPerUpdate then
18:     counter  $\leftarrow 0$ 
19:      $\delta \leftarrow 0$ 
20:     for  $i$  from 1 to  $m$  do
21:       Update( $M_i$ , tick)
22:   return OutputSample(tick,  $\delta$ )

```

Note that a new note may be pressed before the previous note is released: this is known as playing **legato** on a monophonic synthesizer. Some modules might respond specially to this situation. For example, partials generators might gradually slide in pitch from the old note to the new note, a process called **portamento**.

Interpolating the Array of Partials What's the point of δ and α ? The call to OutputSample(...) in Algorithm 6 is called every tick: but the partials are only updated (via Update(...)) every ticksPerUpdate ticks. If ticksPerUpdate > 1 then we will have a problem: even relatively small changes in the amplitude and frequency of the partials can appear as abrupt changes in the underlying sound waves, creating clicks.

The simplest way to fix this is to do partial interpolation. Let A^{t-1} be the amplitudes of the previous partials and A^t be the amplitudes of the current partials. Similarly, let F^{t-1} and F^t be their frequencies. For each partial i , we could define A_i and F_i to be the amplitude and frequency, respectively, used to generate the next sample via $A_i \leftarrow (1 - \delta) \times A_i^{t-1} + \delta \times A_i^t$, and $F_i \leftarrow (1 - \delta) \times F_i^{t-1} + \delta \times F_i^t$. Here δ is 0 when we receive a brand new set of partials, and gradually increases to 1 immediately prior to when we receive the next new set.

In Algorithm 6 we're passing in a δ to OutputSample(...) which can serve exactly this purpose. Note that it's being increased exponentially rather than linearly: I've found an exponential curve to be much more effective at eliminating clicks. But you will need to set α such that, by the time ticksPerUpdate ticks have expired, δ is within, oh, about 0.97 or so.

Warning Imagine that $A_i^t = 0$. Then as interpolation pushes A_i towards A_i^t , you could find A_i mired in the **denormalized**²¹ range, and math with denormal numbers is can be *extremely* slow for many programming languages. You need to detect that you're getting close to the denormals and just set A_i directly to 0. For example, if $A_i^t < s$ and $\delta < s$ for a value of s somewhat above the denormals, then $A_i \leftarrow 0$.

Generating a Sound from an Array of Partials At the end of the day, we must take the final array of partials and produce one sample for our sound. Let us define a partial as a tuple $\langle i, f, a, p \rangle$:

- Each partial has a unique ID $i \in 0...N$. This indicates the sine-wave generator responsible for outputting that partial. If a partial's position in the array never changes, this isn't really necessary: you could just use the array position as the ID. However it might be useful to rearrange the partials in the array (perhaps because you've changed their various frequencies in some module, and then re-sorted the partials by frequency). Keeping track of which partial was originally for which generator is helpful because if a generator suddenly switched to a different partial with a different phase or frequency or amplitude, you might hear an audible pop as the generator's sine wave abruptly changed.
- The frequency f of the partial is relative to the *base frequency* of the note being created: for example, if the note being played is an A 440, and $f = 2.0$, then the partial's frequency is $440 \times 2.0 = 880$.
- To keep things simple, the amplitude $a \geq 0$ of the partial is never negative. If you needed a "negative" amplitude, as in a Triangle wave, you could achieve this by just shifting the phase by π .
- The phase p of the partial could be any value, though for our own sanity, we might restrict it to $0 \leq p \leq 2\pi$.

The sound generation facility maintains an array of sine-wave generators $G_1...G_N$. Each generator has a current **instantaneous phase** x_i . Let's say that the interval between successive timesteps is Δt seconds: for example, 1/44100 seconds for 44.1KHz. Every timestep each generator G_i finds its associated partial $\langle i, f, a, p \rangle$ of ID i . It then increases x_i to advance it the amount that it had changed due to the partial's frequency:

$$x_i^{(t)} \leftarrow x_i^{(t-1)} + f_i \Delta t$$

Let's assume that the **period** of our wave corresponded to the interval $x_i = 0...1$. Since our wave is periodic, it's helpful to always keep x_i in the 0...1 range. So when it gets bigger than 1, we just subtract 1 to wrap it back into that range. One big reason why this is a good idea is that high values of x_i will start having resolution issues given the computer's floating-point numerical accuracy. So we could say:

$$x_i^{(t)} \leftarrow x_i^{(t-1)} + f_i \Delta t \mod 1 \tag{1}$$

²¹ Denormalized numbers are a quirk of the IEEE 754 floating point spec. They are a set of numbers greater than zero but less than the lowest positive exponent. For doubles, that means they're roughly $< 2^{-308}$. Math with them typically isn't handled in hardware: it has to be done in software, if your language can't automatically set denormals to 0. As a result it's hundreds, sometimes thousands, of times slower. You don't want to mess with that.

For our purposes, $\text{mod } 1$ is the same thing as saying “keep subtracting 1 until the value is in the range $0\dots 1$, excluding 1.” In Java, $x \bmod 1$ (for positive x , which is our case) is easily implemented as $x = x - (\text{int}) x$; Once this is done for each x_i , we just adjust all the sine waves by their phases, multiply them by their amplitudes, and add ‘em up. Keep in mind that the period of a sine wave goes $0\dots 2\pi$, so we need to adjust our period range accordingly. So the final sample is defined as:

$$\sum_i \sin(2\pi x_i^{(t)} + p_i) \times a_i$$

We might multiply the final result against a gain (a volume), but that’s basically all there is to it.

Sine Approximation The big cost in additive synthesis is the generation and summation of sine waves. Don’t use the built-in `sin` function, it’s costly. Approximate it with a fast lookup table:

Algorithm 7 Sine Table Initialization

```

1: Global  $S_0\dots S_{2^n-1} \leftarrow$  array of  $2^n$  numbers           ▷ I use  $2^n = 65536$ , but your mileage may vary
2: for  $i$  from 0 to  $2^n - 1$  do
3:    $S_i \leftarrow \sin(2\pi i / 2^n)$ 
```

Algorithm 8 Sine Approximation

```

1:  $x \leftarrow$  argument
2:  $i \leftarrow \left\lfloor x \times \frac{2^n}{2\pi} \right\rfloor \bmod 2^n$            ▷  $\bmod 2^n$  can be done with just bitwise-and  $(2^n - 1)$ 
3: return  $S_i$ 
```

You can make this much more accurate still by interpolating with the **Catmull-Rom** cubic spline (Equation 6, page 120). To map to that equation, let $\alpha = x \times \frac{2^n}{2\pi} - \left\lfloor x \times \frac{2^n}{2\pi} \right\rfloor$. Then let $f(x_1) = S_{(i-1 \bmod 2^n)}$, $f(x_2) = S_{(i \bmod 2^n)}$, $f(x_3) = S_{(i+1 \bmod 2^n)}$, and $f(x_4) = S_{(i+2 \bmod 2^n)}$. Slightly slower than direct table lookup, but still far faster than the built-in `sin` function.

Buffering and Latency This is an important implementation detail you need to be aware of. In most operating systems you will output sound by dumping data into a **buffer**. You can dump it in a sample at a time, or (more efficiently) put in an array of samples all at once. The operating system *must not fully drain the buffer* or it will start emitting garbage (usually zeros) as sound because it has nothing else available. You have to keep this buffer full enough that this does not happen.

The problem is that the operating system won’t drain the buffer a byte at a time: instead it will take chunks out of the buffer in fits and starts. This means you always have to keep the buffer filled to more than the largest possible chunk. Different operating systems and libraries have different chunk sizes. For example, Java on OS X (what I’m familiar with) has an archaic audio facility which requires a buffer size of about 1.5K bytes. Low-latency versions of Linux can reduce this to 512 bytes or less.

You’d like as small a buffer as possible because keeping a buffer full of sound means that you have that much **latency** in the audio. That is, if you have to keep a 1.5K buffer full, that’s going to result in a 17ms audio delay. That’s a lot!

Timing How do you make sure that the `Tick(...)` method is called regularly and consistently? There are various approaches: you could use timing code provided by the operating system, or poll a `getTime(...)` method and call `Tick(...)` when appropriate. But there's an easier way: just rely on the audio output buffer. That is, if the buffer isn't full, fill it, and each time you fill it with a sample, you do so by calling `Tick(...)` once. As discussed before, the buffer gets drained in fits and starts, and so your filling will be fitfull as well: but that doesn't matter: all that matters is that all of your time-sensitive code is in sync with the audio output. So base it directly on the output itself! That is, I'd call the following over and over again in a tight loop:

Algorithm 9 *Buffer Output*

```

1:  $A \leftarrow$  array of samples                                ▷ Large enough to keep the buffer happy
2: for  $i$  from 0 to length of  $A - 1$  do
3:    $A_i \leftarrow \text{Tick}()$ 
4:  $\text{AddToBuffer}(A)$                                      ▷ Presumably this is blocking

```

4.4 Architecture Examples

Let's consider two actual architectures, both of which fall neatly in the architectural implementation discussed before.

Kawai K5 Kawai has probably produced more additive synthesizers than any other company. This synthesizer came out in 1987 in both keyboard and rackmount (K5m) versions. The **Kawai K5** had exactly 126 of partials in its array, organized as **harmonics**: you could not change their frequencies. Furthermore, the phase of each harmonic was fixed to zero. This greatly simplified the options available: all you could do was change the amplitudes of the partials over time: but that alone still involved a great many parameters.

Nonetheless the pipeline for a K5 is quite simple:

- Determine the current pitch of the note (this can be modulated with an LFO or envelope).
- Build an array of 63 harmonics. You can set the amplitude of each of the harmonics separately. You can also modulate the amplitudes of harmonics, either by assigning each of the harmonics to one of four envelopes or to an LFO. The envelopes are the important part here: they allow different harmonics to rise and fall over time, changing the sound timbre considerably.
- Run the harmonics through some kind of **filter**. The filter has its own envelope and can be modulated via a LFO.
- Run the harmonics through an amplifier. This amplifies all the harmonics as a group, much as a sound is amplified (as opposed to earlier in the pipeline, when each harmonic could have its amplitude changed independently). The amplifier has its own envelope and can be modulated via a LFO.



Figure 31 Kawai K5m, a rackmount version of the K5.

- Run the harmonics through a **formant filter**. This filter can be used to adjust the harmonics to simulate the formant properties of the human vocal tract.
- Hand the harmonics to the final output to be summed and emitted.
- This pipeline happens *twice*, for two independent sets of 63 harmonics each. This can be done in parallel to make two independent voices per note, or one set can be assigned to harmonics 1...63, while the other set is assigned to harmonics 65...127 to create a richer sound with many higher-frequency harmonics.²²

The challenge here is that even with this simple architecture, there were 751 parameters, as *every harmonic* had its own amplitude and modulation options. The amplifier, filter, and pitch all had their own 6- or 7-stage envelopes, as well as the four envelopes that the harmonics could be assigned to: and this was for each of the two sets of harmonics. It was not easy to program the Kawai K5.

The K5 was also not a good sounding synthesizer.²³ But ten years later, Kawai tried again with the K5000 series (Figure 31) and produced a much better design. The architecture was similar in many respects, but with one critical difference: every harmonic now had its own independent envelope. This allowed for much richer and more complex sounds (but even more parameters!)

Flow Flow is a fully modular, polyphonic, additive synthesizer of my own design. In a Flow patch, you lay out modules in the form of partials generators, partials modifiers, modulation sources, and modulation modifiers, set their parameters, and then wire them up like you would connect modules in a traditional modular synthesizer. The difference, however, is that while time-

based audio signals were transferred along the cables of a traditional modular synth, the virtual cables in Flow transfer arrays of up to 256 partials. Other cables transfer modulation information.

Flow patches can contain any number of modules, plus one dedicated module called *Out* which gathers partials from whatever cable is plugged into it and uses those partials to produce the final sound. Unusually, Flow patches can be loaded as modules themselves in *other patches*, compete with their own special modulation and partials inputs and outputs.

Flow's pipeline is straightforward: each timestep, all of the modules are pulsed once, left to right. When a module is pulsed, it gathers modulation and partials information from its upstream modules to produce modulation and partials information it wishes to output to later downstream modules.²⁴ Ultimately the *Out* module is pulsed, and it sends its partials information to be outputted for the next *n* samples (currently 32).

Flow has many available modules, and a patch can potentially have a large number of them. This obviously can result in complex patches with very large numbers of parameters: but in fact

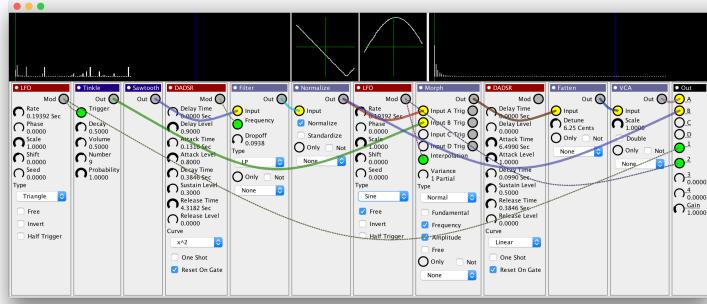


Figure 32 Flow.

²²There was no harmonic 64. I don't know why.

²³Believe me. I owned one and upgraded it considerably. Figure 31 is a picture of my K5m with a brand-new screen.

²⁴This doesn't imply that data in Flow can only go left-to-right: Flow patches can contain cycles in their connections.

the large majority of patches only employ small tweaks of standard modules. Rather than tediously manipulate the individual partials in a sound one by one (though you can do that), Flow is instead geared more towards pushing arrays through various manipulation and filter modules as a whole.

Flow fixes the number of partials, often to 256. It also disregards phase, and a partial only has frequency, amplitude, and an ID. Flow can manipulate the frequency and amplitude of partials in a wide variety of ways and can combine and **morph**²⁵ partials from multiple sources.

Related tools include AIR Music Technology's **Loom**, Native Instruments Inc.'s **Razor**, and Image-Line Software's **Harmor**. These aren't fully modular in the sense that Flow is, but they can organize groups of additive modules together in a linear pipeline. Such tools also may have variable numbers of partials, and may include phase.

²⁵Morphing works like this. For each pair of partials, one from each incoming set, produce a new resulting partial which is the weighted average of the two both in terms of frequency and amplitude. You'd then modulate the weight.

5 Modulation

By themselves, the audio pipeline modules will produce a constant tone: this might work okay for an organ sound, but otherwise it's both boring and atypical of sounds generated by real musical instruments or physical processes. Real sounds change over time, both rapidly and slowly. To make anything which sounds realistic, or at least interesting, requires the inclusion of mechanisms which can change pipeline parameters over *time*. These are **modulation sources**.

Modulation signals come from two basic sources:

- The musician himself through various interface options: buttons, knobs, sliders, and so on. Some of these are general-purpose and can be assigned as the musician prefers. These might include the **modulation wheel**, **pitch bend wheel**, so-called **expression pedals**, and from the keyboard's **velocity**, **release velocity**, and **aftertouch**, among others. For definitions and more information on these modulation interface options, see Section 12.
- **Automated Modulation** procedures which change parameters automatically as time passes. This is the bulk of this Section. Automated modulation procedures often have their own parameters, and these parameters could be themselves modulated by *other* modulation sources. Thus you might see chains or even cycles of modulation signals.

How a modulation signal modulates a parameter depends on the range of the parameter. Some parameters, such as volume, are **unipolar**, meaning that their range is $0\dots N$ (we could just think of this as $0\dots 1$). Other parameters, such as pitch bend, might be **bipolar**, meaning that their range is $-M\dots +M$ (perhaps simplified to $-1/2\dots +1/2$ or $-1\dots +1$). It's trivial to map a unipolar to a bipolar signal or vice versa, of course, and synthesizers will often do this.

Another issue is the **resolution** of the parameter. Some parameters are real-valued with a high resolution; but others are very coarse-grained. And even if a parameter is high-resolution, some modulation signals it could receive—notably those provided over MIDI (Section 12.2)—can be very coarse, often just 7 bits ($0\dots 127$). In this situation, gradually changing the modulation signal will create a **zipper effect** as the parameter clicks from one discretized value to the next.

5.1 Low Frequency Oscillators

This is our first automated modulation option. A **Low Frequency Oscillator** or **LFO** is exactly what's written on the tin: a repeating function at a low frequency whose output is used to slowly modulate some parameter. By *slowly* we mean that an LFO is usually slower than **audio rate**, so it's less than 50Hz or so.²⁶ It's not unusual for an LFO to be 1Hz or less.

An LFO can be used to modulate lots of things. For example, if it were used to shift the pitch of an oscillator, it'd cause **vibrato**. Similarly, if it were used to adjust an oscillator's volume, it'd cause **tremolo**. An LFO is usually bipolar, perhaps ranging $-1\dots 1$.

LFOs often come in a number of classic shapes, including **sine**, **square**, **triangle**, and both **sawtooth** and **ramp**, as shown in Figure 33. LFOs have a number of parameters, including at least the *rate* or *frequency*, and an *amplitude* or *amount*. An LFO's rate could also be defined by a clock in the synthesizer, or synced to the rate of incoming MIDI Clock pulses (see “Clock Messages” in

²⁶One desirable property of an LFO is the ability to go high into audio rate, so as to effect a form of **frequency modulation** (or **FM**) on audio signals. We'll cover FM in Section 9.

Section 12.2.2). The period of an LFO in a voice is normally reset when the musician plays a new note, unless the LFO has been set **free**.²⁷

In a monophonic synthesizer a new note might be pressed before the last one was released (known as playing **legato**). Some LFOs might prefer to *not* reset in this situation, because the new note may be perceived by the listener essentially as a continuation of the previous one.

Once you've got a master clock providing ticks (see Section 4.3), implementing an LFO is pretty straightforward: you just have to map the ticks into the current cycle position (between 0 and 1). You could do this with division, or you could do it by incrementing and then truncating back to between 0 and 1. Each has its own numerical issues. I've chosen the latter below.

Algorithm 10 Simple Low Frequency Oscillator

```

1:  $r \leftarrow \text{rate}$                                 ▷ In cycles per tick
2:  $\text{type} \leftarrow \text{LFO type}$ 
3:  $\text{free} \leftarrow \text{is the LFO free-running?}$ 
4:  $\text{legato} \leftarrow \text{did a legato event occur (and we care about legato)?}$ 

5: global  $s \leftarrow 0$                                 ▷ Current state (0...1)

6: procedure Note Pressed
7:   if not free and not legato then
8:      $s \leftarrow 0$ 

9: procedure Update
10:   $s \leftarrow s + r$ 
11:  if  $s \geq 1$  then
12:     $s \leftarrow s \bmod 1$                                 ▷ Easily done in Java as  $s = s - (\text{int}) s$ 
13:  if type is Square then
14:    return  $\begin{cases} -1 & s < 1/2 \\ 1 & \text{otherwise} \end{cases}$ 
15:  else if type is Triangle then
16:    return  $\begin{cases} s \times 4 - 1 & s < 1/2 \\ 3 - 4 \times s & \text{otherwise} \end{cases}$ 
17:  else if type is Sawtooth then
18:    return  $(1 - s) \times 2 - 1$ 
19:  else if type is Ramp then
20:    return  $s \times 2 - 1$ 
21:  else                                              ▷ Type is Sine. See Section 4.3 for fast Sine lookup
22:    return  $\sin(s \times 2\pi) \times 2 - 1$ 

```

Random LFO Oscillators LFOs often also have a **random** oscillator. For example, every period it might pick a random new target value between -1 and 1 , and then over the course of the period it would gradually interpolate from its current value to the new value, as shown in Figure 33. We might adjust the **variance** in the choice of new random target locations. I'd implement it like this:

²⁷Unlike audio-rate oscillators, phase *matters* for an LFO, since we can certainly detect out-of-phase LFOs used to modulate various things.

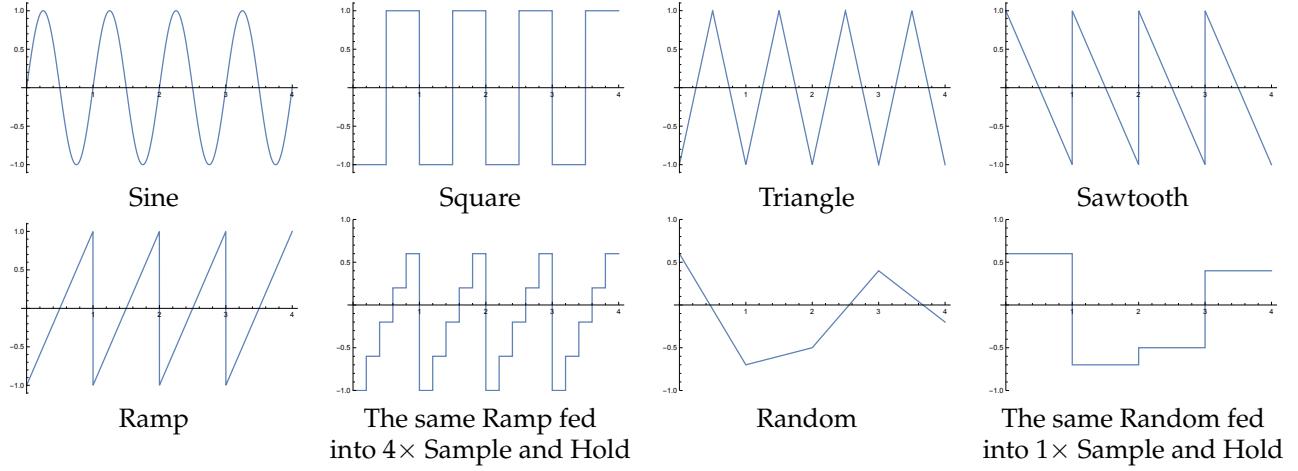


Figure 33 Various Low Frequency Oscillator wave functions.

Algorithm 11 Random Low Frequency Oscillator

```

1: r  $\leftarrow$  rate                                ▷ In cycles per tick
2: var  $\leftarrow$  variance (0...1)                ▷ How randomly we pick new targets.
3: free  $\leftarrow$  is the LFO free-running?
4: legato  $\leftarrow$  did a legato event occur (and we care about legato)?

5: global s  $\leftarrow$  0                                ▷ Current state (0...1)
6: global target  $\leftarrow$  0
7: global previous  $\leftarrow$  0

8: procedure Note Pressed
9:   if not free and not legato then
10:    s  $\leftarrow$  0
11:    ChooseNewTarget( )

12: procedure Update
13:   s  $\leftarrow$  s + r
14:   if s  $\geq$  1 then
15:     s  $\leftarrow$  s mod 1
16:     ChooseNewTarget()                            ▷ Easily done in Java as s = s - (int) s
17:   return (1 - s)  $\times$  previous + s  $\times$  target

18: procedure Choose New Target
19:   previous  $\leftarrow$  target
20:   repeat
21:      $\delta \leftarrow$  random value from -2...2 inclusive
22:     target  $\leftarrow$  previous +  $\delta \times$  var
23:   until  $-1 \leq$  target  $\leq 1$ 
```

Note that we're picking delta values from $-2\dots 2$. This is so that, at maximum variance, if we're currently at -1 , we could shift to any new target value clear up to $+1$ (and similarly vice versa). With smaller and smaller variance, we'll pick new target values closer and closer to our current value.

Sample and Hold Many synthesizers also have a special function called **sample and hold**, or **S&H**, which takes a modulation input and produces a discretized modulation output. Every period it *samples* the current value of the input, and during the course of the period it outputs only that value, ignoring all later inputs. Like an LFO, Sample and Hold may respond to **free running** and to **legato**. Here is one simple implementation:

Algorithm 12 *Sample and Hold*

```

1: x  $\leftarrow$  current input
2: r  $\leftarrow$  rate                                      $\triangleright$  In cycles per tick
3: free  $\leftarrow$  is the LFO free-running?
4: legato  $\leftarrow$  did a legato event occur (and we care about legato)?

5: global s  $\leftarrow$  0                                      $\triangleright$  Current state (0...1)
6: global target  $\leftarrow$  0

7: procedure Note Pressed
8:   if not free and not legato then
9:     s  $\leftarrow$  0
10:    target  $\leftarrow$  x

11: procedure Update
12:   s  $\leftarrow$  s  $+ r$ 
13:   if s  $\geq 1 then
14:     s  $\leftarrow$  s mod 1                                $\triangleright$  Easily done in Java as s = s - (int) s
15:     target  $\leftarrow$  x
16:   return target$ 
```

Sample and hold can be applied to any modulation source to produce a “discretized” version of the modulation,²⁸ but it's particularly common to feed apply it to an LFO. Discretizing sawtooth, ramp, triangle, and random LFO waves are common: in fact, sample and hold is so often applied to random LFO waves that it's typically a full-fledged wave option in many LFO generators. The coarseness of discretization would depend on the sample and hold rate. Figure 33 shows examples of sample and hold, at two different frequencies, applied to ramp and random waves.

5.2 Envelopes

An **envelope** is a time-varying function which, when triggered, starts at some initial value and then follows the function until it is terminated. Very commonly the trigger is when the musician plays a note. Software or hardware which produces an envelope function is called an **envelope generator**.

²⁸Yes, if the sample and hold is a high enough rate, it'll sound just like the **zipper effect** discussed earlier (page 43).

ADSR By far the most common envelope used in synthesizers is the **Attack-Decay-Sustain-Release** envelope, or **ADSR**. When triggered (again, usually due pressing a key), the envelope starts rising from a start value — usually 0 — up to an **attack level** (sometimes fixed to 1.0) over the course of an **attack time** interval. Once the interval has been exhausted, the envelope then begins to drop to a **sustain level** over the course of a **decay time**. At that point the envelope holds at the sustain level until a different trigger occurs (usually due to the musician releasing the key). Then the envelope begins to decay to zero over the course of a **release time** interval.

ADSR envelopes are popular because many musical instruments can be very roughly modeled using them. Many instruments start with a loud and brash splash, then decay rapidly to a quieter and more mellow sustained period, then finally trail off. This can be modeled with two ADSR envelopes, one attached to the overall volume of the note, and the other attached to the filter cutoff. Indeed many synthesizers have *dedicated* ADSR envelopes for these two purposes. Interestingly, these two envelopes don't need to be the same: it's perfectly plausible, for example, that the filter envelope reaches its attack maximum (its brightest sound) before the amplifier envelope reaches its maximum volume.

As shown in Figure 34, an ADSR envelope's rate of change could be linear, or it could be exponential (or something else!). Linear rates are easily implemented when the rise or drop intervals are defined in terms of **time**: but exponential rates are more easily implemented when the rises or drops are defined in terms of **rate** (or, crudely speaking, slope). The choice of rate versus interval or time depends on the synthesizer, and different manufacturers make different choices.²⁹

Implementing an exponential rate-based envelope is easier than it sounds. Imagine that the envelope was just starting its attack. We might set $v = 1.0$, and then every tick we decrease v as $v \leftarrow \alpha v$ where $0.0 < \alpha < 1.0$. Then we set the current the current level $y \leftarrow (1 - v) \times \text{attack level}$. Similarly if we were dropping from the attack level to the sustain level, we'd reset $v = 1.0$, and repeat this same trick, but define the level as $y \leftarrow v \times \text{attack level} + (1 - v) \times \text{sustain level}$. I'll leave the release case as an exercise.

There are several common variations on ADSR. Often we will see an additional **delay time** added prior to the onset of the ADSR, producing a **DADSR** envelope. Sometimes we will see *two* decay stages before settling into sustain. Sometimes we might see a **hold stage** added, such as **AHDSR**. In a hold stage the envelope maintains its current value over the interval of time. Other variations are simplifications: for example, in **AR** (or perhaps **AHR**), the decay and sustain stages are eliminated: the envelope simply attacks, then decays back to zero. Similarly, in a **one-shot envelope**, the sustain is eliminated. As is the case for an LFO, it is very common for ADSR envelopes to respond to **legato** by not resetting.

Simple ADSR Implementations Here's an example of a linear time-based ADSR, followed by an example of an exponential rate-based ADSR.

²⁹There is one big difference you may not have thought about though. Let's say that the release time and decay time are the same amounts (likewise release rate and decay rate). Imagine that the envelope has begun to decay, and then suddenly we let go of the note so it immediately starts releasing. In a time-based envelope, the release time will be consistent. But in a rate-based envelope, the amount of time to complete the release would depend on how high the value was when the note was released.

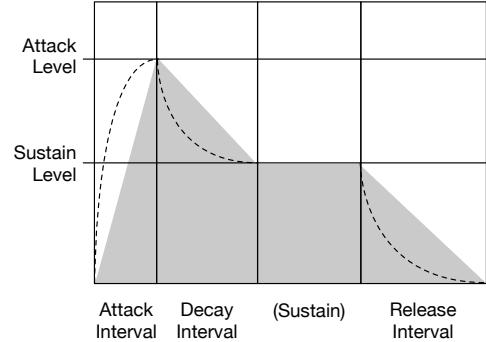


Figure 34 An ADSR envelope. Gray region shows a linear rate of change. Dotted lines show an exponential rate of change.

Algorithm 13 Simple Linear Time-based ADSR

```

1:  $r \leftarrow \text{rate}$                                 ▷ In envelope time units per tick
2:  $X \leftarrow \{X_0, \dots, X_4\}$                   ▷ Time when stage ends.  $X_2, X_4 = \infty$ 
3:  $Y \leftarrow \{Y_0, \dots, Y_4\}$                   ▷ Target parameter value of stage.  $Y_1 = Y_2$ , and  $Y_3 = Y_4 = 0$ 
4:  $\text{legato} \leftarrow \text{did a legato event occur (and we care about legato)?}$ 

5: global  $s \leftarrow 0$                                 ▷ Current state
6: global  $i \leftarrow 0$                                 ▷ Current stage. Attack=0, Decay=1, Sustain=2, Release=3, Done=4
7: global  $p \leftarrow 0$                                 ▷ Current parameter value.
8: global  $p' \leftarrow 0$                                ▷ Parameter value at start of the current stage.

9: procedure Note Pressed                                ▷ Reset everything to the beginning of Attack
10:   if not legato then
11:      $i \leftarrow 0$ 
12:      $s \leftarrow 0$ 
13:      $p' \leftarrow 0$ 
14:      $p \leftarrow 0$ 

15: procedure Note Released                                ▷ Reset everything to the beginning of Release
16:   if  $i < 3$  and not legato then
17:      $i \leftarrow 3$                                          ▷ Release stage
18:      $s \leftarrow 0$ 
19:      $p' \leftarrow p$                                        ▷ We start from where we currently are

20: procedure Update
21:    $s \leftarrow s + r$ 
22:   while  $s \geq X_i$  do                                ▷ Note that for Sustain (2) and Done (4) this will never be true
23:      $s \leftarrow s - X_i$                                 ▷ We don't reset to 0, but to our leftover time
24:      $p' \leftarrow p$ 
25:      $i \leftarrow i + 1$                                      ▷ Go to next stage
26:      $\gamma \leftarrow s/X_i$                                 ▷ This is assuming that  $X_i \neq 0$ , which ought to be the case
27:      $p \leftarrow (1 - \gamma) \times p' + \gamma \times X_i$     ▷ Compute interpolated value
28:   return  $p$ 

```

This envelope changes linearly with time. To change exponentially, a time-based ADSR would need a call to `pow()` to compute the exponential change, which is *very* costly. Instead, you could do a call to, say, x^4 , which works pretty well. To do this I'd just replace line 26 with these two lines:

$$\begin{aligned}\gamma &\leftarrow (s/X_i) \\ \gamma &\leftarrow \gamma \times \gamma \times \gamma \times \gamma\end{aligned}$$

To adjust the rate of attack/decay, just revise the number of times γ appears in the multiplication.

Because it is multiplying rather than adding, an exponential rate-based envelope will never reach its target, in Zeno's Paradox fashion. Thus we need an additional threshold variable, ϵ , which tells us that we're “close enough” to the target to assume that we have finished. This value should be pretty small, but not so small as to get into the denormals (See Footnote 21, page 37).

Algorithm 14 Simple Exponential Rate-based ADSR

```

1:  $X \leftarrow \{X_0, \dots, X_4\}$                                  $\triangleright$  Exponential rate for stage.  $X_2, X_4 = 1$ .
2:  $Y \leftarrow \{Y_0, \dots, Y_4\}$                                  $\triangleright$  Target parameter value of stage.  $Y_1 = Y_2$ , and  $Y_3 = Y_4 = 0$ .
3:  $\epsilon \leftarrow$  Threshold for switching to new stage. Low.     $\triangleright$  Should be large enough to avoid denormals!
4:  $\text{legato} \leftarrow$  did a legato event occur (and we care about legato)?

5: global  $s \leftarrow 1$                                           $\triangleright$  Current state
6: global  $i \leftarrow 0$                                           $\triangleright$  Current stage. Attack=0, Decay=1, Sustain=2, Release=3, Done = 4
7: global  $p \leftarrow 0$                                           $\triangleright$  Current parameter value.
8: global  $p' \leftarrow 0$                                           $\triangleright$  Parameter value at start of the current stage.

9: procedure Note Pressed                                      $\triangleright$  Reset everything to the beginning of Attack
10:   if not legato then
11:      $i \leftarrow 0$ 
12:      $s \leftarrow 1$ 
13:      $p \leftarrow 0$ 
14:      $p' \leftarrow p$ 

15: procedure Note Released                                     $\triangleright$  Reset everything to the beginning of Release
16:   if  $i < 3$  and not legato then
17:      $i \leftarrow 3$                                           $\triangleright$  Release stage
18:      $s \leftarrow 1$ 
19:      $p' \leftarrow p$ 

20: procedure Update
21:    $s \leftarrow s \times X_i$                                           $\triangleright$  Exponential dropoff
22:   if  $s \leq \epsilon$  then                                          $\triangleright$  Note that for Sustain (2) and Done (4) this will never be true
23:      $s \leftarrow 1$ 
24:      $p' \leftarrow p$ 
25:      $i \leftarrow i + 1$                                           $\triangleright$  Go to next stage
26:    $p \leftarrow s \times p' + (1 - s) \times Y_i$                           $\triangleright$  Compute interpolated value
27:   return  $p$ 

```

Multi-Stage Envelopes A multi-stage envelope has some N stages, each with its own target level and time interval or rate. Figure 35 shows an eight-stage envelope. Envelopes of this kind are often used to make slow, sweeping changes over long periods of time for **pads**.³⁰

Multi-stage envelopes often loop through a certain stage interval $M \dots N$ as long as the note is held down—a kind of pseudo-sustain—then finish out the remaining stages during release. If $M = N$, then the envelope effectively has a single sustain stage along the lines of an ADSR. It’s plausible for $M \dots N$ to encompass the entire envelope.

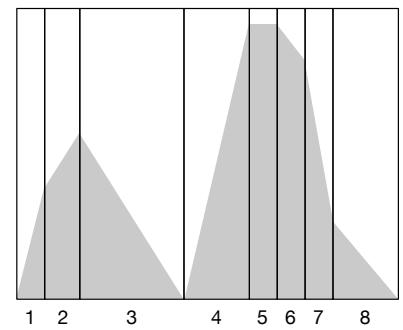


Figure 35 Eight-stage envelope.

³⁰A pad is a synthesizer patch designed to make long, ambient, background chords.

The envelopes discussed so far are generally unipolar. But there do exist **bipolar multi-stage envelopes**. There's nothing special about these: their values can simply range anywhere from $-1\dots+1$ instead of from $0\dots1$.

5.3 Step Sequencers and Drum Machines

A **sequencer** is a modulation device which triggers a series of events in a carefully timed fashion. Sequencers can get very elaborate: here we discuss the simplest form, the **step sequencer**.

A step sequencer maintains an array of parameter values. Every N seconds (or beats, or whatever), it advances to the next stage in its array and changes the modulation output to the value associated with that stage. The number of stages is usually a multiple of 8: it's quite common to use step sequencers with 16 stages, one per sixteenth note in a measure. A step sequencer almost always loops to the beginning when its stages have been expended. Figure 36 shows a possible step sequencer pattern (known, not surprisingly, as a **sequence**).

Because they use a fixed interval of time per stage, step sequencers are often used for melodic or rhythmic tasks beyond modulating parameters. For example, they very often trigger drum sounds, or are used to play notes much like a player piano roll. Step sequencers are also often **multi-track**, meaning that they maintain some M parallel arrays (the "tracks") rather than a single one, all of which are advanced in sync.

A **drum machine**, sometimes called a **drum computer**, usually combines a multi-track step sequencer with a **drum synthesizer** which plays a variety of drum sounds. Each drum sound is assigned to a different track in the sequencer. The track stores whether the drum is to be struck each step, along with the volume of the strike. Figure 37 shows the single most famous drum machine in history, the celebrated **Roland TR-808**. The top two thirds of the machine largely consists of the drum synthesizer, with parameter options (a column of knobs) for each drum. The bottom third largely consists of the multi-track sequencer, one track for each kind of drum sound. The musician would select a track with the knob at top left (surrounded by little yellow labels), and then using the buttons at bottom he would select the steps he wanted that drum to sound on.

Many synthesizers have built-in step sequencers to assist in creating complex rhythmic sounds. Step sequencers (and other kinds of sequencers) also common as external devices sending note or parameter information to synthesizers over MIDI. Let's consider one such example, **Gizmo** (Figure 38),³¹ to get an idea of common step sequencer features.

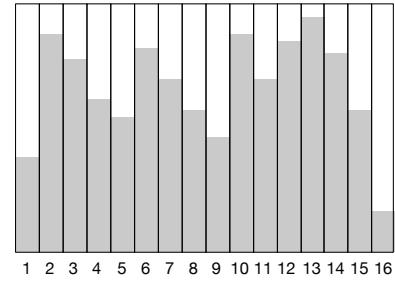


Figure 36 Step sequencer pattern.



Figure 37 Roland TR-808 drum machine.^{©21}



Figure 38 Gizmo.

³¹Why Gizmo? Because I made it of course.

Gizmo is an Arduino-based device for sending and receiving MIDI with many applications. Gizmo's step sequencer is laid out as a 2D array, where the X dimension is the stage or step number, and the Y dimension is the track. Gizmo supports up to 96 steps and up to 12 tracks, depending on how you want to allocate the Arduino's absurdly tiny memory. Each track is a sequence of either notes or parameters, one per cell in the track row. When a track stores notes, its cells contain both pitch and volume (per note), or can also specify **rests** (meaning "don't play anything here") or **ties** (meaning "continue playing the previous note").

After the musician has entered the relevant data into the step sequencer, it will loop through its stages, and at each stage it will emit MIDI information corresponding to all the notes and parameter settings at that stage. Gizmo's step sequencer can be pulsed by an external **clock**, or it can run on its own internal clock, in which case you'd need to specify its **tempo**.

Options Because step sequencers often deal with note or event data, they usually have a number of options. Here are a few of Gizmo's. First you can specify **swing**, that is, the degree of syncopation with which the notes are played. Second, you can specify the **length** of each note before the note is released. Tracks can have independent per-track volume (as opposed to per-note volume) and also have a built-in **fader** to amplify the volume as a whole. Tracks can be **muted** or **soloed**, and you can specify a pattern for automating muting in tracks. Finally, after some number of iterations, the sequencer can load an entirely different sequence and start on that one: this allows you to have multiple **sections** in a song rather than a simple repeating pattern the whole time.

Implementation A step sequencer can be very simple. Here is a very basic step sequencer for modulating parameters much as is done in an envelope or LFO. And like an LFO, a step sequencer may respond to **free** running and to **legato**.

Algorithm 15 Simple Parameter Step Sequencer

```

1:  $r \leftarrow \text{rate}$                                      ▷ In envelope time units per tick
2:  $s \leftarrow 0$                                          ▷ Current state
3:  $i \leftarrow 0$                                          ▷ Current stage
4:  $X \leftarrow \{X_0, \dots, X_N\}$                       ▷ Time when stage ends. Typically these are evenly spaced.
5:  $Y \leftarrow \{Y_0, \dots, Y_N\}$                          ▷ Parameter value of stage
6:  $\text{free} \leftarrow \text{is the step sequencer free-running?}$ 
7:  $\text{legato} \leftarrow \text{did a legato event occur (and we care about legato)?}$ 

8: procedure Note Pressed
9:   if not free and not legato then
10:     $i \leftarrow 0$ 
11:     $s \leftarrow 0$ 

12: procedure Update
13:    $s \leftarrow s + r$ 
14:   while  $s \geq X_i$  do
15:      $s \leftarrow s - X_i$                                 ▷ We don't reset to 0, but to our leftover time
16:      $i \leftarrow i + 1$                                   ▷ Go to next stage
17:     if  $i > N$  then
18:        $i \leftarrow 0$ 
19:   return  $Y_i$ 
```

5.4 Arpeggiators

An **arpeggiator** is a relative of the step sequencer whose purpose is to produce **arpeggios**. An arpeggio is a version of a chord where, instead of playing the entire chord all at once, its notes are played one by one in a pattern. Arpeggiators are only used to change notes: they're not used to modulate parameters, and so are not formally modulation devices. The classic arpeggiator intercepts notes played on the keyboard and sends arpeggios to the voices to play instead. As the musician adds or removes notes from the chord being played, the arpeggiator responds by adding or removing them from its arpeggiated note sequence.

Options An arpeggiator is usually outfitted with a **note latch** facility, which continues to play the arpeggio even after you have released the keys. Only on completely releasing all the keys and then playing a new chord does the arpeggio shift. You can usually also specify the number of **octaves** an arpeggiator plays: with two octaves specified and the chord C E G, the arpeggiator might arpeggiate C E G, then the C E G *above them*, before returning to the originals. Like a sequencer, an arpeggiator might also be subject to **swing**, **tempo**, **note length**, and **note velocity**.

Arpeggiation Patterns Arpeggiators usually offer a variety of arpeggio patterns. Here are some of **Gizmo's** built-in offerings (and they are typical):

- **Up** Repeatedly play the chord notes lowest-to-highest.
- **Down** Repeatedly play the chord notes highest-to-lowest.
- **Up-Down** Repeatedly play the chord notes lowest-to-highest, then back down highest-to-lowest. Do not play the lowest or highest notes twice in a row.
- **Assign** Repeatedly play the chord notes in the order in which they were struck by the musician when he played the chord.
- **Random** Repeatedly play the chord notes in random order.
- **Chord** Repeatedly play the chord as a whole.
- **Custom** Repeatedly play the chord in a pattern programmed by the musician (including ties and rests).

5.5 Gate/CV and Modular Synthesizers

Modular synthesizers rely on a standardized method for communication among the modules via their patch cables. Transferring audio is obvious: just send the audio signal over the wire. What about modulation signals?

Historically modular synthesis has treated modulation signals just like audio signals: they're voltages changing over time. There are two kinds of modulation that need to be encoded in this fashion. First there are **gate** signals: these are just on/off signals, and in Eurorack they are encoded as on=high (perhaps ≥ 8 volts) are off=low (0 volts).³² For example, when a musician presses a

³²Early Moog devices did something slightly different: to indicate "on", the signal was pulled to ground from whatever voltage it was.

note on a keyboard, it would send a gate-high signal to the synthesizer to indicate that some note is being pressed. Gates are also used as triggers from sequencers etc. to indicate new events.

Second, there are **control voltage** or **CV** signals. These are simply signals whose voltage varies continuously within some range. CV comes in both **unipolar** and **bipolar** ranges. For example, most envelopes are unipolar: an envelope's CV range would be 0–5 or 0–8 volts. On the other hand, an LFO is bipolar, and its wave would be outputted in the range ± 5 volts. Note that audio is also bipolar and in a similar range: thus audio and bipolar CV are essentially interchangeable.

In addition to a gate signal (indicating that a note was pressed), a keyboard would normally also output a unipolar CV signal to indicate *which* note was being played. This would be usually be encoded as 1 volt per octave: perhaps the lowest C (note) might be 0 volts, the C one octave above would be 1 volt, the next C would be 2 volts, and so on.³³ A sequencer could similarly be configured.

5.6 Modulation Matrices

With a modular synthesizer, you can practically plug anything into anything to do modulation. But as modular synthesizers gave way to compact, all-in-one units in the 1970s, this ability was lost. Instead, manufacturers created fixed modulation routings for the most common needs. With the digital age, as synthesizers became so complex that the number of hard-coded modulations got out of hand, often outnumbering the actual parameters of the individual elements in the synthesizer. This was to be expected, as the number of modulation routings between n elements grows as $O(n^2)$. Consider the modulation options for the **Kawai K4**, a digital synthesizer circa 1989. Figure 39 reveals how large a proportion the hard-coded modulation options and dedicated envelope parameters were compared to the synthesizer parameters as a whole.

The obvious solution is to replace the cables not with hard-coded modulation routings but with a table in which the musician could enter his desired modulation routings for the given patch: essentially a table describing which cables should go where. This is a **modulation matrix**.³⁴ A typical modulation matrix contains a **modulation source**, a **modulation destination**, and a (possibly negative) **modulation amount** to be multiplied against the source signal before it is fed to the destination.

Some synthesizers augment modulation matrices with **modifier functions**. A modifier takes one or two inputs from modulation sources, runs them through some mathematical function (perhaps multiplying or adding them), then outputs the result as an available modulation source option in the matrix.



Figure 39 Screenshots of portions of the Edisyn patch editor for the Kawai K4: (top) one oscillator and (bottom) one filter. **Orange boxes** indicate K4 parameters for hard-coded modulation routings. **Purple boxes** indicate parameters of dedicated envelopes for amplitude (top) and filter cutoff (bottom).

³³This is known as **volt per octave**. Many Korg and Yamaha synthesizers used an alternative encoding: **hertz per volt**. Here, rather than increasing voltage by 1 per octave, the voltage would *double* for one octave. This had the nice quality of being equivalent to frequency, which likewise doubles once per octave.

³⁴Modulation matrices as an alternative to cables were common in hardware, where they were known as **patch matrices**. For example, the EMS VCS3 sported one, albeit with a source and destination, but no modulation amount. See Figure 48 in Section 6.1. To my knowledge, the first software modulation matrix in a stored-program commercial synthesizer appeared in Oberheim's aptly named **Matrix** series of analog synthesizers.

5.7 Modulation via MIDI

Since the early 1980s, nearly all non-modular synthesizers (and some modular ones!) have been equipped with **MIDI**, a serial protocol to enable one device (synthesizer, computer, controller, etc.) to send messages or remotely manipulate another one. MIDI is most often used to send note data, but it can also be used to send modulation information as well.

For example, consider the keyboard in Figure 40. This keyboard makes no sound: it exists solely to send MIDI information to a remote synthesizer in order to control it. And it is filled with options to do so. In addition to a two-octave keyboard, it has numerous buttons which can send on/off information (the analogue of Gate), and sliders, encoders, potentiometers, a joystick, and even a 2D touch pad which can send one or two real-valued signals each (the analogue of CV).

In MIDI, this kind of control data is sent via a few special kinds of messages, notably **Control Change** or **CC** messages. Note however that CC is fairly low-resolution and slow: changes in response to CC messages may be audibly discretized, unlike the smooth real-valued CV signals in modular systems. We could deal with this by smoothly interpolating the discrete changes in the incoming MIDI signal, but this is going to create quite a lot of **lag**.

See Section 12.2 for more information about MIDI.



Figure 40 Novation Remote 25 controller.^{©22}

6 Subtractive Synthesis

Subtractive synthesis is the most common synthesis method, and while it's not as old as additive synthesis, it's still pretty old: it dates from the 1930s. The general idea of subtractive synthesis is that you'd create a sound, then start slicing into it, removing harmonics and changing its volume, and the parameters of these operations could change in real time via modulation as the sound is played. Quite unlike additive synthesis, subtractive synthesis typically is done *entirely* within the time domain. This can be more efficient than additive synthesis, and involves many fewer parameters, but many things are more difficult to implement: for example, building filters in the time domain is far more laborious than in an additive synthesizer.

Much of the defining feature of a subtractive synthesizer is its pipeline. The basic design of a *typical* subtractive synthesizer (such as in Figure 64) is as follows:

- **Oscillators** produce waveforms (sound). In the digital case, this is one sample at a time.
- These waveforms are **combined** in some way to produce a final waveform.
- The waveform is then **filtered**. This means that it is passed through a device which *removes* or *dampens* some of its harmonics, shaping the sound. This is why this method is called **subtractive synthesis**. The most common filter is a **low pass filter**, which tamps down the high-frequency harmonics, making the sound more mellow or muffled.
- The waveform is then **amplified**.
- All along the way, the parameters of the oscillators, combination, filters, and amplifier can be changed in real time via automated or human-driven **modulation** procedures.

6.1 History

The earliest electronic music synthesizers were primarily additive, but these were eventually eclipsed by subtractive synthesizers, mostly because subtractive synthesizers are much simpler and less costly to build. Whereas additive synthesis has to manipulate many partials at once to create a final sound, subtractive synthesis only has to deal with a single sound at a time, shaping and adjusting it along the way.

Subtractive synthesis has a long and rich history. A good place to start is the **Trautonium** (Figure 41), a series of devices starting around 1930 which were played in an unusual fashion. The devices had a taut wire suspended over a metal plate: when you pressed down on the wire so that it touched the plate, the device measured where the wire was pressed and this determined the pitch.³⁵ You could slide up and down



Figure 41 (Left) The Telefunken Trautonium, 1933.^{©23} (Right) Oskar Sala's "Mixtur-Trautonium", 1952.^{©24}

³⁵The plate would pass electricity into the wire where you pressed it. The wire had high resistance, so it acted like a variable resistor, with the degree of resistance proportional to the length of the wire up to the point where it was touching the plate.

the wire, varying the pitch. The pitch drove an oscillator, which was then fed into a filter. The volume of the sound could be changed via a pedal. Versions of the Trautonium became more and more elaborate, adding many features which we would normally associate with modern synthesizers culminating in sophisticated Trautoniums³⁶ such as the *Mixtur-Trautonium*.

We will unfairly skip many examples and fast forward to the **RCA Mark I / II Electronic Music Synthesizers**. The Mark I (1951) was really a music composition system: but the Mark II (1957) combined music composition with real-time music synthesis; and this was the first time the term “Music Synthesizer” or “Sound Synthesizer” was used to describe a specific device. The Mark II was installed at Princeton and was used by a number of avant-garde music composers. The Mark II was highly influential on later approaches to subtractive synthesis: as can be seen from Figure 43, the Mark II’s pipeline has many elements that are commonly found in music synthesizers today.



Figure 42 RCA Mark II.^{©25}

Modular Synthesizers Starting in 1959 **Harold Bode** developed the **modular synthesizer**, where each of the subtractive synthesis elements (oscillators, combination mechanisms, filters, amplifiers, modulation units) were implemented as separate **modules**.

A module would have knobs and buttons, plus **jacks** where **patch cables** would be inserted to attach the output one to an input on another. By connecting modules via a web of patch cables, a musician could customize the synthesizer’s audio and modulation pipeline. The knob and button settings and patch wiring together defined the instructions for making a sound. Even now, a **patch** is the standard term for the set of parameters in a synthesizer which collectively produce a sound.

In the United States, the two most important synthesizer developers to follow in the footsteps of Harold Bode were **Robert Moog** and **Don Buchla**. Each built modular synthesizers which would come to have considerable influence on the later industry. Robert Moog developed synthesizers for musicians in New York, resulting in devices designed to be musical and (relatively) easily used and programmed. The musician **Keith Emerson** (of ELP) is particularly famous for playing on a Moog modular synthesizer (See Figure 44), as is **Wendy Carlos**, whose ground-breaking album *Switched-On Bach* legitimized the synthesizer as a musical instrument in the public eye. Carlos went on to make the soundtracks for *Tron*, *The Shining*, and *Clockwork Orange*.

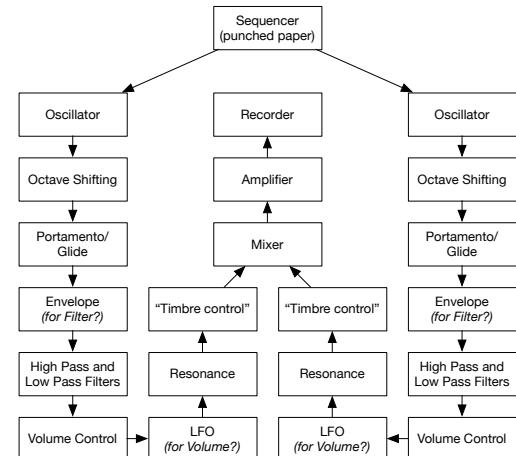


Figure 43 The Mark II’s pipeline. Compare to modules discussed in this Section, and Figure 64.



Figure 44 Moog modular synthesizer being played by Keith Emerson.^{©26}

³⁶Trautonia? Not sure what the plural ought to be.

Don Buchla primarily built machines for academics and avant-garde artists in California, notably **Ramon Sender** and **Morton Subotnick**, and so his devices tended to be much more exploratory in nature. Buchla would use uncommon approaches: a “Low-Pass Gate” (essentially a combination of a **low pass filter** and an **amplifier**), a “Source of Uncertainty”, a “Complex Waveform Generator” (which pioneered the use of **wave folding**), and so on. Buchla also experimented with nonstandard and unusual controllers, including the infamous “Multi Dimensional Kinesthetic Input Port”, shown at the bottom of Figure 45.³⁷

Moog’s and Buchla’s synthesizers were very influential, and formed two schools of synthesizer design, traditionally called **East Coast** (Moog) and **West Coast** (Buchla). The East Coast school, with more approachable architectures and elements, has since largely won out. Most modern subtractive synthesizers are variants of classic East Coast designs. However, the West Coast school has lately enjoyed a resurgence in popularity among modern-day modular synthesizer makers.



Figure 45 Buchla 200e modular synthesizer. “Multi Dimensional Kinesthetic Input Port” at bottom.^{©28}

Semi-Modular Systems As synthesizers became more popular, manufacturers worked to simplify the overall model to make it less expensive. This resulted in **semi-modular** synthesizers with a pre-defined set of built-in modules and a default pipeline, but one where the pipeline could be modified by patch cables if desired.

An example of this is the **ARP Instruments 2600** (Figure 46),³⁸ which could produce sounds with considerable flexibility with no cables at all. Semi-modular synths could be very compact indeed: another much later series were the diminutive **Korg MS10** and **MS20** (Figure 47).

Another approach was to replace the patch cables with a **patch matrix**. Consider the **Electronic Music Studios (EMS) VCS 3**, shown in Figure 48. The rows of the VCS 3 matrix were available **sources** and the columns were **destinations**. By placing a pin into a hole, a patch designer could route a source to a destination. The VCS 3 was also semi-modular in that without any pins, it still had a usable default pipeline built-in.³⁹ Even now many synthesizers have the equivalent of patch matrixes internally in software, known as **modulation matrices** (Section 5.6).⁴⁰



Figure 46 ARP 2600.^{©29}



Figure 47 Korg MS20.^{©30}

³⁷ Suzanne Ciani is an artist famous for using Buchla’s unusual methods to their fullest. Google for her.

³⁸ Fun fact: the ARP 2600 is the synthesizer which produced all of R2-D2’s sounds, as well as those of the Ark of the Covenant in *Raiders of the Lost Ark*. ARP is the initials of its founder, Alan R Pearlman.

³⁹ A monster example of a patch matrix is the predecessor to the ARP 2600, the **ARP 2500**. Google for it. The patch matrix (really a patch bus) appears both above and below the modules. The 2500 was featured in *Close Encounters of the Third Kind*.

⁴⁰ The VCS 3, and its little brother, the **EMS Synthi**, were often used by Pink Floyd. They produced many of the sounds in *On The Run*, a famous instrumental song off of *Dark Side of the Moon*.

Compact (“Analog”) Synthesizers The 1970s also saw the proliferation of synthesizers with very limited modularity or with none at all. Rather, manufacturers assumed a standard pipeline and added many hard-coded optional routing options into the synthesizers in the hopes that patching would not be necessary. The premiere example of a synthesizer like this was the **Moog Minimoog Model D**, widely used by pop and rock performers of the time, and popular even now.

The Model D had a classic and simple pipeline: three oscillators fed into a mixer, which then was put through a low-pass filter and an amplifier. The filter and amplifier each had their own envelope, and the third oscillator could be repurposed to serve as a low-frequency modulation oscillator. And that’s about it! But this very simple framework, typical of Moog design, proved able to produce a wide range of melodic sounds. The Model D is shown in Figure 49, along with a popular competitor, the **ARP Odyssey**.

Polyphonic Synthesizers The subtractive synthesizers discussed so far were all **monophonic**, meaning that they could only play one note at a time. But many instruments, and indeed much of music, is **polyphonic**: it consists of many notes (or **voices**) being played simultaneously.

Many early electronic synthesis devices, such as the Telharmonium and the Hammond Organ (Section 4), were polyphonic. But the first major polyphonic subtractive synthesizer, and indeed the first major device that one would likely view as the predecessor to modern polyphonic synthesizers in general, was the **Hammond Novachord** (circa 1930). This device had a pipeline similar to modern synthesizers, with oscillators, filters, and envelopes; but incredibly it could play all 72 notes at a time.⁴¹



Figure 48 (Left) The EMS VCS 3.^{©31} (Right) A close-up view of the VCS 3 patch matrix.^{©32}



Figure 49 (Left) Moog Minimoog Model D.^{©33} (Right) Arp Odyssey.^{©34}



Figure 50 Hammond Novachord.^{©35}

⁴¹There are different ways you could achieve polyphony. One is to give every key its own monophonic synthesizer pipeline. If you have N keys, that’s essentially N synthesizers: an expensive route! Another approach would be to use a small set of oscillators (perhaps 12 for one octave) to produce notes, and then do **frequency division**—shifting by one or more octaves—to simultaneously produce multiple notes. That’s the approach the Novachord took. A third approach, taken by most modern polyphonic synthesizers, is to have a small bank of M pipelines: when a note is pressed, a free pipeline is temporarily assigned to it to play its voice. Even so, you’d really like to have rather more voices than fingers, because a voice doesn’t stop when you let go of a key. It trails off with the release envelope, and so the synthesizer may need to **steal** a currently playing voice in order to allocate one for a new note being played. That doesn’t sound good.

Finally, a synthesizer could be **paraphonic**. This usually means that it has multiple oscillators, each playing a different note, but then they are combined and pushed through a single filter or amplifier. Many 2- or 3-oscillator monophonic synthesizers can play in paraphonic mode. Note that this means that only one note would trigger the envelopes controlling filters and amplifiers: this isn’t likely to achieve the sound of polyphony that you’d expect.

Polyphony really didn't start to come into its own until the 1970s. The **Oberheim 4-Voice** (Figure 51) and **8-Voice** were the first commercially successful polyphonic synthesizers, and were built up out of many small monophonic synthesizers developed by **Tom Oberheim**, called his **Synthesizer Expander Modules** or **SEMs**.⁴² You could play a single SEM, or two, etc., up to the huge 8-Voice. By design these modules had to be programmed *individually*. This could produce impressive sounds but was tremendous work. A device to the left of the keyboard (see the Figure) made it easier to synchronize their programming for some tasks.

The Yamaha CS series, notably the **Yamaha CS-80** (Figure 52), also offered eight voices, and emphasized expressive playing. These machines are still legendary (and costly!), as the CS-80 was the synthesizer used by **Vangelis** on his legendary soundtracks (notably *Blade Runner* and *Chariots of Fire*) and its unique and easily recognized sound is difficult to replicate.

The Korg PS series continued the Novachord tradition of total polyphony: every key had its own independent note circuitry in the extraordinary, and very expensive, **Korg PS-3300** (Figure 53).

Stored Program Synthesizers and MIDI The integrated circuit—the chip—arrived in the 1980s with it came the ability to store patches in memory and recall them. This was a major step forward for synthesizers, and marked (I think) the major division between “old school” synthesizers and newer ones.

First out of the gate was Sequential Circuits's **Prophet 5** (Figure 54). The Prophet 5 was the first commercial synthesizer to sport RAM and a CPU, and was consequently the first commercial synthesizer that could store and recall patches. For this reason it was very, very successful in the music industry.

Following the Prophet 5 came many synthesizers. These included the **Oberheim OB-X** (Figure 55) and **OB-Xa** (the latter used by **Van Halen** in the song *Jump*),⁴³ the **Moog MemoryMoog** (Figure 56), and Roland's Juno and Jupiter series, most famously its top-of-the-line **Jupiter 8** (Figure 57). These synthesizers were dominant in pop and rock songs through the early 1980s, and found their way onto soundtracks and even orchestral settings.

Compact synthesizers largely lacked the patch points or matrices of modular and semi-modular synthesizers, and the modulation flexibility that came with them. But the CPU soon fixed this, as it made possible sophisticated programmable **modulation matrices** in software, largely heralded by Oberheim's **Matrix** series.



Figure 51 Oberheim 4-Voice.^{©36}



Figure 52 Yamaha CS-80.^{©37}



Figure 53 Korg PS-3300.^{©38}



Figure 54 Sequential Circuits Prophet 5.^{©39}



Figure 55 Oberheim OB-X.^{©40}



Figure 56 Moog MemoryMoog.^{©41}

⁴²The SEM is also famous for its **state-variable** 2-pole filter which worked well with chords and which had a high degree of flexibility. A state-variable filter allows you to smoothly travel from low-pass to (say) band-pass or high-pass.

⁴³You know this song had to make an appearance in a synthesizer text, right?

Sequential Circuits's Dave Smith realized that as synthesizers became cheaper, musicians would be acquiring not just one but potentially many of them. Outfitted with a CPU and the ability to store patches in RAM, such synthesizers would benefit from communicating with one another. This would enable synthesizers or computers to play and control other synthesizers, and to upload and download stored patches. To this end Sequential Circuits and Roland proposed the **Musical Instrument Digital Interface**, or **MIDI**. MIDI soon caught on, and since then essentially all hardware synthesizers now come with it. MIDI is discussed in Section 12.2.

Controllers and Rackmounts MIDI gave rise to the notion that a synthesizer didn't need a keyboard at all, or in some cases any knobs: it could be entirely controlled, and possibly programmed, remotely over MIDI via some other keyboard. As a result, many synthesizers were developed in versions both with keyboards and without: those without were usually designed to be screwed into a 19-inch rack common in the telephone, electronics, and audio industries. Figure 57 shows a **Roland Jupiter-8** and its rackmount equivalent, the **Roland MKS-80 Super Jupiter**.

If a synthesizer didn't need a keyboard, then there was no reason a keyboard needed a synthesizer. MIDI gave birth to a new device, the *keyboard controller*, or *performance controller*, or just **controller**, which was nothing more than a keyboard or set of knobs (or both) designed to send control signals to a remote synthesizer over MIDI. Later on, controllers would also send MIDI data to a computer. These are discussed further in Section 12.

The Rise of Digital The early 1980s also saw the birth of the **digital synthesizer**. This wave, starting with **FM synthesizers**, and culminating in **samplers**, **wavetable synthesizers**, and **PCM playback synthesizers** (derisively known as **ROMplers**), all but eliminated analog synthesizers from the market. While many of these synthesizers employed pipelines similar to subtractive ones, their oscillator designs were quite different. They also generally had many more parameters than analog synthesizers, but to keep costs down their design tended towards a menu system and perhaps a single data entry knob, making them very difficult to program.

In 1995 **Clavia** introduced the **Nord Lead** (Figure 58), a new kind of digital synthesizer. This synthesizer attempted to emulate the characteristics, pipeline, modules, and style of a classic analog subtractive synthesizer, using digital components. Clavia called this a **virtual analog synthesizer**. Since the introduction of the Nord Lead, virtual analog synthesizers have proven popular with manufacturers, largely because they are much cheaper to produce than analog devices. Perhaps the most famous example of this is the **Korg microKORG** (Figure 59). This was an inexpensive virtual analog synthesizer with an additional microphone and **vocoder**, a device to sample and resynthesize the human voice (here, for the purpose of making a singer sound robotic). The microKORG is considered one of the most successful synthesizers in history: it was introduced in 2002, sold at least 100,000 units as of 2009, and is still being sold today.



Figure 57 Roland Jupiter-8^{©42} (top) and MKS-80^{©43} (bottom).



Figure 58 Clavia Nord Lead 2x.^{©44}



Figure 59 Korg microKORG.^{©45}

Virtual analogs are software emulations of synthesizers embedded in hardware: but there is no reason that one couldn't just do software emulation inside a PC. Many digital synthesizers—subtractive or not—now take the form of computer programs commonly called **software synthesizers** or **softsynths**. These often take the form of plugins to **Digital Audio Workstations** using plugin library APIs, such as Steinberg's **Virtual Studio Technology** (or VST) or Apple's **Audio Unit** (or AU). Figure 60 shows two examples: **PG-8X**, a softsynth inspired by Roland's **JX-8P** analog synthesizer; and **OBXD**, an emulation of the **Oberheim OB-X** or **OB-Xa**.

The Return of Analog What goes around comes around. Partly driven by nostalgia for old, rich sounding synthesizers, partly out of a yearning to escape the intangibility of softsynths, and perhaps partly driven by a distaste for the flood of cheap digital synths with poor interfaces, a significant number of musicians have returned to analog devices. Many of these include mono- and polyphonic analog synthesizers in-line with earlier compact designs, such as the **Dave Smith Instruments Prophet 6** (Figure 61), which directly recalls Dave Smith's earlier Sequential design, the **Prophet 5** (Figure 54 on page 59).

Another major trend has been the resurgence of fully modular synthesizers. In 1996, **Doepfer Musikelektronik** began to promote a new modular synthesizer standard called **Eurorack**, built around short modules, small 3.5mm jacks, and standardized power distribution. Eurorack has since attracted a great many small and independent manufacturers. Eurorack pays homage to older modular designs: it is monophonic, uses older control methods (notably **CV/Gate**) instead of MIDI, and has no saved patch capability. And while most of its market is East Coast, Eurorack has given new life to West Coast synthesis and more exotic approaches as manufacturers explore more and more esoteric designs.

Eurorack is also very expensive: but its popularity has led to the recent development of an entirely new market for **semi-modular** synthesizers which provide some of the nostalgic and exploratory appeal of modular but at a more reasonable price point. Synthesizers in this category are largely compatible with Eurorack, and likewise have no saved patch capability or polyphony: but as they are semi-modular, they can be useful synthesizers even with no cables plugged in. One particularly interesting example is the **Make Noise 0-Coast**, a small tabletop semi-modular synthesizer with elements drawn from *both* the East and West Coast schools of synthesizer design.⁴⁴



Figure 60 PG-8X and OBXD software synthesizers.



Figure 61 Dave Smith Instruments Prophet 6.⁴⁶



Figure 62 A Eurorack synthesizer with modules from different manufacturers.⁴⁷



Figure 63 Make Noise 0-Coast.⁴⁸

⁴⁴Hence the name. The “0-” is notionally pronounced “No” as in “No-Coast”.

6.2 Implementation

The classic subtractive synthesis pipeline, shown in Figure 64, is similar in some sense to the additive pipeline in Figure 29. The big difference, of course, is that the modules do not pass partials to one another, but rather pass sound waves. That is, they work in the time domain rather than the frequency domain.

A large number of subtractive synthesizers follow same general pattern: **oscillators** generate sounds, which are then **combined**, then passed through two modifiers, namely a **filter** and an **amplifier**; and any of these can be **modulated**. But there is significant variability in the details. For example, synthesizers might have one, two, three, or even more oscillators; there might be a number of combination options; the filters might get quite extravagant; and there might be a many different modulation options, including the ability for modulators to modulate the parameters of one another. We will discuss oscillators, combiners, and amplifiers in Sections 7.1 and 7.6 coming up, with some follow-up in Section 10. Filters are discussed in Section 8.

Similarly, the top-level algorithmic architecture for a basic, monophonic subtractive synthesizer is very similar to the one used for an additive synthesizer (Algorithm 6). But the additive synthesizer only updated its partials every ticksPerUpdate ticks because the process of updating modules is so costly. Here though that logic has been stripped out and everything simplified (indeed, made simplistic) because a subtractive synthesizer can update all of its modules much more efficiently. So we have:

Algorithm 16 Simple Monophonic Subtractive Synthesizer Architecture

```

1:  $M \leftarrow \langle M_1, \dots, M_m \rangle$  modules
2: tick  $\leftarrow 0$ 

3: procedure Tick
4:   tick  $\leftarrow$  tick +1
5:   if Note Released then
6:     for  $i$  from 1 to  $m$  do
7:       Released( $M_i$ , pitch)
8:   if Note Pressed then
9:     for  $i$  from 1 to  $m$  do
10:      Pressed( $M_i$ , pitch, volume)
11:   for  $i$  from 1 to  $m$  do
12:     Update( $M_i$ , tick)
13:   return OutputSample(tick)

```

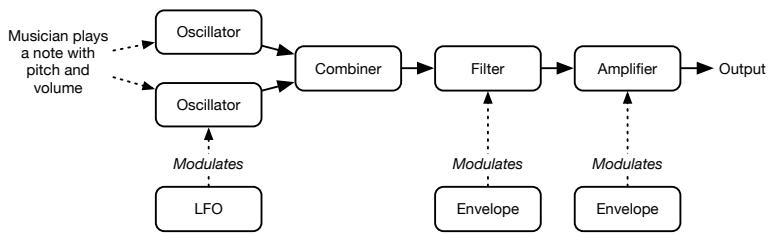


Figure 64 One possible (and pretty typical) two-oscillator subtractive pipeline, in East Coast tradition. Unlike in additive synthesis, where arrays of partials were passed from module to module, here *sound waves* are passed, one sample at a time (in the digital case).

The function OutputSample(tick) would simply take the most recent sample it's received and submit it to the audio stream to be played. And as was the case for the additive version of this

algorithm, in a monophonic subtractive synthesizer a new note could be pressed before the previous note was released (playing **legato**), and some modules might respond specially when this happens, such as doing a **portamento** slide from the old note to the new one.

The end of Section 4.3 had some critical discussion about buffering and latency and how to make the `Tick()` method consistent in timing for additive synthesizers. That discussion applies in this case as well.

6.3 Architecture Examples

We have not yet discussed the details of subtractive synthesizers, and so many of the terms discussed in this next section may not yet make sense. After you have read Sections 7 and 8 come back to this section and the terminology will be clearer.

Below, we discuss two compact polyphonic subtractive synthesizers: the fully analog **Oberheim Matrix 6**, and the virtual analog **Korg microKORG**. The release dates of these two synthesizers differ by two decades, and yet they have a great many things in common. We also discuss a modular synthesizer format, **Eurorack**. Also recall that in Section 1.3 we covered the architecture of a fourth subtractive synthesizer, the **Dave Smith Instruments Prophet '08**.

Oberheim Matrix 6 The **Matrix 6** is a 6-voice, polyphonic analog subtractive synthesizer with analog but **Digitally Controlled Oscillators (DCOs)** produced by Oberheim between 1986 and 1988. The Matrix 6 came in three forms: a keyboard, a rack-mount version without a keyboard (the **Matrix 6R**), and a small rackmount version designed largely for presets (the **Matrix 1000**). Like many digital synthesizers of the time, and contrary to nearly all prior analog synthesizer tradition, the Matrix 6 eschewed knobs and switches. It instead relied entirely on a tedious keypad entry system to set its 100-odd patch parameters. In fact, the Matrix 1000 *could not be programmed at all*⁴⁵ from its front panel: all you could do was select from approximately 1000 presets.

The Matrix 6 had two oscillators per voice, each of which could produce a simultaneous **square wave** and a **sawtooth/triangle** wave. The square wave had adjustable **pulse width**, and the sawtooth/triangle wave could be adjusted from a full sawtooth to a full triangle shape, or something in-between. The two oscillators could be **detuned** relative to one another, and the first oscillator could be **synced** to the second. The second oscillator could also be used to produce **white noise**. These two oscillators were then **mixed** together to form a final sound, which was then passed through a **4-pole resonant low pass filter**, and then finally an **amplifier**. The low-pass filter sported **filter FM** (see Section 9), which enabled the first oscillator to modulate the cutoff frequency of the filter at audio speeds, creating unusual effects.



Figure 65 Oberheim Matrix 6.^{©49}

⁴⁵You can program the Matrix 1000: but you must do so via commands sent over MIDI from a **patch editor**, typically a dedicated software program.

The Matrix 6 was notable for (at the time) its broad array of modulation options. Both the filter and the amplifier had dedicated **DADSR envelopes**, and a third DADSR modulated the degree of filter FM. Oscillator frequency and oscillator pulse width also had dedicated LFOs with many shapes. The amplitude of each LFO could be modulated by a move from 0 to 1 over time. But this was not all. As befitting its name, the Matrix 6 also had a ten-slot **modulation matrix** which could route modulation signals between a wide variety of sources and destinations. In addition to the obvious sources, all of the envelopes, LFOs, and ramps could be repurposed as sources as well. The 32 destinations included parameters for oscillators, the filter, amplifier, and all of the modulation facilities.

A final modulation source, the **tracking generator**, allowed the musician to specify a five-point piecewise linear function $f(x) \rightarrow y$ with x and y ranging 0...1. The tracking generator was attached to a source (as x) and its y value could be used as a modulation source. This allowed the musician to make changes to the output of a single modulation source before sending it to its final destination. This is an example of a **modifier function**, as discussed in Section 5.6.

Korg microKORG Korg appeared relatively early on the virtual analog synthesizer scene with a keyboard synthesizer model called the **Korg MS2000** (Figure 67). This machine was not only a **virtual analog synthesizer**, but could also serve as a digital **vocoder**. The MS2000 had several versions, but Korg didn't hit pay dirt until it released a stripped down version of the MS2000 in 2002: the very successful **microKORG**.

As shown in Figure 67, the MS2000 was a beast of a machine, while the microKORG (Figure 68) was a tiny little thing with cheap minikeys. Yet these two devices had, more or less, the same synthesizer engine built in!⁴⁶

We will skip the vocoder features of the machine and concentrate on the subtractive virtual analog architecture. Each voice contained up to two parallel self-contained virtual synthesizer pipelines Korg called "timbres". If the microKORG was using one timbre per voice, it could play up to four voices; with two timbres per voice it could play only two voices.

Each timbre contained two oscillators. Both oscillators could do sawtooth, triangle, or square waves. The first oscillator also could do a number of other waves, including sine waves, analog input, noise, and 64 different hard-coded **single-cycle digital waves** (we'll discuss this more in Section 10). The second oscillator could also be **ring-modulated** by the first oscillator, **synced** to it, or both.

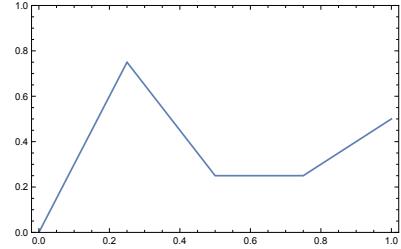


Figure 66 Five-point Piecewise-linear Tracking Generator.



Figure 67 Korg MS2000.^{©50}



Figure 68 Korg microKORG. This is a repeat of Figure 59 on page 60.^{©51}

⁴⁶This is not entirely unheard of. The entire Casio CZ series, discussed later in Section 7.5, used the exact same synthesis engine in machines ranging from the large CZ-1 (Figure 82) for \$1400 to the tiny CZ-101 for \$499. The CZ-101 was the first significant synthesizer to break the \$500 barrier (and was the spiritual predecessor to the microKORG, as it was an early example of a professional synthesizer with minikeys).

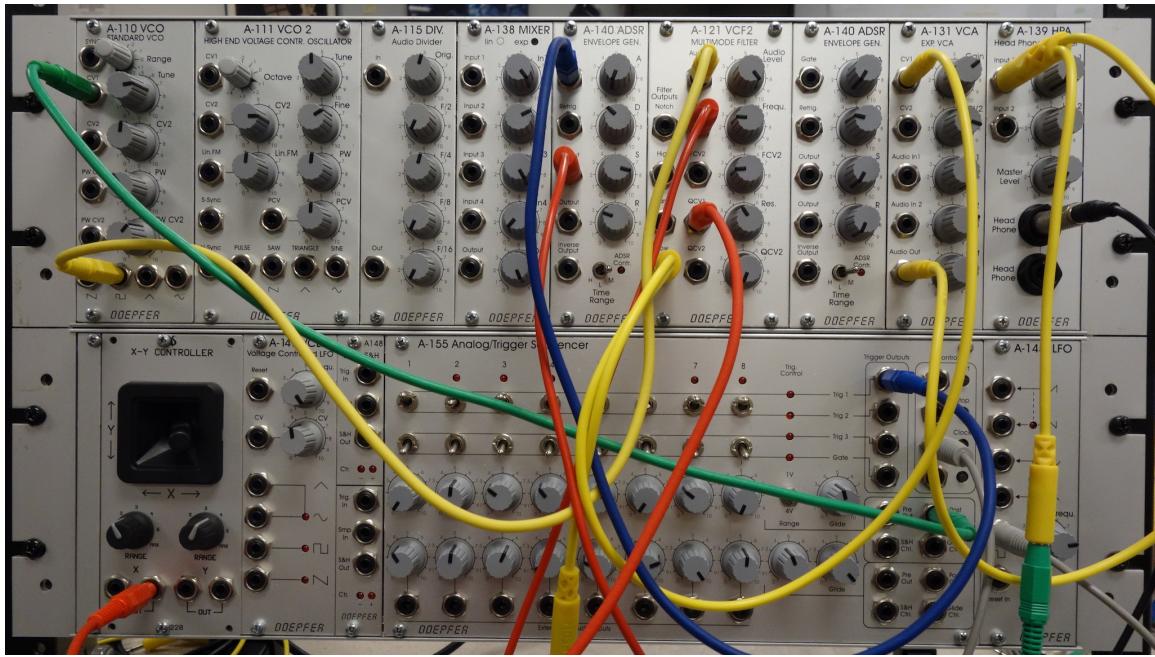


Figure 69 A small Eurorack-format modular synthesizer, with Doepfer and **Analogue Solutions** modules.

Each timbre also contained a resonant filter (**4-pole or 2-pole low pass**, **2-pole band-pass**, or **2-pole high pass**) with its own dedicated ADSR envelope. The sound was then passed through a stereo amplifier with its own ADSR envelope and a distortion effect. A timbre had two free LFOs available, as well as a small, four-slot patch matrix with a small number of sources and destinations.

Finally, the microKORG had a built-in **arpeggiator** and three **effects** units through which the audio was passed. The first effects unit could provide **chorus**, **flanging**, or **phasing**, the second provided **equalization**, and the third provided some kind of **delay**. We'll discuss effects in detail in Section 11. Overall, while the microKORG (and MS2000 before it) had more filter and oscillator options, they had much less modulation flexibility and lower polyphony than the Matrix 6.

Eurorack Modular Synthesizers Eurorack is a popular format for modern **modular synthesizers**. The format was introduced in 1995 by **Doepfer**, and now many manufacturers produce modules compatible with it. Like essentially all hardware modular synthesizers, Eurorack is monophonic: it can produce only one sound at a time.

Eurorack signals normally take one of three forms: audio signals, **gate** signals (which indicate a “1” or a “trigger” by moving from a low voltage to a high voltage, and a “0” by doing the opposite), and **control voltage** (or **CV**) signals, which typically vary in voltage to indicate a real-valued number. Gate and CV are used for modulation. All Eurorack jacks are all the same regardless of the kind of signal they carry: thus there's no reason you couldn't plug an audio output into a CV input to provide very high-rate modulation of some parameter.

The small Eurorack synthesizer shown in Figure 69 is a typical specimen of the breed. It contains all the basic modules you'd find in a subtractive synthesizer; but be warned that the Eurorack community has produced many kinds of modules far beyond these simple ones. This synthesizer contains the following audio modules:

- Two **Voltage-Controlled Oscillators (VCOs)**, which produce sawtooth, square, triangle, or sine waves.
- A **suboscillator** (labelled “Audio Divider” in the Figure) capable of taking an input wave and producing a combination of square waves which are 1, 2, 3, and 4 octaves below in pitch.
- A **mixer** to combine the two oscillators and suboscillator.
- A resonant four-pole **filter** with options for low-pass, high-pass, band-pass, and notch. In the figure, frequency is referred to as “F” and resonance is referred to as “Q” in the labels.
- A **Voltage Controlled Amplifier or VCA**.
- A **headphone amplifier** to output the final sound at headphone levels.

(Mostly) below these modules are modulation modules which output Gate, CV, or both:

- A two-axis **joystick**.
- Two **Low-Frequency Oscillators (LFOs)** producing triangle, sine, square, or sawtooth waves.
- A dual **Sample and Hold** or (**S&H**) module which takes an input signal and a trigger (a gate), and outputs the held value.
- An 8-stage **step sequencer**. This is often clocked by a square wave LFO, and outputs up two triggers and two CV values per step.
- Two **ADSR envelopes**, notionally for the filter and amplifier respectively.

The whole thing might be driven by an additional modulation source: a keyboard with gate (indicating note on or note off) and CV (indicating pitch).

All of the signals in this synthesizer are analog. All of the audio modules in this synthesizer are analog as well; though many Eurorack modules use digital means to produce their synthesized sounds. You’ll note from the picture the presence of cables attaching modules to other modules. These cables transmit audio, gate, or CV information.

7 Oscillators, Combiners, and Amplifiers

A subtractive synthesis pipeline typically consists of **oscillators**, which produce sounds, **combiners** which join multiple sounds into one, and **filters** and **amplifiers**, which modify sounds, producing new ones (plus **modulation** of course, discussed in Section 5). Filters are a complex topic and will be treated separately in Section 8. In this Section we'll cover the others.

7.1 Oscillators

An oscillator is responsible for producing a sound. Oscillators at a minimum will have a parameter indicating the frequency (or pitch) of the desired sound: they may also have a variety of parameters which specify the shape of the sound wave. Many early oscillators took the form of **Voltage Controlled Oscillators** (or VCOs), meaning that their frequency (and thus pitch) was governed by a voltage. This voltage could come from a keyboard, or from a musician-settable dial, or could come from a modulation signal from some modulator. VCOs are not very stable and can drift or wander slightly in pitch, especially as temperature changes. Later oscillators had their frequency governed by a digital signal: these were called **Digitally Controlled Oscillators** or DCOs. A DCO is still an analog oscillator, but the frequency of the analog signal is kept in check by a digital timer.⁴⁷ On the other hand, **Numerically Controlled Oscillators**, or NCOs, are not analog devices: they produce extremely-high-resolution faithful implementations of analog signals typical of VCOs or DCOs.

Other oscillator designs are unabashedly digital: they can produce complex digital waveforms via a variety of synthesis methods, including **wavetables**, **pulse code modulation** (or PCM), or **frequency modulation** (or FM), among many other options. We'll discuss these methods later.

Early on (and even now!) the most common early oscillator waveforms were the **triangle**, **sawtooth**,⁴⁸ and **square**, shown in Figure 70. These waveforms were fairly easy to produce via analog electronics; they had a rich assortment of partials, which provided good raw material for downstream filters to modify; and their partials were all **harmonics**: their frequencies were integer multiples of the fundamental partial. This property made them “tonal” or “musical” sounding.

In these waves, the fundamental is loudest, and amplitudes of higher harmonics drop off from there. In a sawtooth wave, the amplitude of harmonic # i is $1/i$ (where $i = 1$ is the fundamental). In a triangle wave, the

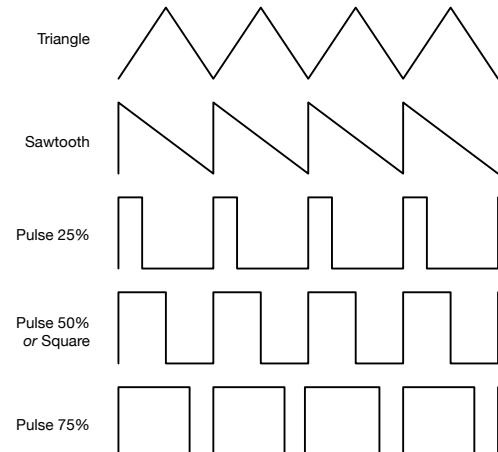


Figure 70 Triangle, sawtooth, and three pulse waveforms at different pulse widths. The 50% pulse width pulse wave is commonly known as a square wave. Note relationship to Figure 33.

⁴⁷DCOs are better technology than VCOs: but nostalgic musicians like the drifting nature of VCOs which, when layered over one another, are thought to produce a more organic or warmer sound, despite their other failings.

⁴⁸Perhaps you recall these wave shapes from Section 5.1. Some synthesizers implement a sawtooth sound using a sawtooth wave, while others produce a **ramp** wave $\diagup\diagup\diagup$. This distinction matters for LFOs used in modulation, but not for oscillators producing audio-frequency waves, because ramp and sawtooth sound the same.

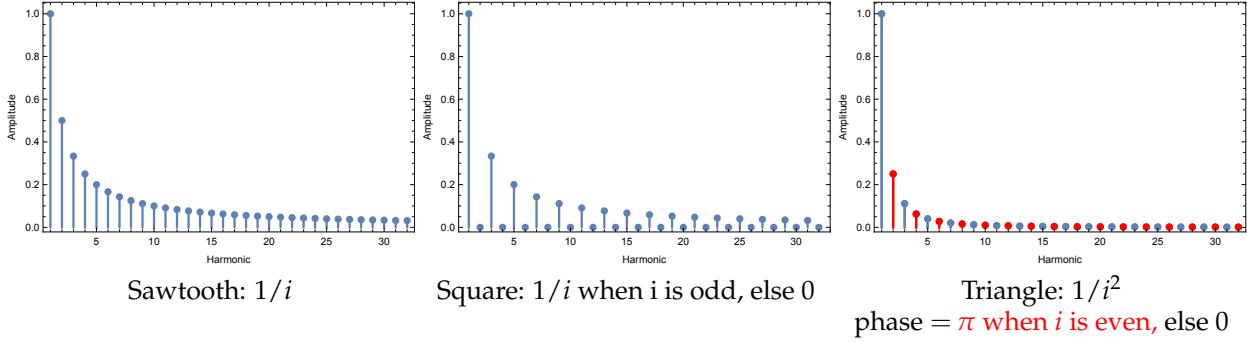


Figure 71 Harmonics of the sawtooth, square, and triangle waveforms. i is an integer ≥ 1

amplitude of harmonic # i is $1/i^2$, but the phase of every even harmonic was shifted by π .⁴⁹ This squared dropoff means that the sum total amplitudes of the triangle wave are much less than the sawtooth wave: and indeed a triangle wave is much quieter.

In a square wave, the amplitude of harmonic # i is $1/i$ when i is odd, but 0 when even. This is also quieter than a sawtooth wave, but louder than a triangle wave. A square wave is just a special case of a **pulse** wave. As can be seen in Figure 70, pulse waves come in different shapes, dictated by a percentage value called the **pulse width** or **duty cycle** of the waveform. The pulse width is the percentage of time the pulse wave stays high versus low. A square wave has a pulse width of 50%.⁵⁰

You'll notice that one very famous wave is curiously missing. Where's the **sine** wave? There are two reasons sine is not as common in audio-rate oscillators. First, it's nontrivial to make a high quality sine wave from an analog circuit. But second and more importantly, a sine wave consists of a *single partial*. That's almost no material for downstream filters to work with. You just can't *do* much with the audio from a sine wave.⁵¹

Similarities to Certain Musical Instruments These waves can be used as raw material for many artificial synthesized sounds. But some of them have properties which resemble certain musical instruments, and thus make them useful in those contexts. For example, when a bow is drawn across a violin string, the string is snagged by the bow (due to friction) and pulled to the side until friction cannot pull it any further, at which time it snaps back. This process then repeats. The wave movement of a violin string thus closely resembles a sawtooth wave.

Brass instruments also have sounds produced by processes which resemble sawtooth waves. In contrast, many reed instruments, such as clarinets or oboes, produce sounds which resemble square waves, and flutes produce fairly pure sounds which resemble sine waves.

Noise Another common oscillator is one which produces *noise* (that is, hiss). Noise is simply random waves made up of *all* partials over some distribution. There are certain particularly common distributions of the spectra of noise, because they are produced by various natural or physical processes. One of the most common is **white noise**, which has a uniform distribution

⁴⁹Recall again that humans can't distinguish phase, so this is of largely academic interest.

⁵⁰Generally speaking, the amplitude of harmonic # i in a pulse wave of pulse width p is $\left| \frac{\sin(\pi i p)}{i} \right|$. You might ask yourself what happens to this equation when the pulse width is 0% or 100% (0.0 or 1.0).

⁵¹Note however that Low Frequency Oscillators can do a lot with sine waves, so it shows up in them all the time.

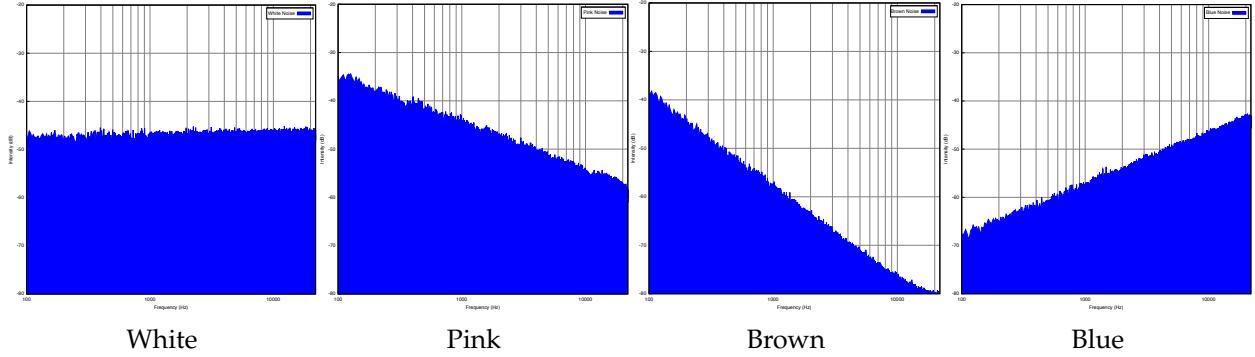


Figure 72 Frequency spectra plots of four common kinds of noise: (Left to right) White, Pink, Brown, and Blue noise. Note that the plots are logarithmic in both directions.⁵²

of its partial spectra across all frequencies. Another common noise is **pink noise**, whose higher frequencies taper off in amplitude (by 3dB per octave). **Brown noise**, so called because it is associated with **Brownian motion**, tapers off even faster (6dB per octave). Finally, **blue noise** increases with frequency, by 3dB per octave. There are plenty of other distributions as well. Noise is often used to dirty up a synthesized sound. It is also used to produce explosive sounds, or sharp, noisy sounding instruments such as snare drums.

How do you create random noise? White noise is very simple: just use a uniform random number generator for every sample (between $-1\dots+1$ say). Other kinds of noise can be achieved by running white noise through a **filter** to cut down the higher (or in the case of Blue noise, lower) frequencies. We'll talk more about filters in Section 8.⁵²

Suboscillators One trick analog synthesizers use to provide more spectral material is to offer one or more **suboscillators**. A suboscillator is very simple: it's just a circuit attached to a primary oscillator which outputs a (nearly always square) waveform that is $1/2$, $1/4$, $1/8$, etc. of the main oscillator's frequency. $1/2$ the frequency would be one octave down. This isn't a true oscillator—it's pitch is locked to the primary oscillator's pitch—but it's a cheap and useful way of adding more complexity and depth to the sound.

⁵²Oh, so you wanted an actual algorithm? Okay, here's a simple but not super accurate one originally from Paul Kellet: see <http://www.firstpr.com.au/dsp/pink-noise/> It generates random white noise samples and then applies an **FIR** low-pass filter to them. More on filters in Section 8.

Algorithm 17 Pink Noise

```

1:  $P \leftarrow \langle p_1\dots p_N \rangle$  pink noise samples
2:  $b_0, b_1, b_2 \leftarrow 1.0$ 
3: for  $i$  from 1 to  $N$  do
4:    $w \leftarrow$  random real-valued number from  $-1$  to  $+1$  inclusive            $\triangleright$  (White Noise)
5:    $b_0 \leftarrow 0.99765 b_0 + 0.0990460 w$ 
6:    $b_1 \leftarrow 0.96300 b_1 + 0.2965164 w$ 
7:    $b_2 \leftarrow 0.57000 b_2 + 1.0526913 w$ 
8:    $p_i \leftarrow b_0 + b_1 + b_2 + 0.1848 w$ 
9: return  $P$ 

```

7.2 Antialiasing and the Nyquist Limit

One critical problem which occurs when an oscillator generates waves is that they can be **aliased** in a digital signal. This issue must be dealt with or the sound will produce considerable undesirable artifacts when played. Aliasing is the *central challenge* in digital oscillator design.

A digital sound can only store partials up to a certain frequency called the **Nyquist limit**. This is one half of the sampling rate of the sound. For example, the highest frequency that 44.1KHz sound can represent is 22,050Hz. If you think about this it makes sense: to represent a sine wave, even at its crudest, you need to *at least* go up and then down: meaning you'll need at least two samples.

But consider a sawtooth wave for a moment. The sawtooth wave has an infinite number of partials, and the high-frequency ones are what cause it to have nice sharp angles. However, we can only represent so many sawtooth partials before the higher frequency partials exceed the Nyquist limit; at which point we must stop. The lower the sampling rate, the fewer partials we can include. Consider Figure 73 at right. Notice that a wave consisting of just the first 20 partials in a sawtooth wave is much cruder than one generated with the first 100 partials (or first 100,000!).

Why do we have to stop there? Because any higher frequency partials we try to include in the signal get “reflected” away from the Nyquist limit.⁵³ That is, if we tried to insert a partial of frequency $N + f$, where N was the Nyquist limit, what would *actually* appear in the signal would be a partial of frequency $N - f$. Furthermore, if $N - f < 0$, then this would reflect back up again as $0 - (N - f)$, and so on.

Figure 74 illustrates this effect. The result is *definitely not* a sawtooth wave; and as the wave increases in frequency (pitch), the reflections start doing unexpected things, resulting in a nonstable, strange sound. This is **aliasing**.

The unfortunate audio effect of aliasing is hard to explain in a text: you have to hear it for yourself. But you’ve probably seen the effect of aliasing in two-dimensional images with lots of checkerboards or straight lines: **Moiré patterns**. Figure 75 shows what this looks like. Aliasing in 2D images is caused by the exact same thing as in sound: higher frequencies than the image’s resolution is capable of supporting.

To counter aliasing in audio we must be diligent in eliminating frequencies higher than Nyquist from the waves generated by the oscillators. That is, our waves must be **band limited**. There are several approaches one could take to do this:

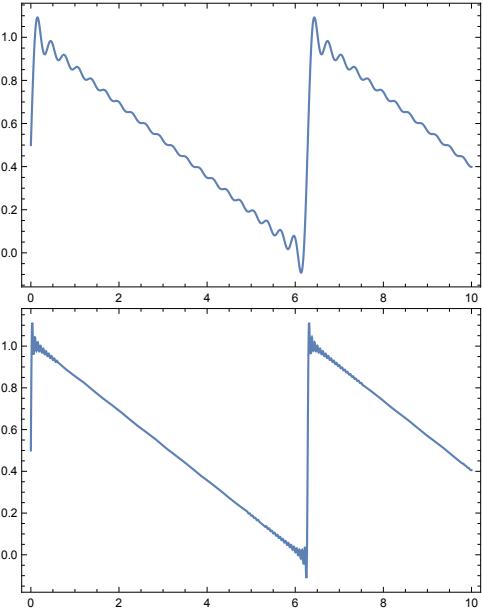


Figure 73 Sawtooth wave approximations consisting of only (top) the first 20 partials or (bottom) the first 100 partials.

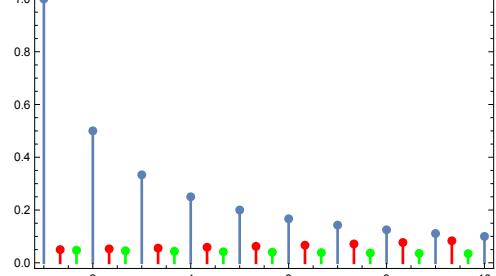


Figure 74 Partials of an Aliased Sawtooth wave. The sequence of sawtooth partials (Blue) reflects off the Nyquist limit (here 11) and continues backward (Red), then bounces *again* off of 0 and continues forward again (Green), and so on. Only the first two bounces shown.

⁵³This is why aliasing is sometimes called **foldover**: the reflected partials are “folded over” the Nyquist limit.

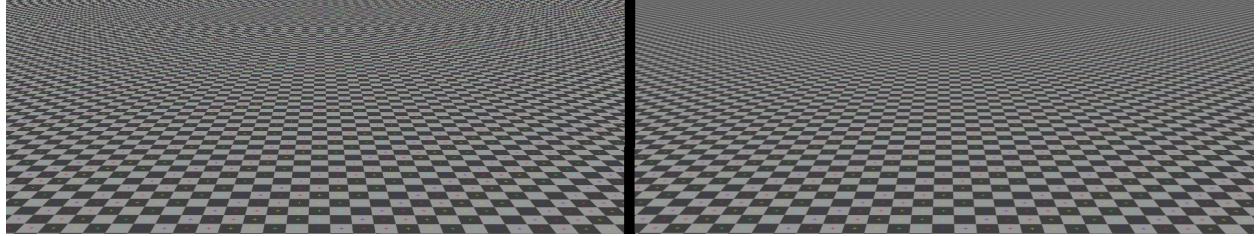


Figure 75 Moiré patterns (left) in an image. This effect is due to the 2D version of aliasing.⁵³

Additive Synthesis We could build the wave by adding partials up to the Nyquist limit.

Filtering and Resampling This is the most common approach. We create a bandlimited wave at a high sampling rate and store a single cycle of it as a digital sample: this is a **single-cycle wave**. When we need to convert wave to a certain frequency, this is equivalent to **resampling** it into a smaller number of samples. We first apply a **low pass filter** to strip out any frequencies higher than the resulting Nyquist limit for our smaller number of samples; then we perform the resampling. This is a variation of so-called **wavetable synthesis** (Section 10.3). The process of resampling is discussed in Section 10.5.

Discrete Summation Formulas (DSFs) This is a clever way of generating a band-limited wave without having to add up lots of sine waves as is the case for additive synthesis.⁵⁴ It turns out that you can add up $N + 1$ sine waves of a certain useful pattern just by using the following identity:

$$\sum_{k=0}^N a^k \sin(\theta + k\beta) = \frac{\sin(\theta) - a \sin(\theta - \beta) - a^{N+1} [\sin(\theta + N\beta + \beta) - a \sin(\theta + N\beta)]}{1 - a^2 - 2a \cos(\beta)}$$

That is an interesting identity: it would allow us to do a variety of waves without computing a sum. Note that you'd not be able to create (for example) a sawtooth from it, as a DSF can only produce something with harmonics of the form $\sum_k a^k \sin(fk)$, whereas a sawtooth drops off as $\sum_k \frac{1}{k} \sin(fk)$, but perhaps with a judicious setting of a you could get a very bad approximation ($a = 0.75$ seems interesting). And it has other interesting timbres as well.

Band Limited Impulse Trains (BLITs)

An impulse train is a wave that largely consists of zeros, but where every N th sample is 1.0. It's a sequence of impulses or delta functions. Not surprisingly, an impulse train has many frequencies above Nyquist: but we can create a version of it with the high frequencies stripped out. This is called a *band limited impulse train*, or **BLIT**.⁵⁵

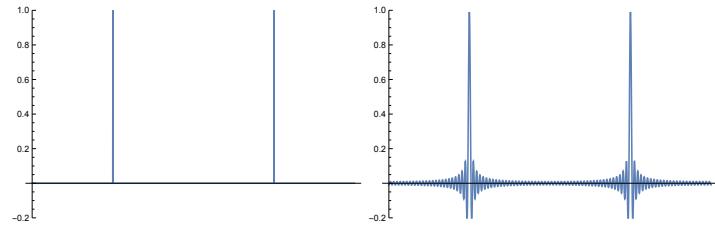


Figure 76 (Left) A Delta or Impulse train. (Right) An equivalent BLIT, band limited to quite low frequencies for illustrative purposes.

⁵⁴J. A. Moorer, 1976, The synthesis of complex audio spectra by means of discrete summation formulae, *Journal of Engineering*, 24, 717–727.

⁵⁵Timothy Stilson and Julius O. Smith, 1996, Alias-free digital synthesis of classic analog waveforms, in *International Computer Music Conference (ICMC)*.

A BLIT is defined as:

$$\text{blit}(x, P) = (M/P) \text{sinc}_M[(M/P)x] \quad M = 2\lfloor P/2 \rfloor - 1 \quad \text{sinc}_M(x) = \begin{cases} 1 & M \sin(\pi x/M) = 0 \\ \frac{\sin(\pi x)}{M \sin(\pi x/M)} & \text{otherwise} \end{cases} \quad (2)$$

Here P is the period of the impulse train in samples, and M is the number of partials (harmonics) to include. x is the x th sample. The maximum number of harmonics happens to be related to P as shown, so we can compute it on the fly.

What can we do with this? Well, a band-limited sawtooth for one:

$$\text{saw}(x, P) = \begin{cases} 0 & x \leq 0 \\ \alpha \text{saw}(x-1, P) + \text{blit}(x, P) - 1/P & \text{otherwise} \end{cases} \quad \alpha = 1 - 1/P \text{ seems good}$$

What's going on is this: we're just integrating (summing) the BLIT over time: this creates a staircase function. And then we're bit by bit subtracting from the staircase to get it back down to zero. Since BLIT shoots up to 1, we need to subtract out a 1 before we get to the next period, so we subtract out $1/P$ each time. I find this is reasonably scaled to $0...1$ as $\text{saw}(\dots) \times 0.8 + 0.47$.

This starts out very noisy but cleans up after about 6 cycles or so. The thing that cleans it up is the α bit: this is a **leaky integrator**: it causes the algorithm to gradually forget its previous (initially noisy) summation.⁵⁶

To do a square wave, we need a new kind of BLIT, where the pulses alternate up and down. We'll call that a BPBLIT:

$$\text{bpblit}(x, P, D) = \text{blit}(x, P) - \text{blit}(x - PD, P)$$

Here D (which ranges from 0 to 1, 1/2 being default) is the duty cycle of the BPBLIT: it'll cause the low pulses to move closer to immediately after the high pulses. Armed with this, we can define a band-limited square wave as just the integration (sum) of the BPBLIT.

$$\text{square}(x, P, D) = \begin{cases} 0 & x \leq 0 \\ \alpha \text{square}(x-1, P, D) + \text{bpblit}(x, P, D) & \text{otherwise} \end{cases} \quad \alpha = 0.999 \text{ seems good}$$

I find this is reasonably scaled as $\text{square}(\dots) \times 0.75 + \text{phase}$.

Last how might we do a triangle? With the same integration trick again, but this time summing over the square wave. This double summing (the square wave was itself summed), with two leaky integrators, means the triangle will have a lot of delay, so this may not work well with a fast pitch bend. That's why I have a very low α , but it makes crummy triangles at low frequencies.

$$\text{tri}(x, P, D) = \begin{cases} 0 & x \leq 0 \\ \alpha \text{tri}(x-1, P, D) + \text{square}(x, P, D)/P & \text{otherwise} \end{cases} \quad \alpha = 0.9 \text{ seems necessary}$$

I'd scale this as $\text{tri}(\dots) \times 4 + 0.5$. Note that Triangle's frequency is twice what you'd expect. If you slowly scan through frequencies, you'll get one or two pops as even 0.9 is not enough to overcome certain sudden jumps due to numerical instability. Instead, you might try something like $\alpha = 1.0 - 0.1 \times \min(1, f/1000)$, where f is the frequency.

⁵⁶The leaky integrator is a common trick not only in digital signal processing but also in machine learning, where this pattern shows up as the learning rate in equations for reinforcement learning and neural networks.

7.3 Wave Shaping

Wave shaping is very simple: it's just mapping an incoming sound signal using a function. That is, a wave shaping function $f(t)$ would modify an existing sound $s(t)$ as $f(s(t))$.

Because wave shapers can have *arbitrary* functions, the mathematics involved could impose too high a computational cost if they have to be called for every single sample in a digital synthesizer. Thus one common approach is to use a **table-based waveshaper**. This is simply a high-resolution lookup table which gives the output value of the function $f(t)$ for every possible input value t .

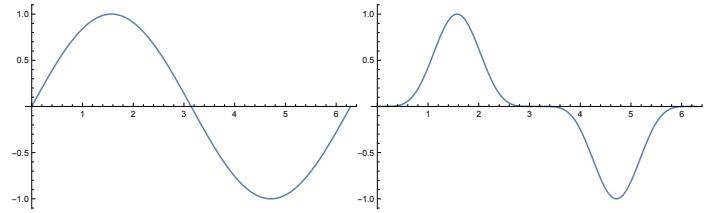


Figure 77 A Sine wave $\sin(t)$, and the result of waveshaping with the function x^5 , producing $\sin^5(t)$.

Waveshaping Polynomials You could use any function to shape an incoming signal if you wished, but it's common to use polynomials. This is because polynomials allow us to predict and control the resulting partials in a sound. Notably, a polynomial of order n can only generate harmonics up to n . Consider the polynomial x^5 applied to a sine wave of frequency ω and amplitude 1, as shown in Figure 77. Our waveshaped signal $w(t)$ would be:

$$w(t) = \sin^5(\omega t)$$

$$w(t) = 1/16 \times (10 \sin(\omega t) - 5 \sin(3\omega t) + \sin(5\omega t)) \quad \text{Trust me, there's a } \sin^5(\theta) \text{ identity}$$

This would create a harmonic at ω , a second at 3ω , and a third at 5ω , with the amplitudes indicated.

One common set of polynomials used in waveshaping are the **Chebyshev Polynomials of the First Kind**. This is a set of polynomials discovered by **Pafnuty Chebyshev** in 1854, and have the property that, for $x \in [-1, 1]$, their output only ranges from $[-1, 1]$, and so they're good for mapping a wave. Another interesting property is: if we wave-shape a sine wave with Chebyshev polynomial $T_n(x)$, we will produce *only* harmonic number n . This allows us to make a wave-shaping function which is the sum of several Chebyshev polynomials that builds sounds with exactly certain harmonics. So that's kind of neat.

Chebyshev polynomials of the first kind follow the following pattern. The first polynomial, $T_0(x)$, is 1. The second polynomial, $T_1(x)$, is x . After that, polynomials are defined recursively: $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. Thus the first seven polynomials are:

$$\begin{aligned} T_0(x) &= 1 \\ T_1(x) &= x \\ T_2(x) &= 2x^2 - 1 \\ T_3(x) &= 4x^3 - 3x \\ T_4(x) &= 8x^4 - 8x^2 + 1 \\ T_5(x) &= 16x^5 - 20x^3 + 5x \\ T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \end{aligned}$$

7.4 Wave Folding

Related to wave shaping is **wave folding**, made popular by **Don Buchla** and west-coast synthesis methods. If the incoming signal is greater than 1.0 or less than -1.0, a wave folder causes it to reflect back towards 0.0 again: it *folds* it back. Of course, if the signal is *much* greater than 1.0 (or -1.0) folding it will cause it to be beyond -1.0 (or 1.0), and so it will be *folded again*, and again, until it's inside the range -1...1. The basic, and simplistic, folding equation is recursive, but easy.

$$\text{Fold}(x) = \begin{cases} x > 1 & \text{Fold}(1 - (x - 1)) \\ x < -1 & \text{Fold}(-1 - (x + 1)) \\ \text{otherwise} & x \end{cases}$$

This particular wave folding equation introduces a great deal of harmonic complexity into a sound, including (typically) a lot of inharmonic partials; some wave folders produce a more rounded shape than shown in Figure 78 either directly or perhaps by applying a low-pass filter after the fact (see Section 8).

You could create even *more* inharmonic distortion using a related method called **wrapping**: here, if the sound exceeds 1.0, it's toroidally wrapped around to -1 (and vice versa). That is:

$$\text{Wrap}(x) = \begin{cases} x > 1 & (x \bmod 1) - 1 \\ x < 1 & 1 - (-x \bmod 1) \\ \text{otherwise} & x \end{cases}$$

This definition is carefully written such that $u \bmod 1$ is only performed on positive values of u : because different systems interpret mod differently for negative values. In Java you could implement $u \bmod 1$ as $u \% 1.0$ or simply as $u - (\text{int})u$ (u is floating point).

Finally, there remains the possibility of **clipping**: here, the sound is simply bounded to be between -1 and 1. This should be obvious:

$$\text{Clip}(x) = \begin{cases} x > 1 & 1 \\ x < -1 & -1 \\ \text{otherwise} & x \end{cases}$$

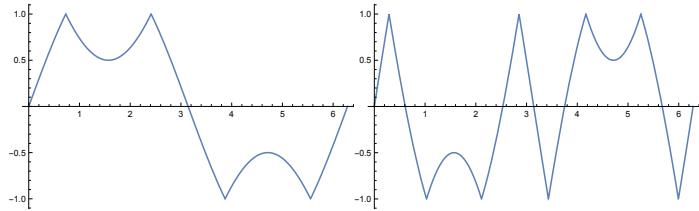


Figure 78 Wave folding the function $1.5 \sin(t)$ (left) and the function $3.5 \sin(t)$ (right). The right figure gives an example of recursive folding.

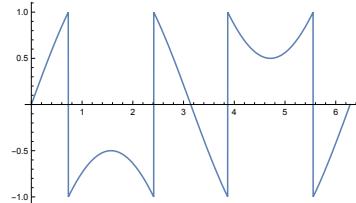


Figure 79 Wrapping the function $1.5 \sin(t)$.

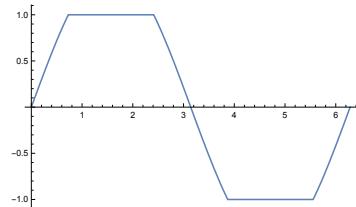


Figure 80 Clipping the function $1.5 \sin(t)$.

Dealing with Aliasing It shouldn't surprise you that these methods can alias like crazy. Much of the problem is due to the hard discontinuities that occur when these waves hit the 1.0 or -1.0 boundary. One cheap way to counter this, to *some* degree, is to round off these corners. For example, you could **compress** the amplitude as it got closer to the boundary, with a function something like:

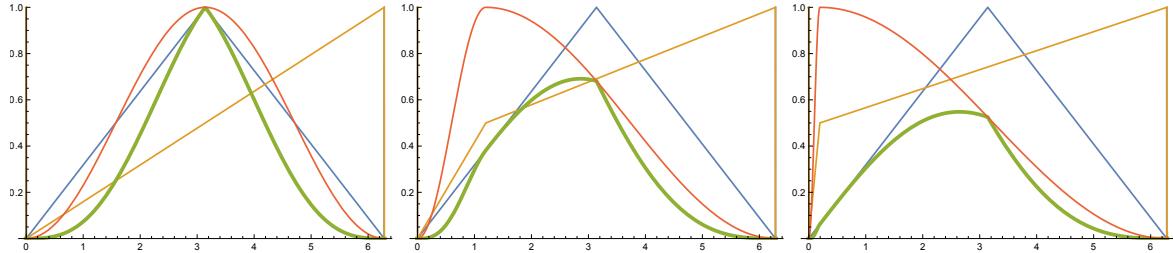


Figure 81 Phase distortion using $g(x, \alpha) = \frac{x}{\alpha}$ when $x < \pi\alpha$, else $\frac{x-\pi\alpha}{2-\alpha} + \pi$. This **g(...)** phase-distortion function is initially the identity function at $\alpha = 0$, then develops a knee at $\pi\alpha$ as x increases. This results in the **cosine wave** eventually distorting into a pseudo-sawtooth wave. If we windowed the result by a **triangle function**, we would produce the **resulting green wave**.⁵⁸

$$\text{Compress}(x) = \begin{cases} x \geq 0 & 1 - (1 - x)^a \\ x < 0 & (1 + x)^a - 1 \end{cases} \quad a = 2 \text{ seems reasonable}$$

You'll still get plenty of aliasing, but it's better than nothing.⁵⁷

7.5 Phase Distortion

Phase distortion, a sort of converse to waveshaping, was special to Casio's CZ series of synthesizers, from the diminutive (and cheap) **CZ-101** to their top of the line **CZ-1**. Whereas waveshaping modifies an existing wave $f(x)$ with a shaping function $g(x)$ as $g(f(x))$, phase distortion does it the other way around, that is, $f(g(x))$. In phase distortion, $f(x)$ is normally a sinusoid like sine or cosine, and so $g(...)$

may be thought of as modifying its **instantaneous phase**, hence the name. Phase distortion is often incorrectly associated with **Phase Modulation**, a variant of **Frequency Modulation** discussed in Section 9 which also modifies phase: but they really are pretty different creatures.

If you use a sinusoid $f(...)$ for waveshaping—which passes through 0—and your $g(...)$ function is such so that $\lim_{x \rightarrow 0} g(x) = 0 \bmod 2\pi$, then you'll have a smooth, cyclic wave resulting from waveshaping with $g(f(x))$. But this isn't the case the other way around, that is, doing $f(g(x))$ even when $f(...)$ is sinusoidal (as it normally is in phase distortion). Depending on your choice of $g(...)$, you can get all sorts of discontinuities as x approaches the periodic 2π boundary. So in order to guarantee a smooth, cyclic function, phase distortion runs the result through a **window** $w(...)$, multiplying it by some function which is 0 at 0 and 2π .

Let's put this all together. To make things easy to visualize, for $f(...)$ I'll use a negative cosine scaled to range from 0 to 1 rather than from -1 to 1. Let's call this "pcos", as in $\text{pcos}(x) = \frac{1-\cos(x)}{2}$. This is the red curve in the left subfigure of both Figures 81 and 83. Phase distortion then outputs the waveform $\text{PhaseDistort}(t)$ using the following equation.



Figure 82 Casio CZ-1.⁵⁴

⁵⁷For a proper treatment of how to deal with antialiasing in wave folding, see Fabian Esqueda, Henri Pöntynen, Julian Parker, and Stefan Bilbao, 2017, Virtual analog models of the Lockhart and Serge wavefolders, *Applied Sciences*, 7, 1328.

⁵⁸This example, minus the triangle windowing, is more or less the example provided in the CZ series user manuals.

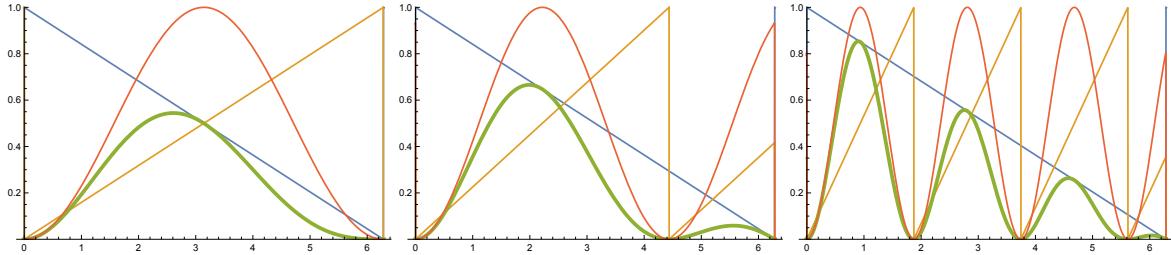


Figure 83 Phase distortion using $g(x, \alpha) = \frac{x \bmod 2\pi\alpha}{\alpha}$. This **g(...)** phase-distortion function is again initially the identity function at $\alpha = 0$, then increases in frequency π as α increases, but resets at 2π . This results in the **cosine wave** increasing in frequency but likewise resetting at 2π . If we windowed the result by a **sawtooth function**, which would eliminate the discontinuity due to the reset, we would produce the **resulting green wave**.⁶¹

$$\begin{aligned} \text{PhaseDistort}(t) &= \text{pcos}(g(x, \alpha)) \times w(x) \\ x &= t \bmod 2\pi \end{aligned}$$

Notice that we're passing $\alpha \in [0, 1]$ into $g(...)$. This lets us specify *degree* of phase distortion we want, generally modulated by an envelope.⁵⁹ How $g(...)$ maps the distortion varies from function to function. The CZ series provided a range of $g(...)$ functions and $w(...)$ window options.⁶⁰

Figure 81 shows a sinusoid being gradually distorted by a $g(...)$, changing according to α . Eventually the sinusoid is distorted into a pseudo-sawtooth. Multiplying this by a triangle window function produces an interesting final wave. Without an optional window function, the distortion function acts as a kind of **quasi-filter**, stripping out the high frequencies of the sawtooth until we're left with a simple one-harmonic sinusoid. We'll get to true filters in Section 8. Phase distortion's quasi-filters in combination with windowing can also provide some degree of **resonance**, as shown in another example, Figure 83.

7.6 Combining

A subtractive synthesizer often has several oscillators per voice. The output of these oscillators is **combined**, and the resulting sound is then run through various shaping methods. There are many different ways that these oscillators could be combined: we'll cover a few of them next.

Mixing The most straightforward way to combine two or more sounds would be to **mix** them: that is, to add them together, multiplied by some weights. For example, if we had n sounds $f_1(t) \dots f_n(t)$, each with a weight $\alpha_1 \dots \alpha_n$ (all $\alpha_i \geq 0$), our mixer might be simply $m(t) = \sum_i \alpha_i f_i(t)$. It's also common to play the weights off each other so as not to exceed the maximum volume and to allow for an easy modulation parameter. For example, we might **cross fade** one sound into a second with a single α value as $m(t) = \alpha f_1(t) + (1 - \alpha) f_2(t)$. More generally, a cross-fading mixer for some n sounds might be $m(t) = \frac{\sum_i \alpha_i f_i(t)}{\sum_i \alpha_i}$. Of course, here not all α_i can be zero.

⁵⁹On the CZ series, this was an elaborate eight-stage envelope which could do really nifty things.

⁶⁰Casio seemingly went to great lengths to obscure how all this worked in their synth interfaces. Windowing was not discussed at all, and the front panel had only a limited set of options. To get the full range of combinations of wave and window functions required undocumented MIDI sysex commands. See <http://www.kasploosh.com/projects/CZ/11800-spelunking/> for an extended discussion of how to do this.

⁶¹This is more or less the example provided in Figures 18–20 of Casio's patent on PD. Masanori Ishibashi, 1987, Electronic musical instrument, US Patent 4,658,691, Casio Computer Co., Ltd., Assignee.

Ring and Amplitude Modulation Other combination mechanisms produce complex sounds resulting from combining two incoming sources. Whereas in mixing, the two sound sources were essentially added, in **ring modulation**,⁶² the two sources are *multiplied* against one another. That is, the resulting sound is:

$$r(t) = f(t) \times g(t)$$

Ring modulation is so named because of how it is historically implemented in circuitry: using a ring of diodes as shown in Figure 84.

A very closely related procedure occurs when one sound source— $f(t)$, say—is used to change the *amplitude* of the other source $g(t)$. This is known as **amplitude modulation**. To do this, $g(t)$ is interpreted as a wave that goes from 0...2 rather than from −1... +1, by adding 1 to it. So we have:

$$a(t) = f(t) \times (g(t) + 1)$$

Let's consider the effect of ring and amplitude modulation in a very simple case, using sine waves as our sound sources, with a_f and a_g as amplitudes and ω_f and ω_g as frequencies respectively. Then we have $f(t) = a_f \sin(\omega_f t)$ and $g(t) = a_g \sin(\omega_g t)$.

The ring-modulated signal $r(t)$ would look like this:

$$\begin{aligned} r(t) &= f(t) \times g(t) \\ &= a_f \sin(\omega_f t) a_g \sin(\omega_g t) \\ &= a_f a_g \sin(\omega_f t) \sin(\omega_g t) \\ &= 1/2 a_f a_g (\cos(\omega_f t - \omega_g t) - \cos(\omega_f t + \omega_g t)) \\ &= 1/2 a_f a_g \cos((\omega_f - \omega_g)t) - 1/2 a_f a_g \cos((\omega_f + \omega_g)t) \end{aligned} *$$

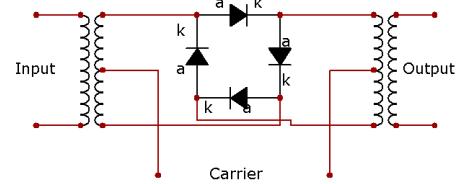


Figure 84 A ring modulation circuit.^{©55}

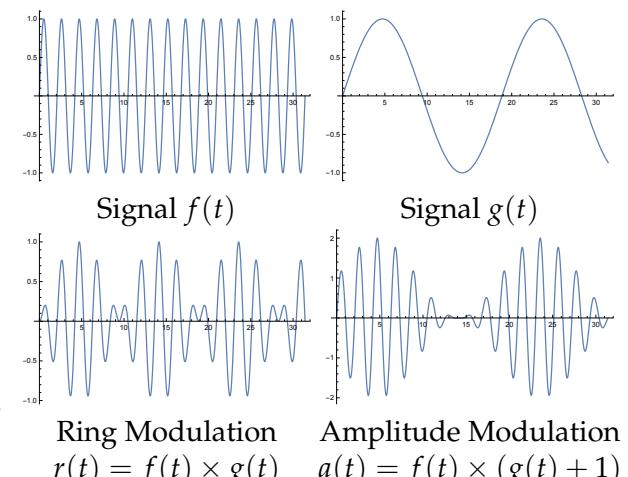


Figure 85 The effects of ring and amplitude modulation when given two sine wave signals, $f(t)$ and $g(t)$.

⁶²This isn't exactly "modulation" in "modulators" sense as discussed earlier.

*This transformation comes from the cosine identities

$$\begin{aligned} \cos(A + B) &= \cos A \cos B - \sin A \sin B && \text{and} \\ \cos(A - B) &= \cos A \cos B + \sin A \sin B \end{aligned}$$

Subtract these two and we get...

$$\begin{aligned} \cos(A - B) - \cos(A + B) &= (\cos A \cos B + \sin A \sin B) - (\cos A \cos B - \sin A \sin B) \\ &= 2 \sin A \sin B \\ \sin A \sin B &= (\cos(A - B) - \cos(A + B))/2 \end{aligned}$$

Recall that the original signals were sine waves, and thus had one partial each at frequencies ω_f and ω_g respectively. The new combined sound also consists of two partials (the two cosines⁶³), one at frequency $\omega_f - \omega_g$ and one at frequency $\omega_f + \omega_g$. The original partials are gone. These two new partials are called **sidebands**. What happens if $\omega_f - \omega_g < 0$? Then the partial is “reflected” back: so we just get $|\omega_f - \omega_g|$.

Now, let’s consider amplitude modulation. Here, $f(t)$ will be our primary signal (the **carrier**), and $g(t)$ will be the **modulator**, the signal that changes the amplitude of $f(t)$. Using the same tricks as we did for ring modulation, we have:

$$\begin{aligned} a(t) &= f(t) \times (g(t) + 1) \\ &= a_f \sin(\omega_f t)(a_g \sin(\omega_g t) + 1) \\ &= a_f a_g \sin(\omega_f t) \sin(\omega_g t) + a_f \sin(\omega_f t) \\ &= 1/2 a_f a_g (\cos(\omega_f t - \omega_g t) - \cos(\omega_f t + \omega_g t)) + a_f \sin(\omega_f t) \\ &= 1/2 a_f a_g \cos((\omega_f - \omega_g)t) - 1/2 a_f a_g \cos((\omega_f + \omega_g)t) + a_f \sin(\omega_f t) \end{aligned}$$

Here $a(t)$ consists of not two but *three* partials: the same $\omega_f - \omega_g$ and $\omega_f + \omega_g$ sidebands as ring modulation, plus the original carrier partial ω_f . So amplitude modulation is in some sense just ring modulation, mixed with the original carrier to form the final sound.⁶⁴

Keep in mind that this fine for just two sine waves: but as soon as the mixed sounds become more sophisticated, the resulting sounds can get more complex.

Frequency Modulation While amplitude modulation allows a modulating signal to change the *amplitude* of a carrier signal, Frequency Modulation (or FM) allows a modulating signal to change the *frequency* of a carrier signal. Frequency modulation is an important subject and has spawned an entire family of synthesis methods all on its own. It’ll be discussed in detail in Section 9.

Sync One last combination mechanism, if it can be called that, is to **sync** one oscillator wave to the other. Syncing forces oscillator A to reset whenever B has completed its **period**. There are many ways that an oscillator can reset; Figure 86 shows two common ones. **Hard sync** causes the oscillator to simply restart its period, that is, reset to position 0. Hard sync is very common in analog synthesizers and has a distinctive sound. **Reversing soft sync** causes the oscillator to *reverse its wave*, creating a mirror image. Some analog synthesizers do not support hard sync but might support reversing soft sync or perhaps some other kind of **soft sync**.

The point of sync methods is that they radically reshape triangle or sawtooth waves, stopping them cold and resetting them. This produces sounds with complex and often non-harmonic partials, adding to our collection of interesting material to “subtract” from via filters.

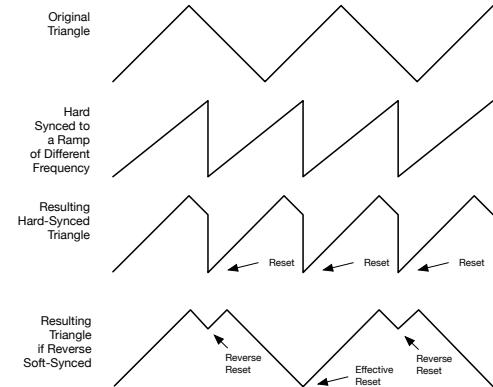


Figure 86 Hard and Reversing Soft Sync. A Triangle wave is hard- and reverse-soft-synced to a Sawtooth wave of a different frequency.

⁶³Sin and Cos only differ from each other in phase: but recall, phase doesn’t matter much to us.

⁶⁴This shouldn’t be at all surprising given that ring modulation is $r(t) = f(t) \times g(t)$ and amplitude modulation is $a(t) = f(t) \times (g(t) + 1) = f(t) \times g(t) + f(t) = r(t) + f(t)$.

Sync will introduce high degrees of aliasing, and so it is not easy to deal with. If you are interested, there exists a method called **minBLEP**⁶⁵ which can be used to implement sync in a bandlimited fashion. You can also use this technique to generate a variety of other bandlimited waveforms, including square and sawtooth.

7.7 Amplification

There's really nothing special to say about amplification: it's just multiplying the amplitude (volume) of the signal $f(t)$ by some constant $a \geq 0$, resulting in $g(t) = a \times f(t)$. The value a can be anything, even $a < 1$, and so amplification is kind of a misnomer: an amplifier can certainly make the sound *quieter*. In fact, an amplifier can be used to shut off a sound entirely. Analog amplifiers are classically controlled via a voltage level, and so they are often known as **voltage controlled amplifiers** or **VCAs**.

If an amplifier is so boring, why is it a module in a subtractive synthesis chain? Because you can make a sound more realistic or interesting by *changing its amplitude in real time*. For example many musical instruments start with a loud splash, then quiet down for their steady state, and then fade away when the musician stops playing the note. Thus an amplifier module is important in conjunction with time-varying modulation.

There are two common modulation mechanisms used for amplification. First, the musician may specify the amplification of a voice through the **velocity** with which he strikes the key. **Velocity sensitive** keyboards are discussed in Section 12.1. Second, a VCA is normally paired with an **envelope**, often **ADSR**, which defines how the volume changes over time. This is so common and useful that a great many synthesizers have dedicated ADSR envelopes solely for this purpose.

⁶⁵Eli Brandt, 2001, Hard sync without aliasing, in *International Computer Music Conference*. BTW, for additional techniques for bandlimited wave manipulation, **Vesa Välimäki** is a name which crops up a lot; you might google for him.

8 Filters

Filters put the “subtractive” in subtractive synthesis, and so they are absolutely critical to the behavior of a subtractive synthesizer. Unfortunately they are also by far the most complex element in a subtractive synthesis pipeline. Filters have a rich and detailed mathematical theory, and we will only just touch on it here.

A filter takes a sound and modifies its partials, outputting the modified sound. It could modify them in two ways: (1) it could adjust the **amplitude** of certain partials, or (2) it could adjust their **phase**. The degree to which a filter adjusts the amplitude or phase of partials depends on their frequency, and so the overall the behavior of the filter on the signal is known as its **frequency response**. This is, not surprisingly, broken into two behaviors, the **amplitude response** and the **phase response** of the filter. Because humans can’t hear differences in phase, we’re usually interested in the amplitude response and will focus on it here; but there are interesting uses for the phase response which we will come to later starting in Section 11.4.

A filter can describe many functions in terms of amplitude (and phase) response, but there are certain very common ones:

- A **low pass** (LP) filter doesn’t modify partials below a certain **cutoff frequency**, but beyond that cutoff it begins to decrease their amplitude. This drop-off is logarithmic, so if you see it on a log scale it looks like a straight line: see Figure 87 (A). A low pass filter is by far the most common filter in synthesizers: so much so that many synthesizers *only* have a low pass filter.
- A **high pass** (HP) filter is exactly the opposite: it only decreases the amplitude of partials if they’re *below* the cutoff frequency. See Figure 87 (B).
- A **band pass** (BP) filter is in some sense a combination of low pass and high pass: it decreases the amplitude of partials if they’re on either side of the cutoff frequency: thus it’s “picking out” that frequency and shutting off the others. See Figure 87 (C).⁶⁶
- A **notch** filter is the opposite of a band pass filter: it decreases the amplitude of partials if they’re *at* the cutoff frequency. See Figure 87 (D).⁶⁷

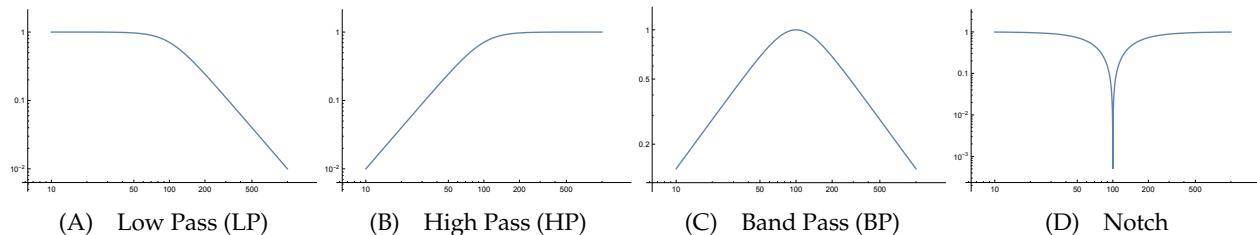


Figure 87 Amplitude response by frequency of four common filters, with a cutoff at 100. The axes are on a log scale. Thus the notch is really more or less an inverted band pass, but looks quite different because of the log scaling.

⁶⁶This term is also used more broadly to describe a filter which passes through a range of frequencies rather than just one. Unlike for a notch filter, I don’t think there are different terms to distinguish these two cases.

⁶⁷A notch filter is a degenerate case of a **band reject** filter, which rejects a certain range of frequencies rather than a specific one.

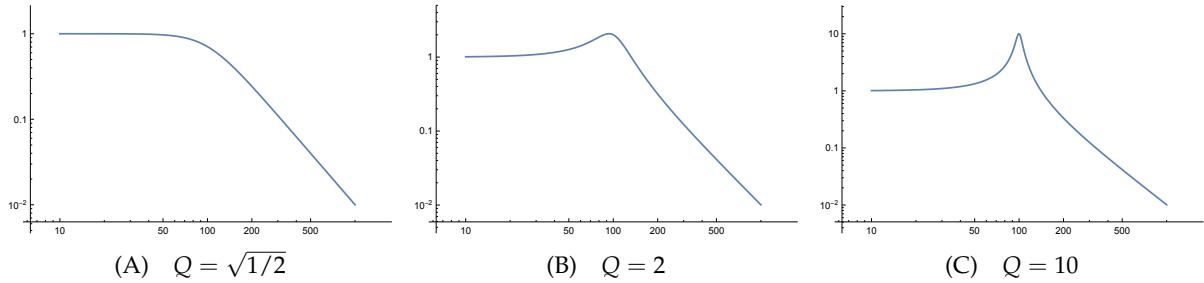


Figure 88 Low-pass filter with a cutoff at 100 and varying amounts of resonance (Q). Note that axes are on a log scale.

These aren't all the filters you can build, not by a long shot. Another common filter is the **comb filter**, discussed at length in Section 11. Another important filter is the notional **brick wall** low pass filter: essentially a very, very steep low pass frequencies above Nyquist.

Phase The filters above are largely distinguished by how they modify the amplitude of the partials: but it doesn't say anything about how they modify phase. In fact, usually when building the four filters above our goal would be to not modify the phase at all,⁶⁸ or at the very least, to shift the phase by the same amount for *all* partials. Filters for the second case are called **linear phase** filters. But there do exist filters *designed* to adjust phase of partials in different ways. The most common subclass of filters of this type are the strangely-named **all pass** (AP) filters. As their name would suggest, these don't modify the amplitude at all; their purpose is solely to shift the phase. We'll see all pass filters more in Section 11.

Gain In modifying the amplitude or phase of a sound, filters often will inadvertently amplify the overall volume of the sound as well. The degree of amplification is called the **gain** of the filter. It's not a big deal that a filter has a gain if we know what it is: we could just amplify the signal back to its original volume after the fact. But it's convenient to start with a filter that makes no modification to the volume, that is, its gain is 1. We call this a **unity gain filter**.

Order and Resonance Filters are distinguished by the number of **poles** and **zeros** they have, which in turn determines their **order**. We'll get back to what these are in a while, but for now it's helpful to know two facts. First, the number of poles can determine how steep the dropoff is: this effect is called the **roll-off** of the filter. In the synthesizer world you'll see filters, particularly low pass filters, described in terms of their roll-off either by the number of poles or by the steepness of the curve. A first-order low pass filter is typically described as **one pole** and will have a roll-off of 6dB per octave (that is, it drops by 6dB every time the frequency doubles). A second-order low pass filter is described as **two pole** and will have a roll-off of 12dB per octave. And a fourth-order low pass filter, normally described as **four pole**, will have a roll-off of 24dB per octave. This is illustrated in Figure 89.

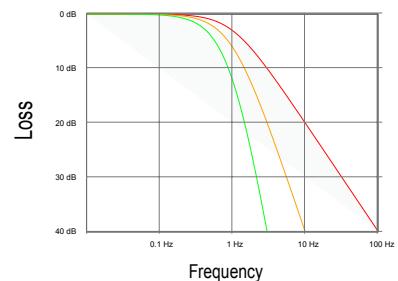


Figure 89 Roll-off of typical low pass filters: **one-pole** (6dB), **two-pole** (12dB), and **four-pole** (24dB).⁵⁶

⁶⁸If your filter is working in real-time, as is the case for a synthesizer, it's not possible to avoid modifying the phase: so you need to fall back to a linear phase filter.

Second or higher order filters can be constructed to exhibit a curious behavior: just before the cutoff point, they will *increase* the amplitude of partials. This is **resonance**, as shown in Figure 88, and creates a squelching sound. The degree of resonance is known as the **quality factor** of the filter and is defined by the value Q . A value of $Q = \sqrt{1/2} \approx 0.707$ has no resonance, and higher values of Q create more resonance. A first-order (“one pole”) filter cannot have resonance.

8.1 Digital Filters

Traditionally filters in electrical engineering are built from capacitors, resistors, inductors, and so on, and so take a continuous analog signal both as input and output. But when we build a filter in software, we’ll be modifying a *digital* signal. This is a quite different process.

One conceptually simple way to make a filter would be to take a digital signal, convert it into the Fourier domain with an FFT, manually change (through multiplication) the amplitudes and phases we want, then convert it back into the time domain with an inverse FFT. But it turns out you can do essentially the same process while staying in the time domain through a procedure called **convolution**.

In convolution, we construct a new signal $y(n)$ from the original signal $x(n)$ by mixing each sample with a bit of its neighboring samples. For example, for each timestep n ,⁶⁹ we might take the current sample $x(n)$, plus some previous samples $x(n-1)$, $x(n-2)$, etc., and build $y(n)$ as the weighted sum of them, as in: $y(n) \leftarrow b_0x(n) + b_1x(n-1) + \dots + b_kx(n-k)$. The constants b_0, b_1, \dots, b_k are determined by us beforehand, and they can be positive or negative. This process is repeated, in real time, for every single sample coming out of $x(\dots)$.

Why does this result in a filter? Consider the equation $y(n) = 1/2x(n) + 1/2x(n-1)$, that is, $b_0 = 1/2$ and $b_1 = 1/2$. This is averaging each sample with the sample before it. If your sound was just a low-frequency sine wave, this wouldn’t have much of an effect, since each sample would be similar to its immediate predecessor. But if you had a very *high* frequency sine wave, then each successive sample would be very different, and this averaging procedure is essentially using them to nullify one another. Thus this is a simple low-pass filter. It is *smoothing* the signal, eliminating the high frequencies.

A filter of this type is called a **Finite Impulse Response** or **FIR** filter. The filter is *finite* because if $x(n)$ is (say) 1.0 at $n = 0$ but 0.0 for $n > 0$ thereafter—that’s the impulse—the filter will exhibit some interesting value for $y(n=0)$ and $y(n=1)$, but after that, it’s always $y(n > 1) = 0.0$. It’s common to describe this filter using a filter diagram, as shown in Figure 92.

The output of the filter $y(n) = b_0x(n) + b_1x(n-1)$ is entirely based on the current input $x(n)$, plus the input $x(n-1)$ exactly *one* timestep prior. Because we only need information one timestep back, this is known as a **first-order filter**.⁷⁰

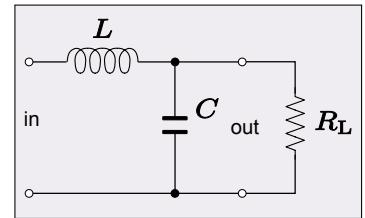


Figure 90 Simple second-order low pass analog filter. No, we’re not doing circuits.

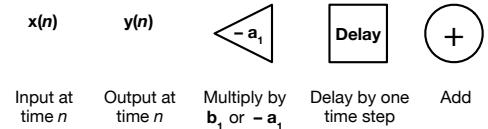


Figure 91 Symbols used in the filter diagrams.

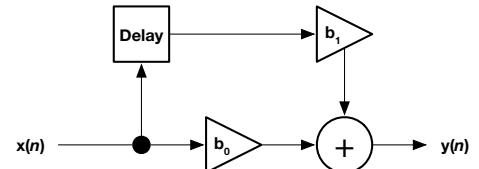


Figure 92 First-order Finite Impulse Response (FIR) filter. See Figure 91 for explanation of these symbols.

⁶⁹As a computer science guy, I’d use t for time, but n is the electrical engineering convention.

⁷⁰If you increase the length of the delay, this becomes a **feedforward comb filter**. We’ll discuss that more in Section 11.

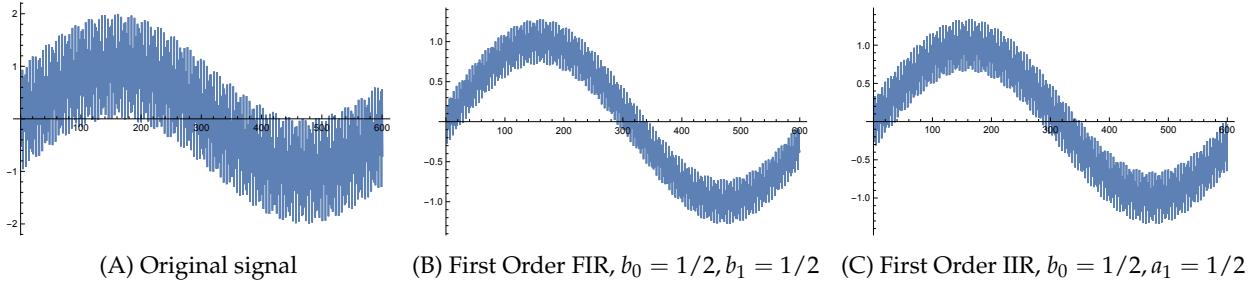


Figure 94 Effects of the first-order digital filters from Figures 92 and 93 on a signal consisting of two sine waves: $f(x) = \sin(x/100) + \sin(x \times 10)$. The high-frequency sine wave is tamped down, but the low-frequency wave is preserved.

Now consider the filter in Figure 93. The output of this filter is specified a little differently. It's $y(n) = b_0x(n) - a_1y(n - 1)$. This is a simplistic form of an **Infinite Impulse Response** or **IIR** filter.⁷¹ This filter is *infinite* in the sense that if $x(n)$ is (say) 1.0 at $n = 0$ but 0.0 for $n > 0$ thereafter—the impulse—the filter will may continue to exhibit some non-zero value for $y(n)$ forever, because it's organized as a kind of feedback loop. It shouldn't surprise you that because of the feedback loop this kind of filter can be unstable. This filter is also a first-order filter because, once again, we're only going back in time by 1 (this time, by $y(n - 1)$ rather than $x(n - 1)$).

Note that the a_1 is negative: we'll get back to that.

Figure 94 shows the effect of these two filters on a signal consisting of the mixture two sine waves, a low-frequency sine wave and a high-frequency one. As you can see, they are effective at tamping down the high frequency sine wave while preserving the low-frequency one. Note however that they also have changed the overall amplitude of the signal: the degree to which a filter does this is called the **gain** of the filter. Ideally we'd like a **unity gain filter**, that is, one which doesn't muck with the amplitude; but if we must, we can live with it and just re-amplify the signal ourselves after the fact.

Higher Order Filters A second order filter requires information two steps back. The second order extension of the FIR filter shown earlier is $y(n) = b_0x(n) + b_1x(n - 1) + b_2x(n - 2)$, and the second order extension of the simple IIR filter shown earlier is $y(n) = b_0x(n) - a_1y(n - 1) - a_2y(n - 2)$.

You can easily diagram these filters by stacking up delay modules, as shown in Figure 95. Indeed, you can make **third-order**, **fourth-order**, and in general **n th-order** filters by continuing to stack up delay modules in the same pattern.

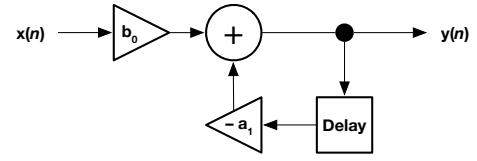


Figure 93 First-order simple Infinite Impulse Response (IIR) filter.

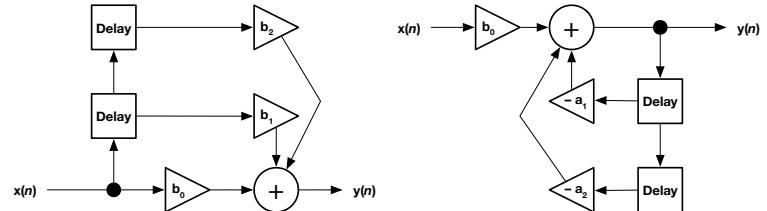


Figure 95 Second-order filters

⁷¹If you increase the length of the delay, this simple filter becomes a **feedback comb filter**. We'll discuss that more in Section 11.

The full Infinite Impulse Response (IIR) filter consists of *both* the FIR and the basic IIR filters shown so far. The general pattern for a second order IIR filter is shown in Figure 96 and is known as the **Direct Form I** of a digital IIR filter.⁷²

This diagram corresponds to the equation

$$y(n) = b_0x(n) + b_1x(n - 1) + b_2x(n - 2) \\ - a_1y(n - 1) - a_2y(n - 2)$$

Here's one reason why the a_i values are defined negatively: because it allows us to rearrange the equation so that all the y elements are on one side and all the x elements are on the other, showing the symmetry of the thing:

$$y(n) + a_1y(n - 1) + a_2y(n - 2) \\ = b_0x(n) + b_1x(n - 1) + b_2x(n - 2)$$

Cute! In general, we have filters of the form:

$$y(n) + a_1y(n - 1) + \dots + a_Ny(n - N) = b_0x(n) + b_1x(n - 1) + \dots + b_Mx(n - M)$$

This is pretty simple to implement in an algorithm. One note however: when computing our first few samples, the algorithm relies on "previous" samples which don't exist yet. For example, to compute $y(0)$, we may need $x(-1)$. What we'll do is define those to be zero. This is called **zero padding**. And we have:

Algorithm 18 Initialize a Digital Filter

- 1: $N \leftarrow$ number of y delays
- 2: $M \leftarrow$ number of x delays

- 3: Global $y \leftarrow \langle y_1 \dots y_N \rangle$ array of N real values, initially all zero ▷ Note: 1-based array
- 4: Global $x \leftarrow \langle x_1 \dots x_M \rangle$ array of M real values, initially all zero ▷ Note: 1-based array

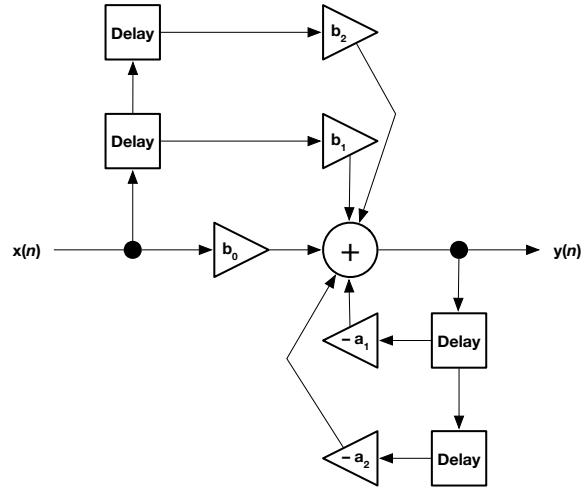


Figure 96 A full second-order IIR filter (Direct Form I).

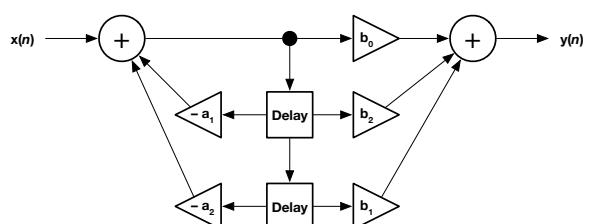


Figure 97 Full second-order IIR filter (Direct Form II).

⁷²There are other forms than just Direct Form I.

For example, consider **Direct Form II** is shown in Figure 97 at right. This form rearranges things to reduce the number of delays. However, because there are two addition points, it's more complex dealing with issues such as overflow when using fixed-point arithmetic. We'll use Direct Form I.

Algorithm 19 Step a Digital Filter

```

1:  $a \leftarrow \langle a_1 \dots a_N \rangle$  array of  $N$  real values           ▷ Note: 1-based array
2:  $b \leftarrow \langle b_1 \dots b_M \rangle$  array of  $M$  real values           ▷ Note: 1-based array
3:  $b_0 \leftarrow$  real value                                         ▷ This is " $b_0$ ", by default 1
4:  $x_0 \leftarrow$  real value                                         ▷ Current input

5: Global  $y \leftarrow \langle y_1 \dots y_N \rangle$  array of  $N$  real values, initially all zero    ▷ Note: 1-based array
6: Global  $x \leftarrow \langle x_1 \dots x_M \rangle$  array of  $M$  real values, initially all zero    ▷ Note: 1-based array

7: sum  $\leftarrow x_0 \times b_0$ 
8: for  $n$  from 1 to  $N$  do
9:   sum  $\leftarrow$  sum  $- a_n \times y_n$ 
10: for  $m$  from 1 to  $M$  do
11:   sum  $\leftarrow$  sum  $+ b_m \times x_m$ 
12: for  $n$  from  $N$  down to 2 do                                         ▷ Note backwards
13:    $y_n \leftarrow y_{n-1}$ 
14:  $y_1 \leftarrow$  sum
15: for  $m$  from  $M$  down to 2 do                                         ▷ Note backwards
16:    $x_m \leftarrow x_{m-1}$ 
17:  $x_1 \leftarrow x_0$ 
18: return sum

```

8.2 Building a Digital Filter

So far we've only seen a low pass filter. But it turns out that with higher-order filters, and with the right choice of constants, we can create all sorts of filter designs. But at this point we still have no idea (1) what order our filter should be nor (2) what the constants $a_i \dots$ and $b_i \dots$ should be to achieve our goals. How do we figure this out? Here's the general process we'll follow.

1. First we determine the behavior of the filter we want. Though we're building a **digital filter**, we'll start by cooking up the requirements in the continuous realm, as if we were planning on building an **analog filter**.
2. We'll then choose the so-called **poles** and **zeros** of the analog filter in the **Laplace domain**, a complex-number space, which will achieve this behavior. The poles and zeros collectively define the **transfer function** of the filter.
3. We can verify the behavior pretty easily by using the poles and zeros to directly plot the amplitude and phase response. This is typically plotted using a **Bode plot**.
4. There is no exact conversion from a continuous (analog) filter to a digital filter: rather we will do an approximation. To do this, we start by first mapping the transfer function from the Laplace domain to a *different* complex-number space, the **Z domain**. The Z domain makes it easy to build a digital filter, but there is no bijection from Laplace coordinates to Z coordinates. Instead we'll use a popular conversion called the **bilinear transform** which will be good enough for our purposes.

5. Once the transfer function is in the Z domain, it's simple to extract from it the **coefficients** with which we will build the digital filter in software.
6. Alternatively you could skip the Laplace domain and just define the poles and zeros in the Z domain (and in fact designers do this as a matter of course). We'll also discuss that strategy.
7. We'll derive the transfer functions (in the Laplace and Z domains) of a popular **Butterworth filter** design which can be used in a basic subtractive synthesizer.

8.3 Transfer Functions in the Laplace Domain

The Laplace domain⁷³ is a 2D complex space which is used to describe the behavior of a *continuous* (not discrete) filter such as is found in an analog synthesizer. The x -axis is the real axis and the y -axis is the imaginary axis. A complex number in the Laplace domain is by convention called s .

To describe the behavior of a filter, we start with a **transfer function**. This function is simply the ratio between the input $X(s)$ to some kind of system and its output $Y(s)$, that is, $H(s) = \frac{Y(s)}{X(s)}$. Both $Y(s)$ and $X(s)$ are (for our purposes) polynomials in s . A filter is a system and so it has a transfer function: $X(s)$ describes the frequencies and phases of the input sound and $Y(s)$ is the output.

It turns out that we can use the transfer function to analyze the **frequency response** of the filter, that is, how a filter changes the amplitude and phase of a partial of any given frequency. To do this, we first need to understand that the Laplace domain doesn't use the same units of measure for frequency as we do. We're using Hertz, that is, cycles per second; but we'll need to convert that to **angular frequency** (in radians per second). It's easy: angular frequency $2\pi\omega = 1$ Hz. So 1000 Hz is $2\pi \times 1000$ radians per second

Next we need to understand that in the Laplace domain, frequency is the *imaginary* component of the complex number s (the real component is zero).⁷⁴ So we'll use $s = i\omega$.

Example. If we want to plug $f = 1000$ Hz into our transfer function, we'll use

$$H(s) = H(i\omega) = H(i 2\pi f) = H(i 2\pi 1000) \approx H(i 6283.185) \approx \frac{Y(i 6283.185)}{X(i 6283.185)}$$

The output of H is a complex number which describes *both* the change in phase *and* in amplitude of the given angular frequency. Importantly, if we wanted to know the **amplitude response** of the filter, that is, how our filter would amplify a partial at a given angular frequency ω , we compute the **magnitude**⁷⁵ of $|H(i\omega)| = \left| \frac{Y(i\omega)}{X(i\omega)} \right| = \frac{|Y(i\omega)|}{|X(i\omega)|}$.

Example. If $H(s) = \frac{s^2 - 4}{s^2 + s + 2}$, and we wanted to know the amplitude change at frequency $1/\pi$ Hz (I picked that to make it easy: $1/\pi$ Hz is $\omega = 2$), we could do:

$$|H(s)| = |H(i\omega)| = |H(2i)| = \frac{|(2i)^2 - 4|}{|(2i)^2 + 2i + 2|} = \frac{|-4 - 4|}{|-4 + 2i + 2|} = \frac{|-8|}{|-2 + 2i|} = \frac{8}{\sqrt{(-2)^2 + (2)^2}} = \sqrt{8}$$

⁷³The Laplace domain is called a *domain* for a reason: it's closely related to the Fourier domain. But don't let that confuse you here.

⁷⁴By the way, I'm using i throughout this section to denote imaginary numbers as is customary, but electrical engineers use j because they already use i for electrical current.

⁷⁵The magnitude of a real number is just its absolute value. The magnitude of a complex number $a + bi$ is $\sqrt{a^2 + b^2}$.

8.4 Poles and Zeros in the Laplace Domain

Given our transfer function $H(s) = \frac{Y(s)}{X(s)}$ we can determine the behavior of the filter from the roots of the equations $X(s) = 0$ and $Y(s) = 0$ respectively. The roots of Y are called the **zeros** of the transfer function, because if s was equal to any of the roots of $Y(s) = 0$, all of $H(s)$ would be equal to zero. Similarly, if s was a root of $X(s) = 0$, then $H(s)$ would be a fraction with zero in the denominator and thus go to infinity. These roots are called the **poles** of the transfer function, because they make the equation surface rise up towards infinity like a tent with a tent pole under it.

Example. Let's try extracting the poles and zeros. We factor the numerator and denominator of the following transfer function:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{2s^2 + 2s + 1}{s^2 + 5s + 6} = \frac{(s + (\frac{1}{2} + \frac{1}{2}i))(s + (\frac{1}{2} - \frac{1}{2}i))}{(s + 3)(s + 2)}$$

From this we can see that the roots of $Y(s) = 0$ are $-\frac{1}{2} - \frac{1}{2}i$ and $-\frac{1}{2} + \frac{1}{2}i$ respectively, and the roots of $X(s) = 0$ are -3 and -2 respectively. The factoring process looks like magic, but it's just the result of the quadratic formula, which you no doubt learned in grade school: for a polynomial of the form $ax^2 + bx + c = 0$ the roots are $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$.

Example. Let's try another example:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{1}{5s - 3} = \frac{1}{5(s - \frac{3}{5})}$$

Thus there are *no* roots of $Y(s)$, and the single root for $X(s)$ is $\frac{3}{5}$.

Finding roots gets hard for higher-order polynomials, but thankfully we won't have to do it! Instead, to design a filter we'd often *start* with the roots we want—the zeros and poles based on the desired filter behavior—and then just multiply them to create the transfer function polynomials.

8.5 Amplitude and Phase Response

If we already have the roots we want, it's really easy to determine what the filter does to the phase and amplitude of a partial. Recall that the magnitude (**amplitude response**) of a filter is $|H(i\omega)| = \frac{|Y(i\omega)|}{|X(i\omega)|}$. If we have factored Y and X into their poles p_1, \dots, p_n and zeros z_1, \dots, z_m , then this is just

$$|H(i\omega)| = \frac{|Y(i\omega)|}{|X(i\omega)|} = \frac{|(i\omega - z_1)(i\omega - z_2) \dots (i\omega - z_m)|}{|(i\omega - p_1)(i\omega - p_2) \dots (i\omega - p_n)|} = \frac{\prod_j |(i\omega - z_j)|}{\prod_k |(i\omega - p_k)|}$$

We can also compute the **phase response**—how much the phase shifts by—as

$$\angle H(i\omega) = \sum_j \angle(i\omega - z_j) - \sum_k \angle(i\omega - p_k)$$

Remember that the magnitude of a complex number $|a + bi|$ is $\sqrt{a^2 + b^2}$ and its angle $\angle(a + bi)$ is $\tan^{-1} \frac{b}{a}$.

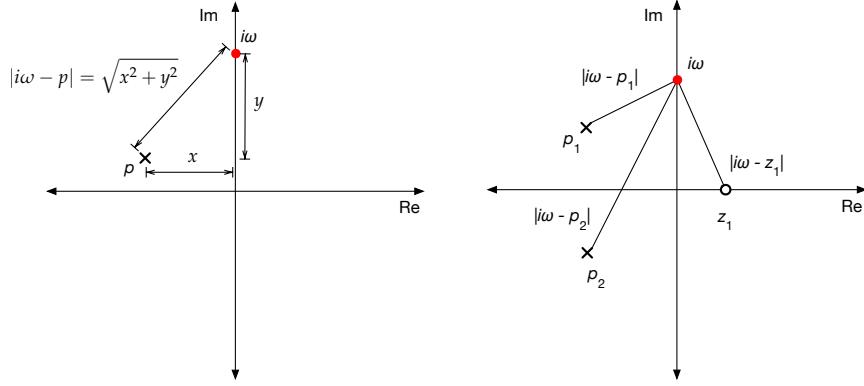


Figure 99 (Left) relationship between a pole p , the current frequency $i\omega$, and its impact on the magnitude of the amplitude at that frequency. (Right) two poles and a zero and their respective magnitudes. In this example, the amplitude at $i\omega$ is $|i\omega - p_1| \times |i\omega - p_2| \times 1/|i\omega - z_1|$.

Example. Given our previous poles and roots, the magnitude of $H(2i)$ is:

$$\begin{aligned} |H(2i)| &= \frac{|2i - (-\frac{1}{2} - \frac{1}{2}i)| \times |2i - (-\frac{1}{2} + \frac{1}{2}i)|}{|2i - (-3)| \times |2i - (-2)|} \\ &= \frac{\left|\frac{1+5i}{2}\right| \times \left|\frac{1+3i}{2}\right|}{|3+2i| \times |2+2i|} \\ &= \frac{\frac{\sqrt{26}}{2} \frac{\sqrt{10}}{2}}{\sqrt{13}\sqrt{8}} = \frac{\sqrt{5}}{4} \end{aligned}$$

In fact, we can easily plot the magnitude of the filter for any value of ω , as shown in Figure 98. The plots shown are a classic **Bode plot** of the amplitude and phase response. Note that the x axis is on a log scale, and (for the amplitude plot up top) the y axis is also on a log scale.

It's easy to conceptualize all this by considering a complex plane as shown in Figure 99. The current frequency $i\omega$, is a positive point on the imaginary (y) axis. As frequency sweeps from low to high, the scaling effect of a pole p on the magnitude at frequency $i\omega$ is simply the distance between them. Similarly, the scaling effect of a zero z on a magnitude at the frequency is $1/distance$. These effects are multiplied together for all the zeros and poles.

Thus if we know the poles and zeros of our filter, we can compute the amplitude change and the phase change for any frequency in the signal to which the filter is applied.

8.6 Pole and Zero Placement in the Laplace Domain

How do you select poles and filters that create a desired effect? This is a complex subject: here we will only touch on a tiny bit of it to give you a bit of intuitive feel for the nature and complexity of the problem. First, some rules:

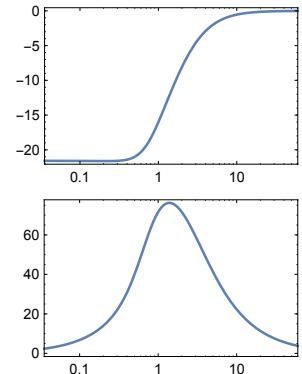


Figure 98 Bode plot of the amplitude response (top) and (awful) phase response (bottom) of a filter with zeros $-\frac{1}{2} - \frac{1}{2}i$ and $-\frac{1}{2} + \frac{1}{2}i$, and with poles -3 and -2 .

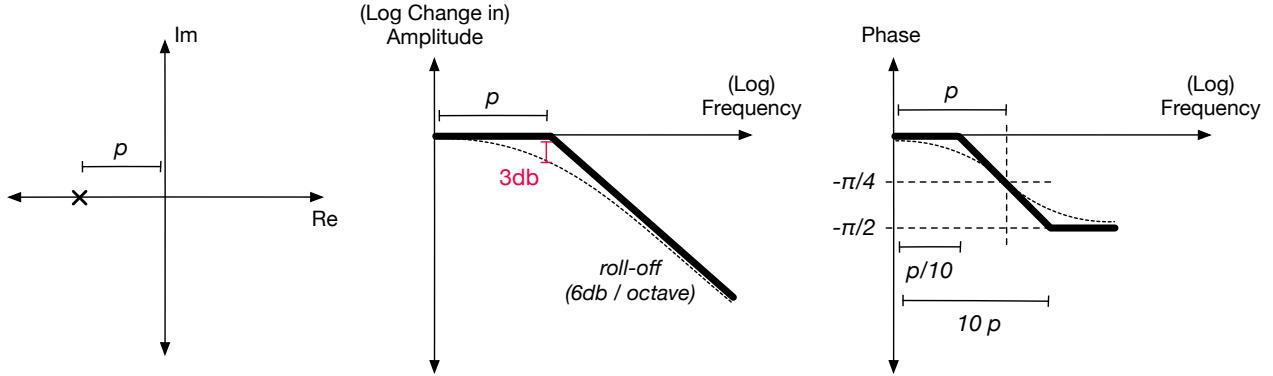


Figure 101 (Left) Position of single pole at $-p$. (Center) Bode plot of the amplitude response of the filter. Boldface line represents idealized filter (and useful as a rule of thumb) while the dotted line represents the actual filter. At the idealized cutoff frequency (p) the real filter has dropped 3db. This being a one-pole filter, at the limit the roll-off is a consistent 6db per octave. (Right) Bode plot of the phase response of the filter. Again, boldface line represents a useful rule-of-thumb approximation of the filter behavior, whereas the curved dotted line represents the actual behavior. Phase begins to significantly change approximately between $p/10$ and $p \times 10$.

- Poles either come in pairs or singles (by themselves). A single pole is only permitted if it lies on the real axis: that is, its imaginary portion is zero. Alternatively poles can exist in complex conjugate pairs: that is, if pole $p_1 = a + bi$, then pole $p_2 = a - bi$. The same goes for zeros. This should make sense given that poles and zeros are just roots of a polynomial equation, and roots are either real or are complex conjugate pairs.
- Zeros follow the same rule: they also must either come in complex conjugate pairs, or may be singles if they lie on the real axis.
- Poles must be on the left hand side of the complex plane: that is, they must have zero or negative real parts. Otherwise the filter will be unstable. This rule does not hold for zeros.

One simple intuitive idea to keep in mind is that a pole generally will cause the slope of the amplitude portion of the Bode plot to go *down*, while adding a zero generally will cause it to go *up*. We can use this heuristic to figure out how to build a filter whose amplitude characteristics are what we want.

Let's start with a single pole lying at $-p$ on the real (x) axis, as shown in Figure 101. As revealed in this Figure, a pole causes the amplitude to drop with increasing frequency. Since this is a single pole, the roll-off will be 6db per octave (recall that Bode plots are in log scale in frequency and in amplitude). The amplitude response of the ideal filter would look like the boldface line in the Figure (center), but that's not possible. Rather, the filter will drop off such that there is a 3db drop between the idealized filter and the actual filter at the cutoff frequency, which is at p !

A filter will also change the phase of partials in the signal. A typical phase response is shown in Figure 101 (right), Again, the boldface line shows the idealized (or in this case more like fanciful) response.

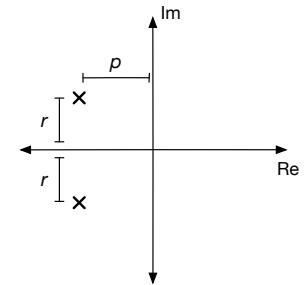


Figure 100 Two poles as a complex conjugate pair. The value p is the same as in Figure 101. The value r is related to the degree of resonance in the filter.

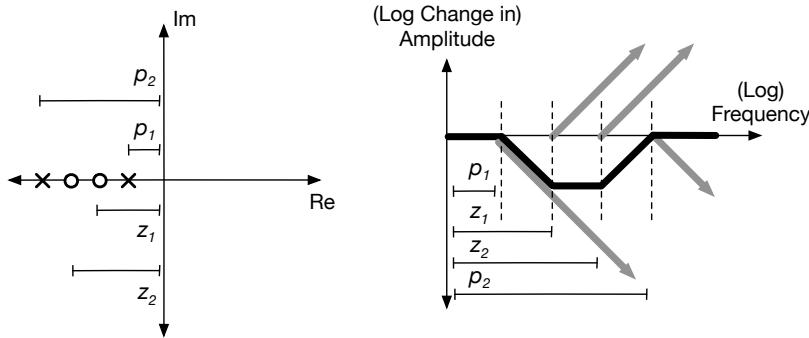


Figure 102 (Left) positions of two poles and two zeros on the real axis. (Right) Approximate Bode plot showing impact of each pole and zero in turn. Bold line shows final combined impact: a band reject filter of sorts. Gray bold lines are the roll-offs of each filter starting their respective cutoff frequency points. Note that because the figure at right is in *log frequency*, to produce the effect at right would require that the poles and zeros be spaced exponentially, not linearly as shown; for example, $p_1 = 1, z_1 = 10, z_2 = 100, p_2 = 1000$.

Now consider two poles. If the poles are not on the real axis, they must be complex conjugates, as shown in Figure 100. Note that the distance r from the real axis is associated with the degree of resonance in the filter. If all the poles are on the real axis, then $r = 0$ and the filter has no resonance. If you think about it this means that a *one pole filter cannot resonate* since its sole pole must lie on the real axis. Second order (and higher) filters can resonate because they have two poles and thus can have complex conjugate pairs. Additionally, if you have two poles, either as a complex conjugate pair, or stacked up on top of one another on the real axis, they essentially double the roll-off at p . Thus the roll-off is now 12db.⁷⁶

We've seen that the presence of a pole will cause the amplitude response to drop over time. Correspondingly, the presence of a zero will cause the amplitude to *rise* by the same amount. Furthermore, the distance p of the pole or zero from the imaginary axis (its negative real value) roughly corresponds to when the pole or zero starts having significant effect: that is, p corresponds to the cutoff frequency for that pole or zero.

We can use this to cause poles and zeros to approximately act against one another. Consider the two-pole, two-zero filter shown in Figure 102. At p_1 the first pole comes into effect, and begins to pull the amplitude down. Then at z_1 the first zero comes into effect, and begins to pull the amplitude up: at this point p_1 and z_1 effectively cancel each other out, so the amplitude response stays flat. Then at z_2 the second zero comes into effect: combined with z_1 it overwhelms p_1 and begins to pull the response up again. Finally at p_2 the final pole takes effect and things even out again. Behold, a band reject filter.⁷⁷

Gain As discussed before, these filters can also change the overall amplitude, or **gain**, of the signal. We'd like to avoid having a change at all (that is, we'd want a **unity gain filter**), or at least be able to control the gain. Here we'll just cover some basics. In general a first-order low-pass filter with a gain of K has a transfer function of the form:

⁷⁶That should sound familiar: in the synthesizer world, 2-pole filters are also (somewhat incorrectly) referred to as "12db" filters. At this point, you might be able to surmise why 4-pole filters are also often referred to (even more incorrectly) as "24db" filters.

⁷⁷Notice that I'm not discussing the phase response: since it's not very important for us, I'm omitting it here. Consider yourself fortunate.

$$H(s) = K \frac{1}{\tau s + 1}$$

Now consider a low-pass filter with a single pole $-p_1$. It has a transfer function

$$H(s) = \frac{1}{s + p_1} = \frac{1}{p_1 s/p_1 + 1}$$

So $K = 1/p_1$. We'd like $K = 1$, so we need to multiply by p_1 , resulting in

$$H(s) = \frac{p_1}{s + p_1} = \frac{1}{s/p_1 + 1}$$

In general, for a multi-pole low pass filter $-p_1, \dots, -p_i$, we need to have $p_1 \times \dots \times p_i$ in the numerator to make the filter have unity gain. Thus we have:

$$H(s) = \frac{p_1 \times \dots \times p_i}{(s + p_1) \times \dots \times (s + p_i)} = \frac{1}{(s/p_1 + 1) \times \dots \times (s/p_i + 1)}$$

Just for fun, let's consider the two-pole low pass case, with $p_1 = p_2 = p$. This implies that the two poles are stacked on top of each other and thus must be on the real axis.

$$H(s) = \frac{1}{(s/p + 1) \times (s/p + 1)} = \frac{1}{\frac{s^2}{p^2} + \frac{2s}{p} + 1}$$

Compare this equation to Equation 3 on page 96. This is effectively a special case of the low pass unity-gain second-order **Butterworth filter** discussed in Section 8.9. You might try working out what happens when p_1 and p_2 are complex conjugates, and its relationship to Equation 3.

8.7 The Z Domain and the Bilinear Transform

It turns out the Laplace domain, meant for analog filters, can't be directly used to build digital ones. To do this, we need our poles and zeros in a *different* complex-number plane called the **Z domain**, from which we can directly extract the information we need to build a digital filter. Unfortunately, there is no one-to-one, onto mapping from Laplace to Z, and all approximate mappings necessarily have failings. We'll get to the structure of the Z domain in the next section, but first let's focus on a common mapping called the **bilinear transform**. It looks like this:

$$s = \frac{2z - 1}{Tz + 1}$$

...where T is the size of the discretization in our digital signal. For example, if we are sampling at 44.1KHz, then $T = 1/44100$. Note that this means that $2/T$ is just the Nyquist limit (in this case, 22050).⁷⁸

Example. Let's convert a Laplace transfer function to the Z domain. To keep things simple, we'll entertain the ridiculous notion that $T = 1$:

$$H(s) = \frac{Y(s)}{X(s)} = \frac{s+2}{s^2 - 1} \longrightarrow H(z) = \frac{\frac{2}{T} \frac{z-1}{z+1} + 2}{\left(\frac{2}{T} \frac{z-1}{z+1}\right)^2 - 1} = \frac{2 \frac{z-1}{z+1} + 2}{4 \left(\frac{z-1}{z+1}\right)^2 - 1}$$

⁷⁸The bilinear transform is sometimes written as $s = \frac{2}{T} \frac{1-z^{-1}}{1+z^{-1}}$. It's the same thing. By the way, the inverse is $z = \frac{2+sT}{2-sT}$.

Yuck. I have no idea how to simplify that. Fortunately, that's what *Mathematica* is for:

$$H(z) = \frac{4z^2 + 4z}{3z^2 - 10z + 3}$$

Now we'll do two more steps. First we want all the z exponents to be negative:

$$H(z) = \frac{4z^2 + 4z}{3z^2 - 10z + 3} \times \frac{z^{-2}}{z^{-2}} = \frac{4 + 4z^{-1}}{3 - 10z^{-1} + 3z^{-2}}$$

Last we want a 1 in the denominator:

$$H(z) = \frac{4 + 4z^{-1}}{3 - 10z^{-1} + 3z^{-2}} \times \frac{1/3}{1/3} = \frac{4/3 + 4/3z^{-1}}{1 - 10/3z^{-1} + z^{-2}}$$

The Payoff These are the coefficients for our digital filter! Specifically if you have a digital filter of the form

$$y(n) + a_1y(n-1) + \cdots + a_Ny(n-N) = b_0x(n) + b_1x(n-1) + \cdots + b_Mx(n-M)$$

... then the transfer function, in z , is:

$$H(z) = \frac{b_0 + b_1z^{-1} + \cdots + b_Mz^{-M}}{1 + a_1z^{-1} + \cdots + a_Nz^{-N}}$$

Example. Let's continue where we had left off. We had

$$H(z) = \frac{4/3 + 4/3z^{-1}}{1 - 10/3z^{-1} + z^{-2}}$$

Thus we have a second order digital filter with $b_0 = 4/3, b_1 = 4/3, a_1 = -10/3, a_2 = 1$.

Delay Notation Notice that a coefficient corresponding to a delay of length n appears alongside z with the exponent form z^{-n} . For this reason it is common in the digital signal processing world to refer to an n -step delay as z^{-n} and thus the one-step delay element in our diagrams would be commonly written as z^{-1} .

Frequency Warping One item to be aware of is that because of peculiarities in the Bilinear Transform's mapping, a frequency of ω_Z in the Z domain doesn't linearly correspond to a frequency of ω_L in the Laplace domain due to the phenomenon of **frequency warping**. We often want to design digital filters with certain **cutoff frequencies**, and to do this we need to know what the equivalent "warped" cutoff frequency should be in the Laplace domain to achieve that. It turns out that the equations for warping frequencies between the Z and Laplace domains are:

$$\omega_L = 2/T \tan(\omega_Z/2) \quad \omega_Z = 2 \tan^{-1}(\omega_L T/2)$$

The second equation is the more important one: we figure out what our desired cutoff frequency is (that's ω_Z), then compute the frequency ω_L to use in our equations for building the filter in the Laplace domain.

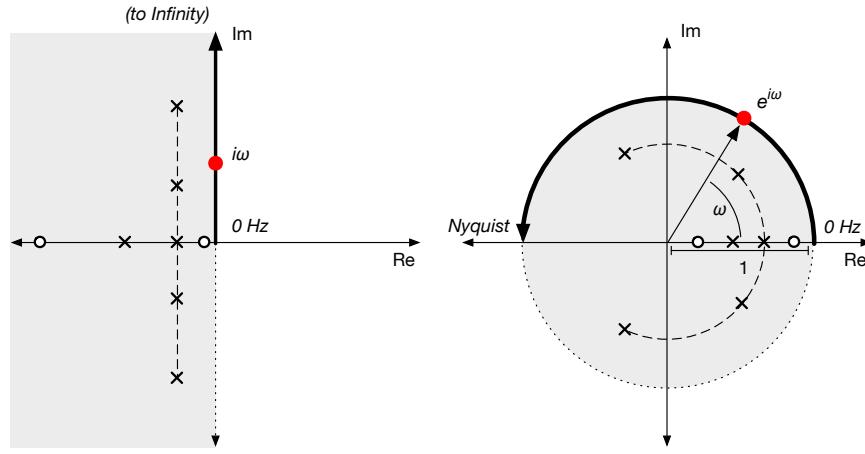


Figure 103 Relationships between the Laplace Domain (left) and the Z Domain (right). Notice that the entire (infinite) left half of Laplace plane is mapped inside the unit circle in the Z plane. Whereas the cutoff frequency ω in the Laplace plane goes up along the imaginary axis from 0 to ∞ as $i\omega$, in the Z domain it runs along the unit circle as $e^{i\omega}$, corresponding to going from 0 to the Nyquist frequency. Note how the example poles and zeros are warped in the mapping.⁷⁹

8.8 Pole and Zero Placement in the Z Domain

Wouldn't it be easier to define the poles and zeros in the Z domain, rather than defining them in Laplace, and then converting via the Bilinear Transform? We didn't do that for a couple of reasons. First, it's good to understand Laplace to start with, and how it relates to the Z domain. Second, in the Laplace domain the poles and zeros have intuitive relationships with the amplitude and phase response, and Z's relationships are rather less intuitive. But there's no reason you couldn't place poles and zeros directly in the Z domain itself, and in fact many designers do this.

It's useful to first understand why the Z domain is a bit odd. Figure 103 shows the basic relationships between the Laplace and Z domains. Note that the *entire* infinite left half of the Laplace complex plane is squished into a finite region in the Z domain, the inside of the unit circle. Furthermore, whereas the frequency moves along the imaginary axis from 0 to positive imaginary infinity in the Laplace domain, in the Z domain, it moves along the unit circle starting at $\omega = 0$ radians and ending at π , which corresponds to the Nyquist frequency. This makes some sense, since continuous filters deal with infinite frequencies, but digital filters only deal with values up to Nyquist.

In Laplace, it was straightforward to compute the magnitude and phase response: given a frequency ω , you first plotted the point $i\omega$, and then computed its distance and/or angle from various poles and zeros. The procedure is basically the same in the Z domain, but as you can see from Figure 103 the point is not $i\omega$ but rather $e^{i\omega}$ since it's running along the unit circle.

For example, recall that in Laplace to get the magnitude response given poles p_k and zeros z_j , you compute:

$$|H(s)| = |H(i\omega)| = \frac{|Y(i\omega)|}{|X(i\omega)|} = \frac{\prod_j |(i\omega - z_j)|}{\prod_k |(i\omega - p_k)|}$$

⁷⁹This diagram is largely a rip-off, with permission, of Figure 33-2 (p. 609) of Steven Smith, 1997, *The Scientist & Engineer's Guide to Digital Signal Processing*, California Technical Publishing, available online at <https://www.dsprelated.com/>.

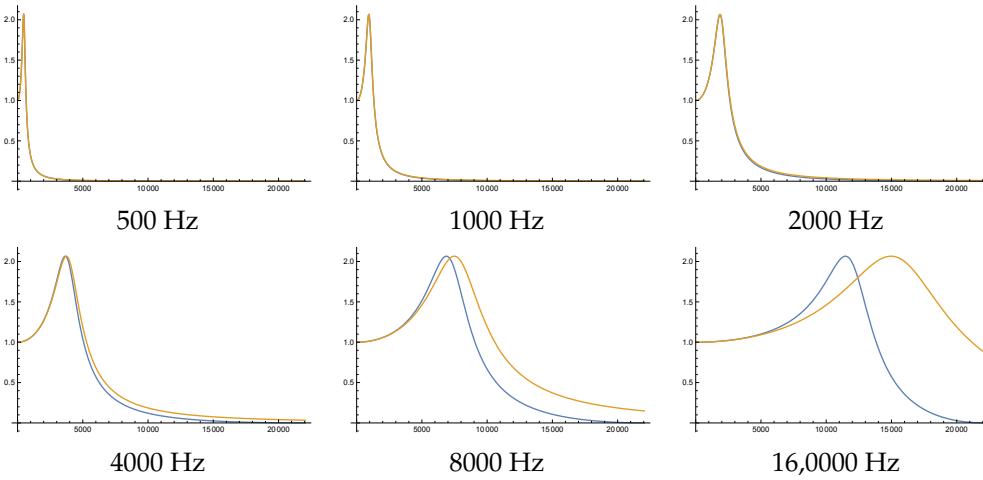


Figure 104 Difference in Magnitude (Amplitude) Response between a Low-pass Butterworth filter in the **Laplace Domain** and one converted to the **Z Domain** (44.1KHz) via a Bilinear Transform, for different cutoff frequencies. Resonance is set high ($Q = 2$) to make things obvious. In 500 and 1000 Hz the two are very nearly identical (Laplace is on directly on top of Z). By 16,000 the divergence is significant.

In the Z domain, you'd more or less do the same thing, but with $e^{i\omega}$ (which, if you recall from Section 3, just means $\cos(\omega) + i \sin(\omega)$):

$$|H(z)| = |H(e^{i\omega})| = \frac{|Y(e^{i\omega})|}{|X(e^{i\omega})|} = \frac{\prod_j |(e^{i\omega} - z_j)|}{\prod_k |(e^{i\omega} - p_k)|} = \frac{\prod_j |(\cos(\omega) + i \sin(\omega) - z_j)|}{\prod_k |(\cos(\omega) + i \sin(\omega) - p_k)|}$$

Similarly, the phase response would be:

$$\angle H(z) = \angle H(e^{i\omega}) = \sum_j \angle(\cos(\omega) - i \sin(\omega) - z_j) - \sum_k \angle(\cos(\omega) - i \sin(\omega) - p_k)$$

Though the Z domain maps frequencies from 0 to Nyquist about the unit circle from 0 to π , this isn't quite what the Bilinear Transform does. Rather, the Bilinear Transform squishes the *entire imaginary axis* into the unit circle. That is, it maps all values of $i\omega$, from 0 to positive imaginary infinity, to the unit circle from 0 to π : the infinite is mapped to the finite. As $i\omega$ gets larger, its mapping gets more and more compressed as it approaches π on the unit circle in Z. This is why you'd need to do frequency warping in Laplace: to get the right frequency values in Z, you need unusual equivalent frequency values in Laplace due to the nonlinearity.

For the same reason, the Bilinear Transform doesn't produce *exactly* the same filter in the Z domain: the filter frequencies are warped to some degree. The good news is that this warping is much more pronounced in the higher frequencies, where we don't care about the disparity so much for audio purposes: at lower frequencies (under 1/4 Nyquist, say) the two are very similar. Figure 104 illustrates this for different cutoff frequencies.

The Bilinear Transform is a useful approximation, and we'll take advantage of it in the next two Sections (8.9 and 8.10). But defining poles and zeros directly in the Z domain has its merits: you can avoid a lot of math if you get a hang of the impact of their placement.⁸⁰

⁸⁰The MicroModeler DSP is a great online tool for building filters in the Z domain directly from poles and zeros. <http://www.micromodeler.com/dsp/>

On the other hand, Vadim Zavalishin's *The Art of VA Filter Design* (VA as in "virtual analog") goes into depth on historical filters and how to replicate them digitally: but the text largely stays in Laplace, with a section on how to convert the result to the Z domain via the Bilinear Transform and other methods. https://www.discodsp.net/VAFilterDesign_2.1.0.pdf

8.9 Basic Second-Order Butterworth Filters

Filter design is largely about compromise. There are many kinds of filter approaches with different and contradictory characteristics. Consider the concepts in Figure 105 at right. Often we might want a very rapid dropoff in the **transition band**, or a large **attenuation**, but filters which can achieve this might also come with significant **ripple** in the **passband** or in the **stopband**; or they might be accompanied by a strong shift in **phase response**. As shown in Figure 106, filters also may be slow to respond to abrupt changes in the sound, may **overshoot** the goal amplitude, and may oscillate considerably before settling down (so-called **ringing**).

In this Section we'll consider one popular filter family in the audio world (and elsewhere), the **Butterworth filter**. As shown in comparison with other common families in Figure 107, Butterworth filters are simple, have a smooth (if not rapid) transition, and have no ripple, though you can augment them with resonance.

They do have downsides. Butterworth filters can cause considerable deviations in phase response, and they typically have significant overshoot and ringing. These effects are pronounced when the transition is steep (that is, when there are many more poles than we consider here).

Perhaps for our purposes, the best feature of Butterworth filters is that they are easy to explain! So we'll use them as our demonstration filter design. We'll start here by defining them in Laplace, then move on to converting them to the Z domain from which we can extract coefficients for a real digital filter.

We'll consider 2-pole (second-order) unity gain Butterworth filters in low pass, high pass, band pass, and notch configurations. These filters all have the same basic transfer function in Laplace:

$$H(s) = \frac{N(s)}{\frac{s^2}{\omega_0^2} + \frac{s}{Q\omega_0} + 1}$$

where s is the (complex) *frequency*, $Q \geq 0$ is the desired (real valued) resonance *quality factor*, and ω_0 is the desired (real valued) *cutoff frequency*. Recall that $Q = \sqrt{1/2}$ is the dead-flat position (above which resonance starts peaking). $N(s)$ is a polynomial which varies for different kinds of filters.

Low Pass For a low pass Butterworth filter at unity gain, $N(s) = 1$. Thus

$$H(s) = \frac{1}{\frac{s^2}{\omega_0^2} + \frac{s}{\omega_0 Q} + 1} \tag{3}$$

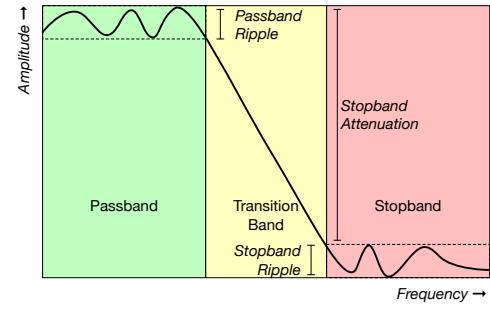


Figure 105 Filter terms (frequency domain).

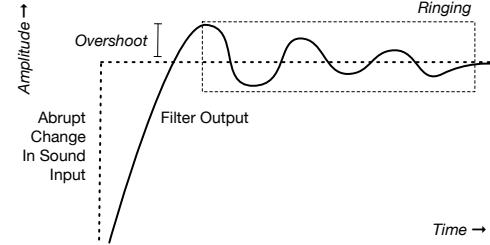


Figure 106 Filter terms (time domain).

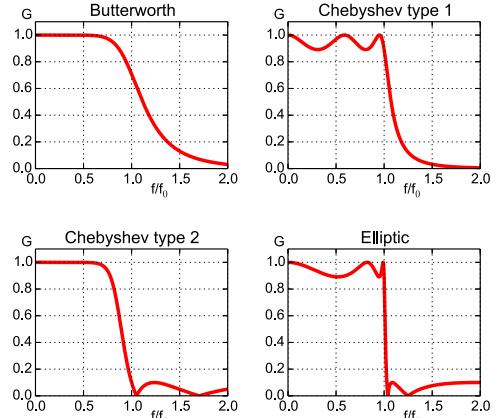


Figure 107 Frequency domain comparison of some common filter designs.⁵⁷

The two poles are $\left(-\frac{1}{2Q} \pm \sqrt{\frac{1}{(2Q)^2} - 1}\right) \times \omega_0$ and there are (of course) no zeros.⁸¹ To get the amplitude response, we have:

$$H(i\omega) = \frac{1}{\frac{(i\omega)^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1} = \frac{1}{-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1}$$

$$LP = |H(i\omega)| = \frac{1}{\left|-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1\right|} = \frac{1}{\left|(1 - \frac{\omega^2}{\omega_0^2}) + i\frac{\omega}{\omega_0 Q}\right|} = \frac{1}{\sqrt{\left(1 - \frac{\omega^2}{\omega_0^2}\right)^2 + \left(\frac{\omega}{\omega_0 Q}\right)^2}}$$

High Pass A high pass Butterworth filter at unity gain has $N(s) = \frac{s^2}{\omega_0^2}$. So

$$H(s) = \frac{\frac{s^2}{\omega_0^2}}{\frac{s^2}{\omega_0^2} + \frac{s}{\omega_0 Q} + 1}$$

The poles are the same as the low pass filter of course. The two zeros are simple: 0 and 0. To get the amplitude response, we have:

$$H(i\omega) = \frac{\frac{(i\omega)^2}{\omega_0^2}}{-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1} = \frac{\frac{-\omega^2}{\omega_0^2}}{-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1}$$

$$HP = |H(i\omega)| = \frac{\left|\frac{-\omega^2}{\omega_0^2}\right|}{\left|-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1\right|} = \sqrt{\left(\frac{-\omega^2}{\omega_0^2}\right)^2} \times LP$$

Band Pass A band pass Butterworth filter at unity gain has $N(s) = \frac{s}{\omega_0 Q}$. So

$$H(s) = \frac{\frac{s}{\omega_0 Q}}{\frac{s^2}{\omega_0^2} + \frac{s}{\omega_0 Q} + 1}$$

The poles are again same as the low pass filter. The sole zero is just 0. To get the amplitude response, we have:

$$H(i\omega) = \frac{\frac{i\omega}{\omega_0 Q}}{-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1}$$

$$BP = |H(i\omega)| = \frac{\left|\frac{i\omega}{\omega_0 Q}\right|}{\left|-\frac{\omega^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1\right|} = \sqrt{\left(\frac{\omega}{\omega_0 Q}\right)^2} \times LP$$

⁸¹You can work this out from the quadratic formula followed by some rearranging: for a polynomial of the form $ax^2 + bx + c = 0$, the roots are $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. In our case, $a = \frac{1}{\omega_0^2}$, $b = \frac{1}{\omega_0 Q}$, and $c = 1$.

Notch Finally, a notch Butterworth filter at unity gain has $N(s) = 1 + \frac{s^2}{\omega_0^2}$. So

$$H(s) = \frac{1 + \frac{s^2}{\omega_0^2}}{\frac{s^2}{\omega_0^2} - \frac{s}{\omega_0 Q} + 1}$$

The poles are again the same. The two zeros are $\pm i\omega_0$. The amplitude response is:

$$H(i\omega) = \frac{1 + \frac{(i\omega)^2}{\omega_0^2}}{-\frac{\omega_0^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1} = \frac{1 - \frac{\omega^2}{\omega_0^2}}{-\frac{\omega_0^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1}$$

$$Notch = |H(i\omega)| = \frac{\left|1 - \frac{\omega^2}{\omega_0^2}\right|}{\left|-\frac{\omega_0^2}{\omega_0^2} + \frac{i\omega}{\omega_0 Q} + 1\right|} = \sqrt{\left(1 - \frac{\omega^2}{\omega_0^2}\right)^2} \times LP$$

8.10 Digital Second-Order Butterworth Filters

In the previous Section we covered common second-order Butterworth filters in the Laplace domain, but not in the Z domain. Now let's convert them to the Z domain. You'd think that this conversion would be icky... and you'd be right. Thankfully we have tools that can do the algebraic simplification for us! Also to make things clearer, I have added a substitution called J . You'll see.

From final equation in each filter case, it's easy to derive the constants a_1, a_2, b_0, b_1, b_2 as simple equations of ω_0, Q , and T . Conveniently, because all four filters have the same poles, they also all have the same a_1 and a_2 constants! Also note that these equations refer to the cutoff frequency ω_0 : I believe that you don't have to do frequency warping on it since it's being pushed forward through the mapping equations already. Just provide it as-is. Remember that if you have a desired cutoff frequency of F Hz, then $\omega_0 = 2\pi \times F$. Also remember that neither ω_0 nor Q can be 0.

Keep in mind that because the Bilinear Transform is a warped approximation, these aren't exact. But as was shown in Figure 104 on page 95, the two are very close well into fairly high frequencies.

Low Pass

$$H(z) = \frac{1}{\frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1}\right)^2 + \frac{1}{\omega_0 Q} \frac{2}{T} \frac{z-1}{z+1} + 1}$$

$$= \frac{\omega_0^2 Q T^2 (1+z)^2}{2\omega_0 T (z^2 - 1) + Q(4(z-1)^2 + \omega_0^2 T^2 (1+z)^2)} \quad \text{Behold the magic of Mathematica}$$

$$= \frac{\omega_0^2 Q T^2 + 2\omega_0^2 Q T^2 z + \omega_0^2 Q T^2 z^2}{(4Q - 2\omega_0 T + \omega_0^2 Q T^2) + (-8Q + 2\omega_0^2 Q T^2)z + (4Q + 2\omega_0 T + \omega_0^2 Q T^2)z^2}$$

Now substitute $J = 4Q + 2\omega_0 T + \omega_0^2 Q T^2$

$$= \frac{\omega_0^2 Q T^2 + 2\omega_0^2 Q T^2 z + \omega_0^2 Q T^2 z^2}{(4Q - 2\omega_0 T + \omega_0^2 Q T^2) + (-8Q + 2\omega_0^2 Q T^2)z + Jz^2} \times \frac{1/J \times z^{-2}}{1/J \times z^{-2}}$$

$$= \frac{1/J \times \omega_0^2 Q T^2 z^{-2} + 1/J \times 2\omega_0^2 Q T^2 z^{-1} + 1/J \times \omega_0^2 Q T^2}{1/J \times (4Q - 2\omega_0 T + \omega_0^2 Q T^2)z^{-2} + 1/J \times (-8Q + 2\omega_0^2 Q T^2)z^{-1} + 1}$$

Thus we have the following coefficients for our digital filter:

Coefficient	Value	Because it's multiplied by...
b_0	$1/J \times \omega_0^2 QT^2$	$z^0 (= 1)$
b_1	$1/J \times 2\omega_0^2 QT^2$	z^{-1}
b_2	$1/J \times \omega_0^2 QT^2$	z^{-2}
a_1	$1/J \times (-8Q + 2\omega_0^2 QT^2)$	z^{-1}
a_2	$1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)$	z^{-2}

High Pass

$$\begin{aligned}
 H(z) &= \frac{\frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1} \right)^2}{\frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1} \right)^2 + \frac{1}{\omega_0 Q} \frac{2}{T} \frac{z-1}{z+1} + 1} \\
 &= \frac{4Q(z-1)^2}{2\omega_0 T(z^2 - 1) + Q(4(z-1)^2 + \omega_0^2 T^2(1+z)^2)} \quad \text{Once again Mathematica} \\
 &= \frac{4Q - 8Qz + 4Qz^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + (4Q + 2\omega_0 T + \omega_0^2 QT^2)z^2}
 \end{aligned}$$

Now substitute $J = 4Q + 2\omega_0 T + \omega_0^2 QT^2$

$$\begin{aligned}
 &= \frac{4Q - 8Qz + 4Qz^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + Jz^2} \times \frac{1/J \times z^{-2}}{1/J \times z^{-2}} \\
 &= \frac{1/J \times 4Qz^{-2} - 1/J \times 8Qz^{-1} + 1/J \times 4Q}{1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)z^{-2} + 1/J \times (-8Q + 2\omega_0^2 QT^2)z^{-1} + 1}
 \end{aligned}$$

Thus we have the following coefficients for our digital filter:

Coefficient	Value	Because it's multiplied by...
b_0	$1/J \times 4Q$	$z^0 (= 1)$
b_1	$1/J \times (-8Q)$	z^{-1}
b_2	$1/J \times 4Q$	z^{-2}
a_1	$1/J \times (-8Q + 2\omega_0^2 QT^2)$	z^{-1}
a_2	$1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)$	z^{-2}

Band Pass

$$\begin{aligned}
H(z) &= \frac{\frac{1}{\omega_0 Q} \frac{2}{T} \frac{z-1}{z+1}}{\frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1} \right)^2 + \frac{1}{\omega_0 Q} \frac{2}{T} \frac{z-1}{z+1} + 1} \\
&= \frac{2\omega_0 Q^2 T (z-1)(z+1)}{2\omega_0 T (z^2 - 1) + Q(4(z-1)^2 + \omega_0^2 T^2 (1+z)^2)} \quad \text{Once again Mathematica} \\
&= \frac{-2\omega_0 Q^2 T + 2\omega_0 Q^2 T z^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + (4Q + 2\omega_0 T + \omega_0^2 QT^2)z^2}
\end{aligned}$$

Now substitute $J = 4Q + 2\omega_0 T + \omega_0^2 QT^2$

$$\begin{aligned}
&= \frac{-2\omega_0 Q^2 T + 2\omega_0 Q^2 T z^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + Jz^2} \times \frac{1/J \times z^{-2}}{1/J \times z^{-2}} \\
&= \frac{-1/J \times 2\omega_0 Q^2 T z^{-2} + 1/J \times 2\omega_0 Q^2 T}{1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)z^{-2} + 1/J \times (-8Q + 2\omega_0^2 QT^2)z^{-1} + 1}
\end{aligned}$$

Thus we have the following coefficients for our digital filter:

Coefficient	Value	Because it's multiplied by...
b_0	$1/J \times 2\omega_0 Q^2 T$	$z^0 (= 1)$
b_1	0	z^{-1} (which isn't there)
b_2	$1/J \times (-2\omega_0 Q^2 T)$	z^{-2}
a_1	$1/J \times (-8Q + 2\omega_0^2 QT^2)$	z^{-1}
a_2	$1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)$	z^{-2}

Notch

$$\begin{aligned}
H(z) &= \frac{1 + \frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1} \right)^2}{\frac{1}{\omega_0^2} \left(\frac{2}{T} \frac{z-1}{z+1} \right)^2 + \frac{1}{\omega_0 Q} \frac{2}{T} \frac{z-1}{z+1} + 1} \\
&= \frac{Q(4(z-1)^2 + \omega_0^2 T^2 (1+z)^2)}{2\omega_0 T (z^2 - 1) + Q(4(z-1)^2 + \omega_0^2 T^2 (1+z)^2)} \quad \text{Once again Mathematica} \\
&= \frac{Q(r + \omega_0^2 T^2) + Q(-8 + 2\omega_0^2 T^2)z + Q(4 + \omega_0^2 T^2)z^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + (4Q + 2\omega_0 T + \omega_0^2 QT^2)z^2}
\end{aligned}$$

Now substitute $J = 4Q + 2\omega_0 T + \omega_0^2 QT^2$

$$\begin{aligned}
&= \frac{Q(4 + \omega_0^2 T^2) + Q(-8 + 2\omega_0^2 T^2)z + Q(4 + \omega_0^2 T^2)z^2}{(4Q - 2\omega_0 T + \omega_0^2 QT^2) + (-8Q + 2\omega_0^2 QT^2)z + Jz^2} \times \frac{1/J \times z^{-2}}{1/J \times z^{-2}} \\
&= \frac{1/J \times Q(4 + \omega_0^2 T^2)z^{-2} + 1/J \times Q(-8 + 2\omega_0^2 T^2)z^{-1} + 1/J \times Q(4 + \omega_0^2 T^2)}{1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)z^{-2} + 1/J \times (-8Q + 2\omega_0^2 QT^2)z^{-1} + 1}
\end{aligned}$$

Thus we have the following coefficients for our digital filter:

Coefficient	Value	Because it's multiplied by...
b_0	$1/J \times Q(4 + \omega_0^2 T^2)$	$z^0 (= 1)$
b_1	$1/J \times Q(-8 + 2\omega_0^2 T^2)$	z^{-1}
b_2	$1/J \times Q(4 + \omega_0^2 T^2)$	z^{-2}
a_1	$1/J \times (-8Q + 2\omega_0^2 QT^2)$	z^{-1}
a_2	$1/J \times (4Q - 2\omega_0 T + \omega_0^2 QT^2)$	z^{-2}

8.11 Formant Filters

We conclude this Section with a short discussion about modeling **formants**. The human vocal tract can be thought of as a reed instrument: a pipe (the throat, mouth, and nasal cavity) is attached to a vibrating reed (the vocal cords) through which air is pumped via the lungs. Because it is fixed in shape, the “pipe” resonates at certain frequencies regardless of the pitch of the sound being produced (that is, regardless of the frequency of the vocal cord vibration). As a result, the human vocal tract acts essentially as a filter on the vocal cords: it emphasizes certain frequencies: these are the formants.

Formants are labelled f_1, f_2, \dots , ordered from lowest frequency to highest. Each formant looks very much like a resonant **band pass filter**: it has a specific frequency, a specific peak amplitude, and a certain width in its dropoff (its **bandwidth**, tunable via resonance). Each vowel has its own set of formants with their own frequencies, amplitudes, and bandwidths.⁸² The particular settings of formants in vowels differ by vowel and also by the age or sex of the speaker due to the impact on the size and shape of the vocal tract. Formants can also vary depending on culture and language.

It’s straightforward to model a vowel by creating a **formant filter** out of multiple resonant bandpass filters, set to the right frequencies and (via resonance) bandwidths, and mixed together with the appropriate gains. A **diphthong**⁸³ could be modeled by morphing from one vowel to another by modifying the filter characteristics. That is, for each formant f_i , we smoothly interpolate from the first vowel’s f_i frequency to that of the second vowel, and similarly the two amplitudes and bandwidths.

To do this right, the input to the filters might be made by a digital waveguide model (Section 11.6), but a square wave, perhaps with a bit of triangle or sine thrown in, works well in a pinch.



Figure 108 First three male voice formants for an “Ahh” sound, as simulated by three resonant bandpass filters added together. Note the differences in frequency, amplitude, and bandwidth.

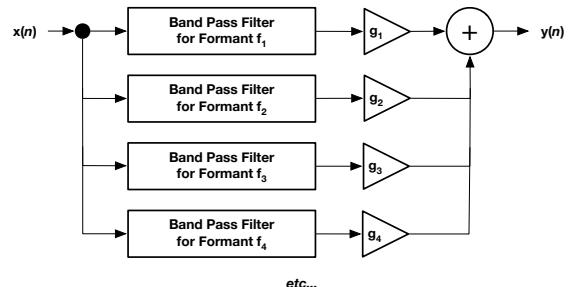


Figure 109 Simulating formants with multiple resonant band pass filters, multiplied by gains and then summed.

⁸²You’ll notice that I’m not providing a table of formants. Amazingly these tables vary quite considerably from one another across the Internet. You might try Table III (“Formant Values”) in the back of the *Csound Manual*, <http://www.csounds.com/manual/html/MiscFormants.html>

⁸³A diphthong is a sound made by combining two vowels. For example, the sound “ay” (as in “hay”) isn’t really a vowel—it’s actually the vowel “eh” followed by the vowel “ee”.

9 Frequency Modulation Synthesis

In 1967, using a computer at Stanford's Artificial Intelligence Laboratory, composer **John Chowning** experimented with **vibrato** where one sine wave oscillator slowly (and linearly) changed the frequency of a second oscillator, whose sound was then recorded. As he increased the frequency of the first oscillator, the resulting sound shifted from vibrato into something else entirely: a tone consisting of a broad spectrum of partials. He then attached an envelope to the first oscillator and discovered that he could reproduce various timbres, including (difficult at the time) brass instrument-like sounds. This was the birth of **frequency modulation** or FM synthesis.⁸⁴

FM, or more specifically its more easily controllable version **linear FM**, is not easy to implement in analog, and so did not come into its own until the onset of the digital synthesizer age. But when it did, it was so popular that it almost singlehandedly eliminated the analog synthesizer market.

Yamaha had obtained an exclusive license to FM for music synthesis from Stanford in 1973 (Stanford later patented it in 1975), and began selling FM synthesizers in 1980. In 1983 Yamaha hit pay dirt with the **Yamaha DX7**, one of the, if not *the*, most successful music synthesizers in history. The DX7 marked the start of a long line of FM synthesizers, largely from Yamaha, which defined much of the sound of pop music in the 1980s and 1990s. Among those, the **Yamaha TX81Z** rackmount synthesizer particularly found its way onto a great many pop songs due to its ubiquity in music studios.

FM synthesis then entered the mainstream with the inclusion of the Yamaha YM3812 chip (Figure 111) on many early PC sound cards, such as the **Creative Labs Sound Blaster** series. From there, the technique has since found its way into a myriad of video game consoles, cell phones, etc. because it is so easy to implement in software or in digital hardware.

9.1 Frequency and Phase Modulation

In fact, nearly all FM synthesizers don't do frequency modulation at all. Rather, they apply a related method called **phase modulation** or PM. This isn't bait-and-switch: phase modulation is slightly different in implementation but achieves the same exact effect.⁸⁵ Both phase and frequency modulation are subsets of a general category of modulation methods called **angle modulation**.⁸⁶ Phase modulation is easier to explain, so we'll begin with that.

⁸⁴The story of the birth of FM synthesis has been told many times. Here's a video of Chowning himself telling it. <https://www.youtube.com/watch?v=w4g92vX1YF4>

⁸⁵Plus nobody's ever heard of "phase modulation" or "PM" outside of music synthesis. When was the last time you heard of a "PM Radio"?

⁸⁶Which for obvious reasons *cannot* be abbreviated "AM".



Figure 110 Yamaha DX7.^{©58} (Repeat of Figure 3).

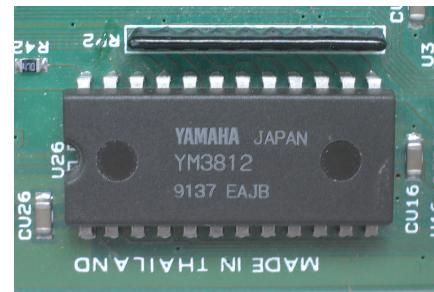


Figure 111 Yamaha YM3812 chip.^{©59}

Phase Modulation Let's consider the output of a single sine-wave oscillator, called the **carrier**, with amplitude a_c and frequency f_c , and which started at timestep $t = 0$ at phase ϕ_c :

$$y(t) = a_c \sin(\phi_c + f_c t)$$

The value $\phi_c + f_c t$ is the oscillator's **instantaneous phase**, that is, where we are in the sine wave at time t . Let's say we wanted to modulate this phase position over time. We could do this:

$$y(t) = a_c \sin(\phi_c + f_c t + m(t))$$

The **modulator** function $m(t)$ is doing **phase modulation** or PM. The **instantaneous frequency** of this sine wave is the frequency of the sine wave at any given timestep t . It's simply the first derivative of the instantaneous phase, that is, it's $\frac{d}{dt}(\phi_c + f_c t + m(t)) = f_c + m'(t)$. Thus by changing the phase of the sine wave in real time via $m(t)$, we're also effectively changing its frequency in real time via $m'(t)$.

Frequency Modulation Now let's say we wanted to *directly* change the frequency in real time with a function rather than indirectly via its derivative. That is, we want the instantaneous frequency to be $f_c + m(t)$. Since we arrived at the instantaneous frequency in the first place by differentiating over t , to get back to $y(t)$, we integrate over t , and so we have:

$$\begin{aligned} y(t) &= a_c \sin\left(\phi_c + \int_0^t f_c + m(x) dx\right) \\ &= a_c \sin\left(\phi_c + f_c t + \int_0^t m(x) dx\right) \end{aligned}$$

Here, instead of adding $m(x)$ to the phase, we're effectively folding in more and more of it over time. This direct modulation of frequency is called, not surprisingly, **frequency modulation** or FM. To change the frequency by $m(t)$, we just need to change the phase by some other function—in this case, by $\int_0^t m(x) dx$. Either way, regardless of whether we use phase modulation or frequency modulation, we're changing the frequency by changing the phase (and vice versa).

Phase and Frequency Modulation are Very Similar To hammer home just how similar phase and frequency modulation are, let's consider the situation where we are using a sine wave for $m(\dots)$.⁸⁷ In PM, we'd have

$$y(t) = a_c \sin(\phi_c + f_c t + a_m \sin(\phi_m + f_m t)) \quad (4)$$

In FM, let's again modify the instantaneous frequency using sine, that is, $f_c + m(t) = f_c + a_m \sin(\phi_m + f_m t)$. Integrating this over t and we get

$$\begin{aligned} \int_0^t f_c + a_m \sin(\phi_m + f_m x) dx &= f_c t + \frac{a_m}{f_m} (\cos(\phi_m) - \cos(\phi_m + f_m t)) \\ &= f_c t + \frac{a_m}{f_m} \cos(\phi_m) - \frac{a_m}{f_m} \cos(\phi_m + f_m t) \end{aligned}$$

⁸⁷In fact, this is a very common scenario in most FM synthesizers, so it's hardly far fetched!

$\frac{a_m}{f_m} \cos(\phi_m)$ is just a constant. Let's call it D . So we have

$$\begin{aligned} y(t) &= a_c \sin \left(\phi_c + f_c t + D - \frac{a_m}{f_m} \cos(\phi_m + f_m t) \right) \\ &= a_c \sin \left(\phi_c + D + f_c t + \frac{a_m}{f_m} \sin(\phi_m - \frac{\pi}{2} + f_m t) \right) \end{aligned} \quad (5)$$

Note how similar this equation is to the phase modulation equation, Equation 4. They differ in just a constant phase (ϕ_c versus $\phi_c + D$ and ϕ_m vs $\phi_m - \frac{\pi}{2}$), and amplitude factor (a_m vs $\frac{a_m}{f_m}$). The phases are typically disregarded anyway, so we might ignore them. The amplitude factor (which is called the **index of modulation** later) will matter, but it's just a constant change. The take-home lesson here is: phase modulation and frequency modulation are not the *same equation* (one is in part the first derivative of the other) but they can be used to produce the *same result*.

Linear and Exponential FM Analog subtractive synthesizers have been capable of doing frequency modulation forever: just plug the output of a sine-wave oscillator module into the frequency control of another sine-wave oscillator module, and you're good to go. So why wasn't FM common until the 1980s?

There is a problem. The frequency control of oscillators in analog synthesizers is historically exponential. Recall that most analog synthesizers were organized in **volt per octave**, meaning that an increase in one volt in a signal controlling pitch would correspond to an increase in one octave, which is a *doubling* of frequency.⁸⁸ Consider a sine wave going from -1 to $+1$ being used to modulate the frequency of our oscillator. The oscillator has a base frequency of, say, 440 Hz. At -1 the sine wave has cut that down by one octave to 220 Hz. At $+1$ it has pushed it up by one octave to 880 Hz. But 440 is not half-way between 220 and 880: the frequency modulation is not symmetric about 440, and the effect is distorted.

This kind of FM is called **exponential FM**, and it's not all that usable. It wasn't until the advent of digital synthesizers, which could easily control frequency linearly, that we saw the arrival of FM as discussed here, **linear FM**. With linear FM our sine wave would shift the frequency between $440 - N$ and $440 + N$, and so the modulation would be symmetric about 440.

9.2 Sidebands, Bessel Functions, and Reflection

Just as was the case in amplitude modulation and ring modulation (see Section 7.6), frequency modulation and phase modulation produce additional partials called **sidebands**. When both the carrier and modulator are sine waves, sidebands generally appear symmetrically on both sides of the carrier's partial (f_c), evenly spaced by f_m . That is, there will be a partial at $f_c \pm \alpha f_m$ for $\alpha = 0, 1, 2, \dots$. It is the complexity of the amplitudes of these generated sidebands which makes frequency modulation an interesting synthesis method.

Most literature which discusses sidebands considers the simple situation of a single sine-wave carrier being modulated by a single sine-wave modulator. In both phase modulation (Equation 4) and frequency modulation (Equation 5) we saw that, disregarding phase, we had an equation of roughly the form $y(t) = a_c \sin(f_c t + I \times \sin(f_m t))$ where we assumed the modulator was a sine wave. The value I is known as the **index of modulation**, and it is a parameter that we can set.

⁸⁸This was mentioned in Footnote 133. But I'm sure you read that.

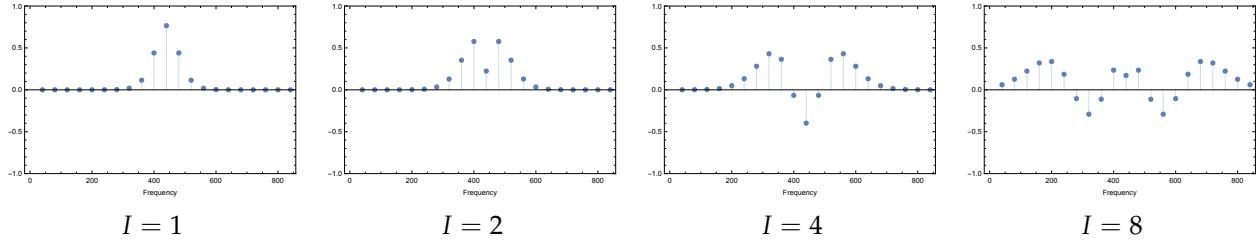


Figure 112 Change and spread of sidebands with increasing index of modulation (I). In all four of these graphs, $f_c = 440$ Hz and $f_m = 40$ Hz. As I increases, the spread (hence bandwidth) of sidebands does as well; and the pattern of sideband amplitude, including the carrier at the center, changes. Negative amplitudes just mean a positive amplitude but shifted in phase by π .

Bandwidth and Aliasing One aspect of the index of modulation is its effect on the dropoff in amplitude of the sidebands, and thus the **bandwidth** we have to deal with. The sidebands go on forever, but a heuristic called **Carson's rule** says that, for frequency modulation, 99% of all of the power of the signal is contained in the range $f_c \pm f_m \times (I + 1)$, and there are $I + 1$ significant sidebands on each side.⁸⁹ Recall that for PM, $I = a_m$, but for FM, $I = \frac{a_m}{f_m}$.

Let's say that $I = 1$, and we're playing a very high note (about 4000 Hz), and $f_m = 16 \times f_c$. Then we will have sidebands out to $4000 + (4000 \times 16) \times (1 + 1) = 128,000$ Hz. Yuck. What to do? We have a couple of options.

- We could have a high sample rate, and then **downsample**, probably via **windowed sinc interpolation** (Section 10.7). As an extreme example, imagine that we were sampling at 441 KHz (!) With a Nyquist frequency of 220,500 Hz, this is big enough to handle a sideband at 128,000 Hz. Downsampling would automatically apply a low pass filter to eliminate all frequencies higher than 22,050 Hz.
- We have to figure out how to prevent the wide bandwidth in the first place. One strategy would be to limit the legal values of I and f_m , or at least reduce the maximum value of I when f_m and f_c are high.

Bessel Functions The index of modulation also comes into play in determining the amplitude of each of the individual sidebands. The actual amplitudes are complex: and in fact the carrier frequency f_c may or may not be the loudest partial. In short, the amplitude of each sideband is determined by a **Bessel function of the first kind**. This complicated function is denoted $J_n(x)$, where n , an integer ≥ 0 , is the **order** of the function. Figure 113 shows the first eight orders, that is, $J_0(x), \dots, J_7(x)$.

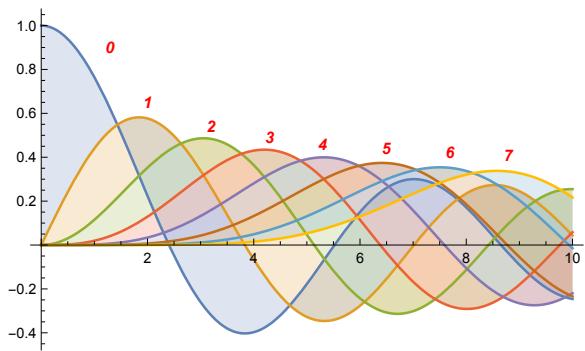


Figure 113 First eight orders of Bessel functions of the First Kind, positive values only (orders labeled in red).

⁸⁹Carson's Rule is not exact. Consider when $I = 0$. Then $y(t) = a_c \sin(f_c t + I \times \sin(f_m t)) = a_c \sin(f_c t)$ and there is no frequency modulation at all—we just have a single partial at f_c —yet Carson's Rule implies that the bandwidth is $f_c \pm f_m \times (I + 1) = f_c \pm f_m$, rather than 0 as it should be.

Here's how it works. Given index of modulation I , then $J_0(I)$ is the amplitude of the carrier, that is, the partial at f_c . Furthermore, $J_\alpha(I)$ is the amplitude of sideband numbers $\pm\alpha$, located at $f_c \pm \alpha f_m$. These can get complicated fast. Figure 114 shows the amplitude of various sidebands, and the carrier (sideband 0), for different modulation index (I) values. Figure 112 shows four cutaways from this graph for I values of 1, 2, 4, and 8. Some things to notice from these figures. First, with a small modulation index, the spectrum of the sound is just a few sidebands (indeed when $I = 0$, it's just the carrier alone), but as the index increases, the number of effected sidebands increases rapidly. Second, as the modulation index increases, some sidebands, including the carrier, can drop in amplitude, or go negative.⁹⁰

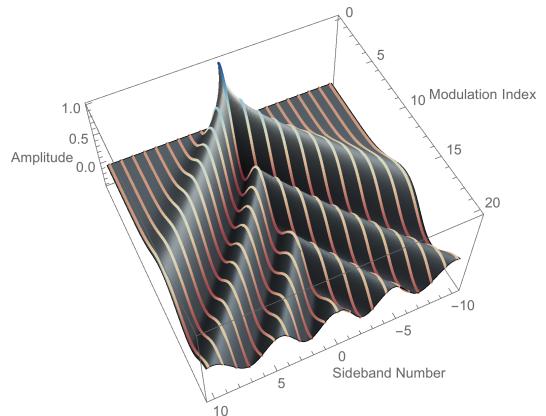


Figure 114 Sideband amplitudes by modulation index. Sideband numbers are integers, and shown as colored stripes. Note that this surface drops below zero in places.

Tonality and Reflection Think about the relationship between f_m and f_c . Consider first what happens when $f_m = f_c$. Then the sidebands to the right of f_c space out as $2f_c$, $3f_c$, and so on. These are **harmonics** of f_c as a **fundamental**. Furthermore, the sidebands to the left of f_c space out as 0 , $-f_c$, $-2f_c$, $-3f_c$, etc. But you can't have negative frequencies, so what happens to them? They're reflected back again, so they appear as 0 (which just the **DC offset**), f_c , $2f_c$, $3f_c$, and so on. These still line up in frequency with f_c and the sidebands to its right. Thus all the sidebands will be harmonics, with f_c as the fundamental (ignoring the DC offset), and the sound will be tonal to our ears.

This is the case for any f_m that is an integer multiple of f_c and $f_m \geq f_c$. Another interesting situation occurs when f_c is an integer multiple of f_m and $f_m < f_c$. Now the reflected sidebands once again match up and things are tonal, but f_c is no longer the fundamental in the sequence, because there were one or more sidebands appearing to its left before reflection occurred. See Figure 115 for illustration of this.⁹¹

In general if $\frac{f_m}{f_c}$ is rational then the positive and reflected negative partials will be integer multiples of *some* frequency, and so we'll get a tonal sound. But if $\frac{f_m}{f_c}$ is irrational, then the reflected negative sidebands won't line up with the positive sidebands, and the result will be **inharmonic**. We'll get an atonal sound: metallic, brash, or noisy.

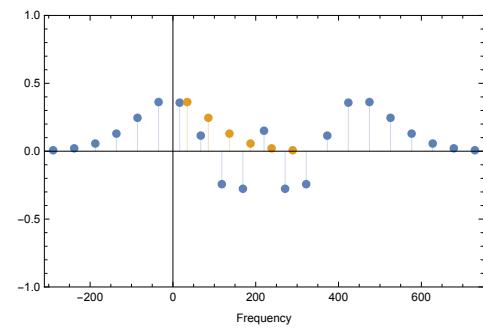
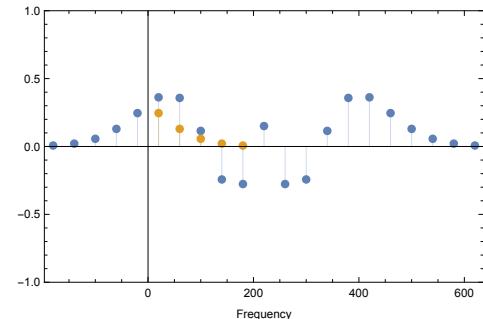


Figure 115 Sideband reflection in FM, with a 220Hz carrier. (Upper Figure) 40Hz Modulator: **reflected negative sidebands** line up with positive sidebands, and the result sounds tonal (though with a much lower fundamental than the original carrier). (Lower Figure) $40 \times \frac{4}{\pi}$ Hz Modulator: **reflected negative sidebands** do not line up and the sound is atonal.

⁹⁰Don't be put off by a negative amplitude. That's just a positive amplitude with a phase that's shifted by π .

⁹¹So what is the fundamental then? f_c will be the fundamental when $f_c = f_m$ as we've seen, or when $2f_c \leq f_m$. In other cases, to determine the fundamental, first set $f \leftarrow f_c$. Then repeatedly perform $f \leftarrow |f - f_m|$ until either $f = f_m$ or $2f \leq f_m$. At that point the ratio $f : f_m$ is in so-called **normal form** and f is the fundamental. See <https://www.sfu.ca/~truax/fmtut.html> for why this works.

9.3 Operators and Algorithms

We typically want the effect of a modulator on a carrier to change over time; otherwise the sound would be static and boring. The most common thing to change over time is the amplitude of each oscillator: this is typically done with its own dedicated envelope. Envelopes would thus effect indexes of modulation as well as the volume of the final outputted sound. The pairing of an oscillator with the envelope controlling its amplitude are together known as an **operator**. Thus we often don't refer to oscillators modulating one another but to *operators* modulating one another.

In the following examples, we'll stick to phase modulation as the equations are simpler. We'll simplify Equation 4 to describe operators as functions being modulated by other operators, that is, the output $y_i(t)$ of operator i is a function of the output of a modulating operator $y_j(t)$. And we'll ignore phase from now on. Accompanying this equation we can make a little diagram with the modulator operator on top and the carrier operator on bottom:

$$y_i(t) = a_i(t) \sin(f_i t + y_j(t))$$

So far we've just discussed a single carrier and a single modulator. But a modulator could easily modulate *several carriers*. Imagine that the oscillators are called i, j , and k . We could have:

$$y_i(t) = a_i(t) \sin(f_i t + y_j(t))$$

$$y_k(t) = a_k(t) \sin(f_k t + y_j(t))$$

Now, there's no reason that a carrier couldn't be modified by several modulators at once, with their modulations added up:

$$y_k(t) = a_i(t) \sin(f_i t + y_i(t) + y_j(t))$$

... or for an operator to modulate another, while being itself modulated by yet another operator.

$$y_j(t) = a_j(t) \sin(f_j t + y_k(t))$$

$$y_i(t) = a_i(t) \sin(f_i t + y_j(t))$$

... or for an operator to modulate itself...

$$y_i(t) = a_i(t) \sin(f_i t + y_i(t - 1))$$

... or for there to be a larger cycle in modulation.

$$y_k(t) = a_k(t) \sin(f_k t + y_i(t - 1))$$

$$y_j(t) = a_j(t) \sin(f_j t + y_k(t))$$

$$y_i(t) = a_i(t) \sin(f_i t + y_j(t))$$

And of course there's no reason why an operator has to be modulated by anyone else, that is,

$$y_i(t) = a_i(t) \sin(f_i t)$$

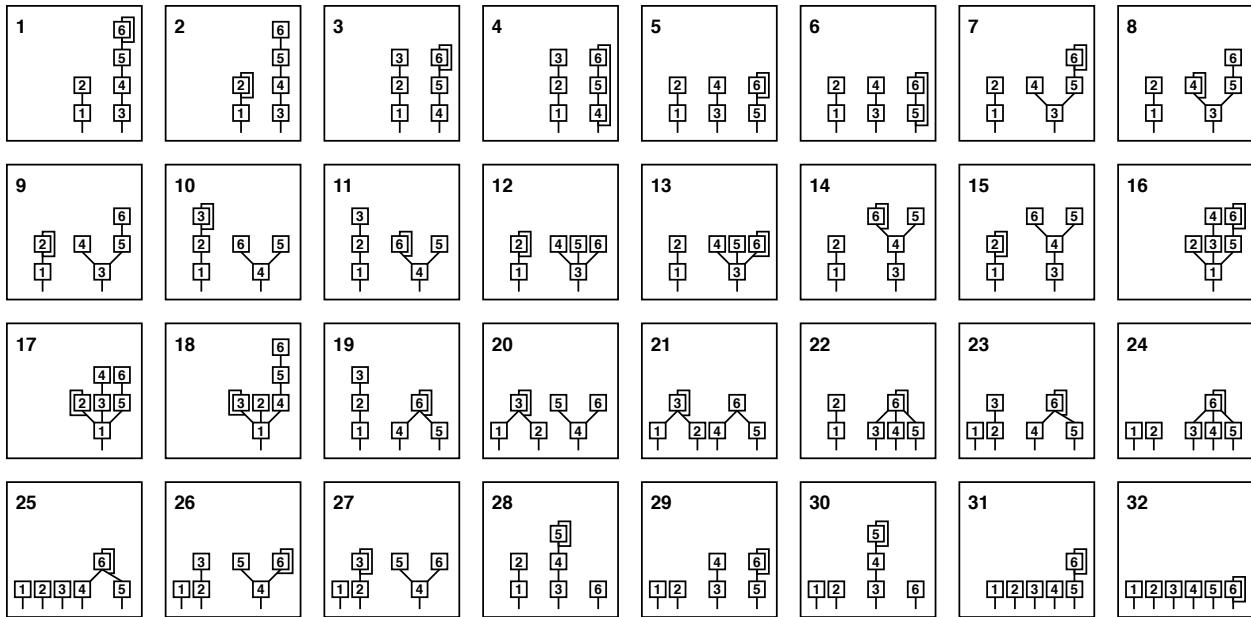


Figure 116 Operator modulation graphs (so-called “algorithms”) for the Yamaha DX7. Operators on the bottom layer (which have bare lines coming out from below them) are mixed together to produce the final sound. Other operators serve only as modulators. Many algorithms sport self-modulating operators, and in a few cases (Algorithms 4 and 6) larger modulation cycles.

The point is: the modulation mechanism in a patch is just a graph structure among some N operators. Some FM synthesizer software allows fairly complex graphs (for example, Figure 117). But many FM synths have followed an unfortunate tradition set by the **Yamaha DX7**: only allowing the musician to choose between some M predefined graph structures. Yamaha called these **algorithms**.

The DX7 had six operators, each of which had a sine wave oscillator and an envelope to control its amplitude. There were 32 preset algorithms using these six operators, as shown in Figure 116. Note that in an algorithm, some operators are designated to provide the final sound, while others are solely used to do modulation. In only three algorithms (4, 6, and 32) did an operator do both tasks. Operators designated to provide sounds ultimately have their outputs summed together, weighted by their operator amplitudes, to provide the final sound.

A few FM synthesizers, such as the **Yamaha FS1R**, had up to eight operators; but the vast majority of FM synths have had just four, with a very limited set of algorithms. However, many of Yamaha’s 4-operator FM synthesizers somewhat made up for their limitation by offering oscillators which could produce more than just sine waves.

Perhaps the most famous of these was the 4-operator, 8-algorithm, 8-waveform **Yamaha TX81Z**. Figure 118 shows the TX81Z’s eight algorithms and its eight possible waveforms. 4-operator synthesizers have since become ubiquitous, having had made their way into numerous PC soundcards, toy musical instruments, cell phones, and so on.



Figure 117 OXE 8-operator FM software synthesizer. Note the “modulation matrix” at right, whose lower-diagonal structure implies a full DAG is possible but not a cyclic graph except for self-modulating operators.

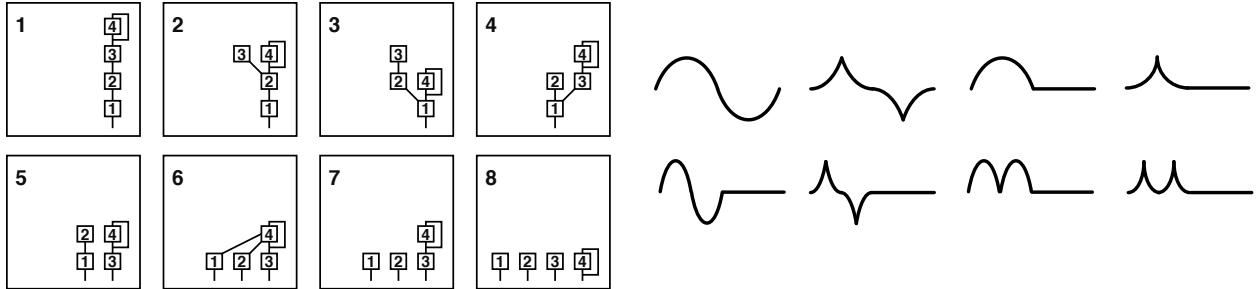


Figure 118 Algorithms (left) and waveforms (right) of the Yamaha TX81Z. Operators on the bottom layer (which have bare lines coming out from below them) are mixed together to produce the final sound: other operators serve only as modulators. Several algorithms sport self-modulating operators. Note that the waveforms are largely constructed out of pieces of the sine wave. Six waveforms are silent (zero amplitude) for half of their period.

9.4 Implementation

FM is a perfect match for software. But how would you implement it? Recall Equation 1 in the Additive Section, page 37. There we were maintaining the current sine wave phase for some oscillator i as:

$$x_i^{(t)} \leftarrow x_i^{(t-1)} + f_i \Delta t \mod 1$$

...where Δt was the sampling interval in seconds: for example, $1/44100$ seconds for 44.1KHz. The final output of this sine wave oscillator was:

$$y_i^{(t)} \leftarrow \sin(2\pi x_i^{(t)}) \times a_i^{(t)}$$

Let's say that this oscillator i is being modulated by the output of one or more oscillators, whose set is called $\text{Mods}(i)$. Then for phase modulation we could update the state of the oscillator x_i and its final output y_i as:

$$\begin{aligned} x_i^{(t)} &\leftarrow x_i^{(t-1)} + f_i \Delta t \mod 1 \\ y_i^{(t)} &\leftarrow \sin \left(2\pi \times \left(x_i^{(t)} + b_i \times \sum_{j \in \text{Mods}(i)} y_j^{(t-1)} \right) \right) \times a_i^{(t)} \end{aligned}$$

Keep in mind that you're also probably modifying a_i over time via the oscillator's accompanying envelope, and so y_i is an operator. Notice the b_i snuck into the equation above. This is just a useful opportunity to specify the degree to which all the incoming modulation signals affect the operator. Without it (or something like it), the index of modulation is largely defined by the a_i envelopes of the modulators, and so if some modulator is modulating different carriers, it will do so with the same index of modulation: you can't differentiate them.⁹² Anyway, if you don't care about this, just set $b_i = 1$.

⁹²Traditional Yamaha-style FM synthesizers don't have a b_i . The index of modulation is entirely controlled by the modulator's envelopes. However certain other FM synthesizers have b_i included, notably the PreenFM2 shown in Figure 119.

So how about frequency modulation? Here we're repeatedly summing the modulation into the updated state (that's the integration). Note again the optional b_i :

$$x_i^{(t)} \leftarrow x_i^{(t-1)} + f_i \Delta t + b_i \times \sum_{j \in \text{Mods}(i)} y_j^{(t-1)} \mod 1$$

$$y_i^{(t)} \leftarrow \sin(2\pi x_i^{(t)}) \times a_i^{(t)}$$

Of course these don't have to be sine waves: they can be any wave you deem appropriate: but sine has a strong tradition and theory regarding the resulting sidebands (and what antialiasing they will require). Most FM synthesizers aren't much more than this. Neither the DX7 nor TX81Z, nor most other Yamaha-style FM synths, had a filter or a VCA envelope.⁹³ They had a single LFO which could modify pitch and volume, plus a few other minor gizmos.

Advantages of Phase Modulation FM and PM have the same computational complexity and are both easy to implement. There are some differences to think about though. For example, imagine that y_j was a positive constant: it never changed. Then phase modulation would have no effect on the output of y_i . However frequency modulation *would* have an effect: y_i would have a higher pitch due to the added integration. Along these same lines, phase modulation can make it a bit easier to get an operator to *modulate itself* as $y_i^{(t)} \leftarrow \sin(x_i + y_i^{(t-1)}) \times a_i$, or to do similar cyclic modulations, without changing the fundamental pitch of y_i .

Overall, phase modulation seems to be somewhat easier to work with, and it is likely this reason that Yamaha chose phase modulation over frequency modulation for their FM (or, er, PM) synthesizers. Yamaha's synths offered self-modulation as an option, though in truth self-modulation tends to create fairly noisy and chaotic sounds. Partly because of these advantages, and partly because of Yamaha's influence, very few synthesizers in history have chosen FM over PM: one notable exception is the open-design **PreenFM2** (Figure 119).



Figure 119 PreenFM2.

Filter FM Last but not least: you can use audio-rate oscillators to modulate many other synthesizer parameters beyond just the frequency or phase of another oscillator. Ever since there were modular synthesizers, musicians have attached the output of oscillators to the modulation input of any number of modules to varying degrees of effect. One particularly common method worth mentioning here is **filter FM**, where an audio-rate oscillator is used to modulate the cutoff frequency of a filter through which an audio signal is being run. This can be used to create a wide range of musical or strongly discordant sounds.

⁹³There are exceptions. For example, the **Elektron Digitone** has both FM synthesis and filters, as do certain virtual analog synths with FM options.

10 Sampling

The synthesizers discussed so far have largely generated sounds algorithmically via oscillators: sawtooth waves, etc. But increases in computer power and (critically) memory capacity have made possible **sampling** sounds directly from the environment. The synthesizer's algorithmic oscillator is replaced in a sampler with an "oscillator", so to speak, which plays back the sampled sound. other portions of the subtractive synthesizer architecture remain.

This approach is now very widely used in the music industry. Major film scores are produced entirely using sampled instruments rather than a live orchestra. Stage pianos are often little more than sample playback devices. Sampling in hip hop has caused all manner of copyright headaches for artists and producers. Some sampled clips, such as the **Funky Drummer** or the **Amen Break**, have spawned entire musical subgenres of their own. It is even common to sample the output of *analog synthesizers*, such as the **Roland TR-808** drum machine, in lieu of using the original instrument.

10.1 History

Sampling and sample playback devices originated with early optical and **tape-replay** devices, the most well known example being the **Streetly Electronics Mellotron** series. These keyboards played a tape loop on which a sample of an instrument had been recorded.⁹⁴ Digital sampling existed as early as the 1960s, but sampling did not come into its own commercially until the late 1970s. Some notable early polyphonic examples were the **Fairlight CMI** and **New England Digital Synclavier**, both sampling and synthesis workstations.

Digital samples use up significant memory, and sample manipulation is computationally costly, so many improvements in samplers are a direct result of the exponential improvement computer chip performance and capacity over time. This has included better bit depth and sampling rates (eventually reaching CD quality or better), more memory and disk storage capacity, better **DACs** and **ADCs**, and improved sample editing facilities. Firms like **E-Mu Systems** and **Ensoniq** rose to prominence by offering less expensive samplers for the common musician, and were joined by many common brands from the synthesizer industry.

Many samplers emphasized polyphony and the ability to **pitch shift** or **pitch scale** samples to match played notes. But samplers were also increasingly used to record drums and percussion: these samplers did not need to vary in pitch in real time, but they *did* need to play many different samples simultaneously (drum sets, for example). This gave rise to a market for **phrase samplers** and sampling **drum machines** which specialized entirely in one-shot sample playback. Notable in this market was the **Akai MPC** series, which was prominent throughout hip-hop.

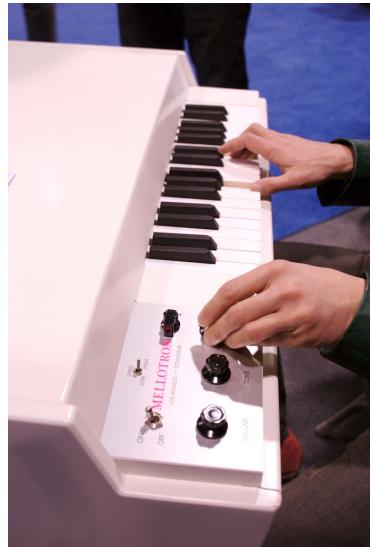


Figure 120 A Mellotron Mk VI, circa 1999.^{©60}



Figure 121 Akai MPC Renaissance sampling drum machine.^{©61}

⁹⁴In most cases you could not record your own samples, thus these were more akin to **romplers** than **samplers**.

Romplers The late 1980s saw the rise of **romplers**. These synthesizers played samples just as samplers did: but they were not samplers as they could not sample the sounds in the first place. Instead, a rompler would hold a large bank of digital samples in memory (in ROM—hence the derisive term “rompler”) which it played with its “oscillators”. Romplers were omnipresent throughout the 1990s, and were used in a great many songs. Romplers were very often rackmount units (as were most later samplers) and sported extensive **multitimbral** features, meaning that they not only had high voice polyphony, but that those voices could play different sounds from one another. This made it possible to construct an entire multi-instrumental song from a single rompler controlled by a computer and keyboard. Romplers generally had poor programming interfaces, as most of them were meant to fill a market demand for preset sound devices.

As computer hardware became cheaper and more capable, samplers and romplers were largely displaced by **digital audio workstations** which could do the same exact software routines in a more standard environment (the laptop).

10.2 Pulse Code Modulation

Pulse code modulation, or **PCM**, is just a fancy way of saying a wave sampled and stored in digital format for playback. A PCM wave is usually an array of numbers, one per sample, which indicate their respective amplitudes. PCM waves may be **one-shot waves**, or they may be meant to **repeat** in an endless loop. In the latter case, a specific location inside the wave might be designated as the point to loop back to (perhaps via a **cross fade**) after the wave has been exhausted.

One common looping construct is the **single-cycle wave**. This is a wave whose length is just one period, and which starts and ends at 0 amplitude, so it can be seamlessly repeated endlessly like a sawtooth or sine wave. It is not uncommon for a rompler patch to consist of a one-shot PCM wave to provide an initial splash, followed by a continuous single-cycle wave playing as long as the key is held down.⁹⁵ Single cycle waves also form the basis of many virtual analog synthesizer oscillators. For example, a common way to produce a sawtooth wave is to store a very high resolution, bandlimited sawtooth wave, and then filter and downsample it as necessary to produce a wave at the desired pitch.

Rompplers have long been derided as being little more than sample-playback devices, and in fact many have been. But there also were many novel rompler approaches taken by synthesizer manufacturers to capitalize on the unique opportunities afforded by PCM. For example, the **Sequential Circuits Prophet VS** introduced the concept of **vector synthesis**, which used one or more envelopes to cross-fade between four different PCM sounds in an elaborately modulated fashion. After Sequential Circuits went bankrupt, its founder **Dave Smith** furthered this idea in the development of the **Korg Wavestation**. Vector synthesis is what the Wavestation is largely known for,⁹⁶ but in fact its most powerful feature was the **wave sequence**: creating a new sound as the cross-faded sequence of potentially hundreds of PCM waves in a row. Another approach was to use ordinary PCM samples but run them through elaborate time-varying filters. E-mu Systems was particularly known for romplers which sported *hundreds* of complex filters: the **Emu Morpheus** and **UltraProteus** in particular were capable of changing filter parameters and shifting from one filter to another in real time.

⁹⁵This trick is very common, so much so that Roland named an entire synthesis brand after it: so-called **linear arithmetic synthesis**, the basis of a very successful line of synthesizers, such as the **Roland D-50**.

⁹⁶The basic PCM sounds in the Prophet VS were single-cycle waves, whereas the basic PCM sounds in the Korg Wavestation could be looping, single-cycle, or one-shot. Because the VS had single-cycle waves, some people incorrectly classify it as a **wavetable** synthesizer (Section 10.3), but it's not.

10.3 Wavetable Synthesis

Another rather different use of single-cycle waves is in the form of **waveTables**.⁹⁷ A wavetable is nothing more than array $W = \langle w_1, w_2, \dots, w_n \rangle$ of digitized single cycle waves. Figure 122 shows a wavetable of 64 such waves. A **wavetable oscillator** selects a particular single cycle wave w_i from this table and constantly plays it. The idea is that you can modulate *which* wave is currently playing via a parameter, much as you could modulate the pulse width of a square wave. As a modulation signal (from an envelope, say) moves from 0.0 to 1.0, the oscillator changes which wave it's playing from 0 to 63. This is more than just cross-fading between two waves, since in the process of going from wave 0 to wave 63 we might pass through any number of unusual waves. Depending on the speed of modulation, this could create quite complex sounds.

It's not surprising that many early wavetable synthesizers sported a host of sophisticated modulation options to sweep through those wavetables in interesting ways. For example, the **Waldorf Microwave** series had an eight-stage "wave envelope" with a variety of looping options, plus an additional four-stage bipolar (signed) "free envelope", in addition to the usual ADSR options. Figure 123 shows the front panel of the **Microwave XT** and its envelope controls.

It might interest you to know that wavetables have historically been stored in one of two forms. As memory is plentiful nowadays, wavetables are normally stored as arrays of single-cycle waves exactly as described earlier. But many historic Waldorf wavetable synthesizers instead held a large bank of available single-cycle waves, and each wavetable was a **sparse array** whose slots were either references to a wave in the bank, or were empty. The synthesizer would fill the empty slots on the fly with interpolations between the wave references on either side. This both saved memory and allowed multiple wavetables to refer to the same waves in ROM. Figure 124 shows a sparse wavetable example from the Microwave XT.

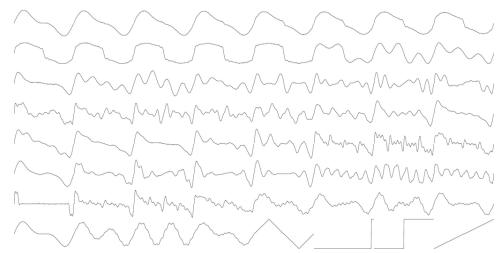


Figure 122 Wavetable #31 of the PPG Wave synthesizer, with 64 single-cycle waves. Most waves move smoothly from one to another, but the last four do not: these are triangle, pulse, square, and sawtooth, and appear in PPG and (minus pulse) Waldorf wavetables for programming convenience.^{©62}



Figure 123 Waldorf Microwave XT (rare "Shadow" version: most are safety orange!). Note the bottom right quadrant of the knob array, devoted entirely to envelopes.

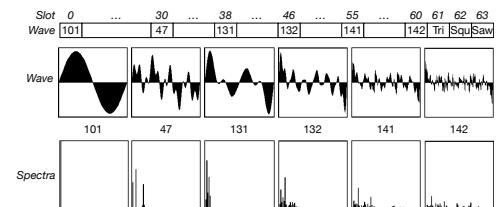


Figure 124 First 61 slots of the Waldorf Microwave XT's wavetable #3, "MalletSyn", a sparse array of empty slots interspersed with references to six single-cycle waves, plus the obligatory Triangle, Square, and Sawtooth.^{©63}

⁹⁷Note that many in the music synthesis community, myself included, use the term *wavetable* differently than its later usage in digital signal processing. In the music synthesis world, a wavetable is an array of digitized single cycle waves, a usage popularized early on by **Wolfgang Palm**. But in the DSP community, a wavetable has since come to mean a single digitized wave in and of itself! What the music synthesizer world typically calls *wavetable synthesis*, the DSP world might call **multiple wavetable synthesis**. To make matters worse, in the 1980s Creative Labs often incorrectly used the term "wavetable" to describe PCM samples generated from their **Sound Blaster** sound card.

Though it now appears in many synthesizers worldwide, wavetable synthesis is strongly linked with Germany: it is often attributed to Wolfgang Palm and his wavetable synthesizer, the **PPG Wave**. Palm later consulted for **Waldorf Music**, which in its various incarnations has produced wavetable synthesizers for over two decades.

Wavetables are nearly always bounded one-dimensional arrays. But the waves could instead be organized as an n -dimensional array.⁹⁸ The array needn't be bounded either: for example, it could be toroidal (wrap-around). Of course, an LFO or envelope can easily specify the index of the wave in the one-dimensional bounded case, but how would you do it in higher dimensions? One possibility is to define a **parametric equation**, that is, a collection of functions, one per dimension, in terms of the modulation value m . For example, if we had a two-dimensional space, we could define our wave index in that space as $i(m) = \langle \cos(\frac{m}{2\pi}), \sin(\frac{m}{2\pi}) \rangle$. Thus as the modulation went from 0 to 1, the index would trace out a circle in the space. If $i(0) = i(1)$, as was the case in this example, we could further use a sawtooth LFO to repeatedly trace out this path forever as an **orbit**.

10.4 Granular Synthesis

Granular synthesis is a family of methods which form sounds out of streams of very short sound snippets (as short as 1ms but more typically 5–50ms) known as **grains**. Though it has its roots in acoustic experiments in the 1940s, granular synthesis is largely attributed to the composer **Iannis Xenakis**, who (I believe) also coined the terms “grain” and “granular”.

Grains can be formed out of single-cycle waves such as sawtooth or a wave in a wavetable, but they are also very commonly formed by cutting up a sampled PCM sound into little pieces. Each grain is then associated with a **window function** (see Section 3.5) so that it starts and ends at zero and ramps smoothly to full volume in the middle. Without the window function, you'd likely hear a lot of glitches and pops as grains came and went. In granular synthesis, the window function is known as a **grain envelope**.

Early granular synthesis experiments used simple triangular (ramp up, then ramp down) or trapezoidal (ramp up, hold steady, then ramp down) windows, but as computer power increased, more elaborate windows became possible. One popular window is the **Hann window**, which is little more than a cosine. That is, applied to a grain of length M , the Hann window ranges from $[-M/2...M/2]$ and is defined as $\text{Hann}(x) = 1/2 \cos(2\pi x/M) + 1/2$. See Figure 125.

Because defining a stream of grains can require a very high number of parameters, granular synthesis methods usually simplify things in one of two ways.⁹⁹ First, **synchronous granular** methods repeat one or more grains in a pattern. These could be used for a variety of purposes:

- If the grains are interspersed with silence, you'll hear beating or rhythmic effects.
- If the grains come one right after the other (or are crossfaded into one another)¹⁰⁰ they could be used to compose new sounds out of their concatenation.
- You could also repeat the same grain over and over, perhaps with crossfading, to lengthen a portion of a sound. This can form the basis of stretching the length of a sample without changing its pitch, a form of **time stretching**.

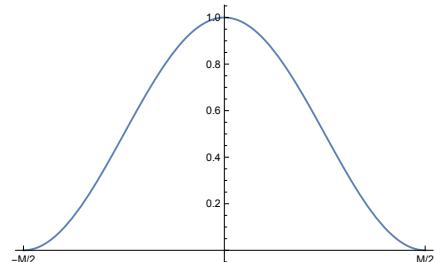


Figure 125 Hann Window.

⁹⁸A two-dimensional array of waves is known as a **wave terrain**, a term coined by **Rich Gold** in John Bischoff, Rich Gold, and Jim Horton, 1978, Music for an interactive network of microcomputers, *Computer Music Journal*, 2(3).

⁹⁹These categories are co-opted out of the five categories described by **Curtis Roads** in Curtis Roads, 2004, *Microsound*, The MIT Press.

¹⁰⁰Does this sound like a **wave sequence**? It does to me. I suppose the difference is that the sounds in a wave sequence can be long (over 100ms) whereas grains are very short.

At the other end of the granular spectrum are **asynchronous granular** methods, which produce a stream of randomly or pseudo-randomly chosen grains. These grains may vary randomly or deterministically in many ways, such as choice of grain, grain length, amplitude, window, grain density (how many of them appear in a given time interval), degree of overlap, location in the sound source, and pitch. A blob of grains in this form is often called a **grain cloud**.

The *length* of a grain has a significant impact on how grains are perceived. Very short grains may simply sound like pops or hiss. As grain length increases beyond 1ms or so we can start to perceive the pitch of the waves embedded in each grain, and this increases as grains grow to about 50ms. The *density* of the grains—that is, how much of the sound interval is occupied by grains—also has a significant impact. Very sparse sounds will produce beating or rhythmic patterns; denser grain sequences result in a single continuous sound; and very dense grains could have high degree of overlap, producing a wall of sound.

Granular synthesis is uncommon. Most granular synthesizers are software; hardware granular synths are very rare, especially polyphonic ones. One of the very few exceptions is the **Tasty Chips Electronics GR-1**, an asynchronous granular synth shown in Figure 126.



Figure 126 Tasty Chips Electronics GR-1.^{©64}

10.5 Resampling

The primary computational concern in sampling, and the other techniques discussed so far, is changing the pitch of a sampled sound. For example, if we have a sample of a trumpet played at A \flat , and the musician plays a D, we must shift the sample so it sounds like a D. There are two ways we could do this. The basic approach would be to perform **pitch shifting**, whereby we adjust the pitch of the sound but allow it to become shorter or longer. This is like playing a record or tape faster: person speaking on the tape is pitched higher but speaks much faster. The much more difficult alternative (without introducing noticeable artifacts in the sound) is **pitch scaling**, where the pitch is adjusted but the length is kept the same. Many samplers and romplers do pitch shifting.

The basic way to do pitch shifting is based on **resampling**. Resampling is the process of changing the sample rate of a sound: for example, converting a sound from 44.1KHz to 96KHz. We can hijack this process to do pitch shifting as follows. To shift a sound to twice its pitch (for example), we just need to squeeze the sound into half the time. To do this, we could resample the sound to half the sampling rate (cutting it to half the number of samples), then treat the resulting half-sized array as if it were a sound in the *original* sampling rate. Similarly, to shift the sound to half its pitch, we'd resample the sound to twice the sampling rate (generating twice the samples), and again treat the result as if it were in the original rate.

Downsampling To resample to a lower sampling rate is called **downsampling**. If the original sampling rate is an *integer multiple* of the new rate (for example, if we're downsampling to half or a third of the rate), then we just have to delete samples, retaining every Nth sample, a process known as **decimation**. For example, to cut to a third of the previous sampling rate, we remove two out of three samples, leaving every third sample. Before we do this we must make sure that the original sound didn't contain any partials above the **Nyquist limit** of the new sampling rate, or else we'd have aliasing in the end result. Thus before performing decimation we must first apply a **low pass filter** to remove those frequencies.

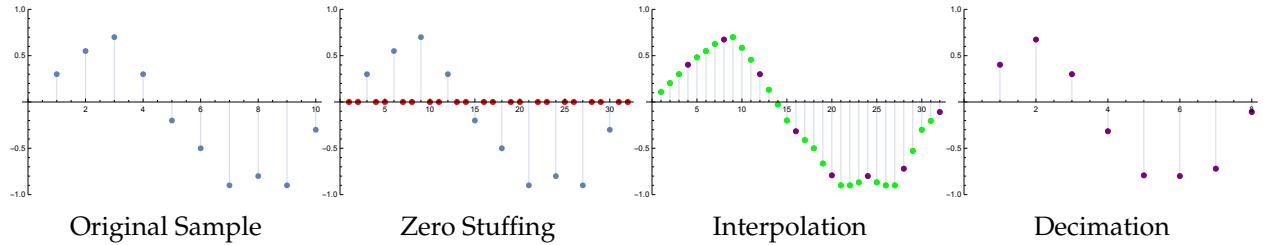


Figure 127 Resampling a sound 3/4 its previous sampling rate. The sound is first stuffed with **two zeros per sample**. The result is smoothed over with a low-pass filter to interpolate the zeros between the original samples (and to prevent frequencies over the final Nyquist limit). Then the sound is **decimated** to remove all but every **fourth sample**.

This all works because one consequence of the **Nyquist-Shannon sampling theorem** is that a continuous signal **bandlimited** to contain partials no higher than a frequency F uniquely passes through a set of discrete samples spaced $\frac{1}{2F}$ apart from one another. We're removing samples but the ones we retain still define the same basic sound, albeit at a lower rate.

Upsampling To resample to a higher sampling rate is called **upsampling**. If the new sampling rate is an integer multiple of the original rate (for example, we're upsampling to twice or three times the rate), then we need to insert new samples in-between the original samples, a process known as **interpolation**. Let's say we wanted to upsample to four times the original rate. Then we'd insert three dummy samples in-between each pair of the original samples. These dummy samples would initially have zero amplitude. To get them to smoothly interpolate between the originals, we could apply a **low pass filter** (yet again!), to smooth the whole thing. Note that this will likely reduce the gain of the sound, so we may need to amplify it again.

Resampling by Rational Values Now let's say that you needed to resample by a *rational value*. For example, you wished to shift from a sample rate of X to $\frac{a}{b}X$, where both a and b are positive integers. To do this, you'd first upsample by a factor of a , then downsample the result by a factor of b . Figure 127 shows this two-step process.

The problem is that small pitch shifts will require fractions of $\frac{a}{b}$ with large values of a or b or both, costing lots of memory and computational time. For example, if you wanted to shift up from C to C \sharp , this is an increase of $2^{1/12} \approx \frac{89}{84}$. That's a *very* rough approximation, and yet it would require upsampling to 89 times the sampling rate, then decimating by 84! Now imagine an even smaller pitch shift, such as via a slight tap of a pitch bend wheel: you could see even closer fractions. A common workaround is to figure out some way to break the fraction into a product of smaller fractions, and then do up/downsampling on each.¹⁰¹ For example, you could break up $\frac{56}{45} = \frac{7}{5} \times \frac{8}{9}$, then do upsample(7), downsample(5), upsample(8), downsample(9). Still very costly.

This technique is also inconvenient to use in real-time scenarios which demand rapid, dynamic changes in the sampling rate—such as someone moving the pitch bend wheel. We need a method which can do interpolation in floating point, so we can change sample rates dynamically, in real time, and without computing large fractions.

¹⁰¹Obviously you couldn't do that with $\frac{89}{84}$, because 89 is prime.

10.6 Basic Real-Time Interpolation

Consider the following very simple approach. Given a sound $A = \langle a_0, \dots, a_{s-1} \rangle$ at our desired sampling rate, but of frequency (pitch) P_A , we want to change its pitch to $P_{A'}$. To do this, instead of moving forward through A one step at a time, we'll move forward $\frac{P_{A'}}{P_A}$ (real-valued) “steps” at a time. Specifically, at timestep t we have a current *real-valued* position x^t in the sound, and to step forward, we set $x^{t+1} \leftarrow x^t + \frac{P_{A'}}{P_A}$. If we have a single-cycle or other looping wave, when x^t exceeds the number of samples s in the wave, set $x^t \leftarrow x^t \bmod s$ to wrap around to the beginning. At any rate, we return the sample $a_{\lfloor x^t \rfloor}$. If we are downsampling, we ought to first apply a low pass filter to the original sound to remove frequencies above Nyquist for the new effective sampling rate. This is the same as removing frequencies below $\frac{F_A}{2} \times \frac{\min(P_{A'}, P_A)}{P_A}$ in the original sound (where F_A is the original sound's sampling rate).

The problem with this method is that $\frac{P_{A'}}{P_A}$ may not be an integer, so this is a rough approximation at best: we're just returning the nearest sample. We could do a bit better by rounding to the nearest sample rather than taking the floor, that is, returning a_n where $n = \text{round}(x^t)$. All this might work in a pinch, particularly if we are shifting the pitch up, so $\frac{P_{A'}}{P_A}$ is large. But what if it's very small? We'd be returning the same value a_n over and over again (a kind of **sample and hold**). We need some way to guess what certain values would be *between* the two adjoining samples $a_{\lfloor x^t \rfloor}$ and $a_{\lceil x^t \rceil}$. We need to do some kind of real-time **interpolation**.

Recall that for a given set of digital samples there exists exactly one band-limited real-valued function (that is, one with no frequencies above Nyquist) which passes through all of them. Let's say that this unknown band-limited function is $f(x)$. What the sampling and interpolation task is *really* asking us to do is to find the value of $f(x)$ for any needed value x given our known samples $\langle a_0 = f(x_0), a_1 = f(x_1), \dots, a_n = f(x_n) \rangle$ at sample positions x_1, x_2, \dots, x_n .

The simplest approach would be to do to **linear interpolation**. Let's rename the low and high bracketing values of x^t to $x_l = \lfloor x^t \rfloor$ and $x_h = \lceil x^t \rceil$ respectively. Using similar triangles, we know $\frac{x - x_l}{x_h - x_l} = \frac{f(x) - f(x_l)}{f(x_h) - f(x_l)}$, and from this we get

$$f(x) = \frac{(x - x_l)(f(x_h) - f(x_l))}{x_h - x_l} + f(x_l)$$

This is just finding the value $f(x)$ on the line between the points $\langle x_l, f(x_l) \rangle$ and $\langle x_h, f(x_h) \rangle$. Linear interpolation isn't great: its first derivative is discontinuous at the sample points, as is the case for its generalization to higher polynomials, **Lagrange interpolation**.¹⁰²

An alternative is to interpolate with a **spline**: a chunk of a polynomial bounded between two points. Splines are often smoothly differentiable at the transition points from spline to spline, and

¹⁰²Named after the Italian mathematician **Joseph-Louis Lagrange**, 1736–1813, though he did not invent it. The goal is to produce a **Lagrange polynomial** which passes exactly through n points: you can then use that polynomial to find other points smoothly between them. To start, note that with a little elbow grease we can rearrange the linear interpolation equation to $f(x) = f(x_h) \frac{x - x_l}{x_h - x_l} + f(x_l) \frac{x - x_h}{x_l - x_h}$. It so happens that we can add a third sample f_m to the mix like this: $f(x) = f(x_h) \frac{(x - x_l)(x - x_m)}{(x_h - x_l)(x_h - x_m)} + f(x_l) \frac{(x - x_h)(x - x_m)}{(x_l - x_h)(x_l - x_m)} + f(x_m) \frac{(x - x_l)(x - x_h)}{(x_m - x_l)(x_m - x_h)}$.

Notice the pattern? In general if you have samples $x_1 \dots x_n$ available, then $f(x) = \sum_{i=1}^n \left(f(x_i) \prod_{j=1, j \neq i}^n \frac{x - x_j}{x_i - x_j} \right)$.

As mentioned, one problem with Lagrange interpolation is that it's not continuously differentiable at the sample points. If you have four sample points x_1, \dots, x_4 and you're interpolating from x_2 to x_3 everything looks great. But once you've reached x_3 and want to start interpolating to x_4 , you'd likely drop x_1 and add a new sample x_5 . But now the polynomial has changed, so it'll immediately launch off in a new direction: hence a discontinuity at x_3 .

they avoid another problem with Lagrange interpolation, namely unwanted oscillation. One simple spline approach is **cubic interpolation**. Let's say we had four points $\langle x_1, f(x_1) \rangle, \dots, \langle x_4, f(x_4) \rangle$ where the four x_i are evenly spaced from each other and increasing in value. That's certainly the case for our audio samples. We're trying to find $f(x)$ for a value x between x_2 and x_3 . Let α be how far x is relative to x_2 and x_3 , that is, $\alpha = (x - x_2)/(x_3 - x_2)$. Then

$$\begin{aligned} f(x) &= \alpha^3(-f(x_1) + f(x_2) - f(x_3) + f(x_4)) \\ &\quad + \alpha^2(2f(x_1) - 2f(x_2) + f(x_3) - f(x_4)) \\ &\quad + \alpha(-f(x_1) + f(x_3)) \\ &\quad + f(x_2) \end{aligned}$$

A better variation, based on the **Catmull-Rom** cubic spline, uses successive differences in $f(\dots)$ to estimate the first derivative for a potentially smoother interpolation.

$$\begin{aligned} f(x) &= \alpha^3(-1/2f(x_1) + 3/2f(x_2) - 3/2f(x_3) + 1/2f(x_4)) \\ &\quad + \alpha^2(f(x_1) - 5/2f(x_2) + 2f(x_3) - 1/2f(x_4)) \\ &\quad + \alpha(-1/2f(x_1) + 1/2f(x_3)) \\ &\quad + f(x_2) \end{aligned} \tag{6}$$

In both cases, at the very beginning and end of the sound, you won't have an x_1 or x_4 respectively: I'd just set $x_1 \leftarrow x_2$ or $x_4 \leftarrow x_3$ in these cases.

These interpolation schemes will produce smooth interpolation values, but the function they produce is *not quite* the actual band-limited function which passes through these points. You'll still get some distortion. And you still have to filter beforehand when downsampling to eliminate aliasing¹⁰³ But it turns out that there exists a method which will, at its limit, interpolate along the *actual* band-limited function, and acts as a built-in brick wall antialiasing filter to boot. That method is **windowed sinc interpolation**.

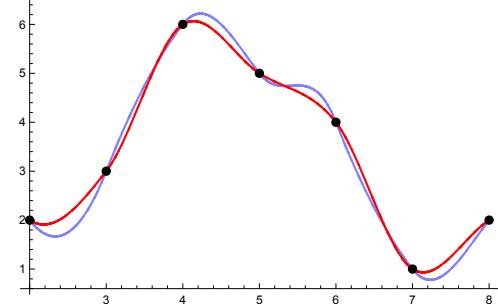


Figure 128 Basic **cubic spine interpolation** and **Catmull-Rom** interpolation. Note that Catmull-Rom is less "bouncy".

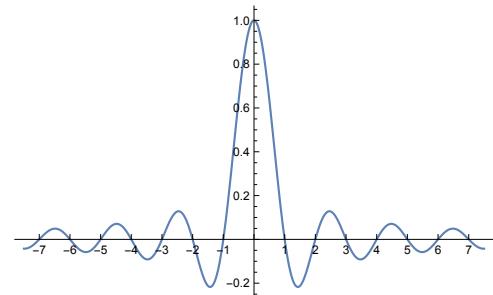


Figure 129 Sinc function.

¹⁰³There are lots of ways to optimize these polynomial interpolators to improve their sound quality. You might check out <http://yehar.com/blog/wp-content/uploads/2009/08/deip.pdf>

¹⁰⁴Sinc is pronounced "sink", and is a contraction of *sinus cardinalis*, (cardinal sine). There are two definitions of sinc, with and without the appearance of π . Sampling uses the one with π (the **normalized sinc function**) because its integral equals 1. Note that we define sinc to be 1 when $x = 0$ because the function divides by zero at that point otherwise. Does all this ring a bell? Look back at the variant of sinc used in Equation 2.

Interpolation with Sinc Recall that there is exactly one bandlimited continuous signal which passes through the points in our digital signal. Sinc is nicknamed the *sampling function* because, applying the **Whittaker-Shannon interpolation formula**, you can use sinc to reconstruct this continuous signal from your digital samples.

Let's say we wanted to retrieve the value of the continuous bandlimited signal $C(t)$ at time t . For now, assume that we have infinite number of samples $A = \langle a_{-\infty}, \dots, a_0, \dots, a_\infty \rangle$ sampled at a sampling rate of F_A . The timestep for sample a_k is k/F_A . For each such timestep, we center a sinc function (scaled by F_A) over that timestep, and multiply it by a_k . $C(\dots)$ is just the sum of all these sincs. That is:

$$C(t) = \sum_{k=-\infty}^{\infty} \text{sinc}(F_A \times (t - k/F_A)) \times a_k$$

This is **convolving** the sinc function against A , as shown in Figure 130. But notice that, because sinc is symmetric around zero, $\text{sinc}(F_A \times (t - k/F_A)) = \text{sinc}(F_A \times (k/F_A - t))$. This means we could instead write things as a **correlation** rather than convolution procedure:

$$C(t) = \sum_{k=-\infty}^{\infty} \text{sinc}(F_A \times (k/F_A - t)) \times a_k$$

This is equivalent but has a rather different interpretation: you can think of it as fixing a single sinc function so that it's centered at t . Then, for each sample a_k , we sample a *coefficient* from sinc at position k/F_A (Figure 131). Each of these coefficients is then multiplied by the corresponding a_k sample value and added up. I prefer this interpretation because it makes figuring the bounds (later on) more intuitive, and it's closer to how we did filters in Section 8.

Sinc is 0 for all integers except for 0, where it is 1. Thus when t lies right on top of one of our original samples, sinc will zero out the other samples and so $C(t)$ simply equals that one digital sample. Hence $C(t)$ describes a function which passes exactly through each of our digital samples.

Now consider: to resample, what we really want to do is reconstruct our continuous signal from the original samples, then sample from this continuous function at the new rate! To compute a sample position $a'_j \in A'$, where A' is our sound at the new sampling rate $F_{A'}$, the timestep t of a'_j is $t = j/F_{A'}$, so we get:

$$a'_j = \sum_{k=-\infty}^{\infty} \text{sinc}(F_A \times (k/F_A - j/F_{A'})) \times a_k \quad (7)$$

Now we can identify the new sample positions in A' and use this equation to compute them one by one.

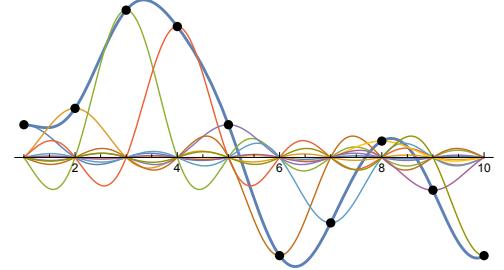


Figure 130 Convolution with Sinc. For each sample point, a sinc function is centered at that point and scaled vertically by the sample value at that point. The functions are added up, and the resulting function is the **bandlimited reconstruction** of the original signal.

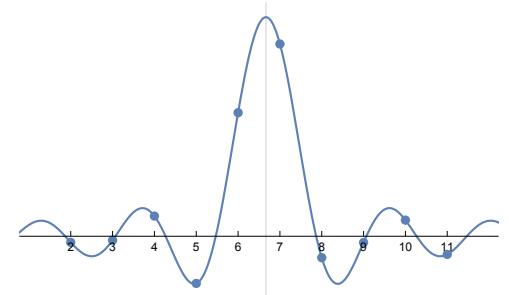


Figure 131 Sinc coefficients (five on a side) for computing a sample at time $t = 6 \frac{2}{3}$. Unwindowed.

When downsampling we need to make sure that the original signal contains no frequencies above the Nyquist limit for the new sampling rate. How can we do this? It so happens that convolution with sinc isn't just an interpolation function: it's also a perfect brick-wall low-pass filter (in theory at least, when we're summing from $-\infty$ to ∞). This is because convolution of two signals in the time domain does the same thing as multiplying the two signals in the frequency domain. And sinc's Fourier transform *just so happens* to be the (brick-wall) **rectangle function**:

$$\text{rectangle}(x) = \begin{cases} 1 & -0.5 \leq x \leq 0.5 \\ 0 & \text{otherwise} \end{cases}$$

To change the cutoff frequency, all we need to do is adjust the width of our sinc function. At present the multiplier F_A in Equation 7 ensures a filter cutoff at $F_A/2$, that is, the Nyquist limit for the original sound. But if we're downsampling, we need it to cut off at the (lower) Nyquist limit for the new sound. We do this by replacing F_A with $\min(F_A, F_{A'})$, like this:

$$a'_j = \sum_{k=-\infty}^{\infty} \text{sinc}(\min(F_A, F_{A'}) \times (k/F_A - j/F_{A'})) \times a_k$$

This will also change the overall volume, so to keep it a **unity gain filter**, we need to scale it back again by $\min(1, F_{A'}/F_A)$:

$$a'_j = \min(1, F_{A'}/F_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}(\min(F_A, F_{A'}) \times (k/F_A - j/F_{A'})) \times a_k$$

To simplify things later, let's pull out a $1/F_A$, like this:

$$a'_j = \min(1, F_{A'}/F_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(F_A, F_{A'})}{F_A} \times (k - F_A \times j/F_{A'})\right) \times a_k$$

Now let's define $J = F_A \times j/F_{A'}$. That is, J is the real-valued location of the new sample a'_j in the coordinate system of the original samples in A . This is the spot about which the sinc function is centered (for example, $J = 6^{2/3}$ in Figure 131), as is obvious when we substitute J into the equation:

$$a'_j = \min(1, F_{A'}/F_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(F_A, F_{A'})}{F_A} \times (k - J)\right) \times a_k$$

Windowing Of course, we don't have an infinite number of samples in our set A : at best we have $A = \langle a_0, \dots, a_{n-1} \rangle$, and even *that* could be an enormous convolution. We need to reduce this. Instead of convolving over the full range $\sum_{k=-\infty}^{\infty}$, maybe we could convolve over just a few nearby samples.

However, the sinc function goes out to infinity: we need it to drop to zero in a short, finite distance without just truncating it (which would sound awful). To do this, we can multiply it against a **window**. Windows were introduced in Section 3.5. We'd like a

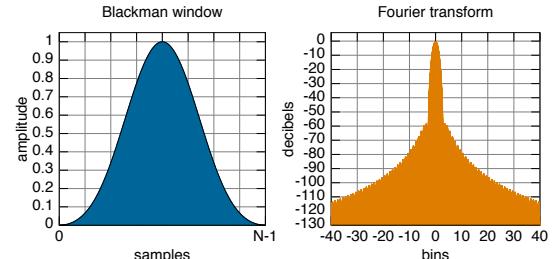


Figure 132 Blackman Window and its effects on the frequency domain (note that the Y axis for the FFT is on a log scale).⁶⁵ For more windows, see Section 3.5.

window which dropped completely to zero, such as the **Hann window**. But there's a somewhat better choice for our purposes: the **Blackman window**, as shown in Figure 132.¹⁰⁵ The Blackman window is applied over a region of length N . It is a function over $n \in 0...N - 1$:

$$w(n, N) = 0.42 - \frac{1}{2} \cos\left(\frac{2\pi n}{N-1}\right) + 0.08 \cos\left(\frac{4\pi n}{N-1}\right)$$

N is normally an odd number. Armed with a window, we could now replace the sinc with a *windowed sinc* which tapers off at $\pm(N - 1)/2$ using the window centered at J like sinc was (plus an offset of $\frac{N-1}{2}$ because Blackman isn't centered around 0):

$$a'_j = \min(1, F_{A'}/F_A) \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(F_A, F_{A'})}{F_A} \times (k - J)\right) \times w(k - J + \frac{N-1}{2}, N) \times a_k$$

Because all values in the sum outside the window region are 0, we can now make the sum finite. So what should our upper and lower bounds be? They should be the outermost sample positions just inside the window taper region. That is, $k_{\text{low}} = \left\lceil J - \frac{(N-1)}{2} \right\rceil$ and $k_{\text{high}} = \left\lfloor J + \frac{(N-1)}{2} \right\rfloor$, thus:

$$a'_j = \min(1, F_{A'}/F_A) \sum_{k=k_{\text{low}}}^{k_{\text{high}}} \text{sinc}\left(\frac{\min(F_A, F_{A'})}{F_A} \times (k - J)\right) \times w(k - J + \frac{N-1}{2}, N) \times a_k$$

And we're done! Hint: if you're trying to shift the pitch from P_A to $P_{A'}$, then $F_{A'} = F_A \times P_A / P_{A'}$. The quality of this approach will largely turn on the size of N , which in turn impacts directly on computational power.¹⁰⁶ It also will impact on the **latency** of the algorithm: because sinc is not a **causal filter**, we must know some of the incoming future sound samples. For much sampling this is probably not an issue, as we probably already have entire PCM sample or the wavetable available to us. But if you were using this method to do (say) pitch-shifting of incoming sounds, you should be aware of this. For example, if you were using 44 sinc coefficients per side on a 44.1KHz sound, the delay would be about $(44, 1000/44 \times 1000) \approx 1$ millisecond.

¹⁰⁵In the literature you'll see much better windows still, notably the **Kaiser window**, but they are difficult to describe and even tougher to implement (Kaiser is based on Bessel functions).

¹⁰⁶CCRMA has a more efficient table-driven windowed sinc interpolation algorithm you would do well to check out. https://ccrma.stanford.edu/~jos/pasp/Windowed_Sinc_Interpolation.html For a simple implementation of the algorithm described here (more or less) see <http://www.nicholson.com/rhn/dsp.html#3> That implementation is translating things so that k_{low} and k_{high} are centered around 0.

11 Effects and Physical Modeling

Most of this text is concerned with the *creation* or *sampled playback* of sounds. But another important aspect are algorithms meant to add some **effect** to a sound to enhance it. The sound being fed into an effect doesn't have to come from a synthesizer or sampler: in fact it's often a live sound like vocals or an instrument. The goal of an effect is to make the sound feel *better* or *different* somehow.

Some of the algorithms we've covered so far qualify as effects in and of themselves, and can be found in guitar pedals and other devices: for example, filters, ring modulation, clipping, and other distortion mechanisms. But many popular effects rely on some kind of some kind of **time delay** to do their magic. These are the bulk of the effects covered in this Section. In some of these effects (delay, reverb) the delays are long and so are perceived as shifts in time; but in other effects (chorusing, flanging) the delays are very short and are instead perceived as changes in timbre.

The Section concludes with a short introduction to **physical modeling synthesis**, an array of techniques for modeling the acoustic and physical properties of certain instruments. Physical modeling is lumped in with effects in this Section because its methods often apply similar delay-based techniques.

11.1 Delays

One of the simplest time-based effects is the humble **delay**. Here, the sound is augmented with a copy of itself from some m timesteps before. A one-shot delay is quite easy to implement: it's essentially the extension of an FIR filter, with a delay portion significantly longer than a single sample, as shown in Figure 133. The delay portion, commonly known as a **digital delay line**. If you recall from Section 8.7, a delay of one sample is often referred to as z^{-1} , as in the module $[z^{-1}]$. Similarly, a long delay line of m samples would be referred to as z^{-m} . This is very easily implemented as a **ring buffer**:

Algorithm 20 Delay Line

- 1: $x \leftarrow$ incoming sample
- 2: Global $B \leftarrow \langle b_0, \dots, b_{m-1} \rangle$ buffer (array of m samples), initially all 0
- 3: Global $p \leftarrow$ position in buffer, initially 0
- 4: $y \leftarrow b_p$
- 5: $b_p \leftarrow x$
- 6: $p \leftarrow p + 1$
- 7: **if** $p \geq m$ **then**
- 8: $p = 0$
- 9: **return** y

Note from Figure 133 that you can cut down the amplitude of both the original and delayed signal. The degree to which you cut down one or the other defines how **dry** or **wet** the signal is. A fully dry signal is one which has no effect at all (the delay is cut out entirely). A fully wet signal is one which has *only* the effect.

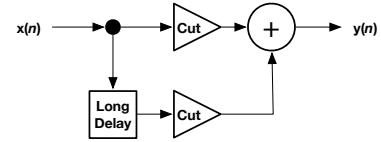


Figure 133 One-shot delay. Compare to the FIR filter in Figure 92.

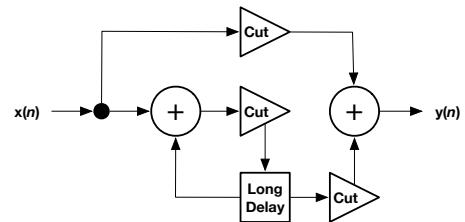


Figure 134 Repeated delay, augmented with two additional cut gains to control wetness. Compare to the basic IIR filter in Figure 93.

What if you wanted a repeating delay? This is also easy: you just need the equivalent of an extended feedback (that is, IIR) filter. The cut-down is particularly important, because if we don't cut down enough, the recurrent nature of this delay will cause it to spiral out of control. Figure 134 shows this delay core, augmented with two outer cut-downs to make it easy to control wetness.

There are lots of variations on delays: you could ping-pong the delay back and forth in stereo, or sync the delay length to the MIDI clock so the delays come in at the right time. Perhaps you might pitch-shift the delay or repeatedly run it through a low-pass filter.

11.2 Flangers

While delay effects involve *long* delays, other effects involve rather *short* delays which are perceived not as delays but as changes in the spectrum of the sound. A classic example of this is the **flanger**. This is an effect whose characteristic sound is due to a signal being mixed with a very short delayed version of itself, where the degree of delay is modulated over time via an LFO, perhaps between 1 and 10ms.

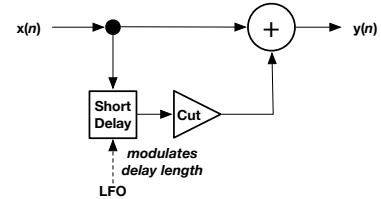


Figure 135 LFO-modulated flanger

Comb Filters When a delay line is very short, as is the case in a flanger, we don't hear a delay any more. Rather we hear the effect of a **comb filter**. One kind of comb filter, a **forward comb filter**, is a simple extension of the classic FIR filter with a longer delay: it takes the form

$$y(n) = b_0 x(n) + b_1 x(n - m) \quad (8)$$

where m is the length of the delay in samples. We'll assume that $b_0 = 1$. Notice the repeated lobes in the comb filter in Figure 136. A larger value of m will result in more of these lobes.¹⁰⁷ You can also see how setting b_1 to different values changes the wetness of the filter.

A comb filter is most easily described in the Z Domain where, with $b_0 = 1$, its transfer function is

$$H(z) = 1 + b_1 z^{-m} = \frac{z^m + b_1}{z^m}$$

From this you can see that the filter will have m poles and m zeros. The poles all pile up at the origin, while the zeros are spaced evenly just inside the unit circle.¹⁰⁸ It is this even spacing which creates the lobes in the magnitude response:

$$|H(e^{i\omega})| = \sqrt{(1 - b_1)^2 + 2b_1 \cos(\omega m)}$$

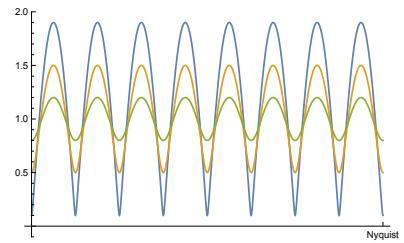


Figure 136 Forward comb filter with $b_0 = 1$ and b_1 set to -0.9, -0.5, and -0.2.

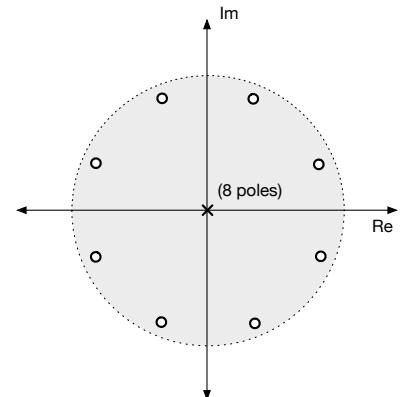


Figure 137 Poles and zeros of a forward comb filter, $m = 8$, $b_1 = -0.5$, in the Z domain.

¹⁰⁷Indeed, if $m = 1$, then we have a standard low-pass or high-pass filter.

¹⁰⁸You might ask yourself what a comb filter would look like in the continuous (Laplace) domain. Since this domain can go to infinity in frequency, a *proper* comb filter would wind up with an infinite number of poles and zeros. That's probably not reasonable to implement.

The **feedback comb filter**, which is the extended equivalent to a basic IIR filter, is just as simple. It takes the form

$$y(n) = b_0x(n) + a_1y(n - m)$$

Again, we may assume that $b_0 = 1$, and so the transfer function, in the Z Domain, is just

$$H(z) = \frac{1}{1 - a_1z^{-m}} = \frac{z^m}{z^m - a_1}$$

Notice how close this is to an inverse of the forward version. It wouldn't surprise you, then, to find that the feedback comb filter has its *zeros* all at the origin and its *poles* spaced evenly just inside the unit circle, the exact opposite of the forward comb filter. The net result of this is that the magnitude response is

$$|H(e^{i\omega})| = \frac{1}{\sqrt{(1 - a_1)^2 - 2a_1 \cos(\omega m)}}$$

This *sort of* resembles the forward comb filter turned upside down, as shown in Figure 138.

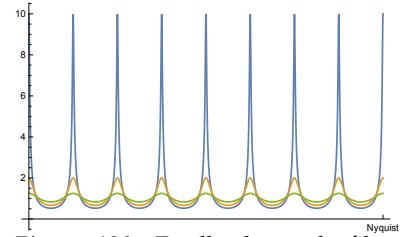


Figure 138 Feedback comb filter with $b_0 = 1$ and a_1 set to -0.9, -0.5, and -0.2.

Fractional Delays So far we've described m as being an integer. But a flanger's LFO must smoothly change the length of the delay, and so m would benefit greatly from being a floating-point value. This means we need a delay which *interpolates* between two sample positions.

A simple way to do this is **linear interpolation**. Let $\alpha = m - \lfloor m \rfloor$. That is, α is a value between 0 and 1 which describes where m is with respect to the integers on either side of it. Now we could modify Equation 8 to roll in a bit of each of the samples on either side of m , that is:

$$y(n) = b_0x(n) + (1 - \alpha)b_1x(n - \lfloor m \rfloor) + \alpha b_1x(n - \lceil m \rceil)$$

Linear interpolation isn't very accurate: and it's particularly bad at delaying high frequencies. There exist more sophisticated interpolation options, as discussed in Section 10.6. Or you could hook a time-varying **all pass filter** to the end of your delay line. We'll discuss all pass filters coming up, in Section 11.4.

11.3 Chorus

Chorus is another short-delay effect which sounds like many copies of the same sound mixed together. And that is basically what it is: the copies are varied in pitch, amplitude, and delay. One easy way to implement this effect is with a **multi-tap delay line**. This is a delay which outputs several different positions in the buffer. It's pretty straightforward:

Algorithm 21 Multi-Tap Delay Line

```

1:  $x \leftarrow$  incoming sample
2:  $A \leftarrow \langle a_0, \dots, a_{q-1} \rangle$  tap positions, each from 0 to  $n - 1$   $\triangleright q \ll n$ 
3: Global  $B \leftarrow \langle b_0, \dots, b_{n-1} \rangle$  buffer (array of  $n$  samples), initially all 0
4: Global  $p \leftarrow$  position in buffer, initially 0
5:  $T \leftarrow \langle t_0, \dots, t_{q-1} \rangle$  results (array of  $q$  samples)
6: for  $i = 0$  to  $q - 1$  do  $\triangleright$  Load taps
7:    $j \leftarrow p + a_i \bmod n$ 
8:    $T_i \leftarrow B_j$ 
9:    $b_p \leftarrow x$   $\triangleright$  Update sample as was done in Delay Line (Algorithm 20)
10:   $p \leftarrow p + 1$ 
11: if  $p \geq n$  then
12:    $p = 0$ 
13: return  $T$ 

```

Like flanging, chorusing likewise would benefit from an interpolated delay line so the tap positions don't have to be integers. It's not difficult to modify the previous algorithm to provide that.

Doppler Effect Clearly we can use this to create different delay lengths (longer than a flanger: perhaps up to 50ms). And we can multiply each of these outputs by their own gain to create different amplitudes. But how can we shift the pitch up and down? This turns out to be easy: just move the tap positions back and forth at different speeds, controlled by an LFO. The speed at which the tap position is being moved will effectively compress or stretch the wave and thus change its pitch.¹⁰⁹ Of course you can only move the tap position so far, so at best you can shift it back and forth, thus changing the pitch slightly up and down.

Shifting the pitch by moving the tap position is essentially simulating the **Doppler effect**, where sounds from objects moving rapidly towards a listener sound higher pitched than they should be, and similarly lower pitched when moving rapidly away: you may have heard this effect as an ambulance rushes by you with its sirens blaring. One use of this is simulating a **rotary speaker** such as the famous **Leslie speaker** attached to the **Hammond Organ**. This was a speaker horn which spun in place, so that at one extreme it was facing the listener and at the other extreme it was facing away. Because the horn was loudest when facing the listener (of course), this resulted in **tremolo**. Additionally, the rapid movement of the speaker horn produced **vibrato** due to the Doppler effect.

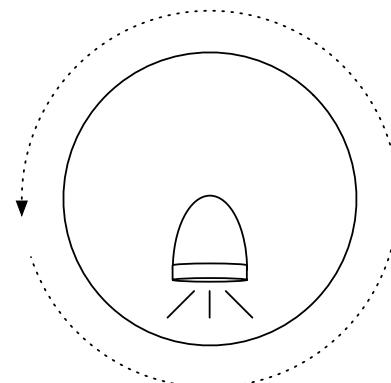


Figure 139 Leslie speaker mounted on a spinning turntable. As the speaker rotates away or toward the listener, the sound wobbles in pitch (vibrato) due to the Doppler Effect, and the sound is alternately blaring or muffled (when facing away), creating tremolo.

¹⁰⁹Yes, that's basically FM.

11.4 Reverb

Reverb, or more properly **reverberation**, attempts to replicate the natural echoes which occur in an enclosed space. These aren't simple delay echoes: there are a very, very many of them and they are affected by the nature of the surfaces involved and the distance from the listener. Furthermore, echoes may bounce off of many surfaces before arriving at the listener's ear.

It's common to model a reverb as follows. For some n timesteps after a sound has been produced, there are no echoes heard at all: sound is slow and hasn't travelled the distance yet. Then come a small collection of **early reflections** which have bounced directly off of surfaces and returned to the listener. Following this come a large, smeared set of **late reflections** which result from the sound bouncing of many surfaces before returning.

Early reflections are more or less multiple arbitrarily-spaced delays and hence are straightforwardly implemented with a multi-tap delay line. Late reflections are more complex: if you implemented them with very short delays (comb filters, say), the result would sound artificial. Better would be to find a way to have different delay lengths for frequencies in the sound, to create a smearing effect. Enter the **all pass filter**.

All-Pass Filters An *all pass filter* is a strange name: why would we want a filter which doesn't change the amplitude of our partials? The reason is simple: the amplitude is left alone, but the *phase* is altered in some significant way. And altering the phase is just adding a small, real-valued delay to the signal. Importantly, this delay can be *very* small, even less than a single sample, and different different frequencies can be (and are) delayed by *different amounts*.

There are many ways to achieve an all-pass filter, but perhaps the simplest is to intertwine two comb filters, as shown in Figure 140. For a delay of length m samples, this is realized as

$$y(n) = b_0x(n) + x(n - m) - b_0y(n - m)$$

This has the Z Domain transfer function¹¹⁰

$$H(z) = \frac{b_0 + z^{-m}}{1 + b_0z^{-m}}$$

This transfer function has an even, 1.0 magnitude response, and its phase response is 0 degrees at 0 Hz, dropping as the frequency increases.

All-pass filters can be strung together in serial, or put in parallel, or interestingly, nested inside one another as shown in Figure 141.

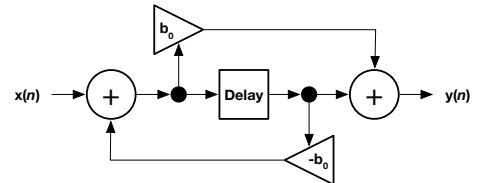


Figure 140 A simple all-pass filter consisting of two intertwined comb filters. Note that the coefficients are the negatives of one another.

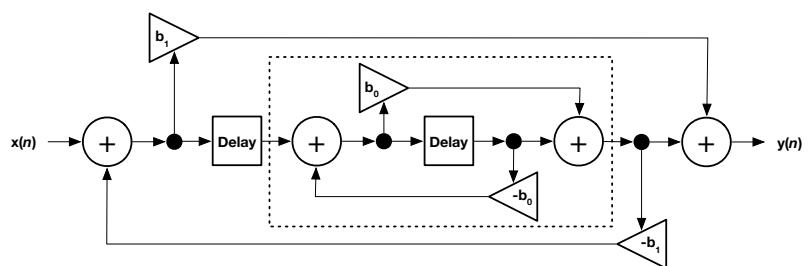


Figure 141 One all-pass filter nested inside another.

¹¹⁰This all assumes that b_0 is a real value, which would be the case in audio applications. If not, then the equation is $y(n) = b_0x(n) + x(n - m) - \bar{b}_0y(n - m)$, with $H(z) = \frac{b_0 + z^{-m}}{1 + \bar{b}_0z^{-m}}$, where \bar{b}_0 is the complex conjugate of b_0 .

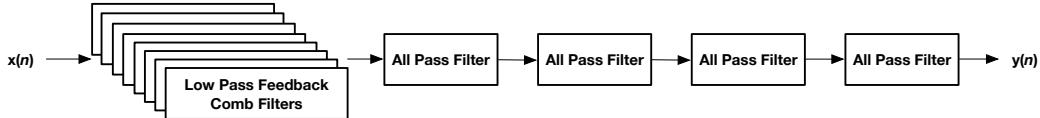


Figure 143 Freeverb architecture

Putting it Together Armed with multi-tap delay lines, comb-filters, and all-pass filters, we have enough material to string together to form a reverberation algorithm. This algorithmic approach is often called **Schroeder reverberation** after **Manfred Schroeder**, an early pioneer of the technique. There are *lots* of ways to arrange these elements, but here's one example architecture.

Freeverb is popular open source reverb implementation by **Jeremy "Jezar" Wakefield**. In this architecture, the input is handed to a bank of eight parallel **low-pass feedback comb filters**. These are just comb filters where a low-pass filter has been inserted in the feedback loop, as shown in Figure 142, to cut down the high frequencies on successive passes. The output of these filters are added up and then passed through a series of all-pass filters which smear the results. The parameters of the comb filters are tuned to be different from one another so as to provide a variety of echoes; similarly, the all pass filters are all tuned to be different from one another. Freeverb has user parameters for "room size" (essentially the delay length), dampening (low-pass cutoff), and of course wetness.¹¹¹

Convolution Reverb A popular alternative approach to the algorithmic reverbs shown so far is to directly sample the reverberation pattern of an environment and apply it to the sound. This approach is called a **convolution reverb**. The idea behind this is actually surprisingly simple. We first sample the echoes resulting from an **impulse** (a single loud, extremely short sound, such as a balloon popping), and then apply these echos over and over again in response to every single sample in our sound. These echoes are known as the **impulse response** of the environment.

If we treated the impulse as a single sample of maximum volume, then the echoes from the impulse could be thought of as the effect that this single input sample has on the volumes of future output samples in the final sound. But if we *reversed* the impulse, we could think of it as the echoed effects of all *previous input samples* on the current output sample. For example, let $e(k)$, $0 \leq k \leq N$ be the impulse response, where $e(0)$ is the original impulse and $e(k)$, $k > 0$ are future echoes. By reversing this, we can gather the echoes from earlier samples and sum them into the current sound:

$$y(n) = \sum_{k=0}^N x(n-k) \times e(k)$$

...where $x(n)$ is the sound input and $y(n)$ is the resulting output with reverb. Obviously we should **zero pad**: if $n - k < 0$ then $x(n-k) \times e(k) = 0$. This equation should look very similar to the **convolution** equations found in Section 8.1: indeed the impulse response is essentially being used as a very long **finite impulse response filter**.¹¹²

¹¹¹I'm not providing the details of these parameters here: but you can examine them, and other architectures, at <https://ccrma.stanford.edu/~jos/Reverb/Reverb.html>

¹¹²Now finally it should make sense why FIR is called a finite *impulse response* filter.

This sampling approach cannot be tuned with a variety of parameters like the Schroeder algorithmic approach can. However, it has the advantage of providing a nearly exact reproduction of the reverberation qualities of a chosen environment. Convolution reverb is expensive, however. If the impulse response sound is N samples long, then adding reverb to M samples of the original sound is $O(MN)$ for long reverbs.

There's a faster way to do it: we can use the FFT! A critical feature of the FFT is that *convolution in the time domain is exactly the same thing as multiplication in the frequency domain*. To start, let's zero-pad the impulse response to be as long as the sound, that is, we set things up so that $M = N$. Let's call the impulse response $e(t)$ and the sound $s(t)$. We take the FFT of the original sound to produce $S(f)$, and similarly the FFT of the reversed impulse response to produce $E(f)$. Next, we multiply the two, that is, for each value f , the result $R(f) = S(f) \times E(f)$. Finally, we take the inverse FFT of $R(f)$ to produce the final resulting sound $r(t)$.

So let's count up the costs: an FFT is $O(N \lg N)$, and so is an IFFT. On top of that, we're doing N multiplies. Overall, this is $O(N \lg N)$, which is likely much smaller than the $O(MN)$ required by direct convolution. Clever! But this means we have to apply reverb in bulk to the entire sample. That won't do.

Instead, if $M \ll N$, we could perform the **Short Time Fourier Transform** or **STFT**. This is little more than breaking a sound into chunks and then doing an FFT on each chunk. To get things right, our chunks ought to overlap by 50%, like bricks in a brick wall. That is, if our first chunk of our sound is $0 \dots M - 1$, then our next chunk is $1/2M \dots 3/2M - 1$, the next chunk is $M \dots 2M - 1$, and so on. The general approach would be to break into these overlapping chunks of size M , perform the FFT and IFFT trick with the reversed impulse response on each of them in turn, then reassemble them by adding them together using the **Overlap-Add Method** as shown in Figure 144.

You'd think that adding overlapped chunks in this way would cause smearing, but it will reassemble properly as long as, prior to performing the FFT, we have *first* windowed each chunk with a window which satisfies the **Constant Overlap-Add** or **COLA** property. Windows with this property include the **triangular window** (or **Bartlett window**), shown in Figure 145, as well as the **Hann window**, shown earlier in Figure 125 in Section 10.4, among others.¹¹³

But even *this* is not enough, because even though $M \ll N$, it's still quite large: to process a sample would require an $O(M)$ delay. Thus convolution reverb algorithms typically break both the sound *and the impulse response* into even smaller chunks to reduce the delay. By using variable length chunks and other clever optimizations the delay can be largely dealt with.¹¹⁴

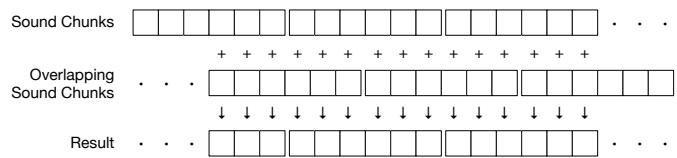


Figure 144 Overlap-Add procedure.

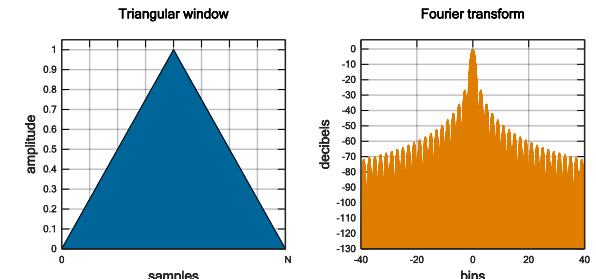


Figure 145 Triangular or Bartlett Window.¹¹⁵

¹¹³I'm not saying the Triangular and Hann windows are the best choices: simply that they are easy examples.

¹¹⁴For a practical introduction, see <https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/convolution.html>

11.5 Phasers

A phaser produces an effect very similar to a flanger. The main difference is that while the flanger's comb filter results in evenly sized and spaced lobes, a phaser's lobes change in size with increasing frequency, often exponentially as shown in Figure 146. Modulating these lobe positions with an LFO produces the distinct **phaser** effect.

A phaser is typically implemented with a long string of all-pass filters with different sizes tuned to provide the phaser's various peaks and troughs when remixed with the original sound. Figure 147 shows one possible implementation.

While an all-pass filter only modifies the phase of its signal (and we generally can't detect that unless it is extreme), this creates interference patterns when added back into the original signal, and if carefully tuned, can produce phaser and other lobe patterns.

Typically two all-pass filters are needed per lobe, and this may result in the need for quite a number of them altogether.¹¹⁵

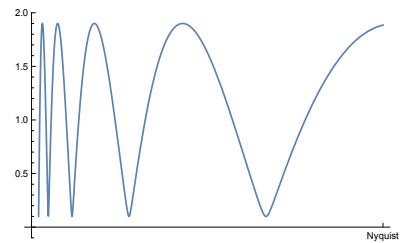


Figure 146 Phaser amplitude effect

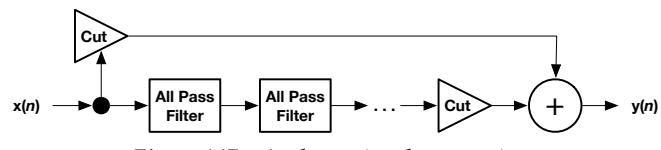


Figure 147 A phaser implementation.

11.6 Physical Modeling Synthesis

Physical modeling synthesis is a cutting-edge approach to realistically reproducing instruments by roughly approximating how they vibrate and work as a physical system. Interestingly, the basic building blocks of physical modeling synthesis are often the same as those found in time-based effects: different kinds of delays and filters.

One of the earlier, simpler, and most well-known physical modeling methods is the **Karplus-Strong** algorithm, named after **Kevin Karplus** and **Alexander Strong**. This algorithm attempts to replicate a plucked string such as on a violin or guitar. The basic algorithm is really quite simple:

Algorithm 22 Karplus-Strong String Synthesis

- 1: Global $B \leftarrow \langle b_0, \dots, b_{m-1} \rangle$ buffer (array of m samples), initially all random noise
- 2: Global $p \leftarrow$ position in buffer, initially 0
- 3: Global $y' \leftarrow$ previous y output, initially b_{m-1}
- 4: $y \leftarrow b_p$
- 5: $b_p \leftarrow 1/2 y + 1/2 y'$
- 6: $p \leftarrow p + 1$
- 7: **if** $p \geq m$ **then**
- 8: $p \leftarrow 0$
- 9: $y' \leftarrow y$
- 10: **return** y

¹¹⁵For one such implementation, see https://ccrma.stanford.edu/realsimple/DelayVar/Phasing_First_Order_Allpass_Filters.html

This should look very familiar: it's closely related to the basic **digital delay line** (Algorithm 20). But unlike a delay line, Karplus-Strong's delay buffer starts filled with random noise. Furthermore, as the buffer is drained it is not refilled with an input sound (in fact there is no input sound) but with a modified version of the latest output. Specifically, the buffer is filled with the average of the most recent output and the output sample immediately before that. See Figure 148. Thus if we have a buffer of length N , then Karplus Strong is roughly the equation

$$y(n) = 1/2 y(n - N) + 1/2 y(n - N - 1)$$

The size N determines the frequency f of the sound. Specifically, $N = f_r/f$, where f_r is the sampling rate. The idea behind Karplus Strong is that a string, when plucked, is initially filled with high-frequency sound, but very rapidly this sound loses its high frequencies until, at the end, it's largely a sine wave. The high frequencies are lost due to the averaging: notice that the averaging is basically a one-pole low-pass filter. Figure 149 shows this effect.

There are some issues. First N is an integer, and this will constrain the possible frequencies. There exist ways to permit any frequency through (what else?) the judicious use of all-pass filters. Second, high frequency sounds will decay faster than low-frequency ones because the buffers are smaller and so all the samples pass through the filter more often. Adjusting this per-frequency can be challenging. One can *shorten* the die-off very easily by replacing the $1/2$ in the equation $y(n) = 1/2 y(n - N) + 1/2 y(n - N - 1)$ with some smaller fraction. Lengthening is more complex. Note that making any adjustments at all may be unnecessary: in real plucked instruments it's naturally the case for high frequency notes to decay faster anyway.¹¹⁶

Traveling Waves One interpretation of Karplus-Strong's delay line is as a poor man's simulation of a **traveling wave** in a plucked string. When a string is plucked, its wave doesn't stay put but rather moves up and down the string; and indeed there are *two* waves moving back and forth, as shown in Figure 150. Karplus-Strong might be viewed as a model of one of these waves as it decays. But

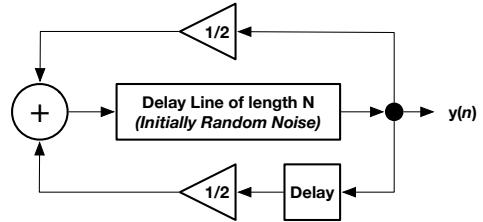


Figure 148 The Karplus-Strong algorithm.

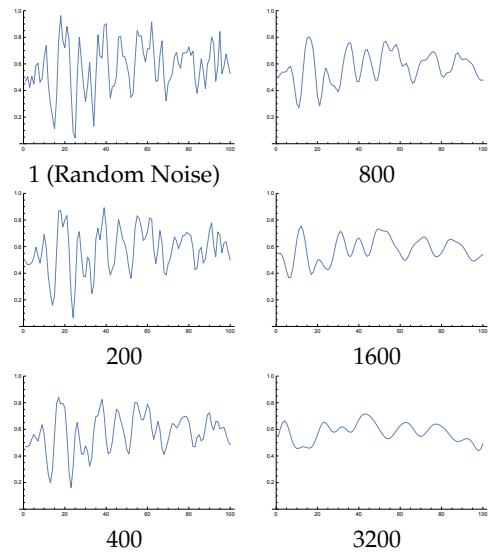


Figure 149 Karplus-Strong buffer (size 100), after various iterations of the algorithm. Note the gradual impact of the low-pass filter.

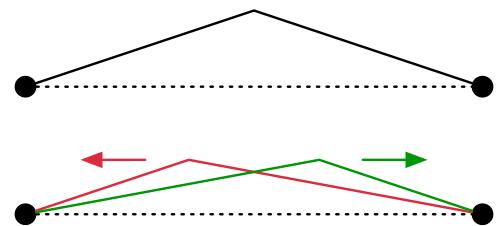


Figure 150 Traveling waves in a plucked string. (Top) The string is initially plucked, pulled away as shown. (Bottom) The wave separates into two traveling waves going opposite directions, towards the endpoints. After reaching the endpoints, the waves invert and reflect back (then reflect back again when reaching the opposite endpoint, and so on).

¹¹⁶For hints on how to deal with both of these issues, see David A. Jaffe and Julius O. Smith, 1983, Extensions of the Karplus-Strong plucked-string algorithm, *Computer Music Journal*, 7(2). This paper also suggests improving the basic algorithm by adding a variety of low-pass, comb, and all-pass filters in the chain.

more sophisticated models of strings use two waves as part of a **waveguide** network. Traveling waves don't just appear in strings: they also occur in the air in tubes or pipes, such as woodwinds, brass, organs, and even the human vocal tract. Modeling waves with waveguide networks has given rise to a form of physical modeling synthesis known as **digital waveguide synthesis**, where elaborate models of waveguides can be used to closely simulate plucked or bowed strings, blown flutes or reed instruments, voices, and even electric guitars.

A bidirectional digital waveguide can be simulated with two multi-tap delay lines as shown in Figure 151. Here's the general idea. Each delay line represents a traveling wave in one direction. When sound exits the delay line, it is considered to have reached the end of the string and is being reflected back. To do this, the sound is first inverted (using a gain of -1) and slightly dampedened with a low pass filter — perhaps with something better than the averaging filter used in Karplus-Strong. Then the sound is fed back into the other delay line to go the other direction.

An excitation $x(n)$ is added into both delay lines at some symmetrical point, that is, if the delay lines are m long, it might be added in at positions a and $m - a$ respectively. This could be an initial impulse noise as in Karplus-Strong, or perhaps some continuous input wave to simulate excitation due to continuously bowing the string at a certain spot. The final sound $y(n)$ is also tapped at some symmetrical point in the delay lines (perhaps at the notional location of the instrument's sound hole), summed, and outputted.

This is a very simple model. There are much more sophisticated ones available involving networks of pairs of delay lines connected via different kinds of junctions to transfer sound back and forth, in order to model surprisingly complex instruments.¹¹⁷

Commercial synthesizers which incorporate these capabilities are not very common: perhaps the most famous in history is the **Yamaha VL1**, a **duophonic** (two-voice) physical modeling keyboard (Figure 152). Physical modeling synthesizers could produce amazing sounds, but the physical modeling revolution did not take hold in the late 1990s. I believe this was likely due to competition with romplers: why spend all that computational effort developing a beautiful sounding model of a shakuhachi when you could just play an adequate pre-sampled one? Modern physical modeling synthesizers are, not surprisingly, all softsynths. Among the most successful are **Audio Modeling's SWAM engine**, which combines physical modeling, rompling, and some other tricks, to very accurately reproduce woodwinds and strings.

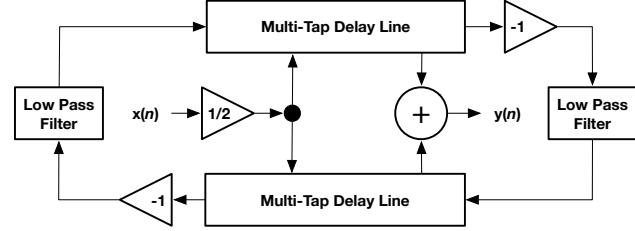


Figure 151 Bidirectional waveguide model of two traveling waves. A gain of -1 , shown twice in this model, means that the signal is inverted.



Figure 152 Yamaha VL1.^{©67}

¹¹⁷A good source of advanced techniques in this area, as well as delay-based effects, is *Physical Audio Signal Processing* by **Julius Smith**, available online at <https://ccrma.stanford.edu/~jos/pasp/>

12 Controllers and MIDI

A **controller** is a device which enables a human to control the notes or parameters of synthesizer in a useful way. Early synthesizer designs incorporated controllers such as keyboards and pedals as part of the system. However with the advent of the **Musical Instrument Digital Interface** or **MIDI**, which enabled one device to remotely control another one, the keyboard and the synthesizer began to part ways. Many synthesizers became simple rackmount devices intended to be manipulated by a controller of one's choosing; and the market began to see controller devices which produced no sound at all, but rather sent MIDI signals intended for a downstream synthesizer. We'll discuss MIDI in Section 12.2.

With the advent of the computer and the **Digital Audio Workstation** we have seen another sea change: controllers which do not send MIDI to a synthesizer, but rather directly to computer software which then either records it or routes it to a software or hardware synthesizer. Indeed many of the cheap controllers found on the market nowadays are outfitted only with USB jacks rather than traditional 5-pin MIDI jacks, and intended solely for this purpose.

Controllers are essentially the user interface of synthesizer systems, and so it is critical that they be designed well. A primary function of a good user interface is to help the musician achieve his goals or tasks as easily, accurately, and rapidly as possible. Playing music is an operation involving changing multiple parameters (pitch, volume, many elements of timbre, polyphony) in real-time, and significant effort in musical interface design has been focused on new ways or paradigms to enable a musician to control this complex, high-dimensional environment intuitively with minimal cognitive load.

12.1 History

Keyboards Among the earliest controllers have undoubtedly been **keyboards**. The modern keyboard is perhaps five hundred years old, dating largely from organs, and later migrating to harpsichords and clavichords.¹¹⁸ These instruments all shared something in common: their keys were essentially switches. No matter how hard you struck a key, it always played the same note at the same volume. A major evolution in the keyboard came about with the **pianoforte**,¹¹⁹ nowadays shortened to **piano**. This instrument hit strings with a felt hammer when a key was played, and critically the **velocity** with which the key was struck translated into the force with which the string was hit, and thus the volume with which the note was sounded.

This critical difference caused piano keyboards to deviate from organ keyboards in their **action**. The action is the mechanics of a keyboard which cause it to respond physically to being played. Early on, Bartolomeo Cristofori (the inventor of the pianoforte) developed an action which resisted being played because playing a key required lifting a weight (the hammer). Because the key didn't just give way immediately on being struck, it formed a kind of force-feedback which helped the performer to "dial in" the amount of volume with which he wanted a note to play. As pianos developed more and more **dynamic range**¹²⁰ this resistive **weighted action** became more and more detailed in order to serve the more sophisticated needs of professional pianists. Organs never adopted a weighted action because they didn't need to: organ keyboards have no dynamic range. Typical organ actions are **unweighted**: the keys give way almost immediately upon being struck.

¹¹⁸In case you were wondering what the difference was between the two: when a note was struck, a harpsichord plucked a string, while a clavichord would hit it with a small metal **tangent**.

¹¹⁹Italian for "soft-loud".

¹²⁰The difference between the loudest possible note and the softest.

Modern synthesizer keyboards traditionally have unweighted actions because early synthesizers, like organs, had no dynamic range; but these unweighted keyboards perhaps stuck around in the synth world because unweighted actions made for cheaper synthesizers. This is a strange fit because modern synthesizers, like pianos, are largely **velocity sensitive** and so have a significant dynamic range. Even more unfortunate is the recent popularity of cheap **mini key** keyboards (see Figure 154) whose **travel** (the distance the key moves) is significantly reduced, or **membrane** or **capacitive “keyboards”** with no travel at all. Such keyboards make it even more difficult, if not impossible, to dial in precise volume, much less play notes accurately. There are other synthesizer keyboards, known as **weighted keyboards**, which simulate the weighted action of a piano. Like pianos, such keyboards vary in how *much* resistive weight they impart, depending on the performer’s tastes.

Simple Expressive Manipulation As synthesizers became more sophisticated, with more and more parameters which could be changed in real time, it became clear that simple velocity-sensitive weighted keyboards were crude tools for providing expressive control over these parameters. The first major attempt to remedy this, dating from far back in organ history, was the **expression pedal**. This is a lever controlled by the foot which can be set to any angle (and stays put until changed by the foot again). On an organ, the expression pedal is primarily (but not entirely) used to control the volume of the instrument.¹²¹ On a synthesizer, which is typically velocity sensitive, an expression pedal is often used to adjust volume, but may be used to adjust the timbre of the sound in some other way.

Early electronic music experimented with a number of other ways to change pitch or timbre. The most famous early electronic instrument, the **theremin**,¹²² was controlled by proximity of one’s hands to two different antennas. The distance of one hand to the vertical antenna controlled the pitch of the sound, while the distance of the second hand to the horizontal antenna controlled the volume. Both could be adjusted in real-time, causing both **vibrato** (rapid change in pitch) and **tremolo** (rapid change in volume), as well as slides in these parameters (such as **pitch bend** or **portamento**).¹²³



Figure 153 Alexandra Stepanoff performing with a theremin on NBC Radio, 1930. ©68

¹²¹On an organ the expression pedal is called a **swell pedal**, as early versions controlled the **swell box**, a set of blinds between the organ’s pipes (stops) and the audience which could be opened or closed to varying degrees to change the amount of sound reaching the audience.

¹²²Named after its inventor, **Léon Theremin**. The theremin remains the only significant musical instrument that is played without touching it. You’ve heard the theremin: it’s the eerie space-ship sounding instrument on *Good Vibrations* by the **Beach Boys**. And now for a fun fact. Léon Theremin was a Russian who developed the instrument based on his Soviet-funded research into radio-based distance sensors. He traveled the world promoting his instrument and popularizing its use in concert halls, classical and popular music, and so on. He was then kidnapped in his New York City apartment by Soviet agents and taken back to a Siberian prison-laboratory and forced to design spy devices for Stalin for 30 years. It was there that he invented an incredible device called **The Thing**. This was a passive (powerless) microphone listening device embedded in a Great Seal of the United States given to the U.S. Ambassador to Russia and which hung in his office for almost a decade before being discovered. Look it up. It’s an amazing story.

¹²³A theremin-inspired controller found on some synthesizers in the 1990s was Roland’s **D-Beam**, which measured the distance of one’s hand with an infrared beam sensor. It’s often, and I think unfairly, ridiculed.

Another approach, popularized by the **Trautonium** and similar devices (see Section 6.1), was to control pitch by pressing on a wire at a certain position; this also allowed sliding up and down the wire to bend the pitch. Variants of this found their way into synthesizers, including the **pitch ribbon**, a touch-sensitive strip on the Yamaha CS-80 (Figure 52 on page 59). This strip was put to heavy use by **Vangelis** for his soundtracks (page 59). Touch strips are found here and there on modern synthesizers, but more common are sliding wheels such as the ubiquitous **pitch bend wheel** and **modulation wheel** found next to almost all modern synthesizer keyboards. The pitch bend wheel, which shifts the pitch of the keyboard, is **self-centering**, meaning that when the performer lets go of it, it springs back to its mid-way position. The modulation wheel, which can often be programmed to control a variety of parameters, stays put much like an expression pedal. A similar effect (in two simultaneous directions) can be achieved with a joystick.¹²⁴

MIDI Control Surfaces We cannot ignore the most obvious way to expressively control parameters on a synthesizer: its own knobs and sliders. Synthesizer designers often give a lot of thought to how these knobs might be best put to use both in programming the synthesizer and in real-time control.¹²⁵

If your synth has no knobs, fear not. MIDI provides a way for a controller to remotely change any of the parameters the synthesizer has exposed. As a result, many controller keyboards (for example, Figure 154) are outfitted not only with keys but with an array of buttons, sliders, and dials which can be programmed by the musician to send arbitrary MIDI parameter-change commands to synthesizers. In fact, there exist standalone controllers, called **control surfaces**, which have *no keyboard at all*, but rather consist *entirely* of these buttons, sliders, and dials. Two well-known examples are the **Novation Remote Zero** and **Behringer BCR2600**. These devices can be used to control synthesizers, digital audio workstations, audio mixers, and a host of other audio recording and reinforcement devices.

Other Instruments Designers have built controllers inspired by common musical instruments and meant to enable musicians who play those instruments to have access to synthesizers. An easy target has always been drums. Since at least the late 1960s musicians have been creating makeshift devices to allow them to control early drum synthesizers of the time. It was not until around 1976 that commercial drums became available, when Pollard Industries released the **Syndrum**, notably followed by the **Simmons SDS-5** (Figure 156). Many electronic and MIDI-based drum kits have



Figure 154 Akai MPK mini MIDI controller, with a self-centering joystick (red), drum pads, assignable knobs, and velocity sensitive mini keys.^{©69}



Figure 155 Novation Zero SL Mk II, with assignable dials, buttons, sliders, and drum pads.



Figure 156 The Simmons SDS-5.^{©70}

¹²⁴Unlike in video games, many synthesizer joysticks are often not self-centering.

¹²⁵For example: you'll notice that often the knob for filter cutoff is larger than other knobs. This is because the larger a knob the more precisely you can dial in a value, and filter cutoff often must be very carefully set.

since been designed, and **drum pads** have been reduced in size where they can be played with fingers and used to augment controller keyboards (such as in Figures 154 and 155).

Drums are not the only option. Software can be added to pickups for guitars and other stringed instruments, converting their audio into MIDI event signals (via **guitar processors**). **Wind controllers** have been devised in the shape of woodwind instruments (see Figure 157). Wind controllers can control more parameters than you might imagine, including finger pressure, bite pressure, breath speed, and other embouchure manipulations. Related is the **breath controller**, where the musician simply blows at a certain rate to maintain a certain parameter value.



Figure 157 Wind Controller.^{©71}

Grid Controllers The 2000s saw a significant degree of influence on the synthesizer industry by the DJ market. One particularly popular digital audio workstation, **Ableton Live**, capitalized on this with a GUI consisting of a grid of buttons tied to samples triggered when the corresponding button was pressed. To support this and similar DAW UIs came a new kind of MIDI controller, the **grid controller**, which provided hardware buttons corresponding to the software ones in Ableton. The first major controller of this type was Novation's **Launchpad** (Figure 158).

A grid controller is simply an array of buttons or velocity sensitive drum pads with a few additional auxiliary buttons or dials. These are not complex devices: their grids can be configured for many performance tasks, but are most commonly they are used as buttons which trigger sound samples, and which light up while the sample is playing.



Figure 158 Novation Launchpad.^{©72}

Multidimensional Expression There are lots of alternative options. These include **Jankó keyboards** which are laid out in a grid or hexagonal grid, controllers worn as rings or gloves, and a host of quite unusual stuff. But much of the recent effort in controller design has been in creating controllers which increase the degree of **expressivity** available to the musician. This is a fancy way of saying the number of parameters that can be simultaneously and straightforwardly controlled. There are two difficulties with this. First, the musician doesn't have twenty fingers: there is an upper bound on his physical capability to play more notes or play more expressively. Second, there is the cognitive limit: a musician can only keep so many balls in the air at one time in his head. Working around these two limits is a nontrivial psychological **human factors** task.



Figure 159 Ensoniq SQ80 synthesizer, one of the few synthesizer keyboards with polyphonic aftertouch.^{©73}

Synthesizers keyboards have attempted to add additional expressivity to individual notes by allowing the musician to modify key values after they have been pressed. This is usually done by pressing *harder* on the keys as you hold them down.¹²⁶ This approach is commonly known as **pressure** or **aftertouch**.

¹²⁶There exists a very rare alternative where the musician could shift a key *sideways*.

Many keyboards implement **channel aftertouch** (a MIDI term: see Section 12.2), whereby the keyboard can detect that *some key* is being pressed harder and by what amount. This only adds one global parameter, like the mod wheel or pitch bend, rather than per-note parameters. It is much more expensive for a keyboard to implement **polyphonic aftertouch** (again, a MIDI term), where the keyboard can report independent aftertouch values for *every key being pressed*. Polyphonic aftertouch is rare: only a few synthesizers and controller keyboards have historically implemented it. Figure 159 shows an Ensoniq SQ80, one synthesizer which had polyphonic aftertouch. Finally, when the musician releases a key, some keyboards report the **release velocity**.

Recent controllers have made possible even more simultaneous parameters. The first controller in this category was the **Haken Continuum Fingerboard**; others include the **ROLI Seaboard** and the **Roger Linn Design LinnStrument** (Figures 160 and 161).

These devices all take the form of flexible sheets which the musician plays by hitting with his fingers. When the musician touches the sheet with a finger, it registers the location touched and the velocity with which the finger hit the sheet: these translate into note pitch and velocity (volume) respectively. The musician can then move his finger about the sheet, which causes the device to report the new pressure with which the finger is touching it, as well as its new X and Y locations. These translate into aftertouch, pitch bend (for the X dimension) and a third parameter of the musician's choice for the Y dimension. Finally, as the musician releases his finger, the sheet reports the release velocity. Critically, this information is reported for *multiple fingers simultaneously and independently*.¹²⁷ Related is the **Eigenlabs Eigenharp**, which combines a multidimensional touch-sensitive keyboard, a controller strip, and a breath controller.

12.2 MIDI

In 1978 **Dave Smith** (of Sequential Circuits, Inc.) released the popular and influential **Prophet 5** synthesizer. The Prophet 5 was the first synthesizer to be able to store multiple patches in memory, and to do this, it relied on a CPU and RAM. Smith realized that as synthesizers began to be outfitted with processors and memory, it would be useful for them to be able to talk to one another. With this ability, a performer could use one synthesizer keyboard to play another synthesizer, or a computer could play multiple synthesizers at once to create a song. So in 1983 he worked with **Ikutaro Kakehashi** (the founder of **Roland**) to propose what would later become the **Musical Instrument Digital Interface**, or **MIDI**. MIDI has since established itself as one of the stablest, and oldest, computer protocols in history.

MIDI is just a one-way serial port connection between two synthesizers, allowing one synthesizer to send information to the other. MIDI was designed for very slow devices and to pack a lot of information into a small space.



Figure 160 Haken Continuum Fingerboard (Top)^{©74} and Roger Linn Design LinnStrument (Bottom).^{©75}



Figure 161 ROLI Seaboard.^{©76}



Figure 162 Eigenharp.^{©77}

¹²⁷Yes, this means, among other things, that these devices effectively have polyphonic aftertouch.

MIDI runs at exactly 31,250 bits per second. This is a strange and nonstandard serial baud rate: why was it chosen? For the simple reason that $31250 \times 32 = 1,000,000$. Thus a CPU running at N MHz could be set up to read or write a MIDI byte every $N/32$ clock cycles, making life easier for early synthesizer manufacturers.

MIDI bytes are sent (in serial port parlance) with 1 start bit, 8 data bits, and 1 stop bit. This means that a single byte requires 10 bits, and thus MIDI is effectively transmitted at 3125 bytes per second. This isn't very fast: many MIDI messages require three bytes, and so a typical MIDI message, such as "play this note", requires about 1 millisecond to transmit. Keep in mind that humans can detect audio delays of about 3 milliseconds. Pile up a few MIDI messages to indicate a large chord, and the delay could be detectable by ordinary ears. Thus a number of tricks are employed, both in MIDI and by manufacturers after the fact, to maximize throughput.

12.2.1 Routing

MIDI is designed to enable one device to control up to 16 other devices. In its original incarnation, MIDI ran over a simple 5-pin DIN serial cable, and a MIDI device had a **MIDI in** port, a **MIDI out** port, and a **MIDI thru** port, as shown in Figure 163. MIDI In received data from other devices, MIDI Out sent data to other devices, and MIDI Thru just forwarded the data received at MIDI In.

To send MIDI data from Synthesizer A to Synthesizer B, you'd just connect a MIDI cable from A's Out port to B's In port. If you wanted send MIDI data from Synthesizer A to Synthesizers B and C, you could connect a cable from A's out to B's In, then connect *another* cable from B's Thru to C's in (and repeat to connect to D, etc.)

An alternative would be to connect A to a device called a **MIDI router** (or **MIDI patchbay**), which contained multiple Thru ports, and connect each of those ports the MIDI In ports of B, C, and D respectively, as shown in Figure 164. And as shown in that figure, there's no reason you couldn't mix the two techniques.

It's common to need device A to send to device B, and B to send to device A. Just connect a MIDI cable from A's Out to B's In, and likewise another cable from B's Out to A's In. Or it might be the case that you wish for A to talk to B and C, but (say) for B to talk exclusively to D. To do this, you simply connect A's Out to B's In, and B's Thru to C's In. Then you connect B's Out to D's In. Device A wouldn't send data to D: but B could.

Note that while MIDI is designed to allow one sender connect to multiple receivers, it is *not* designed to allow multiple senders to send to the same receiver. To enable such magic would require a special gizmo called a **MIDI merge** device, and some wizardry would be involved.

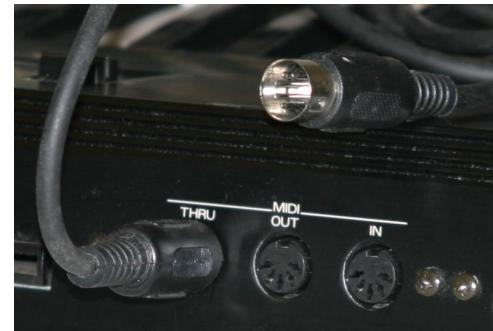


Figure 163 5-Pin DIN MIDI Cable and its In, Out, and Thru ports.^{©78}

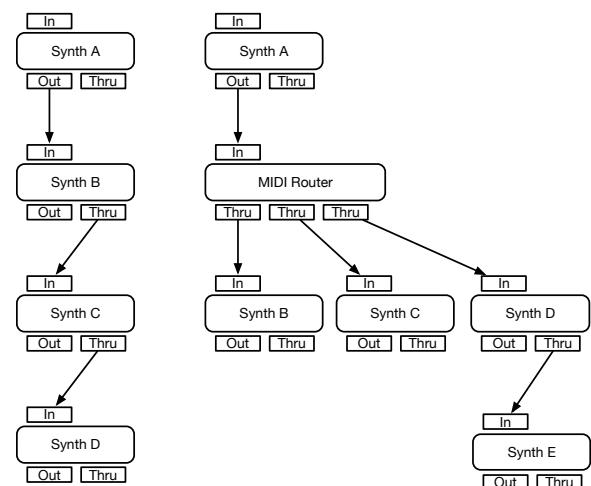


Figure 164 MIDI Routing Options

MIDI over USB MIDI has since been run over Ethernet, Firewire, wireless, Bluetooth, fiber-optic, you name it. But critically MIDI is now very often run over USB, as an alternative to the old 5-Pin DIN cables, often to connect a synthesizer or controller to a computer. Given that USB also allows one device to connect to many, and is much faster than old MIDI serial specs, you'd think this was a good fit. But it's not.

The first problem is that USB connects a *host* (your computer) with a *client* (your printer, say), and indeed they have different shaped ports to enforce which is which. USB devices generally can't be both hosts and clients without separate USB busses. This means that, in almost all cases, traffic has to be routed through the host—your laptop—even you just want a controller to control a synthesizer. Most USB MIDI devices, which lack their own host ports, have lost the peer-to-peer capability which made MIDI so useful. USB is great for connecting mice to your computer. Not so much networking synthesizers with other synthesizers.

Another more serious problem is that USB is not **electrically isolated**. When two devices are attached over USB, they are directly electrically connected, and this often creates problematic electronic noise issues—including the infamous “ground loop”, a 50Hz or 60Hz hum produced when two audio devices are connected which have different grounds. MIDI was originally expressly designed to avoid these issues: its circuitry specification requires an **optoisolator**: essentially a little light bulb and light detector in a small package which, when embedded in the MIDI circuitry, allows two devices to talk to one another without actually being electrically connected at all.

Nonetheless, with the advent of the **Digital Audio Workstation**, more and more music studios are computer-centric, with all the synthesizers and similar devices flowing into a single computer. The popularity of MIDI over USB only promotes this, as USB is highly PC-centric.

12.2.2 Messages

MIDI messages are just strings of bytes. The first byte in the sequence, called the **status byte**, has its high bit set to 1. The remaining **data bytes** in the sequence have their high bits set to 0. The status byte indicates the type of message. Thus MIDI can only transfer 7 useful bits in a byte, as the first bit is used to distinguish the head of a message from the body. For this reason, you'll find that the numbers 128 (2^7) and 16384 ($2^{7 \times 2}$) show up a lot in MIDI, but 256 rarely does. Indeed, 7-bit strings in MIDI are so prevalent that they are often referred to as “bytes”.

MIDI is organized so that the most time-sensitive messages are the shortest:

- *Single byte* messages are largely timing messages. These messages are so time critical that they can in fact be legally sent *in the middle* of other messages.
- *Two- and Three- byte* messages usually signify events such as “play a note”, “release a note”, “change a control parameter to a certain value”, etc.
- There is a single type of variable-length message: a **system exclusive** (or **sysex**) message. This is essentially an escape mechanism to allow devices to send custom data to one another, often in large dumps: perhaps transferring a synthesizer patch from a computer to a synthesizer, for example.

Sysex

0xF0	<i>id...</i>	<i>data...</i>	0xF7
------	--------------	----------------	------

 Sysex messages are manufacturer-specific, but they are required to have a certain pattern. First comes the status byte 0xF0. Next comes a stream of data bytes. The first few data bytes must be the ID of the manufacturer of the synthesizer for which the message is crafted. Manufacturer IDs are unique and registered with the MIDI Association. This allows synthesizers to ignore Sysex messages that they don't recognize. At the end of the stream of data bytes is another status byte, 0xF7, indicating the end of the message.

Channels Some messages (timing messages, sysex, etc.) are broadcast to any and all devices listening. Other messages (like note information) are sent on one of 16 *channels* 0...15. The 3 bits indicating the channel are part of the status byte. A synthesizer can be set up to respond to only messages on a specific channel: that way you can have up to 16 different synthesizers responding to messages from the master. There's no reason a synthesizer can't respond to different channels for different purposes (this is common); and there's no reason you can't set up several synthesizers to respond to the same channel (this is unusual). Finally, many synthesizers are set up by default to respond to messages on *any* channel for simplicity. In MIDI parlance this is called the **omni channel**.

Running Status It takes three bytes (about 1 ms!) just to tell a synthesizer to start playing a note. But recognizing that very often the same kind of message will appear many times in sequence, MIDI has a little compression routine: if message A is of a certain type (say "Note On"), and the very next message B is the same kind of message and on the same channel, then B's status byte may be omitted. If the very next message C is again the same message type and channel, its status byte may be omitted as well, and so on. This allows a stream of (say) Note On messages to start with a 3-byte message, followed by many 2-byte messages.

Channel Voice Messages Most MIDI messages are of this type: they indicate events such as notes being played or released, the pitch bend wheel being changed, etc. All of these messages have associated channels. The channel is specified by the lower four bits of the the status byte (denoted *ch* below): thus 0x86 means a status byte for Note Off (the "8") on channel 6 (the "6").

- **Note On**

0x9ch	<i>note</i>	<i>velocity</i>
-------	-------------	-----------------

 tells a synthesizer that a note should be played. This message comes with two data values, both 0...127: the note in question (middle C is 60, that is, 0x3c), and the *velocity* (how fast the key was struck), which usually translates to the note's volume. Some keyboards may not detect velocity, in which case 64 (0x40) should be used. A velocity of 0 has a special meaning, discussed next.
- **Note Off**

0x8ch	<i>note</i>	<i>release velocity</i>
-------	-------------	-------------------------

 tells a synthesizer that a note should stop being played. This message comes with two data values, both 0...127: the note in question (middle C is 60 or 0x3c), and the *release velocity* (how fast the key was released).

Many keyboards cannot detect release velocity, in which case 64 (0x40) should be used. If we didn't care about release velocity, then instead of sending a Note Off, it is very common to instead send a *Note On* with a velocity of 0, which is specially interpreted as a Note Off of velocity 64. This allows a controller to never have to send a Note Off message, just a string of Note On messages, and so take better advantage of Running Status.

- **Polyphonic Key Pressure or Polyphonic Aftertouch**

0xAch	note	pressure
-------	------	----------

 tells a synthesizer that a key, currently being held down, is now being pressed *harder* (or softer). This message comes with two data values, both 0...127: the note in question (middle C is 60 or 0x3c), and the *pressure level*. Polyphonic key pressure is difficult to implement in a keyboard and so it's not very common, and this is probably good because it tends to flood MIDI with lots of messages.
- **Channel Pressure or Channel Aftertouch**

0xDch	pressure
-------	----------

 tells a synthesizer that *the keyboard as a whole* is now being pressed *harder* (or softer). This message comes with a single data value (0...127): the *pressure level*. Many keyboards implement channel pressure. A synthesizer won't implement both channel and polyphonic key pressure at the same time.
- **Program Change or PC**

0xCch	patch
-------	-------

 asks the synthesizer to change to some new patch (0...127). Many synthesizers have more than 128 patches available, so it's not uncommon for patches to be arranged in *banks* of up to 128 patches, and so a PC message may be preceded by a **bank change** request, discussed later. This message is rarely real-time: many synthesizers take quite a bit of time (milliseconds to seconds) to change to a new patch.
- **Pitch Bend**

0xEch	MSB	LSB
-------	-----	-----

 tells a synthesizer that the pitch bend value has been changed.¹²⁸ Pitch Bend is a high resolution 14-bit value from -8192...+8191. The two values (MSB and LSB) are both 0...127, and the bend value is computed as $\text{MSB} \times 128 + \text{LSB} - 8192$.
- **Control Change or CC**

0xBch	parameter	value
-------	-----------	-------

 tells a synthesizer that some *parameter* (0...127) has been adjusted to some *value* (0...127). You can think of this as informing a synthesizer that a musician wants to tweak some knob on it. The meaning of CC parameters and their respective values varies from synthesizer to synthesizer, and there's some complexity to it, discussed in Section 12.2.3. Also, 0...127 is not particularly fine-grained: also discussed in Section 12.2.3 are options for sending more precise information.

Clock Messages Many music devices, such as drum machines, can play songs or beat patterns all on their own. It's common to want to synchronize several of them so they play their songs or beats at the same time. MIDI has a mechanism to allow a controller to send clock synchronization messages to every listener. MIDI defines a **clock pulse** as 1/24 of a quarter note. This is a useful value, since lots of things (sixteenth notes, triplets, etc.) are multiples of it. A controller can send out clock pulses at whatever rate it likes, like a conductor, and listening devices will do their best to keep up.

To send clock pulses, a device must first send a *Start* message. It then sends out a stream of *Clock Pulse* messages.¹²⁹ It may conclude by sending a *Stop* message. If it wished to start up where it left off, it could then send a *Continue* message and keep going with pulses. Alternatively, if it wished to restart from the beginning, it could send another *Start* message after the *Stop* and continue pulsing.

¹²⁸MSB stands for **M**ost **S**ignificant **B**yte and LSB stands for **L**east **S**ignificant **B**yte, even though neither of them is a byte: they're both 7-bit values.

¹²⁹One annoyance with MIDI Clock is that after a Start has been sent, you can't realistically determine or even estimate the clock rate until two clock pulses have been received, and you have to wing it until then.

- **Clock Pulse or Timing Clock** $0xF8$ Sends a pulse.
- **Start** $0xFA$ Informs all devices to reset themselves and to prepare to begin playing on receiving pulses.
- **Stop** $0xFC$ Informs all devices to pause (or stop) playing.
- **Continue** $0xFB$ Informs all devices to resume playing on receiving pulses.
- **Song Select** $0xF3$ $song$ Informs all devices to prepare to start playing a given *song* (drum-beat pattern, whatnot) 0...127. This is not often used.
- **Song Position Pointer** $0xF2$ MSB LSB Informs all devices to prepare to begin playing the current song at the given *position* $MSB \times 128 + LSB$. The position is defined in “MIDI Beats”: one MIDI Beat is 6 clock pulses, that is, one sixteenth note. Position 0 is the start of the song.

Other Stuff There are several other non-channel messages, none particularly important:

- **MIDI Time Code Quarter Frame** $0xF1$ $data\ byte$ A sequence of these messages collectively send an SMPTE¹³⁰ time code stamp. This is an absolute time value (frames, seconds, minutes, etc.) and is used to synchronize MIDI with video etc. These messages won’t be discussed further here.
- **Tune Request** $0xF6$ Asks all devices to tune themselves. No, seriously. MIDI was created when synthesizers were primitive.
- **Active Sensing** $0xFE$ An optional heartbeat message which assures downstream devices that the controller hasn’t been disconnected. It can be ignored.
- **System Reset** $0xFF$ Asks synthesizers to completely reset themselves as if they had just been powered up. Again, MIDI is old.

12.2.3 CC, RPN, and NRPN

Control Change (CC) messages (of the form $0xBch$ $parameter$ $value$) are meant to allow a controller to manipulate a synthesizer’s parameters, whatever they may be. Synthesizers are free to interpret various control change messages however they deem appropriate, though there are some conventions. Here are a few common ones:

- Parameter 0 often selects the **patch bank**. Thus (for example) a synthesizer might have up to 128 banks, each containing 128 patches (selected with Program Change).
- Parameter 1 often specifies the value of the **modulation wheel**.

¹³⁰SMPTE: Society of Motion Picture and Television Engineers.

- Parameter 2 often specifies the value of a **breath controller** device.
- Parameter 4 often specifies the value of a **foot controller**.
- Parameter 11 often specifies the value of an **expression controller** (this value is usually multiplied against the global overall instrument volume to set its temporary volume).
- Parameter 64 often specifies whether the sustain pedal is down (1) or not (0).
- Parameter 74 often specifies the “**Third Dimension Controller**” specified by **MIDI polyphonic expression** (or **MPE**), discussed later.
- Parameters 6, 32, 96, 97, 98, 99, 100, and 101 are often reserved for **NRPN** and **RPN** (see later).
- Parameters 120–123 are reserved for standardized functions called **MIDI channel mode messages**:
 - Parameter 120 (with value 0) is the **all sound off** message. This tells a synthesizer to immediately cut all sound.
 - Parameter 121 (with value 0) is the **reset all controllers** message. This tells a synthesizer to reset all its parameters to their default settings.
 - Parameter 122 is the **local switch** message. This tells a synthesizer to turn on (127) or off (0) “local mode”. When in local mode, the synthesizer’s own keyboard can send notes to the synthesizer. When not in local mode, this connection is disconnected, but the keyboard can still send messages out MIDI, and the synthesizer can still respond to MIDI.
 - Parameter 123 (with value 0) is the **all notes off** message. This tells a synthesizer to effectively send a *Note Off* message to all its currently played notes. This does *not* immediately cut all sound, as notes may have a long release time in response.
- Parameters 124–127 are reserved for additional, now-obsolete standardized **MIDI channel mode messages** which control so-called omni mode and mono vs. poly modes. An instrument in omni mode responds to any channel. An instrument in mono mode is monophonic, and an instrument in poly mode is polyphonic. While these modes and messages are obsolete, this region is nonetheless still (unfortunately) reserved.¹³¹
 - Parameter 124 (with value 0) turns omni mode off.
 - Parameter 125 (with value 0) turns omni mode on.
 - Parameter 126 (with a value indicating the number of channels) turns mono mode on.
 - Parameter 127 (with value 0) turns poly mode on.

CC messages have two serious problems. The first problem is that there are only 120 of them (disregarding the MIDI channel mode region). But a synthesizer often has hundreds, sometimes thousands, of parameters! The second problem is that the value can only be 0...127. This is a very coarse resolution: if you turned a controller knob which sent CC messages to a synthesizer to (say) change its filter cutoff, the stepping would be very evident—it wouldn’t be smooth.

¹³¹Why 124 and 125 aren’t merged, and similarly 126 and 127, I have no idea.

Early on, the MIDI spec tried to deal with the second problem by reserving CC parameters 32–63 to be the Least Significant Byte (LSB) corresponding to the parameters 0–31 (the Most Significant Byte or MSB). The idea was that you could send a CC for parameter 4 as the MSB, then send a CC for parameter 36 (32+4) as the LSB, and the synthesizer would interpret this as a higher resolution 14-bit value 0...16383, that is, $\text{MSB} \times 128 + \text{LSB}$. This would work okay, except that there were only 32 high-resolution CC parameters, and this scheme reduced the total number of CC parameters—already scarce—by 32. Thus many early synthesizers simply disregarded CC for their advanced parameters and relied on custom messages via the Sysex facility (unfortunately).

But in fact MIDI has a different and better scheme to handle both of these two problems: **Reserved Parameter Numbers (RPN)** and **Non-Reserved Parameter Numbers (NRPN)**. The RPN and NRPN schemes each permit 16384 different parameters, and those parameters can all have values 0...16383. RPN parameters are reserved for the MIDI Association to define officially, and NRPN parameters are available for synthesizers to do with as they wish.

RPN and NRPN work as follows. For NRPN, a controller begins by sending CC Parameter 98 and CC Parameter 99, which define the MSB and LSB respectively of the NRPN Parameter number being sent. Thus if a controller wished to send an NRPN 259 message, it'd send 2 for Parameter 98 and 3 for Parameter 99 ($2 \times 128 + 3 = 259$). For RPN, these CC parameters would be 100 and 101 respectively. Next, the controller would send the MSB and LSB of the *value* of the NRPN (or RPN) message as CC Parameters 6 and 32 respectively. The MSB and LSB of the value can come in any order and either may be omitted, unfortunately complicating matters. The controller could alternatively send an “increment” or “decrement” message (96 and 97 respectively). For example, a CC 96 with a value of 5 would mean that the parameter should be incremented by 5.

Inspired by Running Status, a stream of these value-messages (6, 32, 96, or 97) could be sent without having to send the parameter CC messages again, as long as the NRPN or RPN parameter remained the same. To shut off this running-status-ish stream (perhaps to prevent any further inadvertent NRPN value messages from corrupting things), one could send the **RPN Null** message. This is RPN parameter 16383—that is, MSB 127 and LSB 127—with any value.

The problem with RPN and NRPN is that they are slow: to update the value of a new parameter requires 4 CC messages, or 12 bytes. Another problem with RPN and NRPN is that only some synthesizers implement them, and even more problematically, some lazy Digital Audio Workstation manufacturers do not bother to include them as options.

12.2.4 Challenges

MIDI has been remarkably stable since it was invented in 1983: indeed, the spec is still technically fixed at 1.0!¹³² But MIDI was designed in the age of synthesizer keyboards, and it was not meant to be extended to elaborate multidimensional controllers which manipulate many parameters at once, nor to complex routing scenarios involving software. This produces a number of problems:

- **Many MIDI parameters are per-instrument, not per-voice.** MIDI can support many parameters, but it has only has a few defined parameters which are *per note*: pitch, attack velocity, release velocity, and polyphonic aftertouch. Other parameters are *global* to the whole instrument, whether appropriate or not (often not).

¹³²This is really kind of a lie. MIDI 1.0 in 1983 is fairly different from the MIDI 1.0 of the 2000s. But the MIDI Association never updated the version number. That's finally changing soon though, with MIDI 2.0.

- **MIDI is slow.** MIDI was fixed to 31,250 bits per second in order to support early synthesizers with 1MHz CPUs. This is not fast enough to guarantee smooth transitions beyond the ability for humans to detect.
- **MIDI is low resolution.** Only two standard parameters (pitch bend and song position pointer) are 14-bit: the rest are 7-bit, which is very coarse resolution. There exist two kinds of 14-bit extensions to some parameters (14-bit CC and RPN/NRPN), but they come at the cost of making MIDI up to $3 \times$ slower. One solution to this is not to use MIDI at all, but rather to fall back to traditional **CV/Gate** control used by modular synthesizers. CV/Gate is real-valued and so can be arbitrarily high resolution (in theory) and fast (in theory). A number of current keyboards provide both MIDI and CV/Gate for modular synthesizers such as Eurorack.¹³³
- **MIDI was designed as a one-direction protocol.** There's no standard way to query devices for capabilities and get results back, to negotiate to use a more advanced version of the protocol, etc.

But this situation will be changing soon with many new MIDI protocol features. The first of these are dealt with by a new extension to MIDI called **MIDI polyphonic expression** or **MPE**. The remaining three problems are will be tackled by an upcoming version of MIDI called **MIDI 2.0**. We discuss these below.

12.2.5 MPE

MIDI was designed with the idea that people would by and large use keyboards as controllers. Keyboards are essentially a collection of levers, and the performer is restricted in the number of parameters he can control for each note. In MIDI, a performer can specify at most the note of the key, the velocity with which the key is struck, the velocity with which it is released, and (using polyphonic key pressure) the pressure with which a key is currently being pressed. All other parameters (CC, PC, channel pressure, NRPN, etc.) are global to the whole instrument. Particularly problematic: pitch bend is global.

But many of instruments are more expressive than this: for example, a guitarist can specify the volume and pitch bend of each string independently. A woodwind musician controls all sorts of timbre parameters with his mouth (the **embouchure** with which he plays the instrument). And so on. Many current advanced MIDI controllers seek to enable changing a variety of independent, per-note parameters in real time. But MIDI doesn't permit this.



Figure 165 Futuresonus Parva, the first hardware synthesizer¹³⁴ to support MPE. ©79

¹³³CV/Gate works as follows. A **gate** signal is an analog signal which goes from 0 volts to some positive voltage (or, for some systems, from positive to 0) to indicate that a key has been struck. The opposite occurs to indicate that a key has been released. Accompanying this is a **control voltage** or **CV** signal, which indicates the pitch of the note. Recall from Footnote 33 (page 53) that CV is either encoded in **volt per octave**, where each volt means one more octave, or **hertz per volt**, where voltage doubles with each octave. These are analog signals, and so they are as fast and as precise as necessary. Additional signals could be added to indicate velocity and other parameters.

¹³⁴The Parva was also the first hardware synthesizer to support **USB host for MIDI**. This means that a USB MIDI keyboard controller can be plugged directly into the Parva in order to play it. As mentioned in Section 12.2.1, normally you'd have to control a synthesizer from a USB controller by attaching both to a laptop. The Parva is a rare exception.

To deal with this situation, the high-parameter controller manufacturers (ROLI, Haken, Roger Linn Design, etc.) support a new MIDI standard which provides “five-dimensional” control (attack velocity and note pitch, channel aftertouch,¹³⁵ pitch bend, release velocity, and “Y” dimensional movement, all per-note). This is known as **MIDI polyphonic expression**¹³⁶ or **MPE**.

MPE works by hijacking MIDI’s 16 channels: rather than assign each channel to a different synthesizer (a less common need nowadays), MPE uses them for *different notes currently being held down* on a single synthesizer. MPE divides the 16 channels into up to two **zones**. If there is only one zone, then one channel is designated its **master channel**, for global parameter messages from the controller, and the other 15 channels are assigned to 15 different notes (voices) played through that controller. That way each voice can have its own unique CC messages and its own pitch bend. A special CC parameter, number 74, is by convention reserved as a dedicated standard “third dimension” parameter (beyond pressure and pitch bend). If there are two zones—notionally to allow two instruments to be played by the controller—then each has its own master channel, and the remaining 14 channels may be divvied up among the two zones (perhaps one instrument could be allocated 4 voices and the other 10, say).

The two zones are known as the **upper zone** and **lower zone**. The lower zone uses MIDI channel 1 as its master channel, and has some number of additional channels 2, 3, ... assigned to individual notes. The upper zone has MIDI channel 16 as its master channel and additional channels 15, 14, ... assigned to individual notes. If there is only one zone—by far the most common scenario—it can take up all 15 available channels beyond the master channel, and the controller may choose to use the upper or the lower zone as this sole zone.

MPE zones are either preconfigured in the instrument, or may be specified by the controller using RPN command #6 sent on either channel 1 (to configure the lower zone) or 16 (to configure the upper zone), with a parameter value of 0 (turning off that zone) or 1...15 (to assign up to 15 channels to the zone). All told the RPN message consists of three CC messages:

0xBn	0x64	0x06	0xBn	0x65	0x00	0xBn	0x06	0x0m
------	------	------	------	------	------	------	------	------

with *n* being the zone (0 or F for lower or upper zone), and *m* being the number of channels 0...F.

Thereafter, when a note is played on the controller, it assigns a channel to the note and sends Note On, Note Off, and all other note-related information on that channel only. This potentially includes pitch bend, aftertouch, and CC, NRPN, or RPN commands special to just that note. Additionally, the controller can make changes to *all* the notes under its control by issuing commands on the master channel. There are a lot of subtleties involved in allocating (and reallocating) notes to channels for which suggestions, but not requirements, are made in the MPE specification.¹³⁷

Note that MPE doesn’t extend MIDI in any way: it’s just a convention as to how MIDI channels are allocated and used for a special purpose. There’s no reason you couldn’t (for some reason) use channels 1...14 for a lower MPE zone, and then use channels 14 and 15 to control standard MIDI instruments in the conventional way, for example.

¹³⁵These devices almost always support polyphonic aftertouch too, but if we’re doing MPE, there’s no reason for it: each note is on its own channel and so the aftertouch is already uniquely assigned to each note. Besides, polyphonic aftertouch requires an additional byte.

¹³⁶The original name, which I much prefer, was **multidimensional polyphonic expression**, but the MIDI Association changed the name prior to its inclusion in the MIDI spec. I don’t know why.

¹³⁷MIDI is an open protocol. The MPE specification, as well as other MIDI specifications and documents, are available for free at <https://www.midi.org/>

12.2.6 MIDI 2.0

As of this writing, MIDI 2.0 is not quite released: so we don't know everything about it. But MIDI 2.0 is designed to deal with a number of difficulties in MIDI, not the least of which are its speed,¹³⁸ low resolution, and unidirectionality.

MIDI Capability Inquiry (MIDI-CI) MIDI 2.0 is bidirectional. One consequence of this is that MIDI 2.0 devices can query one another, trade data, and negotiate the protocol to be used.

- **Profile Configuration** A device can tell another device what kinds of capabilities it has. For example, a drum machine, in response to a query, may respond indicating that it has a certain **profile** typical of drum machines. This informs the listening device that it is capable of responding to a certain set of directives covered by that profile.
- **Property Exchange** Devices can query data from one another, or set data, in a standardized format: this might mean patches, sample or wavetable data, version numbers, vendor and device names, and so on. Perhaps this might spell the end of custom and proprietary sysex formats.
- **Protocol Negotiation** Devices can agree on using a newer protocol than MIDI 1.0, such as MIDI 2.0. The MIDI 2.0 protocol has a number of important improvements over 1.0, including higher resolution velocity, pressure, pitch bend, RPN, NRPN, and CC messages; new kinds of articulated event data (more elaborate Note On / Note Off messages, for example); additional high-resolution controllers and special messages on a per-note basis; and up to 256 channels.

MIDI 2.0 tries hard to be backward compatible with 1.0 when possible. If either device fails to respond to a profile configuration request, property exchange, or protocol negotiation, then the other device falls back to MIDI 1.0, at least for that element.

¹³⁸In fact, I do not know how MIDI 2.0 tackles speed yet, but I assume it does.

Sources

In building these lecture notes I relied on a large number of texts, nearly all of them online. I list the major ones below. I would like to point out four critical sources, however, which proved invaluable:

- Steven Smith's free online text, *The Scientist & Engineer's Guide to Digital Signal Processing*,¹³⁹ is extraordinary both in its clarity and coverage. I cannot recommend it highly enough.
- Julius Smith (CCRMA, Stanford) has published a large number of online books, courses, and other materials in digital signal processing for music and audio. He's considered among the very foremost researchers in the field, and several of the algorithms in this text are derivatives of those in his publications. <https://ccrma.stanford.edu/~jos/>
- Curtis Roads's book, *The Computer Music Tutorial*.¹⁴⁰ Roads is a famous figure in the field: in addition to being a prolific author and composer, he is also a founder of the International Computer Music Association and a long-time editor for *Computer Music Journal*.
- And of course... Wikipedia.

Introduction (No sources of consequence used)

Representation of Sound

<http://www.hibberts.co.uk/index.htm>

https://en.wikipedia.org/wiki/Vibrations_of_a_circular_membrane

The Fourier Transform

<http://www.dsprelated.com>

https://en.wikipedia.org/wiki/Fourier_transform

https://en.wikipedia.org/wiki/Discrete_Fourier_transform

https://en.wikipedia.org/wiki/Window_function

<https://www.dsprelated.com/showarticle/800.php>

<https://adamsiembida.com/how-to-compute-the-ifft-using-only-the-forward-fft/>

<https://dsp.stackexchange.com/questions/4825/why-is-the-fft-mirrored>

<https://www.gaussianwaves.com/2015/11/interpreting-fft-results-complex-dft-frequency-bins-and-fftshift/>

<https://www.dsprelated.com/showarticle/901.php>

<http://pages.di.unipi.it/gemignani/woerner.pdf>

https://en.wikipedia.org/wiki/Cooley-Tukey_FFT_algorithm

http://fourier.eng.hmc.edu/e101/lectures/Image_Processing/

<http://fourier.eng.hmc.edu/e161/lectures/fourier/>

<http://www.robots.ox.ac.uk/~sjrob/Teaching/SP/l7.pdf>

<https://engineering.purdue.edu/~bouman/ece637/notes/pdf/CTFT.pdf>

<http://www.inf.ed.ac.uk/teaching/courses/ads/Lects/lectures4.5-4.pdf>

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.331.4791&rep=rep1&type=pdf>

(Michael T. Heideman, Don H. Johnson, and C. Sidney Burrus, 1984, Gauss and the history of the fast fourier transform, *IEEE ASSP Magazine*, 1(4))

¹³⁹ Steven Smith, 1997, *The Scientist & Engineer's Guide to Digital Signal Processing*, California Technical Publishing, available online at <https://www.dsprelated.com/>

¹⁴⁰ Curtis Roads, 1996, *The Computer Music Tutorial*, The MIT Press

Additive Synthesis

https://en.wikipedia.org/wiki/Additive_synthesis
<http://www.doc.gold.ac.uk/~mas01rf/is52020b2013-14/2013-14/slides13.pdf>

Modulation (No sources of consequence used)

Subtractive Synthesis

https://en.wikipedia.org/wiki/Analog_synthesizer
<https://en.wikipedia.org/wiki/Trautonium>
<http://120years.net>

Oscillators, Combiners, and Amplifiers

https://en.wikipedia.org/wiki/Colors_of_noise
https://en.wikipedia.org/wiki/Chebyshev_polynomials
<https://en.wikipedia.org/wiki/Waveshaper>
https://www.tankonyvtar.hu/en/tartalom/tamop412A/2011-0010_szigetvari_timbrel_solfege/ch12.html
https://en.wikibooks.org/wiki/Sound_Synthesis_Theory/Modulation_Synthesis
http://sites.music.columbia.edu/cmc/MusicAndComputers/chapter4/04_06.php
<https://github.com/martinfinke/PolyBLEP>
<http://www.martin-finke.de/blog/articles/audio-plugins-018-polyblep-oscillator/>
<http://pilot.cnxproject.org/content/collection/col10064/latest/module/m10556/latest>
<http://mac.kaist.ac.kr/pubs/ValimakiNamSmithAbel-taslp2010.pdf>
<http://users.spa.aalto.fi/vpv/DAFX13-keynote-slides.pdf>
<http://www.kasploosh.com/projects/CZ/11800-spelunking/>

Filters

<https://www.dspsguide.com/>
<https://ccrma.stanford.edu/~jos/filters/>
<http://keep.uniza.sk/kvesnew/dokumenty/DREP/Filters/SecondOrderFilters.pdf>
("Second Order Filters", J. McNames, Portland State University, with permission)
https://www.oreilly.com/library/view/signals-and-systems/9789332515147/xhtml/ch12_12-2.xhtml
https://en.wikipedia.org/wiki/Butterworth_filter
<https://en.wikipedia.org/wiki/Resonance>
https://en.wikipedia.org/wiki/Comb_filter
http://www.eecs.umich.edu/courses/eecs206/archive/spring02/lab.dir/Lab9/lab9_v3_0_release.pdf
<https://www.staff.ncl.ac.uk/oliver.hinton/eee305/Chapter5.pdf>
<http://web.mit.edu/2.14/www/Handouts/PoleZero.pdf>
https://www.discodsp.net/VAFilterDesign_2.1.0.pdf
<http://www.micromodeler.com/dsp/>

Frequency Modulation Synthesis

https://en.wikipedia.org/wiki/Frequency_modulation_synthesis
<http://www.indiana.edu/~emusic/fm/fm.htm>
https://www.st-andrews.ac.uk/~www_pa/Scots_Guide/RadCom/part12/page1.html
https://ccrma.stanford.edu/sites/default/files/user/jc/fm_synthesispaper-2.pdf
<https://ccrma.stanford.edu/software/snd/snd/fm.html>
<https://www.sfu.ca/~truax/fmtut.html>
<https://www.youtube.com/watch?v=w4g92vX1YF4>

Sampling

<https://www.dspguide.com/ch16.htm>
<https://ccrma.stanford.edu/~jos/resample/resample.pdf>
<http://www.jean-lucsinclair.com/s/Granular-Synthesis.pdf>
<http://www.nicholson.com/rhn/dsp.html>
<http://paulbourke.net/miscellaneous/interpolation/>
<http://msp.ucsd.edu/techniques/v0.11/book.pdf>
<http://yehar.com/blog/wp-content/uploads/2009/08/deip.pdf>
<http://paulbourke.net/miscellaneous/interpolation/>

Effects and Physical Modeling

<https://ccrma.stanford.edu/~jos/pasp/>
<https://ccrma.stanford.edu/~jos/Reverb/>
https://en.wikipedia.org/wiki/All-pass_filter
[https://en.wikipedia.org/wiki/Phaser_\(effect\)](https://en.wikipedia.org/wiki/Phaser_(effect))
<https://en.wikipedia.org/wiki/Reverberation>
<https://en.wikipedia.org/wiki/Flanging>
https://en.wikipedia.org/wiki/Chorus_effect
<https://www.cs.sfu.ca/~tamaras/effects/effects.html>
https://dsp-nbsphinx.readthedocs.io/en/nbsphinx-experiment/recursive_filters/direct_forms.html
<http://www.dreams-itn.eu/uploads/files/Valimaki-AES60-keynote.pdf>
<https://www.music.mcgill.ca/~gary/618/>
<https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/convolution.html>
<http://www.csounds.com/manual/html/MiscFormants.html>

Controllers

<https://www.midi.org/specifications-old/item/the-midi-1-0-specification>
<https://www.midi.org/articles-old/midi-polyphonic-expression-mpe>

Figure Copyright Acknowledgments

- ©¹ kpr2, CC0, <https://commons.wikimedia.org/w/index.php?curid=57571949>
- ©² Andrew Russeth, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=26604703>
- ©³ JR (Flickr), CC-BY-2.0, <https://www.flickr.com/photos/103707855@N05/29583857506/>
- ©⁴ Steve Sims, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=12379187>
- ©⁵ Reprinted with permission by Ed James (Clyne Media).
- ©⁶ Bernd Sieker (Flickr), CC-BY-2.0, https://www.flickr.com/photos/pink_dispatcher/13804312464
- ©⁷ Aquegg, Public Domain, <https://commons.wikimedia.org/wiki/File:Spectrogram-19thC.png>
- ©⁸ Fourier1789, Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=41344802>
- ©⁹ This image is mine, but is inspired by a prior original image by MusicMaker5376, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=17413304>
- ©¹⁰ Gunther, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=821342>
- ©¹¹ Bob K, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=648112>
- ©¹² Olli Niemitalo, CC0, <https://commons.wikimedia.org/w/index.php?curid=24627501>
- ©¹³ Dvortygirl, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=2524720>
- ©¹⁴ Ken Heaton, Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=48242081>
- ©¹⁵ Paulo Ordoveza, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=29694963>
- ©¹⁶ Jane023, CC BY-SA 3.0 nl, <https://commons.wikimedia.org/w/index.php?curid=19732582>
- ©¹⁷ Public Domain, <https://commons.wikimedia.org/w/index.php?curid=391692>
- ©¹⁸ Scientific American, 1907, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=49752962>
- ©¹⁹ JacoTen, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=24356990>
- ©²⁰ Julien Lozelli, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=7767415>
- ©²¹ Brandon Daniel, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=33527250>
- ©²² Allangothic (Wikipedia), CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=2875045>
- ©²³ Museumsinsulaner, Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=12503701>
- ©²⁴ User:MatthiasKabel, Own work, CC BY 2.5, <https://commons.wikimedia.org/w/index.php?curid=1265010>
- ©²⁵ Finnianhughes101, CC0, <https://commons.wikimedia.org/w/index.php?curid=39547075>
- ©²⁶ Surka, Own work, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=8345595>
- ©²⁸ GeschnittenBrot, Flickr: DSC_0117, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=14746344>

- ©29 Kimi95, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=7708499>
- ©30 I, Zinnmann, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=2441310>
- ©31 Museumsinsulaner, Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=12501904>
- ©32 FallingOutside, Own work, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=21742056>
- ©33 Andrew Russeth, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=26604703>
- ©34 Brandon Daniel, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=19964001>
- ©35 Hollow Sun, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=8863073>
- ©36 Robert Brook, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=7509844>
- ©37 Pete Brown, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=38092002>
- ©38 Mojosynths, Own work, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=3815497>
- ©39 Reproduced with permission from Richard Lawson at RL Music. Thanks to Sequential LLC for assistance.
- ©40 CPRdave, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=7633923>
- ©41 Allert Aalders, CC BY-NC-SA 2.0, <https://www.flickr.com/photos/allert/6809874296/>
- ©42 Ed Uthman, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=74691297>
- ©43 Jdmt, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=29340041>
- ©44 Candyman777, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=19810850>
- ©45 F J Degenaar, Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=1671111>
- ©46 Reproduced with permission from Sequential LLC.
- ©47 Paul Anthony, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=67924982>
- ©48 Reprinted with permission from Make Noise, Inc., via CC BY-NC-ND. Photo credit to Eric "Rodent" Cheslak. 0-Coast developed by Tony Rolando.
- ©49 Joshua Schnable, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=26553228>
- ©50 John Athayde, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=7530494>
- ©51 F J Degenaar, Own work, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=1671111>
- ©52 Warrakkk, <https://commons.wikimedia.org/w/index.php?curid=19273415>.
- ©53 BillyBob CornCob. CC0, <https://commons.wikimedia.org/w/index.php?curid=76760547>
- ©54 Cameron Parkins, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=7810459>
- ©55 Gablin, CC BY 2.5, <https://commons.wikimedia.org/w/index.php?curid=1377602>
- ©56 Spinningspark, CC BY-SA 3.0, <https://en.wikipedia.org/w/index.php?curid=23488139>
- ©57 Geek3, CC BY 4.0, <https://commons.wikimedia.org/w/index.php?curid=53958748>

^{©58} Steve Sims, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=12379187>

^{©59} Stonda (assumed) (Wikimedia contributor), CC BY 2.5, <https://commons.wikimedia.org/w/index.php?curid=649793>.

^{©60} Buzz Andersen, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=2801145>

^{©61} Matt Vanacoro, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=39425190>

^{©62} Public Domain, from <https://waveeditonline.com/>

^{©63} Modified version of a figure by James Maier, reproduced with permission, <http://www.carbon111.com> This particular image cropped from <http://www.carbon111.com/table002.png>

^{©64} Reprinted with permission from Tasty Chips Electronics.

^{©65} Olli Niemitalo, CC0, <https://commons.wikimedia.org/w/index.php?curid=24627501>

^{©66} Olli Niemitalo, CC0, <https://commons.wikimedia.org/w/index.php?curid=77818389>

^{©67} Reprinted with permission from Livi Lets.

^{©68} What's On the Air Company, Public Domain, <https://commons.wikimedia.org/w/index.php?curid=56188934>

^{©69} Matt Vanacoro, CC BY 2.0, <https://commons.wikimedia.org/w/index.php?curid=39425186>

^{©70} Ben Franscke, CC BY-SA 2.5, <https://commons.wikimedia.org/w/index.php?curid=7633823>

^{©71} Vlad Spears, CC BY-SA 2.0, <https://commons.wikimedia.org/w/index.php?curid=2930845>

^{©72} LeoKliesen00, Own work, CC BY-SA 4.0, <https://commons.wikimedia.org/w/index.php?curid=48087446>

^{©73} DeepSonic, CC BY-SA 2.0, <https://www.flickr.com/photos/deepsonic/6600702361>

^{©74} By Lippold Haken and Edmund Eagan CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=3710610>

^{©75} Reprinted with permission from Roger Linn.

^{©76} Klaus P. Rausch, CC0, <https://commons.wikimedia.org/w/index.php?curid=60729226>

^{©77} F7oor, CC BY-SA 2.0, <https://www.flickr.com/photos/f7oor/3992788445/>

^{©78} Pretzelpaws CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=142551>

^{©79} Reprinted with permission from Brad Ferguson.

Index

- μ -law, 19
- 5.1 surround sound, 19
- a-law, 19
- Ableton Live, 138
- action, 135
- ADC, 18, 113
- additive synthesis, 8, 14
- ADSR envelope, 35, 47, 66, 79
- aftertouch, 43, 138
- AHDSR envelope, 47
- AHR envelope, 47
- AIR Music Technology Loom, 33, 41
- Akai MPC series, 113
- algorithms, 109
- aliasing, 18, 70
- all notes off, 145
- all pass filter, 82, 127, 129
- all sound off, 145
- alternators, 31
- Amen Break, 113
- amplifier, 12, 34, 55, 57, 62, 63, 67
- amplitude, 13, 24, 81
- amplitude modulation, 77
- amplitude response, 81, 87, 88
- analog filter, 86
- analog synthesizer, 8, 11
- Analog-Digital Converter, 18, 113
- Analogue Solutions, 65
- angle modulation, 103
- angular frequency, 16, 87
- AR envelope, 47
- ARP 2500, 57
- ARP Instruments 2600, 57
- ARP Odyssey, 58
- arpeggiator, 12, 35, 52, 65
- arpeggio, 52
- asynchronous granular, 117
- attack level, 12, 35, 47
- attack rate, 47
- attack time, 12, 35, 47
- attenuation, 96
- AU, 11, 61
- audio interface, 11
- Audio Modeling, 134
- audio rate, 43
- Audio Unit, 11
- automated modulation, 43
- band limited wave, 70, 118
- band pass filter, 34, 65, 81, 101
- band reject filter, 81
- bandwidth, 101, 106
- Bartlett window, 131
- Beach Boys, 136
- Behringer BCR2600, 137
- Bell Labs Digital Synthesizer, 33
- Bessel function of the first kind, 106
- bilinear transform, 86, 92
- bin, 23
- bipolar, 43, 53
- bipolar envelope, 50
- bitrate, 19
- Blackman window, 123
- Blade Runner*, 59
- BLIT, 71
- blue noise, 69
- Bode plot, 86, 89
- Bode, Harold, 56
- breath controller, 138, 145
- brick wall filter, 82
- brown noise, 69
- Brownian motion, 69
- Buchla, Don, 56, 74
- buffer, 38
- Butterworth filter, 87, 92, 96, 98
- Camel Audio Alchemy, 33
- capacitive keyboard, 136
- cardinal sine function, 120
- carillon, 16
- Carlos, Wendy, 56
- carrier, 78, 104
- Carson's rule, 106
- Casio CZ series, 75
- Casio CZ-1, 75
- Casio CZ-101, 75
- Catmull-Rom, 38, 120

causal filter, 123
CC, 54
channel aftertouch, 139
channels, 19
Chariots of Fire, 59
Chebyshev Polynomials of the First Kind, 73
Chebyshev, Pafnuty, 73
chorus, 9, 65
Chowning, John, 103
Ciani, Suzanne, 57
Clavia Nord Lead, 60
clipping, 74
clock, 51
clock pulse, 143
Clockwork Orange, 56
Close Encounters of the Third Kind, 57
coefficients, 87
COLA, 131
comb filter, 82, 126
combiner, 12, 55, 62, 67, 76
companding, 19
complex conjugate, 28
compress, 74
Constant Overlap-Add, 131
Control Change, 54
control surface, 137
control voltage, 53, 65, 147
controller, 8, 11, 60, 135
convolution, 30, 83, 121, 130
convolution reverb, 130
Cooley, James William, 25
Cooley-Tukey FFT, 25
correlation, 121
Creative Labs Sound Blaster, 103, 115
cross fade, 76, 114
cubic interpolation, 120
cutoff frequency, 12, 81, 93
CV, 53, 61, 65, 147
CV/Gate, 65, 147

D-Beam, 136
DAC, 18, 113
DADSR envelope, 12, 47, 64
Dark Side of the Moon, 57
Dave Smith Instruments Prophet '08, 11, 63
Dave Smith Instruments Prophet 6, 61

DAW, 9, 11
DC offset, 107
DC Offset Bin, 24
DCO, 63, 67
decay rate, 47
decay time, 35, 47
decibel, 17
decimation, 117
Deep Note, 7
delay, 9, 65, 125
delay time, 12, 47
dernormals, 37
desktop synthesizer, 8, 12
detune, 12, 63
DFT, 23
digital audio workstation, 9, 11, 61, 114, 135, 141
digital delay line, 125, 133
digital filter, 86
digital synthesizer, 5, 8, 60
digital waveguide synthesis, 134
Digital-Analog Converter, 18, 113
Digitally Controlled Oscillator, 63, 67
diphthong, 101
Direct Form I, 85
Direct Form II, 85
Discrete Fourier Transform, 23
Doepfer Musikelektronik, 61, 65
Doppler effect, 128
downsampling, 106, 117
drawbars, 33
drawknob, 31
drum computer, 50
drum machine, 7, 8, 50, 113
drum pad, 138
drum synthesizer, 50
dry, 125
duophonic, 134
duty cycle, 68
dynamic range, 19, 135

E-Mu Systems, 113
early reflections, 129
East Cost synthesis approach, 57
echo, 9
Edisyn, 5, 53

effect, 65, 125
effects unit, 9
Eigenlabs Eigenharp, 139
electrical isolation, 141
Elektron Digitone, 111
ELP, 56
embouchure, 147
Emerson, Keith, 56
EMS Synthi, 57
EMS VCS 3, 57
Emu Morpheus, 114
Emu UltraProteus, 114
Ensoniq, 113
envelope, 35, 46, 79
envelope generator, 46
equalization, 65
Euler's Formula, 21
Eurorack, 61, 63, 65
exponential FM, 105
expression controller, 145
expression pedal, 43, 136
expressivity, 138

fader, 51
Fairlight CMI, 113
Fairlight Qasar M8, 33
Fast Fourier Transform, 25
feedback comb filter, 84, 127
feedforward comb filter, 83
FFT, 25, 131
filter, 12, 18, 34, 39, 55, 62, 66, 67, 69, 76
filter FM, 63, 111
filtering, 29
finite impulse response filter, 83, 130
FIR, 69, 83
first-order filter, 83
flanger, 65, 126
Flow, 5, 40
FM synthesis, 8, 43, 60, 67, 75, 78, 103, 104,
 128
foldover, 18, 70
foot controller, 145
formant, 101
formant filter, 34, 40, 101
forward comb filter, 126
four pole filter, 63, 65, 82

Fourier Series, 21
Fourier Transform, 14, 21
free, 46, 51
free LFO, 44
Freeverb, 130
frequency, 13, 24
frequency division, 58
frequency domain, 13
frequency modulation synthesis, 8, 14, 43, 67,
 75, 78, 103, 104, 128
frequency response, 81, 87
frequency warping, 93
fully diminished, 16
fundamental, 15, 107
Funky Drummer, 113

gain, 17, 82, 84, 91
gate, 52, 61, 65, 147
Gauss, Carl Friedrich, 25
Gizmo, 5, 50, 52
Gold, Rich, 116
Good Vibrations, 136
grain cloud, 117
grain envelope, 116
grains, 116
granular synthesis, 116
grid controller, 138
guitar processor, 138

Haken Continuum Fingerboard, 139
Hamming window, 29
Hammond Novachord, 58
Hammond Organ, 33, 128
Hann window, 116, 123, 131
hard sync, 78
harmonics, 15, 39, 67, 107
hertz per volt, 53, 147
high pass filter, 34, 65, 81
hold stage, 47
hum tone, 16
human factors, 138

IDFT, 23
IFFT, 28
IIR, 84
image synthesis, 30

Image-Line Harmless, 33
Image-Line Harmor, 30, 33, 41
impulse, 130
impulse response, 130
In The Box, 11
index of modulation, 105
infinite impulse response filter, 84
inharmonic, 107
instantaneous frequency, 104
instantaneous phase, 37, 75, 104
interpolation, 118, 119
Inverse Discrete Fourier Transform, 23
Inverse Fast Fourier Transform, 28
Inverse Fourier Transform, 14, 21
ITB, 11

jack, 56
Jankó keyboard, 138
joystick, 66
Jump, 59

Kaiser window, 123
Kakehashi, Ikutaro, 139
Karplus, Kevin, 132
Karplus-Strong, 132
Kawai K3, 33
Kawai K4, 53
Kawai K5, 33, 39
Kawai K5000, 33
keyboard, 135
Korg microKORG, 60, 63, 64
Korg MS10, 57
Korg MS20, 57
Korg MS2000, 64
Korg PS-3300, 59
Korg Wavestation, 7, 114

lag, 54
Lagrange interpolation, 119
Lagrange polynomial, 119
Lagrange, Joseph-Louis, 119
Laplace domain, 86
late reflections, 129
latency, 38, 123
leak, 28
leaky integrator, 72

legato, 36, 44, 46, 47, 51, 63
Leslie, 33
Leslie speaker, 128
LFO, 12, 35, 43, 64, 66
linear arithmetic synthesis, 114
linear FM, 103, 105
linear interpolation, 119, 127
linear phase filter, 82
local switch, 145
lossy compression, 19
low frequency oscillator, 12, 35, 43, 66
low pass filter, 12, 18, 34, 55, 57, 63, 65, 71, 81, 117, 118
low-pass feedback comb filter, 130
lower zone, 148

magnitude, 87
Make Noise 0-Coast, 61
matrix destination, 57
matrix source, 57
membrane, 136
MIDI, 8, 12, 54, 60, 135, 139
MIDI 2.0, 147
MIDI 2.0 profile, 149
MIDI channel mode messages, 145
MIDI data byte, 141
MIDI in, 140
MIDI interface, 11
MIDI merge, 140
MIDI out, 140
MIDI patchbay, 140
MIDI polyphonic expression, 145, 147, 148
MIDI router, 140
MIDI status byte, 141
MIDI thru, 140
minBLEP, 79
mini key, 136
mixer, 9, 66
mixing, 35, 63, 76
mixing console, 9
mixture, 31
mode, 15
modifier function, 53, 64
modular synthesizer, 7, 56, 65
modulation, 12, 35, 55, 62, 67
modulation amount, 53

modulation destination, 53
modulation matrix, 12, 53, 57, 59, 64
modulation source, 53
modulation wheel, 43, 137, 144
modulator, 78, 104
module, 56
Moiré patterns, 70
monitor, 9
monophonic, 8, 58
Moog MemoryMoog, 59
Moog Minimoog Model D, 8, 58
Moog, Robert, 56
morph, 41
MP3, 19
MPE, 145, 147, 148
MPE master channel, 148
MPE zone, 148
multi-stage envelope, 50
multi-track sequencer, 50
multi-track tape recorder, 9
multidimensional polyphonic expression, 148
multiple wavetable synthesis, 115
multitimbral, 114
Musical Instrument Digital Interface, 60, 135, 139
muti-tap delay line, 127

Native Instruments Razor, 33, 41
NCO, 67
New England Digital Synclavier II, 33
noise floor, 19
Non-Reserved Parameter Numbers, 145, 146
noodling, 9
normal form, 107
normalized sinc function, 120
notch filter, 34, 81
note, 15
note latch, 52
note length, 51, 52
note velocity, 52
Novation Launchpad, 138
Novation Remote Zero, 137
NRPN, 145, 146
nth order filter, 84
Numerically Controlled Oscillator, 67

Nyquist Frequency Bin, 24
Nyquist limit, 18, 24, 70, 117
Nyquist-Shannon sampling theorem, 118

Oberheim 4-Voice, 59
Oberheim 8-Voice, 59
Oberheim Matrix 1000, 63
Oberheim Matrix 6, 63
Oberheim Matrix 6R, 63
Oberheim Matrix series, 53, 60
Oberheim OB-X, 59, 61
Oberheim OB-Xa, 59, 61
Oberheim, Tom, 59
OBXD, 61
octave, 52
omni channel, 142
On The Run, 57
one pole filter, 82
one-shot envelope, 47
one-shot waves, 114
operator, 108
optoisolator, 141
orbit, 116
order, 82, 106
oscillator, 12, 55, 62, 67
Overlap-Add Method, 131
overshoot, 96
overtone, 15
OXE FM synthesizer, 109

pad, 49
Palm, Wolfgang, 115
pan, 17
parametric equation, 116
paraphonic, 58
partial, 13
passband, 96
patch, 7, 12, 56
patch bank, 144
patch cable, 7, 56
patch editor, 63
patch matrix, 53, 57
PCM, 114
PCM synthesis, 60, 67
Pearlman, Alan R, 57
period, 16, 37, 78

PG-8X, 61
phase, 24, 81
phase distortion synthesis, 75
phase modulation synthesis, 75, 103, 104
phase response, 81, 88, 96
phaser, 65, 132
phrase sampler, 113
physical modeling synthesis, 125, 132
piano, 135
pianoforte, 135
Pink Floyd, 57
pink noise, 69
pitch, 15, 17
pitch bend, 136
pitch bend wheel, 43, 137
pitch ribbon, 137
pitch scaling, 30, 113, 117
pitch shifting, 113, 117
PM synthesis, 75, 103, 104
pole, 82, 86, 88
Pollard Industries Syndrum, 137
polyphonic, 8, 11, 58
polyphonic aftertouch, 139
portamento, 36, 63, 136
PPG Wave, 115
PreenFM2, 110, 111
pressure, 138
prime, 16
pulse code modulation, 114
pulse code modulation synthesis, 67
pulse wave, 68
pulse width, 63, 68

quadraphonic, 13, 19
quality factor, 83

R2-D2, 7, 57
rackmount synthesizer, 8
Raiders of the Lost Ark, 57
ramp wave, 43, 67
random wave, 44
rank, 31
rate-based envelope, 47
RCA Mark I / II Electronic Music
 Synthesizers, 56
recorder, 9

rectangle function, 122
rectangular window, 29
release rate, 47
release time, 12, 35, 47
release velocity, 43, 139
repeating waves, 114
resampling, 71, 117
Reserved Parameter Numbers, 145, 146
reset all controllers, 145
resolution, 43
resonance, 63, 76, 83
rest, 51
resynthesis, 30
return, 9
reverb, 9, 129
reverberation, 129
reversing soft sync, 78
ring buffer, 125
ring modulation, 64, 77
ringing, 96
ripple, 96
RMI Harmonic Synthesizer, 33
Roads, Curtis, 116
Roger Linn Design LinnStrument, 139
Roland, 139
 Roland D-50, 114
 Roland Jupiter 8, 59, 60
 Roland JX-8P, 61
 Roland MKS-80 Super Jupiter, 60
 Roland TR-808, 50, 113
ROLI Seaboard, 139
roll-off, 82
ROM, 7
rompler, 7, 8, 60, 114
rotary speaker, 128
RPN, 145, 146
RPN Null, 146

S&H, 46, 66
sample and hold, 46, 66, 119
sampler, 7, 8, 60
sampling, 113
sampling function, 120
sampling rate, 18
sawtooth wave, 43, 63, 67
Schroeder reverberation, 130

Schroeder, Manfred, 130
second order filter, 84
sections, 51
self-centering, 137
SEM, 59
semi-modular synthesizer, 57, 61
send, 9
Sender, Ramon, 57
sequence, 50
sequencer, 8, 11, 35, 50
Sequential Circuits Prophet 5, 59, 61, 139
Sequential Circuits Prophet VS, 114
Short Time Fourier Transform, 30, 131
sidebands, 78, 105
signal to noise ratio, 19
Simmons SDS-5, 137
sinc function, 120
sine wave, 43, 68
single-cycle wave, 64, 71
Smith, Dave, 114, 139
Smith, Julius, 134
soft sync, 78
softsynth, 11, 61
software synthesizer, 5, 9, 11, 61
spectrogram, 13, 29
spline, 119
square wave, 43, 63, 67
state-variable filter, 59
step sequencer, 12, 50, 66
stereo, 13
STFT, 30, 131
stop knob, 31
stopband, 96
stops, 31
stored patch synthesizer, 12
Streetly Electronics Mellotron series, 113
strike tone, 16
Strong, Alexander, 132
suboscillator, 66, 69
Subotnick, Morton, 57
subtractive synthesis, 11, 55
sustain level, 12, 35, 47
SWAM, 134
swell box, 136
swell pedal, 136
swing, 51, 52
Switched-On Bach, 56
sync, 63, 64, 78
synchronous granular, 116
synth, 7
synthesizer, 7
sysex, 141
system exclusive, 141
table-based waveshaper, 73
tabletop synthesizer, 8, 12
tangent, 135
tape-replay, 113
Tasty Chips Electronics GR-1, 117
Taylor series, 21
Telharmonium, 33
tempo, 51, 52
The Shining, 56
The Thing, 136
theremin, 136
Theremin, Léon, 136
third dimension controller, 145
THX, 7
tie, 51
tierce, 16
time domain, 13
time stretching, 116
time-based envelope, 47
tonewheels, 31
track mute, 51
track solo, 51
tracking generator, 64
transfer function, 86, 87
transition band, 96
Trautonium, 55, 137
travel, 136
traveling wave, 133
tremolo, 35, 43, 128, 136
triangle wave, 43, 63, 67
triangular window, 131
Tron, 56
Tukey, John, 25
two pole filter, 65, 82
unipolar, 43, 53
unity gain filter, 82, 84, 91, 122

unweighted, 135
upper zone, 148
upsampling, 118
USB host for MIDI., 147

Välimäki, Vesa, 79
Van Halen, 59
Vangelis, 59, 137
variable bitrate, 19
variance, 44
VCA, 66, 79
VCO, 66, 67
vector synthesis, 8, 114
velocity, 43, 79, 135
velocity sensitivity, 79, 136
vibrato, 12, 35, 43, 103, 128, 136
virtual analog synthesizer, 9, 60, 64
Virtual Studio Technology, 11
visualization, 29
vocoder, 7, 30, 60, 64
voice, 11, 58
voice stealing, 58
volt per octave, 53, 105, 147
Voltage Controlled Amplifier, 66, 79
Voltage Controlled Oscillator, 66, 67
VST, 11, 61

Wakefield, Jeremy “Jezar”, 130
Waldorf Microwave, 115
Waldorf Music, 115

wave folding, 57, 74
wave sequence, 114, 116
wave shaping, 73
wave terrain, 116
waveguide, 134
wavetable, 114, 115
wavetable synthesis, 8, 34, 60, 67, 71, 115
weighted action, 135
weighted keyboard, 136
West Coast synthesis approach, 57
wet, 9, 125
white noise, 63, 68
Whittaker-Shannon interpolation formula, 121
wind controller, 138
window, 28, 75, 116, 122
windowed sinc interpolation, 106, 120
wrapping, 74

Xenakis, Iannis, 116

Yamaha CS-80, 59
Yamaha DX7, 8, 103, 109
Yamaha FS1R, 109
Yamaha TX81Z, 103, 109
Yamaha VL1, 134

Z domain, 86, 92
zero, 82, 86, 88
zero padding, 85, 130
zipper effect, 43, 46