# Computational Music Synthesis

A Set of Undergraduate Lecture Notes by

**Sean Luke**
Department of Computer Science
George Mason University

**First Edition**
Online Version 1.6
March 2023

**Copyright** 2019–2021 (Zeroth Edition), 2021–2023 (First Edition) by Sean Luke.

**Thanks to** Carlotta Domeniconi, Victoria Hoyle, Lilas Dinh, Dan Lofaro, 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, Robert Bristow-Johnson, 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, 2021, *Computational Music Synthesis*, first 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{ Luke2021Synthesis,
       author =    { Sean Luke },
       title =     { Computational Music Synthesis},
       edition =   { first },
       year =      { 2021 },
       note =      { Available for free at http://cs.gmu.edu/$\sim$sean/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 181, 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 Musician'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 Additive Synthesis** 21
3.1 History . . . 21
3.2 Approach . . . 24
3.3 Implementation . . . 26
3.4 Monophony, Paraphony, Polyphony, and Multitimbrality . . . 30
3.5 Architecture Examples . . . 32

**4 Modulation** 35
4.1 Low Frequency Oscillators . . . 35
4.2 Envelopes . . . 38
4.3 Step Sequencers and Drum Machines . . . 42
4.4 Arpeggiators . . . 45
4.5 Gate/CV and Modular Synthesizers . . . 46
4.6 Modulation Matrices . . . 46
4.7 Modulation via MIDI . . . 47

**5 Subtractive Synthesis** 48
5.1 History . . . 48
5.2 Implementation . . . 54
5.3 Architecture Examples . . . 55

**6 Oscillators, Combiners, and Amplifiers** 61
6.1 Oscillators . . . 61
6.2 Antialiasing and the Nyquist Limit . . . 64
6.3 Wave Shaping . . . 68
6.4 Wave Folding . . . 69
6.5 Phase Distortion . . . 71
6.6 Combining . . . 72
6.7 Amplification . . . 76

**7 Filters** 77
7.1 Digital Filters . . . 81
7.2 Building a Digital Filter . . . 84
7.3 Transfer Functions in the Laplace Domain . . . 85
7.4 Poles and Zeros in the Laplace Domain . . . 86
7.5 Amplitude and Phase Response . . . 86
7.6 Pole and Zero Placement in the Laplace Domain . . . 87
7.7 Analog Second-Order Filters . . . 90
7.8 Converting to Digital: the Z Domain and the Bilinear Transform . . . 92
7.9 Frequency Response and Pole and Zero Placement in the Z Domain . . . 95
7.10 Digital Second-Order Filters . . . 96
7.11 Filter Composition . . . 100
7.12 First Order and Ladder Filters . . . 101
7.13 Formant Filters . . . 104

**8 Frequency Modulation Synthesis** 107
8.1 Frequency and Phase Modulation . . . 107
8.2 Sidebands, Bessel Functions, and Reflection . . . 110
8.3 Operators and Algorithms . . . 112
8.4 Implementation . . . 114

**9 Sampling and Sample Playback** 117
9.1 History . . . 117
9.2 Pulse Code Modulation . . . 118
9.3 Wavetable Synthesis . . . 119
9.4 Granular Synthesis . . . 120
9.5 Resampling . . . 121
9.6 Basic Real-Time Interpolation . . . 123
9.7 Windowed Sinc Interpolation . . . 125
9.8 Implementation . . . 128

**10 Effects and Physical Modeling** 133
10.1 Delays . . . 133
10.2 Flangers . . . 134
10.3 Chorus . . . 135
10.4 Reverb . . . 137
10.5 Phasers . . . 140
10.6 Physical Modeling Synthesis . . . 140

**11 Controllers and MIDI** 145
11.1 History . . . 145
11.2 MIDI . . . 149
11.2.1 Routing . . . 150
11.2.2 Messages . . . 151
11.2.3 Control Change (CC) Messages . . . 155
11.2.4 Reserved and Non-Reserved Parameter Number Messages (RPN and NRPN) 156

11.2.5 Challenges . . . 157
11.2.6 MPE . . . 158
11.2.7 MIDI 2.0 . . . 159

**12 The Fourier Transform** **161**
12.1 The Discrete Fourier Transform . . . 162
12.2 Computing Amplitude and Phase . . . 164
12.3 Real-Valued Fourier Transforms . . . 164
12.4 The Fast Fourier Transform . . . 165
12.5 Windows . . . 168
12.6 The Short Time Fourier Transform . . . 170

**Sources** **177**

**Figure Copyright Acknowledgments** **181**

**Index** **185**

## List of Algorithms


0 Bubble Sort . . . . . 6
1 Simple Monophonic Additive Synthesizer Architecture . . . . . 26
2 Sine Table Initialization . . . . . 28
3 Sine Approximation . . . . . 28
4 Buffer Output . . . . . 29
5 Simple Low Frequency Oscillator . . . . . 36
6 Random Low Frequency Oscillator . . . . . 37
7 Sample and Hold . . . . . 38
8 Simple Linear Time-based ADSR . . . . . 40
9 Simple Exponential Rate-based ADSR . . . . . 41
10 Simple Parameter Step Sequencer . . . . . 42
11 Simple Monophonic Subtractive Synthesizer Architecture . . . . . 55
12 White Noise Sample . . . . . 63
13 Pink Noise Sample . . . . . 63
14 Initialize a Digital Filter . . . . . 83
15 Step a Digital Filter . . . . . 84
16 One-Shot Extraction . . . . . 129
17 Looped Extraction . . . . . 129
18 Windowed Sinc Interpolation . . . . . 129
19 Pitch Shifting for One-Shot Playback . . . . . 130
20 Grain Playback . . . . . 130
21 Pitch Shifting for Looped Playback . . . . . 131
22 Looped Single-Cycle Wave Playback . . . . . 131
23 Simple Wavetable Playback . . . . . 132
24 Delay Line . . . . . 133
25 Multi-Tap Delay Line . . . . . 136
26 Karplus-Strong String Synthesis . . . . . 141
27 The Discrete Fourier Transform . . . . . 163
28 The Fast Fourier Transform (FFT) . . . . . 167
29 The Inverse Fast Fourier Transform . . . . . 168
30 Multiply by a Window Function . . . . . 169
31 The Short Time Fourier Transform . . . . . 175
32 The Inverse Short Time Fourier Transform . . . . . 176

# 0 Preface

This book was developed for a senior computer science course I taught starting 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, effects, and physical modeling 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 4.
  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 4.
  http://cs.gmu.edu/∼sean/projects/edisyn/

- **Flow** is an unusual polyphonic additive modular software synthesizer. I refer to it in Section 3.
  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: v⃗ ← ⟨v_1, ..., v_l⟩ vector to sort   ▷ User-provided parameters to the algorithm appear here
                                           ▷ Then a blank space
2: repeat                                  ▷ Algorithm begins here
3:     swapped ← false                     ▷ ← always means "is set to"
4:     for i from 1 to l − 1 do            ▷ Note that l is defined by v_l in Line 1
5:         if v_i > v_{i+1} then
6:             Swap v_i and v_{i+1}
7:             swapped ← true
8: until swapped = false                   ▷ = means "is equal to"
9: return v⃗          ▷ 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**,[2] the famous trademark sound played before movies to indicate the use of **THX**. 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 fundamental 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

[1] An **ARP 2600**. See Figure 48 in Section 5.
[2] Deep Note was computer-generated by a software synthesizer written in custom C code.
[3] This is a derogatory but 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.

*Figure 1* Moog Minimoog Model D.[©2]

**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 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. When rackmount or desktop units were controlled from a keyboard version of the same synthesizer, they were known as **expanders**.

*Figure 3* Yamaha DX7.[©4]

Synthesizers also benefited 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 and related **digital signal processor** (or **DSP**) chips 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** (Figure 3). Digital synthesizer approaches that 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.

**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.

*Figure 4* Propellerhead Reason DAW.[©5]

Analog synthesizers have since seen a renaissance as musicians yearned for the warm-sounding, knobby, 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.

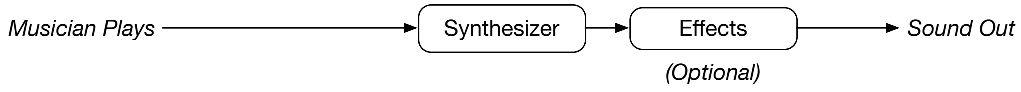
## 1.2 The Synthesizer Musician's Environment

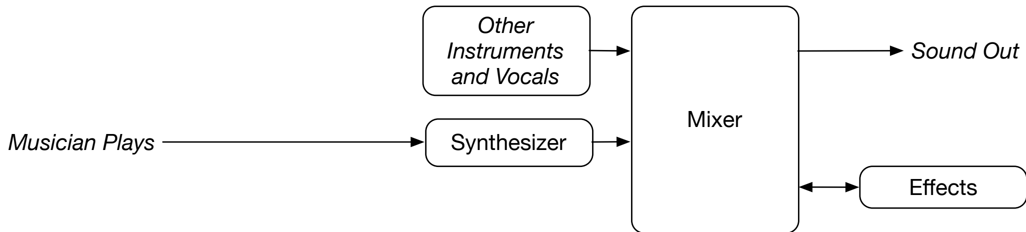
Music synthesizers are ubiquitous in the music and sound effects scene, and so appear in a wide range of of music performance and production scenarios. If you're unfamiliar with the architecture of these scenarios, it's useful to review a few common ones. The scenarios in question are shown in Figure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

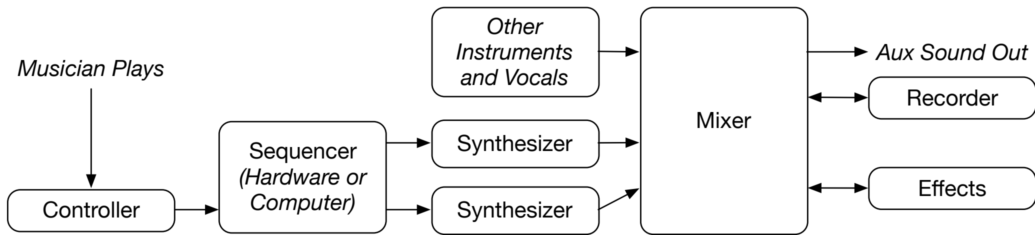
**Playing Around** This is the obvious basic scenario: you own a synthesizer and want to play it. This is sometimes called **noodling**. The important item here is the possible inclusion of an **effects unit**. Effects are manipulations of sound to add some kind of "texture" or "color" to it. For example, 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. A sound with no effects is **dry**.

**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 via automated means, a musician might construct an entire song by playing multiple parts into a



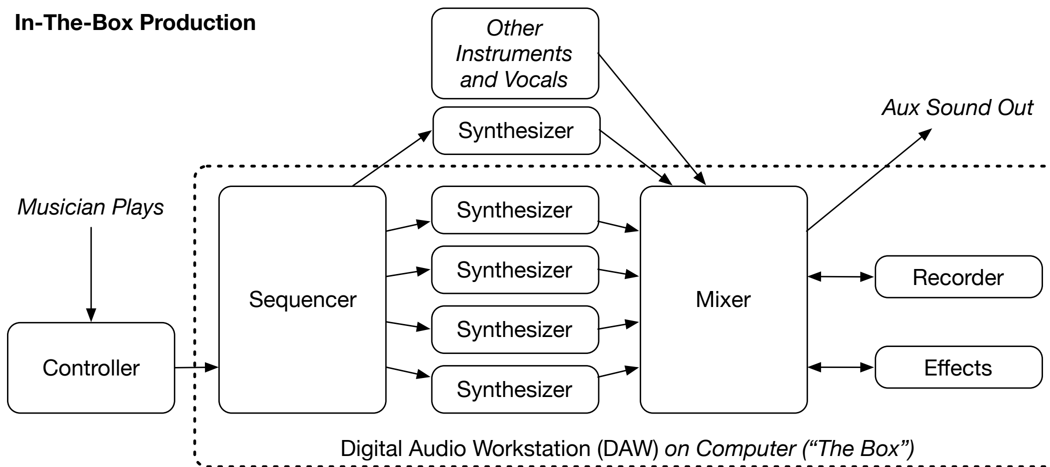




*Figure 5* Four common scenarios in which music synthesizers are used.

**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 often wouldn't play a synthesizer directly, but rather would play a **controller**, typically 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 **VST**s), 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 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.

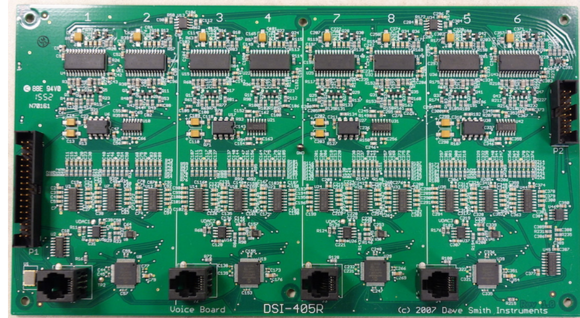



*Figure 7* Eight-voice circuitry for the Prophet '08.

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 **LFO**s 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.

## 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 $f(time)$. 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.

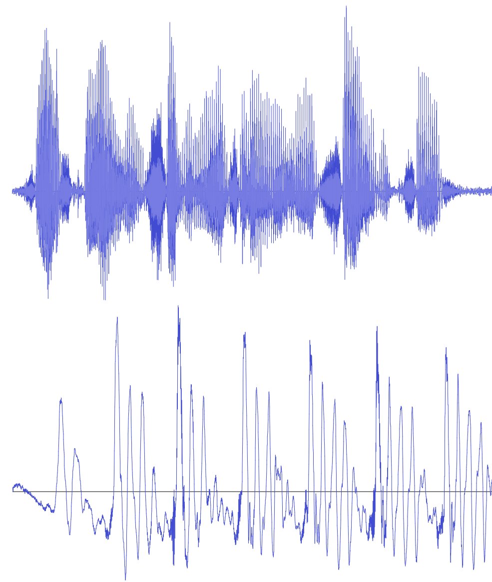


*Figure 8* A sound wave and zoomed-in portion. The $x$-axis is time, the $y$-axis is amplitude.

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**.

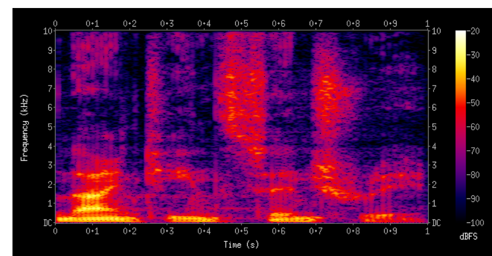


*Figure 9* A spectrogram of the spoken phrase "nineteenth century". The $x$-axis is time, the $y$-axis is frequency, and the color is amplitude.©7

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 frequency[4] 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.

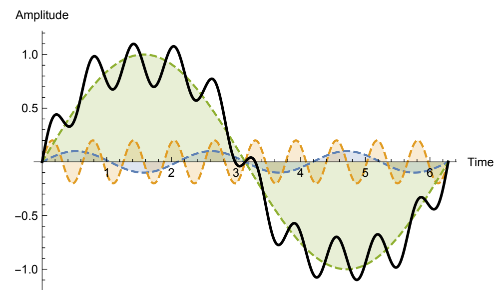



*Figure 10* Three sine waves (colored) and the result of adding them up (in bold).

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 depicting 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 perceiving the sound in the **time domain**.

[4]In 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 12.

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 near a hair's resonant 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.

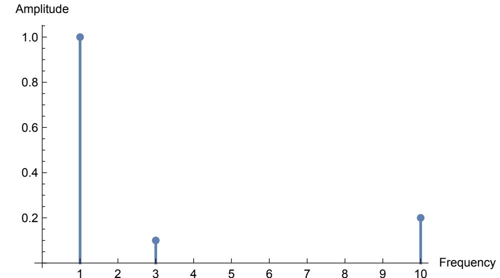



*Figure 11* The three sine waves from Figure 10, plotted by amplitude and frequency.

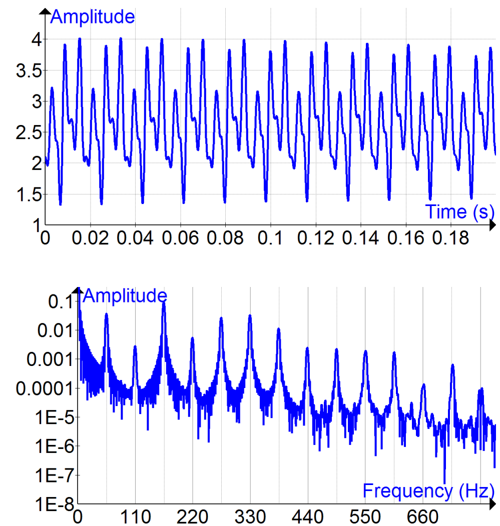



*Figure 12* Time Domain (top) and Frequency Domain (bottom) of a bass guitar note.[©8]

**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.

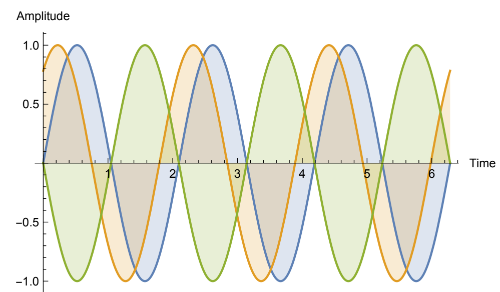



*Figure 13* Three identical sine waves which differ only in phase. Note that the blue and green sine waves have entirely *opposite* phase.

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, depending on the situation, we might *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**), rely heavily on *changes* in phase.

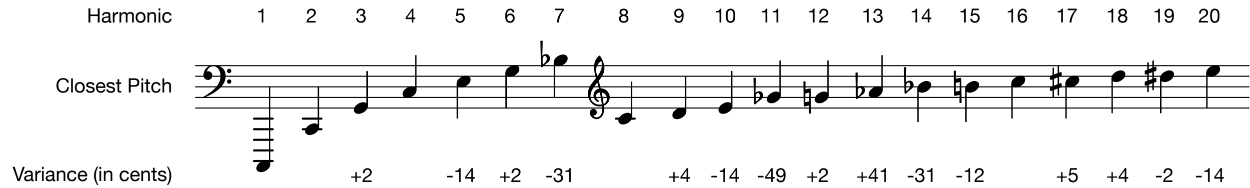



*Figure 15* Harmonic series. The fundamental (harmonic #1) is shown as a low C, and successive harmonics are shown with their equivalent pitches relative to it. The numbers below the notes indicate the degree to which the harmonic deviates from traditional pitches (in cents, that is, 1/100ths of a semitone).©9

**Harmonics and Pitch Identification** Most sounds which we perceive as "tonal" or "musical" have most 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**.

Many instruments, including woodwinds, brass, and strings among others, have partials largely structured as harmonics. These instruments are essentially fixed strings or tubes which can only vibrate according to certain **modes**. Consider Figure 14, where the ends of a violin string are fixed at 0 and $2\pi$ respectively. There are only so many ways that a violin string can vibrate so 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: air vibrations in its tube are essentially "fixed" at the two ends.

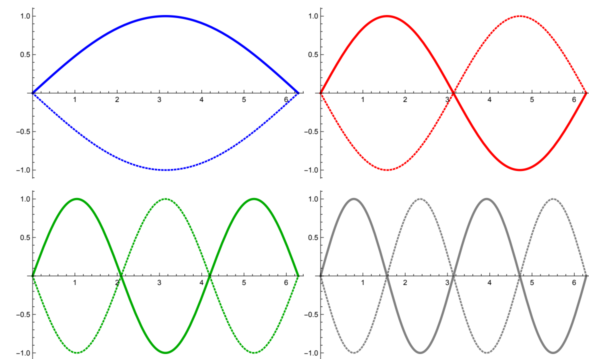


*Figure 14* Modes corresponding to the first, second, third, and fourth harmonics in a stringed instrument.

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.

[5]You might be wondering: *why* are they off? Traditionally 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, spaced such that the ratio between the frequencies of any two successive notes (A / A♭, say, or F/E) is 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**. However, 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, other temperament strategies have been proposed throughout history which adjust notes slightly in order to make them sound more harmonious together. Temperament strategies have been a matter of debate for many centuries.

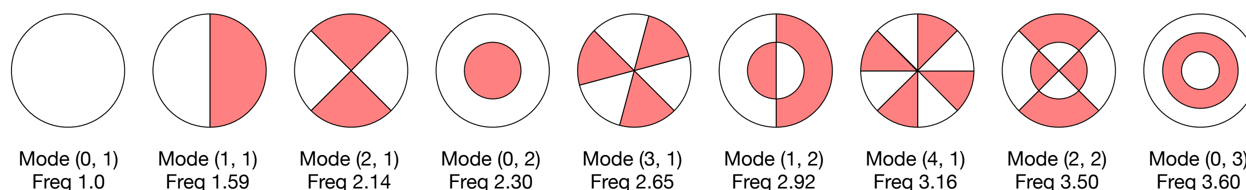



*Figure 16* First nine drum modes, ordered by frequency. Red regions vibrate up when white regions vibrate down and vice versa. Drum vibration patterns may combine these modes. Underneath each mode is the frequency relative to the first mode (the fundamental).

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 commonly 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 loud 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] which is 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** Frequency is often 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 $\omega$ 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.

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

[7] One 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 (with C, E♭, G, the E♭ makes a G♭ tierce). 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.

[8] In 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 is classically measured in **semitones**, that is, half-steps. Going from C to C# is one semitone, as is going from G to G#, etc. You'll also see the finer measure **cents**. A cent is 1/100th of a semitone.

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 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** When we are multiplying an amplitude or volume to make it louder or softer, we are said to be changing the **gain** of the signal. Amplitude is a positive value; but it's common to refer informally to the value of a wave as its "amplitude": and if we centered the wave at zero, it'd go both above and below that value, hence informally having a "negative amplitude" at certain times.

The *change* in amplitude of a signal, or the ratio between that signal and some reference signal, is given in terms of **decibels** (dB). Decibels are a relative measure and thus you will very often see *negative decibels* relative to some signal amplitude, to indicate sounds quieter than that signal.

Computing from decibels to raw amplitudes, or back, isn't very hard: if you have two sounds $i$ and $j$, with amplitudes $A_i$ and $A_j$ respectively, and sound $j$ is $d$ dB louder than sound $i$, then $A_j = A_i \times 10^{d/20}$. Conversely, $d = 20 \log_{10} \frac{A_j}{A_i}$. Doubling the amplitude is approximately an increase of 6dB. A doubling in perceived volume is often described as an increase of 10 dB.

**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 $0...2\pi$ (or if you like, $-\pi...\pi$). In Figure 13 the green and blue sine waves are out of phase of one another by $\pi$.

**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.

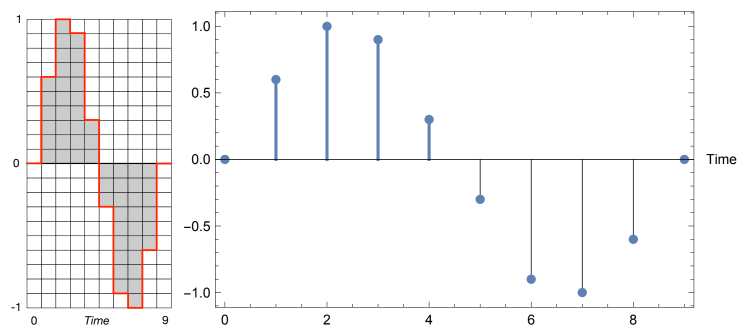



*Figure 17* A sine wave discretized and shown in grid form (left) and as a lollipop graph (right).

Instead, to play a digital sound, it is first fed to a **Digital-Analog Converter** or **DAC**.[9] The job of the DAC (and they're very good at it nowadays) is to convert the digital data into a perfectly smooth analog waveform. 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. For any time other than those specific sample times, there *is no value*: it's undefined.

**Sampling Rates, the Nyquist Limit and Aliasing** 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 collected, or played, every $1/(n \times 1000)$ of a second.

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** (or **Nyquist frequency**).[10] However it is possible to draw digital waves which would seem to contain within them higher frequency partials than the Nyquist limit, but this is an illusion. These waves do not present themselves as proper partials of those given frequencies, but instead create unusual artifacts with lower frequencies, a distortion known as **aliasing** (or **foldover**).[11] A sound wave (digital or analog) which has been carefully groomed to make sure that no frequencies exist above a certain maximum frequency is known as a **band limited wave**, and will have no aliasing when converted to a sampling rate greater than or equal to than twice that maximum. Thus to prevent aliasing when **resampling** sounds to lower sampling rates, audio devices first apply a **low pass filter** to strip out all frequencies higher than Nyquist (for the new sampling rate) before reducing a sound wave. For an extended discussion on aliasing (and it's pretty important), see Section 6.2.

It's important to understand the following critical (but very counterintuitive) notion. Consider the lollipop graph again (Figure 17, right subfigure). This graph was sampled at a certain sampling rate $n$ and thus has a Nyquist limit of $n/2$. It turns out that there exists *only one* analog signal, band-limited (to $n/2$), which passes exactly through the points in this lollipop graph. Thus this digital lollipop graph represents a *single* smooth analog signal. It's the job of a DAC to reproduce this analog signal from the digital signal.

**Common Sampling Rates** 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 the 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? The maximum frequency that humans can hear is roughly 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

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

[10]This is not to be confused with the **Nyquist rate**, which is just the sampling rate. The terminology is confusing.

[11]I 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.

filter at 20 kHz. Low-pass filters cannot cut off frequencies precisely: they need some degree of wiggle-room in frequency, and so require a sampling rate whose Nyquist limit is somewhat above 20kHz. 44.1 kHz, which provides a wiggle room of 2.05 kHz, was chosen by Sony in 1979 because it was compatible with certain video standards and so could be stored on early video recorders. 48 kHz has a similar history, being derived from different compromises due to competing video standards, and ultimately becoming the standard for video. 48 kHz seems more reasonable nowadays: it covers 20 kHz with even more wiggle room (4kHz) for the low-pass filter, and it's divisible by many integers.[12] 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 thus the *resolution* of the amplitude of the wave. The most common bit depth is 16 bits: that is, each sample is a 16-bit unsigned integer.[13] 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. Recall that there is exactly one *smooth* analog signal which passes through the lollipop graph points regardless of their bit depth, and a DAC will try to reproduce that signal. Rather, bit depth largely affects the **dynamic range** of the sound: the difference 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 translates to 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 **$\mu$-law** or **a-law** companding algorithms.[14]

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. As just one 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 of data

[12]Though 44.1 kHz is no slouch here: $44,100 = 2^2 \times 3^3 \times 5^5 \times 7^7$.

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

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

MP3 consumes in order to record a second of audio. But if some a sound has a lot of redundancy in it (as an extreme example: large segments of total silence), some compression schemes take advantage of this to compress different parts of a sound stream at different bitrates. These are known as **variable bitrate** schemes.

**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.

# 3 Additive Synthesis

An additive synthesizer builds a sound by producing and modifying a large set of partials — sine waves of different amplitude, frequency, and (sometimes) phase — and then adding them up at the end to form the final sound wave. 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 (all those frequencies, amplitudes, and phases). It's not easy to develop an additive synthesizer that isn't tedious to use. The high computational cost of adding all those sine waves in additive synthesizers has also restricted their availability compared to other techniques.

## 3.1 History

Additive synthesis is easily the oldest form of music synthesis, and if we relaxed its definition to allow adding up waves to 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 18, 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**.

*Figure 18* (Left) Ranks of Organ Stops.[©10] (Right) Organ Stop Knobs.[©11]

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**".

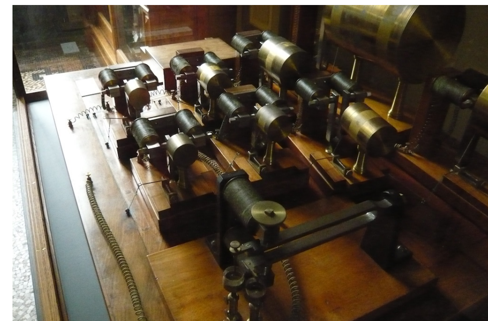


*Figure 19* Rudolf Koenig's synthesizer.[©12]

Early electro-mechanical synthesis devices were largely additive, using **tonewheels** (or **alternators**). A tonewheel, originally devised by **Hermann von Helmholtz** and later **Rudolf Koenig** (Figure 19), is a metal disk or drum with teeth (Figure 20). 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.[15] We can do a simple kind of additive synthesis by summing the sounds from multiple tonewheels at once.

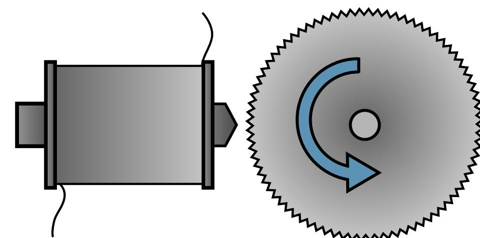


*Figure 20* Diagram of a tonewheel. As the wheel spins, its teeth alternatively get closer to or farther from an electromagnet, causing the magnet to produce a wave.[©13]

[15]This magnetic induction is essentially the same concept as an electric guitar pickup.

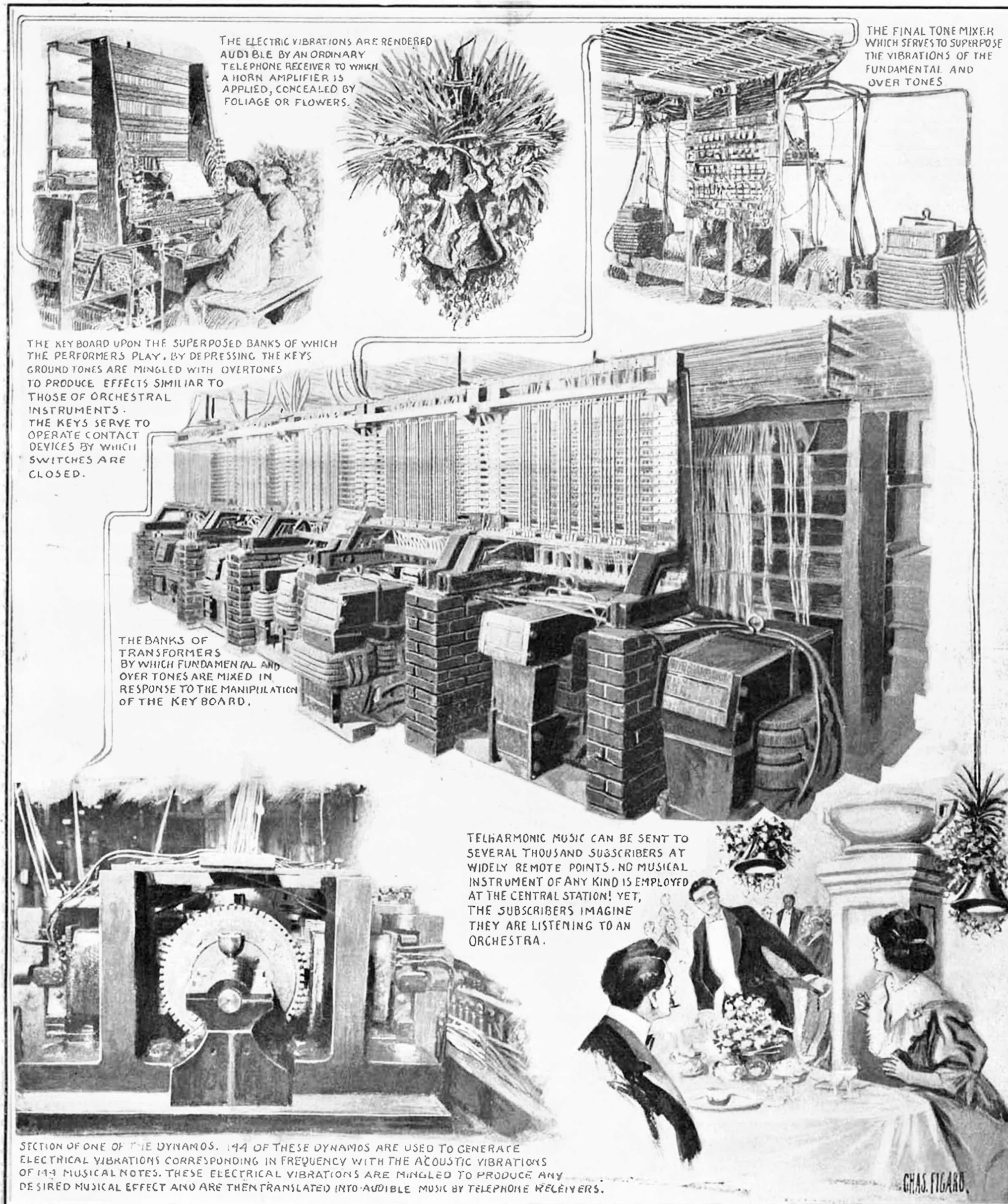
# SCIENTIFIC AMERICAN

[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
$3.00 A YEAR.




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

*Figure 21* The Telharmonium. Tonewheel (called a "dynamo") shown at bottom left.©14

The first significant electronic (or at least electric) music synthesizer in history, **Thaddeus Cahill's** enormous **Telharmonium**, relied on summing tonewheels and was thus an additive synthesizer. The Telharmonium was not small nor simple: as shown in Figure 21, it filled an entire room. As shown in the patent illustration for the Telharmonium, Figure 22, multiple tonewheels were attached to a spinning rod and so could produce many harmonics simultaneously for the same note. The motivation 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.

Tonewheels later formed the sound-generation mechanism 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.

The Hammond Organ was often paired with one or more **Leslie** rotating speakers, an early example of an effects unit. Leslie speakers added both **vibrato** and **tremolo** to the resulting sound. See Section 10.3, which discusses this effect in more detail. A Hammond B3 Organ is shown in Figure 23.

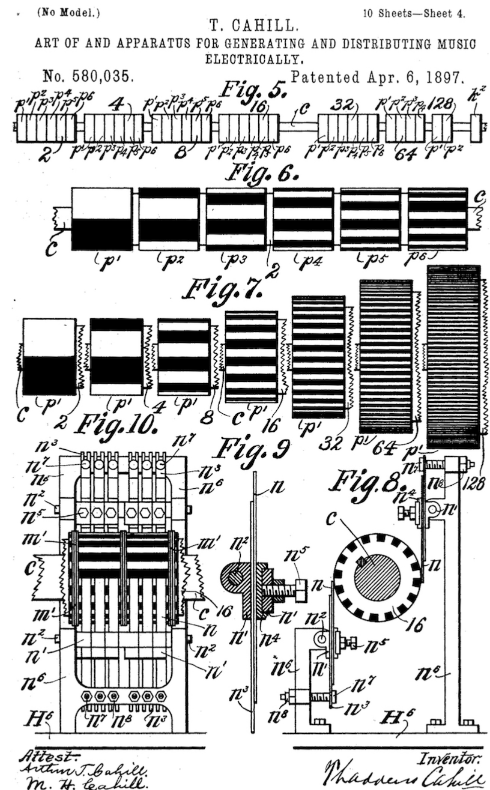



*Figure 22* Tonewheel figure from Thaddeus Cahill's 1897 Telharmonium patent.[©15]

*Figure 23* Hammond B3 Organ.[©16]

*Figure 24* Kawai K5000s.[©17]

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. The **Fairlight Qasar M8** generated samples by manipulating partials, and then used an **Inverse Fast Fourier Transform** (or **IFFT**) to produce the final sound (See Section 12). 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 Camel Audio (now Apple)'s **Alchemy**.

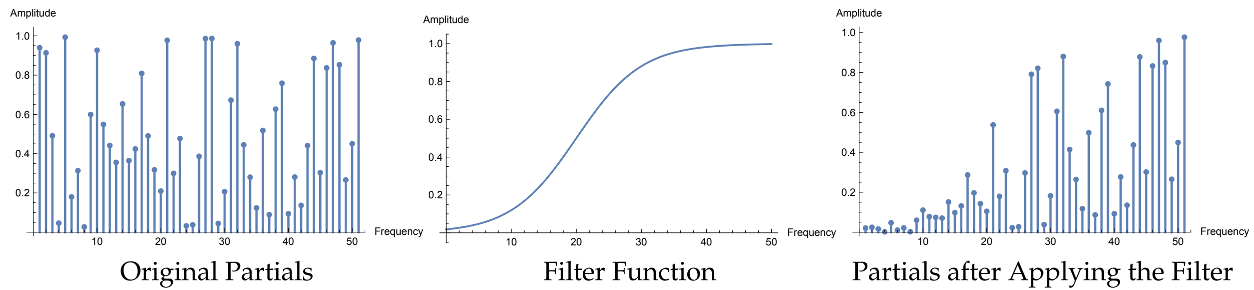



*Figure 26* Effect of a filter on the amplitudes of partials in an additive synthesizer.

## 3.2 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.

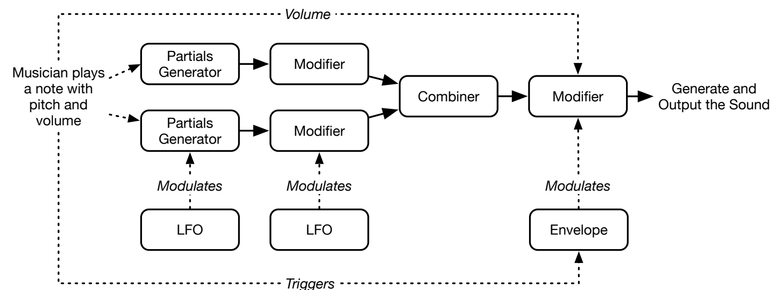



*Figure 25* One possible monophonic pipeline for an additive synthesizer (of many). Arrays of partials are passed from module to module.

Figure 25 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 26. 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 26 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 7.13).

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 7 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 multiplied against the partials based on their frequencies.[16]

- **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 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 mellower 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.

[16]If you'd like a basic filter function for an additive synthesizer, try using the two-pole filter amplitude response equations in Section 7.7. 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 $\omega$ (in radians again), multiply its amplitude against $1 \big/ \sqrt{\left(1 - \omega^2/\omega_0^2\right)^2 + \left(\omega/(\omega_0 Q)\right)^2}$ to get a basic low-pass filter. To convert a frequency from Hz to radians, just multiply by $2\pi$. Also, resonance normally doesn't drop below $Q = \sqrt{1/2}$, which is generally considered the minimum "no resonance" position.

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 4.

### 3.3 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 1** *Simple Monophonic Additive Synthesizer Architecture*

```
1: M ← ⟨M_1, ..., M_m⟩ modules
2: tick ← 0
3: counter ← 0
4: δ ← 0
5: α ← interpolation factor
6: ticksPerUpdate ← number of ticks to wait between updates    ▷ 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 ← tick +1
15:    counter ← counter +1
16:    δ ← (1 − α) × δ + α
17:    if counter >= ticksPerUpdate then
18:        counter ← 0
19:        δ ← 0
20:        for i from 1 to m do
21:            Update(M_i, tick)
22:    return OutputSample(tick, δ)
```

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 $\delta$ and $\alpha$? The call to OutputSample(...) in Algorithm 1 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 $\delta$ 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 1 we're passing in a $\delta$ 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 $\alpha$ such that, by the time ticksPerUpdate ticks have expired, $\delta$ 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**[17] floating-point 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 $\pi$.

[17] Denormalized numbers are a quirk of the IEEE 754 floating point spec. They are a set of non-zero numbers between the positive and negative values with the smallest exponent. For doubles, that means they're roughly at $-2^{-308} < d < +2^{-308}$, $d \neq 0$. Math with denormals is custom and must be handled with very slow hardware, or worse, in software, if your language doesn't automatically round denormals to 0. As a result they can be hundreds of times slower or worse depending on your CPU. You don't want to mess with that.

- 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 $\Delta t$ 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:

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

For our purposes, $\mod 1$ is the same thing as saying "keep subtracting 1 until the value is in the range 0...1, excluding 1." In Java, $x \mod 1$ (for positive $x$, which is our case) is easily implemented as `x = x - (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 $0...2\pi$, 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 2** *Sine Table Initialization*

1: Global $S_0 ... S_{2^n-1} \leftarrow$ array of $2^n$ numbers ▷ I myself use $n = 16$, so $2^n = 65536$

2: **for** $i$ from 0 to $2^n - 1$ **do**
3: $\quad S_i \leftarrow \sin(2\pi i / 2^n)$

**Algorithm 3** *Sine Approximation*

1: $x \leftarrow$ argument

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

You can make this much more accurate by interpolating with the **Catmull-Rom** cubic spline (Equation 6, page 124). To do this, let $\alpha = x \times \frac{2^n}{2\pi} - \left\lfloor x \times \frac{2^n}{2\pi} \right\rfloor$. Then let $f(x_1) = S_{(i-1 \mod 2^n)}$, $f(x_2) = S_{(i \mod 2^n)}$, $f(x_3) = S_{(i+1 \mod 2^n)}$, and $f(x_2) = S_{(i+2 \mod 2^n)}$. Then apply Catmull-Rom, and return $f(x)$. Slightly slower than direct table lookup, but still much 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 an up to 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 fitful 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 4** *Buffer Output*

```
1: A ← array of samples                          ▷ Large enough to keep the buffer happy

2: for i from 0 to length of A − 1 do
3:     A_i ← Tick()
4: AddToBuffer(A)                                ▷ Presumably this is blocking
```

**Dealing with Drifts in Phase** We've largely trivialized phase as less important to human hearing than amplitude and frequency: but that's not quite true. Phase plays a significant role in the color of sounds, particularly mid-to-low frequency ones. As a result, at the very least you'd like partials to not start varying in phase (unintentionally) relative to one another as the sound progresses, or its tonal character may change. This is particularly noticeable if the partials, when summed up, approximately form waves with large vertical drops such as a **sawtooth** or **square wave** (see Figure 31 on page 37 for illustrations).

If the partials were permitted to change in frequency relative to one another over time, you might have a problem. Imagine you had a partial at frequency $N$ and another at $2N$, both in phase. If the $2N$ partial was temporarily (and intentionally) changed to $2.123N$, and then returned to $2N$, the two partial generators would now be out of phase relative to one another, and wouldn't sound quite the same. Even if the partials were fixed in frequency (they were only permitted to be harmonics), it's possible their phases might drift relative to one another due to numerical instability.

Here are some strategies for dealing with this. First, you could ignore it. Second, some additive synthesizers simply reset the phases of a voice's partials every time it's used to play a new note. Third, instead of wrapping around $x_i$, compute it directly each time as $x_i^{(t)} \leftarrow t \times f_i \times \Delta t \mod 1$. This would guarantee that when we returned to $2N$ it'd be back in phase: but even subtle changes in frequency might create pops just like sudden changes in amplitude, for moderate $t$. I'd do #2.

## 3.4 Monophony, Paraphony, Polyphony, and Multitimbrality

A **monophonic** synthesizer can only play one note at a time. A **polyphonic** synthesizer can play multiple notes at a time, each effectively played through its own independent synthesizer (called a **voice**). Most early synthesizers were monophonic. Recall the additive pipeline shown before in Figure 25 (page 24). This essentially shows a monophonic pipeline, and likewise Algorithm 1 (page 26) describes a monophonic synthesizer architecture. (Though they're introduced here for additive synthesizers, monophony and polyphony are qualities of all synthesizers.)

**Monophonic Note Resolution** What happens when we play two notes on a monophonic synthesizer keyboard? Which one plays? The approach taken to resolve this impacts on the playability and performance options on the machine. Monophonic synthesizers have adopted different **note priority** rules: for example, the lowest note would get priority, or the highest note. Perhaps the most useful priority rule is to have the **last note played** get priority. Another issue that might arise: what should happen when a note is played, and then while it is being held down, another note is played which assumes priority? One option is to have the synthesizer simply reset itself and play the second note from its start. Another option, commonly called **legato**, is to continue playing the voice in the style of the first note but to shift its pitch to the new note. A third option, called **glide**, works like legato, but over time *slides* the pitch from the first note to the new note.

**Paraphony** As monophonic synthesizers developed, they gained the ability to have their two or three oscillators play different notes at once. This was done by having them play different pitches (the different notes): but these sounds were still combined and jointly pushed through a single modification stage together as one voice, whose parameters and modulation triggers were dictated by only the priority note, as shown in Figure 27. This so-called **paraphony** is a stand-in for true polyphony — the voices aren't independent — but it was a useful enough extension to monophonic synthesizers.

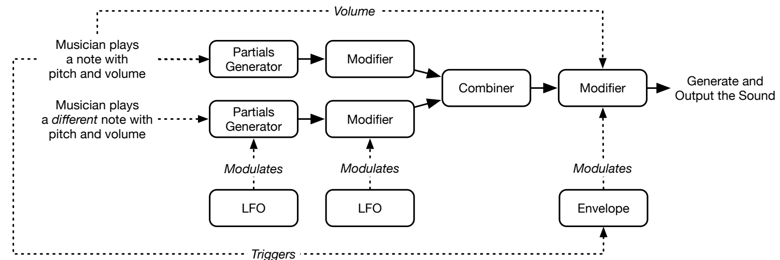



*Figure 27* A paraphonic pipeline for an additive synthesizer. Notice that the oscillators play different notes, but the volume (and other features discussed in Section 7, such as filter cutoff), is controlled only by one note. Compare with a *monophonic* pipeline as shown in Figure 25, page 24.

**Polyphony** A polyphonic synthesizer can be thought of as some $N$ completely independent monophonic synthesizers — the *voices* — each programmed with the same patch, as shown in Figure 28. When you play a note, one of these $N$ synthesizers is assigned to this note, reset, and instructed to begin playing. But which voice should be assigned? The most common approach is to assign the *least recently assigned* voice, that is, the oldest one.

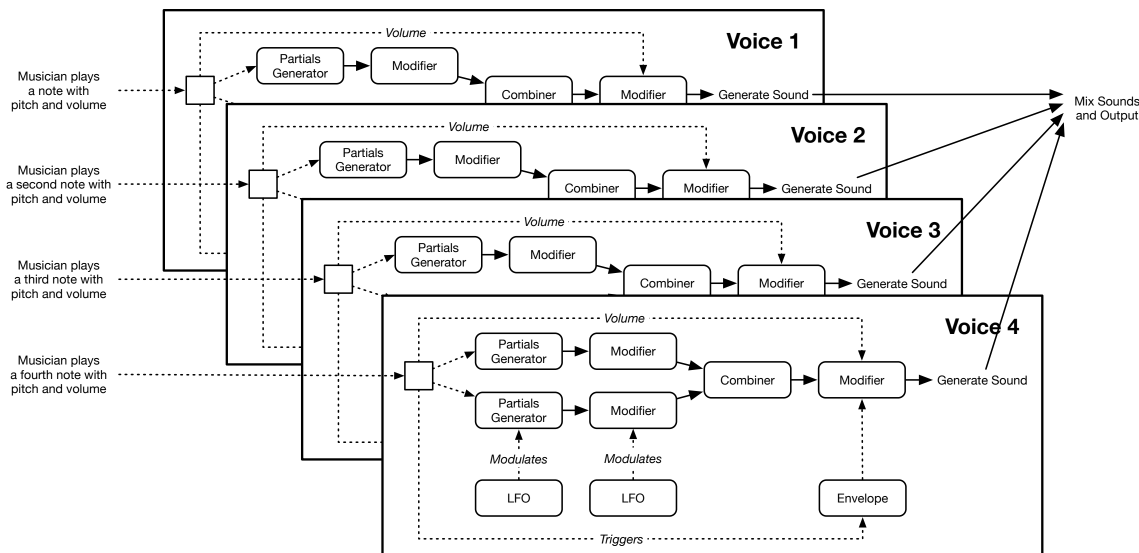



*Figure 28* A four-voice polyphonic pipeline for an additive synthesizer. As each new note is played, a different voice (essentially a monophonic synthesizer) is assigned to play that note. If there are no more voices, one is reassigned to play the next note (commonly the one assigned the furthest in the past).

One difficulty that arises with polyphonic synthesizers is the issue of **voice stealing**. Let's say you have a four-voice polyphonic synthesizer. You play a four-note chord. Then while playing it, you also play a fifth note. The synthesizer will be forced to reallocate one of the four voices to the new note. The allocated voice will immediately stop playing its old note and play the new, likely completely different note. This will probably result in a "click" or "pop" as the voice is lurched from one wave to another. Voice stealing obviously happens in a monophonic synthesizer as well, but little can be done about that.

You'd like to avoid voice stealing: this means having enough voices that there's always one available. But how many is enough? You'd think it'd be the largest chord you're ever likely to play — perhaps 8 notes? 10 notes? But you'd be wrong. In many patches, after you let go of a key, the note continues to play as it slowly dies out (over a so-called **release time**). This means that you could play a large chord, and play a *second* or even *third* chord before the first chord has entirely faded out and is available to reallocated without the clicks and pops of voice stealing. Thus a synthesizer benefits from a large number of notes, perhaps 16 or more.

Many early synthesizers could only afford to offer five or six note polyphony, which wasn't great but was serviceable. Later synthesizers offered 8 or more, and some digital synthesizers offer a huge number of voices, so many that you could never play them all. Which brings us to...

**Multitimbrality** A **multitimbral** synthesizer is a polyphonic synthesizer with a twist: its voices can *play different patches*. You might have a 24-voice multitimbral synthesizer which has allocated 8 voices to a pad sound, 8 voices to a lead, 2 voices to a bass, and 6 voices to one drum sound each. In a multitimbral synthesizer, you could play the pad on the keyboard, but (for example), let a computer play the lead, bass, and drums remotely via MIDI (see Section 11). Or you might split your keyboard so that you can play the pad in the upper notes with your right hand and the bass in the lower notes with your left hand.

Multitimbral synthesizers came into their own in the 1990s as rackmount synthesizers were developed to work with a computer to play an entire song. Most multitimbral synthesizers have special **multimode** patches in addition to standard single patches. The idea is as follows: each single patch holds the parameters which collectively defined of the sound of a voice, as usual. You can play a single patch by itself. Or you can instead load a mutlimode patch. A multimode patch contains parameters which define some $N$ **parts**. Each part has a reference[18] to some single patch in your memory, plus information such as how many voices should be allocated to that part, how it should be played (for example, via the keyboard; or just the upper part of the keyboard; or remotely over MIDI), which audio port it should output its sound to, and so on. When a synthesizer loads a multimode patch, it will also load each of the single patches referred to in the multimode patch's parts, allocate voices to them, and set them up.

## 3.5 Architecture Examples

Here are two actual architectures which fall in the architectural implementation discussed before.

*Figure 29* Kawai K5m

**Kawai K5** If you disregard the Hammond Organ, Kawai has probably produced more additive hardware synthesizers than any other company. This particular 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.
- 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. See Section 7.13.

[18] Except for a few rare cases (such as the Nord Lead 2, see Figure 60, page 53) the multimode patch won't *contain* copies of the single patches, but rather just refer to their patch locations. Thus if you modify single mode patches, this would also effect the multimode patches which refer to them.

- 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.[19]

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 16-voice polyphonic, and up to 16-part multitimbral. The K5 was also not a particularly good sounding synthesizer.[20] But ten years later, Kawai tried again with the K5000 series (Figure 29) and produced a far better sounding machine. 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!)

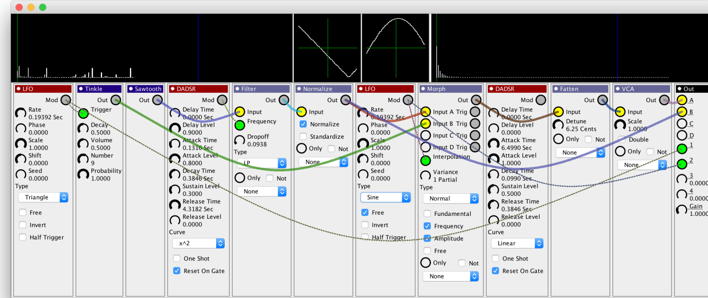


*Figure 30* Flow

**Flow** Flow is a fully modular, polyphonic, multitimbral, 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*, complete 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 (left) modules to produce modulation and partials information it wishes to output to later downstream modules.[21] 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

[19]There was no harmonic 64. I don't know why.

[20]Believe me. I owned one and upgraded it considerably. Figure 29 is a picture of my K5m with a brand-new screen.

[21]This 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, usually to 256. It also disregards phase, and a partial only has a frequency, an amplitude, and an ID. Flow can manipulate the frequency and amplitude of partials in a wide variety of ways and can combine and **morph**[22] partials from multiple sources.

[22]Morphing 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.

# 4 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**, the **pitch bend wheel**, so-called **expression pedals**, and from the keyboard's **velocity**, **release velocity**, and **aftertouch**, among others. These modulation sources tend to effect *all voices simultaneously* in a polyphonic synthesizer. For definitions and more information on these modulation interface options, see Section 11.

- **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. These modulation sources tend to be allocated to *separate, individual voices* in a polyphonic synthesizer.

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 $0...N$ (we could just think of this as 0...1). Other parameters, such as pitch bend, might be **bipolar**, meaning that their range is $-M... + M$ (perhaps simplified to $-1/2... + 1/2$ or $-1... + 1$). 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 11.2) — can be very coarse, often just 7 bits (0...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.

## 4.1 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.[23] 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 $-1... + 1$. LFOs often come in a number of classic shapes, including **sine, square, triangle,** and both **sawtooth** and **ramp** (Figure 31). LFOs have a number of parameters, including at least the *rate* or *frequency*, and an *amplitude* or *amount*.

[23] One 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 8.

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 Section 11.2.2). 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**.[24]

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 3.3), 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 5** *Simple Low Frequency Oscillator*

```
 1: r ← rate                                                    ▷ In cycles per tick
 2: type ← LFO type
 3: free ← is the LFO free-running?
 4: legato ← did a legato event occur (and we care about legato)?

 5: global s ← 0                                                ▷ Current state (0...1)

 6: procedure Note Pressed
 7:     if not free and not legato then
 8:         s ← 0

 9: procedure Update
10:     s ← s + r
11:     if s ≥ 1 then
12:         s ← s mod 1                       ▷ Easily done in Java as s = s - (int) s
13:     if type is Square then
14:         return { −1         s < 1/2
                   { 1          otherwise
15:     else if type is Triangle then
16:         return { s × 4 − 1  s < 1/2
                   { 3 − 4 × s  otherwise
17:     else if type is Sawtooth then
18:         return (1 − s) × 2 − 1
19:     else if type is Ramp then
20:         return s × 2 − 1
21:     else                          ▷ Type is Sine. See Section 3.3 for fast Sine lookup
22:         return sin(s × 2π) × 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 31. We might adjust the **variance** in the choice of new random target locations. I'd implement it like this:

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

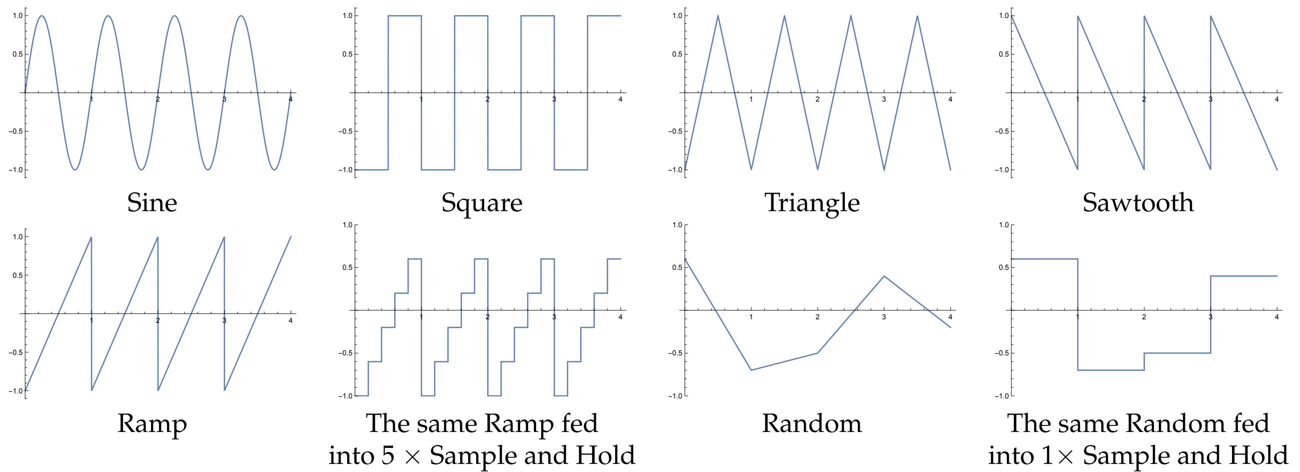



*Figure 31* Various Low Frequency Oscillator wave functions.

**Algorithm 6** *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 $-2...2$. 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. At the start of each 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 7** *Sample and Hold*

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

5:  global s ← 0                                          ▷ Current state (0...1)
6:  global target ← 0

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

11: procedure Update
12:     s ← s + r
13:     if s ≥ 1 then
14:         s ← s mod 1                ▷ Easily done in Java as s = s - (int) s
15:         target ← x
16:     return target
```

Sample and hold can be applied to any modulation source to produce a "discretized" version of the modulation, but it's particularly common to apply it to an LFO. Applying sample and hold to sawtooth, ramp, and triangle is common, but by far the most common application is sample and hold applied to a random LFO wave: indeed on many synths the "sample and hold" (or "S&H") option is *only* applied to random waves. The coarseness of discretization would depend on the sample and hold rate. Figure 31 shows examples of sample and hold, at two different frequencies, applied to ramp and random waves.

## 4.2 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**.

**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.

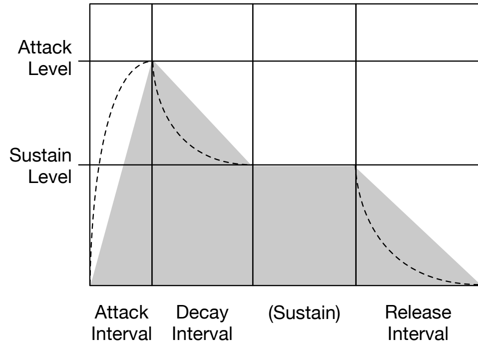



*Figure 32* An ADSR envelope. Gray region shows a linear rate of change. Dotted lines show an exponential rate of change.

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 32, 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**. The choice of rate versus interval or time depends on the synthesizer, and different manufacturers make different choices.[25] Exponential rate is straightforward to implement in analog synths, and can mimic the exponential decay of certain musical instruments, and so it's historically much more common.

Building an exponential rate-based envelope is easy. 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 $y \leftarrow (1 - v) \times \textit{attack level}$. Similarly if we were dropping from the attack level to the sustain level, we'd set $v = 1.0$, and repeat this trick, but define the level as $y \leftarrow v \times \textit{attack level} + (1 - v) \times \textit{sustain level}$. 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 a specified 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 period is eliminated: the envelope attacks, then decays to the sustain level, then immediately releases to zero. As is the case for an LFO, it is very common for ADSR envelopes to respond to **legato** by not resetting.

[25]There 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.

**Linear Time-based ADSR Implementation** This kind of ADSR just requires you to do linear interpolation through the attack, decay, and release stages.

**Algorithm 8** *Simple Linear Time-based ADSR*

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

 5: global s ← 0                                          ▷ Current state
 6: global i ← 0        ▷ Current stage. Attack=0, Decay=1, Sustain=2, Release=3, Done=4
 7: global p ← 0                                          ▷ Current parameter value.
 8: global p' ← 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 ← 0
12:         s ← 0
13:         p' ← 0
14:         p ← 0

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

20: procedure Update
21:     s ← s + r
22:     while s ≥ X_i do           ▷ Note that for Sustain (2) and Done (4) this will never be true
23:         s ← s − X_i                       ▷ We don't reset to 0, but to our leftover time
24:         p' ← p
25:         i ← i + 1                                     ▷ Go to next stage
26:     γ ← s/X_i                  ▷ This is assuming that X_i ≠ 0, which ought to be the case
27:     p ← (1 − γ) × p' + γ × Y_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 okay as an approximation. To do this I'd just insert the following immediately after line 26:

$$\gamma \leftarrow \gamma \times \gamma \times \gamma \times \gamma$$

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

**Exponential Rate-based ADSR** Because it is multiplying rather than adding, an exponential rate-based envelope will never reach its target, Zeno's Paradox style. Thus we need a threshold

variable, $\epsilon$, which tells us that we're "close enough" to the target to assume that we have finished. This value should be small, but not so small as to get into the denormals (See Footnote 17, page 27).

**Algorithm 9** *Simple Exponential Rate-based ADSR*

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

5:  global s ← 1                                                          ▷ Current state
6:  global i ← 0         ▷ Current stage. Attack=0, Decay=1, Sustain=2, Release=3, Done = 4
7:  global p ← 0                                                ▷ Current parameter value.
8:  global p' ← 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 ← 0
12:         s ← 1
13:         p ← 0
14:         p' ← 0

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

20: procedure Update
21:     s ← s × X_i                                                  ▷ Exponential dropoff
22:     if s ≤ ε then                 ▷ Note that for Sustain (2) and Done (4) this will never be true
23:         s ← 1
24:         p' ← p
25:         i ← i + 1                                                 ▷ Go to next stage
26:     p ← s × p' + (1 − s) × Y_i                               ▷ Compute interpolated value
27:     return p
```

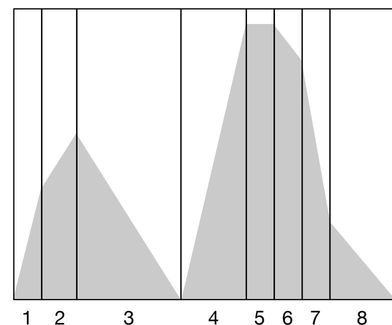


*Figure 33* Eight-stage envelope.

**Multi-Stage Envelopes** A multi-stage envelope has some $N$ stages, each with its own target level and time interval or rate. Figure 33 shows an eight-stage envelope. Envelopes of this kind are often used to make slow, sweeping changes over long periods of time for **pads**.[26]

Multi-stage envelopes often loop through a certain stage interval $M...N$ as long as the note is held down (a looping sustain), then finish out the remaining stages during release. If $M = N$, then the envelope effectively has one sustain stage along the lines of an ADSR. It's plausible for $M...N$ to encompass the entire envelope.

[26] A 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 other than that their values can range anywhere from $-1...+1$ instead of from 0...1.

## 4.3 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**.

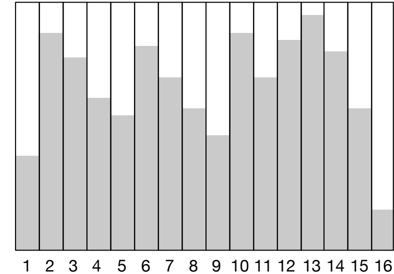



*Figure 34* Step sequencer pattern.

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 typically loops to the beginning when its stages have been expended. Figure 34 shows a possible step sequencer pattern (known, not surprisingly, as a **sequence**).

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 10** *Simple Parameter Step Sequencer*

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

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

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

In a synthesizer, step sequencers are not just used to modulate parameters, but are even more often used to play notes. To do this, each step must provide more than a single piece of data, including as note pitch, **velocity** (note volume), and perhaps auxiliary information such as whether the step is in fact a **rest** (meaning "don't play anything this step") or a **tie** (meaning "continue

playing the previous note"). Most such step sequencers are also **multi-track**, meaning that they maintain some $M$ parallel arrays of sequences (the "tracks"), all of which are advanced in sync.

Multitrack sequences can also often be used to indicate **drum beats**: since the drum isn't pitched, the beat doesn't need to include note information. Furthermore because the drum beat is just a pulse, ties are also not relevant. Drum beats have their own special terminology. A louder than usual drum beat is known as an **accent**, and a quieter than usual one is known as a **ghost note**. Drum beats might also be **flams**, that is, two or more hits in very rapid succession to create a fattened beat sound.

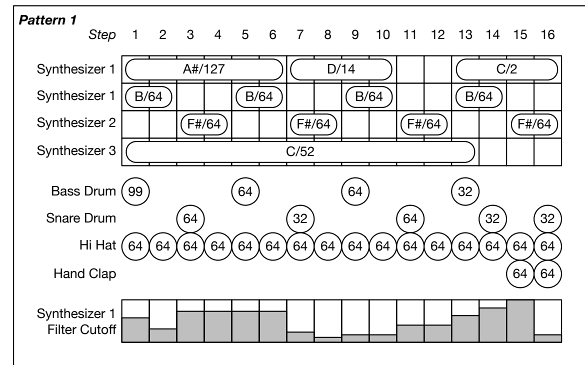



*Figure 35* Multitrack 16-step sequencer pattern, with four tracks of synthesizer notes (of different lengths), four tracks of drum beats, and one track of sequenced modulation information. Numbers in notes or beats indicate velocity. Don't play this, it sounds horrible.

Sequences could in fact consist of some $M$ tracks of notes, or of drum beats, or of sequenced information, or all three in together, as shown in Figure 35. A looped sequence of some set of multiple tracks is called a **pattern**. Patterns can be any length, though 16 beats (or on **bar**) is common for drum beats in much electronic dance music. We might also add to a pattern information about its **tempo** and **swing**. Swing is the degree of syncopation in a pattern, and is usually implemented by delaying the even steps in a pattern by some percentage of the total step time.

Many early step sequencers would also provide **song mode**, where you defined multiple patterns, and then strung them together to form a complete "song", as shown in Figure 36. Song mode was born out of the limited memory constraints of early sequencers, often for drums, and it has a serious restriction: the various tracks in patterns must stop and start at hard boundaries and cannot overlap. This is fine for some kinds of music but very limiting for others. Incredibly most modern hardware step sequencers *still* only have song mode.[27]

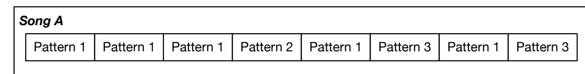



*Figure 36* Song mode.

**Linear Arrangers** Sequencers do not just appear inside synthesizers: there are also dedicated multi-track sequencers which drive synthesizers remotely via MIDI. Some have just song mode, but others have more sophisticated capabilities. For example, sequencers such as the **Yamaha QY Series** permitted multiple long tracks of any timeline of polyphonic music, drum, and modulation data. While playing the song you've laid out so far, you could **punch in** and record new notes in real time on a given track. These sequencers were known as **linear arrangers**. This approach has since evolved into the ability to record **clips** of polyphonic music, drum, and modulation data, and then copy and lay these clips out however one liked, as shown in Figure 37. These clips can start and end at any time and can overlap with one another: they are no longer restricted to starting and stopping at hard boundaries like patterns. Most modern **digital audio workstation** software employs this approach, including clips of audio recordings in addition to note, drum, and modulation data.

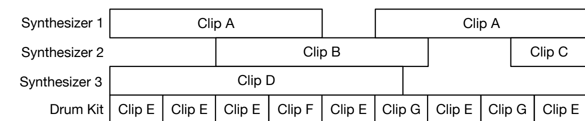



*Figure 37* Clip-style linear arranger sequence.

[27] Indeed, an argument can be made that the style of modern electronic dance music is due in part to its reliance on machines which only had song mode available. That is: the restrictions of early devices have since defined the genre.

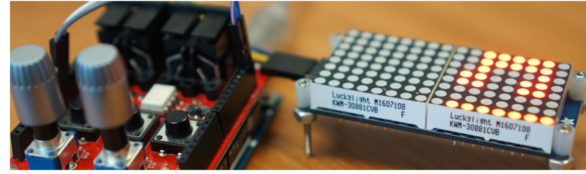


*Figure 38* Gizmo.

**Step Sequencer Example** Consider one such example, **Gizmo**[28] (Figure 38), an Arduino-based device for sending and receiving MIDI. Gizmo has many applications, such as an **arpeggiator** (mentioned in Section 4.4), a note recorder, and a small MIDI control surface (such as discussed in Section 11). But most importantly, Gizmo has two different step sequencers: one for general note and modulation data, and one for drum beats. The step sequencers are both laid out as 2D arrays, where the $X$ dimension is the step number, and the $Y$ dimension is the track.

The general step sequencer supports a song of up to ten patterns. Each pattern can be of up to 96 steps and up to 12 tracks, depending on how you allocate the Arduino's absurdly tiny memory. Each track is a sequence of either notes or modulation data, 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 or ties.

After the musician has entered the relevant data into the step sequencer, it will loop through its steps, and at each step it will emit MIDI information corresponding to all the notes and parameter settings at that step. 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**. Each pattern loops some (different) $N$ times, and then Gizmo proceeds with the next pattern.

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.

Gizmo's drum sequencer is similar, but has more tracks (up to 20) and patterns because drum beats require less data per step than note and modulation information. Each track controls a specific note, notionally associated with a single drum sound on a drum synthesizer. Pattern mode is more elaborate: a song has space for loops of up to 20 patterns drawn from 14 unique patterns. The drum sequencer can then play a sequence of up to 10 looping *songs*.

**Drum Machines** By far the most common use of a step sequencer is to trigger drum sounds, with a different drum sound assigned to each track in the sequencer. A multi-track step sequencer mated to a **drum synthesizer** to produce these sounds is collectively known as a **drum machine** (sometimes called a **drum computer**). If the drum machine also sports other synthesizers or **samplers** (Section 9), and a more general purpose sequencer to support them, it is commonly known as a **groovebox**.

*Figure 39* Roland TR-808 drum machine.©[18]

[28]Why Gizmo? Because I made it of course.

Figure 39 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.

Most drum machines have multi-track, multi-pattern step sequencers with song mode only. Steps might include the volume of the drum strike (that is, whether or not it's struck with an accent or ghost note). Drum machines usually support many of the features discussed so far, including swing,[29] muting, soloing, flams, and so on. They also have additional tricks up their sleeves, such as the ability to make abrupt changes in tempo.

### 4.4 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 looping 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. Ordinarily, you'd 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).

[29] Indeed, swing in a step sequencer was invented by **Roger Linn** for the influential **Linn LM-1 Drum Computer**, which also was the first drum machine to feature drum sounds as **PCM samples**.

### 4.5 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 $\geq 8$ volts) are off=low (0 volts).[30] For example, when a musician presses a 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 $\pm 5$ 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.[31] A sequencer could similarly be configured.

### 4.6 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 more complex, 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 40 reveals how large a proportion the hard-coded modulation options and dedicated envelope parameters were compared to the synthesizer parameters as a whole.

*Figure 40* Screenshots of portions of the **Edisyn** patch editor of the Kawai K4 for (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). The remaining regions were the actual oscillator and filter parameters.

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

[31] This 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.

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**.[32] A typical entry in a 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. Many synthesizers now sport a modulation matrix plus hard-coded modulation routings for the most commonly used routings.

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

*Figure 41* Screenshot of part of the **Edisyn** patch editor for the **Oberheim Matrix 6**, showing (at top) its 10-entry modulation matrix. Each entry has a source, a destination, and a modulation amount. At bottom is the Matrix 6 **tracking generator**, a form of modifier function. The Matrix 6 is discussed in more detail in Section 5.3.

## 4.7 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.

*Figure 42* Novation Remote 25 controller.©19

For example, consider the keyboard in Figure 42. 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 **latency**.

See Section 11.2 for more information about MIDI.

[32]Modulation 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 50 in Section 5.1. 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 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 66) 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.

## 5.1 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 wave, shaping and adjusting it along the way.

Subtractive synthesis has a long and rich history. A good place to start is the **Trautonium** (Figure 43), 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.[33] You could slide up and down

*Figure 43* (Left) The Telefunken Trautonium, 1933.©20 (Right) Oskar Sala's "Mixtur-Trautonium", 1952.©21

[33]The 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 Trautoniums[34] 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, Figure 44) 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 the **Princeton-Columbia Computer Music Center** and was used by a number of avant-garde music composers. The Mark II was highly influential in later approaches to subtractive synthesis: as can be seen from Figure 45, the Mark II's pipeline has many elements that are commonly found in music synthesizers today.

*Figure 44* RCA Mark II.©22

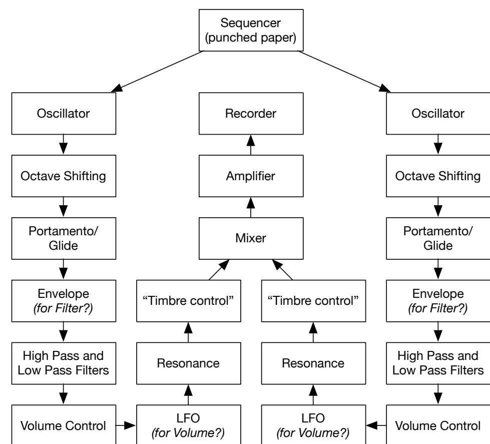



*Figure 45* Mark II pipeline. Compare to modules discussed in this Section, and Figure 66.

**Modular Synthesizers** In 1959 **Harald Bode** started to develop the notion of 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 of one module to the input of 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 cable 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 professional musicians on the East Coast, resulting in devices designed to be musical and (relatively) easily used and programmed. The musician **Keith Emerson** (of **ELP**) is particularly famous for performing on a Moog modular synthesizer (See Figure 46), 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 46* Moog modular synthesizer being played by Keith Emerson.©23

[34]Trautonia? 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 unusual techniques: 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**, Section 6.4), and so on. Buchla also experimented with nonstandard and unusual input and control devices, including the infamous "Multi Dimensional Kinesthetic Input Port", shown at the bottom of Figure 47.[35]

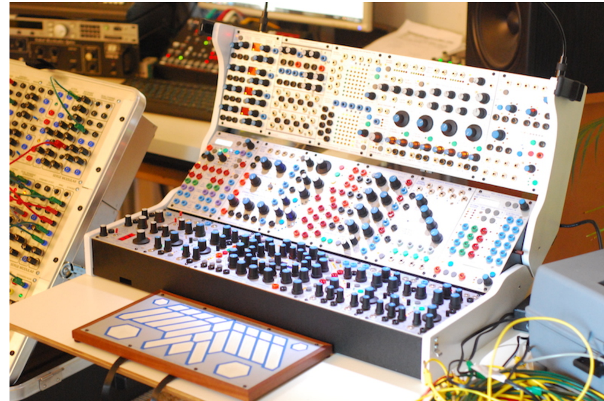


*Figure 47* Buchla 200e modular synthesizer. "Multi Dimensional Kinesthetic Input Port" at bottom.©25

Moog's and Buchla's synthesizers were both very influential, and they formed two schools of synthesizer design, traditionally called **East Coast** (Moog) and **West Coast** (Buchla). The East Coast school, with its more approachable architecture, 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.

**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 ones where the pipeline could be modified by patch cables if desired.

An example of this is the **ARP Instruments 2600** (Figure 48),[36] which could produce a wide range of sounds 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 49).

*Figure 48* ARP 2600.©26

Another approach was to replace the patch cables with a **patch matrix**. In the **Electronic Music Studios (EMS) VCS 3**, shown in Figure 50, the rows of the patch matrix were **sources** and the columns were **destinations**. By placing a pin into a hole, a patch designer could route a source to a destination. Different kinds of pins had different resistance and could vary the magnitude of the signal. The VCS 3 was also semi-modular in that without any pins, it still had a usable default pipeline built-in.[37] Even now many synthesizers have the equivalent of patch matrixes internally in software, known as **modulation matrices** (Section 4.6).[38]

*Figure 49* Korg MS20.©27

[35] **Suzanne Ciani** is an artist famous for using Buchla's unusual methods to their fullest. Google for her.

[36] 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**.

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

[38] 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. Manufacturers started with a standard pipeline and added many hard-coded optional routing options into the synthesizers in the hopes that patching would not be necessary. The quintessential example was the **Moog Minimoog Model D**, widely used by pop and rock performers of the time, and popular even now.

*Figure 50* (Left) The EMS VCS 3.[©28] (Right) A close-up view of the VCS 3 patch matrix.[©29]

The Model D had a very simple pipeline: three oscillators fed into a mixer, then into a low-pass filter and an amplifier. The filter and amplifier each had their own envelope, and the third oscillator could be repurposed as a **low frequency oscillator** for modulation. And that's about it! But this framework, typical of Moog design, proved able to produce a wide range of melodic sounds. The Model D is shown in Figure 51, along with a popular competitor, the **ARP Odyssey**.

*Figure 51* (Left) Moog Minimoog Model D.[©30] (Right) Arp Odyssey.[©31]

**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.

*Figure 52* Hammond Novachord.[©32]

Many early electronic synthesis devices, such as the Telharmonium and the Hammond Organ (Section 3), were polyphonic. But the first major polyphonic (really paraphonic) subtractive synthesizer, and indeed the predecessor to modern polyphonic synthesizers, was the **Hammond Novachord**, circa 1930. Like a subtractive synth, the Novachord had oscillators, filters, and envelopes; but incredibly it could play all 72 notes simultaneously. This was achieved with a technique called **frequency division**: a synthesizer would have twelve high pitched oscillators, one for each note in an octave, and then use a process to *divide down* the appropriate pitch by one or more octaves to play a given note.

Polyphony really didn't come into its own until the 1970s. The **Oberheim 4-Voice** (Figure 53) and **8-Voice** were the first commercially successful polyphonic synthesizers, and were made up of many small monophonic synthesizers developed by **Tom Oberheim**, called his **Synthesizer Expander Modules** or **SEMs**.[39] 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 Figure 53) made it easier to synchronize some programming.

*Figure 53* Oberheim 4-Voice.[©33]

[39]The 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.

The Yamaha CS series, notably the **Yamaha CS-80** (Figure 54), also offered eight voices, and boasted many features for expressive playing. These machines are still legendary (and costly!), as the CS-80 was the synthesizer used by **Vangelis** on his movie soundtracks (notably ***Blade Runner*** and ***Chariots of Fire***) and its unique and easily recognized sound is difficult to replicate.

*Figure 54* Yamaha CS-80.[©34]

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 55).

*Figure 55* Korg PS-3300.[©35]

**Stored Program Synthesizers and MIDI** The integrated circuit (the chip) arrived in the 1980s and with it the ability to store patches in RAM. This major step forward marked (I think) the major division between "old school" synthesizers and modern ones.

First out of the gate was Sequential Circuits's **Prophet 5** (Figure 56). 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.

*Figure 56* Sequential Circuits Prophet 5.[©36]

Following the Prophet 5 came many synthesizers in the same vein. These included the **Oberheim OB-X** (Figure 57) and **OB-Xa** (used by **Van Halen** in the song ***Jump***),[40] the **Moog MemoryMoog** (Figure 58), and Roland's Juno and Jupiter series, most famously its top-of-the-line **Jupiter 8** (Figure 59). These synthesizers were dominant in pop and rock songs through the early 1980s, and found their way onto soundtracks and even orchestral settings.

*Figure 57* Oberheim OB-X.[©37]

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.

**MIDI, Controllers, and Rackmounts** 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, these 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 jointly proposed the **Musical Instrument Digital Interface**, or **MIDI**. MIDI soon caught on, and since then essentially all hardware synthesizers have come with it. MIDI is discussed in Section 11.2.

*Figure 58* Moog MemoryMoog.[©38]

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

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

and without: those without were usually designed to be screwed into a 19-inch rack common in the telephone, electronics, and audio industries. Figure 59 shows the celebrated **Roland Jupiter 8** and its rackmount near-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 and buttons (or both) designed to send control signals to a remote synthesizer over MIDI. Later on, controllers would also be used to send MIDI data to a computer. Controllers are discussed in detail in Section 11.

*Figure 59* Roland Jupiter-8©39 (top) and MKS-80 (bottom).©40

**The Rise of Digital** The early 1980s also saw the birth of the **digital synthesizer**. This wave, starting with **FM synthesizers** (Section 8), and culminating in **samplers**, **wavetable synthesizers**, and **PCM playback synthesizers**, derisively known as **romplers** (all in Section 9), more or less drove analog synthesizers from the market. While some of these synthesizers employed subtractive pipelines similar to analog synths, 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 difficult to program.

*Figure 60* Clavia Nord Lead 2x.©41

In 1995 **Clavia** introduced the **Nord Lead** (Figure 60), a new kind of digital synthesizer. This synthesizer attempted to emulate the characteristics, pipeline, modules, and style of a classic analog subtractive synthesizer, entirely in software 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.

*Figure 61* Korg microKORG.©42

A famous example of this is the **Korg microKORG** (Figure 61). 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.

Virtual analogs are software emulations of synthesizers embedded in hardware: but there is no reason that one couldn't just do software emulation in a PC. Many digital synthesizers now are just 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 62 shows two examples: **OBXD**, an emulation of the **Oberheim OB-X** or **OB-Xa**, and **PG-8X**, a softsynth inspired by Roland's **JX-8P** analog synthesizer.

*Figure 62* OBXD and PG8X software synthesizers.

**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 intangibleness of softsynths, and perhaps partly driven by a distaste for the flood of cheap digital synths with poor interfaces, many musicians have returned to analog devices. Some of these include monophonic and polyphonic analog synthesizers in-line with earlier compact analog designs, such as the **Dave Smith Instruments Prophet 6** (Figure 63), which directly recalls Dave Smith's earlier Sequential design, the **Prophet 5** (Figure 56 on page 52).[41]

*Figure 63* Dave Smith Instruments Prophet 6.©43

*Figure 64* A Eurorack synthesizer with modules from different manufacturers.©44

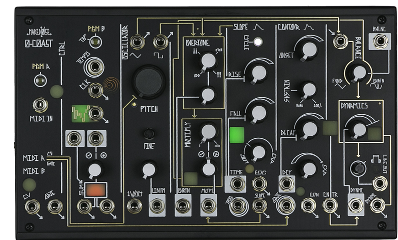


*Figure 65* Make Noise 0-Coast.©45

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 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.[42]

## 5.2 Implementation

The classic subtractive synthesis pipeline, shown in Figure 66, is similar in some sense to the additive pipeline in Figure 25. 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.

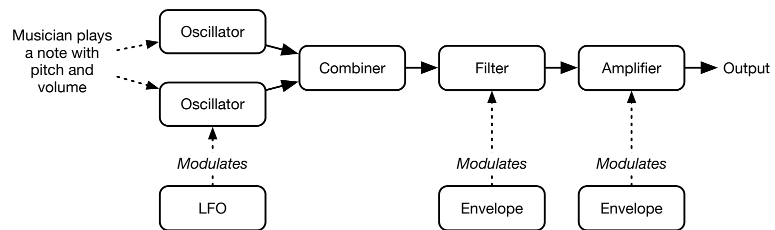



*Figure 66* One possible (and pretty typical) two-oscillator subtractive pipeline, in East Coast tradition. Unlike in additive synthesis, where arrays of partials were passed from module to module, here *sound waves* are passed, one sample at a time (in the digital case).

A large number of subtractive synthesizers follow same general

[41]The Prophet 5 itself has since been reissued.
[42]Hence the name. The "0-" is notionally pronounced "No" as in "No-Coast".

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 6.1 and 6.6 coming up, with some follow-up in Section 9. Filters are discussed in Section 7.

Similarly, the top-level algorithmic architecture for a basic, monophonic subtractive synthesizer is similar to the one used for an additive synthesizer (Algorithm 1). But the additive synthesizer only updated its partials every ticksPerUpdate ticks because the process of updating modules is so costly. Here 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 11** *Simple Monophonic Subtractive Synthesizer Architecture*

1: $M \leftarrow \langle M_1, ..., M_m \rangle$ 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)

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 3.3 had important notes about buffering and latency and how to make the Tick() method consistent in timing for additive synthesizers. That discussion applies here as well.

### 5.3 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 6 and 7 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** (**DCO**s) produced by Oberheim between 1986 and 1988. The Matrix 6 came in three forms: a keyboard (Figure 67), a rack-mount version without a keyboard (the **Matrix 6R**), and a smaller rackmount version designed largely for presets (the **Matrix 1000**, Figure 68). Like many digital synthesizers of the time, and contrary to much 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*[43] from its front panel: all you could do was select from approximately 1000 presets.

*Figure 67* Oberheim Matrix 6.[©46]

*Figure 68* Oberheim Matrix 1000

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 8), which enabled the first oscillator to modulate the cutoff frequency of the filter at audio speeds, creating unusual effects.

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 **LFO**s with many shapes. The amplitude of each LFO could be modulated by a simple envelope Oberheim called a **ramp**, which shifted from 0 to 1 over time. But this was not all. As befitted its name, the Matrix 6 also had a ten-slot **modulation matrix** (Figure 41 on page 47) which could route modulation signals between many 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.

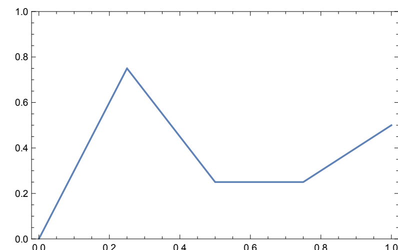



*Figure 69* Five-point Piecewise-linear Tracking Generator.

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 $y$ both 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 nonlinear 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 4.6.

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

**Korg microKORG** Korg appeared relatively early on the virtual analog synthesizer scene with a keyboard synthesizer model called the **Korg MS2000** (Figure 70). 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**.

*Figure 70* Korg MS2000.©47

*Figure 71* Korg microKORG. This is a repeat of Figure 61 on page 53.©48

As shown in Figure 70, the MS2000 was a beast of a machine, while the microKORG (Figure 71) was a tiny little thing with cheap minikeys. Yet these two devices had, more or less, the exact same synthesizer engine built in![44]

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 9). The second oscillator could also be **ring-modulated** by the first oscillator, **synced** to it, or both.

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 an 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 10. 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

[44]This is not entirely unheard of. The entire Casio CZ series, discussed later in Section 6.5, used practically the same synthesis engine in machines ranging from the large CZ-1 (Figure 91) for $1400 to the tiny CZ-101 for $499. The only significant differences lay in the presence of arpeggiators, velocity- and aftertouch-sensitive keybeds, and the hardware interface design. 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).

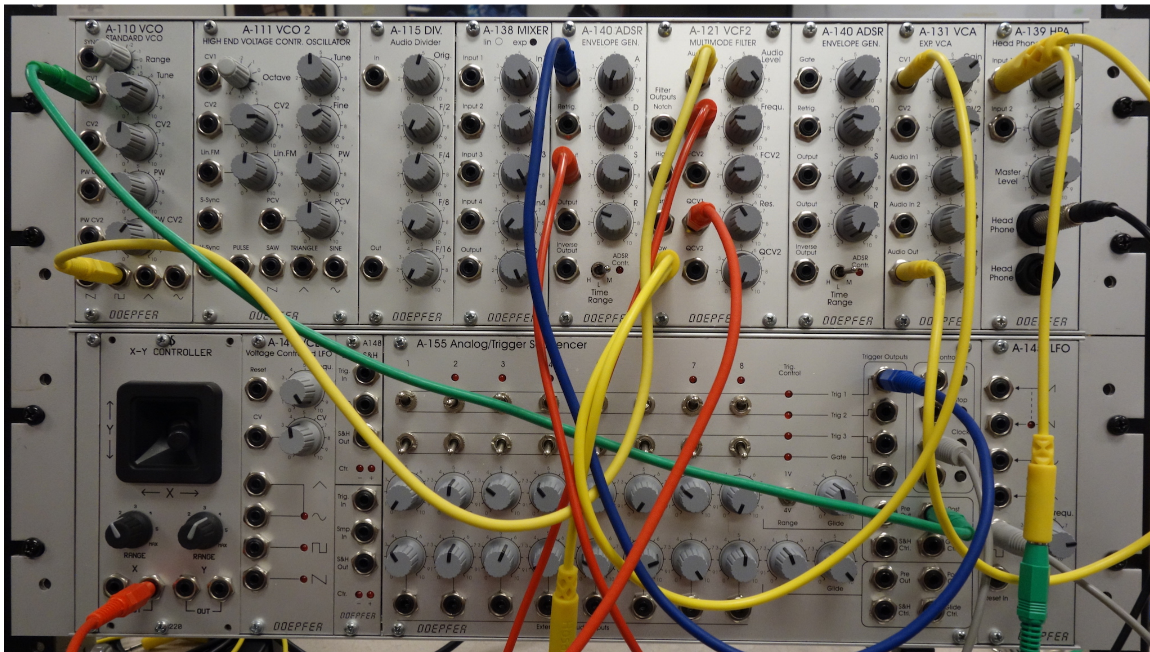



*Figure 72* A small Eurorack-format modular synthesizer, with Doepfer and **Analogue Solutions** modules.

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 72 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** (**VCO**s), 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** (**LFO**s) 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.

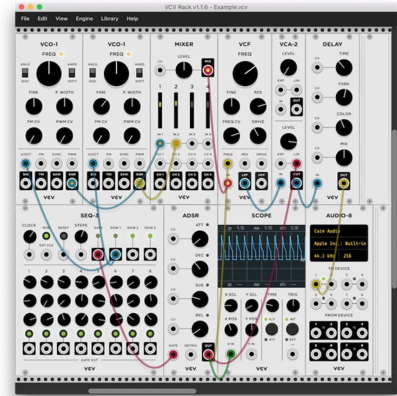


*Figure 73* VCV Rack.©[49]

Many Eurorack style modules have been emulated in a popular software framework called **VCV Rack**.[45] This is an open source program in which you can load modules and attach them in software just as you would in Eurorack hardware. VCV Rack comes with its own modules, but a great many third-party open source and commercial modules have also been made for it, including many direct emulations of popular Eurorack hardware modules.

[45]https://vcvrack.com

# 6 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 4). Filters are a complex topic and will be treated separately in Section 7. In this Section we'll cover the others.

## 6.1 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 analog oscillators took the form of **Voltage Controlled Oscillators** (or **VCO**s), 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 **DCO**s. A DCO is still an analog oscillator, but the frequency of the analog signal is kept in check by a digital timer.[46] On the other hand, **Numerically Controlled Oscillators**, or **NCO**s, 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**,[47] and **square**, shown in Figure 74. These waveforms were fairly easy to produce via analog electronics. They also had a rich assortment of partials, which provided good raw material for downstream filters to modify, and their partials were all **harmonics**, that is, their frequencies were integer multiples of the fundamental partial. This property made them "tonal" or "musical" sounding.

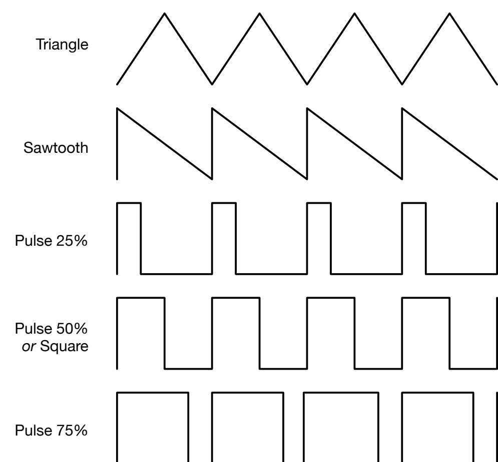
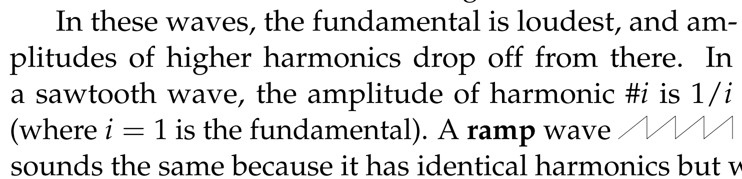
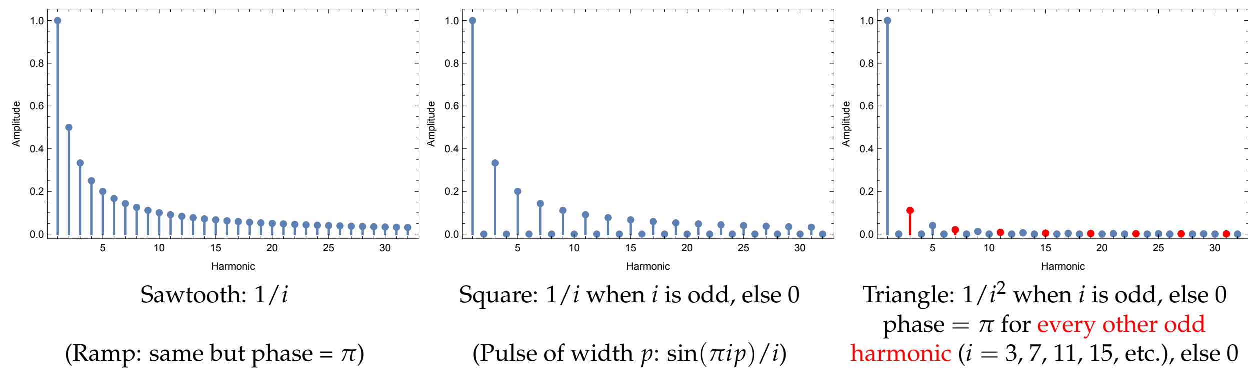



*Figure 74* Triangle, sawtooth, and three pulse waveforms at different pulse widths. The 50% pulse width pulse wave is commonly known as a square wave. Note relationship to Figure 31.

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). A **ramp** wave  sounds the same because it has identical harmonics but with all phases shifted by $\pi$.

[46]DCOs 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.

[47]Perhaps you recall these wave shapes from Section 4.1. And now for some confusion! The term *sawtooth* is often abused to describe *both* sawtooth and ramp waves. Indeed analog synthesizers generally implement a sawtooth sound using a ramp wave. This distinction matters for LFOs, but not for oscillators producing audio-frequency waves, as ramp and sawtooth have identical harmonics except for opposite phase: they'll sound the same.




Sawtooth: $1/i$

(Ramp: same but phase = $\pi$)

Square: $1/i$ when $i$ is odd, else 0

(Pulse of width $p$: $\sin(\pi i p)/i$)

Triangle: $1/i^2$ when $i$ is odd, else 0
phase $= \pi$ for every other odd harmonic ($i = 3, 7, 11, 15$, etc.), else 0

*Figure 75* Harmonics of the sawtooth, square, and triangle waveforms. $i$ is an integer $\geq 1$

In a square wave, the amplitude of harmonic #$i$ is $1/i$ when $i$ is odd, but 0 when even: this is quieter than a sawtooth wave. A square wave is just a special case of a **pulse** wave. As shown in Figure 74, 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%.[48]

A triangle wave is like square wave, except that the amplitude of harmonic #$i$ is $1/i^2$, and the phase of every second non-zero amplitude harmonic is shifted by $\pi$.[49] This squared dropoff in amplitude means that the triangle wave is quieter than a square wave and *much* quieter than a sawtooth wave.

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.[50]

**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 of

[48]Generally speaking, the amplitude of harmonic #$i$ in a pulse wave of pulse width $p$ is $\sin(\pi i p)/i$. You might ask yourself what happens to this equation when the pulse width is 0% or 100% ($p = 0.0$ or $1.0$).

[49]Recall that humans distinguish phase poorly, so this fact is of lesser importance.

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

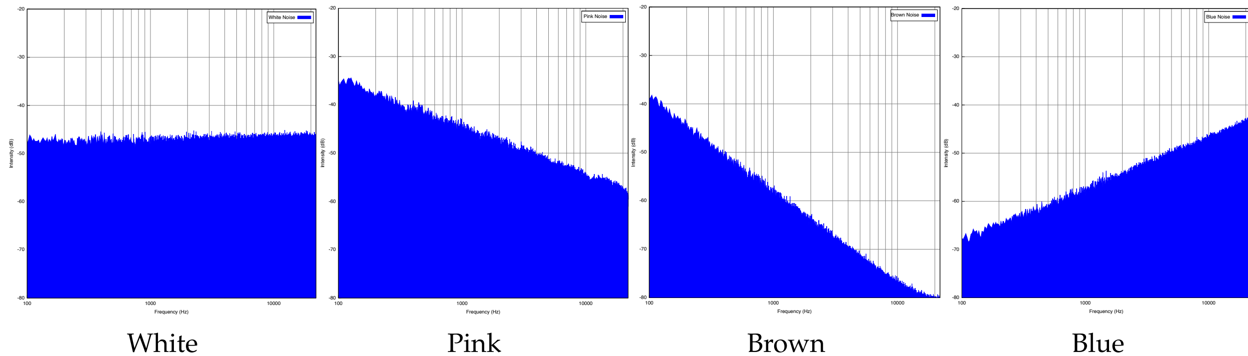



*Figure 76* Frequency spectra plots of four common kinds of noise: (Left to right) White, Pink, Brown, and Blue noise. Note that the plots are logarithmic in both directions.[©50]

its partial spectra across all frequencies.[51] 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 $-1... +1$ 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 7.

**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 — its 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.

[51] White noise is trivial, as you can see below. But pink noise pushes white noise through a 3dB low-pass filter, which is a bit difficult to make well. Here's a quick-and-dirty but not super accurate one originally from Paul Kellet: see http://www.firstpr.com.au/dsp/pink-noise/ for this and better examples. It generates random white noise and then applies an **FIR low pass filter** to it. More on filters in Section 7.

**Algorithm 12** *White Noise Sample*

1: **return** random real-valued number from $-1$ to $+1$ inclusive ▷ Really Complicated!

**Algorithm 13** *Pink Noise Sample*

1: Global $b_0, b_1, b_2 \leftarrow$ initially 1.0

2: $w \leftarrow$ White Noise Sample
3: $b_0 \leftarrow 0.99765 \times b_0 + 0.0990460 \times w$
4: $b_1 \leftarrow 0.96300 \times b_1 + 0.2965164 \times w$
5: $b_2 \leftarrow 0.57000 \times b_2 + 1.0526913 \times w$
6: **return** $b_0 + b_1 + b_2 + 0.1848 \times w$

## 6.2 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 many 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 77 at right. Notice that a wave consisting of just the first 20 partials in a sawtooth wave is cruder looking than one generated with the first 100 partials (or first 100,000!).

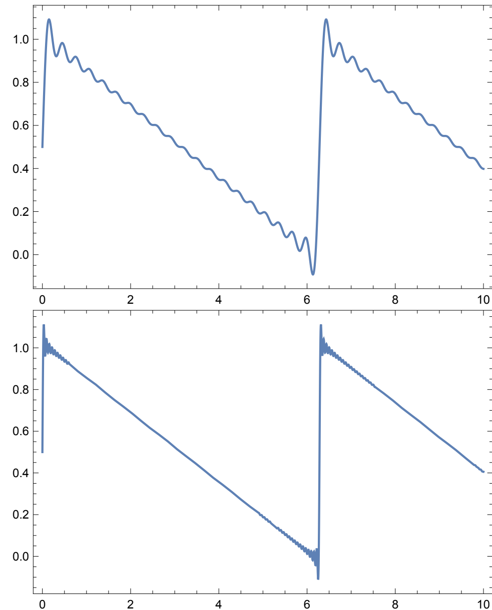


*Figure 77* Sawtooth wave approximations consisting of only (top) the first 20 partials or (bottom) the first 100 partials.

We have to stop at Nyquist because any higher frequency partials we try to include in the signal get "reflected" away from the Nyquist limit.[52] 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 78 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**.

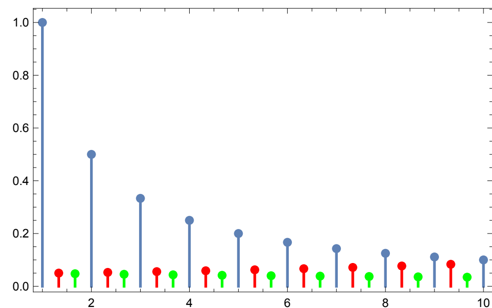


*Figure 78* Partials of an Aliased Sawtooth wave. The sequence of sawtooth partials (Blue) reflects off the Nyquist limit (here 10 ⅓) and continues backward (Red), then bounces *again* off of 0 and continues forward again (Green), and so on. Only the first two bounces shown.

The unfortunate audio effect of aliasing is hard to explain in 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 79 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:

[52] This is why aliasing is sometimes called **foldover**: the reflected partials are "folded over" the Nyquist limit.

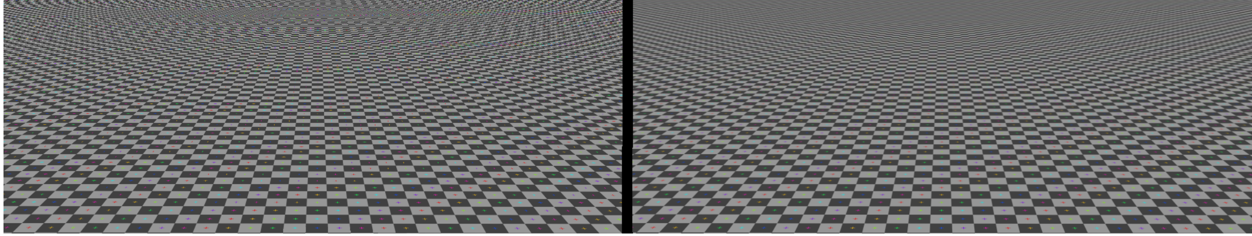


*Figure 79* Moiré patterns (left) in an image. This effect is due to the 2D version of aliasing.©51

**Additive Synthesis** We could build the wave by adding sine waves up to the Nyquist limit.

**Resampling** This is the most common approach. We create a band limited wave at a high sampling rate, usually with additive synthesis, and store a single cycle of it as a digital sample: this is a **single-cycle wave**. When we need to convert the wave to a certain frequency, this is equivalent to **resampling** it into a different number of samples. The process of resampling is discussed in Sections 9.5 through 9.7, and an algorithm for resampling a single-cycle wave is given in Section 9.8 (Algorithm 22, "Looped Single-Cycle Wave Playback").

**Discrete Summation Formulae (DSF)** 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.[53] 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. I'd set $\theta = 2\pi t f$, and set $\beta = 2\pi t f m$, where $f$ is the fundamental frequency and $t$ is the current timestep. The value $m$ scales the partials relative to the fundamental frequency: if $m$ is a positive integer, you'll have harmonics, else you'll get inharmonic partials.

Let's set $m = 1$, so you have harmonics. We'd still 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 $a$ one could get a very bad approximation ($a = 0.55$ to $0.75$ seem useful). 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 partials above Nyquist: but we can create a version of it with the high frequency partials stripped out. This is called a **band limited impulse train**, or **BLIT**.[54]

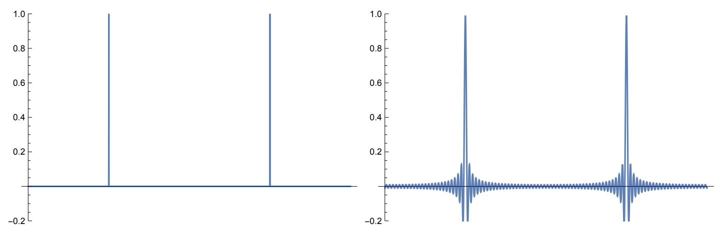



*Figure 80* (Left) A Delta or Impulse train. (Right) An equivalent BLIT, band limited to quite low frequencies for illustrative purposes.

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

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

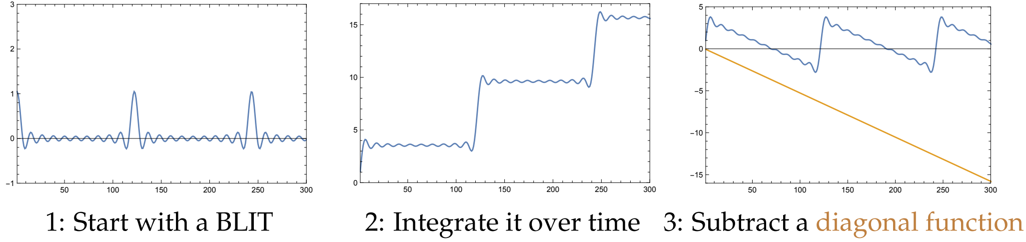



*Figure 81* Creating a band limited sawtooth. This is done by integrating or summing the BLIT starting at 0 to create a stairstep, then subtracting a diagonal function from the stairstep to pull it back down every period.

A BLIT is defined as:

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

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

What can we do with this? Well, a band-limited sawtooth for one. The idea, as shown in Figure 81, is to integrate the BLIT over time, which creates a stairstep function. From this, we can subtract a constant linear function to pull it back down to 0 every period and create a sawtooth.

We can generate this on-the-fly for every new sample $x$ by taking our previous value at $x - 1$ and adding the current blit value to it (that's the integration part) then subtracting $1/P$ (that's the linear function subtraction). The result looks like this:

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

This starts out very noisy but cleans up after about 6 cycles or so. The thing that cleans it up is the $\alpha$ bit. This is a **leaky integrator**: it causes the algorithm to gradually forget its previous (initially noisy) summation.[55] I find this is reasonably scaled to 0...1 as saw(...) $\times\, 0.8 + 0.47$.

To do a square wave, we need a new kind of BLIT, where the pulses alternate up and down. This is a **BPBLIT** (for *bipolar* BLIT):

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

Here $D$ (which ranges from 0 to 1, 1/2 being the 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.

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

I find this is reasonably scaled to 0...1 as $(\text{square}(...) + D) \times 0.7 + 0.15$.

[55]The 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.

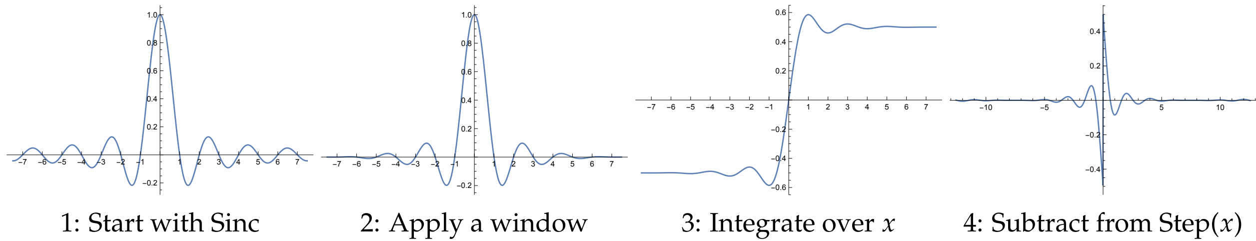



*Figure 82* Producing a BLEP. We begin (1) with the sinc function, then (2) window it, then (3) take the integral with regard to $x$, and then finally (4) subtract the result from Step($x$), which is $-0.5$ if $x < 0$, and $+0.5$ otherwise. Windowing is a procedure which forces a function to go to zero beyond a finite interval (+/-7 in this example). More on sinc and windowing in Section 9.7, where they're used heavily.

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 $\alpha$, but it makes crummy triangles at low frequencies.

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

I'd scale this as tri(...) $\times\, 4 + 0.5$. 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.

**Band Limited Step Functions** Instead of building a wave carefully out of band-limited components, wouldn't it be easier to just draw the wave and then tweak it? You can't apply a filter per se, as the problematic high-frequency partials have already been aliased and added into the wave. But there's a trick you can apply, if you recognize that most of these aliased partials are due to the sharp discontinuities (corners) in the wave. Perhaps if you shaved off those corners you could decrease the aliasing significantly. That's the idea behind the **band limited step function** or **BLEP**.[56]

You may have noticed that, when band-limited, sharp vertical discontinuities have little "wiggles" before and after them. You can see this in Sawtooth (look back to Figure 77), and Square waves (Figure 83 at right). These wiggles happen to be derived from the **sinc function**,[57] $\text{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & x \neq 0 \\ 1 & x = 0 \end{cases}$ We won't go into detail about sinc here: it's discussed at length in Section 9.7.

The idea is to reduce aliasing by strategically introducing these wiggles both immediately before and after each discontinuity. This introduced wiggle is called a BLEP.

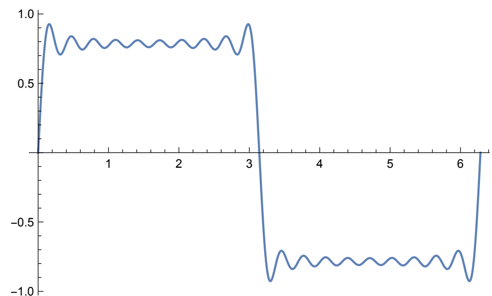


*Figure 83* Square wave approximation consisting of only the first 20 partials.

[56] No, the acronym makes no sense to me either. BLEPs were introduced in Eli Brandt, 2001, Hard sync without aliasing, in *International Computer Music Conference*.

[57] Yes, sinc is related to $\text{sinc}_M$ in BLITs. That's why BLITs produce band-limited square, sawtooth, and triangle waves.

Before we drop BLEPs into our sound wave, we must first generate and store one, as shown in Figure 82. A BLEP is just the residual left over when we take the integral of the sinc function (the wiggles) to produce a band-limited step function, and then subtract it from a sharp (non-band limited) step, that is, a sudden discontinuity. If you added this back to a sharp discontinuity in your sound, you'd get the band-limited version again. The idea is to build a BLEP, store it at high resolution, and then, for every major discontinuity in your hand-drawn sawtooth or square wave, add a BLEP (or its reverse) to the wave centered at that discontinuity.

A simpler alternative to building and pre-storing a high resolution BLEP is to roughly approximate it with a polynomial that's easily calculated on the fly. This approach is known as a **PolyBLEP**. As (just) one example, you might use a third-order piecewise Lagrange polynomial approximation, shown in Figure 84, which as follows:

$$\begin{array}{ll} (x+2)^4/24 - (x+2)^2/12 & -2 \leq x < -1 \\ -(x+1)^4/8 + (x+1)^3/6 + (x+1)^2/2 - 1/24 & -1 \leq x < 0 \\ x^4/8 - x^3/3 - x^2/4 + x - 1/2 & 0 \leq x < 1 \\ -(x-1)^4/24 + (x-1)^3/6 - (x-1)^2/6 + 1/24 & 1 \leq x \leq 2 \end{array}$$

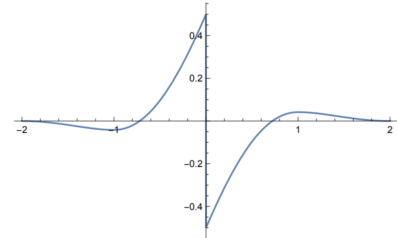


*Figure 84* Very simple BLEP approximated with the piecewise polynomial at left. Note this is flipped from the example in Figure 82 (4), sorry.

This function is scaled so that each sample is length 1. Thus this approximation doesn't go out very far: just 2 samples away from the discontinuity in each direction. This isn't very costly to add, but surprisingly is sufficient to reduce aliasing by a fair amount.[58]

### 6.3 Wave Shaping

Once we have a basic sound wave, we can then mutate it into something more interesting. The first mutation method we'll discuss is **wave shaping**.

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

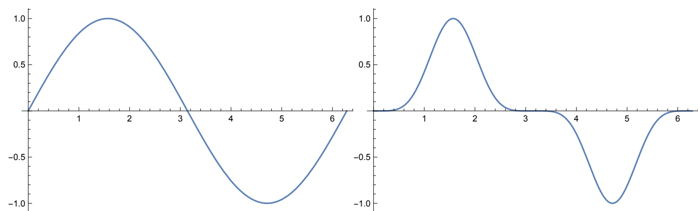


*Figure 85* A Sine wave $\sin(t)$, and the result of waveshaping with the function $x^5$, producing $\sin^5(t)$.

Because wave shapers can have *arbitrary* functions $f(x)$, 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 $f(x)$ for every possible input value $x$.

[58] For more on this approximation, and a number of other PolyBLEP approximations, see Vesa Välimäki and Jussi Pekonen, 2012, Perceptually informed synthesis of bandlimited classical waveforms using integrated polynomial interpolation, *Journal of the Acoustical Society of America*. Indeed, this example is taken directly from that paper. Note that the authors also suggest post-processing the sound with a simple **high pass filter** because the polynomial approximation tends to dampen certain (valid) high frequencies. They give the specific coefficients for an **FIR** filter which you can build straightforwardly after reading Section 7.

**Waveshaping Polynomials** Keeping aliasing in mind, you could in theory use any function you wished to shape an incoming signal: but it's common to use polynomials. This is because polynomials allow us to predict and control the resulting partials in a sound. In particular, a polynomial of order $n$ can only generate harmonics up to $n$. Consider the polynomial $x^5$ applied to a sine wave of frequency $\omega$ and amplitude 1, as shown in Figure 85. Our waveshaped signal $w(t)$ would be:

$$\begin{aligned} w(t) &= \sin^5(\omega t) \\ &= 1/16 \times (10 \sin(\omega t) - 5 \sin(3\omega t) + \sin(5\omega t)) \qquad \textit{Trust me, there's a } \sin^5(\theta) \textit{ identity} \end{aligned}$$

(Don't confuse the $w$ and $\omega$) This would create a harmonic at $\omega$, a second at $3\omega$, and a third at $5\omega$, 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.

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 eight polynomials are:

$$\begin{aligned} T_0(x) &= 1 & T_4(x) &= 8x^4 - 8x^2 + 1 \\ T_1(x) &= x & T_5(x) &= 16x^5 - 20x^3 + 5x \\ T_2(x) &= 2x^2 - 1 & T_6(x) &= 32x^6 - 48x^4 + 18x^2 - 1 \\ T_3(x) &= 4x^3 - 3x & T_7(x) &= 64x^7 - 112x^5 + 56x^3 - 7x \end{aligned}$$

## 6.4 Wave Folding

Related to wave shaping is **wave folding**, popularized 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 reflects it 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 go beyond -1.0 (or 1.0), and so it will be *folded again*, and again, until it's inside the range $-1...1$. Thus you can modulate the effect by amplifying the incoming wave before shaping it with a wavefolder (see Section 6.7 coming up). The basic folding equation is recursive, but easy.

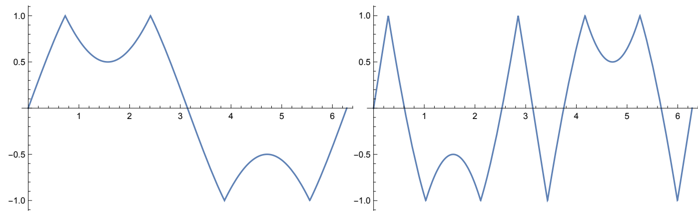



*Figure 86* Wave folding the function $\text{Fold}(1.5 \sin(t))$ (left) and the $\text{Fold}(3.5 \sin(t))$ (right). The right figure gives an example of recursive folding.

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

This particular wave folding equation, as shown in Figure 86, introduces a great deal of harmonic complexity into a sound, including (typically) a lot of inharmonic partials. Some wave folders attempt to produce a more rounded — and less aliased — shape. To do this, first realize that though it's defined recursively here, Fold($x$) is in fact just wave-shaping with the kind of Triangle wave shown in Figure 87.[59]

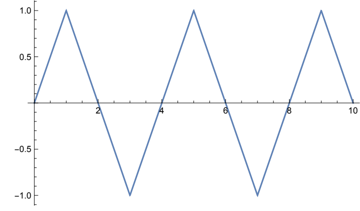



*Figure 87* The Fold function is really just a Triangle function.

Now consider using a different, softer oscillating function to do the waveshaping, such as Sine. If you defined Fold($x$)=Sin($2\pi x$),[60] you'd get the rounded effect shown in Figure 88. It still wouldn't do much for sharp discontinuities (such as folding a Sawtooth or Square wave).

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:

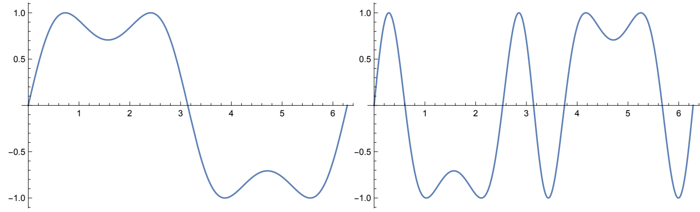



*Figure 88* Wave folding the function $1.5\sin(t)$ (left) and the function $3.5\sin(t)$ (right) using a sine wave, that is, $\sin(2\pi \times 1.5\sin(t))$ and $\sin(2\pi \times 3.5\sin(t))$ respectively. Compare to Figure 86.

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

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 - (int)u` ($u$ is floating point).

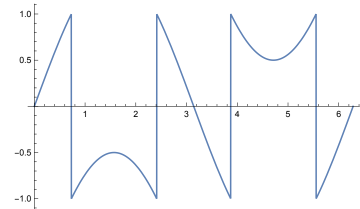



*Figure 89* Wrapping the function $1.5\sin(t)$.

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

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

**Dealing with Aliasing** It shouldn't surprise you that these methods can alias like crazy due to the hard discontinuities that occur when these waves hit the 1.0 or -1.0 boundary. To deal with this you'll need to develop soft versions of these functions (such as was done with wave folding), or perhaps add **BLEP**s (Section 6.2) after the fact.[61]

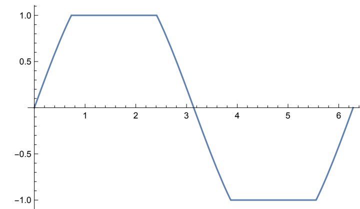



*Figure 90* Clipping the function $1.5\sin(t)$.

[59] That is, Fold($x$)= $\text{Tri}(x) = \begin{cases} M(x) - 1 & M(x) < 2 \\ 4 - M(x) - 1 & \text{otherwise} \end{cases}$ where $M(x) = (x+1) \bmod 4$.

[60] If you think pushing a Sine wave through a Sine wave feels a bit like FM synthesis (Section 8), you get a gold star.

[61] For 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.

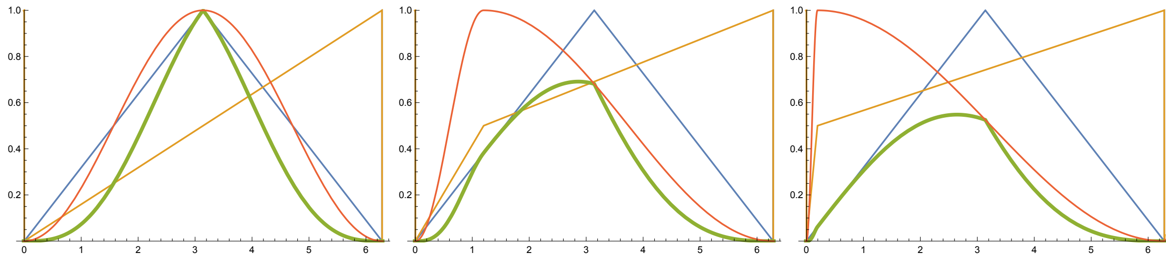



*Figure 92* Phase distortion using $g(x, \alpha) = \frac{x}{\alpha}$ when $x < \pi\alpha$, else $\frac{x - \pi\alpha}{2-\alpha} + \pi$. This $g(...)$ phase-distortion function is initially the identity function at $\alpha = 0$, then develops a knee at $\pi\alpha$ as $x$ increases. This results in the 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.[62]

## 6.5 Phase Distortion

*Figure 91* Casio CZ-1.©52

**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 8 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 that $\lim_{x \to 0} g(x) = 0 \mod 2\pi$, 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 $2\pi$ 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 $2\pi$.

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 92 and 93. Phase distortion then outputs the waveform $\text{PhaseDistort}(t)$ using the following equation:

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

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.[63] How $g(...)$ maps the distortion varies from function to function. The CZ series provided a range of $g(...)$ functions and $w(...)$ window options.[64]

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

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

[64]Casio 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 (discussed in Section 11.2.2). See http://www.kasploosh.com/projects/CZ/11800-spelunking/ for an explanation of how to do this.

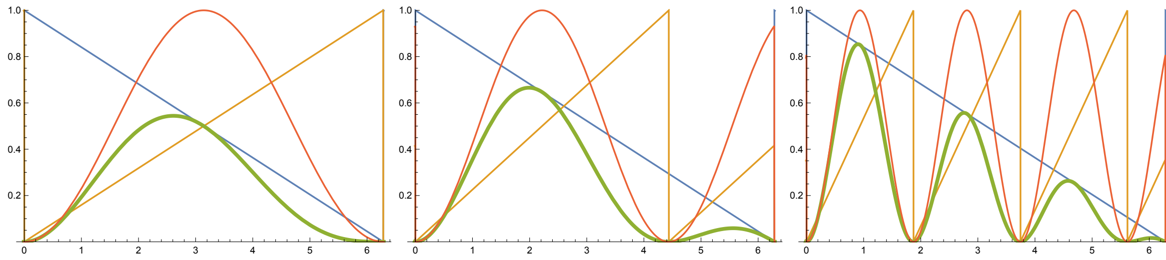



*Figure 93* Phase distortion using $g(x, \alpha) = \frac{x \mod 2\pi\alpha}{\alpha}$. This $g(...)$ phase-distortion function is again initially the identity function at $\alpha = 0$, then increases in frequency $\pi$ as $\alpha$ increases, but resets at $2\pi$. This results in the cosine wave increasing in frequency but likewise resetting at $2\pi$. 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.[65]

Figure 92 shows a sinusoid being gradually distorted by a $g(...)$, changing according to $\alpha$. 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 7. Phase distortion's quasi-filters in combination with windowing can also add **resonance** to the wave, as shown in another example, Figure 93.[66]

## 6.6 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 filters and amplifiers. There are many different ways that these oscillators could be combined: we'll cover a few of them here.

**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 $f_1(t)...f_n(t)$, each with a weight $\alpha_1...\alpha_n$ (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 $\alpha$ 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 $m(t) = \frac{\sum_i \alpha_i f_i(t)}{\sum_i \alpha_i}$.

**Vector Synthesis** Let's consider a different direction to expand on the cross-fade mechanism. What if we had four sounds, and paired sounds 1 and 2 with a cross-fader, then paired sounds 3 and 4 with a cross-fader, and finally added the two cross-faded results? That is, we'd do $m(t) = \alpha f_1(t) + (1 - \alpha) f_2(t) + \beta f_3(t) + (1 - \beta) f_4(t)$. Thus we'd have two parameters, $\alpha$ and $\beta$, and so we'd now have two-dimensional space of linear mixtures among the four sounds.[67] We might then define a function that wanders through the $\langle \alpha, \beta \rangle$ space over time, creating a sound which morphs and evolves. We could define this movement of $\alpha$ and $\beta$ independently using envelopes or

[65] This 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.

[66] I do not know what tricks Casio employed, if any, to reduce or eliminate aliasing from Phase Distortion.

[67] The four sounds here are only linked pairwise, not fully. But in fact it is possible to graphically link *three* sounds together. Place each sound $i$ at a point (corner) $p_i$ of an equilateral triangle (a **simplex**). Let $x_i$ be the distance from a control point within the triangle to the edge opposite point $p_i$. Then the fraction of sound $i$ in the total sound is $\frac{x_i}{\sum_j x_j}$.

LFOs, or we could define a point-by-point **trajectory** over time. This approach, known as **vector synthesis**, was pioneered by the **Sequential Circuits Prophet VS**.

The VS sported a joystick with which the musician could control $\alpha$ and $\beta$ during real-time performance, or to program a real-time trajectory. The VS was famous for producing swirling, complex sounds.[68] Of course you could do vector synthesis with more than just four sounds, and thus more than just two parameters: but then you'd not be able to control them all with a single joystick, and that wouldn't be as fun.

**Detuning** By far the most common trick to fatten up a sound is to take two or more oscillators at the same pitch and **detune** one of them relative to the other, then mix the two together. This has an interesting psychoacoustic effect: it creates a rich, thicker sounding tone. To detune an oscillator means to shift its pitch by a very small amount: somewhere in the 3–10 cents range is common. See Algorithm 19, page 130 ("Pitch Shifting for One-Shot Playback") to pitch-shift.

Detuning also causes **beating**, a pattern where the sound is iteratively louder, then quieter. This is because the frequency of one oscillator is slightly higher than the other. The two start in phase (and so when mixed their volumes double when added), but quickly one oscillator outpaces the other to the point that their phases are opposite one another (and thus cancel each other out in volume). The outpacing continues until the faster oscillator laps the slower one and they get back in phase again. When the frequency difference is large, such as two oscillators played as two notes in a chord, this lapping is so fast that we don't notice any beating. But when the difference is very small, as happens with detuning, the beating is quite prominent. The smaller the detuning, the longer the laps take and so the slower the beating.

**Ring and Amplitude Modulation** Whereas in mixing, the two sound sources were essentially added, in **ring modulation**,[69] the two sources $f(t)$ and $g(t)$ are *multiplied* against one another, producing the sound:

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

Ring modulation is so named because of how it is commonly implemented in circuitry: using a ring of diodes as shown in Figure 94.

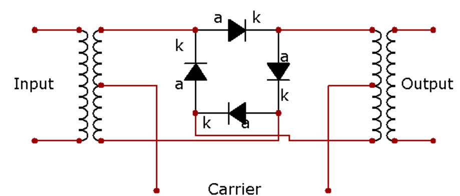



*Figure 94* A ring modulation circuit.©53

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

$$\begin{aligned} a(t) &= f(t) \times (g(t) + 1) \\ &= f(t) \times g(t) + f(t) \\ &= r(t) + f(t) \end{aligned}$$

[68]Vector synthesis is often compared to **wavetable synthesis** (Section 9.3), as they both produce complex changing sounds. Each has their advantages: wavetable synthesis is technically more powerful (vector synthesis can only make linear combinations of its various sound sources), but it's easier, and more fun, to program complex sounds using a vector synthesis joystick! The Prophet VS is often mistakenly described as a wavetable synthesizer, but it is not. Vector and wavetable synthesis are also both connected to yet *another* combination approach, the **wave sequence**, pioneered by a later vector synthesizer, the **Korg Wavestation** (see Section 9.2).

[69]This isn't exactly "modulation" in sense discussed in Section 4.

Thus amplitude modulation is simply ring modulation with one of the original signals mixed in again. Let's consider the effect of ring and amplitude modulation in a simple case, using sine waves as our sound sources, with $a_f$ and $a_g$ as amplitudes and $\omega_f$ and $\omega_g$ 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 be:

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

Recall that the original signals were sine waves, and thus had one partial each at frequencies $\omega_f$ and $\omega_g$ respectively. The new combined sound also consists of two partials (the two cosines), 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 $|\omega_f - \omega_g|$. Similarly, if $\omega_f + \omega_g$ is greater than Nyquist, it's "reflected" back off of Nyquist as was done with aliased frequencies. We'll see a lot more on sidebands and reflection in Section 8.

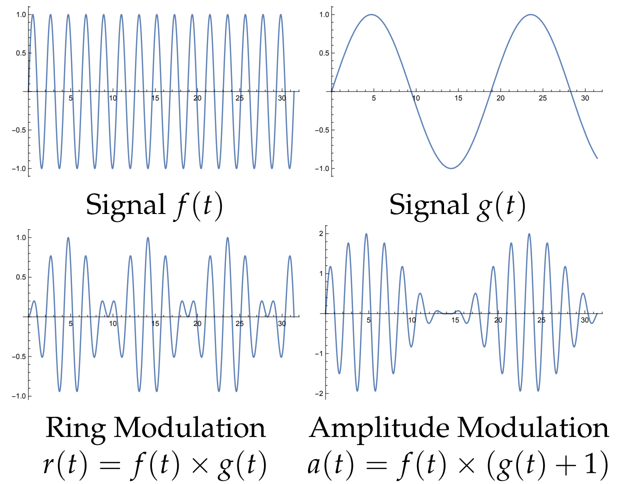



*Figure 95* The effects of ring and amplitude modulation when given two sine wave signals, $f(t)$ and $g(t)$.

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 in ring modulation, plus the original carrier partial $\omega_f$. This shouldn't be at all surprising given the relationship between the two as discussed earlier.

---

[*] This transformation comes from the cosine identities

$$\cos(A + B) = \cos A \cos B - \sin A \sin B \qquad \textit{and} \qquad \cos(A - B) = \cos A \cos B + \sin A \sin B$$

Subtract these two and we get...

$$\begin{aligned} \cos(A - B) - \cos(A + B) &= (\cos A \cos B + \sin A \sin B) - (\cos A \cos B - \sin A \sin B) \\ &= 2 \sin A \sin B \\ \sin A \sin B &= 1/2\, (\cos(A - B) - \cos(A + B)) \end{aligned}$$

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.[70]

**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.[71] Frequency modulation is an important subject in and of itself, and has spawned an entire family of synthesis methods all on its own. It'll be discussed in detail in Section 8.

**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 96 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.[72]

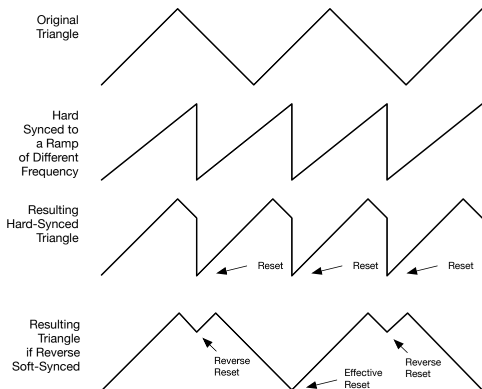



*Figure 96* Hard and Reversing Soft Sync. A Triangle wave is hard- and reverse-soft-synced to a Sawtooth wave of a different frequency.

**Architectures for Combination** Many modern synths have a fixed architecture for combination. For example, we might have two oscillators, one of which can be detuned or hard-synced to the other, and then the two are the mixed along with the ring-modulation of the two, and also noise. However some synthesizers have taken a more flexible approach to combination hearkening back to classic modular synths. For example, the **Ashun Sound Machines (ASM) Hydrasynth** pushes each of two oscillators through a chain of two modules each (ASM calls the four modules *mutators*). Each module modifies the incoming sound in of several ways of your choice, such as doing hard-sync or FM against the output of any other oscillator or mutator; or adding together five detuned copies of the wave, etc. The output of the second mutator for each of the two oscillators, plus the output of a third oscillator, ring mod, and noise, are then fed into a mixer.

[70]If you have to choose between implementing ring modulation or AM, pick ring modulation. The musician could always mix the carrier sound back into the result; but the converse is not true, that is, it'd be harder to *remove* the carrier from AM to produce ring modulation. The **Kawai K1** synthesizer had ring modulation, and the later **Kawai K4** foolishly replaced it with AM. While the K1 is quite primitive compared to the K4, the K1 has a cult following the K4 lacks simply because of its ring modulation.

[71]Many early synthesizers sported so-called **cross modulation** between two oscillators. This generic term was very often some form of FM, though it occasionally referred to other kinds of modulation.

[72]Sync will introduce high degrees of aliasing: **BLEP**s (Section 6.2) seem the right way to deal with this. Indeed BLEPs were themselves introduced in a paper which specifically attempted to deal with aliasing in hard sync (see Footnote 56).

## 6.7 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 or to invert the signal. Analog amplifiers are classically controlled via a voltage level, and so they are often known as **voltage controlled amplifiers** or **VCA**s.

If an amplifier is so boring, why is it such an important 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 11.1. 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.

# 7 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 easily the most complex elements 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 each partial depends on the partial's 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** (or **magnitude frequency response**) and the **phase response** of the filter. Because humans can't hear differences in phase very well, we're usually more interested in the amplitude response and will focus on it here; but there are important uses for the phase response which we will come to later starting in Section 10.4.

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 above 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 97 (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 97 (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, and particularly if they're far from it: thus it's "picking out" that frequency and shutting off the others. See Figure 97 (C).[73]

- A **notch** filter is the opposite of a band pass filter: it decreases the amplitude of partials if they're *near* the cutoff frequency. See Figure 97 (D).[74]

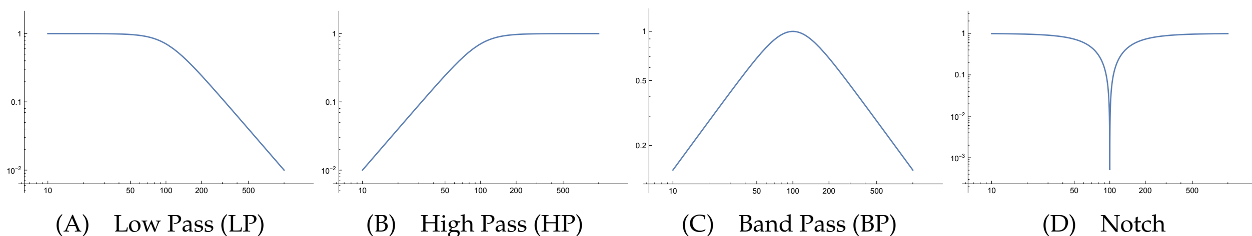



*Figure 97* Amplitude response by frequency of four common filters, with a cutoff at 100. The axes are on a log scale. Thus the notch is really more or less an inverted band pass, but looks quite different because of the log scaling.

[73]This 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.

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

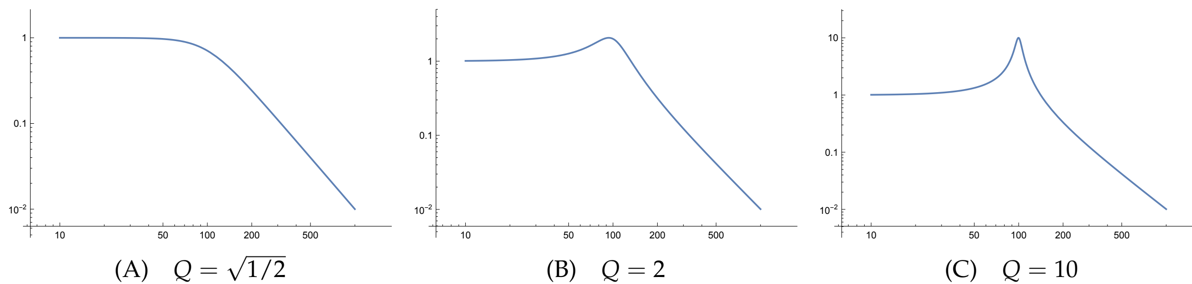



*Figure 98* Low Pass filter with a cutoff at 100 and varying amounts of resonance ($Q$). Note that axes are on a log scale.

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 10. And another important filter is the notional **brick wall** low pass filter: essentially a very, very steep low pass often used to cut out frequencies above Nyquist.

**Phase** The filters above are largely differentiated by how they modify the amplitude of the partials: but it doesn't say anything about how they modify the partial's phase. In fact, usually when building the four filters above our goal would be to not modify the phase at all,[75] 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. 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 in various ways. We'll see all pass filters more in Section 10.

**Gain** In modifying the amplitude or phase of certain partials, 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, Resonance, and Tracking** 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 later, but for now it's helpful to know two facts. First, the number of poles can determine how steep the filter drops off: 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 is normally described as **four pole** and will have a roll-off of 24dB per octave. These are illustrated in Figure 99. Most filters in audio synthesizers are second order ("2 pole", "12dB") or fourth order ("4 pole", "24dB").

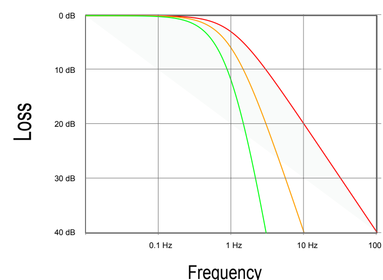



*Figure 99* Roll-off of typical low pass filters: one-pole (6dB), two-pole (12dB), and four-pole (24dB).©54

[75] If 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 98, 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. Some filters are capable of **self-resonance**, a where the filter, when set to extreme resonance, can produce a tone all on its own without an incoming audio signal.

Last, filters are sometimes outfitted with **keyboard tracking**, where the cutoff frequency of the filter moves to some degree with the note being played. Thus high notes might have a higher cutoff frequency than low notes and thus sound brighter. This can help the synthesizer sound more "natural", as many real instruments tend to follow different brightness profiles depending on pitch.

**Filter Design Goals and Tradeoffs** The goal of a filter is to modify the characteristics (amplitude, phase) of partials at certain frequencies while not overly affecting partials elsewhere. This is not easily achieved, and filter design is largely about compromise. Consider the low-pass filter in Figure 100. Here we want to decrease the amplitude of partials in the **stopband** but not affect the amplitude of partials in the **passband**. The roll-off must largely occur in the **transition band**, but if the transition band has to be very narrow (the roll-off has to be steep), we may be forced to accept **ripple** in the stopband or passband. Further, the filter may produce an undesirable shift in phase in some partials, that is, a bad **phase response**. These aren't things you want in an audio filter: humans ears are sensitive.

There are issues in the time domain as well: for example as shown in Figure 101, filters also may be overly slow to respond to abrupt changes in the sound, may **overshoot** the goal amplitude, and may oscillate considerably before settling down (so-called **ringing**).

Figure 102 shows a few classic filters which illustrate some tradeoffs. Notice that Butterworth filters have no ripple (which is good for an audio filter): but have a very gradual roll-off. We can get a high-order Butterworth filter to have a steep roll-off, but at the cost of ringing and a considerable shift in phase (which are probably not good).

The higher-order the filter, the more parameters we have available to achieve our design goals. But a higher-order filter is more complex and its larger parameter space presents a bigger design challenge. One common approach is to build higher-order filters by **composing** lower-order filters, that is, stacking them in series or in parallel. We'll discuss this strategy in Section 7.11, with examples in Sections 7.12 and 7.13.

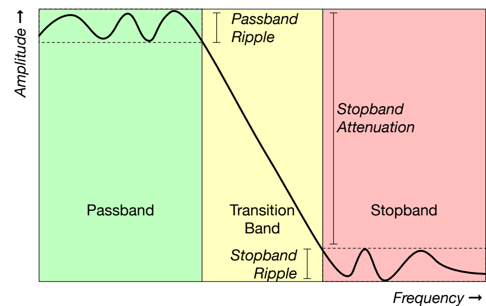



*Figure 100* Filter terms (frequency domain).

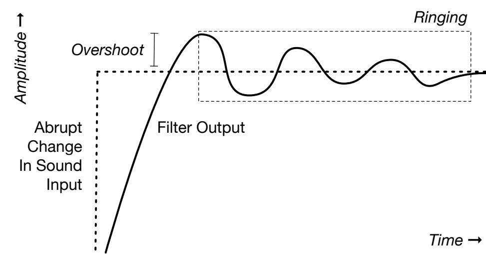



*Figure 101* Filter terms (time domain).

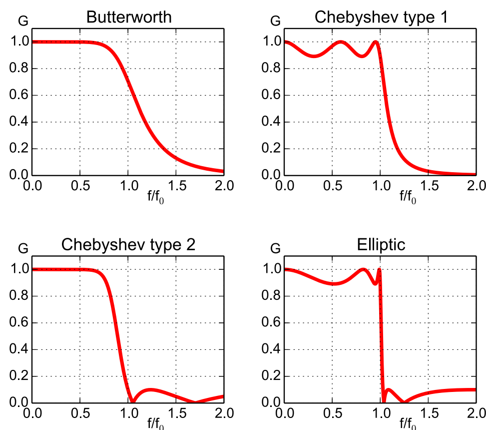



*Figure 102* Plots of frequency domain amplitude of some common filter designs. ©55

**Some Well-Known Historical Synthesizer Filters** Filters are so fundamental to subtractive synthesizers that they have developed a history and lore all their own. It's worthwhile discussing a few classic ones.

Easily the most famous synthesizer filter in history is **Robert Moog's** 24 dB (4-pole) low pass filter, found on many of his designs, including the **Moog Minimoog Model D** (Figure 103). This filter, known as a **transistor ladder filter** because of the layout of its patented circuit (Figure 122, page 101), has often been cloned by competitors and has been the subject of considerable legal wrangling.[76] This filter gives the Model D a potent, rich sound that made it popular as a lead or solo instrument. However when the resonance is increased, the filter tends to morph into a bandpass filter and so cuts bass sounds as well. This often requires augmenting it with some kind of bass boosting compensation. To this day, the Moog filter remains the standard against which new filters are compared. Ladder filters are discussed at length in Section 7.12.

*Figure 103* Moog Minimoog Model D. Repeat of Figure 51. ©56

Moog's early competitor, **Don Buchla**, took a different approach, employing a **low pass gate** in his designs (such as Figure 47, page 50). This may be thought of as a mild (6 dB or 1-pole) low pass filter which also acts to some degree as an amplifier, and which has a complex nonlinear relationship between the filter cutoff and the volume of the incoming signal.

Another competitor, Korg, combined a 6 dB (1 pole) high pass filter with a 12 dB (2-pole) low pass filter to create a famously aggressive, harsh sound in several machines, notably the **Korg MS20** (Figure 49, page 50). Another example of an aggressive, indeed "dirty" sounding, filter is found in the famous **Electronic Dream Plant** (or **EDP**) **Wasp**, shown in Figure 104. Both of these filters have been replicated in many other synths since.

*Figure 104* EDP Wasp. Yes, it really was that yellow.©58

Filters like Korg's and Moog's were good choices for creating cutting monophonic leads. In contrast, **Tom Oberheim's** early polyphonic **SEM** machines (such as the **Oberheim 4-Voice** and **8-Voice**, see Figure 53, page 51) were distinguished by their 12-dB (2-pole) **state-variable** or **multimode filter** which could be swept smoothly from low pass, through notch, to high pass, or could be switched to band pass. Unusually, the SEM filter was not **self-resonant**. Another very well regarded polyphonic synthesizer filter was Roland's IR3109 2- and 4-pole filter chip used in its Jupiter, Juno, and JX lines, such as the **Roland Jupiter 8** and **Roland MKS-80 Super Jupiter** (Figure 59, page 53).

**Doug Curtis's** company, **Curtis Electromusic Specialties** (or **CEM**), introduced many analog chips used by synthesizer manufacturers to simplify their circuitry. CEM designs, notably the **CEM 3320** multimode 24-dB (4-pole) filter chip, found their way onto synths ranging from the **Prophet 5** (Figure 56, page 52) to the **Oberheim OB-Xa** (page 52) to the **PPG Wave** (Footnote 115, page 119).

While most synthesizers come with only one or two filters, a few have a great many. For example, the **E-mu Systems Morpheus** would have just been a boring **rompler** had it not sported *hundreds* of different filters, each up to 14 poles, and with dynamically tweakable parameters.

[76]For example, ARP changed a copycat filter in the **ARP 2600**, Figure 48, page 50, under threat of lawsuit.

## 7.1 Digital Filters

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

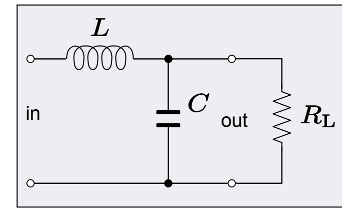



*Figure 105* Simple second-order low pass analog filter. No, we're not doing circuits.©59

One conceptually simple way to make a filter is to take a digital signal, convert it into the **frequency domain**, manually change (through multiplication) the amplitudes and phases we want, then convert it back into the **time domain**. The conversion to the frequency domain and back again can be done using a **Fast Fourier Transform** (or **FFT**), discussed in Section 12. But it turns out you can do essentially the same process while staying wholly in the time domain through a simple 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 $n$,[77] 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: $y(n) \leftarrow b_0x(n) + b_1x(n-1) + \cdots + b_kx(n-k)$. The constants $b_0, b_1, ..., b_k$ 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 $y(...)$.

Why does this result in a filter? Consider the equation $y(n) = 1/2\ x(n) + 1/2\ x(n-1)$, that is, $b_0 = 1/2$ and $b_1 = 1/2$. This averages 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 the one before. But if you had a very *high* frequency sine wave, then each successive sample would be very different, and this procedure is essentially using them to nullify one another. Thus this is a simple low pass filter, as shown in Figure 109. It is *smoothing* the signal, reducing the amplitude of the high frequencies.

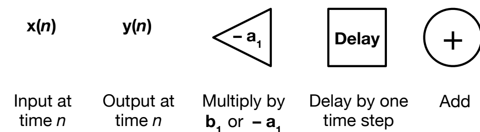



*Figure 106* Filter diagram symbols.

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 107.

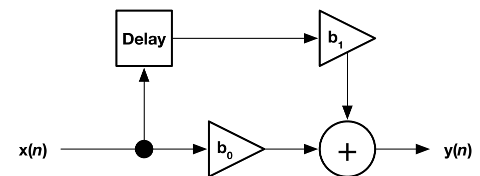



*Figure 107* First-order Finite Impulse Response (FIR) filter. See Figure 106 for explanation of these symbols.

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 $x(n-1)$ delayed exactly *one* timestep prior. Because we only need information one timestep back, this is known as a **first-order filter**.[78]

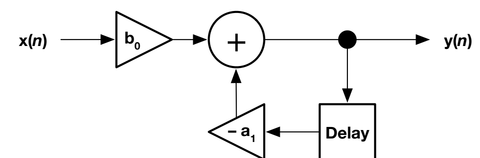
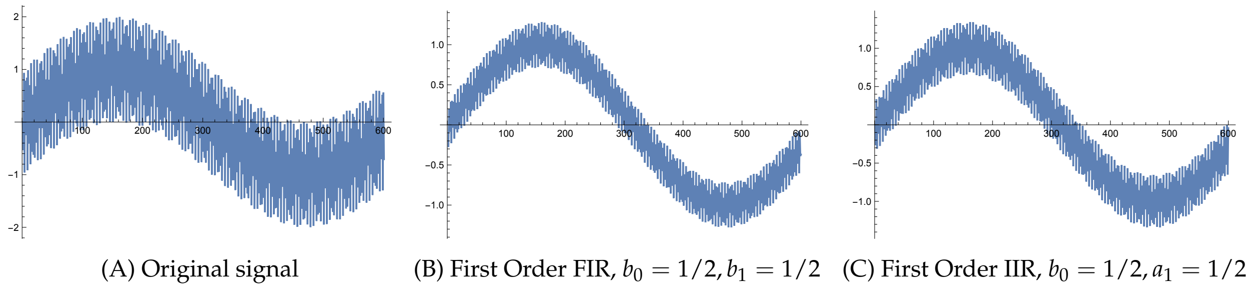



*Figure 108* First-order simple Infinite Impulse Response (IIR) filter.

Now consider the filter in Figure 108. The output of this filter is specified a little differently. It's of the form

[77]As a computer science guy, I'd use $t$ for time, but $n$ is the electrical engineering convention.

[78]If you increase the length of the delay, this becomes a **feedforward comb filter**. We'll discuss that more in Section 10.



(A) Original signal (B) First Order FIR, $b_0 = 1/2, b_1 = 1/2$ (C) First Order IIR, $b_0 = 1/2, a_1 = 1/2$

*Figure 109* Effects of the first-order digital filters from Figures 107 and 108 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. Note that the amplitude has also changed (from around 2 to around 1), as these are not unity gain filters.

$y(n) = b_0 x(n) - a_1 y(n-1)$. This is a simple form of an **Infinite Impulse Response** or **IIR** filter.[79] This filter is *infinite* in the sense that if $x(n)$ is (say) 1.0 at $n = 0$ but is 0.0 for $n > 0$ thereafter — the impulse again — the filter may continue to exhibit some non-zero value for $y(n)$ forever, as 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 first-order 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 subtracted: we'll get back to that.

Figure 109 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. [80]

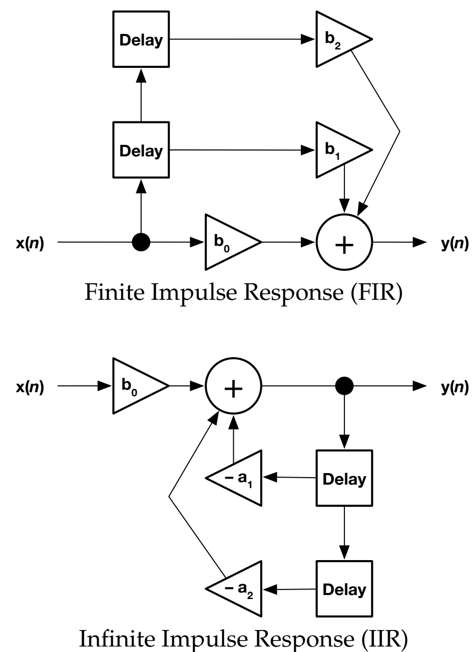



*Figure 110* Simple second-order filters

**Higher Order Filters** A **second order filter** requires information two steps back. The second order extension of the FIR filter shown earlier is $y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2)$, and the second order extension of the simple IIR filter shown earlier is $y(n) = b_0 x(n) - a_1 y(n-1) - a_2 y(n-2)$. You can diagram these filters by stacking up delay modules, as shown in Figure 110. Indeed, you can make **third-order, fourth-order**, and in general ***n*th-order** filters by stacking up more delay modules in the same pattern.

[79] If you increase the length of the delay, this filter becomes a **feedback comb filter**. We discuss that in Section 10.

[80] The filters shown here are **linear filters**: the most common kind. Consider a sound $X = \langle x_0, x_1, ... \rangle$. A filter $y(n)$ is some function of $X$, that is, $y(n) = f(X, n)$. In a linear filter, the filter doesn't change with a louder sound (it just gets louder), and filtering two sounds added together is the same thing as filtering them separately and then adding them. That is, $\forall g : f(gX, n) = gf(X, n)$, and for any two $X_1$ and $X_2$, $f(X_1 + X_2, n) = f(X_1, n) + f(X_2, n)$. There also exist **nonlinear filters** which are more difficult to design and use but can often produce better results.

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 111 and is known as the **Direct Form I** of a digital IIR filter.

This diagram corresponds to the equation

$$y(n) = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2) \\ - a_1 y(n-1) - a_2 y(n-2)$$

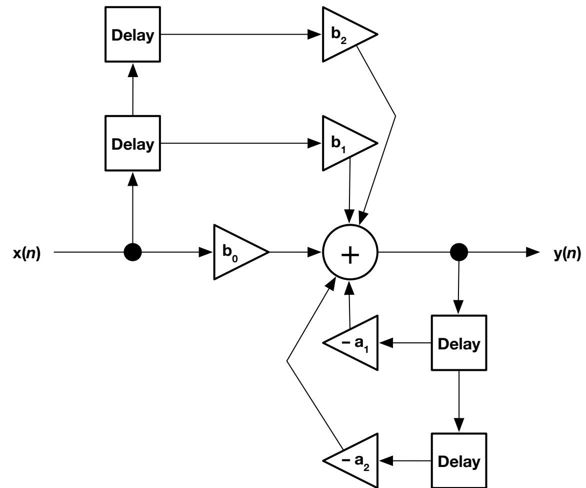



*Figure 111* A full second-order IIR filter (Direct Form I).

By the way, there are other forms than just Direct Form I. For example, consider **Direct Form II** is shown in Figure 112 at right. This form rearranges things to reduce the number of delays. However, because there are two addition points, it's more complex in dealing with issues such as overflow when using fixed-point arithmetic. We'll stick with Direct Form I.

Here's one advantage to subtracting all the $a_i$ values: 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:

$$y(n) + a_1 y(n-1) + a_2 y(n-2) \\ = b_0 x(n) + b_1 x(n-1) + b_2 x(n-2)$$

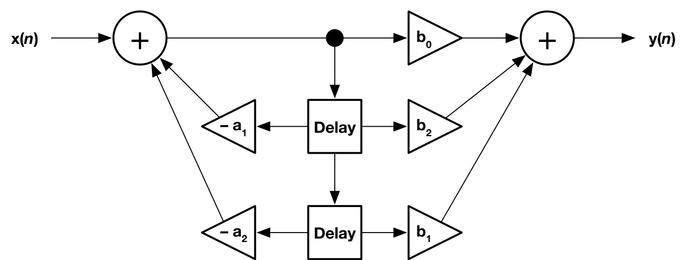



*Figure 112* Full second-order IIR filter (Direct Form II).

Cute! In general, we have filters of the form:[81]

$$y(n) + a_1 y(n-1) + \cdots + a_N y(n-N) = b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M)$$

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 14** *Initialize a Digital Filter*

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

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

[81]We don't need or want an $a_0$, and it doesn't matter anyway, because if you insisted on it you could always make one by multiplying all the terms by whatever non-zero $a_0$ you wanted.

**Algorithm 15** *Step a Digital Filter*

```
1:  a ← ⟨a1...aN⟩ array of N real values                    ▷ Note: 1-based array
2:  b ← ⟨b1...bM⟩ array of M real values                    ▷ Note: 1-based array
3:  b0 ← real value                                         ▷ This is "b0", by default 1
4:  x0 ← real value                                         ▷ Current input
5:  Global y ← ⟨y1...yN⟩ array of N real values             ▷ Note: 1-based array
6:  Global x ← ⟨x1...xM⟩ array of M real values             ▷ Note: 1-based array

7:  sum ← x0 × b0
8:  for n from 1 to N do
9:      sum ← sum − a_n × y_n
10: for m from 1 to M do
11:     sum ← sum + b_m × x_m
12: for n from N down to 2 do                               ▷ Note backwards
13:     y_n ← y_{n−1}
14: y1 ← sum
15: for m from M down to 2 do                               ▷ Note backwards
16:     x_m ← x_{m−1}
17: x1 ← x0
18: return sum
```

## 7.2 Building a Digital Filter

So far we've only seen a (trivial) 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. At this point we still have no idea (1) what order our filter should be nor (2) what the constants $a_i$... and $b_j$.. 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 ultimately 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 which explains its behavior.

3. We can verify this 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 will map the transfer function from the Laplace domain to a *different* complex-number space, called the **Z domain**. The Z domain makes it easy to build a digital filter, but there is no straightforward conversion from Laplace coordinates to Z coordinates. Instead we'll use an approximate conversion called the **bilinear transform**. Converting an analog filter design to a digital design is called **discretizing** the filter.

5. Once the transfer function is in the Z domain, it's simple to extract from it the **coefficients** $a_i$.... and $b_j$.... with which we will build the digital filter in software.

6. Alternatively one 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 also discuss **composing** multiple filters to create a more complex one.

Along the way, we will derive the transfer functions (in both the Laplace and Z domains) of well known filters which can be used in a basic subtractive synthesizer.

### 7.3 Transfer Functions in the Laplace Domain

The **Laplace domain**[82] 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 its **transfer function** $H(s)$. This function relates the input $X(s)$ of some kind of system to its output $Y(s)$, that is, $Y(s) = H(s)X(s)$. It's more commonly written as the ratio between $X(s)$ and $Y(s)$, as $H(s) = \frac{Y(s)}{X(s)}$. 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 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** $\omega$ (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, at least for audio).[83] 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 $\omega$, we compute the **magnitude**[84] 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:

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

[82] The Laplace domain is called a *domain* for a reason: it's closely related to the **frequency domain** from Section 2. But don't let that confuse you here.

[83] By the way, I'm using $i$ throughout this Section to denote imaginary numbers as is customary in math and computer science, but electrical engineers use $j$ because they already use $i$ for electrical current.

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

We can also compute the filter's **phase response** — how much the phase shifts for partials at a given frequency — as $\angle H(i\omega)$, where $\angle(a+bi) = \tan^{-1}\frac{a}{b}$. Using our example:

$$H(s) = \frac{s^2-4}{s^2+s+2} \qquad H(2i) = \frac{(2i)^2-4}{(2i)^2+(2i)+2} = 2+2i \qquad \angle H(2i) = \angle(2+2i) = \tan^{-1}\left(\frac{2}{2}\right) = \frac{\pi}{4}$$

### 7.4 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 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. The roots of $X(s)$ are thus called the **poles** of the transfer function: think of them making 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:

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

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.

### 7.5 Amplitude and Phase Response

If we already have the roots we want, it's 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 $p_1, ..., p_n$ and zeros $z_1, ..., z_m$, then this is just

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

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

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

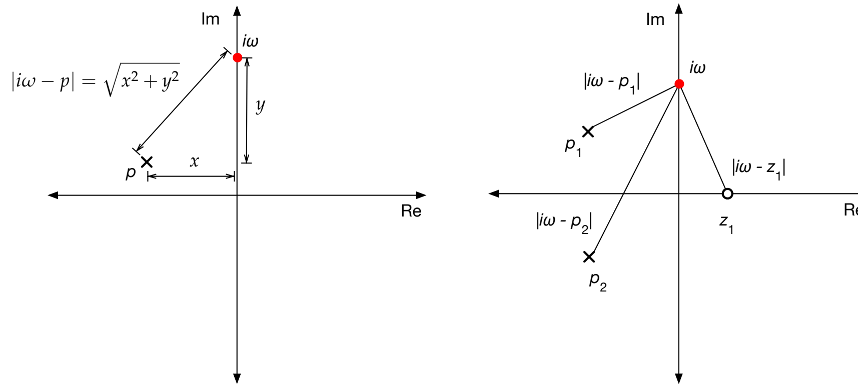



*Figure 114* (Left) relationship between a pole $p$, the current frequency $i\omega$, and its impact on the magnitude of the amplitude at that frequency. (Right) two poles and a zero and their respective magnitudes. In this example, the amplitude at $i\omega$ is $|i\omega - p_1| \times |i\omega - p_2| \times 1/|i\omega - z_1|$.

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

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

$$|H(2i)| = \frac{|2i - (-\frac{1}{2} - \frac{1}{2}i)| \times |2i - (-\frac{1}{2} + \frac{1}{2}i)|}{|2i - (-3)| \times |2i - (-2)|}$$

$$= \frac{\left|\frac{1+5i}{2}\right| \times \left|\frac{1+3i}{2}\right|}{|3+2i| \times |2+2i|}$$

$$= \frac{\frac{\sqrt{26}}{2}\frac{\sqrt{10}}{2}}{\sqrt{13}\sqrt{8}} \qquad = \frac{\sqrt{5}}{4}$$

In fact, we can easily plot the magnitude of the filter for any value of $\omega$, as shown in Figure 113. 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 114. 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.

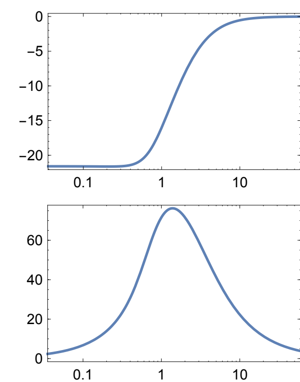



*Figure 113* Bode plot of the amplitude response (top) and phase response (bottom) of a filter with zeros $-\frac{1}{2} - \frac{1}{2}i$ and $-\frac{1}{2} + \frac{1}{2}i$, and with poles $-3$ and $-2$.

## 7.6 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:

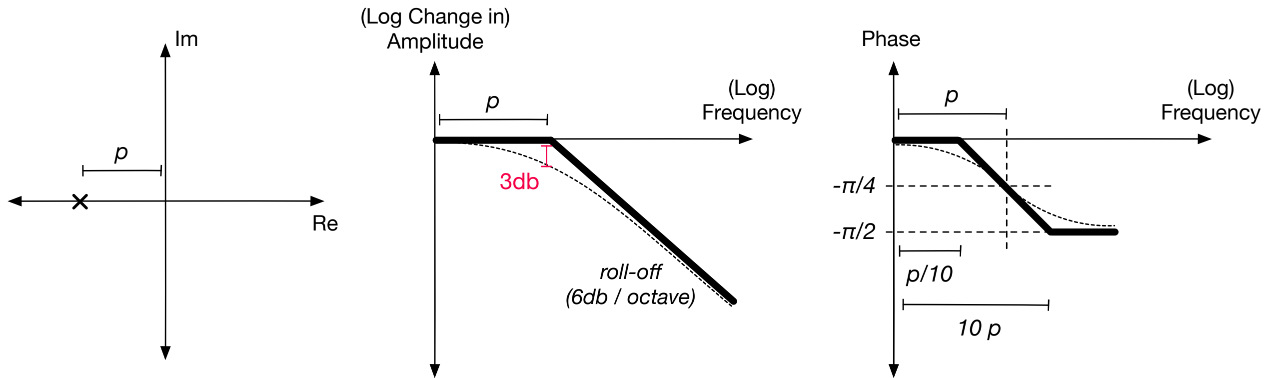



*Figure 116* (Left) Position of single pole at $-p$. (Center) Bode plot of the amplitude response of the filter. Boldface line represents idealized filter (and useful as a rule of thumb) while the dotted line represents the actual filter. At the idealized cutoff frequency ($p$) the real filter has dropped 3 dB. This being a one-pole filter, at the limit the roll-off is a consistent 6 dB per octave. (Right) Bode plot of the phase response of the filter. Again, the 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, for a pair of poles $\langle p_1, p_2 \rangle$, if $p_1 = a + bi$, then $p_2 = a - bi$. One is above the imaginary axis by $b$ and one is below by $b$. See Figure 115.

- 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.

- 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.

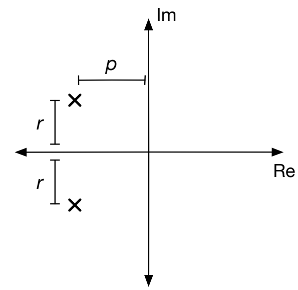



*Figure 115* Two poles as a complex conjugate pair. The value $p$ is the same as in Figure 116. The value $r$ is related to the degree of resonance in the filter.

Let's start with a single pole lying at $-p$ on the real ($x$) axis, as shown in Figure 116. 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 6 dB 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 3 dB 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 116 (right), Again, the boldface line shows the idealized (or in this case more like fanciful) response.

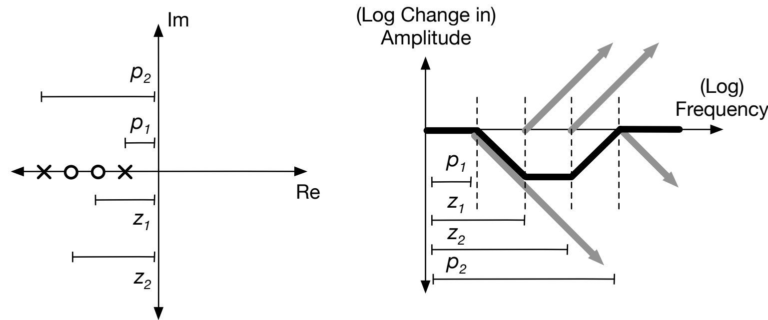



*Figure 117* (Left) positions of two poles and two zeros on the real axis. (Right) Approximate Bode plot showing impact of each pole and zero in turn. Bold line shows final combined impact: a band reject filter of sorts. Gray bold lines are the roll-offs of each filter starting at their respective cutoff 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 115. 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 12 dB.[85]

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 117. 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.[86]

**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:

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

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

[86]I'm not discussing the phase response, since it's not as important for us. Consider yourself fortunate.

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

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

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 $-p_1, ..., -p_i$, we need to have $p_1 \times ... \times p_i$ in the numerator to make the filter have unity gain. Thus we have:

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

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 91. This is effectively a special case of the low pass unity-gain second-order filter discussed in Section 7.7. You might try working out what happens when $p_1$ and $p_2$ are complex conjugates, and its relationship to Equation 3.

## 7.7 Analog Second-Order Filters

Now let's build some standard second-order analog filters. In this Section we'll define the transfer functions for these filters, and derive their poles, zeros, and amplitude response. In Section 7.10 we'll then convert these filters to their digital filter forms.

A first-order filter (discussed later in Section 7.12) has essentially no parameters: you can make a trivial low-pass or high-pass filter. But as the order of a filter increases, so does the number parameters with which we may play to shape its frequency response. Striking the middle ground are second-order filters. Basic second-order filters can come in low-pass, high-pass, band-pass, and notch configurations, and they all have one major tweakable parameter: **resonance**.

All second-order unity gain filters 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 $\omega_0$ is the desired (real valued) **cutoff frequency**. $N(s)$ is a polynomial which varies for different kinds of filters (low pass, high pass, etc.) Recall that $Q = \sqrt{1/2}$ is the dead-flat position (above which resonance starts peaking). If you set the resonance to this minimum, this transfer function degenerates to a well known filter called a second-order **Butterworth filter** (recall Figure 102).

You'll often see this equation in the literature without any $\omega_0$: that is, $H(s) = \frac{N(s)}{s^2 + \frac{s}{Q} + 1}$. This is the canonical form where $\omega_0$, our cutoff frequency, has been set to 1. To include the cutoff frequency, just replace $s$ with $\frac{s}{\omega_0}$ everywhere, including in whatever polynomial $N(s)$ is.

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

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

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.[87] 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 second-order 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 second-order 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:

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

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

[87] You 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 $c = 1$.

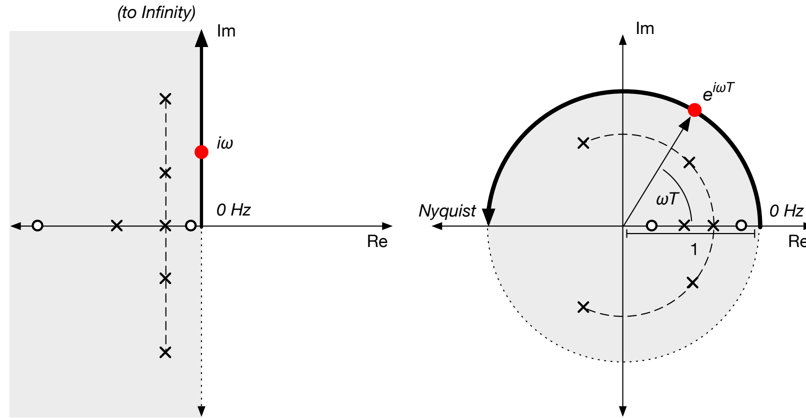



*Figure 118* Relationship between the Laplace (left) and Z Domains (right). Notice that the entire (infinite) left half of the Laplace plane is mapped to inside the unit circle in the Z plane. Whereas the cutoff frequency $\omega$ in the Laplace plane goes up along the imaginary axis from 0 to $\infty$ as $i\omega$, in the Z domain it runs along the unit circle as $e^{i\omega T}$, corresponding to going from 0 to the Nyquist frequency. Note how the example poles and zeros are warped in the mapping. This 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.dspguide.com/.

**Notch** Finally, a notch second-order filter at unity gain has $N(s) = 1 + \frac{s^2}{w_0^2}$. 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:

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

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

## 7.8 Converting to Digital: the Z Domain and the Bilinear Transform

Now we're ready to convert our analog filter to digital, so we can extract the coefficients of the filter to plug into Algorithms 14 and 15. But there's a problem. It turns out that the Laplace domain, meant for analog filters, cannot be directly used to build digital filters. To do this, we need our poles and zeros in a *completely different* complex-number space called the **Z domain**, from which we can directly extract the information we need to build a digital filter. While $H(s)$ is a transfer function in Laplace, $H(z)$ is a transfer function in Z. The mapping from one to the other is strange: the entire left hand side of the Laplace domain (the negative real region) gets warped and squished to within the unit circle of the Z domain, as shown in Figure 118. In the Laplace domain, frequency $\omega$ goes from 0 to $\infty$, represented as $i\omega$ going up the imaginary axis. But in the Z domain, frequency only goes from 0 to Nyquist, traveling along the upper border of the unit circle as $e^{i\omega T}$. $T$ is the **sampling interval**, the inverse of the **sampling rate**. If you're sampling at 44.1KHz, then $T = 1/44100$.

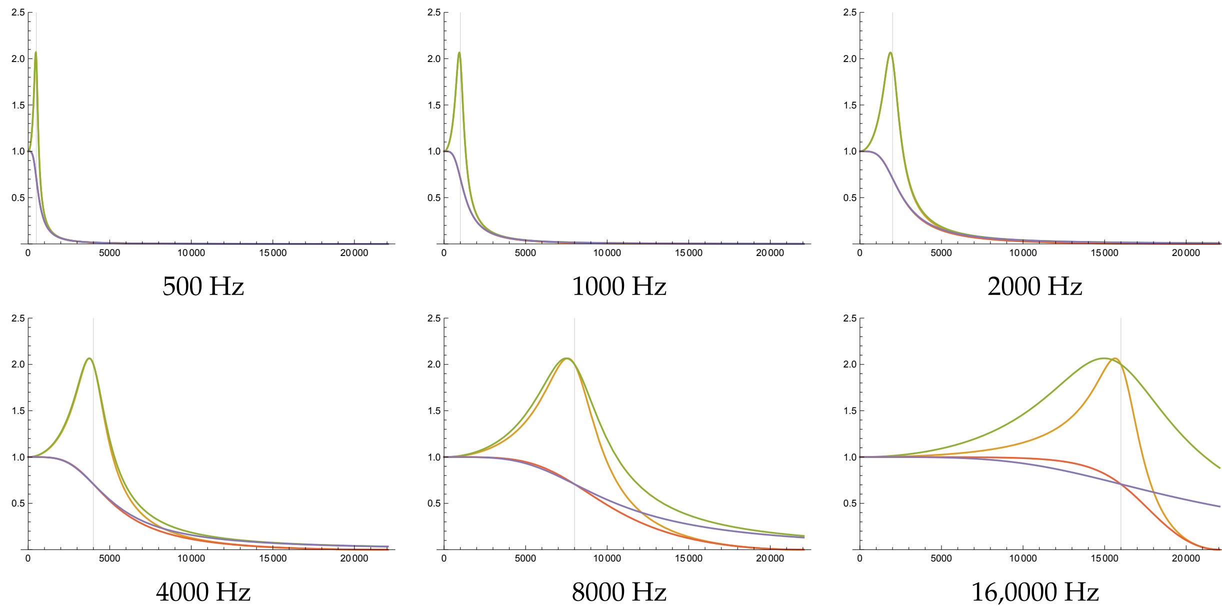



*Figure 119* Comparison of Magnitude (Amplitude) Responses of equivalent second-order low-pass analog and digital filters. A Laplace Domain filter with resonance $Q = 2$ is compared to an equivalent Z Domain filter converted at 44.1KHz via a Bilinear Transform with $c$ set up so that the filters all line up at the cutoff frequency (the gray vertical line). The same two filters (in Laplace and Z Domains) are also compared without resonance. At cutoff frequencies of 500 and 1000 Hz the two are very nearly identical (The Laplace plot sits directly on top of Z and so hides it). By 16,000 Hz the divergence between analog and digital amplitude response is significant.

While the Laplace domain is infinite, the Z domain only goes to Nyquist, and so to convert one to the other we must squish the infinite space into the finite. Thus all transfer function mappings from Laplace to Z are approximate and necessarily have failings. The most commonly used approximate mapping is called the **bilinear transform**. It looks like this:[88]

$$s = c\frac{z-1}{z+1}$$

The primary part of this conversion is $\frac{z-1}{z+1}$, which has the property that, in the Z domain, you can still use $|H(e^{i\omega})|$ to compute the amplitude response just like you'd compute it in Laplace with $|H(iw)|$. Phase response is also similar, with a few tweaks.

But because we're mapping the (0...∞) frequencies of the Laplace domain into the (0... Nyquist) frequencies of Z, this necessarily subjects the frequencies to a nonlinear **frequency warping**. Thus we cannot create exactly the same filter response in Z that we can in Laplace, though we can try to match the responses as best we can by adjusting the $c$ constant. Normally we'd set $c$ so that the Laplace and Z amplitude responses have equal value at the cutoff frequency.[89] This is done by setting $c$ as:

$$c = \omega_0 \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)} \quad = \omega_0 \cot(\omega_0 T/2)$$

[88]The bilinear transform is sometimes written as $s = c\frac{1-z^{-1}}{1+z^{-1}}$. It's the same thing.

[89]Outside of audio, one often sees the simpler $c = 2/T$, which lines up the cutoff frequencies when they are very low.

Figure 119 shows the effect of the bilinear transform. As can be seen for small cutoff frequency values, the analog and digital frequency responses are quite close, but as the cutoff frequency increases, they begin to diverge markedly.

*Example.* Let's convert a Laplace transfer function to the Z domain. To keep things simple, we'll entertain the ridiculous notion that $c = 2$:

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

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 zero or negative, with the highest one in the denominator to be zero:

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

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

$$y(n) + a_1 y(n-1) + \cdots + a_N y(n-N) = b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M)$$

... then the transfer function, in the Z domain, is:

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + \cdots + a_N z^{-N}}$$

*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 [$z^{-1}$].

## 7.9 Frequency Response and Pole and Zero Placement in the Z Domain

Even though the Z domain is shaped differently, computing the amplitude and phase response in Z is surprisingly similar to doing it in Laplace. Given a frequency $\omega$, we used $i\omega$ in Laplace. For Z, we'll use $e^{i\omega T}$ instead, and add a little twist when computing the phase response.

Recall that the amplitude response of a unity gain analog filter, given poles $p_k$ and zeros $z_j$, is:[90]

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

In the Z domain, you'd more or less do the same thing, but with $e^{i\omega T}$, which can be written[91] as $\cos(\omega T) + i\sin(\omega T)$:

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

Remember that $T$ is your sampling interval: if you were sampling at 44.1KHz, then $T = 1/44100$.

Phase is slightly different. Recall that phase response of a unity gain filter in Laplace is:

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

In $Z$ it's in the same theme with a little twist at the beginning:

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

Note the unexpected term $\omega T(k-j)$: $j$ and $k$ are the number of poles and zeros respectively. If your transfer function has the same number of poles and zeros, then this term will equal 0.

**Designing in Z** Wouldn't it be easier to define the poles and zeros in the Z domain, rather than defining them in Laplace, then extracting the transfer function, and then converting the transfer function via the Bilinear Transform? Sure it would.

We didn't do that for a couple of reasons. First, since we're largely modeling analog synths, it's important to learn about the Laplace domain, 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 less intuitive. And third, most classic filters are in analog, with well-understood properties; it makes more sense to study them in analog and then convert as best we can to digital afterwords.[92]

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. The Bilinear Transform is a useful approximation, and we'll take advantage of it in the next two Sections (7.7 and 7.10). 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.[93]

[90] As a reminder, the magnitude of a complex number $|a+bi|$ is $\sqrt{a^2+b^2}$ and its angle $\angle(a+bi)$ is $\tan^{-1}\frac{b}{a}$.

[91] If you've forgotten, see Footnote 174 on page 162.

[92] This is very common. For example, Vadim Zavalishin's *The Art of VA Filter Design* (*VA* referring to **virtual analog** synths) goes in depth on historical filters and how to replicate them digitally: but the text largely stays in Laplace. https://cs.gmu.edu/~sean/book/synthesis/VAFilterDesign.2.1.0.pdf

[93] MicroModeler DSP is a great online tool for building filters in the Z domain directly from poles and zeros. http://www.micromodeler.com/dsp/

## 7.10 Digital Second-Order Filters

In Section 7.7 we covered analog second-order filters in the Laplace domain. Now let's convert them as best we can to the Z domain to extract their coefficients and build the filter.[94] 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! To make things clearer, I have added a substitution called $J$.

From the 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 $\omega_0$, $Q$, and $T$. Conveniently, because all four filters have the same poles, they also all have the same $a_1$ and $a_2$ constants!

Some things to remember. First, if you have a cutoff frequency of $F$ Hz, then $\omega_0 = 2\pi \times F$. Second, neither $\omega_0$ nor $Q$ should be 0. Third, we'll assume $c = \omega_0 \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$, where $T$ is the sampling interval (for example, if you're sampling at 44.1KHz, then $T = 1/44100$).

In all these equations we can remove a $\frac{1}{\omega_0}$ in several places by cancelling it with the corresponding $\omega_0$ at the beginning of $c$. To do this simplification, I have added a substitution $d$ for $c$ which strips the $\omega_0$ off. If you think about it, this is effectively converting the $H(s)$ to its canonical form where $\omega_0 = 1$ (recall Section 7.7) prior to doing the Bilinear Transform.

**Low Pass**

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

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

Now substitute $d = \frac{1}{\omega_0}c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

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

$$= \frac{Q + 2Qz + Qz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + (d + d^2Q + Q)z^2}$$

*Thanks to Mathematica*

Now substitute $J = d + d^2Q + Q$, and also multiply by $\frac{1/J \times z^{-2}}{1/J \times z^{-2}}$

$$= \frac{Q + 2Qz + Qz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + Jz^2}$$

$$= \frac{1/J \times Qz^{-2} + 1/J \times 2Qz^{-1} + 1/J \times Q}{1/J \times (-d + d^2Q + Q)z^{-2} + 1/J \times (-2d^2Q + 2Q)z^{-1} + 1}$$

[94] Instead of using these derivations, you might consider using the ones from a classic cookbook of equalizer and filter coefficients found here: https://webaudio.github.io/Audio-EQ-Cookbook/audio-eq-cookbook.html They produce the same results I believe. Note that you'll need to divide all the coefficients by $a_0$ (see the cookbook's Formula 2), and the "$\omega_0$" in the cookbook is actually $\omega_0 T$, as stated in Step 2 (just after Formula 6).

Thus we have the following coefficients for our digital filter:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J \times Q$ | $z^0 (= 1)$ |
| $b_1$ | $1/J \times 2Q$ | $z^{-1}$ |
| $b_2$ | $1/J \times Q$ | $z^{-2}$ |
| $a_1$ | $1/J \times (2Q - 2d^2Q)$ | $z^{-1}$ |
| $a_2$ | $1/J \times (Q - d + d^2Q)$ | $z^{-2}$ |

where $J = Q + d + d^2Q$, and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

**High Pass**

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

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

Now substitute $d = \frac{1}{\omega_0} c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

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

$$= \frac{d^2Q - 2d^2Qz + d^2Qz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + (d + d^2Q + Q)z^2}$$

*Thanks to Mathematica*

Now substitute $J = d + d^2Q + Q$, and also multiply by $\frac{1/J \times z^{-2}}{1/J \times z^{-2}}$

$$= \frac{d^2Q - 2d^2Qz + d^2Qz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + Jz^2}$$

$$= \frac{1/J \times d^2Qz^{-2} - 1/J \times 2d^2Qz^{-1} + 1/J \times d^2Q}{1/J \times (-d + d^2Q + Q)z^{-2} + 1/J \times (-2d^2Q + 2Q)z^{-1} + 1}$$

Thus we have the following coefficients for our digital filter:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J \times d^2Q$ | $z^0 (= 1)$ |
| $b_1$ | $1/J \times -2d^2Q$ | $z^{-1}$ |
| $b_2$ | $1/J \times d^2Q$ | $z^{-2}$ |
| $a_1$ | $1/J \times (2Q - 2d^2Q)$ | $z^{-1}$ |
| $a_2$ | $1/J \times (Q - d + d^2Q)$ | $z^{-2}$ |

where $J = Q + d + d^2Q$, and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

**Band Pass**

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

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

Now substitute $d = \frac{1}{\omega_0}c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

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

$$= \frac{-d + dz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + (d + d^2Q + Q)z^2}$$

*Thanks to Mathematica*

Now substitute $J = d + d^2Q + Q$, and also multiply by $\frac{1/J \times z^{-2}}{1/J \times z^{-2}}$

$$= \frac{-d + dz^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + Jz^2}$$

$$= \frac{1/J \times -dz^{-2} + 1/J \times d}{1/J \times (-d + d^2Q + Q)z^{-2} + 1/J \times (-2d^2Q + 2Q)z^{-1} + 1}$$

Thus we have the following coefficients for our digital filter:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J \times d$ | $z^0 (= 1)$ |
| $b_1$ | $0$ | $z^{-1}$ |
| $b_2$ | $1/J \times -d$ | $z^{-2}$ |
| $a_1$ | $1/J \times (2Q - 2d^2Q)$ | $z^{-1}$ |
| $a_2$ | $1/J \times (Q - d + d^2Q)$ | $z^{-2}$ |

where $J = Q + d + d^2Q$, and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

**Notch**

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

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

Now substitute $d = \frac{1}{\omega_0}c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

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

$$= \frac{(Q + Qd^2) + (2Q - 2d^2Q)z + (q + d^2q)z^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + (d + d^2Q + Q)z^2}$$

*Thanks to Mathematica*

Now substitute $J = d + d^2Q + Q$, and also multiply by $\frac{1/J \times z^{-2}}{1/J \times z^{-2}}$

$$= \frac{(Q + d^2Q) + (2Q - 2d^2Q)z + (Q + d^2Q)z^2}{(-d + d^2Q + Q) + (-2d^2Q + 2Q)z + Jz^2}$$

$$= \frac{1/J \times (Q + d^2Q)z^{-2} + 1/J \times (2Q - 2d^2Q)z^{-1} + 1/J \times (Q + d^2Q)}{1/J \times (-d + d^2Q + Q)z^{-2} + 1/J \times (-2d^2Q + 2Q)z^{-1} + 1}$$

Thus we have the following coefficients for our digital filter:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J \times (Q + d^2Q)$ | $z^0 (= 1)$ |
| $b_1$ | $1/J \times (2Q - 2d^2Q)$ | $z^{-1}$ |
| $b_2$ | $1/J \times (Q + d^2Q)$ | $z^{-2}$ |
| $a_1$ | $1/J \times (2Q - 2d^2Q)$ | $z^{-1}$ |
| $a_2$ | $1/J \times (Q - d + d^2Q)$ | $z^{-2}$ |

where $J = Q + d + d^2Q$, and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

## 7.11 Filter Composition

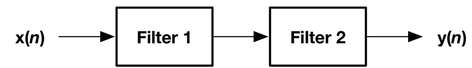



*Figure 120* Filters in Series.

Another way to build a filter is by combining other filters. This is pretty easy, because filters can be straightforwardly **composed** to form more complex ones.

One particularly useful way to do this is to string multiple filters in **series**, as shown in Figure 120. This has the effect of *multiplying* their transfer functions (either in Laplace or in Z). Let's say you had two filters $A$ and $B$. Then you have:

$$H_{series}(z) = H_A(z) \times H_B(z) = \frac{Y_A(z)}{X_A(z)} \times \frac{Y_B(z)}{X_B(z)}$$

*Example.* Let's put our previous two-pole example (we'll call it $A$) in series with another two-pole filter:

$$\begin{aligned} H_{series}(z) = H_A(z) \times H_B(z) =& \frac{4/3 + 4/3z^{-1}}{1 - 10/3z^{-1} + z^{-2}} \times \frac{1 + 3z^{-1} - 2z^{-2}}{1 - 2z^{-2}} \\ =& \frac{4/3 + 28/3z^{-1} - 2/3z^{-2} - 2/3z^{-3}}{1 - 10/3z - z^{-2} - 20/3z^{-3} - 1/2z^{-4}} \end{aligned}$$

The important thing to notice here is what happened to the order of the filter. Previously our highest order coefficients were $z^{-2}$. Since we multiplied those together, our highest order coefficients now can be up to $z^{-4}$. In general, multiplying two filters of order $a$ and $b$ produces a filter of up to order $a + b$. So if you want a four-pole filter, one way to get it is to *put two two-pole filters in series.*[95] In Sections 7.7 and 7.10 we saw some two-pole filters: if you wanted a four-pole filter, you could just double-up those. But be aware that you're not just changing the magnitude (amplitude), but also the phase. Multiple filters in series can have quite an effect on phase distortion or delay.

Filters can also be combined in **parallel**, as shown in Figure 121. The result is to *add* the two transfer functions:

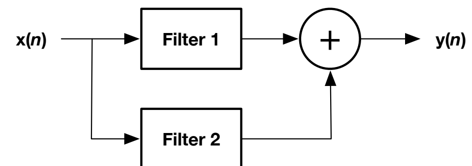



*Figure 121* Filters in Parallel.

$$H_{parallel}(z) = H_A(z) + H_B(z) = \frac{Y_A(z)}{X_A(z)} + \frac{Y_B(z)}{X_B(z)}$$

Again, this can be done either in Laplace or in Z. This can also increase the order of the filters, though the math is less straightforward when $X_A(z) \neq X_B(z)$.

You can guess the effect here: we're just taking the sound, running it through two separate filters, and then mixing the result. We'll see an example of this in Section 7.13. Remember however that filters change both the phase and the magnitude of a sound: if your filters are changing the phase in different ways, then you'll wind up with phase interference effects. This isn't necessarily bad: many sound effects (Section 10) rely on it. But you should be aware of the possibility.

In addition to adding two filters or sounds, you can *subtract* one from the other. To subtract a sound (or filter output) $A$ from another sound $B$, multiply the amplitude of $A$ by $-1$, that is, invert its wave; then put $A$ and $B$ in parallel. Subtraction makes it easy to build high pass, band pass, and

[95] Note that in an ideal math world, it won't matter in what order you put the filters when in series. But in a digital floating-point environment, you should expect somewhat different results due to numerical inaccuracy.

notch filters from just one single low pass filter. For example, take a sound and run it through a low pass filter, producing a sound with only the low frequencies remaining. Then *subtract this sound* from the original sound wave: all that's left are the high-frequency elements: you're produced a high pass filter. You can make a band pass filter by simply putting a low pass filter and a high-pass filter in series. And finally, you can make a notch filter by subtracting a band pass filter from the original sound.

## 7.12 First Order and Ladder Filters

The **Moog transistor ladder filter** is a resonant four-pole (24dB) low pass filter patented by **Robert Moog** in 1969. It is easily the most famous filter in synthesizer history because of its association with many Moog products, and especially the **Moog Minimoog Model D** (see Figure 103, page 80).

Moog's filter is called a **ladder filter** because its schematic, as shown in Figure 122 at right, has a series of four pairs of transistors stacked on top of one another like rungs in a ladder. Each of the pairs effectively forms one 6dB (1 pole) low pass filter. Related ladder filters use similar components (such as **diode ladder filters**)[96] to achieve the same effect.

A basic ladder filter thus is just four 6dB filters in series, plus a gizmo to add resonance. So before we can compose those 6dB filters, we first need to talk a bit about them:

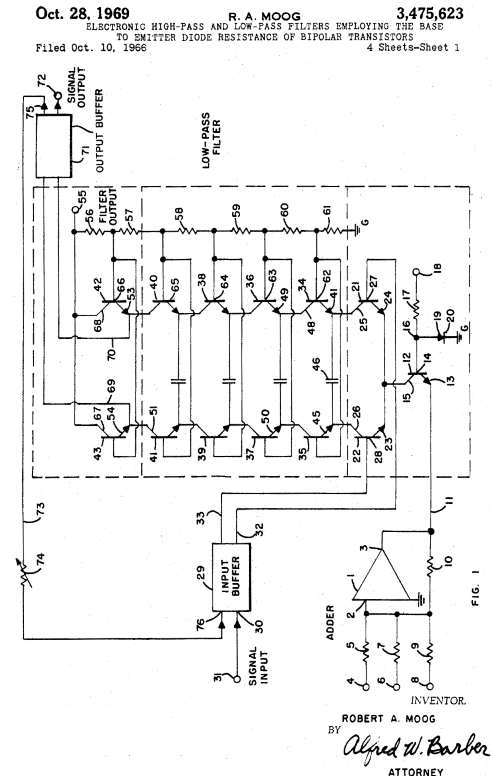



*Figure 122* Moog Low Pass Transistor Ladder Filter Circuit (Patent Filing). Note the dashed box (center of three) labelled "Low-Pass Filter" with four stacked transistor pairs (the "ladder"), and the feedback line (73, bottom left, gain-modulated by 74) providing resonance.[©60]

**First Order Low Pass Filters** The basic **first-order filter** is a 6dB (1 pole) low pass filter.[97] A unity gain filter of this kind has a very simple transfer function:

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

This filter has a very, *very* gradual roll-off. There is no $Q$ variable: first order filters cannot have resonance.

Converting this to a digital filter is straightforward. (For reminders about $c$ and $T$ and $\omega_0$, see Section 7.10, page 96). We start with the Bilinear Transform as usual:

$$H(z) = \frac{1}{\frac{1}{\omega_0} c \frac{z-1}{z+1} + 1}$$

[96] There are other ladder filters with different dynamics, beyond the scope of this book.

[97] A first-order filter can also be a 6dB high pass filter. This has a transfer function of $H(s) = \frac{\frac{s}{\omega_0}}{\frac{s}{\omega_0}+1} = \frac{1}{\frac{\omega_0}{s}+1}$. As an exercise you might try converting this to Z and extracting its digital filter coefficients.

Now substitute $d = \frac{1}{\omega_0} c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

$$H(z) = \frac{1}{d\frac{z-1}{z+1} + 1}$$
$$= \frac{1+z}{(1-d) + (1+d)z}$$

Now multiply by $\frac{z^{-1}}{z^{-1}}$, and substitute $J = 1 + d$

$$= \frac{z^{-1}+1}{(1-d)z^{-1} + J} \times \frac{1/J}{1/J}$$
$$= \frac{1/J \times z^{-1} + 1/J}{1/J \times (1-d)z^{-1} + 1}$$

Thus we have the following coefficients for our digital filter:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J$ | $z^0 (= 1)$ |
| $b_1$ | $1/J$ | $z^{-1}$ |
| $a_1$ | $1/J \times (1-d)$ | $z^{-1}$ |

where $J = 1 + d$ and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

**4-Pole Low-Pass Ladder Filters** A basic 4-pole low pass ladder filter consists of four one-pole 6dB low pass filters in series (24dB in total), plus a feedback loop to provide resonance. Looking back at Moog's design in Figure 122, we can see both the ladder of transistor pairs as well as the explicit feedback line. The overall structure is shown in Figure 123.

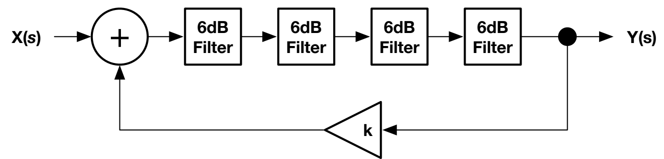



*Figure 123* Analog 4-pole low pass ladder filter structure.

Let's handle the four 6dB filters in series first. As discussed in Section 7.11, this just multiplies their transfer functions:

$$H'(s) = \frac{1}{(\frac{s}{\omega_0}+1)} \times \frac{1}{(\frac{s}{\omega_0}+1)} \times \frac{1}{(\frac{s}{\omega_0}+1)} \times \frac{1}{(\frac{s}{\omega_0}+1)} \quad = \frac{1}{(\frac{s}{\omega_0}+1)^4}$$

Now is a good time to understand that **resonance** in an analog filter is normally caused by **feedback**: some of the output of the filter typically gets added back into the input. But so far we have no feedback: we have just strung together four simple one-pole 6dB filters, and one-pole filters cannot have resonance, no matter how many you string together. To make the filter resonant, we need to explicitly introduce feedback. As shown in Figure 123, the ladder filter does this by taking the output of the filter and feeding it directly into the input, multiplied by a gain of $k$.

To revise our transfer function to accommodate this, recall that the output of our basic filter (without the feedback portion) is $Y(s) = X(s)H'(s)$, where $X(s)$ is the input. But with the feedback added, the input to the system is no longer just $X(s)$ but is $X(s) + kY(s)$, that is, it is the input plus a bit of the output rolled back in. The output is now:

$$Y(s) = (X(s) + kY(s))H'(s)$$

Solving for $Y(s)$, we get

$$Y(s) = \frac{H'(s)X(s)}{1 + kH'(s)}$$

And so our final transfer function $H(s)$ is:

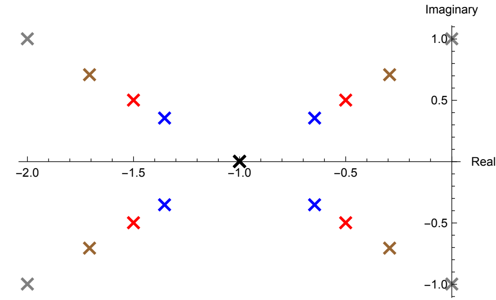



*Figure 124* Poles of a 24dB low pass ladder filter, in Laplace, $w_0 = 1$, for $k = 0$ (four stacked poles), $k = 1/16$, $k = 1/4$, $k = 1$, and $k = 4$. Notice that at $k = 4$ the right two poles cross the imaginary axis, beyond which the filter is unstable. Thus the maximum resonance for this filter is at $k = 4$.

$$H(s) = \frac{Y(s)}{X(s)} = \frac{\frac{H'(s)X(s)}{1+kH'(s)}}{X(s)} = \frac{H'(s)}{1 + kH'(s)} = \frac{1}{1/H'(s) + k} = \frac{1}{(\frac{s}{\omega_0} + 1)^4 + k}$$

Note that there's no $Q$ value for resonance: instead we'll rely on the $k$ value. At $k = 0$ there's no feedback and hence no resonance. Resonance increases with more $k$. The four poles are arranged in an interesting X-shaped configuration, located at $-w_0 \pm (-1)^{3/4}k^{1/4}w_0$ and $-w_0 \pm (-1)^{1/4}k^{1/4}w_0$ respectively. At $k = 0$, all four poles are located at $-w_0$, but as $k$ increases the poles head off in four directions, as shown in Figure 124, until $k = 4$ when two of the poles cross the imaginary axis and the filter becomes unstable.

Ladder filters are notorious for having the bass drop out with more resonance. As shown in Figure 125, as the resonance increases, the bass (look in the 0 Hz to 2000 Hz range) drops dramatically, and in some sense the filter starts looking like a band pass filter. Our previous filters didn't have this problem (recall Figure 119, page 93). You could fix it by scaling the volume to bring the bass back up: but it'd increase the resonant peak amplitude too.

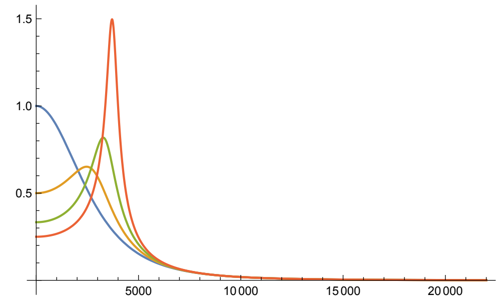



*Figure 125* Bode plot of the amplitude response of a 24dB low pass analog ladder filter with a cutoff at 4000 Hz and resonance values of $k = 0$, $k = 1$, $k = 2$, and $k = 3$. Notice that as $k$ increases, the bass amplitude (near 0 Hz) drops.

The transfer function for $H(s)$ looks pretty simple, but to convert this to the $Z$ domain is kind of a mess because of the power of four in the denominator. (For reminders about $c$ and $\omega_0$, see Section 7.10, page 96). Let's get started:

$$H(s) = \frac{1}{(\frac{s}{\omega_0} + 1)^4 + k}$$

$$H(z) = \frac{1}{\left(\frac{1}{\omega_0} c \frac{z-1}{z+1} + 1\right)^4 + k}$$

Now substitute $d = \frac{1}{\omega_0} c = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

$$= \frac{1}{(d\frac{z-1}{z+1} + 1)^4 + k}$$

*Oh boy, here we go...*

$$H(z) = \frac{1+4z+6z^2+4z^3+z^4}{(1-4d+6d^2-4d^3+d^4+k)+(4-8d+8d^3-4d^4+4k)z+(6-12d^2+6d^4+6k)z^2+(4+8d-8d^3-4d^4+4k)z^3+(1+4d+6d^2+4d^3+d^4+k)z^4} \times \frac{z^{-4}}{z^{-4}}$$

$$= \frac{z^{-4}+4z^{-3}+6z^{-2}+4z^{-1}+1}{(1-4d+6d^2-4d^3+d^4+k)z^{-4}+(4-8d+8d^3-4d^4+4k)z^{-3}+(6-12d^2+6d^4+6k)z^{-2}+(4+8d-8d^3-4d^4+4k)z^{-1}+(1+4d+6d^2+4d^3+d^4+k)}$$

Now set $J = 1 + 4d + 6d^2 + 4d^3 + d^4 + k$ and multiply by $\frac{1/J}{1/J}$

$$= \frac{1/J\times z^{-4}+1/J\times 4z^{-3}+1/J\times 6z^{-2}+1/J\times 4z^{-1}+1/J}{1/J\times(1-4d+6d^2-4d^3+d^4+k)z^{-4}+1/J\times(4-8d+8d^3-4d^4+4k)z^{-3}+1/J\times(6-12d^2+6d^4+6k)z^{-2}+1/J\times(4+8d-8d^3-4d^4+4k)z^{-1}+1}$$

Yuck! Thus we have the following coefficients:

| *Coefficient* | *Value* | *Because it's multiplied by...* |
|---|---|---|
| $b_0$ | $1/J$ | $z^0 (= 1)$ |
| $b_1$ | $1/J \times 4$ | $z^{-1}$ |
| $b_2$ | $1/J \times 6$ | $z^{-2}$ |
| $b_3$ | $1/J \times 4$ | $z^{-3}$ |
| $b_4$ | $1/J$ | $z^{-4}$ |
| $a_1$ | $1/J \times (4 + 8d - 8d^3 - 4d^4 + 4k)$ | $z^{-1}$ |
| $a_2$ | $1/J \times (6 - 12d^2 + 6d^4 + 6k)$ | $z^{-2}$ |
| $a_3$ | $1/J \times (4 - 8d + 8d^3 - 4d^4 + 4k)$ | $z^{-3}$ |
| $a_4$ | $1/J \times (1 - 4d + 6d^2 - 4d^3 + d^4 + k)$ | $z^{-4}$ |

where $J = 1 + 4d + 6d^2 + 4d^3 + d^4 + k$ and $d = \frac{\cos(\omega_0 T/2)}{\sin(\omega_0 T/2)}$

### 7.13 Formant Filters

*Figure 126* 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 among the three 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 $f_1, f_2, ...$, ordered from lowest frequency to highest. Each formant looks very much like a resonant **band pass filter**: it has a *frequency*, a *peak amplitude*, and a **bandwidth** (the width of the filter spread, normally handled via filter resonance).[98] Different vowels are formed

[98] Bandwidth is the difference in frequency between the peak and the point where the amplitude has dropped by 3dB. For a 2-pole bandpass filter, the resonance $Q$ = formant peak frequency / bandwidth.

by changing the shape of the tract (especially the mouth and tongue), and so each vowel has its own set of formants with their own frequencies, amplitudes, and bandwidths.[99] For similar reasons formants also vary depending on the age or sex of the speaker, among other factors.

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, as shown in Figure 127. A **diphthong**[100] 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. Done right, the input to the filters would be drawn from detailed model of the vocal cords, but a sawtooth or square wave might work in a pinch.

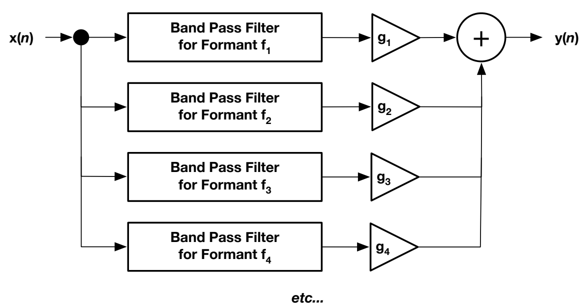



*Figure 127* Simulating formants by composing multiple resonant band pass filters in parallel, multiplied by gains and then summed.

Formants in the human vocal tract don't just affect vowels, but the whole gamut of speech sounds. For example, sounds such as "s" or "t" or "ch" or "f" can be modeled by pushing a noise source through a formant filter. And formants aren't just for human speech: many resonating cavities in musical instruments create formant effects and so formants can be used to model them.

**Formant Synthesizers** Models of the vocal tract are largely the domain of specialized speech synthesizer software and hardware. There exists well-regarded software to model human singing, such as Yamaha's **Vocaloid**, but most *music* synthesizers with formant filters tend to use them as an auxiliary effect to make an oscillator sound vaguely "human". For example, the **Kawai K5** (discussed in Section 3.5) included a fixed formant filter at the end of its pipeline.

There is one well-known exception in hardware: the **Yamaha FS1R**. The FS1R was a full-throated attempt to incorporate formant filters as a central feature of its pipeline in order to generate a wide range of unusual sounds, including models of instruments, though obviously the human voice was a particular specialty. To this end, the FS1R coupled formant filters to a sophisticated version of **Frequency Modulation** (or **FM**) synthesis. Yamaha branded its formant filter architecture **formant synthesis**.

*Figure 128* Yamaha FS1R

FM will be introduced later in Section 8. For purposes of discussion here, the FS1R sported eight oscillators organized as FM **operators** (Section 8.3), some or all of which could be arranged to output sound. The sound-producing oscillators were each pushed through their own dedicated bandpass filters and then summed up. These eight bandpass filters could be customized per-oscillator in many ways: but if the oscillators all made the same sound, the bandpass filters acted as an up to eight-formant filter on the resulting output, producing a vowel. This would be effectively the same result as in Figure 127.

[99] You'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

[100] A 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".

To this the FS1R added eight more “unvoiced operators”: these were noise generators, and they too could be pushed through their own individual bandpass filters and amplifiers with envelopes, then summed up, resulting in consonants. The formant frequencies, pitches, and amplitudes of the various voiced and unvoiced operators could then be controlled via a special **sequencer** to produce words and phrases. Though it was well known for its voices, the FS1R’s FM + formants architecture was capable of a broad range of synthesized musical sounds.

# 8 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.[101]

*Figure 129* Yamaha DX7.©61 (Repeat of Figure 3).

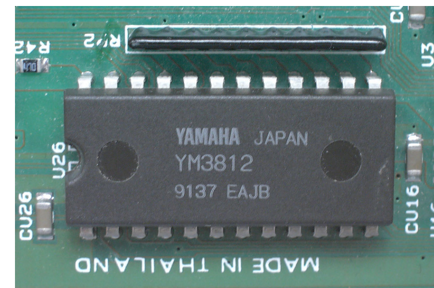



*Figure 130* Yamaha YM3812 chip.©62

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 commercially successful FM synthesizers, largely from Yamaha, which defined much of the sound of pop music in the 1980s and 1990s. Along with the DX7, the **Yamaha TX81Z** rackmount synthesizer notably 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 130) 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.

## 8.1 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.[102] Both phase and frequency modulation are subsets of a general category of modulation methods called **angle modulation.**[103] Phase modulation is easier to explain, so we'll begin with that.

[101]The 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

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

[103]Which for obvious reasons *cannot* be abbreviated "AM".

**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 $\phi_c$:

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

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 $m'(t)$.

Theoretically the modulator function could be anything, but they are very often oscillating functions. For example, it would be reasonable for $m(t)$ to be another sine wave,[104] that is, $m(t) = a_m \sin(f_m t)$. As a result, the instantaneous frequency would be $f_c + a_m \sin'(f_m t) = f_c + a_m f_m \cos(f_m t)$. Imagine if $f_m$ was small: this would effectively be a **low frequency oscillator** and it would slowly push the instantaneous frequency of the carrier up and down, inducing **vibrato**. But if $f_m$ was large, this would cause the frequency of the carrier to change radically before it even had a chance to complete one period. This would create an entirely different-sounding wave. Depending on the value of $a_m$ we could control how *much* of an effect $m(t)$ would have on the carrier. Thus while $f_m$ would dictate the nature of the wave that the carrier was being mutated into, $a_m$ would could be used to largely specify the degree to which the mutation occurred.

**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 of frequency modulation, *we're changing the frequency by changing the phase* (and vice versa).

[104]In fact, this is a very common scenario in many FM synthesizers, so it's hardly far fetched!

**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 $m(...)$. Carrier $c$ will be modulated by $m(t) = a_m \sin(\phi_m + f_m t)$. In PM, we'd have

$$y(t) = a_c \sin(\phi_c + f_c t + m(t)) = a_c \sin(\phi_c + f_c t + a_m \sin(\phi_m + f_m t)) \tag{4}$$

In FM, let's again modulate 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

$$\begin{aligned}\int_0^t f_c + a_m \sin(\phi_m + f_m x) dx &= f_c t + \frac{a_m}{f_m}\left(\cos(\phi_m) - \cos(\phi_m + f_m t)\right)\\ &= f_c t + \frac{a_m}{f_m}\cos(\phi_m) - \frac{a_m}{f_m}\cos(\phi_m + f_m t)\end{aligned}$$

$\frac{a_m}{f_m}\cos(\phi_m)$ is just a constant. Let's call it $D$. So plugging this into the carrier, we have

$$\begin{aligned}y(t) &= a_c \sin\left(\phi_c + f_c t + D - \frac{a_m}{f_m}\cos(\phi_m + f_m t)\right)\\ &= a_c \sin\left(\phi_c + D + f_c t + \frac{a_m}{f_m}\sin(\phi_m - \frac{\pi}{2} + f_m t)\right)\end{aligned} \tag{5}$$

Note how similar this equation is to the phase modulation equation, Equation 4. They differ in just a constant phase ($\phi_c$ versus $\phi_c + D$ and $\phi_m$ 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 can 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 (or, er, phase) 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 an increase in one octave, which is a *doubling* of frequency.[105] 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.

[105]This is also discussed in Footnote 166.

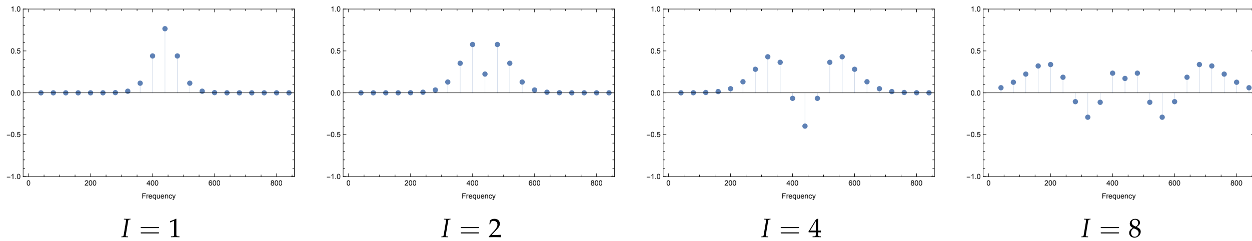



*Figure 131* Change and spread of sidebands with increasing index of modulation ($I$). In all four of these graphs, $f_c = 440$ Hz and $f_m = 40$ Hz. As $I$ increases, the spread (hence bandwidth) of sidebands does as well; and the pattern of sideband amplitude, including the carrier at the center, changes. Negative amplitudes just mean a positive amplitude but shifted in phase by $\pi$.

## 8.2 Sidebands, Bessel Functions, and Reflection

Just as was the case in amplitude modulation and ring modulation (see Section 6.6), frequency modulation and phase modulation produce additional partials called **sidebands**. In FM and PM, when both the carrier and modulator are sine waves, many sidebands spread out symmetrically from 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 $\alpha = 0, 1, 2, ....$ 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 $y(t) = a_c \sin(f_c t + I \times \sin(f_m t))$ if we assumed that 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.

**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 so we can assume that there are only $\lfloor I + 1 \rfloor$ significant sidebands on each side.[106] 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 $4000 + (4000 \times 16) \times (1 + 1) = 132,000$ Hz. Yuck, that will produce considerable aliasing. 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 9.7). As an extreme example, imagine that we were sampling at 480 KHz (!) With a Nyquist frequency of 240,000 Hz, this is big enough to handle a sideband at 132,000 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.

[106] Or $\lceil I + 1 \rceil$, whatever. Carson'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.

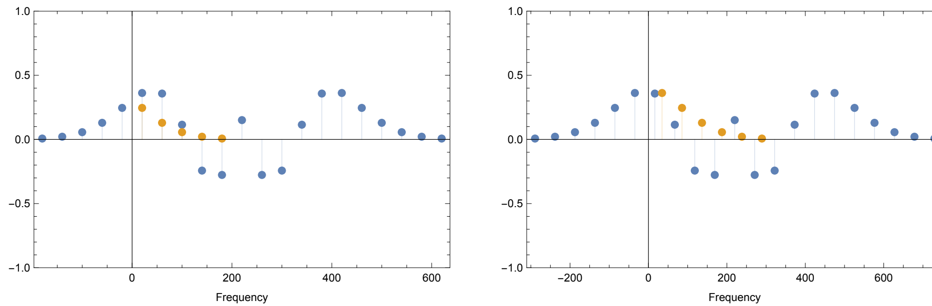



*Figure 132* Sideband reflection in FM, with a 220Hz carrier. *(Left Figure)* 40Hz Modulator: reflected negative sidebands line up with positive sidebands, and the result sounds tonal (though with a much lower fundamental than the original carrier). *(Right Figure)* $40 \times \frac{4}{\pi}$Hz Modulator: reflected negative sidebands do not line up and the sound is atonal.

**Bessel Functions** The index of modulation also comes into play in determining the amplitude of each of the individual sidebands. The specific amplitudes are tricky, 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 $\geq 0$, is the **order** of the function. Figure 133 shows the first eight orders, that is, $J_0(x), ..., J_7(x)$.

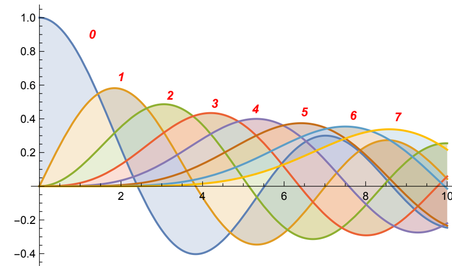



*Figure 133* First eight orders of Bessel functions of the First Kind, positive values only (orders labeled in red).

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 134 shows the amplitude of various sidebands, and the carrier (sideband 0), for different modulation index ($I$) values. Figure 131 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.[107]

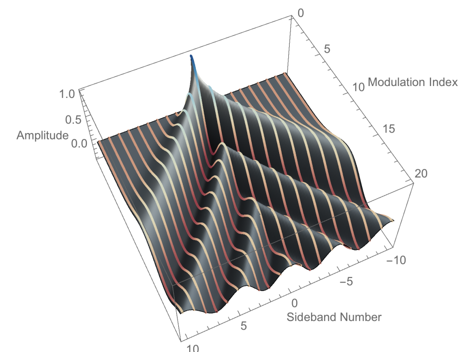



*Figure 134* Sideband amplitudes by modulation index. Sideband numbers are integers shown as colored stripes. Note that this surface drops below zero in places.

**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. Because $f_m = f_c$, 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.

[107] Don't be put off by a negative amplitude. That's just a positive amplitude with a phase that's shifted by $\pi$.

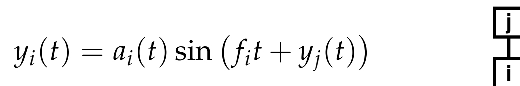
This is the case for any $f_m$ that is an integer multiple of $f_c$ with $f_m \geq f_c$. Another interesting situation occurs when $f_c$ is an integer multiple of $f_m$ and $f_m < f_c$. The reflected sidebands once again match up and things are tonal, but $f_c$ is no longer the fundamental, because there were one or more sidebands appearing to its left before reflection occurred. See Figure 132 (upper) for an illustration of this.[108]

Overall, when $\frac{f_m}{f_c}$ is rational then the positive and reflected negative partials will be integer multiples of *some* fundamental (maybe not $f_c$), and 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 sound will be **inharmonic**. That is, atonal, metallic, brash, or noisy.

## 8.3 Operators and Algorithms

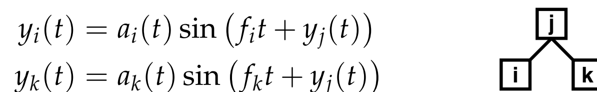
We typically want the effect of a modulator on a carrier to change over time; otherwise the sound would be static and boring. The most common thing to change over time is the amplitude of each oscillator: this is typically done with its own dedicated envelope. Envelopes would thus effect indexes of modulation 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:



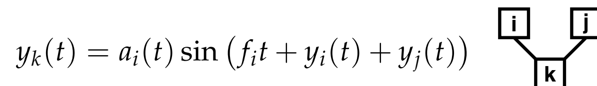

$$y_i(t) = a_i(t)\sin\left(f_i t + y_j(t)\right)$$

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:



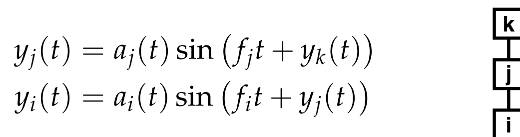

$$y_i(t) = a_i(t)\sin\left(f_i t + y_j(t)\right)$$
$$y_k(t) = a_k(t)\sin\left(f_k t + y_j(t)\right)$$

Now, there's no reason that a carrier couldn't be modified by several modulators at once, with their modulations added up:




$$y_k(t) = a_i(t)\sin\left(f_i t + y_i(t) + y_j(t)\right)$$

... or for an operator to modulate another, while being itself modulated by yet another operator.




$$y_j(t) = a_j(t)\sin\left(f_j t + y_k(t)\right)$$
$$y_i(t) = a_i(t)\sin\left(f_i t + y_j(t)\right)$$

[108]So 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.

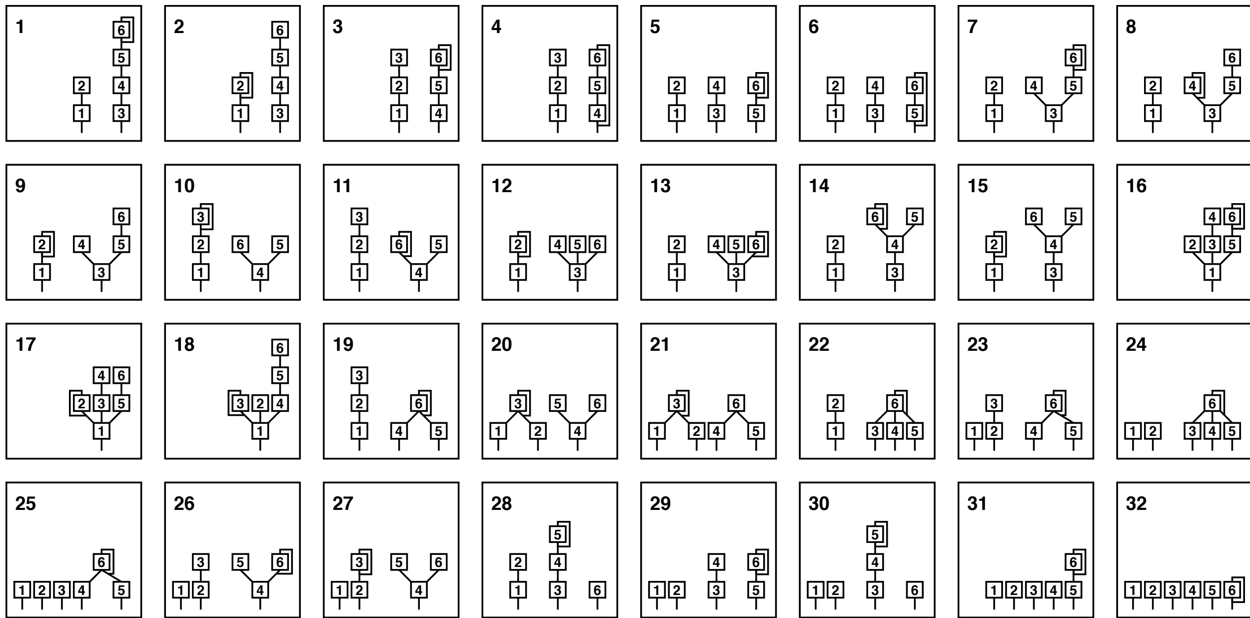


*Figure 135* Operator modulation graphs (so-called "algorithms") for the Yamaha DX7. Operators on the bottom layer (with bare lines coming out from below) are mixed to produce the final sound. Other operators serve only as modulators. Many algorithms sport self-modulating operators, and in a few cases (Algorithms 4 and 6) larger modulation cycles.

... or for an operator to modulate itself...

$$y_i(t) = a_i(t)\sin\left(f_i t + y_i(t-1)\right)$$

... or for there to be a larger cycle in modulation.

$$\begin{aligned} y_k(t) &= a_k(t)\sin\left(f_k t + y_i(t-1)\right) \\ y_j(t) &= a_j(t)\sin\left(f_j t + y_k(t)\right) \\ y_i(t) &= a_i(t)\sin\left(f_i t + y_j(t)\right) \end{aligned}$$

And of course there's no reason why an operator has to be modulated by anyone else, that is,

$$y_i(t) = a_i(t)\sin\left(f_i t\right)$$

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 136). 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 135. 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.

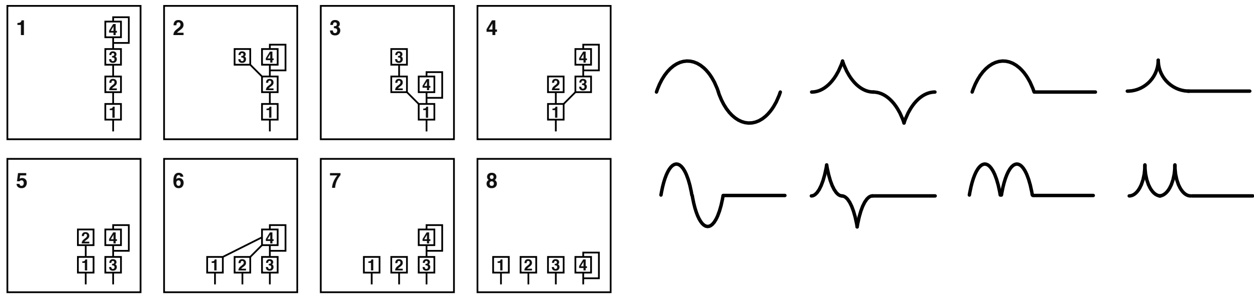


*Figure 137* Algorithms (left) and waveforms[109] (right) of the Yamaha TX81Z. Operators on the bottom layer (which have bare lines coming out from below them) are mixed together to produce the final sound: other operators serve only as modulators. Several algorithms sport self-modulating operators.

The **Yamaha FS1R** (Section 7.13), had eight operators and 88 algorithms; but the large 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**, shown in Figure 138. The TX81Z was a fixture in music studios, and so found its way onto a great many songs in the late 1980s and 1990s.

Figure 137 shows the TX81Z's eight algorithms and eight possible waveforms. 4-operator synthesizers have since become ubiquitous, having had made their way into numerous PC sound cards, toy musical instruments, cell phones, and so on.

*Figure 136* **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.

## 8.4 Implementation

FM is a perfect match for software. But how would you implement it? Recall Equation 1 in the Additive Section, page 28. There we were maintaining the current sine wave phase for some oscillator $i$ as:

*Figure 138* Yamaha TX81Z

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

...where $\Delta t$ 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 Mods($i$). Then for phase modulation we could update the state of the oscillator $x_i$ and

[109]It would appear that the waveforms are largely constructed out of pieces of a sine wave plus silence, and Yamaha's documentation would suggest exactly this. In fact the actual waves are slightly different from this: notably, the "peaky" waves are more rounded and less peaky. But you get the idea.

its final output $y_i$ as:

$$x_i^{(t)} \leftarrow x_i^{(t-1)} + f_i \Delta t \mod 1$$
$$y_i^{(t)} \leftarrow \sin\left(2\pi \times \left(x_i^{(t)} + b_i \times \sum_{j \in \text{Mods}(i)} y_j^{(t-1)}\right)\right) \times a_i^{(t)}$$

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 $b_i$ that 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.[110] Anyway, if you don't care about this, just set $b_i = 1$.

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 (ideally differentiable) wave you deem appropriate: but sine has a strong tradition and theory regarding the resulting sidebands (and what antialiasing they will require). Many FM synthesizers aren't much more than this. Both the DX7 and TX81Z, as well as most other Yamaha-style FM synths, lacked a filter[111] and sported just a single LFO which could modify pitch and volume.

**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 $y_i^{(t)} \leftarrow \sin\left(x_i + y_i^{(t-1)}\right) \times a_i$, or to do similar cyclic modulations, without changing the fundamental pitch of $y_i$.

*Figure 139* PreenFM2.

[110]Traditional Yamaha-style FM synthesizers don't have a $b_i$. Instead, 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 139.

[111]There are exceptions. For example, the **Elektron Digitone** has both FM synthesis and filters, as does the **Yamaha FS1R**, and certain virtual analog synths with FM options.

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 139).

**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.

# 9 Sampling and Sample Playback

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 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.

## 9.1 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.[112] 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.

*Figure 140* A 1999 Mellotron Mk VI. This version is digital, not using the classic Mellotron tape loop.©63

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 **DAC**s and **ADC**s, 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, including **Yamaha**, **Roland**, and **Korg**.

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 141* Akai MPC Renaissance sampling drum machine.©64

[112] In 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, but were not samplers as they could not record sounds. Instead, a rompler held 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 used in a great many songs: indeed the **Korg M1**, a rompler, may be the best-selling synthesizer in history (at 250,000 units).

Romplers were very often rackmount units, 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-instrument song from a single rompler controlled by a computer and keyboard. Most romplers had poor programming interfaces, as they were largely meant to fill a market demand for preset sound devices. As computers became more powerful, samplers and romplers were displaced by **digital audio workstations** which could do the same sampling and playback routines entirely within the computer itself.

## 9.2 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. Often a rompler patch would consist of a one-shot PCM wave for an initial splash — a so-called **transient** — followed by a continuous single-cycle wave playing as long as the key is held down.[113] 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 single-cycle wave (perhaps generated originally via additive synthesis), and then downsample it as necessary to produce a wave at the desired pitch. For implementations, see Algorithms 19 (page 130) and 21 (page 131) for one-shot and looped PCM playback, and 22 (page 131) for single-cycle waves.

Romplers have long been derided as little more than sample-playback devices, and in fact many were. But there also were many novel rompler approaches taken by synthesizer manufacturers to capitalize on the unique opportunities afforded by PCM. For example, the **Korg Wavestation** allowed you to cross-fade between many PCM sounds over time using complex envelopes. For example, one might start with a trumpet sound, then quickly fade into a guitar sound, then fade into silence, then into a sequence of drum sounds. This approach was known as a **wave sequence**.[114]

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 **E-mu Systems Morpheus** and **UltraProteus** in particular were capable of changing filter parameters in real time from among almost 300 complex filters.

[113]This 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**.

[114]The Wavestation has an interesting backstory. After the Sequential Circuits went bankrupt, its founder **Dave Smith**, developed the Wavestation at Korg using the same **vector synthesis** (Section 6.6) approach taken by the **Sequential Circuits Prophet VS**. But unlike the VS, the Wavestation was a rompler: and it added wave sequences to the vector synthesis mix. Smith also expanded on vector synthesis for the **Yamaha SY/TG Series** synthesizers, where one vector synthesis dimension was cross-faded between two FM oscillators, while the other was cross-faded between two PCM sounds.

## 9.3 Wavetable Synthesis

Another rather different use of single-cycle waves is in the form of **wavetables**.[115] A wavetable is nothing more than array $W = \langle w_1, w_2, ...w_n \rangle$ of digitized single cycle waves. Figure 142 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. For an implementation, see Algorithm 23 (page 132).

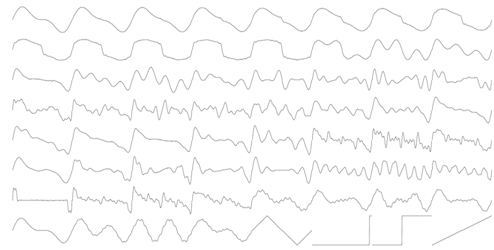


*Figure 142* Wavetable #31 of the PPG Wave synthesizer, with 64 single-cycle waves. Most waves move smoothly from one to another, but the last four do not: these are triangle, pulse, square, and sawtooth, and appear in PPG and (minus pulse) Waldorf wavetables for programming convenience.©65

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 143 shows the front panel of the **Microwave XT** and its envelope controls.

*Figure 143* Waldorf Microwave XT (rare "Shadow" version: most were safety orange!). Note the bottom right quadrant of the knob array, devoted entirely to envelopes.

It might interest you to know that wavetables have historically been stored in one of two forms. As memory is plentiful nowadays, wavetables are now 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 in memory, 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. Figure 144 shows a sparse wavetable example from the Microwave XT.

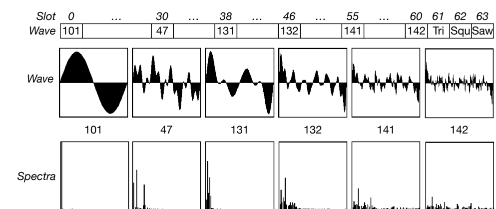



*Figure 144* First 61 slots of the Waldorf Microwave XT's wavetable #3, "MalletSyn", a sparse array of empty slots interspersed with references to six single-cycle waves, plus the obligatory Triangle, Square, and Sawtooth.©66

[115]Note that many in the music synthesis community, myself included, use the term *wavetable* differently than its much later unfortunate 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*, some in 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 synthesizers from many manufacturers, 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.[116] 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 $i(m) = \langle \cos(\frac{2\pi}{m}), \sin(\frac{2\pi}{m}) \rangle$. As the modulation went from 0 to 1, then $i(m)$ would trace out a circle in the space. Assuming $i(0) = i(1)$, as was the case in this example, we could further use a sawtooth LFO to repeatedly trace out this path forever as an **orbit**.

## 9.4 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".

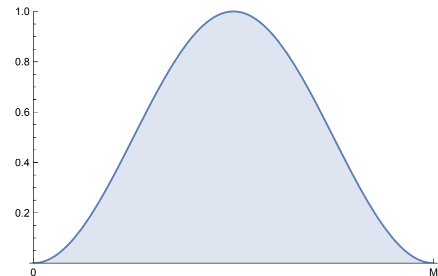



*Figure 145* Hann window.

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 multiplied against a **window** function (per Section 12.5) 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 down, per Figure 189) or **Trapezoidal** (ramp up, hold steady, ramp down) windows, but as computer power increased, more elaborate windows became possible. One popular window nowadays, at least for small grains, is the **Hann window**, discussed in Section 12.6. For an illustration, see Figure 145.

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.[117] 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)[118] 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**.

[116] A 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).

[117] These categories are co-opted out of the five categories described by **Curtis Roads** in Curtis Roads, 2004, *Microsound*, The MIT Press.

[118] Does this sound like a **wave sequence** (Section 9.2, page 118)? It does to me. I suppose the difference is that the sounds in a wave sequence can be long (over 100ms) whereas grains are usually 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.

*Figure 146* Tasty Chips Electronics GR-1.[©67]

Granular synthesis is uncommon. Most granular synthesizers are software; hardware granular synths are rare, especially polyphonic ones. One exception is the **Tasty Chips Electronics GR-1**, an asynchronous granular synth shown in Figure 146. Other recent examples include the **Waldorf Quantum** and the **Mutable Synthesis Clouds** module. For a simple implementation of granular playback, see Algorithm 20 (page 130).

## 9.5 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♭, 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: a person speaking on the tape is pitched higher but speaks much faster. The more difficult alternative (without introducing noticeable artifacts in the sound) is **pitch scaling**, where the pitch is adjusted but the length is kept this same. Many samplers and romplers just 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. For example, to shift a sound up by one octave, that is, to twice its frequency, 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 down one octave, to half its frequency, 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.

Audio requires real-time, high-quality resampling. The right way to do this is via real-time **interpolation**: estimating the original real-valued signal function using our samples, then gathering new samples from that. We'll do interpolation in Section 9.6 coming up. But to introduce the issue, let's start with classic (but flawed) *bulk* resampling methods: bulk downsampling and upsampling.

It's worth mentioning that, unless you're wiling to use great deal of computer power, resampling and interpolation algorithms will all introduce some degree of aliasing, and the problem gets worse

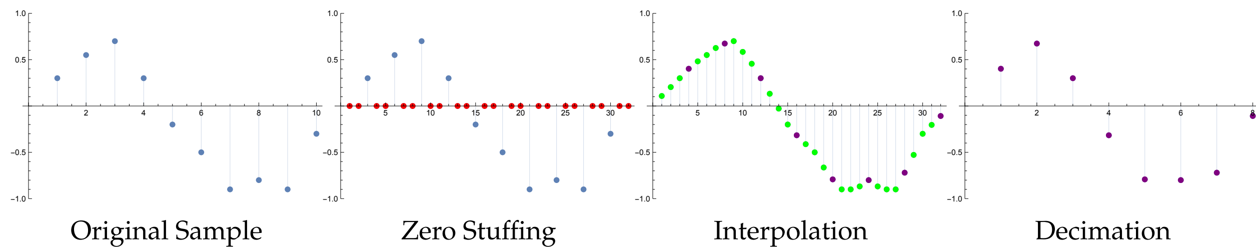



*Figure 147* Resampling a sound to 3/4 its previous sampling rate. The sound is is first stuffed with two zeros per sample. The result is smoothed over with a low-pass filter to interpolate the zeros between the original samples (and to prevent frequencies over the final Nyquist limit). Then the sound is decimated to remove all but every fourth sample.

the further you pitch-shift. Thus historically many digital synthesizers contained not one but several copies of the same sample (or similar ones) pre-shifted to different pitches, spaced from one another by perhaps three octaves or so. This allowed playability all along the keyboard without significant distortion at the extreme ends.

**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 $N$th 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 first apply a **low pass filter** to 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.

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 signal, 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: thus this process is called **zero stuffing**. 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 reduce the overall gain of the sound, so we will 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 147 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♯, 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 downsampling by 84! Now imagine a 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.[119] 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 messy and 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. It's just not going to work. We need a method which can do interpolation in floating point, and one where we can change sample rates dynamically, in real time, and without computing large fractions.

### 9.6 Basic Real-Time Interpolation

Consider instead the following very simple (and very bad) approach. Given a sound $A = \langle a_0, ... \rangle$ at our desired sampling rate and at a pitch represented by frequency $F_A$, we want to generate a sound $B = \langle b_0, ... \rangle$ which is $A$ shifted to a different pitch, hence a different frequency $F_B$. To do this, instead of moving through $A$ one step (sample) at a time, we'll move forward $\frac{F_B}{F_A}$ (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 $x^{t+1} \leftarrow x^t + \frac{F_B}{F_A}$. 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 $\frac{R_A}{2} \times \frac{\min(F_B, F_A)}{F_A}$ in the original sound (where $R_A$ is the original sound's sampling rate).

The problem with this method is that $\frac{F_B}{F_A}$ 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 $\frac{F_B}{F_A}$ 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 $\langle a_0 = f(x_0), a_1 = f(x_1), ..., a_n = f(x_n) \rangle$ at sample positions $x_1, x_2, ..., x_n$.

The simplest approach would be 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 $\langle x_h, f(x_h) \rangle$ inclusive. Linear interpolation is fine for some problems, but it's not great in audio: its first

[119] Obviously you couldn't do that with $\frac{89}{84}$, because 89 is prime.

derivative is discontinuous at the sample points, as is also the case for its generalization to higher polynomials, **Lagrange interpolation**.[120]

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 they avoid another problem with Lagrange interpolation, namely unwanted oscillation. One simple spline approach is **cubic interpolation**. Let's say we had four points $\langle x_1, f(x_1)\rangle, ..., \langle x_4, f(x_4)\rangle$ where the four $x_i$ are evenly spaced from each other and $x_1 < x_2 < x_3 < x_4$. 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 $\alpha$ be how far $x$ is relative to $x_2$ and $x_3$, that is, $\alpha = (x - x_2)/(x_3 - x_2)$. Then

$$\begin{aligned} f(x) = &\alpha^3(-f(x_1) + f(x_2) - f(x_3) + f(x_4)) \\ &+ \alpha^2(2f(x_1) - 2f(x_2) + f(x_3) - f(x_4)) \\ &+ \alpha(-f(x_1) + f(x_3)) \\ &+ f(x_2) \end{aligned}$$

A variation, based on the **Catmull-Rom** cubic spline, uses successive differences in $f(...)$ to estimate the first derivative for a potentially smoother interpolation.[121]

$$\begin{aligned} f(x) = &\alpha^3(-1/2f(x_1) + 3/2f(x_2) - 3/2f(x_3) + 1/2f(x_4)) \\ &+ \alpha^2(f(x_1) - 5/2f(x_2) + 2f(x_3) - 1/2f(x_4)) \\ &+ \alpha(-1/2f(x_1) + 1/2f(x_3)) \\ &+ f(x_2) \end{aligned} \tag{6}$$

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 functions they produce are *not quite* the actual band-limited function which passes through these points. You'll still get distortion. And you still ought to filter beforehand when downsampling to eliminate aliasing.[122] But it turns out that there exists a method which will, at its limit, interpolate along the *actual* band-limited function, and act as a built-in brick wall antialiasing filter to boot. This method is **windowed sinc interpolation**.

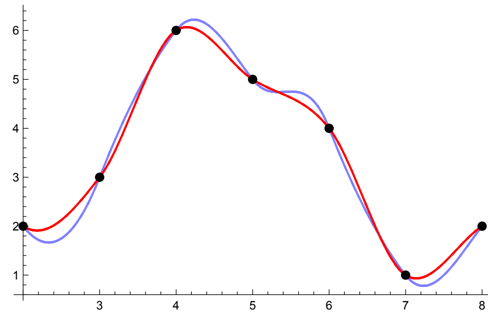



*Figure 148* Basic cubic spine interpolation and Catmull-Rom interpolation. Note that Catmull-Rom is less "bouncy". That may or may not be desirable (compare to Figure 150).

[120]Named 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 sample other points between them. To start, note that with a little elbow grease we can rearrange the aforementioned linear interpolation equation into $f(x) = f(x_h)\frac{x-x_l}{x_h-x_l} + f(x_l)\frac{x-x_h}{x_l-x_h}$. 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 $x_1...x_n$ 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 $x_1, ..., x_4$ 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 $x_3$.

[121]Smoother isn't necessarily better: the original sound is probably somewhat overshooting: see Figure 150, page 125.

[122]There 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

## 9.7 Windowed Sinc Interpolation

The **sinc function**, sometimes called the **cardinal sine function** or **sampling function**, is shown in Figure 149. It extends from positive to negative infinity. Sinc is:[123]

$$\text{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} & x \neq 0 \\ 1 & x = 0 \end{cases}$$

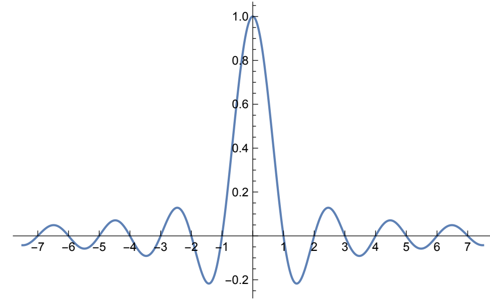


*Figure 149* Sinc function.

**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, by applying the **Whittaker-Shannon interpolation formula**, you can use sinc to exactly reconstruct this continuous signal from your digital samples.[124]

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 $A = \langle a_{-\infty}, ..., a_0, ..., a_\infty \rangle$ from $C(t)$ sampled at a rate of $R_A$. The timestep for sample $a_k$ is thus $k/R_A$. For each such sample $a_k$, we center a sinc function (scaled by $R_A$) over its timestep, and multiply it by $a_k$. $C(...)$ is just the sum of all these sincs. That is:

$$C(t) = \sum_{k=-\infty}^{\infty} \text{sinc}(R_A \times (t - k/R_A)) \times a_k$$

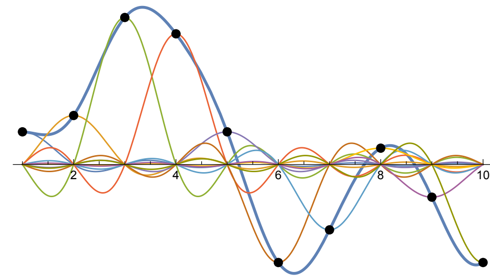


*Figure 150* Convolution with Sinc. For each sample point, a sinc function is centered at that point and scaled vertically by the sample value at that point. The sinc functions are added up, and the resulting function is the bandlimited reconstruction of the original signal.

This is **convolving** the sinc function against $A$, as shown in Figure 150. But notice that, because sinc is symmetric around zero, $\text{sinc}(R_A \times (t - k/R_A)) = \text{sinc}(R_A \times (k/R_A - t))$. This means we could instead write things as a **correlation** rather than convolution procedure:

$$\begin{aligned} C(t) &= \sum_{k=-\infty}^{\infty} \text{sinc}(R_A \times (k/R_A - t)) \times a_k \\ &= \sum_{k=-\infty}^{\infty} \text{sinc}(k - t \times R_A) \times a_k \end{aligned}$$

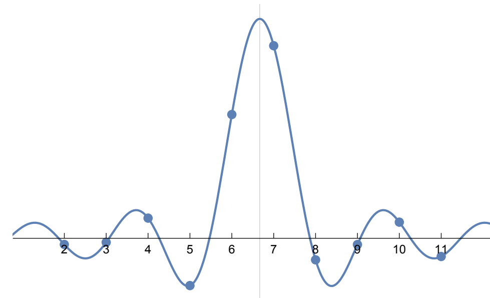


*Figure 151* Sinc coefficients (five on a side) for computing a sample at time $t = 6\,2/3$. Unwindowed.

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 $t$, and scaled by $R_A$. Then, for each sample $a_k$, we sample a *coefficient* from our sinc at position $k$ (Figure 151). Each of these coefficients is then multiplied by

[123] Sinc is pronounced "sink", and is a contraction of *sinus cardinalis*, (cardinal sine). There are two definitions of sinc, with and without the appearance of $\pi$. Sampling uses the one with $\pi$ (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, page 66.

[124] If Windowed Sinc Interpolation can be used to perfectly reproduce the original continuous signal, why bother with other interpolation methods? Because Windowed Sinc Interpolation is very costly, involving many calls to sin( ).

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 7.

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 for timesteps $t$ which correspond to our digital samples, $C(t)$ describes a function which passes through exactly those 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 $b_j \in B$, where $B$ is our sound at the new sampling rate $R_B$, the timestep $t$ of $b_j$ is $t = j/R_B$, so we get:

$$b_j = \sum_{k=-\infty}^{\infty} \text{sinc}(R_A \times (k/R_A - j/R_B)) \times a_k \tag{7}$$

Using this equation we can now identify the sample values at positions $j$ in $B$.

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 turns out that convolution with sinc isn't just an interpolation function: it's also a **brick wall filter** (in theory at least, when we're summing from $-\infty$ to $\infty$). 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 cast in the frequency domain, sinc *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 $R_A$ in Equation 7 ensures a filter cutoff at $R_A/2$, 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 $R_A$ with $\min(R_A, R_B)$ like this:

$$b_j = \sum_{k=-\infty}^{\infty} \text{sinc}(\min(R_A, R_B) \times (k/R_A - j/R_B)) \times a_k$$

This will also change the overall volume, so to keep it a **unity gain filter**, we need to scale it back again by $\min(1, R_B/R_A)$:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}(\min(R_A, R_B) \times (k/R_A - j/R_B)) \times a_k$$

To simplify things later, let's pull a $1/R_A$ out of the $(k/R_A - j/R_B)$, like this:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(R_A,R_B)}{R_A} \times (k - R_A \times j/R_B)\right) \times a_k$$

Now let's define $J = R_A \times j/R_B$. That is, $J$ is the real-valued location of the new sample $b_j$ in the coordinate system of the original samples in $A$. This is perhaps more intuitive if we rearrange $J$ to be $J = j \times R_A/R_B$. Anyway, $J$ is the spot about which the sinc function is centered (for example, $J = 6\ 2/3$ in Figure 151), as is obvious when we substitute $J$ into the equation:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(R_A,R_B)}{R_A} \times (k - J)\right) \times a_k$$

**Windowing** Of course, we don't have an infinite number of samples in our set $A$: at best we have $A = \langle a_0, ..., a_{n-1} \rangle$, and even *that* could be a huge 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.

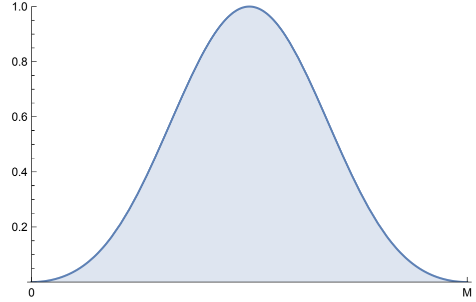



*Figure 152* Blackman Window. For more windows, see Section 12.5.

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 bad). To do this, we can multiply it against a **window**. Windows were introduced in Section 12.5. We'd like a window which drops completely to zero, such as the **Hann window** (Figure 194). Another option, which we'll use here, is the **Blackman window**, shown in Figure 152.[125] As usual, the Blackman window runs from 0 ... $M$ inclusive, and we'll set values outside this range to 0. When discretized, this is $M + 1$ integers (the so-called "length" of the window).[126] An odd "length" (thus an even $M$) is desirable because in the trivial situation where $R_B$ is a multiple of $R_A$, the center of the window will be exactly above some sample in $a \in A$ and an odd-length window will be balanced exactly on both sides of $a$. The Blackman window is:

$$w(m, M) = \begin{cases} 0.42 - \frac{1}{2}\cos\left(\frac{2\pi m}{M}\right) + 0.08\cos\left(\frac{4\pi m}{M}\right) & m \in [0 \dots M] \\ 0 & \text{otherwise} \end{cases}$$

Armed with a window, we could now replace the sinc with a *windowed sinc* which tapers off at $J \pm M/2$, using the window centered at $J$ like sinc was. Like most window functions, Blackman is centered around $\frac{M}{2}$, so to recenter it around $J$, we have:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=-\infty}^{\infty} \text{sinc}\left(\frac{\min(R_A,R_B)}{R_A} \times (k - J)\right) \times w(k - J + \frac{M}{2}, M) \times a_k$$

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, $k_{low} = \lceil J - \frac{M}{2} \rceil$ and $k_{high} = \lfloor J + \frac{M}{2} \rfloor$, thus:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=k_{low}}^{k_{high}} \text{sinc}\left(\frac{\min(R_A,R_B)}{R_A} \times (k - J)\right) \times w(k - J + \frac{M}{2}, M) \times a_k$$

... where $J = j \times R_A/R_B$. How big should $M$ be? It's a trade-off of sound quality for computational power. Some early samplers employing Windowed Sinc Interpolation used as little as $M = 8$. It's gone up since then.[127] For starters, you might try $M = 16$ and adjust from there.[128]

[125]In the literature you'll see better windows still, notably the **Kaiser window**, but they are difficult to describe here and even tougher to implement (Kaiser is based on **Bessel functions**).

[126]I put "length" in quotes because, for real values, the length of a window region 0...$M$ is obviously just $M$, not $M + 1$.

[127]Some samplers didn't use Windowed Sinc Interpolation at all, but relied on cheaper interpolation methods such as splines. If you're low on computer power, you could try something like that.

[128]CCRMA has an 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

A large value of $M$ will also impact on the **latency** of the algorithm: because sinc is not a **causal filter**[129] we must know some of the future incoming sound samples. For basic sampling this is probably not an issue, as we probably already have the entire PCM sample or the wavetable available to us. But if you were using this method to do (say) pitch-shifting of incoming sounds in real-time 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.

## 9.8 Implementation

Pitch shifting is fundamental to a great many sound-generation methods in music synthesis, such as all those discussed so far in Section 9. This includes one-shot and looped PCM sound playback, granular synthesis, playing single-cycle waves, and wavetables. In this Section we first put together the Windowed Sinc Interpolation algorithm laid out previously, then use it to build playback functions for these methods one by one. They're little more than syntactic sugar, but it might be useful to understand how they're done.

**Windowed Sinc Interpolation Algorithm** Before we start with this algorithm, there's one minor detail that needs to be handled first. Namely, when you're near the start or end of your sample, then the above equation will request sample values $a_k$ outside of $A = \{a_0, a_1, ..., a_{n-1}\}$. What should your $a_k$ "value" be then? It depends on what your original sound $A$ is:

- If your sound is *one-shot*, such as a piano sample, then just zero-pad. That is, if the equation asks for $a_k$, where $k \geq n$ or $k < 0$, just return 0. (To do this right, ideally the sound ought to start and end at 0 amplitude and have a **DC offset** of 0).
- If your sound is a *loop*, such as a single cycle in a wavetable, then just wrap it around. The common case is where your sound is longer than your window (that is, $n \geq M$). In this case, you need only wrap once: if the algorithm asks for $a_k$, where $k \geq n$, just return $a_{k-n}$. And if $k < 0$ then just use $a_{k+n}$.[130] For the rare case ($n < M$), just use $a_{k \bmod n}$, though it's costly. The mod operation on negative numbers is handled differently in different programming languages: we want it to return a positive value. In Java you can do that as: `k = k % n` when $k \geq n$, but when $k < 0$, use the more general `k = k % n; if (k < 0) k = k + n;`

We can abstract these and other options by replacing $a_k$ with an "extraction function" $e(A, k)$ which extracts and returns the correct sample depending on your criteria:

$$b_j = \min(1, R_B/R_A) \times \sum_{k=k_{low}}^{k_{high}} \text{sinc}\left(\frac{\min(R_A, R_B)}{R_A} \times (k - J)\right) \times w(k - J + \frac{M}{2}, M) \times e(A, k)$$

Here are simple extraction functions for the aforementioned one-shot and looped scenarios. For simplicity, we assume that the looping goes back to 0 rather than a designated loop point.

[129] A causal filter is one which relies only on the past sound samples, not future ones. All the filters in Section 7 were causal filters because the output $y(n)$ relied solely on the current input $x(n)$ and the *past* input $x(n-1), x(n-2), ...$ and output $y(n-1), y(n-2), ...$ to the filter. But sinc relies in part on *future* samples $x(n+1), x(n+2), ....$ The only way to do that is to buffer up those samples so they're available when sinc is running.

[130] If you're just starting the sound, then perhaps you might wish to treat it as one-shot (zero-pad) with regard to values of $k < 0$ until you've looped once. This assumes of course that your first sample starts near 0.

**Algorithm 16** *One-Shot Extraction*

1: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples
2: $k \leftarrow$ sample position we wish to return ▷ May be outside the bounds $0...n-1$

3: **if** $k < 0$ or $k \geq n$ **then**
4: **return** 0 ▷ Zero-pad
5: **else**
6: **return** $a_k$

**Algorithm 17** *Looped Extraction*

1: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples
2: $k \leftarrow$ sample position we wish to return ▷ May be outside the bounds $0...n-1$

3: **if** $k < 0$ **then**
4: $k \leftarrow k + n$ ▷ Wrap around once
5: **if** $k < 0$ **then** ▷ If it's *really* far out of bounds...
6: $k \leftarrow k \mod n$ ▷ Wrap around fully. In Java: `k = (k % n); if (k < 0) k += n;`
7: **else if** $k \geq n$ **then**
8: $k \leftarrow k - n$ ▷ Wrap around once
9: **if** $k \geq n$ **then** ▷ If it's *really* far out of bounds...
10: $k \leftarrow k \mod n$ ▷ Wrap around fully. In Java: `k = (k % n)`
11: **return** $a_k$

Armed with these, we can now define our full function:

**Algorithm 18** *Windowed Sinc Interpolation*

1: $R_A \leftarrow$ original sampling rate
2: $R_B \leftarrow$ new sampling rate
3: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples in the original sampling rate $R_A$
4: $j \leftarrow$ sample position (starting at 0) in the new sampling rate which we wish to compute
5: $M \leftarrow$ window range ▷ Window will be from $0...M$ inclusive. Best if $M$ was even
6: $w(m, M) \leftarrow$ window ▷ Blackman: $w(m, M) \leftarrow \begin{cases} 0.42 - \frac{1}{2}\cos(\frac{2\pi m}{M}) + 0.08\cos(\frac{4\pi m}{M}) & m \in 0...M \\ 0 & \text{otherwise} \end{cases}$
7: $e(A, k) \leftarrow$ extraction function ▷ Might vary depending on if we're looping or one-shot, etc.

8: $J \leftarrow j \times R_A / R_B$ ▷ Location of new sample position $j$ among the old samples $a_0...a_{n-1}$
9: $k_{low} \leftarrow \lceil J - M/2 \rceil$ ▷ Low and high bounds for the window
10: $k_{high} \leftarrow \lfloor J + M/2 \rfloor$
11: $f \leftarrow \frac{\min(R_A, R_B)}{R_A}$ ▷ Scaling sinc to avoid aliasing
12: $b_j \leftarrow \min(1, R_B/R_A) \times \sum_{k=k_{low}}^{k_{high}} \text{sinc}\left(f \times (k - J)\right) \times w(k - J + \frac{M}{2}, M) \times e(A, k)$
13: **return** $b_j$

The general use is as follows. You have a set of $n$ samples $A = \{a_0, a_1, ..., a_{n-1}\}$ with a sampling rate $R_A$. You want to build a new set of samples $B = \{b_0, b_1, ...\}$ with a different sampling rate $R_B$. Simply perform Windowed Sinc Interpolation for a given value $j$ to compute each individual $b_j$.

Now let's go on to some example algorithms using Windowed Sinc Interpolation for our previous discussed methods. The goal of all of these algorithms is to shift the pitch of a sound from an original frequency $F_A$ to a new frequency $F_B$, both in Hz. We'll assume that the sampling rate $R_A$ is the same for both the original and final sound. Thus we compute $R_B \leftarrow R_A \times F_A / F_B$.

**One-Shot PCM Sound Playback** Per Section 9.2, suppose that the musician has struck a note and you wish to play a one-shot PCM sound, pitch-shifted to that note. To do this you repeatedly compute and play samples, starting at $j = 0$ and incrementing $j$, until you no longer want to play the sound, or until you have exhausted the original samples. You'll exhaust them when $j \times F_A / F_B > n_1$ (your sound $A$ is $n$ samples long), at which point the samples will all be zeros. We'll assume a fixed and predefined window size $M$ and window function $w(m, M)$.

**Algorithm 19** *Pitch Shifting for One-Shot Playback*

1: $R_A \leftarrow$ sampling rate of original (and new) samples
2: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples in the original sampling rate $R_A$
3: $F_A \leftarrow$ frequency (in Hz) of the pitch of the original samples
4: $F_B \leftarrow$ frequency (in Hz) of the pitch of the new sample
5: $j \leftarrow$ sample position (starting at 0) of the new sample we wish to compute

6: $R_B \leftarrow R_A \times F_A / F_B$
7: $M \leftarrow$ window range ▷ 8 or 16, say
8: $w(m, M) \leftarrow$ window ▷ Blackman: $w(m,M) \leftarrow \begin{cases} 0.42 - \frac{1}{2}\cos(\frac{2\pi m}{M}) + 0.08\cos(\frac{4\pi m}{M}) & m \in 0...M \\ 0 & \text{otherwise} \end{cases}$
9: $e(A, k) \leftarrow$ One-Shot Extraction function ▷ Algorithm 16
10: $b_j \leftarrow$ Windowed Sinc Interpolation with $R_A$, $R_B$, $A$, $j$, $M$, $w(...)$, and $e(...)$ ▷ Algorithm 18
11: **return** $b_j$

**Granular Playback** Granular synthesis was covered in Section 9.4. Algorithm 19 can also be used to play the samples for a generated grain. You will need to specify the length of the grain and the grain windowing function (perhaps the **Hann window**). You will also need to add an offset so the grain being played starts at the right place in the original samples.

**Algorithm 20** *Grain Playback*

1: $R_A \leftarrow$ sampling rate of original (and new) samples
2: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples in the original sampling rate $R_A$
3: $F_A \leftarrow$ frequency (in Hz) of the pitch of the original samples
4: $F_B \leftarrow$ frequency (in Hz) of the pitch of the new sample
5: $q \leftarrow$ grain offset in original samples, $0 \leq q \leq n - 1$
6: $M \leftarrow$ grain length
7: $w(m, M) \leftarrow$ grain window ▷ Hann is common: $w(m,M) \leftarrow \begin{cases} 1/2 - 1/2\cos(2\pi m/M) & m \in 0...M \\ 0 & \text{otherwise} \end{cases}$
8: $j \leftarrow$ sample position (starting at 0) of the new sample we wish to compute

9: $A' \leftarrow \{a_q, a_{q+1}, ..., a_{n-1}\}$ subset of $A$ starting at offset $q$
10: $b_j \leftarrow$ Pitch Shifting for One-Shot Playback with $R_A$, $A'$, $F_A$, $F_B$, and $j$ ▷ Algorithm 19
11: **return** $b_j \times w(j - q, M)$

It's possible you might wish the grain length, like the offset, to be measured relative to the original samples rather than the new ones, as $M'$. Then $M \leftarrow M' \times R_A / R_B$, computed like $J$ was.

**Looped PCM Sound Playback** Again, per Section 9.2, let's say that you have a repeating sound loop, rather than a one-shot PCM sample, that you want to play when the musician strikes a note. Unlike the one-shot scenario discussed earlier, you'll never run out of samples to play because it's just wrapped around. As you can see, the function below differs from Algorithm 19 only in that it uses a different extraction function. Again, we'll assume a fixed and predefined window size $M$ and window function $w(m, M)$.

**Algorithm 21** *Pitch Shifting for Looped Playback*

1: $R_A \leftarrow$ sampling rate of original (and new) samples
2: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples in the original sampling rate $R_A$
3: $F_A \leftarrow$ frequency (in Hz) of the pitch of the original samples
4: $F_B \leftarrow$ frequency (in Hz) of the pitch of the new sample
5: $j \leftarrow$ sample position (starting at 0) of the new sample we wish to compute

6: $R_B \leftarrow R_A \times F_A / F_B$
7: $M \leftarrow$ window range ▷ 8 or 16, say
8: $w(m, M) \leftarrow$ window ▷ Blackman: $w(m, M) \leftarrow \begin{cases} 0.42 - \frac{1}{2}\cos(\frac{2\pi m}{M}) + 0.08\cos(\frac{4\pi m}{M}) & m \in 0...M \\ 0 & \text{otherwise} \end{cases}$
9: $e(A, k) \leftarrow$ Looped Extraction function ▷ Algorithm 17
10: $b_j \leftarrow$ Windowed Sinc Interpolation with $R_A$, $R_B$, $A$, $j$, $M$, $w(...)$, and $e(...)$ ▷ Algorithm 18
11: **return** $b_j$

**Single-Cycle Wave Playback** In Section 9.2 (and Section 6.2) it was mentioned that a very common way of generating a bandlimited sawtooth, square, triangle, or other wave[131] was to first store a single band-limited cycle of it in very high quality (a high sampling rate, thus large). This could be generated using additive synthesis. Then we'd pitch-shift the cycle as necessary to play the sound. This is simply Pitch Shifting for Looped Playback with a fixed $F_A$, since the original pitch is determined by the the single cycle's length and the given sampling rate.

**Algorithm 22** *Looped Single-Cycle Wave Playback*

1: $R_A \leftarrow$ sampling rate of original (and new) samples
2: $A \leftarrow \{a_0, a_1, ..., a_{n-1}\}$ samples in the original sampling rate $R_A$
3: $F_B \leftarrow$ frequency (in Hz) of the pitch of the new sample
4: $j \leftarrow$ sample position (starting at 0) of the new sample we wish to compute

5: $F_A \leftarrow R_A / n$ ▷ $n$ is the number of samples in $A$
6: $b_j \leftarrow$ Pitch Shifting for Looped Playback with $R_A$, $A$, $F_A$, $F_B$, and $j$ ▷ Algorithm 21
7: **return** $b_j$

[131] Looking for interesting single-cycle waves? **Kristoffer Ekstrand**, under the stage name **Adventure Kid**, has released an enormous collection of them for free. See https://www.adventurekid.se

**Wavetable Playback** Section 9.3 discussed **wavetables**, which are arrays of single-cycle waves. At any time you have a value from a modulation signal between 0 and 1 which determines which wave, or interpolation between two waves, should be played. Once you have extracted the relevant wave or wave combination, all you need to do is apply Looped Single-Cycle Wave Playback to it to play it at the right pitch.[132]

**Algorithm 23** *Simple Wavetable Playback*

1: $R_A \leftarrow$ sampling rate of original (and new) samples
2: $\mathcal{A} \leftarrow \{A_0, ..., A_{p-1}\}$ wavetable of $p$ single-cycle waves, each $n$ samples long, of sampling rate $R_A$
3: $F_B \leftarrow$ frequency (in Hz) of the pitch of the new sample
4: $j \leftarrow$ sample position (starting at 0) of the new sample we wish to compute
5: $x \leftarrow$ wavetable modulation value, ranging from 0 to 1 inclusive

6: $w \leftarrow x \times (p-1)$
7: $\alpha \leftarrow w - \lfloor w \rfloor$
8: **if** $\alpha = 0$ **then** ▷ $w$ is an integer, so pure wave, no interpolation
9: $b_j \leftarrow$ Looped Single-Cycle Wave Playback with $R_A$, $A_w$, $F_B$, and $j$ ▷ Algorithm 22
10: **else** ▷ Linearly interpolate between a wave and the next one
11: $b_j^{(0)} \leftarrow$ Looped Single-Cycle Wave Playback with $R_A$, $A_{\lfloor w \rfloor}$, $F_B$, and $j$ ▷ Algorithm 22
12: $b_j^{(1)} \leftarrow$ Looped Single-Cycle Wave Playback with $R_A$, $A_{\lfloor w \rfloor + 1}$, $F_B$, and $j$ ▷ Algorithm 22
13: $b_j \leftarrow (1-\alpha) \times b_j^{(0)} + \alpha \times b_j^{(1)}$
14: **return** $b_j$

[132] For simplicity, this algorithm resamples both cycles and then interpolates them. This is twice as expensive as you'd need. You could instead figure out which original samples will be accessed during resampling, then interpolate those samples first, and pass the result a single time into the resampler.

# 10 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.

## 10.1 Delays

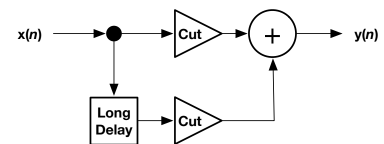



*Figure 153* One-shot delay. Compare to the FIR filter in Figure 107.

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 153 The delay portion, commonly known as a **digital delay line**. If you recall from Section 7.8, 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 24** *Delay Line*

1: $x \leftarrow$ incoming sample

2: Global $B \leftarrow \langle b_0, ..., b_{m-1} \rangle$ 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: $\quad p = 0$
9: **return** $y$

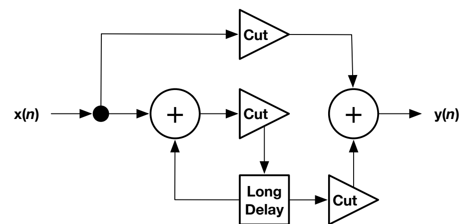



*Figure 154* Repeated delay, augmented with two additional cut gains to control wetness. Compare to the basic IIR filter in Figure 108.

Note from Figure 153 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.

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 154 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.

## 10.2 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.

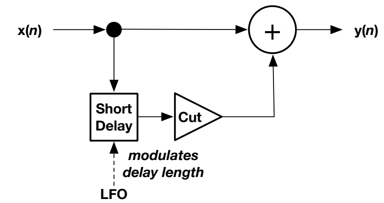



*Figure 155* LFO-modulated flanger

**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 slightly longer delay: it takes the form

$$y(n) = b_0 x(n) + b_m x(n-m) \tag{8}$$

where $m$ is the length of the delay in samples. We'll assume that $b_0 = 1$. Notice the repeated lobes in the amplitude response of the comb filter in Figure 156. A larger value of $m$ will result in more of these lobes.[133] You can also see how setting $b_m$ to different values changes the wetness of the filter.

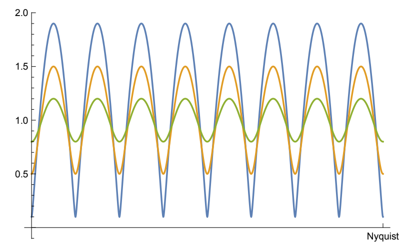



*Figure 156* Amplitude response of a forward comb filter with $b_0 = 1$ and $b_m$ set to -0.9, -0.5, and -0.2.

A comb filter is most easily described in the Z Domain where, with $b_0 = 1$, its transfer function is

$$H(z) = 1 + b_m z^{-m} = \frac{z^m + b_m}{z^m}$$

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.[134] It is this even spacing which creates the lobes in the amplitude response:

$$|H(e^{i\omega})| = \sqrt{(1-b_m)^2 + 2b_m \cos(\omega m)}$$

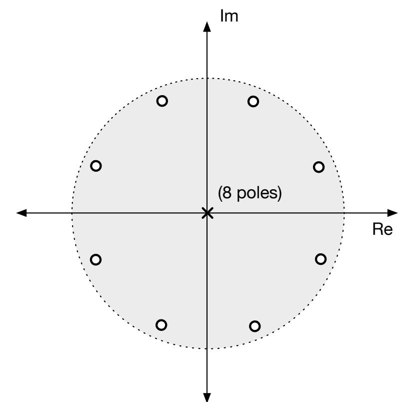



*Figure 157* Poles and zeros of a forward comb filter, $m = 8, b_m = -0.5$, in the Z domain.

[133] Indeed, if $m = 1$, then we have a standard low-pass or high-pass filter.

[134] You 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.

Notice that the forward comb filter is literally nothing more than a short delay line, and so any basic delay line of length $m$ is just an FIR filter of the form $y(n) = b_0 x(n) + b_m x(n-m)$. There's nothing more to it.

The **feedback comb filter**, which is the extended equivalent to a basic IIR filter, is just as simple. It takes the form

$$y(n) = b_0 x(n) + a_1 y(n-m)$$

Again, we may assume that $b_0 = 1$, and so the transfer function, in the Z Domain, is just

$$H(z) = \frac{1}{1 - a_1 z^{-m}} = \frac{z^m}{z^m - a_1}$$

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 amplitude 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 158.

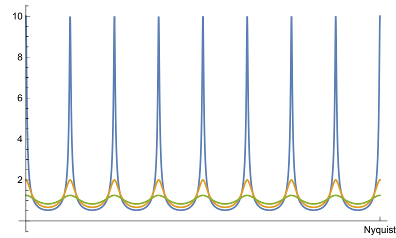



*Figure 158* Amplitude response of a feedback comb filter with $b_0 = 1$ and $a_1$ set to -0.9, -0.5, and -0.2.

**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, $\alpha$ is a value between 0 and 1 which describes where $m$ is with respect to the integers on either side of it. Let's presume we have a delay at least length $\lceil m \rceil$. Now we could modify Equation 8 to roll in a bit of each of the samples on either side of $m$, that is:

$$y(n) = b_0 x(n) + (1-\alpha) b_m x(n - \lfloor m \rfloor) + \alpha b_m x(n - \lceil m \rceil)$$

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 9.6. 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 10.4.

## 10.3 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 25** *Multi-Tap Delay Line*

1: $x \leftarrow$ incoming sample
2: $A \leftarrow \langle a_0, ..., a_{q-1} \rangle$ tap positions, each from 0 to $n-1$ ▷ $q \ll n$

3: Global $B \leftarrow \langle b_0, ..., b_{n-1} \rangle$ buffer (array of $n$ samples), initially all 0
4: Global $p \leftarrow$ position in buffer, initially 0
5: $T \leftarrow \langle t_0, ..., t_{q-1} \rangle$ results (array of $q$ samples)
6: **for** $i = 0$ to $q-1$ **do** ▷ Load taps
7: $\quad j \leftarrow p + a_i \mod n$
8: $\quad T_i \leftarrow B_j$
9: $b_p \leftarrow x$ ▷ Update sample as was done in Delay Line (Algorithm 24)
10: $p \leftarrow p + 1$
11: **if** $p \geq n$ **then**
12: $\quad p = 0$
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.[135] 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.

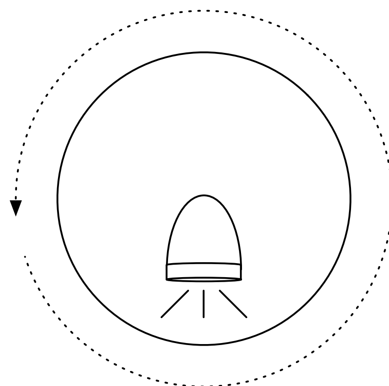


*Figure 159* Leslie speaker mounted on a spinning turntable. As the speaker rotates away or toward the listener, the sound wobbles in pitch (vibrato) due to the Doppler Effect, and the sound is alternately blaring or muffled (when facing away), creating tremolo.

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.

[135] Yes, that's basically **FM**.

### 10.4 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 traveled 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 different 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 different partials in the signal. Importantly, this delay can be *very* small, even less than a single sample, and different frequencies can be (and are) delayed by *different amounts*.

A trivial all-pass filter is just a delay, that is, $y(n) = x(n - m)$, but this is not particularly interesting as it changes the phase linearly, that is, by $-d\omega$ for a given angular frequency $\omega$. We're interested in more complex all-pass filters. It turns out that a general IIR filter consisting of both IIR and FIR components (recall Figure 111 on page 83) can be made into an all-pass filter if the IIR and FIR components *cancel each other out*. This nullifies the magnitude-changing effects of the filter, but interestingly does not nullify the phase-changing effects. For example, a filter of the form

$$y(n) = b_0 x(n) + x(n-1) - a_1 y(n-1)$$

is an all-pass filter if $a_1 = b_0$. Why? Because the transfer function of this filter is

$$H(z) = \frac{b_0 + z^{-1}}{1 + a_1 z^{-1}} = \frac{b_0 + 1 \times z^{-1}}{1 + b_0 z^{-1}}$$

Notice that the coefficients of the polynomial in the denominator are the "reverse", so to speak, of those in the numerator.[136] The coefficients in the numerator are $b_0$ and 1, whereas the coefficients in the denominator are 1 and $b_0$. This reversing pattern is what causes the FIR and IIR to cancel their magnitudes. The pattern continues for more detailed polynomials. In general, you can make an all-pass filter with the transfer function

$$H(z) = \frac{b_0 + b_1 z^{-1} ... + b_{m-1} z^{-(m-1)} + z^{-m}}{1 + b_m z^{-1} + ... + b_1 z^{-(m-1)} + b_0 z^{-m}}$$

[136]This is only the case for real-valued filters, such as in audio. For complex-valued filters, the pattern is somewhat more... complex. The denominator coefficients must also consist of complex conjugates of those in the numerator.

One common pattern is to generalize our $y(n) = b_0 x(n) + x(n-1) - a_1 y(n-1)$ filter to:

$$y(n) = b_0 x(n) + x(n-m) - b_0 y(n-m)$$

This is just a comb filter in the FIR section mashed together with a comb filter of the same size in the IIR section. It's clearly an all-pass filter as it has the transfer function

$$H(z) = \frac{b_0 + z^{-m}}{1 + b_0 z^{-m}}$$

You can save some computation by rearranging things so that the two comb filters share the same delay, which produces the intertwined comb filter pattern in Figure 160.[137]

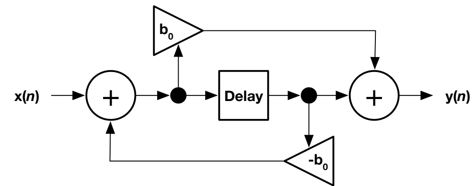



*Figure 160* A simple all-pass filter consisting of two intertwined comb filters. Note that the coefficients are the negatives of one another.

All-pass filters can be strung together in serial or put in parallel, and the result is still an all-pass filter. Also, any shared delay in an all-pass filter can be replaced an all-pass filter, so you can create an all-pass filter out of *nested* all-pass filters. For example, we could nest our intertwined-comb all-pass filter inside another intertwined-comb as shown in Figure 161.

This transfer function has an even, 1.0 amplitude response, and its phase response is 0 degrees at 0 Hz, dropping as the frequency increases.

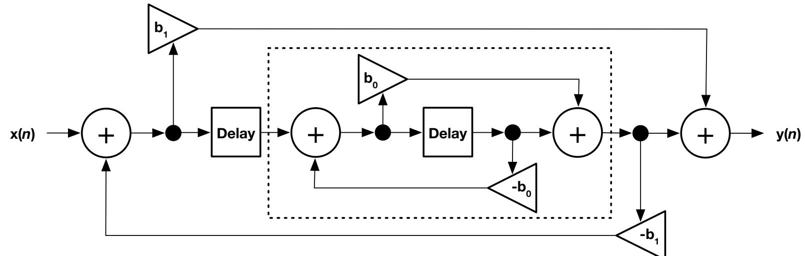



*Figure 161* One all-pass filter nested inside another.

**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 a 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 162, 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.[138]

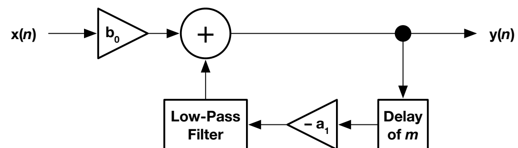



*Figure 162* Low Pass Feedback Comb Filter.

[137]This rearrangement is directly derived from IIR Filter Direct Form II as was shown in Figure 112 on page 83.

[138]I'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

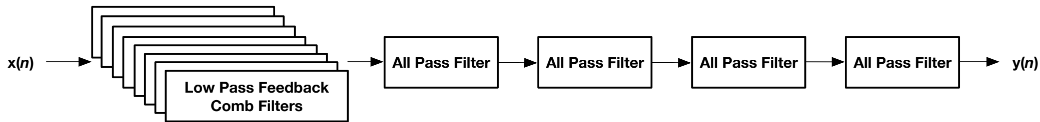



*Figure 163* Freeverb architecture

**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 record 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 7.1: indeed the impulse response is essentially being used as a very long **finite impulse response filter**.[139]

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 **Fast Fourier Transform** (or **FFT**). As discussed in Section 12, the FFT converts a sound from the time domain to the frequency domain, and the **Inverse Fast Fourier Transform** (or **IFFT**) does the opposite. A critical fact here is that *convolution in the time domain is exactly the same thing as multiplication in the frequency domain*. It's much faster to convert the sound and impulse to the frequency domain, multiply them against each other there, and then convert the result back to the time domain.

To start, let's zero-pad the impulse response to be as long as the sound, 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 $S(f)$ — a function over frequency $f$ — 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 IFFT of $R(f)$ to produce the final resulting sound $r(t)$.

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 $O(N \lg N)$, as opposed to the $O(MN)$ required by direct convolution. Clever! But this means we have to apply reverb in bulk to the entire sample.

[139] Now finally it should make sense why FIR is called a finite *impulse response* filter.

That won't do. Instead we could perform FFT-multiply-IFFT trick little-by-little on chunks of the sound via the **Short Time Fourier Transform** or **STFT** (discussed later in Section 12.6). The size of a chunk would determine the degree of delay (latency) in the sample. But by using clever optimizations the delay can be reduced.[140]

## 10.5 Phasers

A phaser is 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 164. Modulating these lobe positions with an LFO produces the **phaser** effect.

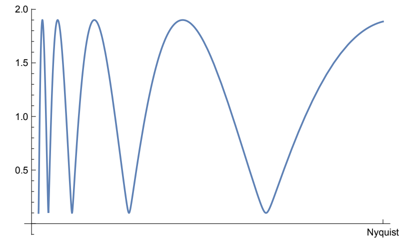



*Figure 164* One possible phaser amplitude response

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 165 shows one possible implementation.

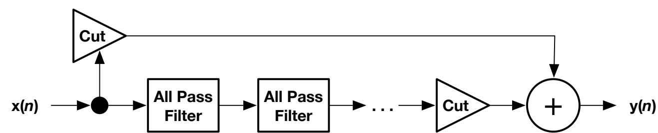



*Figure 165* A phaser implementation.

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.[141]

## 10.6 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:

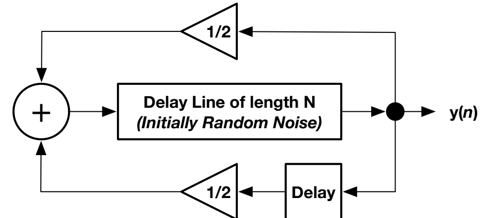



*Figure 166* The Karplus-Strong algorithm.

[140] For a practical introduction, see https://dvcs.w3.org/hg/audio/raw-file/tip/webaudio/convolution.html
[141] For one such implementation, see
https:/ccrma.stanford.edu/realsimple/DelayVar/Phasing_First_Order_Allpass_Filters.html

**Algorithm 26** *Karplus-Strong String Synthesis*

1: Global $B \leftarrow \langle b_0, ..., b_{m-1} \rangle$ 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: $\quad p \leftarrow 0$
9: $y' \leftarrow y$
10: **return** $y$

This should look very familiar: it's closely related to the basic **digital delay line** (Algorithm 24). 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 166. Thus if we have a buffer of length $N$, then Karplus Strong is roughly the equation

$$y(n) = 1/2\, y(n - N) + 1/2\, y(n - N - 1)$$

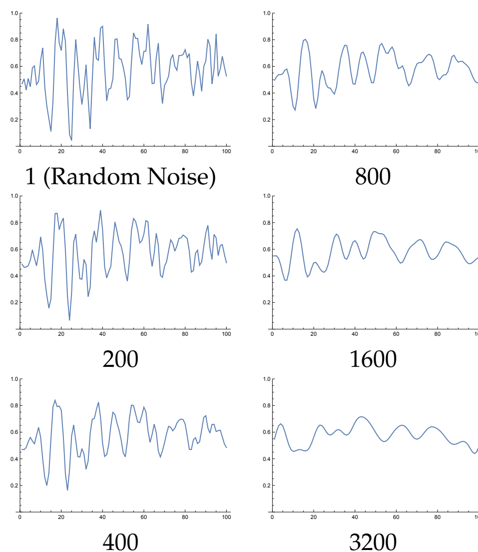



*Figure 167* Karplus-Strong buffer (size 100), after various iterations of the algorithm. Note the gradual impact of the low-pass filter.

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 167 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.[142]

[142] For 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.

**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 168. Karplus-Strong might be viewed as a model of one of these waves as it decays. But 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.

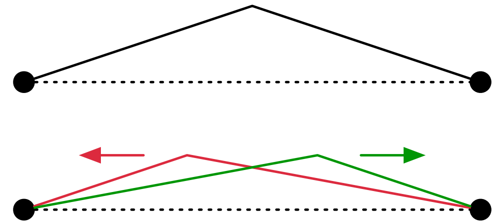


*Figure 168* Traveling waves in a plucked string. (Top) The string is initially plucked, pulled away as shown. (Bottom) The wave separates into two traveling waves going opposite directions, towards the endpoints. After reaching the endpoints, the waves invert and reflect back (then reflect back *again* when reaching the opposite endpoint, and so on).

A bidirectional digital waveguide can be simulated with two multi-tap delay lines as shown in Figure 169. 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 dampened 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.

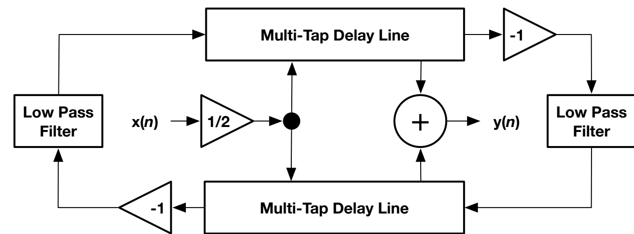



*Figure 169* Bidirectional waveguide model of two traveling waves. A gain of -1, shown twice in this model, means that the signal is inverted.

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.[143]

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 170). Physical modeling

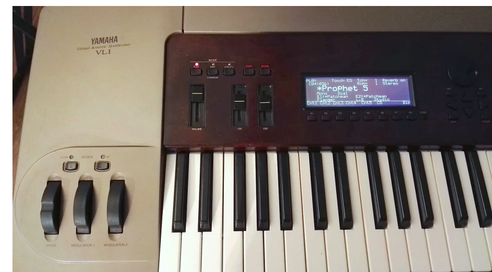



*Figure 170* Yamaha VL1.©68

[143] A 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/

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 programming and 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, largely all softsynths. Among the most successful are **Audio Modeling**'s **SWAM** engine, which combines physical modeling, PCM playback, and some other tricks, to very accurately reproduce woodwinds and strings.

# 11 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 11.2.

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.

## 11.1 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.[144] These instruments all shared something in common: their keys were essentially switches. No matter how hard or in what way 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**,[145] nowadays shortened to **piano**. This instrument hit strings with a felt hammer when a key was played, and importantly 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**[146] 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.

[144] In 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**.

[145] Italian for "soft-loud".

[146] The 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 poor 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 172) 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 usually 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.[147] On a synthesizer, which is typically velocity sensitive, an expression pedal is often used to adjust volume, but may also be used to adjust the timbre of the sound in some other way.

*Figure 171* Alexandra Stepanoff performing with a theremin on NBC Radio, 1930.©69

Early electronic music experimented with a number of other ways to change pitch or timbre. The most famous early electronic instrument, the **theremin**,[148] 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**).[149]

[147]On 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.

[148]Named after its inventor, **Léon Theremin**. The theremin remains the only significant musical instrument that is played without touching it. Its eerie sound has made it a staple in sci-fi movies and television shows, famously ***The Day the Earth Stood Still***. And now for an incredible story. 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. Then he disappeared and the popularity of the theremin collapsed. This was because Stalin has imprisoned him in a Siberian prison-laboratory and forced him to design spy devices for the USSR for 30 years. It was there that he invented **The Thing**, 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.

[149]**D-Beam** was a theremin-inspired controller found on some Roland synthesizers in the 1990s. It measured the distance to one's hand with an infrared beam and served as a modulation option. It's often, and I think unfairly, ridiculed.

Another approach, popularized by the **Trautonium** and similar devices (see Section 5.1), 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 54 on page 52) and the **ASM Hydrasynth**. This strip was put to heavy use by **Vangelis** for his soundtracks (page 52). 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.[150]

*Figure 172* Akai MPK mini MIDI controller, with a self-centering joystick (red), drum pads, assignable knobs, and velocity sensitive mini keys.[©70]

**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.[151]

*Figure 173* Novation Remote Zero SL Mk II, with assignable dials, buttons, sliders, and drum pads.

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 172) 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 SL** and the **Behringer BCR2000**. 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 174). Many electronic and MIDI-based drum kits have

*Figure 174* The Simmons SDS-5.[©71]

[150]Unlike in video games, many synthesizer joysticks are often not self-centering.

[151]For 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 172 and 173).

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 175). 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 175* Wind controller.[©72]

**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 176).

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 176* Novation Launchpad.[©73]

**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 177* Ensoniq SQ80 synthesizer, one of the few synthesizer keyboards with polyphonic aftertouch.[©74]

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.[152] This approach is commonly known as **pressure** or **aftertouch**.

[152]There exists a very rare alternative where the musician could shift a key *sideways*.

Many keyboards implement **channel aftertouch** (a MIDI term: see Section 11.2), 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 177 shows an **Ensoniq SQ80**, one synthesizer which had polyphonic aftertouch. Finally, when the musician releases a key, some keyboards report the **release velocity**.

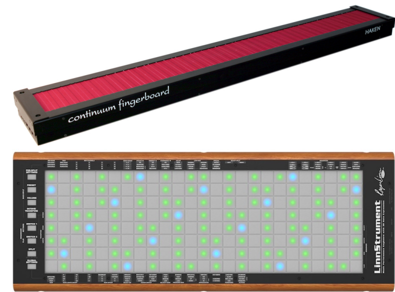



*Figure 178* Haken Continuum Fingerboard (Top)[©75] and Roger Linn Design LinnStrument (Bottom).[©76]

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 178 and 179).

*Figure 179* ROLI Seaboard.[©77]

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*.[153] Related is the **Eigenlabs Eigenharp**, which combines a multidimensional touch-sensitive keyboard, a controller strip, and a breath controller.

## 11.2 MIDI

In 1978 **Dave Smith** (of **Sequential Circuits**) 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.

*Figure 180* Eigenharp.[©78]

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.

[153] Yes, 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 "start playing 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.

### 11.2.1 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 181. 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.

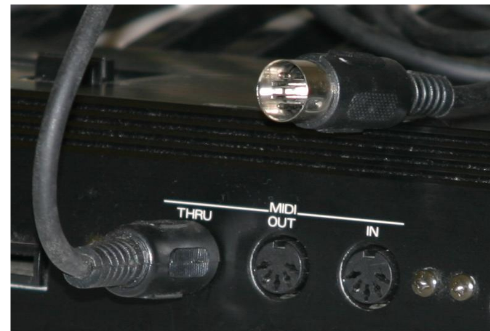



*Figure 181* 5-Pin DIN MIDI Cable and its In, Out, and Thru ports.©79

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 182. And as also shown in that Figure, there's no reason you couldn't mix the two techniques: forwarding D to E for example.

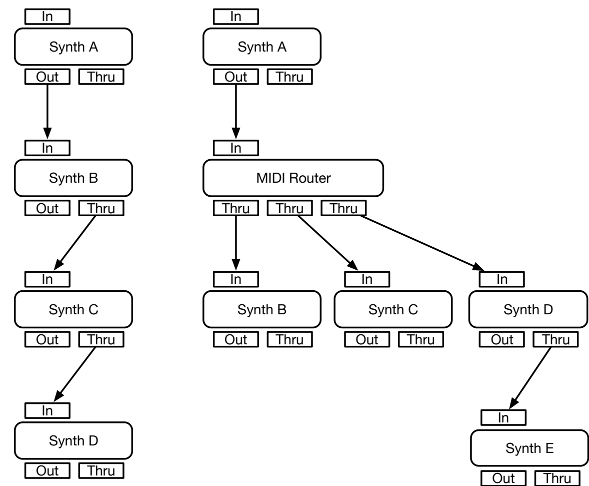



*Figure 182* MIDI Routing Examples

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 would.

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.

**MIDI over USB** Beyond its classic 5-pin DIN serial connection, MIDI has since been run over Ethernet, Firewire, wireless, Bluetooth, fiber-optic cable, you name it. But critically MIDI is now very often run over USB, often to connect a synthesizer or a controller keyboard 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* (normally your computer) with a *client* (your printer, say), and indeed prior to USB-C they have had different shaped ports to enforce which is which. USB devices generally can't be both hosts and clients without separate USB buses. Because USB is so focused on connecting devices to computers, nearly all hardware USB MIDI devices are *clients*. This means that to just connect one device to another (a controller to a synthesizer, say), you must go through a host device — often your laptop. The peer-to-peer connection capability which made MIDI so useful has been lost. 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 communicate 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.

### 11.2.2 Messages

MIDI messages are just sequences of bytes. The first byte in a sequence, called the **status byte**, has its high bit set to 1. The remaining **data bytes** have their high bits set to 0. 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", and we will sometimes do so here as well.

The status byte indicates the type of message, and in some cases is the entire message in and of itself. 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.

**Channels** Some messages (timing messages, sysex, etc.) are broadcast to any and all devices listening. Other messages (like note information) are intended for devices listening in on one of 16 *channels* 1...16 (stored in MIDI as 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 known as responding to the **omni channel**.

**Running Status** It takes three bytes (about 1 ms!) just to tell a synthesizer to start playing a note. That's costly. But recognizing that very often the same kind of message will appear many times in a row, 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 as A 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 7 (the "6").

- **Note On** | `0x9`*ch* | *note* | *velocity* | 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** | `0x8`*ch* | *note* | *release velocity* | 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 as a default. 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** | `0xA`*ch* | *note* | *pressure* | tells a synthesizer that a certain key, currently being held down, is now being pressed *harder* than when first played. This message comes with two data values, both 0...127: the note in question (middle C is 60 or `0x3c`), and the *pressure level*. A level of 0 means the key isn't being pressed harder. 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** `0xD`*ch* | *pressure* tells a synthesizer that *the keyboard as a whole* is now being pressed *harder* than when first played. This message comes with a single data value (0...127): the *pressure level*. A level of 0 is default: it means the key isn't being pressed harder. Many keyboards implement channel pressure. A keyboard wouldn't implement both channel and polyphonic key pressure at the same time; but a synthesizer might respond to both.

- **Program Change** or **PC** `0xC`*ch* | *patch* asks the synthesizer to change to some new patch (0...127). Many synthesizers have more than 128 patches available, so it's common for patches to be arranged in **banks** of up to 128 patches, and thus a PC message may be preceded by a **bank change** request, discussed in Section 11.2.3. A PC message is rarely real-time: many synthesizers take quite a bit of time (milliseconds to seconds) to change to a new patch.

- **Pitch Bend** `0xE`*ch* | *LSB* | *MSB* tells a synthesizer that the pitch bend value has been changed.[154] 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 MSB $\times$ 128 $+$ LSB $-$ 8192.

- **Control Change** or **CC** `0xB`*ch* | *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 11.2.3. Also, 0...127 is not particularly fine-grained: also discussed in Section 11.2.3 are options for sending more precise information.

**Clock Messages** Many music devices, such as drum machines, can play songs or beat patterns 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 for 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.[155] 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.

- **Clock Pulse** or **Timing Clock** `0xF8` Sends a pulse.

- **Start** `0xFA` Informs all devices to reset themselves to the beginning of the song and to prepare to begin playing upon receiving pulses. Start is interpreted as a song position pointer of 0 followed by a Continue.

[154]MSB stands for **Most Significant Byte** and LSB stands for **Least Significant Byte**, even though neither of them is a byte: they're both 7-bit values.

[155]One 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.

- **Stop** `0xFC` Informs all devices to pause (or stop) playing.

- **Continue** `0xFB` Informs all devices to resume playing upon receiving pulses.

- **Song Select** `0xF3` | *song* Informs all devices to prepare to start playing a given *song* (drum-beat pattern, whatnot) from 0...127. This is not often used.

- **Song Position Pointer** `0xF2` | *LSB* | *MSB* 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.

**Sysex** `0xF0` | *id...* | *data...* | `0xF7` Sysex messages are manufacturer-specific, but they are required to have a certain pattern. First comes the status byte `0xF0`.[156] 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 is a namespace scheme, and 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.[157]

Manufacturer IDs `0x7E` and `0x7F` are special: they are so-called *Universal System Exclusive* messages reserved by the MIDI association for its own purposes. Examples of universal system exclusive protocols include standards for sample dumps, device inquiry and response, file dumps, tuning standards, and a variety of real-time transfer protocols.

**Other Stuff** There are several other non-channel messages, none particularly important:[158]

- **MIDI Time Code Quarter Frame** `0xF1` | *data byte* A sequence of these messages send an SMPTE[159] time code stamp. This is an absolute time value (frames, seconds, minutes, etc.) used to synchronize MIDI with video etc. These messages aren't 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 and only occasional 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.

---

[156]It is convention in C/C++, Java, and related languages to describe the hexadecimal number $N$ as `0xN` to distinguish it from decimal numbers. So, for example, the number F0 (that is, 240) is written as `0xF0`.

[157]*Technically* it doesn't need to end with `0xF7` as long as another message immediately follows. But everyone does it.

[158]By the way, the status bytes `0xF4` `0xF4` `0xF9` and `0xFD` are undefined.

[159]SMPTE: Society of Motion Picture and Television Engineers.

### 11.2.3 Control Change (CC) Messages

Control Change (CC) messages (of the form | 0xB*ch* | *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:

- Parameters 0 and/or 32 often select the **patch bank**. Each bank would usually contain up to 128 patches (selected with Program Change).[160]
- Parameter 1 often specifies the value of the **modulation wheel**.
- 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 ($\geq 64$) or not ($< 64$).
- Parameter 74 often specifies the **third dimension controller** specified by **MIDI Polyphonic Expression** (or **MPE**), discussed later in Section 11.2.6.
- Parameters 6, 38, 96, 97, 98, 99, 100, and 101 are often reserved to implement the **NRPN** and **RPN** protocols (see Section 11.2.4, next).
- Parameters 120–123 are reserved for so-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 "local mode" on (127) or off (0). When in local mode, the synthesizer's own keyboard can send notes to the synthesizer. When not in local mode, this connection is broken, 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 voices for all notes. This does *not* immediately cut all sound, as voices may have a long release time in response to a note-off.
- Parameters 124–127 are reserved for additional, now largely useless, standardized **MIDI channel mode messages** which control so-called **omni mode** and **mono** vs. **poly** modes. An

[160]Early synthesizers used Parameter 0 to specify the bank, and so there could be up to 128 banks. Later synthesizers treated Parameter 0 as the MSB and Parameter 32 as the LSB, so the bank value was LSB $\times 128$ + MSB and thus in theory there could be 16834 banks. However, many such synths had far fewer than 128 banks, so only the LSB (Parameter 32) was used in reality! Kind of a mess.

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 not very useful nowadays, this region is nonetheless still (unfortunately) reserved.[161]

- Parameter 124 (with value 0) turns omni mode off.
- Parameter 125 (with value 0) turns omni mode on.
- Parameter 126 (with a value $n$ from 0 to 16) turns mono mode on and poly mode off. You can treat your synthesizer's voices (or controller's outgoing notes) as $n$ individual mono devices. If $n \geq 1$, then the synth is treated as $n$ individual mono synthesizers (or $n$ individual mono controller keyboards), one per channel, starting at the channel on which this message was sent (and not wrapping around). If $n = 0$ then the device is treated as 16 individual mono synthesizers (or controllers), one per MIDI channel.[162]
- Parameter 127 (with value 0) turns poly mode on and mono mode off.

#### 11.2.4 Reserved and Non-Reserved Parameter Number Messages (RPN and NRPN)

CC messages have two serious problems. The first problem is that there are only 120 of them (disregarding the MIDI channel mode region, 120...127). 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.

Early on, the MIDI spec tried to deal with the second problem by optionally 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, MSB $\times$ 128 + LSB. Unfortunately, there would be 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 proprietary 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 synthesizer manufacturers to do with as they wish.

RPN and NRPN work as follows. For NRPN, a controller begins by sending CC Parameter 99 and CC Parameter 98, which 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 99 and 3 for Parameter 98 ($2 \times 128 + 3 = 259$). For RPN, these CC parameters would be 101 and 100 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

[161] Why 124 and 125 aren't merged, and similarly 126 and 127, I have no idea. Also, all four of these messages are also supposed to trigger an all-notes-off event (see Parameter 123 on page 155).

[162] This sounds a lot like MPE to me (Section 11.2.6): it's not clear to me why MPE wasn't built off of this facility.

CC 96 with a value of 5 would mean that the parameter should be incremented by 5. Most synths ignore the increment value and just increment (or decrement) by 1.

Inspired by Running Status, a stream of these value-messages (6, 32, 96, or 97) could be sent without sending the NRPN/RPN Parameter Number 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.[163]

The problem with RPN and NRPN is that they are slow: to update the value of a new parameter requires 4 CC messages.[164] Another problem with RPN and NRPN is that only some synthesizers implement them, and even more problematically, many lazy Digital Audio Workstation manufacturers do not bother to include them as options.

### 11.2.5 Challenges

MIDI has been remarkably stable since it was invented in 1983: indeed, the spec is still technically fixed to 1.0![165] 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:

- **MIDI is slow.** Notoriously so. MIDI was fixed to 31,250 bits per second to support early synthesizers with 1MHz CPUs (31,250 $\times$ 32 = 1 million). This is not fast enough to guarantee smooth transitions beyond the ability of 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.[166]

- **MIDI is a one-direction protocol.** There's no standard way to query devices for their capabilities, or to negotiate to use a more advanced version of the protocol, etc.

---

[163] Actually no value at all need be provided for RPN Null to do its job.

[164] This is not as bad as it sounds. Let's imagine that you wanted to send a single NRPN message. You'd send a CC 99, then a CC 98, then maybe a CC 6 and a CC 32. Normally these CC messages would be 3 bytes long each, but recall that they are *all* CC messages, so you can take advantage of running status. So you'd have 3 + 2 + 2 + 2 bytes total, just 9 bytes. Every time you wanted to send another message of the same parameter but with different values — perhaps you were doing a filter sweep — you'd probably just send one more CC 6 or CC 32 (thus just another 2 bytes). If you wanted to add an RPN Null at the end, you'd send a CC 100 and CC 101 (no need to send the data portion). This would be 4 more bytes.

[165] This is kind of a lie. MIDI 1.0 in 1983 is fairly different from the MIDI 1.0 of the 2000s. But the MIDI Association has never updated the version number. That's finally changing soon though, with MIDI 2.0.

[166] CV/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 31 (page 46) 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.

- **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).

But this situation may change soon with many new MIDI protocol features. The last of these problems is dealt with by a new extension to MIDI called **MIDI Polyphonic Expression** or **MPE**. The remaining three problems will be tackled by the upcoming **MIDI 2.0**. We discuss these below.

#### 11.2.6 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, per-key, its note, the the velocity with which it is struck, the velocity with which it is released, and (using polyphonic key pressure) the pressure with it is being pressed. All other parameters (CC, channel pressure, NRPN, and especially pitch bend) are global to the whole instrument rather than being per-key.

But many other 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. In the same vein, it is the goal of many current advanced MIDI controllers to enable a musician to change a many independent parameters on a per-note basis in order to increase the expressivity by which he may play. But MIDI simply doesn't permit this.

*Figure 183* The Futuresonus Parva, the first hardware synthesizer[167] to support MPE.[©80]

To deal with this situation, many high-parameter controller manufacturers support a new MIDI standard which provides CC, pitch bend, and so on on a per-key basis. This allows these manufacturers to provide (at least) a "five-dimensional" control surface (attack velocity and note pitch, aftertouch,[168] pitch bend, release velocity, and so-called "Y" dimensional movement, all per-key). This scheme is known as **MIDI Polyphonic Expression**[169] or **MPE**.

MPE works by hijacking MIDI's 16 channels: rather than assign each channel to a different synthesizers, MPE uses them for *different notes currently being held down* on a single synthesizer. MPE divides the 16 channels into one or two **zones**. If there is 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

[167]The 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 11.2.1, normally you'd have to control a synthesizer from a USB controller by attaching both to a laptop. The Parva is a rare exception.

[168]This is done, as it so happens, with a modified version channel aftertouch. These 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.

[169]The 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.

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.[170]

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...13 for a lower MPE zone, and then use channels 14 and 15 to control standard MIDI instruments in the conventional way, for example.

### 11.2.7 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,[171] 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, etc. Perhaps this might spell the end of custom and proprietary sysex formats.

[170] MIDI is an open protocol. The MPE specification, as well as other MIDI specifications and documents, are available for free at https://www.midi.org/

[171] In fact, I do not know how MIDI 2.0 tackles speed yet, but I assume it does.

- **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.

# 12 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 3, we'll see how to use 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.

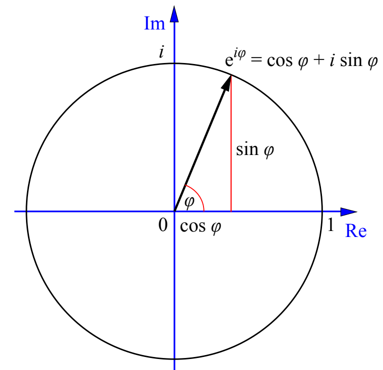



*Figure 184* Euler's Formula on the complex plane. The X axis is the real axis, and Y is the imaginary axis.©81

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**.[172] 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 the 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 $\omega$ 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.[173]

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

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

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. We'll deal with that issue in a moment.

[172]The 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**).

[173]You might be asking: why all the complexity involving imaginary components along with real-valued ones? Why not just break a sound into a bunch of real-valued sine waves? And the answer is, surprisingly: there's not a good reason. Indeed, as we'll discover later, the Fourier Transform is pretty wasteful when it's being applied to a real-valued signal (like sound): half of the result is useless repeated information. In fact there exists another common method, called the **Discrete Cosine Transform** or **DCT**, which does exactly what you'd want: it breaks the sound down into multiple cosine functions and that's it. The DCT is popular in compression schemes. But the Fourier Transform has been around for almost 200 years, with a long and cherished history, while the DCT has been around since 1972, when it was invented by **Nasir Ahmed**. So there you go.

This isn't the classic way to describe these transforms. Instead, we'd use **Euler's Formula**,[174] $e^{i\theta} = \cos(\theta) + i\sin(\theta)$, to cast the cos and sin into an exponential. (Also remember that $\cos(a) = \cos(-a)$ while $-\sin(a) = \sin(-a)$). This all results in:

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

$$S(i\omega) = \int_{-\infty}^{\infty} s(t)\left(\cos(\omega t) - i\sin(\omega t)\right) dt = \int_{-\infty}^{\infty} s(t)\left(\cos(-\omega t) + i\sin(-\omega t)\right) dt = \int_{-\infty}^{\infty} s(t) e^{-i\omega t} dt$$

## 12.1 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 have $N$ evenly spaced samples of sound totaling a time $T$, and so the sampling rate is $R = N/T$. The samples will be called $t = 0, 1, ..., N-1$. Not only are the number of samples discrete, but the number of *frequencies* will wind up being discrete too. We produce an array of $N$ complex numbers corresponding to angular frequencies from 0 to $2\pi$, and thus the real frequencies (in Hz) from 0 to $R$.[175] Using our discretized $t$ and $f$ values, our two equations transform to:[176]

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

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 $T$.

[174] There 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 184. But maybe it's easier to just explain with **Taylor series**. Here are three classic Taylor series expansion identities:

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

This means that

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

[175] Note that this means the highest frequency is $R$, which is impossible (remember Nyquist): but as you'll see we'll only be using the first half of the slots.

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

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

You could see it in sine/cosine form by applying Euler's Formula[177] again: remember it's $e^{i\theta} = \cos(\theta) + i\sin(\theta)$. This yields:

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

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

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

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 27** *The Discrete Fourier Transform*

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

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

[177] By 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.

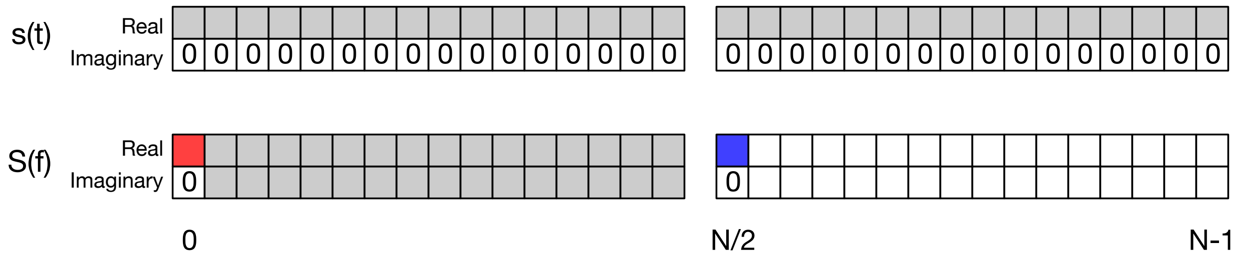



*Figure 185* Values of interest in the Time domain $s(t)$ and Frequency domain $S(f)$ arrays for a Real-Valued DFT. Gray, Red, and Blue boxes show numerical values of interest. Boxes with 0 in them should (or will) be set to 0. The red box at 0 is the value of the DC Offset. The blue box at $N/2$ is 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.

## 12.2 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 $n = 0...(N-1)$, the frequency value is a complex number $Y_n$ consisting of real $Yr_n$ and imaginary $Yi_n$ components. From this we can extract:

- The **amplitude** of the bin is just the magnitude $|Y_n|$, that is, $\sqrt{(Yr_n)^2 + (Yi_n)^2}$. Note that when $Y_n$ is only real valued, the magnitude is simply its absolute value.
- The **phase** of the bin is $\tan^{-1}\left(\frac{Yi_n}{Yr_n}\right)$
- 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.

## 12.3 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 $Xi$ will be all zeros. This has an interesting effect: there will be a certain mirror symmetry among the outputs $Y_1...Y_{N-1}$. Specifically, each output $Y_n$ will be the complex conjugate of output $Y_{N-n}$ starting at $n = 1$. That is, for all $n > 1, Yi_n = -Yi_{N-n}$.

Output $Y_0$ will not be part of this symmetry pattern: it will be real-valued only (that is, $Yi_0 = 0$). This is the "0 Frequency" bin, or **DC offset bin**. This bin contains the amount of **DC offset** contained in the sound, that is, the degree to which the whole sound is shifted up or down vertically. See Figure 186.

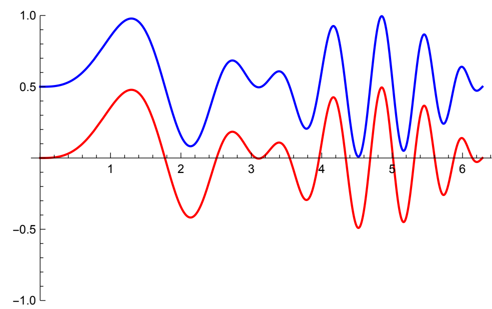



*Figure 186* Two functions which differ only in their DC offset. The red function has an offset of 0, while the blue function has an offset of 0.5.

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 (because $Yi_{N/2}$ must then be 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 185 shows this situation. This symmetry means that, when the time domain is real-valued, only slots $0...N/2-1$ (and possibly $N/2$) are relevant in the frequency domain: the remaining slots are just reflective complex conjugates which can be reconstructed from something else. This should make sense: as was mentioned in Section 2.2, the **Nyquist limit** is the largest possible frequency which can be embedded in a digital signal, and 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) matter.

## 12.4 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**.[178] The FFT uses a divide-and-conquer approach to recursively call smaller and smaller FFTs.

Recall that the forward DFT looks like this:

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

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:

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

[178] The 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 1822, but did not publish the results. No one noticed even when his collected works were published in 1866.

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 $0...S(N/2-1)$, we can *reuse* our $E(f)$ and $O(f)$ to compute $S(f)$ for $N/2...N-1$. 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 $e^{-i\pi} = 1$:

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

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 $f = 0....N-1$, we could do it like this:

1. Compute $E(f)$ and $O(f)$
2. For all $f = 0...N/2-1$, compute $S(f) = E(f) + e^{-i2\pi f\frac{1}{N}} \times O(f)$
3. For all $f = 0...N/2-1$, 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 $0...M-1$, 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 $\langle 1, 2, 4, 8, ..., N\rangle$ 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 28** *The Fast Fourier Transform (FFT)*

```
1:  Xi ← ⟨Xi_0...Xi_{N−1}⟩ array of N elements representing the imaginary values of the input
2:  Xr ← ⟨Xr_0...Xr_{N−1}⟩ array of N elements representing the real values of the input

3:  N ← length of Xi (and Xr)
4:  if N = 1 then
5:      return Xr and Xi
6:  else
7:      Yi ← ⟨Yi_0...Yi_{N−1}⟩ array of N elements representing the imaginary values of the output
8:      Yr ← ⟨Yr_0...Yr_{N−1}⟩ array of N elements representing the real values of the output
9:      M ← N/2
10:     Ei ← ⟨Ei_0...Ei_{M−1}⟩ even-indexed elements from Xi          ▷ ∀x : Ei_x = Xi_{2x}
11:     Er ← ⟨Er_0...Er_{M−1}⟩ even-indexed elements from Xr          ▷ ∀x : Er_x = Xr_{2x}
12:     Oi ← ⟨Oi_0...Ei_{M−1}⟩ odd-indexed elements from Xi           ▷ ∀x : Oi_x = Xi_{2x+1}
13:     Or ← ⟨Or_0...Or_{M−1}⟩ odd-indexed elements from Xr           ▷ ∀x : Or_x = Xr_{2x+1}
14:     Ei, Er ← FFT(Ei, Er)
15:     Oi, Or ← FFT(Oi, Or)
16:     for n from 0 to M − 1 do                    ▷ e^{−i2πf/N} = cos(2πf/N) − i sin(2πf/N)
17:         θ ← −2πn/N
18:         Yr_n ← Er_n + cos(θ)Or_n
19:         Yi_n ← Ei_n − sin(θ)Oi_n
20:     for n from M to N − 1 do
21:         θ ← −2πn/N
22:         Yr_n ← Er_n − cos(θ)Or_n
23:         Yi_n ← Ei_n + sin(θ)Oi_n
24:     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 complex conjugate of the FFT on the complex conjugate of the data.[179] That is:

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

...where 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 29** *The Inverse Fast Fourier Transform*

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

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

And that's all there is to it!

## 12.5 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 187. 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.

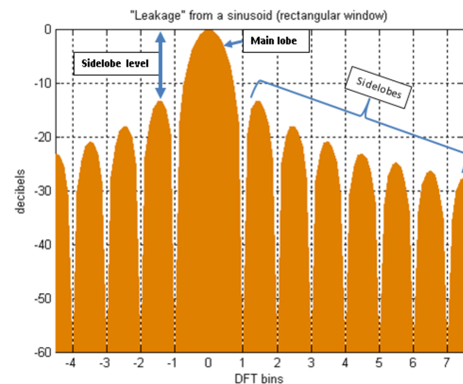



*Figure 187* Sidelobes in an FFT (note that the Y axis is on a log scale).©82

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

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. A window function $w(m, M)$ is just some function that runs from $m = 0$ to $m = M$ inclusive.

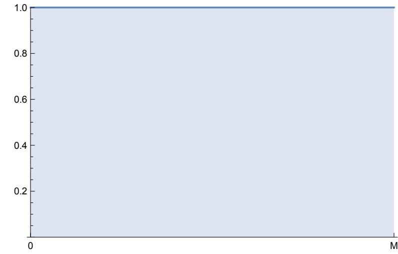


*Figure 188* Rectangular Window

Using a window function $w(m, M)$ is very simple: you just multiply it against each of your samples $s_0, ..., s_{N-1}$, resulting in $s_0 \times w(0, N-1), s_1 \times w(1, N-1), ..., s_{N-1} \times w(N-1, N-1)$:

**Algorithm 30** *Multiply by a Window Function*

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

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

If you have no window function, then $w(n, N) = 1$ for all $n$. This is called the **rectangular window**. Most useful 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 simple (and often bad) example is the **Bartlett** or **Triangular window**:

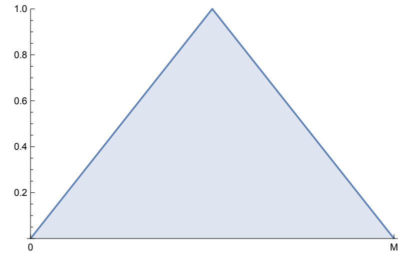


*Figure 189* Triangular Window

$$w(m, M) = \begin{cases} \frac{2m}{M} & m < \frac{M}{2} \\ 2 - \frac{2m}{M} & m \geq \frac{M}{2} \end{cases}$$

However probably the most common general-purpose window function is the **Hamming window**:

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

Windows are used in a variety of other ways as well, such as creating grains for granular synthesis (Section 9.4) and tapering off the Sinc function used in resampling (Section 9.7). And note that the windows shown so far are *far* from the only windows available: later on we'll see **Blackman** and **Kaiser** (Section 9.7), and the **Hann** (Section 12.6 coming up, as well as Section 9.4). There's also Gaussian, Tukey, Planck-Taper, Slepian, Dolph-Chebyshev, Ultraspherical, Poisson, Lanczos, MLT Sine.... There's a complex theory behind choosing the right window for your task.

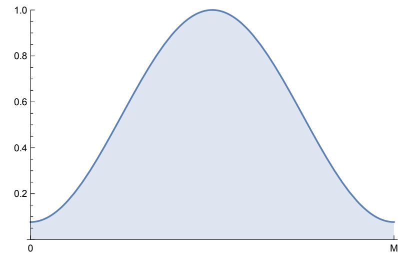


*Figure 190* Hamming Window

## 12.6 The Short Time Fourier Transform

So far we've discussed taking an *entire* sound and converting *all* of it, with an FFT, into the frequency domain to analyze it, or to modify it and convert back via an IFFT. But music synthesis is a real-time thing: we rarely have the entire sound to manipulate at once: and besides, even though the FFT is $O(\lg N)$, with a long sound, that $N$ could be awfully big.

Instead, it's often the case that we might wish to deal with the sound *bit by bit*, in real-time. There are three basic uses for this:

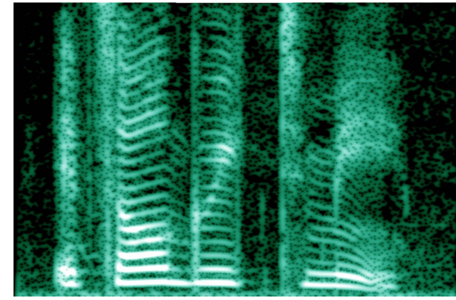


*Figure 191* A Spectrogram (Sonogram) of the human voice. The $x$ axis is time, the $y$ axis is frequency, and brightness is the amplitude of the partial at that frequency.©83

- **Analyze** the sound by converting it into the frequency domain and examining or visualizing how it changes over time. A visualization of this sort is known as a **spectrogram** (or **sonogram**). Consider Figure 191, where the amplitude of various frequencies ($y$ axis) changes over time ($x$ axis).
- **Synthesize** a sound as chunks in the frequency domain, and then converting them into the time domain to be played (this is not very common).
- **Modify** a sound by converting it bit-by-bit into the frequency domain, mucking about with it there, and then converting it back.

To *analyze* the sound, we might divide the sound into small chunks, then run an FFT on each of them independently, and display the results in a spectrogram fashion. To *synthesize* the sound we could do the opposite: create chunks in the frequency domain, then do an IFFT on them and assemble the results into the sound. Finally, to *modify* the sound we'd do both: first break it into chunks, then do an FFT to cast them into the frequency domain, tinker with them there, do an IFFT to cast back into the time domain, and reassemble. The general procedure of breaking a sound into multiple pieces, and then performing the FFT on them, is known as **Short Time Fourier Transform** or **STFT**.

**Applications and Alternatives** The STFT has many applications in sound synthesis:

- **Visualization** You can easily analyze how amplitudes and frequencies change over time.
- **Filtering** You can accentuate, lower, or entirely strip out partials by converting the sound to the frequency domain with an STFT, modifying (or zeroing out!) the amplitudes of interest, then converting back to the time domain with an Inverse STFT. Similarly, you could modify the phases of various partials. Section 10.4 discusses an example of this in depth.
- **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 STFT 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 STFT, then just move each partial in the frequency domain to the spot representing twice its frequency. Then do an inverse STFT to go back to the original sound.

- **Resynthesis** A sound is sampled and analyzed, and then recreated (sort of) 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, perhaps drawn by the musician as if it were an image: then use the Inverse STFT to convert the result into a sound. Such tools are called **image synthesis** synthesizers. Some additive synthesizer tools, such as **Image-Line Software's Harmor**, also sport image-synthesis facilities.

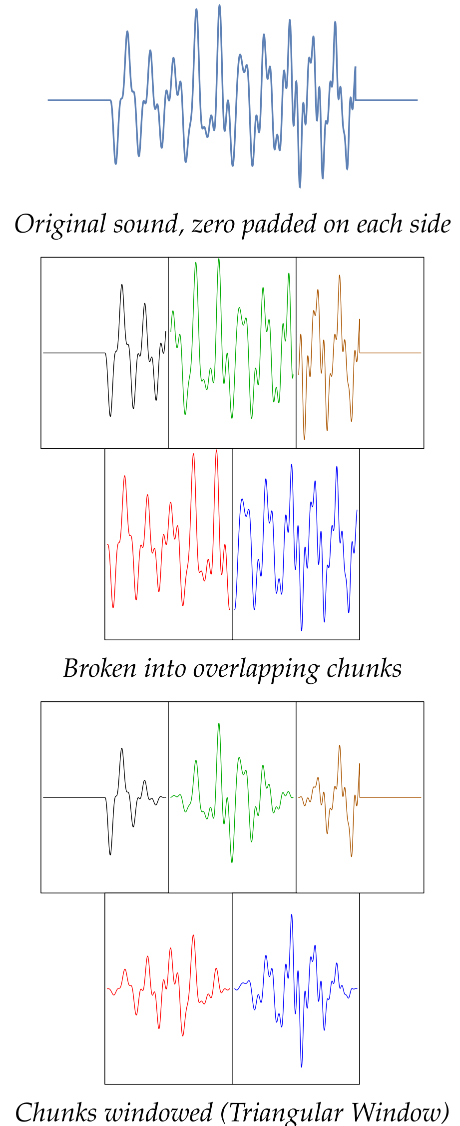



*Figure 192* Preparing a sound for the STFT. Note that the Triangular Window was used here for clarity, but this is normally not a good choice. Obviously for a typical sound, a great many more than five chunks would be employed.

There are alternatives. For example, it turns out that multiplying the amplitude or phase of partials in the frequency domain corresponds to **convolution** in the time domain. Convolution is the basic tool used to develop filters solely in the time domain, and is the approach taken in Section 7 as well as in Section 9.7. Similarly, there are various clever algorithmic tricks to approximate pitch scaling in the time domain involving removal or interpolation of individual samples.

Okay, let's get started.

**Cutting Up the Sound** If we're performing analysis or modification, we'll first need to cut up the sound into chunks in order to perform the FFT on each one. A naive approach would be to just snip the sound into pieces and apply a window called the **analysis window** to each piece prior to the FFT. This might work okay for analysis, but not for modification. Here is why. Suppose that we cut up a sound $S$ into two chunks $A$ and $B$. Because the sound is smooth, the $B$ wave starts at the amplitude where $A$ left off. Suppose that we performed an FFT on $A$ and on $B$ respectively, heavily mucked with their frequency components, and performed an IFFT on each, resulting in the modified sounds $A'$ and $B'$. Now there is *no guarantee* that $A'$ will once again end where $B'$ starts, so when we concatenate them to form the final sound $S'$, it'll have a big discontinuity in the middle of it, resulting in a loud pop sound.

We could deal with this by applying another window to $A$ and $B$ *after* the IFFT. This is called the **synthesis window** and it will certainly guarantee that $A'$ and $B$ meet at zero. And this might work for two chunks: but if you had a large number of small chunks, they'd all be meeting at zero at regular intervals, creating a beating sound. Instead we need a way to **cross fade** all these chunks into each other, which in turn requires a more clever way of breaking them up in the first place.

What we'll do is break the sound into *overlapping* chunks, each of size $M$.[180] In Figure 192, the chunks overlap half-way, kind of like bricks, and so they are spaced by $M/2$. This spacing is known as the **hop size**. There are other overlap options, but let's stick with half-way. We also need to add sufficient **zero padding** to each end of the sound. We need enough padding that the total length is a multiple of $M/2$, and also so that (if we're doing modification rather than analysis) there is at least $M/2$ padding on each end so the sound doesn't fade in and out afterwards (as we'll see).

The next step is to apply an analysis window to each chunk. In Figure 192 we applied a Triangular window for clarity, but something else would be better, as discussed later. After applying the analysis window, we perform an FFT independently on each chunk. If we're doing analysis, we're done. If we're doing modification, we modify the chunks as we like in the frequency domain, and then reassemble.

**Modifying the Sound** Once the sound is in the frequency domain, we can muck with it. We might filter out the amplitudes of certain frequencies, or shift all the frequencies up, or modify the phases of certain partials. Or if we're doing *synthesis*, we might construct the partials' amplitudes and phases out of whole cloth.

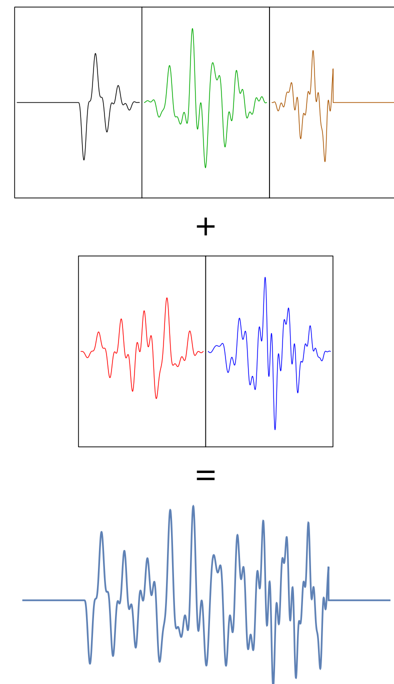



*Figure 193* Reassembly of chunks with Overlap-Add. Windowed overlapping chunks are summed, producing the final (still zero-padded) sound.

**Reassembling the Sound** Modification and synthesis will require us to build (or rebuild) the sound in the time domain. The general procedure is easy: we perform the IFFT to get our sound chunks back. We then apply the synthesis window to each, overlap them, and add them up to form the final sound, as illustrated in Figure 193. This is known as the **Overlap-Add Method**. More formally, note from the figure that every sample is covered by two chunks: some chunk $C^i = \langle C^i_0, ..., C^i_{N-1} \rangle$ and the next chunk $C^{i+1} = \langle C^{i+1}_0, ..., C^{i+1}_{N-1} \rangle$. Let's say that $C^i$ starts at position $j$ and $C^{i+1}$ thus starts at position $j + N/2$. Sample number $k$ lying within the boundaries of both chunks is thus just $C^i_{k-j} + C^{i+1}_{k-j-N/2}$.

Here's the payoff comes from selecting overlapping chunks: the sound can be elegantly recreated. In fact, because the STFT is invertible, if we made no changes in the frequency domain, it's possible to *perfectly* reconstruct the sound. It just requires the right choice of windowing, as we'll see later. Once we've reconstructed the sound, all that remains is to snip off the zero-padding.

**Selecting a Window** Choosing the right analysis and synthesis windows is tricky. The problem is that you now have *two* goals in choosing a window:

- The **analysis window** should reduce the spectral leakage into sidelobes as much as possible when producing the FFT.
- The **synthesis window** (and to some extent the analysis window) should enable chunk overlaps to assemble (or reassemble) the sound properly.

[180]Since we're doing an FFT ultimately, $M$ ought to be a power of 2.

If we're just doing *analysis*, there's only one window (the analysis window) and so the objective is simple: pick a window with the best spectral properties.

Selecting a synthesis window involves more considerations. We want a window which starts and ends at zero, so that we don't have pops in the sound. But we also want one which doesn't introduce subtle distortions in the sound as the overlapping windows go up and down. That is, we're looking for a window which, when overlapped with copies of itself, sums to a constant (often 1) except possibly at the ends of the sound. If you sum it up as $\sum_{i=0}^{B} w(n \times M/2, M)$, the result will be 1 except in the end regions ($0...M/2$, and $B - M/2...B$)), because at the ends there's no additional window to overlap with. This is known as the **Constant Overlap-Add** or **COLA** property.

The Rectangular window of course sums to 2 when overlapped with itself spaced half-way, so it has the COLA property: but the Rectangular window doesn't start and end at zero (nor does the Hamming window). If you think about it, should be obvious that the Bartlett or Triangular window not only starts and ends and zero, but also has the COLA property.

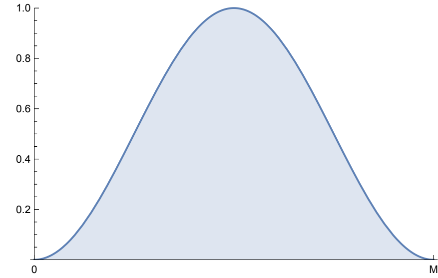



*Figure 194* Hann window (Repeat of Figure 145)

Surprisingly, certain other windows have this property as well: consider for example the **Hann window**, which is little more than a cosine:[181]

$$w(m, M) = 1/2 - 1/2 \cos\left(\frac{2\pi m}{M}\right)$$

The Hann window is shown in Figure 194. The COLA property of the Hann window is illustrated in Figure 195. This also shows why we're adding at least $M/2$ zero padding on each end of the original sound: so that a COLA window doesn't fade the sound in and out at the ends.

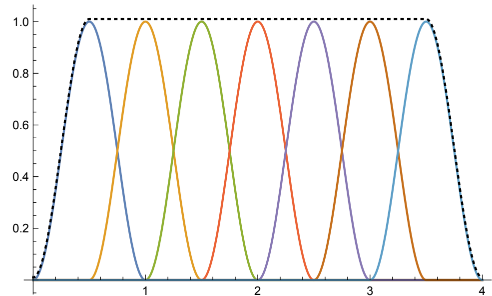



*Figure 195* The Hann window's COLA property. When spaced half-way apart, a series of Hann windows sums to 1 (dotted black line), except at the ends.

If you're doing *synthesis* with the STFT, then you'd want to pick a COLA synthesis window. But for modification it's more complicated. In this case are windowing the sound *twice*, at the analysis step and at the synthesis step. It's possible to use a COLA window both times (for example), but there's a problem. The Inverse FFT is just that: the *inverse* of the FFT: so $\text{IFFT}(\text{FFT}(s)) = s$. Now suppose that we had *not made any modification of the sound* while we were in the frequency domain. The sound was windowed twice with a COLA window $w()$, so now for each chunk we have:

$$w(\text{IFFT}(\text{FFT}(w(s)))) = w(w(s)) = w^2(s) \neq w(s)$$

That is, after performing the STFT and inverse STFT, we're not really overlapping with a COLA window: we're overlapping with its square.

One plausible way around this this is to make the analysis and synthesis windows each be the *square root* of a COLA window, that is, $r(...) = \sqrt{w(...)}$. Then we'd have (at least for the identity case where we didn't modify anything in the frequency domain):

$$r(\text{IFFT}(\text{FFT}(r(s)))) = r(r(s)) = w(s)$$

[181] Oddly, the **Blackman window** (Section 9.7) isn't COLA when spaced half-way, but *is* COLA when spaced by 1/3.

But this works only if we're effectively doing nothing. The more significant modifications we make to the sound in the frequency domain, the less or modifications look like the identity function and the more they resemble synthesis from scratch. If we're making *light* modifications, using the square root of COLA might be a reasonable choice for the analysis and synthesis windows; but with heavy or unusual modifications, something closer to a straight COLA window might be called for, at least for synthesis.

**Chunk Sizes** One item that has been left out until now: what should be the chunk size $M$? Using the FFT methods discussed so far, $M$ should be a power of two, but *which* power of two involves the following trade-off. A larger size of $M$ will result in a larger FFT, with more bins, and so the frequency domain will be more detailed (higher resolution). However a larger size of $M$ also means that each chunk will take up a larger part of your sound. Let's suppose you had a short "blip" in your sound, much smaller than the size of your chunk. Your STFT wouldn't reflect its sudden appearance and disappearance, as it'd be averaged in with other samples in its chunk. This is known as the trade-off between the **time resolution** and **frequency resolution** of your Fourier transform. You have to decide which is more important.

**Implementation** In summary, if your goal is to *analyze* (or visualize) the sound:

1. Perform the **Short Time Fourier Transform:**
   (a) Zero-pad the sound to make its length a multiple of $M/2$
   (b) Break into overlapping chunks $M$ long
   (c) Window each chunk to deal with spectral leakage
   (d) Perform FFT on each chunk (thus $M$ should be a power of 2)
2. Analyze or display the results

If your goal is to *synthesize* a new sound from scratch in the frequency domain:

1. Create the sound chunks from scratch in the frequency domain
2. Perform the **Inverse Short Time Fourier Transform:**
   (a) Perform IFFT to return the chunks to the time domain
   (b) Window each chunk as discussed before (the synthesis window)
   (c) Overlap and sum the chunks
   (d) Trim the ends if needed

Finally, if your goal is to *modify* a sound:

1. Perform the **Short Time Fourier Transform** with a few caveats:
   (a) Zero-pad the sound to make its length a multiple of $M/2$ and to add at least $M/2$ padding on each end
   (b) Break into overlapping chunks $M$ long
   (c) Window each chunk as discussed before (the analysis window)
   (d) Perform FFT on each chunk (thus $M$ should be a power of 2)
2. Modify the sound in the resulting frequency domain as you like
3. Perform the **Inverse Short Time Fourier Transform** with a few caveats:
   (a) Perform IFFT to return the chunks to the time domain
   (b) Window each chunk *again* as discussed before (the synthesis window)
   (c) Overlap and sum the chunks
   (d) Remove zero-padding / trim ends if needed

Minding the appropriate choices for our analysis and synthesis windows, and for the chunk size $M$, our algorithms then are:

**Algorithm 31** *The Short Time Fourier Transform*

1: $X \leftarrow \langle X_0, ..., X_{N-1} \rangle$ samples of a sound
2: $M \leftarrow$ chunk length length ▷ Should be a power of 2
3: $w_a(m, M) \leftarrow$ analysis window function

4: $X' \leftarrow \langle X'_0, ..., X'_{P-1} \rangle$ the original sound $X$ padded with $M/2$ zeros on each end, plus additional zeros on the end sufficient to make its length $P$ a multiple of $M/2$
5: $C \leftarrow \{C_0, ..., C_{P/(M/2)-1}\}$ sound $X'$ broken into overlapping chunks. Each chunk $C_i$ is $M$ samples long and consists of the snippet $\langle X'_{i\times M/2}, ..., X'_{i\times M/2+M-1} \rangle$
6: $C' \leftarrow \{C'_0, ..., C'_{P/(M/2)-1}\}$ chunks from $C$, each multiplied by the window $w_a(m, M)$ ▷ Algorithm 30
7: **for** each $C'_i \in C'$ **do**
8: $\quad F_i \leftarrow$ FFT on $C'_i$ ▷ Algorithm **??**, keeping in mind that $C'_i$ consists of all real parts
9: $F \leftarrow \{F_0, ..., F_{P/(M/2)-1}\}$ ▷ Per-chunk FFT results
10: **return** $F$

**Algorithm 32** *The Inverse Short Time Fourier Transform*

1: $F \leftarrow \{F_0, ..., F_{P/(M/2)-1}\}$
2: $M \leftarrow$ chunk length length ▷ Should be a power of 2
3: $w_s(m, M) \leftarrow$ synthesis window function
4: $N \leftarrow$ length of original sound

5: **for** each $F_i \in F$ **do**
6: $\quad C'_i \leftarrow$ IFFT on $F_i$ ▷ Algorithm 29, disregarding imaginary parts
7: $C \leftarrow \{C_0, ..., C_{P/(M/2)-1}\}$ chunks from $C'$, each multiplied by the window $w_s(m, M)$ ▷ Algorithm 30
8: $X' \leftarrow \langle X'_0, ..., X'_{P-1} \rangle$ samples, all zero
9: **for** each $X'_i \in X'$ except for the first $M/2$ and last $M/2$ samples **do** ▷ Reassembly
10: $\quad j \leftarrow \lfloor i/(M/2) \rfloor$ ▷ $j$ will start at 1
11: $\quad A \leftarrow \langle A_{(j-1)\times M/2}, ..., A_{(j-1)\times M/2+M-1} \rangle$ snippet of samples in chunk $C_{j-1}$
12: $\quad B \leftarrow \langle B_{j\times M/2}, ..., A_{j\times M/2+M-1} \rangle$ snippet of samples in chunk $C_j$
13: $\quad X'_i \leftarrow A_i + B_i$ ▷ They overlap and each will contain sample $i$ in its range
14: $X \leftarrow X'$ trimmed of its first $M/2$ samples, then after that trimmed at the end to bring $X$ down to $N$ samples long ▷ If you're just doing synthesis, you might not want to trim at all
15: **return** $X$

## 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*,[182] 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 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*.[183] 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.dspguide.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))


[182] Steven Smith, 1997, *The Scientist & Engineer's Guide to Digital Signal Processing*, California Technical Publishing, available online at https://www.dspguide.com/

[183] 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_timbre_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.dspguide.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://cs.gmu.edu/~sean/book/synthesis/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/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] Ken Heaton, Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48242081

©[11] Paulo Ordoveza, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=29694963

©[12] Jane023, CC BY-SA 3.0 nl, https://commons.wikimedia.org/w/index.php?curid=19732582

©[13] Public Domain, https://commons.wikimedia.org/w/index.php?curid=391692

©[14] Scientific American, 1907, Public Domain, https://commons.wikimedia.org/w/index.php?curid=49752962

©[15] Public-domain figure taken from US Patent US580035A.

©[16] JacoTen, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=24356990

©[17] Julien Lozelli, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=7767415

©[18] Brandon Daniel, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=33527250

©[19] Allangothic (Wikipedia), CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2875045

©[20] Museumsinsulaner, Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=12503701

©[21] User:MatthiasKabel, Own work, CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=1265010

©[22] Reproduced with permission of the Columbia University Computer Music Center

©[23] Surka, Own work, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=8345595

©[25] GeschnittenBrot, Flickr: DSC_0117, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=14746344

©[26] Kimi95, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=7708499

©[27] I, Zinnmann, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2441310

©[28] Museumsinsulaner, Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=12501904

©[29] FallingOutside, Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=21742056

©[30] Andrew Russeth, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=26604703

©[31] Brandon Daniel, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=19964001

©[32] Hollow Sun, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=8863073

©[33] Robert Brook, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=7509844

©[34] Pete Brown, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=38092002

©[35] Mojosynths, Own work, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=3815497

©[36] Reproduced with permission from Richard Lawson at RL Music. Thanks to Sequential LLC for assistance.

©[37] CPRdave, Public Domain, https://commons.wikimedia.org/w/index.php?curid=7633923

©[38] Allert Aalders, CC BY-NC-SA 2.0, https://www.flickr.com/photos/allert/6809874296/

©[39] Ed Uthman, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=74691297

©[40] Jdmt, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=29340041

©[41] Candyman777, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=19810850

©[42] F J Degenaar, Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1671111

©[43] Reproduced with permission from Sequential LLC.

©[44] Paul Anthony, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=67924982

©[45] Reprinted with permission from Make Noise, Inc., via CC BY-NC-ND. Photo credit to Eric "Rodent" Cheslak. 0-Coast developed by Tony Rolando.

©[46] Joshua Schnable, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=26553228

©[47] John Athayde, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=7530494

©[48] F J Degenaar, Own work, Public Domain, https://commons.wikimedia.org/w/index.php?curid=1671111

©[49] My screenshot and patch example: but VCV Rack's module graphics are copyright 2018 by Grayscale (https://grayscale.info/) and licensed CC BY-NC-ND 4.0.

©[50] Warrakkk, https://commons.wikimedia.org/w/index.php?curid=19273415.

©[51] BillyBob CornCob. CC0, https://commons.wikimedia.org/w/index.php?curid=76760547

©[52] Cameron Parkins, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=7810459

©[53] Gablin, CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=1377602

©[54] Spinningspark, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=23488139

©[55] Geek3, CC BY 4.0, https://commons.wikimedia.org/w/index.php?curid=53958748

©[56] Andrew Russeth, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=26604703

©[58] Julianfincham, CC BY-SA 3.0, https://commons.wikimedia.org/wiki/File:EDP_Wasp.JPG

©[59] Spinningspark, CC BY-SA 3.0, https://en.wikipedia.org/w/index.php?curid=27292262

©[60] Public-domain figure taken from US Patent US3475623A.

©[61] Steve Sims, Public Domain, https://commons.wikimedia.org/w/index.php?curid=12379187

©[62] Stonda (assumed) (Wikimedia contributor), CC BY 2.5, https://commons.wikimedia.org/w/index.php?curid=649793.

©[63] Buzz Andersen, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=2801145

©[64] Matt Vanacoro, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=39425190

©[65] Public Domain, from https://waveeditonline.com/

©[66] 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

©[67] Reprinted with permission from Tasty Chips Electronics.

©[68] Reprinted with permission from Livi Lets.

©[69] What's On the Air Company, Public Domain, https://commons.wikimedia.org/w/index.php?curid=56188934

©[70] Matt Vanacoro, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=39425186

©[71] Ben Franscke, CC BY-SA 2.5, https://commons.wikimedia.org/w/index.php?curid=7633823

©[72] Vlad Spears, CC BY-SA 2.0, https://commons.wikimedia.org/w/index.php?curid=2930845

©[73] LeoKliesen00, Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=48087446

©[74] DeepSonic, CC BY-SA 2.0, https://www.flickr.com/photos/deepsonic/6600702361

©[75] By Lippold Haken and Edmund Eagan CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3710610

©[76] Reprinted with permission from Roger Linn.

©[77] Klaus P. Rausch, CC0, https://commons.wikimedia.org/w/index.php?curid=60729226

©[78] F7oor, CC BY-SA 2.0, https://www.flickr.com/photos/f7oor/3992788445/

©[79] Pretzelpaws CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=142551

©[80] Reprinted with permission from Brad Ferguson.

©[81] Gunther, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=821342

©[82] Bob K, Public Domain, https://commons.wikimedia.org/w/index.php?curid=648112

©[83] Dvortygirl, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=2524720

# Index


$\mu$-law, 21
5.1 surround sound, 22

a-law, 21
Ableton Live, 150
accent, 45
action, 147
ADC, 20, 119
additive synthesis, 10, 16
ADSR envelope, 27, 41, 61, 78
Adventure Kid, 133
aftertouch, 37, 150
AHDSR envelope, 41
Ahmed, Nasir, 163
AHR envelope, 41
AIR Music Technology Loom, 25
Akai MPC series, 119
algorithms, 115
aliasing, 20, 66
all notes off, 157
all pass filter, 80, 137, 139
all sound off, 157
alternators, 23
*Alvin and the Chipmunks*, 172
*Amen Break*, 119
amplifier, 14, 26, 50, 52, 57, 58, 63
amplitude, 15, 79, 166
amplitude modulation, 75
amplitude response, 79, 87, 88
analog filter, 86
analog synthesizer, 10, 13
Analog-Digital Converter, 20, 119
Analogue Solutions, 60
analysis window, 173, 174
angle modulation, 109
angular frequency, 18, 87
AR envelope, 41
ARP 2500, 52
ARP 2600, 9, 52, 82
ARP Odyssey, 53
arpeggiator, 14, 28, 46, 47, 59
arpeggio, 47
ASM Hydrasynth, 77, 149
asynchronous granular, 123
attack level, 14, 27, 41
attack rate, 41
attack time, 14, 27, 41
AU, 13, 55
audio interface, 13
Audio Modeling, 145
audio rate, 37
Audio Unit, 13
automated modulation, 37

band limited impulse train, 67
band limited step function, 69
band limited wave, 20, 66, 124
band pass filter, 26, 59, 79, 106
band reject filter, 79
bandwidth, 106, 112
bank, 155, 157
bar, 45
Bartlett window, 171
beating, 75
Behringer BCR2000, 149
Bell Labs Digital Synthesizer, 25
Bessel Function, 113, 129
Bessel function of the first kind, 113
bilinear transform, 86, 95
bin, 166
bipolar, 37, 48
bipolar envelope, 44
bitrate, 21
Blackman window, 129, 171, 175
*Blade Runner*, 54
BLEP, 69, 72, 77
BLIT, 67
blue noise, 65
Bode plot, 86, 89
Bode, Harold, 51
BPBLIT, 68
breath controller, 150, 157
brick wall filter, 80, 128
brown noise, 65
Brownian motion, 65
Buchla, Don, 9, 51, 71, 82
buffer, 31
Butterworth filter, 92

Cahill, Thaddeus, 25
Camel Audio Alchemy, 25
capacitive keyboard, 148
cardinal sine function, 127
carillon, 18
Carlos, Wendy, 51
carrier, 76, 110
Carson's rule, 112
Casio CZ series, 73
Casio CZ-1, 73
Casio CZ-101, 73
Catmull-Rom, 30, 126
causal filter, 130
CC, 49
CEM, 82
CEM 3320, 82
cents, 19
channel aftertouch, 151
channels, 22
*Chariots of Fire*, 54
Chebyshev Polynomials of the First Kind, 71
Chebyshev, Pafnuty, 71
chorus, 11, 59
Chowning, John, 109
Ciani, Suzanne, 52
Clavia Nord Lead, 55
clip, 45
clipping, 72
clock, 46
clock pulse, 155
*Clockwork Orange*, 51
*Close Encounters of the Third Kind*, 52
coefficients, 87
COLA, 175
comb filter, 80, 136
combiner, 14, 50, 57, 63, 74
companding, 21
complex conjugate, 170
Constant Overlap-Add, 175
Control Change, 49
control surface, 149
control voltage, 48, 59, 159
controller, 10, 13, 55, 147
convolution, 83, 127, 141, 173
convolution reverb, 141
Cooley, James William, 167
Cooley-Tukey FFT, 167
correlation, 127
Creative Labs Sound Blaster, 109, 121
cross fade, 74, 120, 173
cross modulation, 77
cubic interpolation, 126
Curtis Electromusic Specialties, 82
Curtis, Doug, 82
cutoff frequency, 14, 79, 92
CV, 48, 56, 59, 159
CV/Gate, 59, 159

D-Beam, 148
DAC, 20, 119
DADSR envelope, 14, 41, 58
*Dark Side of the Moon*, 52
Dave Smith Instruments Prophet '08, 13, 57
Dave Smith Instruments Prophet 6, 56
DAW, 11, 13
DC offset, 113, 130, 166
DC offset bin, 166
DCO, 58, 63
DCT, 163
decay rate, 41
decay time, 27, 41
decibel, 19
decimation, 124
Deep Note, 9
delay, 11, 59, 135
delay time, 14, 41
denormals, 29
desktop synthesizer, 10, 14
detune, 14, 58, 75
DFT, 165
digital audio workstation, 11, 13, 45, 55, 120, 147, 153
digital delay line, 135, 143
digital filter, 86
digital signal processor, 10
digital synthesizer, 7, 10, 55
digital waveguide synthesis, 144
Digital-Analog Converter, 20, 119
Digitally Controlled Oscillator, 58, 63
diode ladder filter, 103
diphthong, 107

Direct Form I, 85
Direct Form II, 85
Discrete Cosine Transform, 163
Discrete Fourier Transform, 165
discretization, 86
Doepfer Musikelektronik, 56, 59
Doppler effect, 138
downsampling, 112, 124
drawbars, 25
drawknob, 23
drum beat, 45
drum computer, 46
drum machine, 9, 10, 46, 119
drum pad, 150
drum synthesizer, 46
dry, 11, 135
DSP, 10
duophonic, 144
duty cycle, 64, 68
dynamic range, 21, 147

E-mu Systems, 119
E-mu Systems Morpheus, 82, 120
E-mu Systems UltraProteus, 120
early reflections, 139
East Cost synthesis approach, 52
echo, 11
Edisyn, 7, 48, 49
EDP Wasp, 82
effect, 59, 135
effects unit, 11
Eigenlabs Eigenharp, 151
Ekstrand, Kristoffer, 133
electrical isolation, 153
Elektron Digitone, 117
ELP, 51
embouchure, 160
Emerson, Keith, 51
EMS Synthi, 52
EMS VCS 3, 52
Ensoniq, 119
Ensoniq SQ80, 151
envelope, 27, 40, 78
envelope generator, 40
equal temperament, 17
equalization, 59
Euler’s Formula, 164
Eurorack, 56, 57, 59
expander, 10
exponential FM, 111
expression controller, 157
expression pedal, 37, 148
expressivity, 150

fader, 46
Fairlight CMI, 119
Fairlight Qasar M8, 25
Fast Fourier Transform, 83, 141, 167
feedback, 104
feedback comb filter, 84, 137
feedforward comb filter, 83
FFT, 83, 141, 167
filter, 14, 26, 34, 50, 57, 60, 63, 65, 74, 79
filter composition, 81, 87, 102
filter FM, 58, 118
filtering, 172
filters in parallel, 102
filters in series, 102
finite impulse response filter, 83, 141
FIR, 65, 70, 83
first-order filter, 83, 103
flams, 45
flanger, 59, 136
Flow, 7, 35
FM synthesis, 10, 37, 55, 63, 73, 77, 107, 109, 110, 138
foldover, 20, 66
foot controller, 157
formant, 106
formant filter, 27, 35, 107
formant synthesis, 107
forward comb filter, 136
four pole filter, 58, 59, 80
Fourier Series, 163
Fourier Transform, 16, 163
Fourier, Joseph, 163
free, 40, 44
free LFO, 38
Freeverb, 140
frequency, 15, 166
frequency division, 53
frequency domain, 15, 83, 87

frequency modulation synthesis, 10, 16, 37, 63, 73, 77, 107, 109, 110, 138
frequency resolution, 176
frequency response, 79, 87
frequency warping, 95
fully diminished, 18
fundamental, 17, 113
*Funky Drummer*, 119

gain, 19, 80, 84, 91
gate, 48, 56, 59, 159
Gauss, Carl Friedrich, 167
ghost note, 45
Gizmo, 7, 46, 47
glide, 32
Gold, Rich, 122
grain cloud, 123
grain envelope, 122
grains, 122
granular synthesis, 122
grid controller, 150
groovebox, 47
guitar processor, 150

Haken Continuum Fingerboard, 151
Hamming window, 171
Hammond Novachord, 53
Hammond Organ, 25, 138
Hann window, 122, 129, 132, 171, 175
hard sync, 77
harmonics, 17, 34, 63, 113
hertz per volt, 48, 159
high pass filter, 26, 59, 70, 79, 103
hold stage, 41
hop size, 174
hum tone, 18
human factors, 150

IDFT, 165
IFFT, 25, 141, 170
IIR, 84
image synthesis, 173
Image-Line Harmor, 25, 173
impulse, 141
impulse response, 141
In The Box, 13
index of modulation, 111, 112
infinite impulse response filter, 84
inharmonic, 114
instantaneous frequency, 110
instantaneous phase, 30, 73, 110
interpolation, 123–125
Inverse Discrete Fourier Transform, 165
Inverse Fast Fourier Transform, 25, 141, 170
Inverse Fourier Transform, 16, 163
ITB, 13

jack, 51
Jankó keyboard, 150
joystick, 60
*Jump*, 54

Kaiser window, 129, 171
Kakehashi, Ikutaro, 151
Karplus, Kevin, 142
Karplus-Strong, 142
Kawai K1, 77
Kawai K3, 25
Kawai K4, 48, 77
Kawai K5, 25, 34, 107
Kawai K5000, 25
Kawai K5m, 34
Kellet, Paul, 65
keyboard, 147
keyboard tracking, 81
Koenig, Rudolf, 23
Korg M1, 120
Korg microKORG, 55, 57, 59
Korg MS10, 52
Korg MS20, 52, 82
Korg MS2000, 59
Korg PS-3300, 54
Korg Wavestation, 9, 75, 120

ladder filter, 103
Lagrange interpolation, 126
Lagrange polynomial, 126
Lagrange, Joseph-Louis, 126
Laplace domain, 86, 87
last note priority, 32
late reflections, 139
latency, 31, 49, 130

leak, 170
leaky integrator, 68
legato, 28, 32, 38, 40, 41, 44, 57
Leslie, 25
Leslie speaker, 138
LFO, 14, 27, 37, 53, 58, 60
linear arithmetic synthesis, 120
linear arranger, 45
linear filter, 84
linear FM, 109, 111
linear interpolation, 125, 137
linear phase filter, 80
Linn LM-1 Drum Computer, 47
Linn, Roger, 47
local switch, 157
lollipop graph, 20
lossy compression, 21
low frequency oscillator, 14, 27, 37, 53, 60, 110
low pass feedback comb filter, 140
low pass filter, 14, 20, 26, 50, 52, 58, 59, 65, 79, 103, 124
lower zone, 161

magnitude, 87
magnitude frequency response, 79
Make Noise 0-Coast, 56
matrix destination, 52
matrix source, 52
membrane, 148
MIDI, 10, 14, 49, 54, 147, 151
MIDI 2.0, 160
MIDI 2.0 profile, 161
MIDI channel mode messages, 157
MIDI data byte, 153
MIDI in, 152
MIDI interface, 13
MIDI merge, 152
MIDI out, 152
MIDI patchbay, 152
MIDI Polyphonic Expression, 157, 160
MIDI router, 152
MIDI status byte, 153
MIDI thru, 152
mini key, 148
mixer, 11, 60
mixing, 27, 58, 74
mixing console, 11
Mixtur-Trautonium, 51
mixture, 23
mode, 17
modifier function, 49, 58
modular synthesizer, 9, 51, 59
modulation, 14, 27, 50, 57, 63
modulation amount, 49
modulation destination, 49
modulation matrix, 14, 49, 52, 54, 58
modulation source, 49
modulation wheel, 37, 149, 157
modulator, 76, 110
module, 51
Moiré patterns, 66
monitor, 11
mono mode, 157
monophonic, 10, 32, 53
Moog MemoryMoog, 54
Moog Minimoog Model D, 10, 53, 82, 103
Moog transistor ladder filter, 103
Moog, Robert, 9, 51, 82, 103
morph, 36
MP3, 21
MPE, 157, 160
MPE master channel, 160
MPE zone, 160
multi-stage envelope, 44
multi-tap delay line, 137
multi-track sequencer, 45
multi-track tape recorder, 11
Multidimensional Polyphonic Expression, 160
multimode, 34
multimode filter, 82
multiple wavetable synthesis, 121
multitimbral, 33, 120
Musical Instrument Digital Interface, 54, 147, 151
Mutable Synthesis Clouds, 123

Native Instruments Razor, 25
NCO, 63
New England Digital Synclavier II, 25
noise floor, 21

Non-Reserved Parameter Numbers, 157, 158
nonlinear filter, 84
noodling, 11
normal form, 114
normalized sinc function, 127
notch filter, 26, 79
note, 17
note latch, 47
note length, 46, 47
note priority, 32
note velocity, 47
Novation Launchpad, 150
Novation Remote Zero SL, 149
NRPN, 157, 158
*n*th order filter, 84
Numerically Controlled Oscillator, 63
Nyquist frequency, 20
Nyquist frequency bin, 167
Nyquist limit, 20, 66, 124, 167
Nyquist rate, 20
Nyquist-Shannon sampling theorem, 124

Oberheim 4-Voice, 53, 82
Oberheim 8-Voice, 53, 82
Oberheim Matrix 1000, 58
Oberheim Matrix 6, 49, 57
Oberheim Matrix 6R, 58
Oberheim Matrix series, 49, 54
Oberheim OB-X, 54, 55
Oberheim OB-Xa, 54, 55, 82
Oberheim, Tom, 53, 82
OBXD, 55
octave, 47
omni channel, 154
omni mode, 157
*On The Run*, 52
one pole filter, 80
one-shot envelope, 41
one-shot waves, 120
operator, 107, 114
optoisolator, 153
orbit, 122
order, 80, 113
oscillator, 14, 50, 57, 63
Overlap-Add Method, 174
overshoot, 81
overtone, 17
OXE FM synthesizer, 116

pad, 43
Palm, Wolfgang, 121
pan, 19
parametric equation, 122
paraphony, 32
part, 34
partial, 15
passband, 81
patch, 9, 14, 51
patch cable, 9, 51
patch editor, 58
patch matrix, 49, 52
PCM, 47, 120
PCM synthesis, 55, 63
Pearlman, Alan R., 52
period, 18, 30, 77
PG-8X, 55
phase, 79, 166
phase distortion synthesis, 73
phase modulation synthesis, 73, 109, 110
phase response, 79, 81, 88
phaser, 59, 142
phrase sampler, 119
physical modeling synthesis, 135, 142
piano, 147
pianoforte, 147
Pink Floyd, 52
pink noise, 65
pitch, 17, 19
pitch bend, 148
pitch bend wheel, 37, 149
pitch ribbon, 149
pitch scaling, 119, 123, 172
pitch shifting, 119, 123
PM synthesis, 73, 109, 110
pole, 80, 86, 88
Pollard Industries Syndrum, 149
poly mode, 157
PolyBLEP, 70
polyphonic, 10, 13, 32, 53
polyphonic aftertouch, 151
portamento, 28, 57, 148
PPG Wave, 82, 121

PreenFM2, 117, 118
pressure, 150
prime, 18
Princeton-Columbia Computer Music Center, 51
Prophet 5, 82
pulse code modulation, 120
pulse code modulation synthesis, 63
pulse wave, 64
pulse width, 58, 64
punch in, 45

quadraphonic, 15, 22
quality factor, 81, 92

R2-D2, 9, 52
rackmount synthesizer, 10
*Raiders of the Lost Ark*, 52
ramp function, 58
ramp wave, 37, 63
random wave, 38
rank, 23
rate-based envelope, 41
RCA Mark I / II Electronic Music Synthesizers, 51
recorder, 11
rectangle function, 128
rectangular window, 171
release rate, 41
release time, 14, 27, 33, 41
release velocity, 37, 151
repeating waves, 120
resampling, 20, 67, 123
Reserved Parameter Numbers, 157, 158
reset all controllers, 157
resolution, 37
resonance, 58, 74, 81, 92, 104
rest, 45
resynthesis, 173
return, 11
reverb, 11, 139
reverberation, 139
reversing soft sync, 77
ring buffer, 135
ring modulation, 59, 75
ringing, 81
ripple, 81
RMI Harmonic Synthesizer, 25
Roads, Curtis, 122
Roger Linn Design LinnStrument, 151
Roland, 151
Roland D-50, 120
Roland Jupiter 8, 54, 55, 82
Roland JX-8P, 55
Roland MKS-80 Super Jupiter, 55, 82
Roland TR-808, 46, 119
ROLI Seaboard, 151
roll-off, 80
ROM, 9
rompler, 9, 10, 55, 82, 120
rotary speaker, 138
RPN, 157, 158
RPN Null, 159

S&H, 40, 61
sample and hold, 40, 61, 125
sampler, 9, 10, 55
sampling, 119
sampling function, 127
sampling interval, 94
sampling rate, 20, 94
sawtooth wave, 31, 37, 58, 63
Schroeder reverberation, 140
Schroeder, Manfred, 140
second order filter, 84
sections, 46
self-centering, 149
self-resonant filter, 81, 82
SEM, 53, 82
semi-modular synthesizer, 52, 56
semitones, 19
send, 11
Sender, Ramon, 52
sequence, 44
sequencer, 10, 13, 28, 44, 108
Sequential Circuits, 151
Sequential Circuits Prophet 5, 54, 56, 151
Sequential Circuits Prophet VS, 75, 120
Short Time Fourier Transform, 142, 172
sidebands, 76, 112
sidelobe, 170
signal to noise ratio, 21

Simmons SDS-5, 149
sinc function, 69, 127
sine wave, 37, 64
single-cycle wave, 59, 67
Smith, Dave, 120, 151
Smith, Julius, 144
soft sync, 77
softsynth, 13, 55
software synthesizer, 7, 11, 13, 55
song mode, 45
sonogram, 172
spectrogram, 15, 172
spline, 126
square wave, 31, 37, 58, 63
state-variable filter, 53, 82
step sequencer, 14, 44, 61
stereo, 15
STFT, 142, 172
stop knob, 23
stopband, 81
stops, 23
stored patch synthesizer, 14
Streetly Electronics Mellotron series, 119
strike tone, 18
Strong, Alexander, 142
suboscillator, 60, 65
Subotnick, Morton, 52
subtractive synthesis, 13, 50
sustain level, 14, 27, 41
sustain pedal, 157
SWAM, 145
swell box, 148
swell pedal, 148
swing, 45–47
*Switched-On Bach*, 51
sync, 58, 59, 77
synchronous granular, 122
synth, 9
synthesis window, 173, 174
synthesizer, 9
sysex, 153
system exclusive, 153

table-based waveshaper, 70
tabletop synthesizer, 10, 14
tangent, 147
tape-replay, 119
Tasty Chips Electronics GR-1, 123
Taylor series, 164
Telharmonium, 25
tempo, 45–47
*The Day the Earth Stood Still*, 148
*The Shining*, 51
The Thing, 148
theremin, 148
Theremin, Léon, 148
third dimension controller, 157
THX, 9
tie, 45
tierce, 18
time domain, 15, 83
time resolution, 176
time stretching, 122
time-based envelope, 41
tonewheels, 23
track mute, 46
track solo, 46
tracking generator, 49, 58
trajectory, 75
transfer function, 86, 87
transient, 120
transistor ladder filter, 82, 103
transition band, 81
Trapezoidal window, 122
Trautonium, 50, 149
travel, 148
traveling wave, 144
tremolo, 25, 28, 37, 138, 148
triangle wave, 37, 58, 63
Triangular window, 122, 171
*Tron*, 51
Tukey, John, 167
two pole filter, 59, 80

unipolar, 37, 48
unity gain filter, 80, 84, 91, 128
unweighted, 147
upper zone, 161
upsampling, 124
USB host for MIDI., 160

Van Halen, 54

Vangelis, 54, 149
variable bitrate, 22
variance, 38
VCA, 60, 78
VCO, 60, 63
VCV Rack, 61
vector synthesis, 10, 75, 120
velocity, 37, 44, 78, 147
velocity sensitivity, 78, 148
vibrato, 14, 25, 28, 37, 109, 110, 138, 148
virtual analog synthesizer, 11, 55, 59, 97
Virtual Studio Technology, 13
visualization, 172
vocoder, 9, 55, 59, 173
voice, 13, 32, 53
voice stealing, 33
volt per octave, 48, 111, 159
Voltage Controlled Amplifier, 60, 78
Voltage Controlled Oscillator, 60, 63
von Helmholtz, Hermann, 23
VST, 13, 55

Wakefield, Jeremy "Jezar", 140
Waldorf Microwave, 121
Waldorf Music, 121
Waldorf Quantum, 123
wave folding, 52, 71
wave sequence, 75, 120, 122
wave shaping, 70
wave terrain, 122
waveguide, 144
wavetable, 121, 134
wavetable synthesis, 10, 26, 55, 63, 75, 121
weighted action, 147
weighted keyboard, 148
West Coast synthesis approach, 52
wet, 11, 135
white noise, 58, 64
Whittaker-Shannon interpolation formula, 127
wind controller, 150
window, 73, 122, 129, 171
windowed sinc interpolation, 112, 126
wrapping, 72

Xenakis, Iannis, 122

Yamaha CS-80, 54, 149
Yamaha DX7, 10, 109, 115
Yamaha FS1R, 107, 116, 117
Yamaha QY Series, 45
Yamaha SY/TG Series, 120
Yamaha TX81Z, 109, 116
Yamaha VL1, 144
Yamaha Vocaloid, 107

Z domain, 86, 94
zero, 80, 86, 88
zero padding, 85, 141, 174
zero stuffing, 124
zipper effect, 37