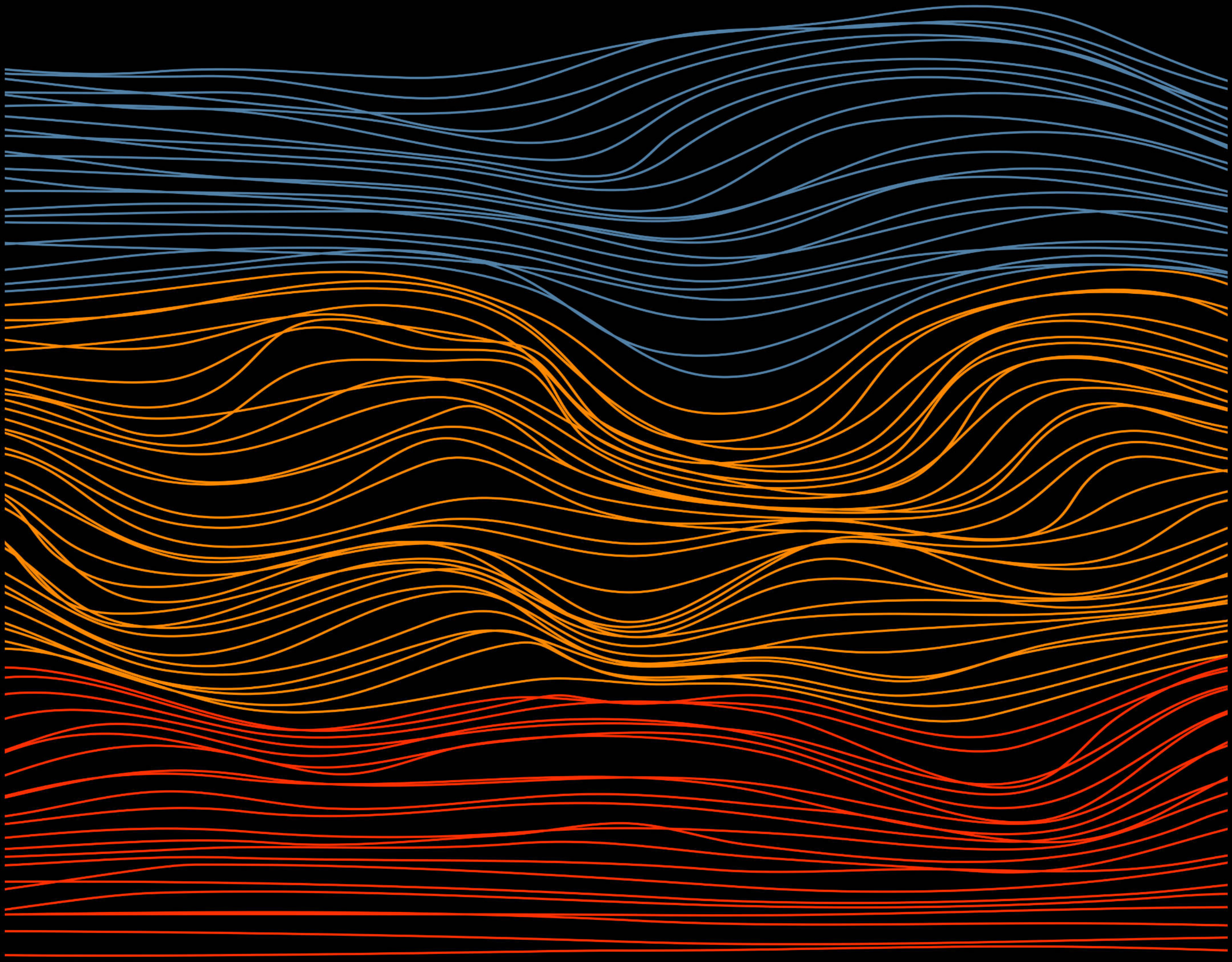
Code Your Own

Synth Plug-Ins

With C++ and JUCE

(and other frameworks too)



Matthijs Hollemans

Code Your Own Synth Plug-Ins With C++ and JUCE

Matthijs Hollemans

This book is for sale at http://leanpub.com/synth-plugin

This version was published on 2022-09-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2022 M.I. Hollemans

Contents

Acknowledgments

My deepest gratitude to Paul Kellett who wrote the original synthesizer this book is based on, and who was generous enough to open source it. Without your MDA JX10 synth, this book probably wouldn’t exist.

Also many thanks to the members of The Audio Programmer Discord server for lots of interesting discussions, and for helping me become a better audio programmer!

And cheers to all of you DSP fanatics out there sharing knowledge and code snippets through discussion forums, chat groups, mailing lists, blogs, videos, and books. May this great tradition continue forever!

About the author

Hi, my name is Matthijs Hollemans.

I'm an audio programming enthusiast and hobbyist musician. I am currently employed as a machine learning engineer.

My first ever lines of code were written in 1985 on the family Commodore 64 and I've been hacking away at creative projects ever since.

You can find my open source projects on GitHub[1].

I'm also a somewhat decent songwriter, a barely competent instrumentalist, a bedroom producer, and a horrible singer — so I won't link to any of my recordings here. ;-)

This is the eighth book that I've written or contributed to. My most notable work is *The iOS Apprentice* published by the popular tutorial website raywenderlich.com.

You can find me on Twitter as @mhollemans[2].

Enjoy the book, it was a lot of fun to write!

— Oosterhout, Netherlands, summer of 2022

[1] https://github.com/hollance
[2] https://twitter.com/@mhollemans

Introduction

How does one make sound out of nothing? In the physical world, this is not so hard: simply take a hammer and bang it against something — the louder the better!

To get a more musical sound, and fewer destroyed objects around your house, use the hammer to hit a string under tension. This produces a tone playing at a certain pitch. In fact, this is what a piano does: when you push down a key on the piano, a felt-covered wooden hammer will swing upwards to strike a string.

There are lots of other ways to produce musical sounds in the physical world, invented and perfected over millennia. If you're a musician you probably have your preferred method: plucking, strumming, blowing, bowing, striking, singing, slapping. What all these have in common is that they manipulate objects — strings, plates, membranes, columns of air, vocal cords — to make them vibrate.

But what about the world of computers? How can we make sounds with electronical circuits and software programs? You can certainly take a hammer to your computer (and who doesn't want to from time to time) but making music requires a more refined approach.

We don't have any strings, tubes, plates, or hammers to manipulate, only bits and bytes. In a synthesizer the production of sound is done completely electronically, or in the case of software, completely digitally.

I have always found this fascinating — how would that even work? How do you turn ones and zeros into new sounds?

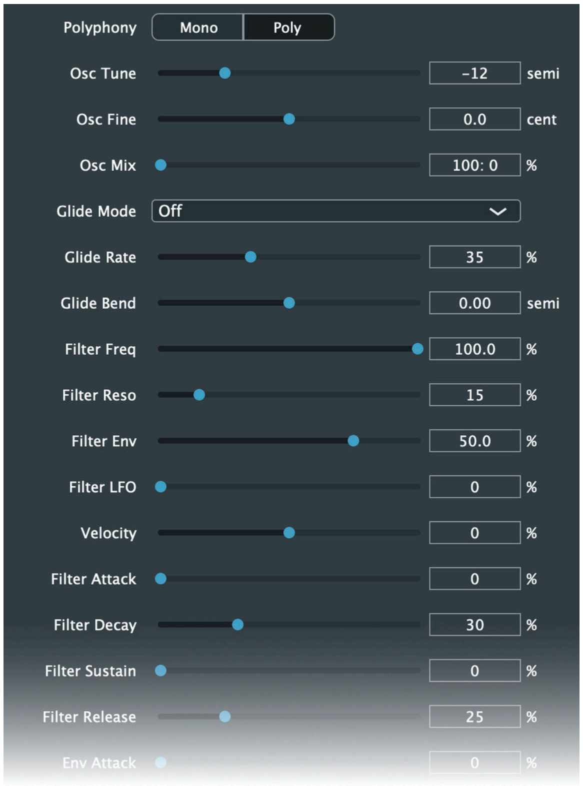
Over the past sixty or so years, people have come up with a variety of ways to produce sounds electronically and in software, and in this book you'll explore some of these methods. You will learn the principles of how synthesizers work, and how to implement this theory in real C++ code.

This book will show you how to build a synthesizer plug-in that can be used in audio software such as Logic Pro, Ableton Live, REAPER, FL Studio, Cubase, and any other DAW that supports VST or Audio Unit plug-ins.

What you will build

The synth that you'll create over the course of this book is named JX11. It's a retro synth that emulates the features of classic analog Roland hardware synths from the 1980s.

Here's what it looks like:



Polyphony Mono Poly
Osc Tune -12 semi
Osc Fine 0.0 cent
Osc Mix 100: 0 %
Glide Mode Off
Glide Rate 35 %
Glide Bend 0.00 semi
Filter Freq 100.0 %
Filter Reso 15 %
Filter Env 50.0 %
Filter LFO 0 %
Velocity 0 %
Filter Attack 0 %
Filter Decay 30 %
Filter Sustain 0 %
Filter Release 25 %
Env Attack 0 %

The user interface of JX11

More importantly, this is what it sounds like: Follow this link to listen to audio demos[3].

The user interface shown above is automatically generated from the synth's parameters. It's fully functional but, admittedly, not very good looking.

This book is primarily about the audio programming part of the synthesizer, not the user interface, and so we'll focus on the code for doing digital signal processing or DSP. Making a cool user interface for this synth is left as an exercise for the reader. But don't worry, chapter 13 will explain everything you need to give your synths a shiny UI with proper knobs and stylish looks.

[3] https://github.com/hollance/synth-plugin-book/tree/main/Audio%20Demos

Play the synth!

You might be curious exactly what this JX11 synth will sound like when it's done. The best way to find out is to grab the finished synth and play with it.

There currently are no binary downloads for the plug-in, so you will have to compile it yourself. Here is a quick start guide on how to get the source code, compile it, and run the synth. If you run into any issues, skip ahead to chapter 2 to learn more about setting up JUCE and using the Projucer.

1. Download[4] and install JUCE.
2. Clone the GitHub repo[5] for the book.
3. Double-click the **JX11.jucer** file from the folder **Finished Project**. This should open Projucer.
4. In Projucer, choose **File** > **Save Project and Open in IDE…** This should launch your development environment.
5. In your IDE, activate the **JX11 Standalone Plugin** target, then build and run.
6. If all went well, the synth should open in its own app. Click the **Options** button at the top and select your MIDI input device, if you have one. Now you can play the synth!

The synth comes with more than 50 factory presets (ready-made sounds) but unfortunately the standalone app has no menu for choosing these presets. To try out the presets and enjoy JX11 in its full glory, it's best to load it as a plug-in into a DAW. That also lets you use the synth if you don't have a MIDI keyboard.

Instructions for building the synth as a plug-in:

a. Back in the IDE, activate the **JX11 VST3** target and build. This will create the VST3 plug-in version of the synth.

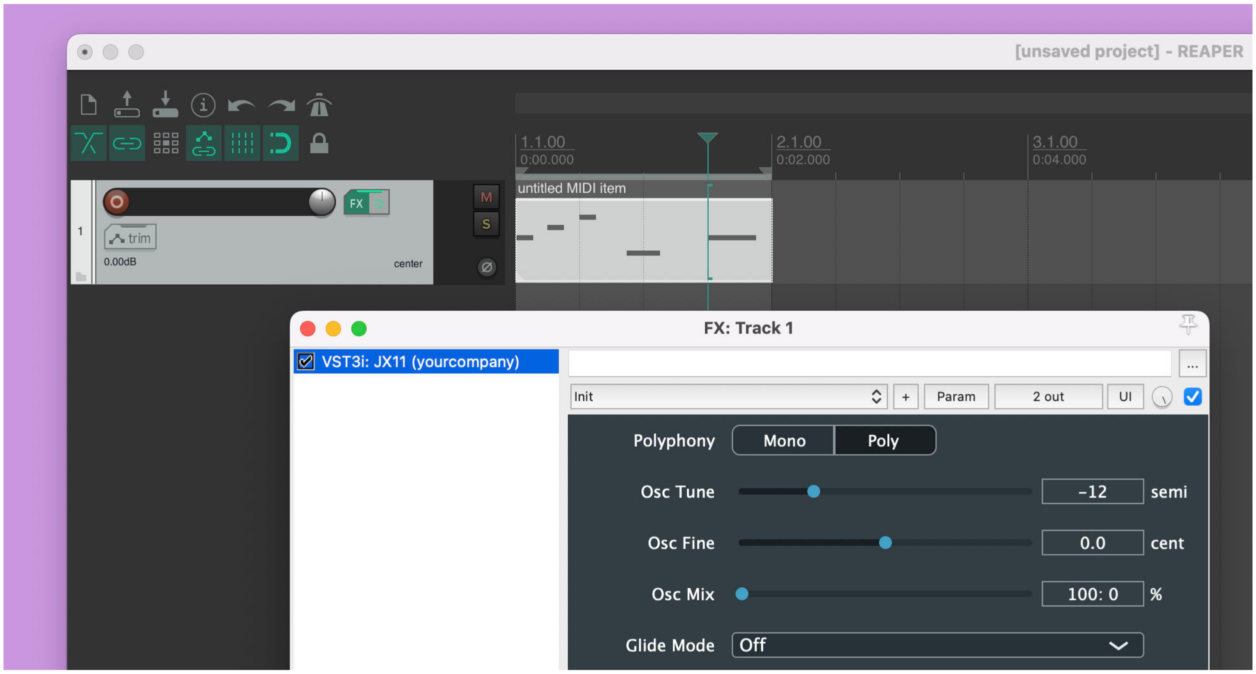
b. Verify that the **JX11.vst3** file was copied into the folder where the DAW will look for plug-ins. On Mac this is `/Users/yourname/Library/Audio/Plug-Ins/VST3`. On Windows this is `C:\Program Files\Common Files\VST3`. If not, copy the file from the Builds folder by hand (you may need to change permissions for this in Windows).

[4] https://juce.com/get-juce/download

[5] https://github.com/hollance/synth-plugin-book

c. On Mac, you can also build the **JX11 AU** version. This is necessary to use the plug-in with Logic Pro or GarageBand.

If all went well, you can find the plug-in in your DAW under **yourcompany** > **JX11**. You may need to restart the DAW before the new plug-in shows up. The presets can be accessed through the DAW's factory presets menu. Below is a screenshot of JX11 in REAPER.



Polyphony: Mono | Poly

Osc Tune: -12 semi

Osc Fine: 0.0 cent

Osc Mix: 100: 0 %

Glide Mode: Off

The synth on a track in REAPER

Have fun playing with this synth! It can make some really interesting sounds — for a laugh, try the Squelchy Frog preset.

In the rest of the book, we'll look at every single piece of code that makes up this synth, and you'll learn what it does, how it works, and why it's done that way.

The MDA plug-ins

JX11 is based on the MDA JX10 synthesizer from maxim digital audio.

The MDA plug-ins were a set of free audio plug-ins developed by Paul Kellett about two decades ago. The MDA plug-in suite consisted of a wide variety of audio effects, such as a limiter, overdrive, multi-band distortion, stereo widening, vocoder, and many others. Also included were a couple of synths, including the virtual analog synth JX10.

You can find the original MDA plug-ins and their source code at this link[6].

The MDA plug-ins were written for the VST2 standard but this is now ancient and deprecated technology. Fortunately, Paul made the source code available as open source, and over the years people have ported these plug-ins to new frameworks such as VST3 and LV2.

When I got into audio programming, I decided to convert the MDA plug-ins to JUCE, mostly as a learning exercise for myself. But I wanted to do more than hack the code to make it work with the JUCE API — my main motivation for doing this work was to understand how these plug-ins worked. And so I ended up renaming variables, cleaning up the logic, and adding plenty of documentation in the source code.

You can find the JUCE versions of the MDA plug-ins on my GitHub[7].

As I dug into the MDA source code, I realized that it was a treasure trove of DSP knowledge. Studying this collection of plug-in code is pretty much a complete starter course in DSP programming, even if the code is over twenty years old.

The JX10 synth in particular gives us a great opportunity to peek under the hood of a real synthesizer to see everything that goes on. This is not some basic demo project that someone put together for a tutorial — it's a serious product that musicians have used in real productions!

More about MDA JX10

JX10 is a basic but capable virtual analog synth that can make some sweet sounds. It's from 2001, so it's obviously not the latest and greatest in VA synthesis, but the synth can still hold its own. Read for yourself what KVR users had to say[8] about the original plug-in.

In this book you will use the design and implementation of MDA JX10 to learn about creating software instruments and audio plug-ins in general. Of course, there are many other ways to build synths, but this is a great place to start from if you want to learn how to build your own software synthesizer!

JX10 is a bit quirky, which, depending on your perspective, means it has bugs or that it has character. That makes it a great learning tool because some of these issues aren't immediately obvious and they easily can creep into your own synth code too if you don't understand them.

[6] http://mda.smartelectronix.com
[7] https://github.com/hollance/mda-plugins-juce
[8] https://www.kvraudio.com/product/jx10_by_mda/reviews

Some of the DSP techniques used by this synth are now outdated. But that isn't necessarily a bad thing: we can study these shortcomings and learn something in the process. The book suggests plenty of improvements that you can explore to make this synth even better.

This book is basically a tour of the source code of MDA JX10 but cleaned up and modernized significantly to make it easier to understand, with lots of explanations and illustrations. I've called this new version of the synth JX11 to honor the original.

Why did I write this book?

That's simple: there aren't enough tutorials about writing synths!

Audio programming is a niche field and there aren't a lot of books about it. The teaching materials that do exist tend to focus more on effects and sound processing than on synthesis. A lot of sound synthesis and DSP knowledge can be found in academic papers but these papers are often heavy on the math and are written for other academics, not for audio programming enthusiasts.

Fair enough, there are a few synthesizer books available, and if you're into building synths you should probably get them all. But there's still a big difference between reading the theory and putting it all together into an actual finished product.

As I was converting the MDA JX10 synth to JUCE, it dawned on me that this could serve as a great introduction to writing a real synthesizer. The project is not trivial, it has some interesting oddities, and makes for a good story.

So I dusted off the source code, fixed some of the issues, and wrote this book to explain in detail how it all works and why it was designed that way. Essentially, these are my own notes from when I was trying to figure out how this synth worked but written for someone who is new to audio programming.

This isn't the only way to build a synth plug-in, or necessarily the best way, but by following along with this book you should be able to get a good sense of everything that's involved and how it all fits together. And then use this as a starting point for making your own synths.

> **Tip:** When you've finished with this book, I suggest you take a look at the other MDA plug-ins on my GitHub. Lots of other great stuff in there to be discovered!

It’s the JUCE that makes it juicy

This book uses the JUCE framework[9] as that is the most convenient way to build audio plug-ins that will work on macOS, Windows, Linux, and iOS.

JUCE is a wrapper around the various plug-in formats, most notably VST3, Audio Units, AAX, and LV2. Since it’s a hassle to rewrite your plug-in for each of the different formats, many audio developers use JUCE or another wrapper library to handle the platform-specific implementation details. You write the plug-in just once and let JUCE deal with the underlying audio APIs. In addition, JUCE has cross-platform support for creating user interfaces, so that you also have to write the UI only once.

That’s why this book uses JUCE: so you don’t have to worry about the different plug-in formats, audio APIs, or operating systems and you can focus on the fun part: writing the sound synthesis code.

JUCE has a DSP module that includes ready-made building blocks for audio programming, such as delay lines, oscillators, filters, and so on. It even comes with a basic synthesizer class. We won’t be using that part of JUCE in this book, except for certain optional sections. The whole point of this book is to learn how to implement a synthesizer from scratch, so we’re not going to “cheat” by using pre-made modules for this particular synth.

There are alternatives to JUCE, such as iPlug2[10] or using the VST and Audio Unit SDKs directly, but this book does not cover those other frameworks. Or perhaps you’re interested in making a synth for a device that doesn’t even have a plug-in SDK, such as an embedded device. The good news is: the techniques shown in this book apply even if you’re using a different audio framework. I’ve kept the usage of JUCE to a minimum on purpose, so that it should be fairly straightforward to adapt the code to your audio API of choice.

If for some reason you can’t or don’t want to use JUCE, these are the things you will need to change in the source code:

- the audio callback (chapter 3)
- handling MIDI input (chapter 4)
- the parameter definitions (chapter 7)
- state saving and restoring (chapter 7)
- the user interface (chapter 13)

[9] https://juce.com
[10] https://iplug2.github.io

How to use this book

What you need:

- At least basic C++ skills. Many software developers shy away from using C++ because it has a bit of a reputation, but the fact is that most audio programming these days is done in C++. If your C++ is rusty, I can recommend Stroustrup's *A Tour of C++*. There are also many YouTube videos[11] about learning C++.
- Some familiarity with music making. I will be using terms like "pitch" (how high or low a sound is) and "semitone" (the distance between two adjacent notes in the Western music scale) and "legato" (a style of playing). You don't need to be an expert synth player but it's useful to know some of the terminology.
- A Mac or a PC running Windows or Linux.
- An IDE. On the Mac this is usually Xcode, on Windows this is Visual Studio. Both can be downloaded for free. On Linux you can use your editor of choice and build the projects from a Makefile.

  Important: For Visual Studio be sure to get the IDE[12], not VSCode. While it is possible to build JUCE plug-ins using VSCode, this book does not discuss that workflow.
- JUCE. This is the framework that makes it easy to build audio plug-ins. We'll cover installation of JUCE in chapter 2. As mentioned above, it's possible to use other audio frameworks but you'll have to do some additional work to translate the JUCE-specific code to the other API.
- Some kind of MIDI keyboard. Don't worry, if you don't have a MIDI keyboard, you can use a DAW that lets you create MIDI sequences. REAPER[13] is a popular DAW that is free to try. In a pinch you can also use the on-screen virtual MIDI keyboard in JUCE's AudioPluginHost. But if you're serious about making synths, you should think about getting a good MIDI controller.

All the source code you need is listed in this book. You'll build up the JX11 synthesizer project step-by-step, in tutorial fashion. That means you should really read this book from front to back. If you skip sections, you might miss out important code snippets.

Maybe it's my age and I fondly remember typing in source code listings from computer magazines, but I do believe that it's good to type in all the source code yourself. It's

[11] https://www.youtube.com/c/TheChernoProject
[12] https://visualstudio.microsoft.com
[13] https://www.reaper.fm

more work than copy-pasting the code but typing everything in does make you pay more attention to what's going on. Besides, copy-pasting from PDF files doesn't always work very well.

You can also follow along with the source code on GitHub[14]. The source code in the git repo is split up by chapter. You can also find any errata there.

If you're unfamiliar with git or GitHub, go to the link, click the big green Code button and choose Download ZIP from the popup. This downloads all the source code as a zip file to your computer.

Aargh… Math?!?!

Now for the elephant in the room: Do you need to know math to do audio programming?

I'm not going to lie: knowing math certainly helps! However, you can get pretty far with just basic knowledge. If your math skills aren't strong, don't let that stop you from making synths. This book is written so that you should be able to follow along even if your math is not very good.

However, if this book gets you all excited about audio programming and you're planning to make a career out of it, eventually you'll need to brush up on your math. There's no shame in this, I did it myself. A few years ago, I bought some beginner math books for adults and worked my way back up from early high school level math to college level math. It's not easy, but it's worth it, especially in the field of DSP.

Credits

The original MDA JX10 synthesizer was written by Paul Kellett of maxim digital audio. The factory presets included in the synth are designed by Paul Kellett with additional patch design by Stefan Andersson and Zeitfraktur, Sascha Kujawa.

Thanks for creating the MDA plug-ins, Paul, and especially for open sourcing them!

JUCE is copyright © Raw Material Software.

The Lato font is copyright (c) 2010-2014 Łukasz Dziedzic and is licensed under the SIL Open Font License.

Cover illustration made from artwork by Brett Jordan[15]. License: CC-BY-2.0

[14] https://github.com/hollance/synth-plugin-book
[15] https://bit.ly/3J9TXT9

Chapter 1: What is a synth, exactly?

To make sound, molecules in the air need to be moved back and forth with enough force so that our ears can pick up the vibrations. These air vibrations must happen quite rapidly for the human brain to register it as sound, between 20 and 20,000 times per second.

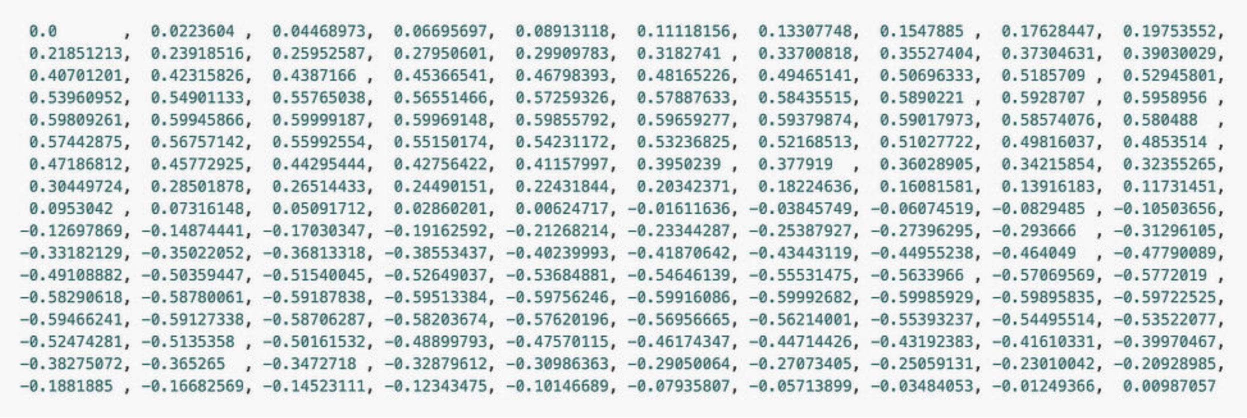
To become music, the air molecules should also move in a way that feels pleasant to listen to — or unpleasant if grindcore is your thing.

As a programmer, you don’t need to worry about physically making the air vibrate. The computer’s loudspeakers take care of this. You do need to describe to the computer exactly how you want the loudspeakers to move. This description consists of a stream of 44100 or more numbers per second.

The computer’s DAC or digital-to-analog converter turns this stream of numbers into electrical signals that rapidly move the electromagnetic coil in the loudspeaker back and forth, which in turn causes the speaker’s diaphragm to push and pull against the air. The resulting air movement is what we call sound.

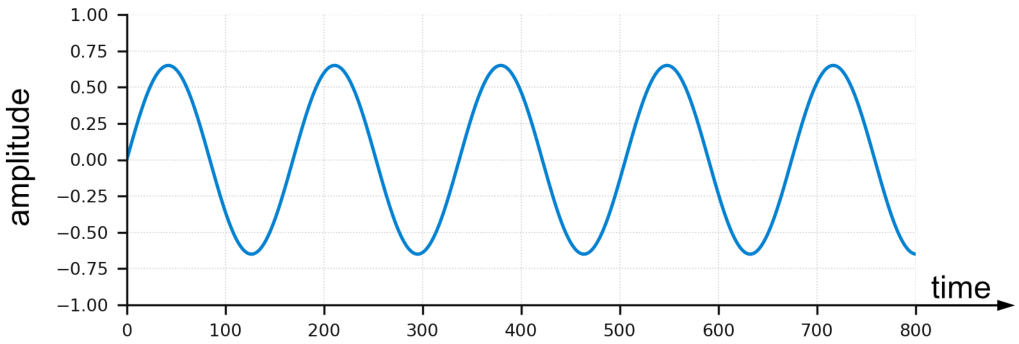
DSP programming is all about dealing with such streams of numbers. In JUCE and most other modern audio APIs, these numbers are floating point values between –1.0 and +1.0.

For example, if you tell the computer to output the following sequence of numbers through its DAC, you will hear a sound with a frequency of 261.63 Hz, which is the pitch of middle C on the piano.



```
 0.0       ,  0.0223604 ,  0.04468973,  0.06695697,  0.08913118,  0.11118156,  0.13307748,  0.1547885 ,  0.17628447,  0.19753552,
 0.21851213,  0.23918516,  0.25952587,  0.27950601,  0.29909783,  0.3182741 ,  0.33700818,  0.35527404,  0.37304631,  0.39030029,
 0.40701201,  0.42315826,  0.4387166 ,  0.45366541,  0.46798393,  0.48165226,  0.49465141,  0.50696333,  0.5185709 ,  0.52945801,
 0.53960952,  0.54901133,  0.55765038,  0.56551466,  0.57259326,  0.57887633,  0.58435515,  0.5890221 ,  0.5928707 ,  0.5958956 ,
 0.59809261,  0.59945866,  0.59999187,  0.59969148,  0.59855792,  0.59659277,  0.59379874,  0.59017973,  0.58574076,  0.580488  ,
 0.57442875,  0.56757142,  0.55992554,  0.55150174,  0.54231172,  0.53236825,  0.52168513,  0.51027722,  0.49816037,  0.4853514 ,
 0.47186812,  0.45772925,  0.44295444,  0.42756422,  0.41157997,  0.3950239 ,  0.377919  ,  0.36028905,  0.34215854,  0.32355265,
 0.30449724,  0.28501878,  0.26514433,  0.24490151,  0.22431844,  0.20342371,  0.18224636,  0.16081581,  0.13916183,  0.11731451,
 0.0953042 ,  0.07316148,  0.05091712,  0.02860201,  0.00624717, -0.01611636, -0.03845749, -0.06074519, -0.0829485 , -0.10503656,
-0.12697869, -0.14874441, -0.17030347, -0.19162592, -0.21268214, -0.23344287, -0.25387927, -0.27396295, -0.293666  , -0.31296105,
-0.33182129, -0.35022052, -0.36813318, -0.38553437, -0.40239993, -0.41870642, -0.43443119, -0.44955238, -0.464049  , -0.47790089,
-0.49108882, -0.50359447, -0.51540045, -0.52649037, -0.53684881, -0.54646139, -0.55531475, -0.5633966 , -0.57069569, -0.5772019 ,
-0.58290618, -0.58780061, -0.59187838, -0.59513384, -0.59756246, -0.59916086, -0.59992682, -0.59985929, -0.59895835, -0.59722525,
-0.59466241, -0.59127338, -0.58706287, -0.58203674, -0.57620196, -0.56956665, -0.56214001, -0.55393237, -0.54495514, -0.53522077,
-0.52474281, -0.5135358 , -0.50161532, -0.48899793, -0.47570115, -0.46174347, -0.44714426, -0.43192383, -0.41610331, -0.39970467,
-0.38275072, -0.365265  , -0.3472718 , -0.32879612, -0.30986363, -0.29050064, -0.27073405, -0.25059131, -0.23010042, -0.20928985,
-0.1881885 , -0.16682569, -0.14523111, -0.12343475, -0.10146689, -0.07935807, -0.05713899, -0.03484053, -0.01249366,  0.00987057
```

Of course, a list of numbers isn’t particularly easy to read, so we will usually plot this in a graph, such as the one shown on the next page.



Sine wave with a frequency of 261.63 Hz

This is a picture of a sine wave, made by the mathematical function `sin()`. The amplitude is on the vertical axis and describes the position of the loudspeaker's diaphragm at a given instance in time. The time is on the horizontal axis, going from left to right.

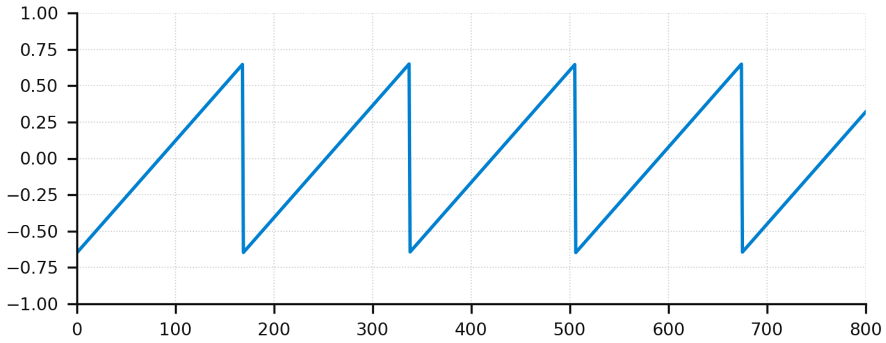
Any individual point on this graph is called a sample. For this particular sine wave, it takes 44100 of these samples to output one second of sound. The 800 or so samples in the image above describe only a tiny fraction of time, about 18 milliseconds worth of audio.

Ultimately, a synth is nothing more than a computer program that produces a stream of numbers that create interesting patterns of air movement.

The sawtooth oscillator

The sine wave from the picture above was produced by a so-called oscillator. There are many ways to build synthesizers but they all use one or more oscillators that get combined in various ways.

The synth that you'll be making in this book uses an oscillator that outputs a sawtooth wave:



Sawtooth wave with a frequency of 261.63 Hz

This waveform looks straightforward enough, just a straight line that zigzags up and down. It's not hard to come up with a mathematical function that describes this waveform. However, correctly implementing a sawtooth oscillator is actually trickier than you might think. More about this in chapter 6.

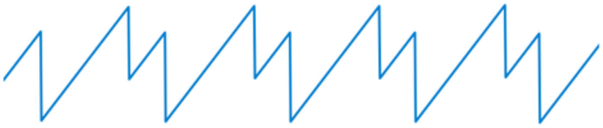
The sawtooth wave is a popular choice amongst synth designers because it's much more appealing than a plain sine wave.

A sine wave consists of only a single frequency and therefore sounds rather… boring. Want to know how boring? Grab your phone and tap the numeric keypad for making a call. I'm sure you'll agree with me that the beeps this makes are quite bland, but they are already more interesting than a sine wave! These so-called DTMF tones are made by combining two sine waves and as such consist of two frequency components instead of just one.

A sawtooth wave at the same pitch as a sine wave has the same fundamental frequency but it also has a bunch of additional frequencies known as harmonics. Those extra frequencies make the sawtooth more exciting to listen to because there's a lot more going on, sonically speaking. The frequencies that are in the sound, known as its spectrum, are what creates the timbre of the instrument. Different waveform shapes result in different timbres.

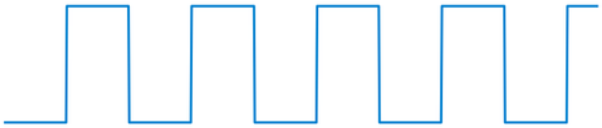
The JX11 synth has not one but two sawtooth oscillators. By making the second oscillator run at a slightly different pitch than the first oscillator, known as detuning, the sound becomes even more complex — and thereby, more compelling.

The following picture shows the result of two sawtooth oscillators mixed together, where the second one is detuned by 10 cents. Note that a cent is 1/100th of a semitone.



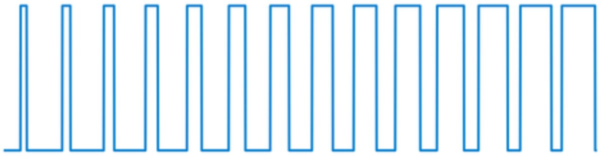
Two sawtooths with the second one slightly detuned

The two sawtooths can also be combined to make a so-called square wave or pulse wave. This has its own particular sound because it has different harmonics than only a single sawtooth wave.



Square wave, also known as pulse wave

By smoothly changing the relative positions of the two sawtooths over time, the width of the pulse changes, a classic technique known as pulse width modulation (PWM):



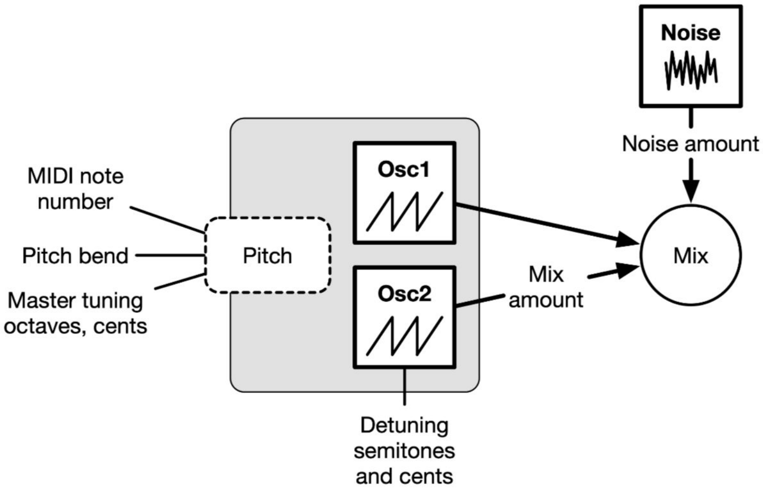
Pulse width modulation

As you can see from these examples, using a couple of simple oscillators together can already produce rich, flavorful waveforms. This was the technique used by the earliest analog synthesizers, which is why JX11 is a “virtual analog” synth.

Even though this idea of combining simple oscillators goes back a long time, it is still a common way to make synths. Since those early days, many other techniques have been invented for making sound. JX11 mixes the two oscillators together by simply adding them up. To get a different effect, you can multiply them together instead. This is known as ring modulation or amplitude modulation. Another thing you could do is use the first oscillator to set the pitch of the second oscillator, known as frequency modulation (FM).

Why stop at two oscillators? With three you can do something called vector synthesis, where the ratios of how the three oscillators are mixed can change dynamically over time. And so on… the possibilities are endless. For the synth in this book, you’re going to keep it uncomplicated and stick with two oscillators.

When designing a synth, it’s a good idea to draw a schematic of how the synth works. Here is the oscillator portion of JX11:



The oscillators in JX11

The pitch of the oscillators is determined by the note that's playing and the amount of pitch bending applied by the pitch wheel on the MIDI controller. Besides detuning Oscillator 2 relative to Oscillator 1, JX11 also lets you set the master tuning in octaves and cents. Pitching the sound down by an octave or two is useful for bass sounds. Changing the tuning in cents is useful for playing along with music that isn't entirely in tune.

Notice the block in the schematic that says Noise. It's common for analog synths to mix a certain amount of white noise into the sound as well. This isn't really an oscillator as such, just a random number generator, but a bit of noise can definitely add flavor to the sound.

In the rest of this chapter, we'll be adding more pieces to this schematic. The oscillators are only the beginning!

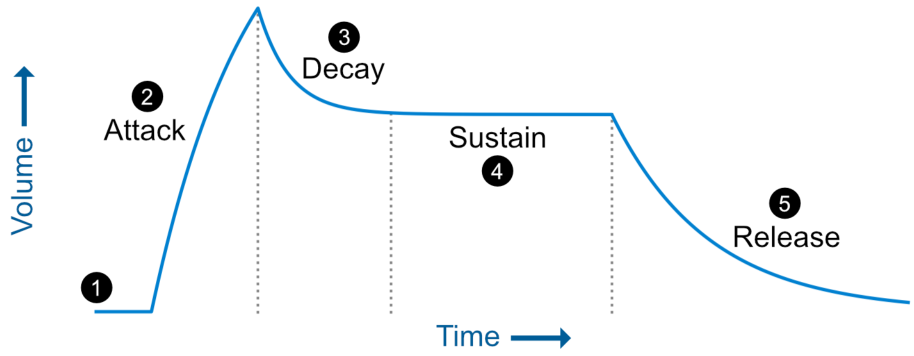
It sounds better if it changes

We now have a way to make the computer's speaker cone vibrate in proportion to the up and down motion described by the oscillator. This certainly produces sound. However, it's a very steady, monotonous sound. It instantly starts and keeps playing back the same thing over and over until you stop the oscillator.

Human ears get tired of repetitious sounds very quickly and the oscillator by itself won't hold our interest for more than a couple of seconds. To make sounds that keep listeners wanting to hear more, the synth will have to vary the oscillator output over time somehow.

An essential part of every synthesizer ever made is the amplitude envelope. The envelope quickly fades in the sound when it starts, holds it steady for the duration, and then fades it out again after the sound stops.

The ADSR envelope is the most common type of envelope and looks like the following. If you've ever used a synthesizer, you'll be familiar with it.

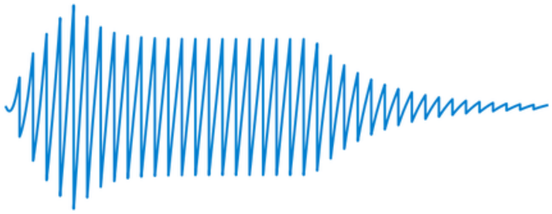


The ADSR envelope

The graph describes how the volume, or loudness, of the sound changes over time:

1. At the beginning there is no sound and thus the volume is 0.
2. When a new note is played, the attack stage fades in the envelope until it reaches the maximum volume level. Since we use floating point numbers, the maximum volume level is 1.0.
3. After the envelope reaches its peak, the volume is gradually decreased again during the decay stage.
4. Once the attack and decay stages have completed, the volume stays at the sustain level for as long as the note is playing.
5. Finally, when the note should stop playing, the release stage fades out the sound until the volume is zero again.

Here is what such an envelope looks like applied to the sawtooth oscillator:



The ADSR envelope applied to the sawtooth wave

The oscillator produces the steadily repeating sawtooth waveform as before, but now is multiplied by the envelope to produce a sound that is more dynamic and therefore more interesting to listen to.

By setting the times for the attack, decay, and release stages, as well as the volume level for the sustain stage, the user can design a whole range of different sounds, from aggressive stabbing tones with a fast attack and a quick decay, to slow droning sounds that gradually fade in and out.

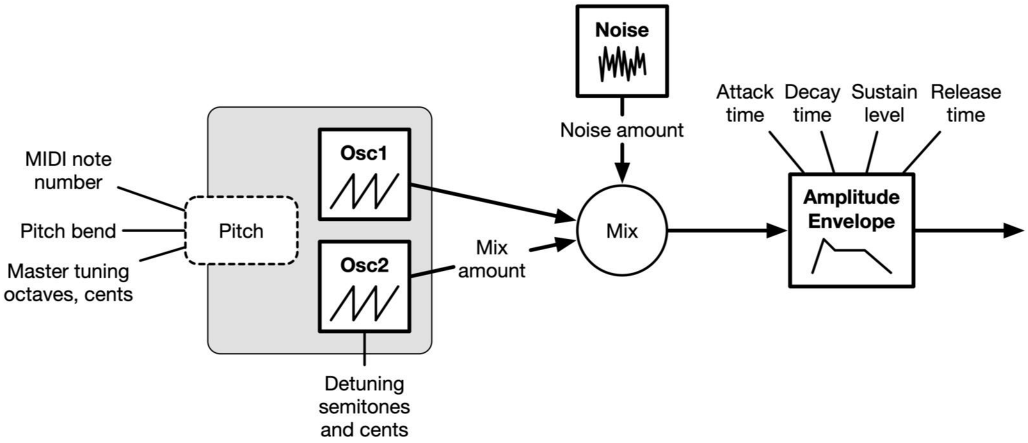
Notice how the envelope isn’t made of straight lines but is curved. This is typical for the kinds of analog synths that JX11 is based on. These synthesizers used capacitors that were charged and discharged over time to create the voltage levels for the envelope.

Our digital version of the synth keeps this curved shape because human hearing is logarithmic in nature. This means we can hear differences between soft sounds very well, but the louder the sound becomes, the less sensitive our ears are. To make a sound fade out evenly, it has to be the opposite of logarithmic, which is exponential, and that’s exactly the kind of curve the envelope shows.

Readers with a keen eye might have spotted that the attack is exponential the wrong way around. That makes it quicker than a regular fade in, but this actually sounds better. Many real instruments have a fast attack from the burst of energy that is released when the hammer hits the piano string or from the pluck of a guitar pick.

Modern synths may offer a choice of different kinds of curves for each envelope stage, allowing the user to tweak the envelope to be exactly as they like it.

After incorporating the envelope, the schematic of the synth looks like this:



The oscillators feed into the amplitude envelope

The amplitude envelope is only one of several techniques that you will use to shape the sound over time. The next one we’re going to look at is the filter.

Filtering the sound

Arguably the most important part of a (virtual) analog synth is the resonant filter. The filter is what gives the synth the ability to make a wide range of sounds.

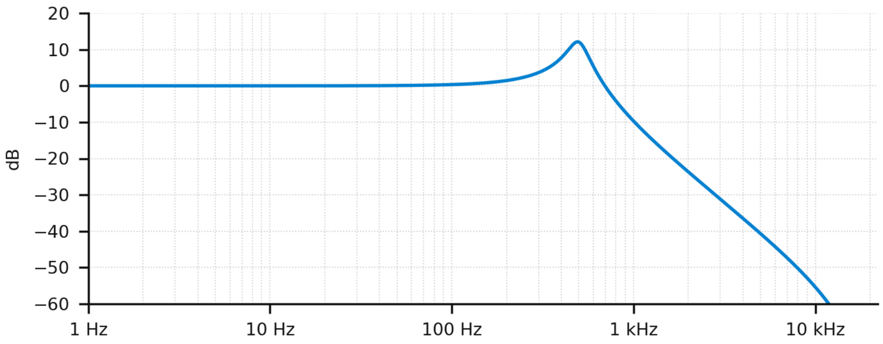
A filter is used to remove frequencies from the audio signal. This is why a synth that is made up of oscillators that are followed by a filter is said to use subtractive synthesis. You start with a sound that has lots of harmonics in it and then use the filter to remove, or at least attenuate, some of these harmonics.

But synthesizers don't just use any old filter — this is a filter that can resonate. That means the filter can also boost certain frequencies to make them louder.

This is another reason why sawtooth or square wave oscillators are used instead of plain sine waves. The filter can't add new frequencies to the sound, it can only change frequencies the sound already has. Where a sine wave contains just a single frequency, sawtooth and square waves have many additional frequencies, the harmonics. The existence of such harmonics gives us more opportunities to modify the frequency content of the sound.

The filter in JX11 is a so-called low-pass filter because it passes through the low frequencies unchanged, but above a certain point it will fade out the higher frequencies. This point is known as the cutoff frequency. In addition, the filter can boost frequencies around the cutoff point using the resonance control, often called Q (short for "quality factor").

The operation of a filter is usually drawn as a frequency response plot:



The frequency response of a low-pass filter with cutoff = 500 Hz and Q = 4

The horizontal axis shows the frequencies from low to high. This axis is logarithmic because we hear frequencies in a logarithmic fashion, similar to how we perceive loudness. Our ears are very sensitive to sounds with low frequencies (down to about 20 Hz), but the higher the frequency goes, the less clearly we hear the differences. For example, the transition from a 200 Hz tone to a 300 Hz tone is extremely obvious but the same 100 Hz difference between 9800 Hz and 9900 Hz is much less noticeable.

The vertical axis shows the magnitude response for the frequencies, or how much the loudness of each frequency is affected by the filter. In the image above, the magnitude response is a flat line up until the cutoff frequency at 500 Hz, then it goes up a bit for the resonance boost, and beyond that it quickly drops off.

This axis is also logarithmic. The scale used here is the decibel or dB scale, where 0 dB means that the sound level does not change, positive dB means the sound is amplified, and negative dB means the sound is attenuated (reduced in volume).

The filter envelope

The filter becomes even more interesting when you apply an envelope to it. This raises and lowers the filter's cutoff frequency over time. Whereas the envelope from the previous section affected the amplitude or loudness of the sound, the filter envelope will alter the frequencies that are in the sound. With fancy words: it modifies the spectral content.

Recall that steady sounds quickly become stale to the human ear. Regularly changing what frequencies are active, by doing so-called filter sweeps, allows us to keep our listeners' ears entertained.

In many acoustic instruments, the higher frequencies decay faster than the lower frequencies. The tone is brighter when the note is first struck but becomes duller as the tone fades out. To do this in a synth, you'd start with the filter cutoff at a high value and use the envelope to gradually lower it.

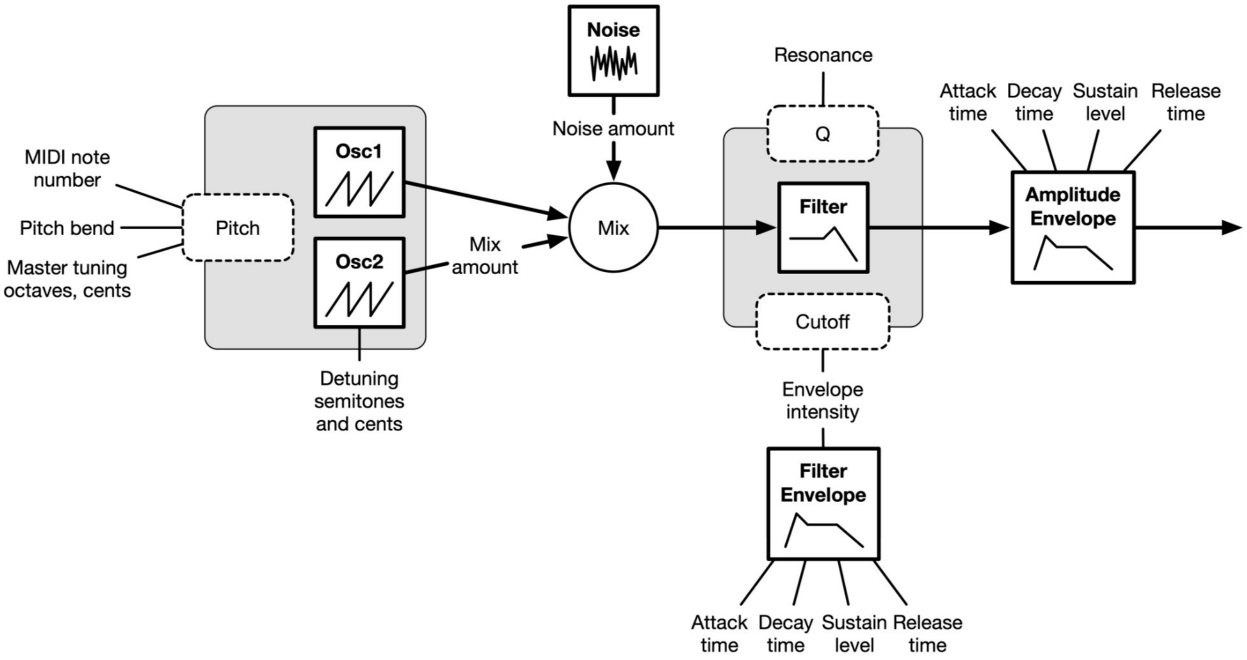
The filter envelope has the same ADSR shape as the amplitude envelope, with attack, decay, sustain, and release sections. For the amplitude envelope it was pretty clear how to apply it: simply multiply the waveform coming out of the oscillator with the envelope. However, if you were to do this with the filter, the cutoff frequency would always swing between the lowest and highest possible frequencies, which is a bit extreme.

To limit the range of frequencies involved, there's a setting for how much effect the filter envelope should have: the filter envelope intensity or depth. This setting ranges from –100% to +100%.

- At 0%, the filter envelope has no effect at all and the filter will always have a fixed cutoff frequency.
- If the amount is between 0% and 100%, the sound becomes brighter in the attack phase, a little less bright in the decay phase, remains steady while the note plays at the sustain level, and finally drops back to the original cutoff point during the release stage.
- The amount can also be negative, in which case the envelope is flipped around. Now the attack makes the cutoff go lower and the sound initially becomes duller. The decay and release stages bring the cutoff frequency back up again.

It may take a little getting used to how the filter envelope works, but it is one of the key features for doing sound design using a virtual analog synth.

The schematic for the synth with the filter section added:



The filter with its envelope

Modulation is what makes it exciting

Another thing you can do to vary the sound over time is apply modulation. This means changing one parameter of the synth using another parameter. Modulation is a powerful technique that gives synths the ability to make many kinds of sounds.

You've already seen an example of modulation: the envelopes. The amplitude envelope modulates the loudness of the oscillators, while the filter envelope modulates the filter's cutoff frequency. In certain synths, you're given two or more envelope generators that can be used to modulate whatever you want, whether that is the amplitude, the filter, or anything else. In JX11, however, the envelopes have fixed functions.

Another common modulator is the velocity. This measures how hard — or really how fast — the user presses the keys on their MIDI controller keyboard.

Velocity is usually connected to the loudness of the sound: the higher the velocity, the louder the note. In addition to the amplitude envelope, the oscillator output is multiplied by the velocity to get the final waveform amplitude. That is also the case in JX11, although there is an option to disable velocity so that all notes play with the same loudness.

In JX11, the velocity can also be used to set the filter's cutoff frequency. The harder you press the key, the higher the filter's cutoff frequency becomes. This is how many real instruments work, too: the louder you play the piano, the brighter the sound. The user can set the sensitivity for how much the velocity affects the filter. If this is 0, velocity has no effect on the filter cutoff. It can even be a negative number, in which case the cutoff frequency drops lower the harder you strike the key.

The filter cutoff is also determined by the note that you're playing, a practice known as key tracking. This uses the MIDI note number as a modulator. In JX11 this is hardcoded to set the cutoff frequency based on the pitch of the note. The user has a manual control for tweaking this, so that the cutoff can be set higher or lower, but it's always relative to the MIDI note number — and affected by the velocity.

Besides velocity and envelopes, another typical modulator is the LFO or low frequency oscillator. This is an oscillator just like the ones we talked about before, but it runs at a frequency that is too low to be audible. If you output a sine wave with a frequency of 20 Hz or more, it produces sound. But if the frequency is less than approximately 20 Hz, our brains do not register it as sound anymore.

LFOs are typically used to slowly modify one or more aspects of the sound over time. Unlike an envelope, the LFO repeats itself, so the changes it makes to the sound are periodic.

In JX11, the LFO is a plain sine wave. In other synths you can often choose different waveforms for the LFO, such as sawtooth and square wave but also triangle wave, random numbers, exponential shapes, and so on. Each of these lets you create different effects. Often there is more than one LFO and you can assign each LFO to multiple modulation targets.

JX11 has a single LFO. It can be used to modulate the filter and/or the pitch of the sound. The user can set the speed at which the LFO oscillates between 0.01 and 20 Hz.

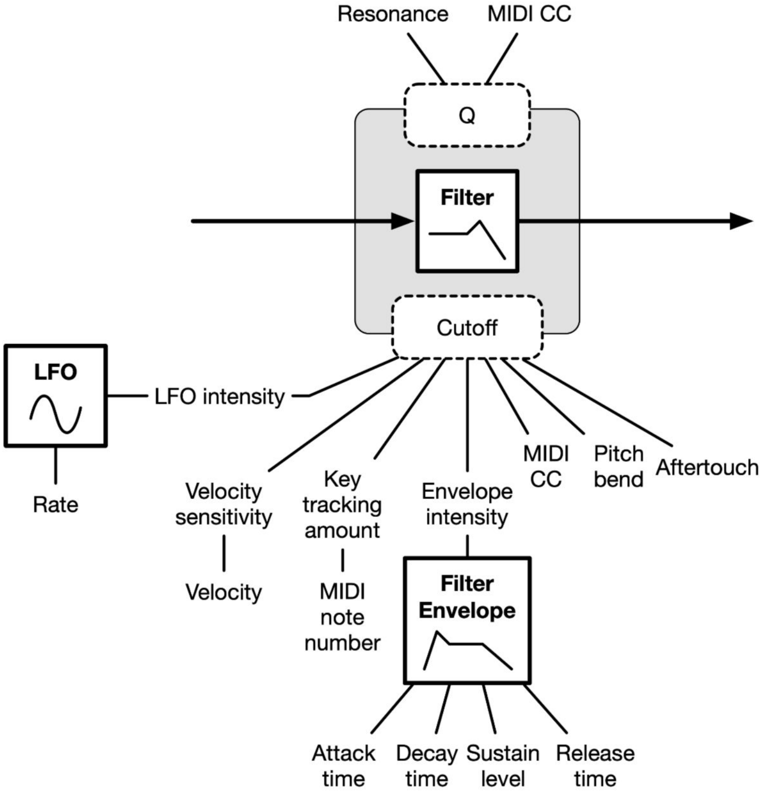
When using the LFO to modulate the filter, this periodically swings the filter's cutoff frequency back and forth around the point determined by the MIDI note number and the velocity. But wait a minute… isn't the filter envelope already changing the cutoff frequency too? That is correct: multiple things at the same time may be affecting the filter's cutoff.

In fact, the filter will be modulated by all of the following:

- the pitch of the note, which determines the initial cutoff frequency
- the key velocity, which also determines the initial cutoff
- the user's setting for the amount of key tracking
- the shape of the filter envelope

- the filter envelope intensity, which determines how “big” the filter envelope is
- the LFO intensity, the amount by which the LFO’s sine wave affects the cutoff
- the current pitch bend amount
- aftertouch
- MIDI Control Change messages (MIDI CC)

Here is the updated schematic for the filter, incorporating all the possible modulation sources:



All the modulators for the filter

Phew, that’s quite a few things. The pitch bend is included because the pitch of the current note determines the cutoff frequency, so bending the pitch up or down should also bend the cutoff frequency.

Aftertouch lets you change the sound by manipulating a key after you pressed it. This is an advanced feature on many analog synths that’s usually not available on inexpensive MIDI keyboards but JX11 supports it in case you have it. MIDI Control Change messages are sent by additional knobs and sliders on your MIDI controller.

Just like the user can determine how much the filter envelope affects the cutoff frequency, they can choose how much effect the LFO has, from hardly modifying the filter's cutoff frequency at all to modulating it over the full range between 20 Hz and 20,000 Hz. This is known as the depth or intensity of the LFO.

With so many ways to influence what the filter is doing, you can imagine this makes it possible to create all kinds of delightful sounds. However, the way these modulations interact is quite complex, so it can be tricky to make this do exactly what you want. An important part of synth design is to give the user a lot of flexibility while keeping the complexity of the instrument manageable.

All these filter modulations are only for the filter's cutoff frequency. Some synths allow you to modulate the amount of resonance or Q, but JX11 doesn't. It does map the resonance amount to a MIDI CC message, so you can use one of the knobs on your MIDI controller to change the resonance while you're playing.

Besides modulating the filter, another typical usage of an LFO is to subtly shift the pitch of the sound up and down. This gives a vibrato effect. JX11 lets you choose the amount of vibrato, which roughly goes from zero to two semitones. Since JX11 uses the same LFO for the filter and vibrato, both these things are always modulated at the same speed. If your MIDI controller has a modulation wheel, this can be used to add even more vibrato.

JX11 can also use the LFO for pulse width modulation or PWM. To do this, it combines the two sawtooth oscillators into a square wave, while using the LFO to rapidly change the position of the second oscillator's waveform relative to the first. This gives a very old school sound.

Other features

So far, all the functionality described has been about sound generation, but there are also choices to be made about how the synth can be played.

Early analog synths were monophonic, meaning that only one note at a time could be played. These days most synths are polyphonic, allowing for two or more notes to be heard at the same time, which allows for playing chords. Still, it can be useful to set the synth to mono mode, especially for playing things like leads or bass lines. JX11 can do both mono and eight voice polyphony. It also supports the sustain pedal.

Another common synthesizer feature is portamento or glide. If you play two notes in succession, the sound smoothly glides from the first pitch to the next in a set amount of time. Glide can be enabled in two different modes: always glide or only glide when the user is playing legato-style. That is, glide if the player presses the second key before releasing the first key. With JX11, glide works in both mono and poly modes. There is an option to add a fixed amount of glide, up to ±36 semitones, to any new note regardless of how it's played.

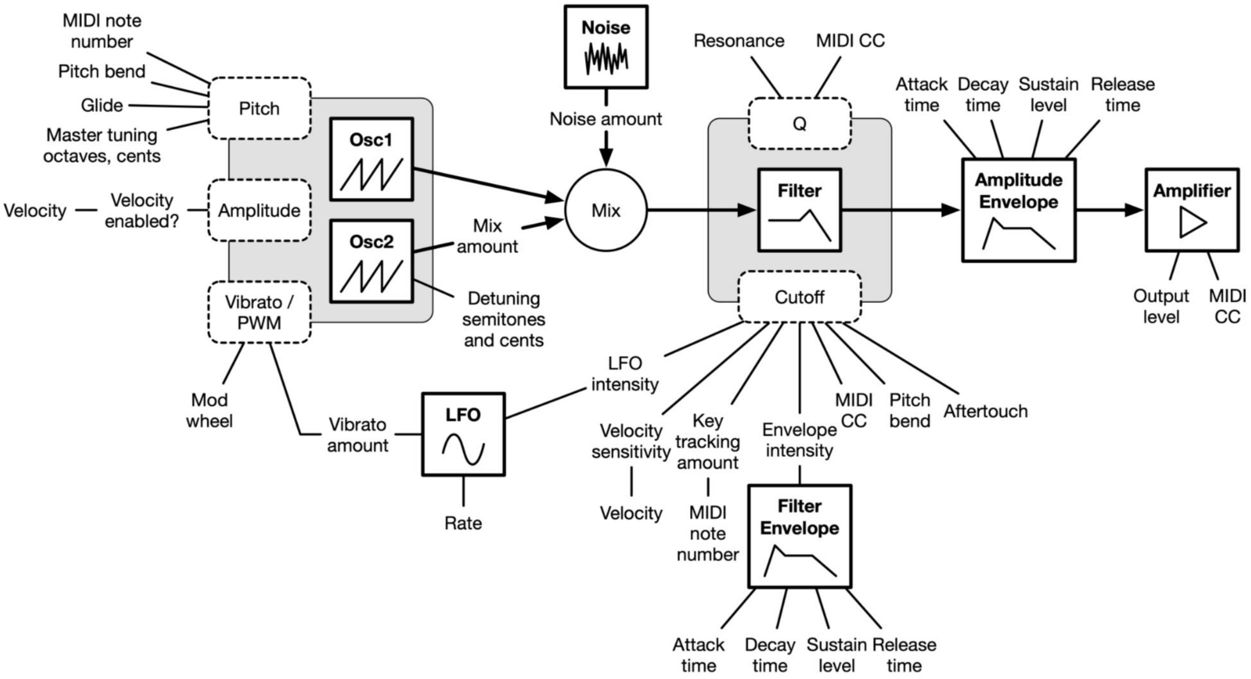
Finally, JX11 has an output level control that lets you adjust the final volume of the sound. The amplitude of the waveform is primarily determined by the envelope shape and the velocity of the note, but the filter can make certain frequencies louder or quieter, and playing more than one note at a time increases the total volume.

Since we're working with floating point numbers in the range –1 to +1, it's possible that all these factors add up to create samples that go outside this range. That's bad and it will create unwanted distortions. The output level setting is used to turn down the volume if it's too loud or boost it if too quiet. There is also a MIDI CC for this.

The finished synth design

In summary: The synth consists of oscillators that make something go up and down at a frequency that's audible to the human ear, followed by a bunch of things to shape that sound over time — envelope, filter, modulation, etc. — to keep the ear captivated.

Here is the completed schematic of JX11 with all the pieces that make up the synth:



The full schematic of the synth

All these features taken together allow the user to make a wide variety of sounds — as you hopefully will have found out for yourself already if you played with the synth's many presets.

JX11's design is fairly typical but even so, this is just one way to make a synth, there are many possible variations. Once you understand how this synth works you should be well prepared to tackle any other type of synthesizer design.

To give you a taste of what else is out there:

- ROMplers are "synths" that merely play back sampled sounds from real instruments. Two other MDA plug-ins, MDA Piano and MDA EPiano, are examples of this. You can find their source code on my GitHub[16]. These kinds of synths can produce quite realistic sounds but they are not very flexible.
- I already briefly mentioned FM synthesis. MDA DX10, also on my GitHub, is an example of such a synth but by far the most famous is the classic Yamaha DX7 that defined the sound of the 1980s. FM is still a popular synthesis technique, often combined with virtual analog oscillators into a "hybrid" synth design.
- The most popular and versatile technique at the moment is wavetable synthesis, found in synths such as Massive and Serum. A wavetable is similar to a sampled sound but much shorter, usually only a single cycle. Instead of making the waveform using a mathematical formula, the oscillator loops through this wavetable at the desired pitch.
- Physical modeling synths use a mathematical model of the inner workings of an instrument to simulate what happens when the instrument is played. Pianoteq is a well-known example, using physical modeling to approximate the physics of all the parts that make up a piano.
- In addition to using an envelope or a filter to shape the sound coming out of the oscillator, other techniques can be used to spice up the sound. For example, waveshaping applies a non-linear function to the audio signal, adding new frequencies that make the sound even richer. In many synths special effects such as delays, chorus, reverb, etc. are added at the end of the signal processing chain to make everything sound even bigger.

What makes a synthesis method unique is really all about timbre. The other properties of sound — pitch, amplitude, duration — are all pretty easy to achieve, but timbre is much more complex. When it comes down to it, all the synthesis methods that have ever been invented are basically different ways to create unique timbres.

[16] https://github.com/hollance/mda-plugins-juce

Chapter 2: Getting started with JUCE

This chapter introduces the JUCE library[17] and what problems it solves for us.

JUCE is a cross-platform framework for writing audio applications as well as VST, AU, and AAX plug-ins. It supports all the major operating systems: Mac, Windows, Linux, iOS, and Android. You can use your favorite IDE and C++ compiler to build JUCE projects.

Technically speaking, the synthesizer from this book doesn't need JUCE at all. However, to make the synth work as a plug-in, it needs to receive MIDI events from somewhere, and its audio output needs to be sent to the computer's speakers somehow. Exactly how to do that is different across the various platforms.

Unless you enjoy suffering and hardship, you don't want to write all this low-level audio code yourself. It's much more convenient to let a framework such as JUCE deal with the nasty details. This allows you to stay focused on the thing you actually want to do: writing the synthesizer logic.

> **Note:** This book is for JUCE version 7. JUCE is actively maintained and it is always possible that a new version breaks older code. The code for this book was written for JUCE 7.0. If you run into compilation errors and you believe this might be due to an incompatibility with the version of JUCE you're using, please visit the book's GitHub page[a] for errata and known issues.
>
> [a] https://github.com/hollance/synth-plugin-book

What are plug-ins?

This book is about writing a synthesizer plug-in. So what exactly is a plug-in?

These days, most music is recorded and edited using a DAW application, or digital audio workstation. Examples are Logic Pro, Ableton Live, REAPER, Cubase, FL Studio, GarageBand, Bitwig Studio, Pro Tools, and many others. DAWs come with tons of functionality already built in, but they also allow third-party developers such as yourself to add new functionality through plug-ins.

[17] https://juce.com

A plug-in is a separate piece of software that is loaded into the DAW. Usually it is placed on a track in the user's project, so that it gets inserted into the audio processing for that track. The nice thing for plug-in developers is that plug-ins can generally be used by any DAW, so it doesn't matter what audio software the user prefers.

There are a few types of plug-ins but the two most common are:

1. the effect or FX plug-in
2. the instrument plug-in — what this book calls a synth

The main difference is that an FX plug-in processes existing sound, while a synth creates completely new sounds. It's also possible to make plug-ins that do both.

From the point-of-view of the plug-in, the DAW is the host application that provides access to the computer's audio inputs and outputs. An FX plug-in receives audio from the track, modifies it in some fashion, and sends the result back to the DAW. For a synth plug-in, the DAW provides it with MIDI events that let the synth know which notes to play.

For plug-ins to be possible, there needs to be some kind of communication protocol between the DAW host program and the plug-in, so that the DAW can tell the plug-in what to do. Conversely, the plug-in must be able to notify the host of any changes, for example when the user interacts with the plug-in's user interface.

Here's the rub: There are multiple of these communication protocols and they are all incompatible.

- **VST2** is the original standard that created a thriving ecosystem of plug-ins, but it has now been deprecated and most developers won't be able to make new VST2 plug-ins anymore.
- **VST3** is the replacement for VST2. Like its predecessor, it is cross-platform and can be used on Mac, Windows, and Linux. Not everyone is a fan of VST3's design — the SDK is universally despised.
- **Audio Units** (AU) are Apple's plug-in standard. Like VST, there are several versions. AUv3 has found a home on iOS devices, but on the Mac, developers are mostly sticking with the older AUv2.
- **AAX** and RTAS are the plug-in formats for Avid's Pro Tools software. These types of plug-ins are used only by Pro Tools, but this product is still the industry standard for professional audio work and therefore worth supporting.

- **LV2** is a plug-in standard that is especially popular on Linux. This is the successor to the older LADSPA format.
- There are a number of other plug-in formats but they are less common. A group of developers from the audio community have been working on a new open source format, CLAP, which looks to be quite exciting but it was not fully supported by JUCE at the time of writing this book.

As a maker of audio plug-ins, of course you want as many users as possible to use your products. That means you'll need to support more than one plug-in format. VST3 is the most universal format as it works on all platforms, but not all DAWs support VST3. A good example is Logic Pro, which only accepts Audio Units.

Most commercially available plug-ins come as VST3, AU, and AAX versions, for both Windows and Mac. Not everyone supports Linux but that's slowly improving. Some plug-ins are also available on iOS or Android.

Each of these plug-in formats has their own SDK. It's certainly possible to make separate versions of your plug-in for each SDK, as well as for each operating system — but that's a lot of work!

This is why many audio developers create wrappers around these different SDKs, and build their plug-ins against the wrapper. JUCE is one of those wrapper libraries: you write your plug-in for the JUCE API only, and JUCE will handle VST3, AU, AAX, LV2, or any other new plug-in format in the future. It seems reasonable to assume that JUCE will eventually support CLAP too.

Why this book uses JUCE

This book is first and foremost about how to make sound out of nothing. But once you have that sound, it does need to go somewhere so that you can actually hear it.

As a book author, there are different ways to deal with this. The easiest would be to avoid all this complexity and simply have the synth write to a WAV file — but that doesn't make it plug-in. Most people who are interested in writing synths want to create plug-ins.

I could make a new version of the synth for each plug-in API, but that is a massive undertaking. It would be smarter to make my own wrapper and write the synth against that, but that wouldn't teach you anything about the libraries people in the industry use.

The thing that made most sense for this book, was to go with the best wrapper library that's currently out there. JUCE is an industry standard and many commercial plug-ins are made with it. JUCE also has very good support from its development team. Many people in the audio community have experience with JUCE, so it's possible to get help if you get stuck. Plus, JUCE makes it easy to create cross-platform user interfaces. All of these things together make JUCE the best place to get started.

By the way, JUCE isn't just for plug-ins. You can build full-fledged desktop and mobile applications with it, including complete host applications. The Tracktion DAW engine[18], for example, is written in JUCE.

Having said all that, in this book you won't be using all of JUCE's built-in functionality. You will separate the synthesizer logic from JUCE as much as possible, so that the synth can work with other plug-in frameworks too. Separation of responsibilities is a good software design principle in any case.

Here's the part you may not like: JUCE is not always free. You can use JUCE in a few different ways:

- License your plug-in under the terms of GPLv3[19]. This means making your plug-in open source. If you're a fan of open source and think the GPL is an acceptable license, then this is a great option. The awesome Surge synthesizer[20], for example, uses JUCE and is licensed as GPLv3.
- For educational or personal use, JUCE is free but the plug-in is required to show a "Made with JUCE" splash screen when it starts up.
- For commercial products you have to pay a monthly license fee, although there is an option for a one-time fee. See the available plans[21] on the JUCE website.

Not everyone who wants to make open source projects is a fan of the GPL. If that's you, then you can choose to give your plug-ins the splash screen, or use another framework such as iPlug2[22] which has a less restrictive open source license.

Besides the JUCE license there are other legalities to take care of. Steinberg, who owns the VST3 SDK and trademark, requires that developers sign a licensing agreement to distribute

[18] https://www.tracktion.com/develop/tracktion-engine
[19] https://www.gnu.org/licenses/gpl-3.0.html
[20] https://surge-synthesizer.github.io
[21] https://juce.com/get-juce
[22] https://iplug2.github.io

commercial VST3 plug-ins. The VST3 SDK is included in JUCE so you don't need to download it separately, but this does not automatically give you the right to distribute VST3 plug-ins. Once you decide to make some money from your plug-ins, you'll need to sign the agreement with Steinberg.

No such agreement is needed for Audio Units. It is already covered by the agreement you sign with Apple to be part of their developer program. However, there is a separate licensing agreement for the use of Audio Unit logos on your website.

JUCE can also build AAX plug-ins but only after you have installed the Pro Tools SDK. We won't be dealing with AAX in this book, as this requires entering into a legal agreement with Avid before you can even download the Pro Tools SDK. Personally, I'd skip this step until you have a product that people are willing to pay for.

> **Note:** Speaking of licenses, the code for this book is open source under the terms of the MIT license. That is also the original license for MDA JX10. How can I release JUCE code under the MIT license instead of GPLv3? I'm only publishing the synth's source code, not the actual plug-in. To use this synth, you will need to compile the plug-in yourself. You can use the code from this book in any of your projects, even commercial ones, but as soon as you decide to distribute this synth as an actual VST or AU plug-in you'll need to abide by the licensing terms for JUCE.

Installing JUCE

Let's set up the project files for our synthesizer. The first step is to get JUCE. Go to juce.com/get-juce/download[23] and choose the download option for your operating system.

The download is a ZIP file containing the JUCE source code, the Projucer tool, and a number of example projects. After unzipping, you can put the `JUCE` folder anywhere you want on your computer. A good place is in your home folder:

- Windows: `C:\JUCE` or `C:\Users\yourname\Documents\JUCE`
- Mac: `~/JUCE` or `/Users/yourname/JUCE`
- Linux: `~/JUCE` or `/home/yourname/JUCE`

[23] https://juce.com/get-juce/download

Alternatively, advanced users may want to clone the JUCE source code[24] from GitHub. This contains the same things as the ZIP file, except that you still need to compile the Projucer tool yourself. See the README[25] for instructions. The advantage of cloning the repo is that this gives access to bug fixes and improvements that have not been integrated into the latest release yet. For this book, however, I suggest sticking with the official download.

If you're an experienced developer, you might be tempted open up your favorite IDE, create a new project, and add the JUCE library and headers to the project. While it is possible to do things this way, JUCE prefers a slightly different workflow where Projucer is in charge of the project, not the IDE.

Introducing Projucer

What exactly is Projucer and why do you need it?

Let's say you develop your plug-in on a Mac using Xcode. In theory, since JUCE is cross-platform, the exact same code should run without problems on Windows. But Windows does not have Xcode… You'd need to make a new project in Visual Studio and import all your source files. Any time you make a change to one of these projects, you'd also have to remember to update the other one. What a headache! And it gets worse when you add Linux or Android into the mix.

The idea behind JUCE is that it should be easy to develop your plug-in on one platform, and then just as easy to make it run on another platform. The only way to do this is to take the IDE out of the loop. That's where Projucer comes in: You exclusively use Projucer to create and manage your JUCE projects.

Of course, you still need to use an IDE to compile the code. Projucer has a **Save and Open in IDE** button that exports an Xcode or Visual Studio project. Also supported are Linux Makefiles, Android Studio, and Code::Blocks. To save space, I'll only discuss Xcode and Visual Studio in this book, as those are the two most common choices.

Once you have a Projucer file for your project, making a Mac version is as easy as using the Xcode exporter. To get a Windows version of the same project, simply use the Visual Studio exporter. Projucer doesn't support cross-compiling, so you need access to both a Mac and a Windows computer — or a virtual machine — to actually build the different versions, but at least you don't have to worry about maintaining separate projects for multiple IDEs.

You edit and compile the code inside your IDE as usual, but Projucer is where you'll manage the project. Adding new source files, for example, must be done in Projucer and not in the IDE. Likewise, you should set any compiler options in Projucer instead of the IDE.

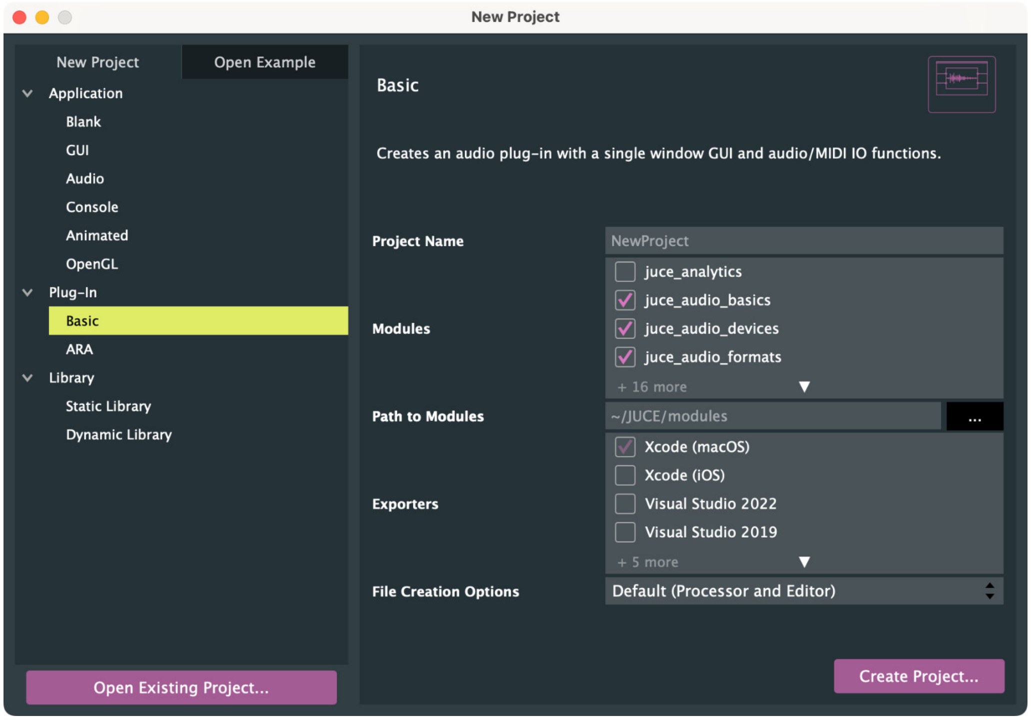
[24] https://github.com/juce-framework/JUCE

[25] https://github.com/juce-framework/JUCE/blob/master/README.md

Any changes that you make to the project structure or the settings inside the IDE will be overwritten by Projucer, so it's important to treat the Xcode and Visual Studio projects as temporary. This may be a different way of working than you're used to, but it's quite convenient for doing cross-platform stuff.

> **Note:** Projucer is a handy tool to quickly get started but it can become cumbersome for larger projects. That's why JUCE also supports CMake. In this book we're sticking with Projucer but it's worth exploring CMake once you start running into Projucer's limitations.

Go to the folder where you put JUCE and launch **Projucer**. It looks something like this:



Basic

Creates an audio plug-in with a single window GUI and audio/MIDI IO functions.

Project Name: NewProject

Modules:
- □ juce_analytics
- ☑ juce_audio_basics
- ☑ juce_audio_devices
- ☑ juce_audio_formats
- + 16 more ▼

Path to Modules: ~/JUCE/modules ...

Exporters:
- ☑ Xcode (macOS)
- □ Xcode (iOS)
- □ Visual Studio 2022
- □ Visual Studio 2019
- + 5 more ▼

File Creation Options: Default (Processor and Editor)

The home screen of Projucer

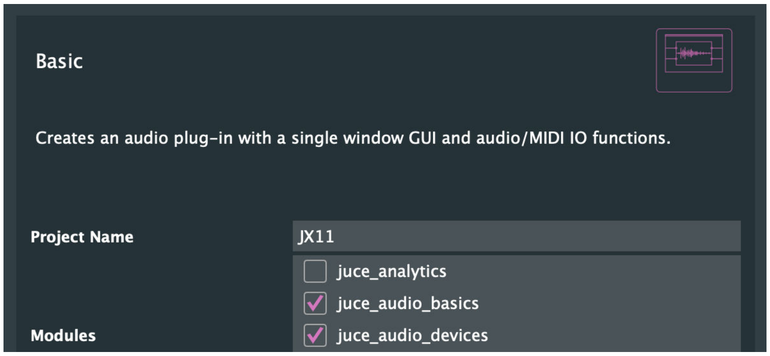
Before continuing, make sure that Projucer knows where to find the JUCE files. On Mac, go to **Projucer** > **Global Paths…** in the menu bar. On Windows or Linux, choose **File** > **Global Paths…**

In the new window that pops up, make sure the **Path to JUCE** and **JUCE Modules** fields are pointing to the correct locations. On my computer I put JUCE in my home directory and so the paths start with `~/JUCE`. If you unzipped JUCE somewhere else, fill in the correct paths here or you will get compilation errors later.

Creating the JUCE project

Now it's time to create the JUCE project for the synth. In the main Projucer window, make sure the **New Project** tab is selected and choose **Basic** under **Plug-In**.

On the right, for **Project Name** fill in **JX11**.



Basic

Creates an audio plug-in with a single window GUI and audio/MIDI IO functions.

Project Name: JX11

Modules:
□ juce_analytics
☑ juce_audio_basics
☑ juce_audio_devices

Creating a new plug-in

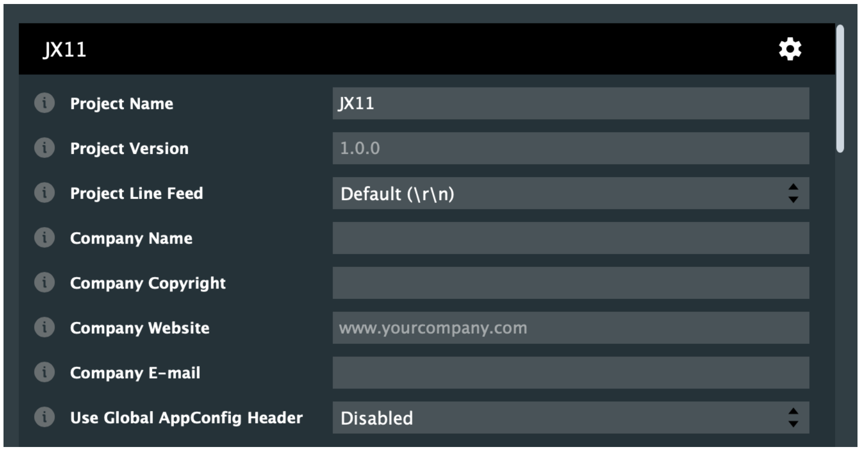
Make sure your IDE is selected in the **Exporters** section. For me this has "Xcode (macOS)" selected since I'm using a Mac.

Click the purple **Create Project…** at the bottom and choose where to save the project. This will automatically create a new folder.

The Projucer screen now changes to show the newly created project. In the sidebar on the left are three sections, click on their headers to reveal their contents.

- **File Explorer**. This shows the source files for your project. Right now, it contains .cpp and .h files for PluginProcessor and PluginEditor. Over the course of this book, you'll be adding new files to this list.
- **Modules**. This lists the included JUCE modules. To keep things flexible, the JUCE codebase is split up into different modules that each do their own specific thing. For example, `juce_audio_plugin_client` contains the wrappers for the different plug-in formats such as VST and AU, `juce_graphics` contains functionality for drawing user interfaces, and so on. You can also create your own modules with reusable code. If the project needs additional modules, you can add them here.
- **Exporters.** This section lists the available exporters for your project. If your IDE isn't listed here, click the round + button to add it. For cross-platform projects you can add multiple exporters.

The gear icon at the top of the window opens the project settings pane:



JX11

Project Name	JX11
Project Version	1.0.0
Project Line Feed	Default (\r\n)
Company Name	
Company Copyright	
Company Website	www.yourcompany.com
Company E-mail	
Use Global AppConfig Header	Disabled

The project settings in Projucer

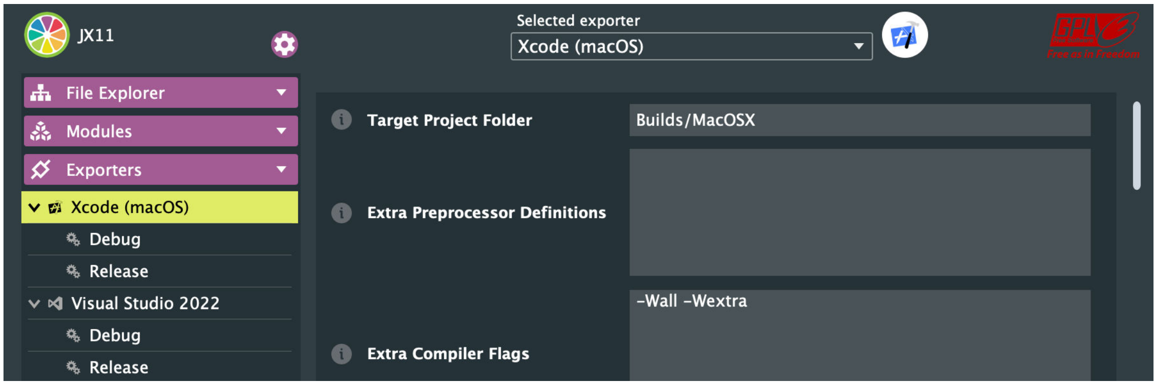
Here you can fill in the details of what your plug-in does. You'll keep the defaults for most of these fields, except for the following:

- **Plugin Formats**. This should have at least **VST3** and **Standalone** selected. You may also choose **AU** if you're a Logic Pro user like me.
- **Plugin Characteristics**. Select the first two options: **Plugin is a Synth** and **Plugin MIDI Input**.
- **Plugin Manufacturer**. This is where you put your company name. Also fill in a four-letter code in the **Plugin Manufacturer Code field**, starting with an uppercase letter. For this book I'm leaving these at the default `yourcompany` and `Manu` but feel free to change them. The plug-in will show up in your DAW under this name, as most DAWs will group plug-ins by manufacturer.
- **Plugin Code**. Projucer has filled this in with a random four-character code. Change this to `JX11`. This code uniquely identifies the plug-in.
- **C++ Language Standard**. Set this to **C++17**.

By default, JUCE projects will have the "Made with JUCE" splash screen upon startup. You can hide this by setting the option **Display the JUCE Splash Screen** in the project settings to **Disabled**.

There may now be a popup at the bottom of the Projucer window saying, "Incompatible License and Splash Screen Setting. Save and export is disabled." To get rid of this, click on **Sign In** or use the face icon in the top-right corner of the window. If you have a paid JUCE license, use this panel to sign in with your JUCE account. Otherwise, select **Enable GPL Mode** to indicate you're going to make the plug-in available under the GPLv3 license.

Next, you need to configure the exporters. As I mentioned before, you should not use the IDE to change the settings of your project. Compiler options such as preprocessor definitions or linker flags need to be set in Projucer in the Exporters section.



Selected exporter: Xcode (macOS)

Target Project Folder: Builds/MacOSX

Extra Preprocessor Definitions:

Extra Compiler Flags: -Wall -Wextra

Configuring the exporter for your IDE

Go to **Exporters** in the sidebar and select your IDE from the list. Make sure to select the top-level item, not the Debug or Release options below it. This brings up the relevant compiler settings.

If you're using Xcode or a Linux Makefile, change the following option:

- **Extra Compiler Flags**. Set this to `-Wall -Wextra`

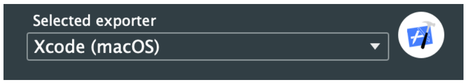
If you're using Visual Studio, change the following option:

- **Extra Compiler Flags**. Set this to `/W4`

This will enable additional warnings from the compiler. It's not strictly necessary to enable these warnings, but it's good practice. Plus, I needed an excuse to show you where to change these kinds of options. Note that you can have different settings for the Debug and Release builds of your app.

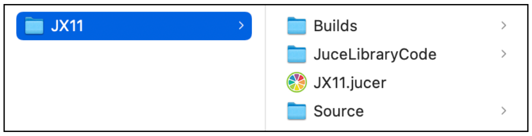
For the Visual Studio exporter, go into the **Debug** section and locate the **Enable Plugin Copy Step**. This is disabled by default, but it's smart to enable it, so that Visual Studio will automatically copy the plug-in into the correct location after compiling. On Windows this is `C:\Program Files\Common Files\VST3`. (This step isn't needed for Xcode.)

When you're done setting up the project, go to the top of the Projucer window where it says **Selected exporter**. Make sure the correct IDE is selected and then click the big round button. Now Projucer will generate a new Xcode or Visual Studio project and opens it in the IDE.



The export button

If you navigate to the folder where the project was saved it will look like this:



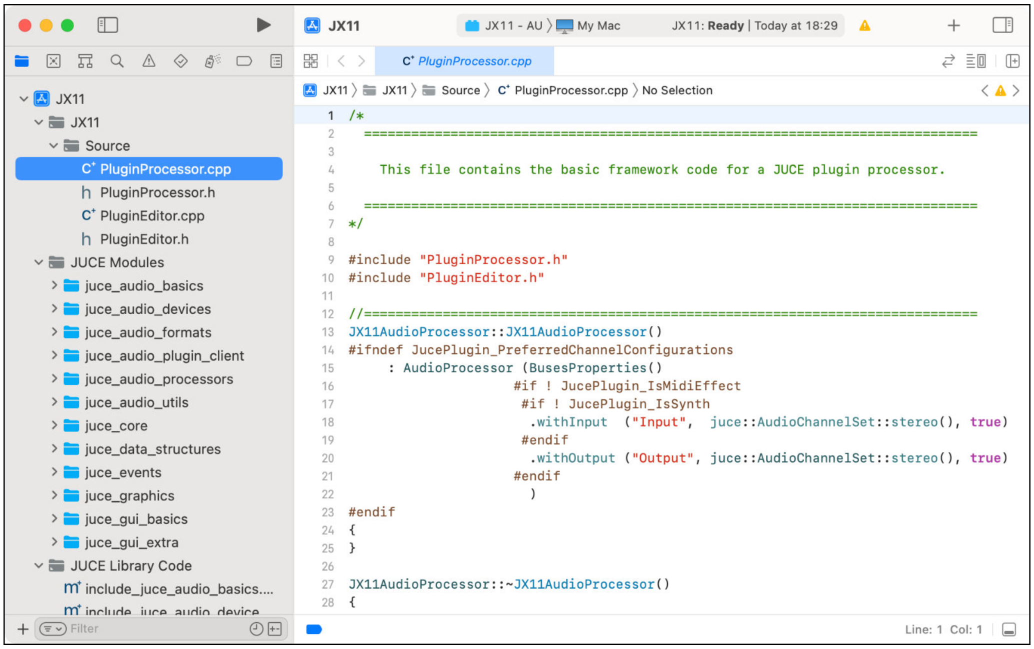
The generated files on macOS

The contents of this folder are:

- **JX11.jucer**. This is an XML file containing the Projucer project. If your computer does not show file extensions, note that jucer files have the same colorful icon as Projucer.
- **Source**. This has the source code for the plug-in. When you add new files to the project in Projucer, they go into this folder.
- **Builds**. This is where Projucer has saved your Xcode or Visual Studio project. These files are auto-generated by Projucer and therefore you should not edit these yourself. This is also where the compiled binaries go when you build the plug-in.
- **JuceLibraryCode**. This contains the JUCE modules that will be added to your project, plus the **JuceHeader.h** file that is the main include for JUCE. As with the stuff in the Builds folder, this is all auto-generated and will be overwritten by Projucer.

Let's have a look at the project in the IDE. Xcode's project navigator shows several groups:

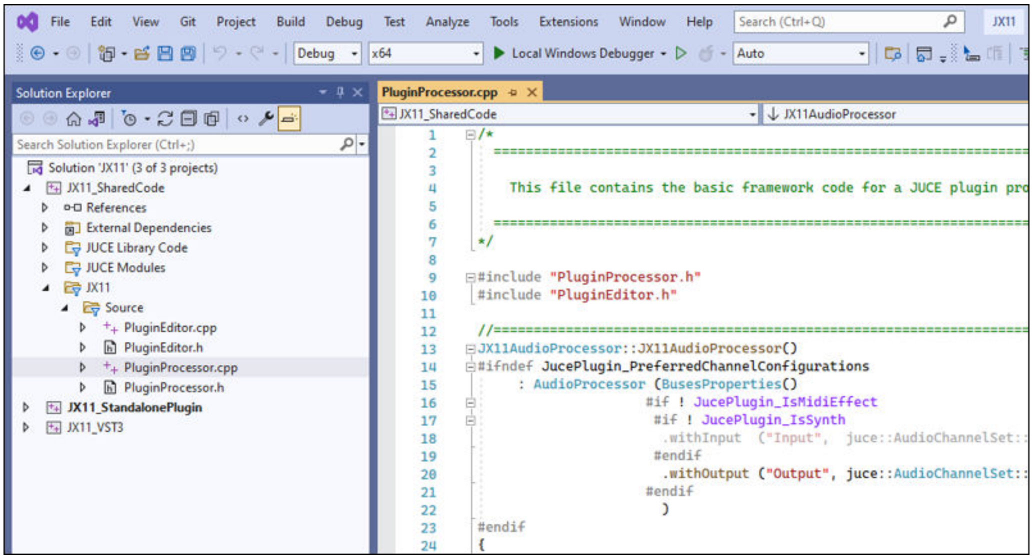
- **JX11/Source**. This has the plug-in's source code files.
- **JUCE Modules**. These are references to the JUCE source code. The folders are blue, which means this code isn't inside your project folder but is an absolute path to the JUCE installation folder.
- **JUCE Library Code**. The contents of the `JuceLibraryCode` folder described above. These files pull the relevant JUCE framework sources into the project.



```
/*
  ==============================================================================

    This file contains the basic framework code for a JUCE plugin processor.

  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
JX11AudioProcessor::JX11AudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
}

JX11AudioProcessor::~JX11AudioProcessor()
{
```

The new project in Xcode

If you're using Visual Studio, the Solution Explorer will show a similar structure. The source code is found under the **JX11_SharedCode** target.



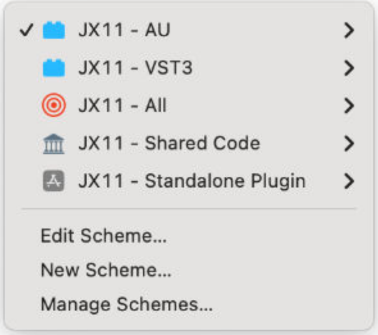
```
/*
  ==============================================================================

    This file contains the basic framework code for a JUCE plugin pro

  ==============================================================================
*/

#include "PluginProcessor.h"
#include "PluginEditor.h"

//==============================================================================
JX11AudioProcessor::JX11AudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::
                     #endif
                       )
#endif
{
```

The new project in Visual Studio 2022

You can ignore the JUCE Modules and JUCE Library Code groups, I always collapse them. The important parts are all in **JX11/Source**.

Compiling the plug-in

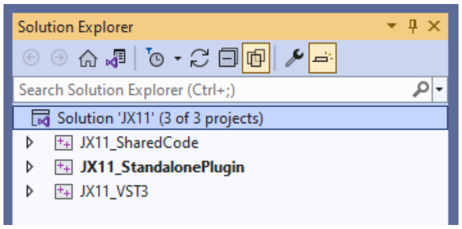
Once the project is loaded into your IDE, you should be able to see that it has several different targets. In Xcode, it shows:



The project targets in Xcode

There is a target for each supported plug-in format: **AU** and **VST3**. There is also a **Standalone Plugin** target that lets you run the plug-in without needing a separate host application. The **Shared Code** target builds the code that all plug-ins have in common. The **All** target builds all of the above.

In Visual Studio, the targets are:


Solution Explorer
Search Solution Explorer (Ctrl+;)
Solution 'JX11' (3 of 3 projects)
JX11_SharedCode
JX11_StandalonePlugin
JX11_VST3

The project targets in Visual Studio

The solution is configured to build a VST3 plug-in using the **JX11_VST3** target (AU is not available for Windows). The other two targets are **JX11_SharedCode**, which contains the actual source code for the synth, and **JX11_StandalonePlugin**.

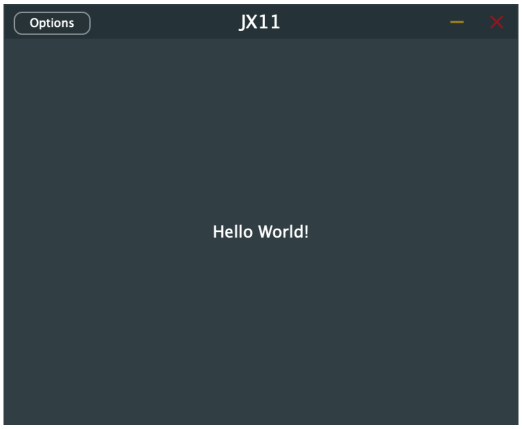
Let's build the Standalone Plugin first, as that's the easiest one. This embeds the plug-in in a very basic host application, so that you can run it on its own. Many synths ship with such a standalone version, enabling users to play the synth outside of a DAW.

In Xcode, select **JX11 - Standalone Plugin** from the scheme picker. In Visual Studio, make sure **JX11_StandalonePlugin** is set as the "startup project" (the bold one) in the Solution

Explorer. Then build and run the app.

You may get a few warning messages during the build process, but you can ignore these for now. This happens because you added the extra warning flags in Projucer earlier.

After compilation successfully completes, you should have a small window on the screen looking like this:



The standalone app

This app doesn't do anything useful yet but at least it shows that everything works correctly. The "Hello World!" message is the default user interface for plug-ins created by Projucer. The **Options** button at the top can be used to configure the audio and MIDI inputs and outputs.

> **Note:** If you get compilation errors at this point, make sure the Global Paths option in Projucer points at the folder where you installed JUCE.

Let's compile an actual plug-in now. Go back to the IDE to activate the VST3 target. In Xcode, select **JX11 - VST3** from the scheme picker. In Visual Studio, right-click on **JX11_VST3** in the Solution Explorer and choose **Set as Startup Project**.

Building this target creates the VST3 plug-in and installs it in the correct location. If you're on a Mac, feel free to also build the AU target. Personally, I stick with the VST3 version until the plug-in is ready for testing.

After building the plug-in, nothing appears to happen. In fact, clicking the Run button in Visual Studio will give an error message that it is unable to start the program. That's because a plug-in is just a binary file containing a shared library or DLL, it's not a runnable application. The build process compiled this shared library and copied it into the place that DAWs will look for plug-ins.

On macOS, the **JX11.vst3** file is placed in the folder: `~/Library/Audio/Plug-Ins/VST3`.

On Windows, the **JX11.vst3** file is placed in `C:\Program Files\Common Files\VST3`. However, Visual Studio only copies the file into that folder if you've enabled this in the Visual Studio exporter in Projucer (see above). If that option is disabled, you'll need to copy the file by hand.

It's possible the VST3 file fails to be copied on Windows and the build log shows an error message. This happens when your user account does not have sufficient rights to modify the system VST3 folder. To enable this, you need to change the security properties on `C:\Program Files\Common Files\VST3` to give full control to the "Users" group.

Now that you have a plug-in, it's time to load it into some kind of host application. Open your favorite DAW and see if you can find the plug-in and put it on a track. It should be under **yourcompany** > **JX11**. Logic Pro users, remember that Logic does not load VST3 files so you'll need to build the AU version too. Does it work? Congrats, your new plug-in is now successfully running in a host application!

Testing with AudioPluginHost

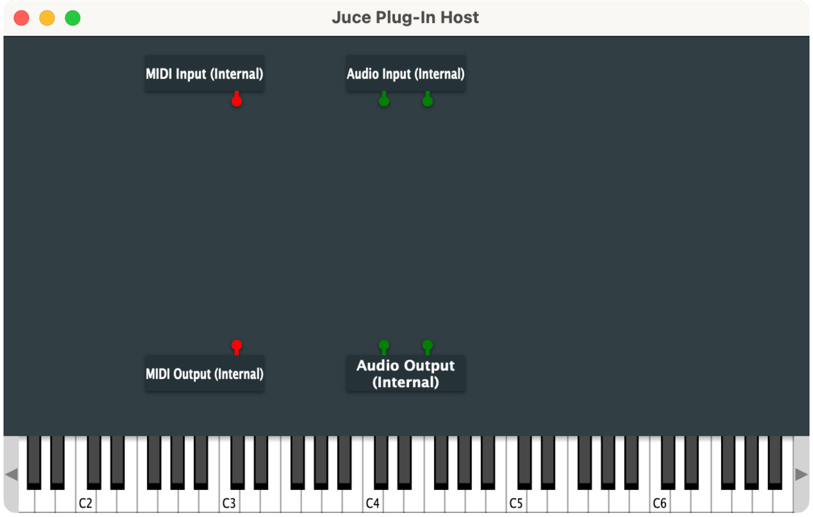
During development you can certainly test the plug-in in your favorite DAW, but it can be slow going as these DAWs are big programs that can take a bit of time to load. After you make a change to the plug-in, usually the DAW will need to be restarted to pick up the new version.

Fortunately, JUCE comes with a basic host application called, obviously enough, **AudioPluginHost**. One small problem: you need to compile this program yourself. Since this is a JUCE app, it comes with a jucer file.

Go to your JUCE installation folder, then navigate to **extras/AudioPluginHost** and double-click **AudioPluginHost.jucer** to open it in Projucer. Click the round export button at the top to open the project in your IDE and then build and run it.

In the AudioPluginHost window, you should see several blocks for MIDI input and output and also Audio input and output. These blocks may look a little different on your computer, depending on what kind of audio hardware you have. At the bottom of the window is a virtual piano keyboard.

Great, the AudioPluginHost is running. Now let's add the plug-in to it!

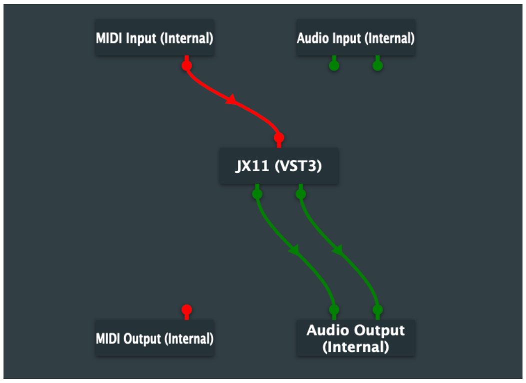


The AudioPluginHost window

From the AudioPluginHost's **Options** menu, choose **Edit the List of Available Plugins**. This brings up a new window with all the plug-ins that AudioPluginHost knows about. Initially this list is empty, since AudioPluginHost hasn't scanned the plug-ins on your computer yet.

Click the **Options…** button at the bottom and choose **Scan for new or updated VST3 plug-ins**. Then press the **Scan** button. Once it's done you should have an entry for JX11 in the list. You can now close this window.

In AudioPluginHost's main window, right-click and from the popup choose **yourcompany > JX11**. This adds a new block for your plug-in. Connect the red input to **MIDI Input** and the two green outputs to **Audio Output**, or the equivalent blocks on your system.



Connecting the plug-in to MIDI input and audio output

Click on the piano keys at the bottom of the window and AudioPluginHost will send MIDI events to the plug-in that tell it what notes to play. It also sends the audio coming out of the plug-in to the computer's speakers. Of course, the plug-in doesn't do anything yet so there is no actual sound being produced.

If you have a MIDI keyboard, double-click on the **MIDI Input** block. This brings up the Audio Settings panel. Make sure your MIDI controller is selected under **Active MIDI Inputs**.

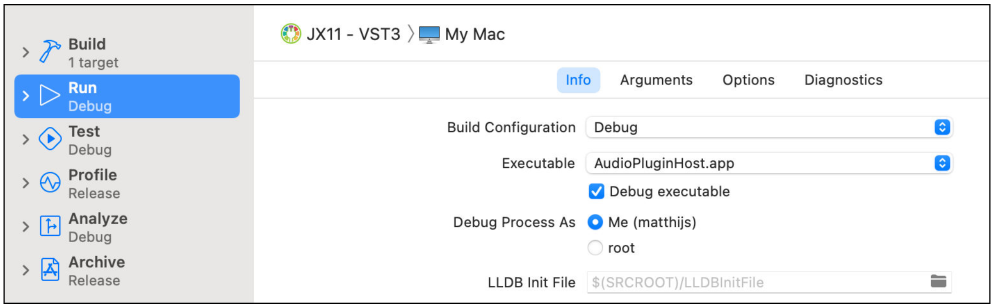
Double-click the **JX11 (VST3)** block to bring up the UI for the plug-in. This should show the same “Hello World!” message as before.

> **Tip:** AudioPluginHost comes with a simple synthesizer, **Sine Wave Synth (Internal)**. To verify that MIDI input and audio output are indeed working, add this synth to the graph and hook it up. If you have any other synth plug-ins installed on your computer, you can also add these to AudioPluginHost.

From the menu choose **File** > **Save**. This puts the graph you’ve just created into a file so that you can use it again the next time you run AudioPluginHost. I suggest naming this **JX11.filtergraph** and saving it in the JX11 project folder.

For convenience, you can set up the IDE so that it launches AudioPluginHost every time you build the plug-in.

In Xcode, select the **JX11 - VST3** target and then choose **Edit Scheme…** In the **Run** section, click on **Executable** and choose **Other…** from the popup. Now navigate to the folder where AudioPluginHost was built and select **AudioPluginHost.app**. This binary is located in `JUCE/extras/AudioPluginHost/Builds/MacOSX/build/Debug`. (Tip: Copy this app somewhere easier to find such as the top-level JUCE folder.)



Build Configuration: Debug

Executable: AudioPluginHost.app

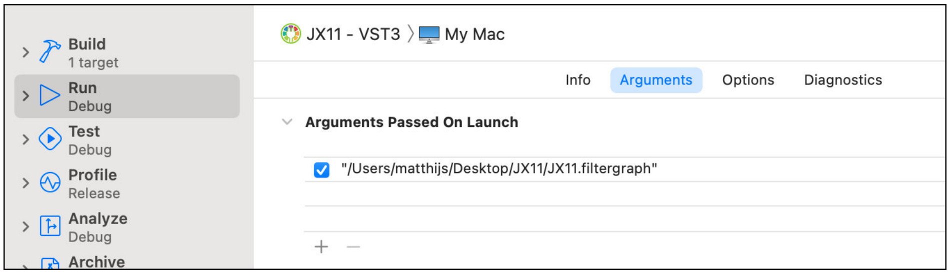
☑ Debug executable

Debug Process As: ● Me (matthijs) ○ root

LLDB Init File: $(SRCROOT)/LLDBInitFile

Setting up Xcode to automatically run AudioPluginHost

AudioPluginHost will automatically load the most recent graph when it starts up, but you can also tell it to load a specific graph. Still in the Xcode scheme editor, in the **Arguments** tab, under **Arguments Passed On Launch**, add the path to the **JX11.filtergraph** file. If there are spaces in this path, put the whole thing in double quotes.

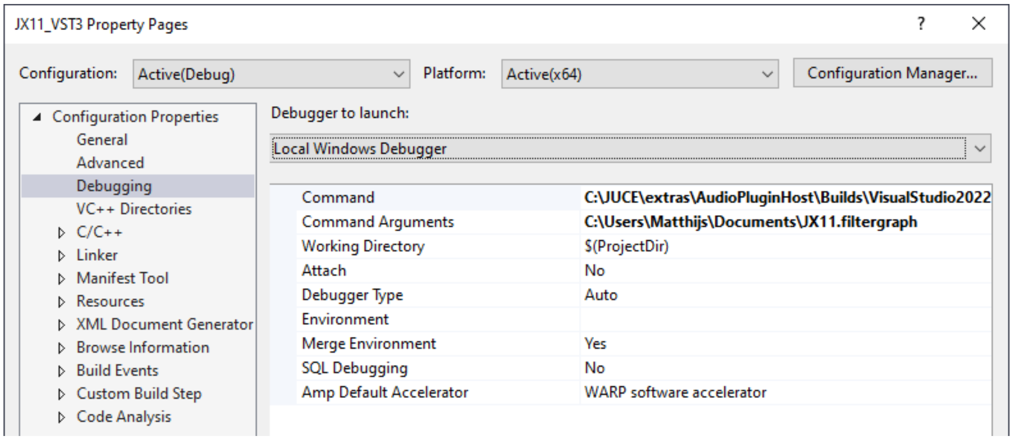


Arguments Passed On Launch

☑ "/Users/matthijs/Desktop/JX11/JX11.filtergraph"

Telling AudioPluginHost which graph to load

In Visual Studio, right-click **JX11_VST3** in the Solution Explorer and choose **Properties**. In the window that pops up, go to **Debugging**. Change **Command** to where **AudioPluginHost.exe** lives. This is in the JUCE folder and then `extras\AudioPluginHost\Builds\VisualStudio2022\x64\Debug\App\AudioPluginHost.exe`. Set **Command Arguments** to the path to where you saved the JX11 filtergraph.



Configuration: Active(Debug) Platform: Active(x64)

Debugger to launch: Local Windows Debugger

Command	C:\JUCE\extras\AudioPluginHost\Builds\VisualStudio2022
Command Arguments	C:\Users\Matthijs\Documents\JX11.filtergraph
Working Directory	$(ProjectDir)
Attach	No
Debugger Type	Auto
Environment	
Merge Environment	Yes
SQL Debugging	No
Amp Default Accelerator	WARP software accelerator

Setting up Visual Studio to automatically run AudioPluginHost

Now when you press the run button, it will first compile the VST3 file and copy it into the system folder for VSTs. Then it launches AudioPluginHost, which loads your plug-in and hooks it up to MIDI and the audio output.

Note: You may sometimes find that, after making certain changes in Projucer, that the settings for launching AudioPluginHost have mysteriously vanished from your IDE. Usually Projucer will update the exported project without breaking things but sometimes it completely overwrites the project and you'll have to set everything up again. To minimize this kind of thing happening with Visual Studio in particular, use **File** > **Save Project** from Projucer rather than the export button. Now Visual Studio will ask to reload the project rather than treat it as a completely new project.

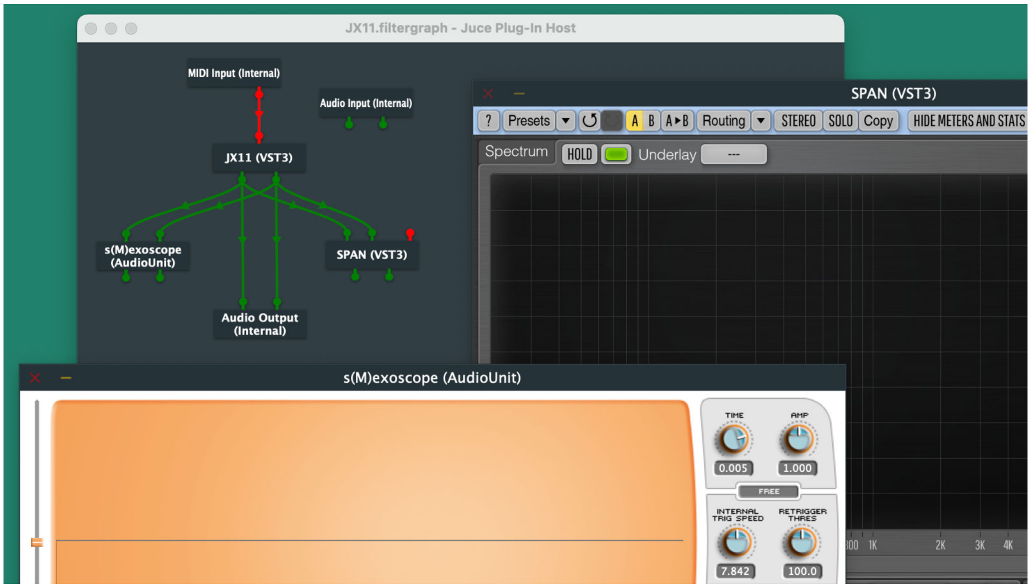
Add some more plug-ins

AudioPluginHost is really nice for testing and debugging your plug-ins. It is much more lightweight than using a full DAW. To make this even more powerful, I suggest that you also download and install an oscilloscope plug-in and a spectrum analyzer plug-in.

The oscilloscope lets you see what the output of the synth looks like in the time domain, while the spectrum analyzer shows it in the frequency domain. Both are indispensable for making sure what the synth does is correct.

I personally use s(M)exoscope[26] as the oscilloscope and Voxengo SPAN[27] as the spectrum analyzer. Another good option is the free bundle from MeldaProduction[28], which contains MAnalyzer and MOscilloscope. All these plug-ins are free and available for Mac and Windows.

Here's what my graph looks like in AudioPluginHost with the oscilloscope and spectrum analyzer plug-ins added:



AudioPluginHost with oscilloscope and spectrum analyzer

Unfortunately these plug-ins do not seem to be available for Linux, but I'm sure you'll be able to find substitutes. Your DAW may have such tools built in. For example, REAPER comes bundled with the very capable JS: Oscilloscope Meter and JS: Frequency Spectrum Analyzer Meter.

[26] http://armandomontanez.com/smexoscope/
[27] https://www.voxengo.com/product/span/
[28] https://www.meldaproduction.com/MFreeFXBundle

Making an iOS version

To make an iOS version of the plug-in, you need a Mac with Xcode. It also requires an Apple Developer Program membership. I'm going to assume you've set this up already and are able to put apps on a test device using Xcode.

Making an iOS build is fairly straightforward, as JUCE already does all the work for you. The one complication is that it won't work if code signing isn't set up properly — if you've developed iOS apps before, this is something you're no doubt familiar with.

VST3 and AU plug-ins are shared libraries or DLLs that are loaded into memory by the host. On iOS this works somewhat differently. The plug-in exists as an app extension that is embedded inside a regular app. When users download a plug-in from the App Store, they are really downloading an app that contains the plug-in as an app extension.

The format of this plug-in is AUv3, which is a slightly more modern version of the Audio Unit format used on the Mac. When the app is installed, the AUv3 is automatically registered with the system and can be used by host apps such as GarageBand. By the way, AUv3 can also be used on macOS and is delivered the same way: embedded inside an app.

In **Projucer**, go to the project settings screen (gear icon at the top). Under **Plugin Formats**, enable both **AUv3** and **Standalone**. When you start clicking in here, it might uncheck what was previously selected, so make sure **VST3** remains enabled (and possibly **AU** if you're building for Logic Pro).

Next, go to the **Exporters** pane and add a new exporter for **Xcode (iOS)**. In the settings for this exporter, scroll all the way down and under **Development Team ID** fill in your Apple Developer Account's 10-letter identifier. You can find this identifier in Keychain Access: select your development certificate, choose Get Info, look under Organizational Unit. This is necessary for code signing to work when installing the app to your device.

If you want, you can also fill in a bundle ID under **Exporter Bundle Identifier**. If you leave this blank, it will be `com.yourcompany.JX11`. The exporter settings also have entries for icons, screen orientation, and other typical iOS app configuration options.

At the top of the Projucer window, under **Selected exporter**, choose **Xcode (iOS)** and press the export button. Now the iOS version of the plug-in will open in the IDE.

The target to build is **JX11 - Standalone Plugin**. Since AUv3 are always delivered inside an app, you need to build the standalone app and install it on your device, just as if someone downloaded the app from the App Store.

Connect your iOS device to the Mac. In the scheme picker, select this device and then build and run the app. If all goes well, the app is now running on your device and showing the plug-in's UI in fullscreen. The app is very barebones, it doesn't even have an icon. For a real synth that you plan to release on the App Store, you need to do a bit of work to make it more appealing. For example, it would be good to show an on-screen piano keyboard here so that users can directly play the synth from within the app.

To use the synth as a plug-in, simply load up a DAW such as GarageBand. In GarageBand's instruments picker, go to the **External** card and tap on **Audio Unit Extensions** at the bottom. In the popup there should be an icon for JX11. Select that and you can play JX11 inside GarageBand, sweet!

> **Note:** If you get the error message "Unable to install JX11" when building the app, it means you didn't set up the code signing options correctly in Projucer.

JUCE plug-in architecture

All right, if you've made it to this point you're in good shape. You've managed to set up JUCE, create the plug-in project, and are able to test it in a host application. Let's finally dig into some code!

The source files that implement the actual plug-in are:

- PluginProcessor.h and .cpp
- PluginEditor.h and .cpp

The **processor** is the main class of the plug-in. The **editor** is the plug-in's user interface.

There is always an instance of the processor, but the editor will only exist when the host application needs to show the UI. In AudioPluginHost that happens when you double-click the JX11 block. The content of that window — the "Hello World!" message — is created by the editor.

Have a look at **PluginEditor.cpp**. There isn't much going on in this file yet, just a `paint` method that draws the "Hello World!" text. For the majority of this book we'll ignore the PluginEditor source code and focus only on the audio processing logic. In chapter 13 you will learn how to implement the UI.

The main source files you'll be concerning yourself with are PluginProcessor.h and .cpp. These files contain the definition of a single class, `JX11AudioProcessor`. Any communication between the DAW and the plug-in happens through this object.

Open **PluginProcessor.h**. It starts out like this:

```
#pragma once

#include <JuceHeader.h>

//==============================================================================
/**
*/
class JX11AudioProcessor  : public juce::AudioProcessor
{
public:
    //==============================================================================
    JX11AudioProcessor();
    ~JX11AudioProcessor() override;

    //==============================================================================
    void prepareToPlay (double sampleRate, int samplesPerBlock) override;
    void releaseResources() override;

    /* many other methods omitted */
```

The include of `<JuceHeader.h>` is necessary to pull in all the JUCE modules. This is the only JUCE file you need to explicitly include.

The `JX11AudioProcessor` class extends `juce::AudioProcessor`. All the JUCE classes and functions live in the `juce` namespace.

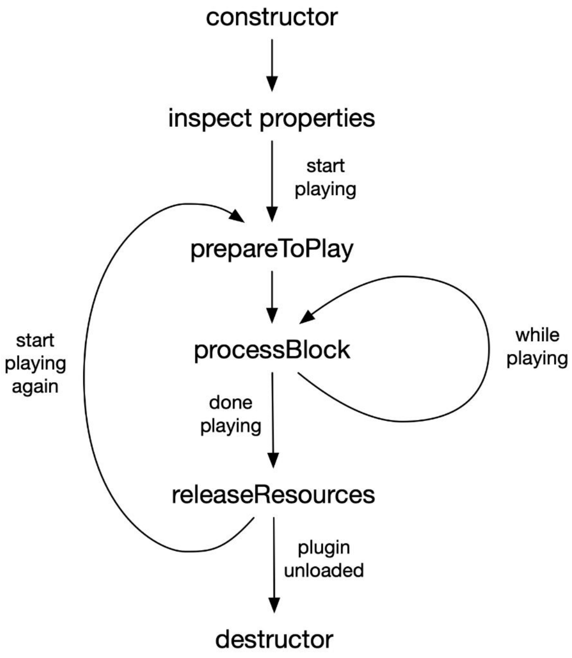
There are several method declarations in `JX11AudioProcessor`. These make up the communications protocol that allows the plug-in to talk to the host application. For example, the host calls the `prepareToPlay` method to let the plug-in know it is about to be activated. Writing a plug-in is essentially a matter of implementing all these methods.

At the bottom of the class definition is the following:

```
private:
    //==============================================================================
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (JX11AudioProcessor)
};
```

You'll see this `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` macro at the bottom of most JUCE classes. It disables the copy constructor for the class so that it cannot be copied, and it enables JUCE's built-in memory leak detector.

The easiest way to understand what all the methods inside `JX11AudioProcessor` are for, is to look at the lifecycle of a plug-in. The plug-in goes through the following stages.



The lifecycle of a plug-in

When the user tells their DAW to load the plug-in, the DAW will construct the plug-in and inspect its properties:

- is this an effect or a synth?
- does it accept MIDI messages? does it output new MIDI messages?
- how many input and output channels does it have?
- does the plug-in have a UI?
- are there any factory presets?
- is the output of the plug-in delayed; in other words, is there any latency?
- does the output of the plug-in continue after the input has stopped?
- and so on…

There are methods in `JX11AudioProcessor` for all these questions: `acceptsMidi`, `hasEditor`, `getNumPrograms`, `getTailLengthSeconds`, and so on.

Once the host understands what the plug-in does and wants to start the audio processing, it calls `prepareToPlay`. This also tells the plug-in about the sample rate and audio buffer size. Here the plug-in would do things such as allocate memory and reset its state.

Allocating resources is normally the kind of thing you'd do in a constructor, but the rules for plug-ins are slightly different. The constructor is called only once, but `prepareToPlay` may be called any number of times in the plug-in's lifecycle — possibly every time the user presses the play button in the DAW.

The counterpart of `prepareToPlay` is `releaseResources`, which is called when the DAW finished playing audio. This is a good place for the plug-in to free up any resources it no longer needs, like what you'd normally do in a destructor.

The most important method is `processBlock`. This is called hundreds or even thousands of times per second by the host application and is where your plug-in will do all its work.

Now let's have a quick peek inside **PluginProcessor.cpp**. The constructor looks like this:

```
JX11AudioProcessor::JX11AudioProcessor()
#ifndef JucePlugin_PreferredChannelConfigurations
     : AudioProcessor (BusesProperties()
                     #if ! JucePlugin_IsMidiEffect
                      #if ! JucePlugin_IsSynth
                       .withInput  ("Input",  juce::AudioChannelSet::stereo(), true)
                      #endif
                       .withOutput ("Output", juce::AudioChannelSet::stereo(), true)
                     #endif
                       )
#endif
{
}
```

Here, but also in other places in the source code, there are `#ifdef` and `#if` preprocessor statements. These conditionals, such as `JucePlugin_IsSynth`, correspond to options from the Projucer. They exist to make the Projucer template code generic and adaptable to different kinds of plug-ins. In case you're curious, you can find these defines in **JUCE Library Code/JucePluginDefines.h**, which is automatically imported by JuceHeader.h.

As you scroll through the source file, you may have noticed the spelling: JUCE uses British English. I'm from Europe but I've seen too many American movies that I prefer US spelling. From time to time you might be wondering why something doesn't code complete in the IDE — chances are you're spelling it wrong. For example, it's `Colour` instead of `Color`, and `initialise` instead of `initialize`.

Also note that JUCE uses its own, somewhat peculiar, coding style. I'm not a fan of that style, so in this book I'm making no effort to comply with it.

In the following chapters you'll be heavily editing the PluginProcessor.cpp source file, and I'll explain in more detail what each of its methods does.

Conclusion

This was a quick introduction to JUCE, the Projucer project management tool, and the AudioPluginHost test environment.

As you're developing in JUCE it's useful to keep the API documentation close at hand. You can find this documentation at docs.juce.com[29].

For example, here is the documentation for `juce::AudioProcessor`[30], the base class of `JX11AudioProcessor`. If you're ever wondering what any of these methods do, look it up in the docs!

JUCE also comes with a set of useful tutorials[31]. In fact, I'd suggest you put this book down now and read at least the tutorials from the Getting Started section.

To learn more about Projucer, you can find the official manual here[32].

If you're stumped by a problem, the JUCE forums[33] are a good place to ask for advice. Chances are someone else ran into this problem too, so be sure to search the forums before asking.

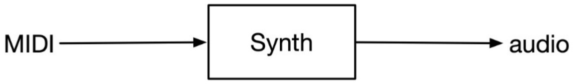
Finally, don't be afraid to look through the JUCE source code[34]. I often do this if I want to understand how something works and the docs aren't telling me everything I need to know.

> **Tip:** JUCE ships with a number of example projects. You can access these through Projucer. Choose **New Project…** from the menu and then switch to the **Open Example** tab to check out the included examples.

[29] https://docs.juce.com
[30] https://docs.juce.com/master/classAudioProcessor.html
[31] https://juce.com/learn/tutorials
[32] https://juce.com/discover/stories/projucer-manual
[33] https://forum.juce.com/
[34] https://github.com/juce-framework/JUCE

Chapter 3: MIDI in, sound out

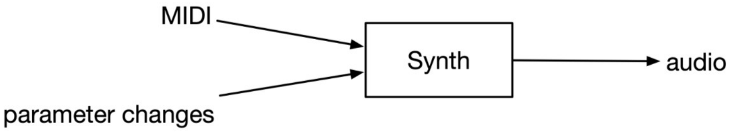
In its most basic form, a synthesizer is a piece of software that converts a stream of MIDI commands into a stream of audio samples.



The MIDI commands are things like “now play this note”, “stop playing that note”, “the sustain pedal is pushed down”, “the pitch bend wheel has been moved”, and so on. These commands can come from a MIDI controller, such as the user playing a physical keyboard, or from the DAW playing back a region with recorded MIDI commands.

The audio output generated by the synth is a stream of typically 44100 or 48000 samples per second. For stereo this number is doubled since there is one sample for the left speaker and one sample for the right speaker. MIDI commands happen much more rarely, typically a few per second at most.

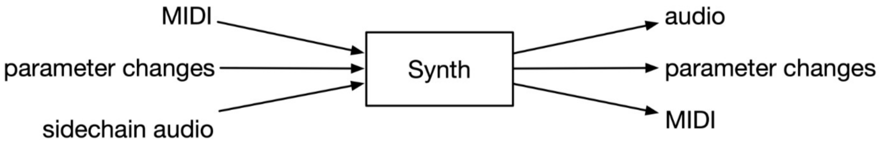
But this is not the whole picture. There is another stream of input data: parameter changes.



The parameters describe the synth’s knobs and other controls. Whenever the user interacts with these knobs, the synth will have to make changes in how it generates the audio. Parameters can also be automated in the user’s DAW. Just like MIDI commands, parameter changes happen relatively infrequently, not more than a few times per second.

Another input that is possible, although less common, is a stream of audio coming from a different track in the DAW. This is also known as a sidechain. The synth can analyze this audio data somehow and use the results of the analysis to create new sounds. The archetypical example is the vocoder, which uses the human voice on the sidechain to turn the audio from the synth into something that sounds like a robot.

The full picture is as follows:



Besides the usual audio output, the synth can send out updates for the parameter values. For example, JX11 will change the master volume parameter in response to certain MIDI events. The DAW has to be told about this so that parameters stay in sync between the host and the plug-in.

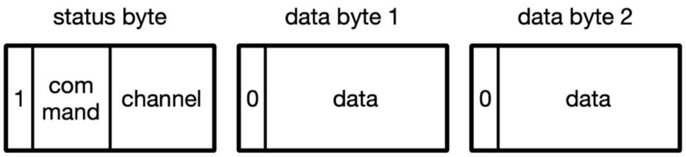
Finally, the synth can output new MIDI commands. MIDI output is not very common in synths and may not even be supported by all DAWs. We will not cover MIDI output or the sidechain in this book.

So what exactly is MIDI?

MIDI, which stands for Musical Instrument Digital Interface, was originally invented in the 1980s as a way for instruments to communicate electronically over a basic serial cable. It quickly became the industry standard for connecting digital music hardware. These days MIDI is also used by DAWs to record note information into MIDI regions, even if you're not using any external gear.

MIDI is not sound, but a description of the sounds that should be played. Think of it as sheet music in electronic or digital form.

The MIDI protocol is quite straightforward. When one MIDI device wants to communicate with another MIDI device, it sends a short message consisting of one, two, or three bytes. For example, the message `0x90 0x45 0x73` means: play the note 45 (the A above middle C) with velocity 73 on channel 0. These numbers are in hexadecimal.



A MIDI message made up of three bytes

The first byte in the MIDI message is the so-called status byte. It tells the receiver what kind of command must be performed and on what channel. The status byte always has the most significant bit set to 1, giving it a value between 128 and 255, or `0x80` and `0xFF` in hexadecimal. This leaves three bits for the command and four bits for the channel.

The possible commands are:

- `8x` – Note Off
- `9x` – Note On
- `Ax` – Key Pressure
- `Bx` – Control Change
- `Cx` – Program Change
- `Dx` – Channel Pressure
- `Ex` – Pitch Bend
- `Fx` – system messages

Again, these numbers are in hexadecimal. Here, `x` is the channel number and can be between 0 and 15 (or `0x0F` hex).

MIDI defines 16 possible channels. In a hardware MIDI setup, you'd have each instrument play a part on its own channel. For example, it's common to put drums on channel 10. However, in software the channel is less important, as usually the messages from a MIDI region are sent only to a single plug-in at a time. JX11 will simply ignore the channel information and accepts notes on any channel, known as "omni" mode.

The meaning of the data bytes that follow the status byte depends on the command. The most significant bit in a data byte is always 0. That's how the protocol can tell the status byte apart from data bytes. Having only seven bits left over means data bytes carry values between 0 and 127, or `0x00` and `0x7F` in hex. Sometimes the two data bytes are combined into a 14-bit value.

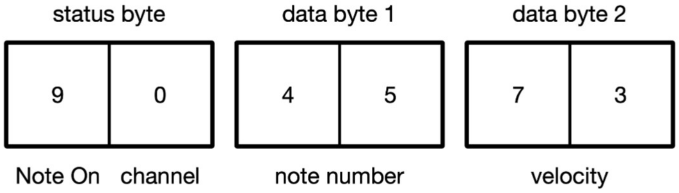
> **Note:** Most of the MIDI messages we'll use in the synth have two data bytes, a few have only one data byte. Certain MIDI hardware uses special timing messages consisting of a single status byte and no data bytes at all — but these are generally ignored by synthesizers. There are also MIDI messages that can have more than three bytes, the so-called system exclusive messages, but these are not relevant for our purposes and we can safely skip them.

Note On and Note Off

The two most important commands are Note On (`9x`) and Note Off (`8x`). Earlier, I said MIDI is like electronic sheet music, but there is an important difference: in sheet music, notes have durations. This is not true for MIDI.

The Note On command tells the synth to play a certain note, but it does not specify the length of this note. That's because MIDI can be used for live playing. The Note On message is sent when the user presses a key on their MIDI keyboard — but we don't know in advance how long they will hold that note down for. Once the key is released, a Note Off message is sent for that key.

Let's look at that Note On message in detail again:

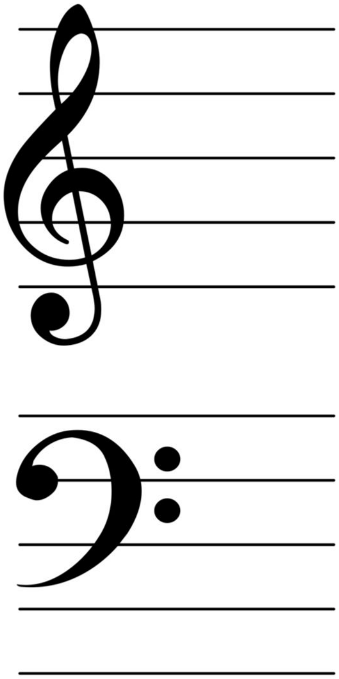


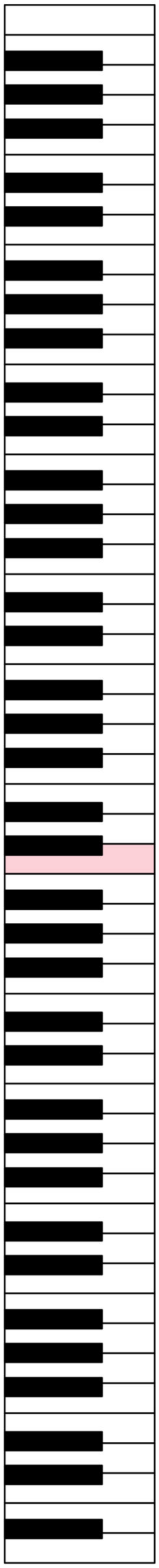
Example of a Note On message

The first byte means this is a Note On message — the command starts with 9 — on channel 0. The Note Off message has the same format as Note On, except that the status byte starts with 8 instead of 9.

The status byte is followed by two data bytes. For Note On and Note Off messages, the first data byte is the note number. There are 128 possible note numbers. That should be more than enough — even a piano, which can play very low and very high notes, only has 88 keys.

On the next page there is a useful MIDI note numbering chart, reproduced from a design by Joe Wolfe.





Note	Number		Frequency	
C8	108		4186.0	
B7	107		3951.1	
A7	105	106	3520.0	3729.3
G7	103	104	3136.0	3322.4
F7	101	102	2793.8	2960.0
E7	100		2637.0	
D7	98	99	2349.3	2489.0
C7	96	97	2093.0	2217.5
B6	95		1975.5	
A6	93	94	1760.0	1864.7
G6	91	92	1568.0	1661.2
F6	89	90	1396.9	1480.0
E6	88		1318.5	
D6	86	87	1174.7	1244.5
C6	84	85	1046.5	1108.7
B5	83		987.77	
A5	81	82	880.00	932.33
G5	79	80	783.99	830.61
F5	77	78	698.46	739.99
E5	76		659.26	
D5	74	75	587.33	622.25
C5	72	73	523.25	554.37
B4	71		493.88	
A4	**69**	70	**440.00**	466.16
G4	67	68	392.00	415.30
F4	65	66	349.23	369.99
E4	64		329.63	
D4	62	63	293.66	311.13
C4	**60**	61	**261.63**	277.18
B3	59		246.94	
A3	57	58	220.00	233.08
G3	55	56	196.00	207.65
F3	53	54	174.61	185.00
E3	52		164.81	
D3	50	51	146.83	155.56
C3	48	49	130.81	138.59
B2	47		123.47	
A2	45	46	110.00	116.54
G2	43	44	97.999	103.83
F2	41	42	87.307	92.499
E2	40		82.407	
D2	38	39	73.416	77.782
C2	36	37	65.406	69.296
B1	35		61.735	
A1	33	34	55.000	58.270
G1	31	32	48.999	51.913
F1	29	30	43.654	46.249
E1	28		41.203	
D1	26	27	36.708	38.891
C1	24	25	32.703	34.648
B0	23		30.868	
A0	21	22	27.500	29.135

Note 60 (or `0x3C` in hexadecimal) is middle C. In terms of sheet music, this is the point where the bass clef and the treble meet. Another important note number is 69 (`0x45` in hex) for the A above middle C. This note has a pitch of 440 Hz, which is the reference that the pitches of all other notes are derived from using equal temperament tuning.

One common point of confusion is the naming of the notes. Some DAWs say that C3 is the name of middle C (note 60), others call this note C4. Personally, I prefer C4 as that makes the first full octave on the piano start at note C1 and the last note is C8. The lowest note on a standard 88-key piano is A0. But that's perhaps a very piano-centric point of view.

- If you consider C4 to be the middle note, the lowest possible MIDI note number (0) is C–1 and the highest possible note number (127) is G9.
- If the middle note is named C3, then the lowest note is C–2 and the highest note is G8.

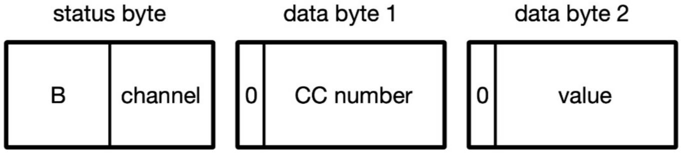
Either way, the numbering is weird and we end up with negative octave numbers for the lowest notes. Just be aware that different people use different naming schemes.

The second data byte in the Note On message is the velocity. This records how hard, or really how fast, the key was pressed and is a value between 0 and 127. Very inexpensive MIDI controllers may not support this, or may have coarse velocity levels with only 16 possible values instead of 128. In the example, the velocity was `0x73` or 115, meaning that the player was really hammering on those keys!

In a Note Off message, the second data byte is also for velocity. Many MIDI controllers provide a velocity measurement on Note Off. One way to take advantage of this, is to set the release time for the envelope using the Note Off velocity, so that notes that are released slowly will take longer to ring out. For MIDI controllers that don't support release velocity, the second byte will be set to 0 in the Note Off message.

Control Change (CC)

Besides Note On/Off, another important MIDI command is Control Change. This is used for many other features that MIDI controllers might have, such as the modulation wheel, the sustain pedal, and any other knobs and sliders. The Control Change message looks like this:



The format of a Control Change message

The command is `Bx` where `x` is the channel number. This means a control change message applies to all the notes currently playing in that channel. For JX11 this is not important as we ignore the channel number anyway.

The first data byte contains the identifier for the controller that is being changed. The second data byte is the new value for the controller and is 0 – 127.

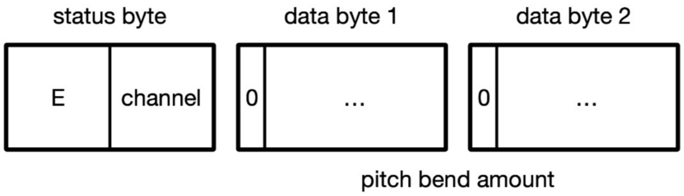
Here are a few of the more commonly used controller codes, known as MIDI CC or “continuous controller” codes:

- `01` – Modulation Wheel
- `07` – Channel Volume
- `0A` – Panning
- `0B` – Expression
- `40` – Sustain Pedal
- `7B` – All Notes Off (panic / reset)

The Control Change message is the most complicated one to handle because a lot of different functionality has been stuffed into it. But it’s also what makes MIDI very powerful.

Other MIDI messages

There is one controller on the MIDI keyboard that has a message type all of its own: the pitch wheel. The Pitch Bend message looks like this:



The format of a Pitch Bend message

As with Control Change, this applies the pitch bend to all the notes in the given channel. The reason this has its own dedicated message is that the two 7-bit data bytes get combined into a single 14-bit value. Instead of having a fairly limited range of 0 – 127 like the other controllers, Pitch Bend has a range of –8192 to +8191 where 0 means no pitch bending is applied.

The MIDI commands for Key Pressure (`Ax`) and Channel Pressure (`Dx`) are for changing the character of the sound after the note has started playing and the key is pressed down further, also known as aftertouch. This is an advanced feature that may not be available on inexpensive MIDI controllers (mine doesn't have it). The JX11 synth will support channel pressure in case it is available.

The Program Change command (`Cx`) is for changing the instrument that's playing on a certain channel. In JX11 we use this to cycle through the factory presets, but it's really more a feature that's used with MIDI hardware and not so much with plug-ins. Unlike the other message types you've seen so far, the Channel Pressure and Program Change messages only have one data byte, not two.

The System Messages (`Fx`) are for things like timing, setting the song position, and the "free for all" system exclusive or sysex message. We can safely ignore these in our software synthesizer.

Since the MIDI protocol is very old and hardware was much slower back then, the MIDI protocol allows for some optimizations. In the MIDI wire protocol, the status byte may be omitted if the previous message started with the same status byte. This is called "running status". Fortunately, JUCE will always give you all the bytes for each MIDI message so this isn't something you need to worry about in the plug-in.

However, there is an optimization you do need to handle: a Note On event with velocity set to 0 should be interpreted as a Note Off event. The reason this exists is that Note Off velocity is an advanced feature that isn't used by a lot of MIDI hardware, so instead of sending `90 XX YY` followed by `80 XX 00`, running status allows the protocol to send `90 XX YY` followed by `XX 00`, saving one byte. To the plug-in this looks like two Note On messages in a row, but the second one should be treated as Note Off.

MIDI summary

This was a quick overview of how MIDI works. As a synth creator, you need to understand at least how to handle Note On, Note Off, and Control Change messages, because this is how the DAW tells the synth what music it needs to play.

Either the user is performing live and those MIDI messages are sent from their controller keyboard to your plug-in in real-time, or the MIDI messages have been recorded into a MIDI region and the DAW is playing back these commands. To the plug-in it doesn't really matter where the MIDI messages come from, only that it needs to respond to them and turn these commands into actual audio somehow.

A limitation of the original MIDI standard is that messages such as Control Change, Pitch Bend, and Channel Pressure affect all the notes on the channel in the same way. Players do not get control over the individual notes with these messages. Fairly recently, an extension to MIDI was adopted, MPE or MIDI Polyphonic Expression, that works around this limitation.

The idea behind MPE is that each new note is now assigned to its own channel so that per-channel messages effectively become per-note messages. This works up to 16 notes, since there are 16 channels. JX11 was not made for MPE but JUCE does have support for it in case you want to put MPE in your own synths.

In 2020, MIDI 2.0 was released. This is a significant improvement over the original spec. It delivers fine-grained controller values that are now 32 bits instead of only 7 bits, up to 256 channels, per-note control as in MPE, and much more. Since MIDI 2.0 is still very new, this book does not cover it.

To read more about MIDI, the MIDI Association publishes all the specifications on their website. You do need to make a free account to download them.

- Official MIDI Specifications[35]
- Summary of MIDI 1.0 Messages[36]
- MIDI 2.0 Specification[37]

So what exactly is digital audio?

Earlier I mentioned that sound, once it's inside the computer, is described by a stream of 44100 or more numbers per second that are called samples. These samples represent the displacement of the loudspeaker cone over time. They describe how the speaker's diaphragm should move back and forth to make the air vibrate — and turn it into something that our ears and brains interpret as sound.

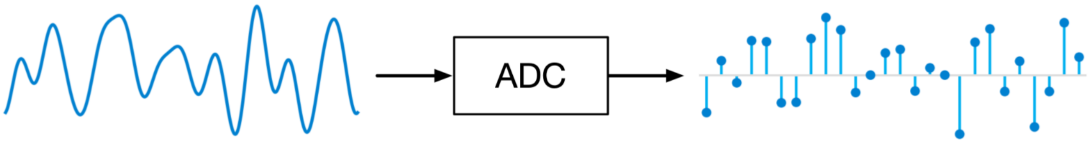
Why 44100 samples per second, that seems like such a strange number?

Instead of producing sound, let's say that we're going to record it with a microphone. The movement of air molecules in the sound wave will cause the microphone's membrane to vibrate inside a magnetic field, creating a continuously changing voltage. This voltage is

[35] https://www.midi.org/specifications
[36] https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message
[37] https://www.midi.org/specifications/midi-2-0-specifications

converted by the computer’s ADC (analog-to-digital converter) into a series of numbers, the samples.

The ADC does this by taking a voltage reading every couple of microseconds and storing that voltage into a 16-bit or 24-bit integer. It takes an analog, continuous signal that is made up of a smoothly changing voltage, and turns it into a digital, discrete signal consisting of a stream of integer sample values. The host application maps these integers into floating point numbers between –1 and +1, as they are easier to work with.



From analog to digital

There is a very important theorem in digital signal processing that says the sampling rate, or the speed at which the ADC takes these voltage readings, must be at least twice the highest frequency in the signal that is being sampled. If the sampling rate is not high enough, the original analog signal will not be faithfully represented by the sampled digital signal.

Human hearing tops out at roughly 20000 Hz (or 20 kHz) although most people cannot hear frequencies that high — I’m in my mid-forties and like most people my age, I can’t hear anything over 15 kHz. If we take 20 kHz as the upper limit of the frequencies to record, the sampling rate should be at least 40 kHz or 40000 samples per second.

In practice, a sampling rate of 44100 samples per second or 44.1 kHz is used. This odd number is the result of choices that were made in older technologies for video and audio, such as the Compact Disc, and the math for that happened to work out to 44.1 kHz. As this sampling rate is sufficiently high to capture the highest frequencies that (young) human ears can hear, that’s what the industry settled on.

These days, it’s common to use a sampling rate of 48 kHz or 48000 samples per second. The difference with 44.1 kHz is small and mostly does not matter. Some folks even use 96 kHz or higher, although a higher sample rate isn’t necessarily better. 44.1 kHz is already good enough to capture all sounds humans can hear, and the higher the sample rate, the more numbers it takes to describe the sound. At some point, increasing the sampling rate is a waste of perfectly good numbers.

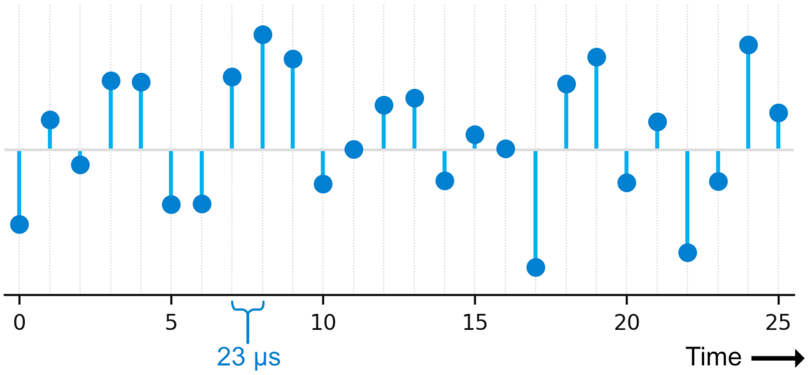
That said, there are valid reasons for using a higher sample rate inside a plug-in, a technique known as oversampling. This helps to reduce the occurrence of every audio programmer’s nemesis: aliases. We’ll talk more about aliases in the coming chapters.

Note: In your audio programming career, you may run into audio files with much lower sample rates such as 16 kHz or even 8 kHz. These lower sample rates aren't typically used for synthesis but can be OK for recording if the original sound doesn't have many high frequencies, such as speech.

A closer look at samples

For DSP programmers, dealing with sampled data is our bread and butter. Therefore, it's a good idea to take a closer look at exactly what samples are.

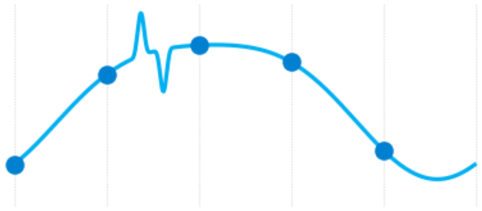
Here's a picture of a short snippet of audio. The sample rate is 44100 Hz and there are 26 samples in this snippet, so this corresponds to approximately 0.6 milliseconds of sampled data. (If you're wondering how I got that number: 26 samples divided by 44100 samples per second gives 0.000589 seconds or 0.6 msec rounded off.)



A zoomed-in view of a sampled signal

The horizontal axis has the time in seconds. Note that there is spacing between the samples. This is correct, since the computer's ADC takes voltage readings every few microseconds. At the sample rate of 44.1 kHz, that is one sample every 1 / 44100 = 0.0000227 seconds or 23 microseconds, so the time interval from one sample to the next is also 23 microseconds. The higher the sample rate, the smaller this spacing.

You might be wondering how the sampled data can faithfully represent the original signal. Surely, since we only take readings every so often, we must lose some information along the way? For example, what if the following happens...



A wiggle in between two samples

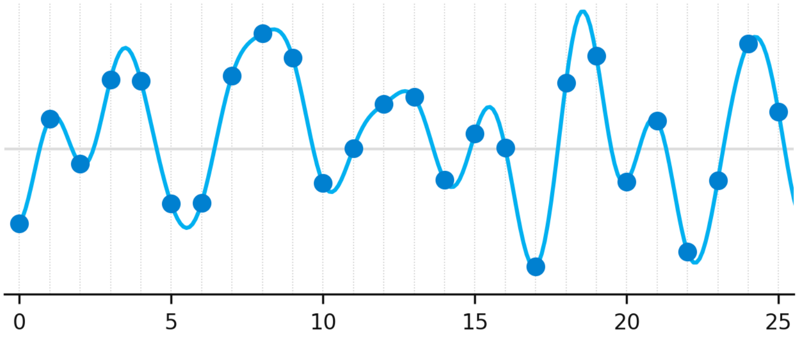
After sampling this signal, we have no idea that this “wiggle” occurred in between the two samples, since the ADC never saw it happen. Here’s the kicker: if such a wiggle can take place, the chosen sampling rate is too low. The original analog signal apparently has much higher frequencies in it than we expected.

When using sampled data, we will make the assumption that the signal is band-limited, meaning it does not contain frequencies that are higher than half the sampling rate. In other words, we need to make sure that the sampling rate is at least twice as high as the highest frequency that occurs in the signal.

The maximum frequency the signal can contain is known as the Nyquist limit. When using a sampling rate of 44.1 kHz, the Nyquist limit is 22.5 kHz, and the sampled signal can faithfully represent any frequencies lower than 22500 Hz — well above the hearing range of humans. But frequencies higher than 22.5 kHz won’t work.

By the way, an ADC has an electronic circuit that filters out all frequencies that are too high for the chosen sampling rate, so the “wiggle” situation isn’t a problem in practice — unless you actually want to capture these high frequency fluctuations in your digital signal, in which case you’d need to increase the sampling rate.

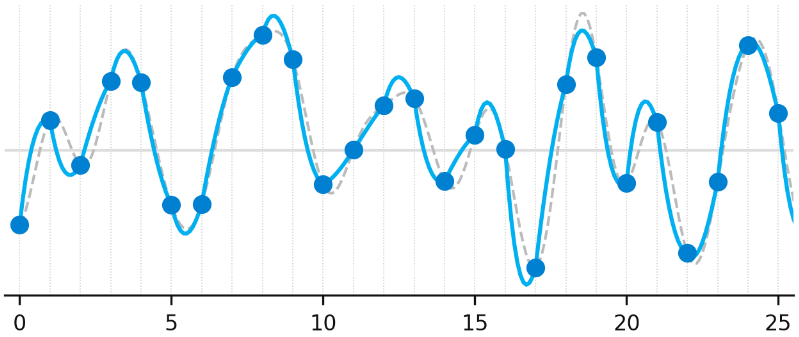
The takeaway here is that, when the sampling rate is high enough, the sampled signal is equivalent to the original, continuous signal. Given a band-limited signal, we know *exactly* what the audio looks like between the samples, as this is the only possible signal that can be described by these samples. For the samples shown previously, the real signal is as follows:



Reconstruction of the signal between the samples

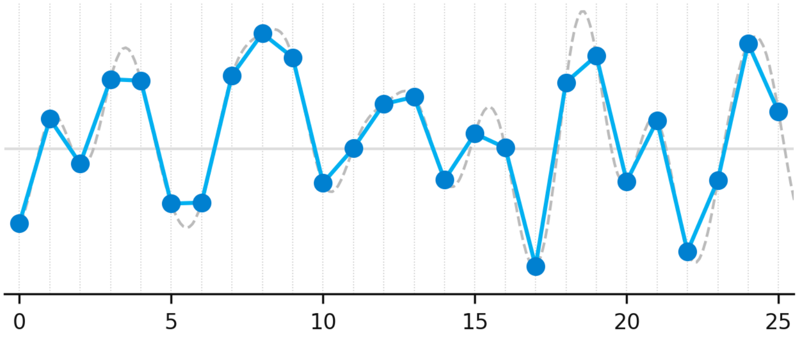
Note that the line that connects the samples is smooth. It may sometimes go higher or lower than the two closest samples, but it never wiggles. We can calculate this signal exactly from the given samples. For readers who are good at math: this is done by convolving the sampled signal with the function `sin(πx) / πx`. You may immediately forget this for now.

The important thing to remember is: There is only one correct mathematical solution. You can't just draw any line between the samples. For example, a polynomial fit might also go through all the sample points but this does not represent the original analog signal (the dotted line is the true original):



Polynomial interpolation of the signal

Linear interpolation is even worse. I'm pointing this out because determining what the signal looks like in between samples is a very common task in audio programming. Often linear interpolation is used where we imagine there is a straight line between each pair of samples. As you can tell from the picture, that isn't even close to what the signal really looks like.



Linear interpolation of the signal

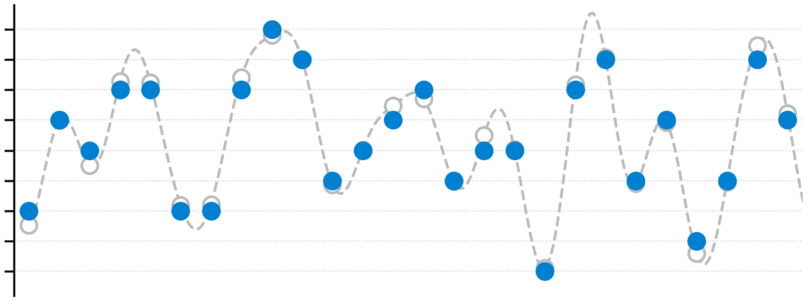
The polynomial approximation might be fairly similar to the real signal at first glance, but sometimes still misses the mark completely. Unfortunately, the math for finding the exact shape of the continuous analog signal is slow, so in practice we have no choice but to use some kind of approximation if we want to know what happens in between the samples.

Quantization and bit depth

So far, we've discussed what happens along the time axis, but the vertical axis — the height of the samples — is also important. When recording audio, each sample is a measurement of the air pressure on the microphone's membrane at that instance. As we play back that sample, it represents how much air pressure the speaker cone generates.

In JUCE the sample values are floating point numbers between –1 and +1, but the ADC and DAC work with integers. On modern computers, those are usually 16-bit or 24-bit integers. This is called the bit depth of the sample. With a bit depth of 16 bits, the sample can record the sound pressure level as 65536 possible values. With a 24-bit integer this becomes 16.8 million possible values.

While sampling, the ADC must put the continuous voltage reading it takes into a discrete number with a limited number of bits. It does this by rounding off the voltage to the nearest whole number. That rounding-off process is called quantization and the error it introduces into the signal is called quantization noise.



Exaggerated depiction of quantization. The blue dots are rounded to the nearest integer.

The more bits you're using to store the samples, the smaller the effect of this quantization noise. 24 bits gives a more faithful representation of the original voltage than 16 bits. The question is: does this difference matter in practice?

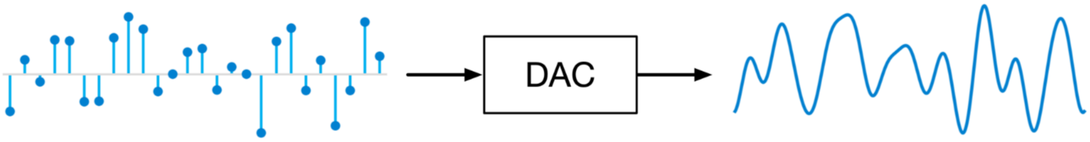
With 16-bit samples, the noise from the quantization error is 65536 times quieter than the actual sound, giving a signal-to-noise ratio of 96 dB. It's pretty much impossible for human ears to hear sounds at –96 dB, unless a massive amount of amplification is applied. For this reason, 16-bit audio is generally considered to be good enough. However, 24-bit audio is very common these days and makes the noise floor even lower.

For a plug-in programmer, this distinction doesn't really matter that much, as we're using floating point numbers. If you've worked with floats before you might be a little suspicious because this data type can have precision issues. For example, the number 0.1 is really

0.10000000149. However, 32-bit single-precision floats have a precision of 24 bits, so they should be able to handle most audio just fine. If necessary, JUCE and most other audio APIs let you use 64-bit double-precision floating point.

Digital audio summary

What I described above is what happens when audio is recorded through a microphone or audio interface into the computer. Digital audio works exactly the same way when you synthetically generate audio in a plug-in, with the main difference that now you create the sampled signal from scratch. Eventually those samples are turned into real sound by the DAC.



From digital to analog

If the sample rate is set to 44.1 kHz, the plug-in needs to produce 44100 samples per second, which is one sample every 23 microseconds. The computer's DAC takes this sampled data and turns it back into a continuous voltage that it uses to drive the loudspeakers. For stereo sound, the plug-in needs to output two times 44100 samples per second, one stream for the left speaker and one stream for the right speaker.

Bit depth and quantization noise aren't really something to worry about when writing plug-ins, especially synths. At the end of the signal path, the DAW can add so-called dithering. This adds small amounts of noise into the sound on purpose to hide the rounding errors from quantization.

However, it *is* important to understand the sample rate and what happens in between the samples. The main things to remember are:

1. The audio should not try to describe any frequencies that are higher than the Nyquist limit of half the sample rate. In other words, the digital signal should be band-limited. When recording, the ADC's filter takes care of this already, but when writing a synthesizer it's very easy to go over the Nyquist limit. We'll revisit this in chapter 6 where you'll experience first-hand what happens when this fundamental rule is broken.

2. Even though you only have samples that describe a fraction of the true analog signal, this is enough to reproduce the analog signal exactly. Assuming that it is band-limited, sampling a signal does not lose any information about that signal! You can always predict exactly what the continuous signal looks like between any two samples.

In one of the audio programming groups I'm in, we recently had a debate whether it makes sense to speak of "samples" even if you're generating the sound synthetically. Clearly, an ADC literally takes periodic readings of the voltage, but a synthesizer creates these sample values out of thin air — so are they really "samples"?

The way I think about it is as follows: Even though we're working with sampled signals, these samples still represent an analog, continuous signal. Whether the digital signal actually gets turned into real sound at some point is irrelevant. For example, the synthesizer output may go into some other plug-in rather than directly to the speakers. It still makes sense to think of the stream of numbers as being samples of some imaginary analog signal, even if that signal never really existed in the analog world.

Tip: Christopher "Monty" Montgomery made an excellent video[a] about how digital audio works. Recommended viewing for all audio programmers!

[a] https://www.youtube.com/watch?v=cIQ9IXSUzuM

Audio blocks and buffers

Enough theory, let's look at some code.

You've seen that a synth takes MIDI messages as input, and it produces audio samples as output. Both these things happen in the plug-in's most important method, `processBlock`.

The signature for this method is:

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                       juce::MidiBuffer& midiMessages)
```

This method has two parameters. The first is a `juce::AudioBuffer` object. This is where the synth will place the audio samples it generates. The second parameter is a `juce::MidiBuffer` object that contains the incoming MIDI messages.

Whenever the host application needs new audio from the plug-in, it will call `processBlock`. That's why this method is referred to as the "audio callback". This happens hundreds or even thousands of times per second. If there are also new MIDI messages at that point, these get passed into the `MidiBuffer` object.

I’ve mentioned a few times now that digital audio is made up of a “stream” of 44100 or more samples per second. While this is technically correct, it would be really inefficient for the DAW to ask our synth to output its audio one sample at a time. That’s why the audio stream is chopped up into smaller portions, so-called blocks. The plug-in will output one block’s worth of samples at a time.

A typical block contains 128 samples but they can be as small as 16 samples or as large as 4096 samples. If the block size is 128 samples and the sample rate is 44.1 kHz, the plug-in’s `processBlock` is called 44100 / 128 = 344.53 times per second on average, so let’s say 345 times. That’s a lot, and so you can imagine the audio callback needs to be fast. The smaller the size of the block, the more often `processBlock` needs to be called by the host.

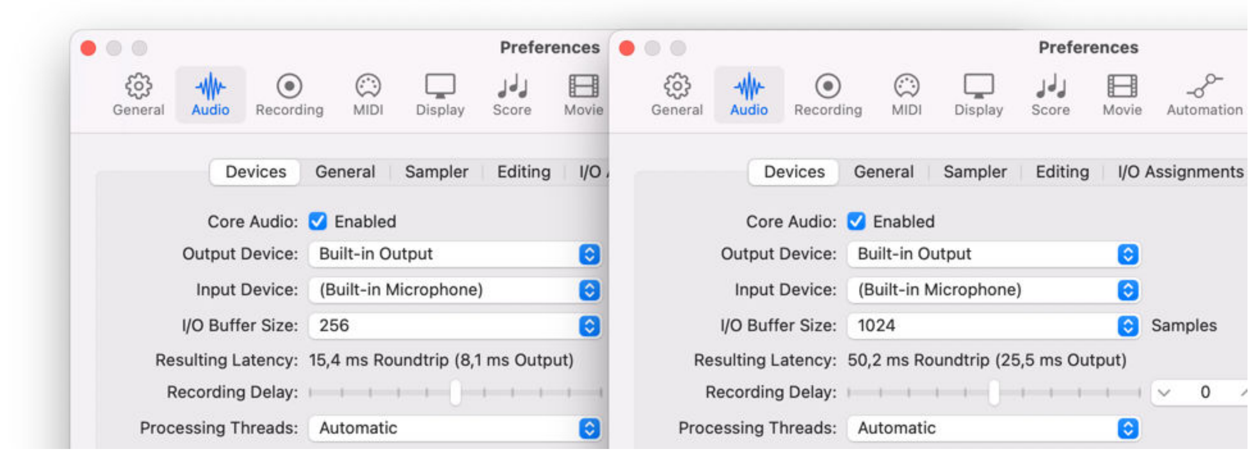
The size of these blocks is an important trade-off between latency and overhead:

- There is a certain cost for the host to call `processBlock`. By making the plug-in work on blocks of samples instead of individual samples, the same amount of overhead is spread out across multiple samples, which is more efficient.
- On the other hand, the larger the block is, the longer the delay between asking for the block and receiving the results from the plug-in. That delay is known as the latency.

Imagine what happens if the block size is set to 44100 samples. Since each call to `processBlock` now needs to fill up the `juce::AudioBuffer` with one second of audio, there is a one second delay between the user pressing keys on their MIDI controller and sound coming out of the speakers. Let me tell you, that makes it really tricky to play the instrument! Ideally, the latency is so small that you don’t notice it.

Latency is unavoidable when using digital audio and it comes from several different places: the USB driver from your MIDI keyboard, the computer’s audio interface, the internal processing of the host, and finally, the block size used for audio processing. Most DAWs let the user set the block size, although they usually call it the buffer size instead — both terms mean the same thing and are used interchangeably.

For example, the image below shows the audio settings screen in Logic Pro with I/O Buffer Size being the setting for the block size. Notice how this calculates the total latency, under Resulting Latency.



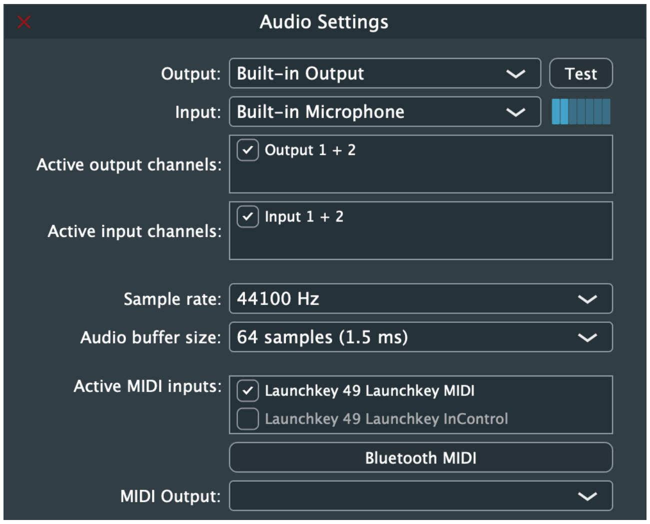
Core Audio: ☑ Enabled
Output Device: Built-in Output
Input Device: (Built-in Microphone)
I/O Buffer Size: 256
Resulting Latency: 15,4 ms Roundtrip (8,1 ms Output)
Recording Delay:
Processing Threads: Automatic

Core Audio: ☑ Enabled
Output Device: Built-in Output
Input Device: (Built-in Microphone)
I/O Buffer Size: 1024 Samples
Resulting Latency: 50,2 ms Roundtrip (25,5 ms Output)
Recording Delay: 0
Processing Threads: Automatic

Setting the I/O buffer size in Logic Pro

For larger buffer sizes, the latency increases. At a sample rate of 44.1 kHz, a buffer of 256 samples has 5.8 milliseconds latency while a buffer of 1024 samples has 23.2 milliseconds latency. That's exactly the difference shown in these two images: 8.1 ms output latency versus 25.5 ms. The remainder of the latency, about 2.3 ms, is caused by the computer's hardware, the operating system, and Logic Pro itself.

You can also set the audio buffer size in AudioPluginHost. Launch the graph that you made in the previous chapter, and double-click on the **Audio Input** block to open the settings window. Here it is set to 64 samples:



Output: Built-in Output
Input: Built-in Microphone
Active output channels: ☑ Output 1 + 2
Active input channels: ☑ Input 1 + 2
Sample rate: 44100 Hz
Audio buffer size: 64 samples (1.5 ms)
Active MIDI inputs: ☑ Launchkey 49 Launchkey MIDI
□ Launchkey 49 Launchkey InControl
MIDI Output:

Setting the audio buffer size in AudioPluginHost

Why would you not set the buffer size to the smallest possible? After all, that would get rid of the latency, wouldn't it… True, but there is another thing to consider: can the computer keep up?

Don't miss that deadline!

One of the most important rules in audio programming is that we want to avoid glitches in the sound. Glitches can be things like cracks or pops or other nasty sounds that aren't supposed to be there. There are lots of ways glitches can happen but one common cause is the plug-in being unable to meet its processing deadline.

Because `processBlock` gets called hundreds or thousands of times per second, each of these invocations has only a very short time available to do its work. With a 128-sample buffer at 44.1 kHz, `processBlock` is called 345 times per second on average. That means each of these calls should complete in less than 1/345 seconds or approximately 2.9 milliseconds. That's not a lot of time, and it includes the overhead that the host incurs for everything that's involved in calling `processBlock`, so the actual time is even shorter.

In this example, a call to `processBlock` has a deadline 2.9 ms in the future. The smaller the buffer size, the shorter this deadline becomes. At some point, if the buffer size is too small, the overhead plus the actual work the synthesizer does may cause the plug-in to miss its deadline. This results in an audible glitch, as the host will not have received the filled-up audio buffer in time and so it doesn't have any sound to output. This is bad! Users won't like your plug-in if it glitches.

On a fast enough computer, the buffer size can probably be set to the smallest possible size, often 16 or 32 samples. But you may find that the audio starts cracking up. That's a surefire sign that the blocks are too small and the audio processing is missing its deadlines.

This deadlines thing is serious business! It can even be an issue when the buffer size is ample. The `processBlock` method runs on a special high-priority thread, known as the audio thread. It is essential that your audio callback never blocks this thread from running. A lot of operations that are perfectly fine anywhere else in your program are forbidden from the audio thread, such as memory allocations and system calls. Any operation that can potentially put the audio thread to sleep should be avoided.

To communicate between `processBlock` and the plug-in's UI, which run in different threads, you might be tempted to use lock such as a mutex. Bzzt! Can't do that because it might pause the audio thread for an indeterminate amount of time so that it ends up missing the deadline. This is why there are special idioms in audio programming for communicating between different threads, such as atomic variables and lock-free circular buffers, that aren't common in regular programming.

Even something like appending to a `std::vector` is a no-no. This is a so-called amortized O(1) operation, meaning that most of the time it runs really fast, but sometimes — when the vector is full — it needs to allocate new, larger storage and copy over the vector's contents.

That allocation can cause the audio thread to freeze up, which is why this is not a good idea. But even if the allocation doesn't block, how much time the resizing operation takes is unpredictable. It can work fine 99% of the time but that one time it takes too long you'll still miss the deadline and glitch.

The take-away here is that we process audio in blocks of multiple samples at a time. The size of these blocks matters because larger blocks mean more latency, while smaller blocks means it becomes harder to meet the deadline. There are also things you're not allowed to do in the audio thread because it should never be kept waiting.

The block size isn't always the same

Even though the user configures the audio buffer size in the host, the size of the block isn't necessarily the same from one invocation of `processBlock` to the next. The host may decide to call `processBlock` with a smaller block size if it thinks that is a good idea. Your plug-in can't assume anything here, except that the block size won't be larger than the `samplesPerBlock` value that is passed into the plug-in's `prepareToPlay` method.

This method is defined in **PluginProcessor.cpp** and has the following signature:

```
void JX11AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
```

`prepareToPlay` is called before the host starts using the plug-in. It lets the plug-in know what the current sample rate is and what the maximum block size is that it can expect. This maximum is generally the buffer size chosen by the user.

FL Studio is an example of a host that uses variable block sizes. It can even call `processBlock` with a block size of 1, a single sample. This is unfortunate because that creates the kind of overhead we were trying to avoid. But such is the life of a plug-in developer: the plug-in runs inside the host environment and you'll have to work with whatever the host gives you.

The reason DAWs may occasionally use smaller block sizes, is when automation is active on a track. The DAW wants to make sure that the plug-in processes parameter changes from these automation events in time. Usually parameter changes are handled at the start of the block, and waiting until the start of the next block may be too late, so the solution is to force the plug-in to work on smaller blocks.

Inside `processBlock` you can ask the `juce::AudioBuffer` object what the actual block size is with `buffer.getNumSamples()`. You should always output exactly that number of samples, no more, no less.

In **PluginProcessor.cpp**, the `processBlock` method currently contains some boilerplate code from the Projucer template. Change `processBlock` to the following:

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                       juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;

    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    // Clear any output channels that don't contain input data.
    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i) {
        buffer.clear(i, 0, buffer.getNumSamples());
    }
}
```

This simply clears out the `AudioBuffer` object by setting all the samples to zero. This is necessary because plug-ins can also have an audio input. In that case, the incoming audio data is provided in the `AudioBuffer` object and the plug-in is supposed to overwrite these samples with the processed audio. That's how effect plug-ins work, but even synths can have a sidechain input.

JX11 does not have an audio input and JUCE does not guarantee that the `AudioBuffer` is already clear. It may contain garbage values, which in the best case results in silence when played back, but in the worst case result in extremely loud feedback — not pleasant, to say the least. Therefore, it's a good defensive programming practice to clear out the buffer first so we're sure it contains silence.

Note: Perhaps you're wondering what the `juce::ScopedNoDenormals` line does in `processBlock`. When floating point numbers become very small — almost but not quite zero — they switch to a different internal representation that's called a denormal or subnormal number. The problem with such denormals is that they are much slower than regular floats. Since denormal numbers are so small that they are practically zero, we might as well treat them as being zero. That's what the `juce::ScopedNoDenormals` line does. It sets a special CPU flag for the duration of `processBlock` that will automatically truncate floating point numbers to zero instead of turning them into denormal numbers.

Buses: mono and stereo

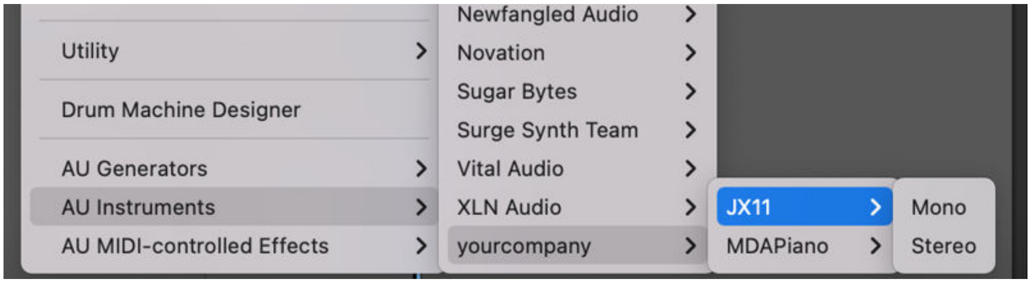
The `juce::AudioBuffer` object is where the synth will write the sample data that it generates. `AudioBuffer` may contain the samples for multiple channels. If the sound is mono, there is one channel. Stereo sound has two channels. It's also possible to have more than two channels, such as for surround sound. What kind of channels your plug-in supports is configured by its buses.

If you look past the `#if` preprocessor cruft, the constructor for `JX11AudioProcessor` does the following:

```
JX11AudioProcessor::JX11AudioProcessor()
     : AudioProcessor (BusesProperties()
                         .withOutput ("Output", juce::AudioChannelSet::stereo(), true))
```

This creates a `BusesProperties()` object with a stereo output bus named "Output". There is no input bus, so the plug-in will not receive audio data from the host.

However, if you try to load JX11 in its current form in a DAW, it may offer the option to use the plug-in in mono. Here's what it looks like in Logic Pro:



The plug-in shows up as mono and stereo

This happens because `JX11AudioProcessor` has another method, `isBusesLayoutSupported`, that is called by the DAW to query the number of channels it supports. The default implementation of this method is as follows:

```
bool JX11AudioProcessor::isBusesLayoutSupported (const BusesLayout& layouts) const
{
  #if JucePlugin_IsMidiEffect
    juce::ignoreUnused (layouts);
    return true;
  #else
    // This is the place where you check if the layout is supported.
    // In this template code we only support mono or stereo.
    // Some plugin hosts, such as certain GarageBand versions, will only
    // load plugins that support stereo bus layouts.
    if (layouts.getMainOutputChannelSet() != juce::AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != juce::AudioChannelSet::stereo())
        return false;

    // This checks if the input layout matches the output layout
   #if ! JucePlugin_IsSynth
    if (layouts.getMainOutputChannelSet() != layouts.getMainInputChannelSet())
        return false;
   #endif

    return true;
  #endif
}
```

Because `JucePlugin_IsMidiEffect` is false and `JucePlugin_IsSynth` is true, the only lines from this method that are used in our synth are:

```
    if (layouts.getMainOutputChannelSet() != juce::AudioChannelSet::mono()
     && layouts.getMainOutputChannelSet() != juce::AudioChannelSet::stereo())
        return false;

    return true;
```

This tells the host that the plug-in supports both mono and stereo buses. If the plug-in is used on a mono track, the `AudioBuffer` object will have one channel instead of two.

To change this plug-in so that it only can be used in stereo, change the `if` statement to:

```
    if (layouts.getMainOutputChannelSet() != juce::AudioChannelSet::stereo())
        return false;
```

Now rebuild the plug-in and try to open it in your DAW again. It won't make any difference in AudioPluginHost, but in Logic Pro there no longer is a Mono option. If your DAW still shows both mono and stereo options, you may have to reboot your computer first. Hosts don't always pick up changes to the supported bus layouts straightaway.

JX11 will support both mono and stereo channels, so if you changed this method, put it back to how it was.

In the next chapter you will take what you've learned about MIDI and digital audio and put the two together to make a very simple synth that reads MIDI input and produces an audio signal in response.

Chapter 4: Handling MIDI events in JUCE

You've seen that the audio processor's `processBlock` is given an `AudioBuffer` object for the audio data and a `MidiBuffer` object that contains the MIDI messages.

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                        juce::MidiBuffer& midiMessages)
```

Depending on the buffer size set in the host application, `processBlock` is called tens, hundreds, or thousands of times per second. If there are new MIDI messages at that point, these get passed into the `MidiBuffer` object.

But MIDI messages happen relatively rarely, at most a few times per second — unless someone is playing *Flight of the Bumblebee* — and so the `MidiBuffer` is usually empty. But when it's not empty, you'll have to do something with those MIDI messages!

Let's look at `MidiBuffer` in a bit more detail. Whenever you encounter a JUCE class you haven't seen before, it's always a good idea to look up the documentation[38]. Since you have access to the JUCE source code, you can even open the code for the `MidiBuffer` class in your IDE in case the documentation isn't sufficient.

`MidiBuffer` is a container object that holds MIDI messages. To look at the MIDI messages, you can iterate through the buffer like so:

```
for (const auto metadata : midiMessages) {
    if (metadata.numBytes == 3) {
        // the status byte is in metadata.data[0]
        // the data bytes are in metadata.data[1] and metadata.data[2]
    }
}
```

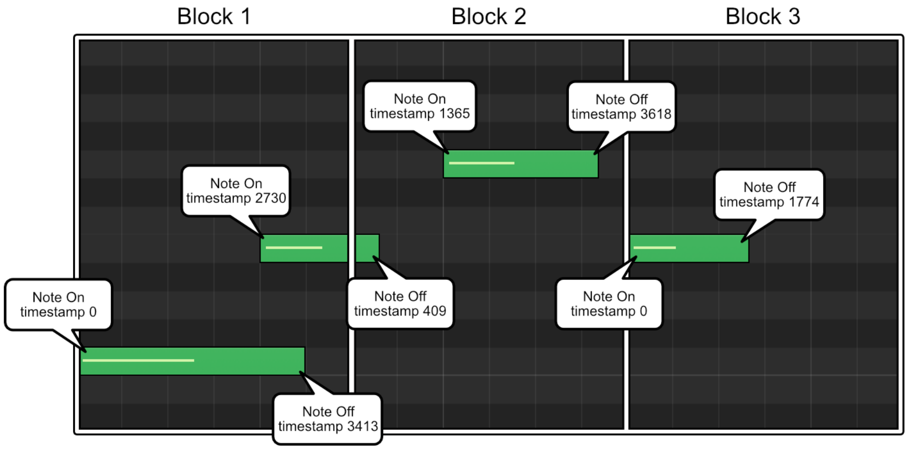
The object that performs the iteration is `MidiBufferIterator`, and it gives you `MidiMessageMetadata` objects. Again, feel free to check out the documentation for these classes on the JUCE website. `MidiMessageMetadata` is pretty simple and it just has the three bytes that make up the MIDI message, plus a timestamp.

The timestamp is in `metadata.samplePosition`. This is the number of samples relative to the start of the `AudioBuffer`. What is this for?

[38] https://docs.juce.com/master/classMidiBuffer.html

Let’s assume the user is playing back a prerecorded MIDI region in their DAW, and the buffer size is 4096 samples. That’s a large buffer size but the user isn’t playing live, so latency isn’t really important here.

At a sample rate of 44.1 kHz, each block covers 4096 / 44100 = 0.093 seconds or almost one-tenth of a second. Since the block is so large, it’s quite likely that the timing of the MIDI events is such that notes should be started or stopped somewhere in the middle of that block. The timestamps tell you exactly when the MIDI event is supposed to happen.

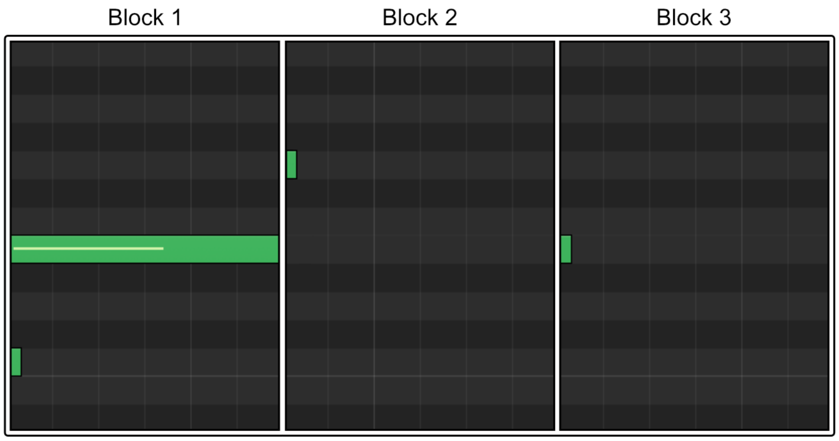


The timestamps are relative the start of the block

One way you could handle the MIDI messages is to loop through them at the start of `processBlock`, like so:

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                       juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;

    // ...clear the buffer...

    for (const auto metadata : midiMessages) {
        // if this is a Note On event, start the note
        // if this is a Note Off event, stop the note
    }

    for (int sample = 0; sample < buffer.getNumSamples(); ++sample) {
        // ...output audio for the playing notes...
    }
}
```

The problem is that now you’re always starting and stopping all notes at the beginning of the block. That messes up the timing of the MIDI events. Instead of playing the notes at the correct times, what happens now is shown in the following image.

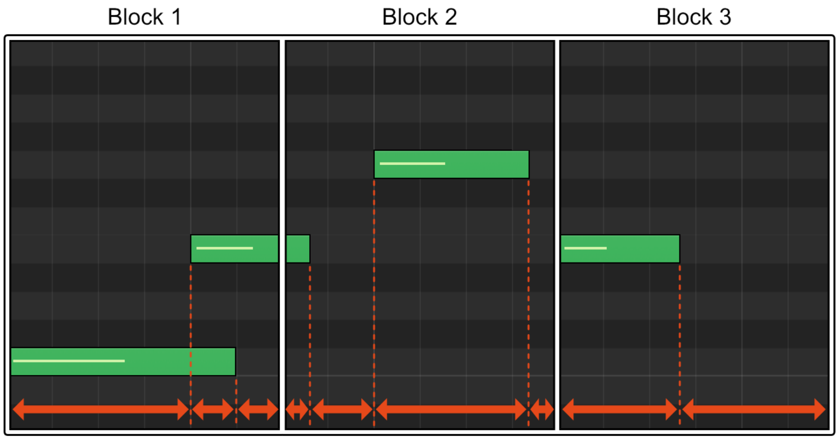


Processing all MIDI events at the start of the block

That’s not going to sound great… If a Note On message is followed by its Note Off in the same block, that note will never actually play!

You could actually get away with this approach if the block size is small enough. For a block of 32 samples instead of 4096, the timing is off by less than a millisecond. This error is small enough that listeners might never notice. But the plug-in is not in charge of the block size, the host is, and you can’t assume anything about it. So, you should definitely take the timestamps of the MIDI events into consideration!

The strategy that JX11 uses for dealing with the timing of MIDI messages, is to split the `AudioBuffer` into smaller pieces:



The block is split whenever a MIDI event occurs

Obviously, if there are no MIDI events you can simply process the `AudioBuffer` in its entirety. Otherwise, you will first process any MIDI messages at the start of the buffer, render a chunk of audio, then handle the next message, render the next chunk, and so on.

Note: MIDI messages will also have timestamps even if the user is playing live instead of using a recorded MIDI region. It's up to the DAW to handle the timing of the incoming MIDI events. It's up to the plug-in to respect these timestamps.

To implement this logic, you will add three new methods to `JX11AudioProcessor`. First add the declarations to the class inside **PluginProcessor.h**. These are private methods.

```
private:
    void splitBufferByEvents(juce::AudioBuffer<float>& buffer, juce::MidiBuffer& midiMessages);
    void handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2);
    void render(juce::AudioBuffer<float>& buffer, int sampleCount, int bufferOffset);
```

Then add the following code to **PluginProcessor.cpp**, below `processBlock`:

```
void JX11AudioProcessor::splitBufferByEvents(juce::AudioBuffer<float>& buffer,
                                             juce::MidiBuffer& midiMessages)
{
    int bufferOffset = 0;

    for (const auto metadata : midiMessages) {
        // Render the audio that happens before this event (if any).
        int samplesThisSegment = metadata.samplePosition - bufferOffset;
        if (samplesThisSegment > 0) {
            render(buffer, samplesThisSegment, bufferOffset);
            bufferOffset += samplesThisSegment;
        }

        // Handle the event. Ignore MIDI messages such as sysex.
        if (metadata.numBytes <= 3) {
            uint8_t data1 = (metadata.numBytes >= 2) ? metadata.data[1] : 0;
            uint8_t data2 = (metadata.numBytes == 3) ? metadata.data[2] : 0;
            handleMIDI(metadata.data[0], data1, data2);
        }
    }

    // Render the audio after the last MIDI event. If there were no
    // MIDI events at all, this renders the entire buffer.
    int samplesLastSegment = buffer.getNumSamples() - bufferOffset;
    if (samplesLastSegment > 0) {
        render(buffer, samplesLastSegment, bufferOffset);
    }

    midiMessages.clear();
}
```

JUCE has already sorted the MIDI messages by their timestamp, so if there is more than one MIDI event in `midiMessages`, they are already in the correct order.

For every MIDI event in `midiMessages`, you first process the audio up to that event's timestamp, using a `render` method that you're going to add in a minute. This is necessary for when the first MIDI event does not happen exactly at the start of the buffer, which it usually won't.

Next, you process the MIDI event using the new `handleMIDI` method. This receives the bytes that make up the MIDI message. For some commands such as Program Change there is only one data byte instead of two, in which case you set the remaining data byte to 0.

The loop continues until all MIDI messages have been processed. After the loop finishes and there is any audio remaining in the buffer, you call `render` on this final chunk of audio. If there were no MIDI messages at all, this statement processes the entire `AudioBuffer`.

Suppose there are three MIDI events in the `MidiBuffer` with the following timestamps, where messages 2 and 3 are intended to take place at the same time:

```
message 1: 90 45 73, timestamp = 132  (Note On)
message 2: 80 45 00, timestamp = 245  (Note Off)
message 3: 90 3C 64, timestamp = 245  (Note On)
```

Then `splitBufferByEvents` will call `render` and `handleMIDI` in the following sequence:

```
// start at sample 0 and render 132 time steps
render(buffer, 132, 0)

// handle the MIDI event at timestamp 132
handleMIDI(0x90, 0x45, 0x73)

// start at sample 132, render 113 steps until sample 245
render(buffer, 113, 132)

// handle the MIDI events at timestamp 245
handleMIDI(0x80, 0x45, 0x00)
handleMIDI(0x90, 0x3C, 0x64)

// render 11 steps from sample 245 until 255 (end of buffer)
render(buffer, 11, 245)
```

Thanks to this, you no longer need to worry about the timing of the MIDI events. Just process the MIDI message in `handleMIDI` and change the state of the synth accordingly. And in `render`, the next chunk of samples is always guaranteed to be between two MIDI events and does not need to be interrupted by anything.

Note: In practice, most of the time the MidiBuffer will be empty and render is called only once per block. In between two MIDI events there can easily be thousands of samples that get rendered. From the perspective of the plug-in, the rate at which it receives input data is slow compared to the rate at which it sends out data: a few MIDI events versus tens of thousands of samples every second.

However, if there are multiple MIDI events in quick succession, render may be called with a very small sample count, perhaps as little as one sample. This could be a problem if render does a fixed amount of work each time it's called and expects to spread the cost for that over multiple samples. One way to improve this approach is to round off the MIDI timestamps to the nearest 32 or so samples so that render is never called on less than 32 samples. You lose some timing precision this way but it might be worth the decreased CPU usage.

At the end of splitBufferByEvents, you call midiMessages.clear(). This lets JUCE know you've consumed all the MIDI events. It's not super important to do this for a synth, but for plug-ins that have MIDI output, any messages that are still in the MidiBuffer will be sent back to the host as new MIDI events.

Change processBlock so that it will call this new splitBufferByEvents method:

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                       juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;

    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i) {
        buffer.clear(i, 0, buffer.getNumSamples());
    }

    splitBufferByEvents(buffer, midiMessages);  // add this line
}
```

Next, you need to write the handleMIDI and render methods to do something with the MIDI data. To verify that MIDI actually works, you'll simply print the three MIDI bytes to the debug console.

```
void JX11AudioProcessor::handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2)
{
    char s[16];
    snprintf(s, 16, "%02hhX %02hhX %02hhX", data0, data1, data2);
    DBG(s);
}

void JX11AudioProcessor::render(
    juce::AudioBuffer<float>& buffer, int sampleCount, int bufferOffset)
{
    // do nothing yet
}
```

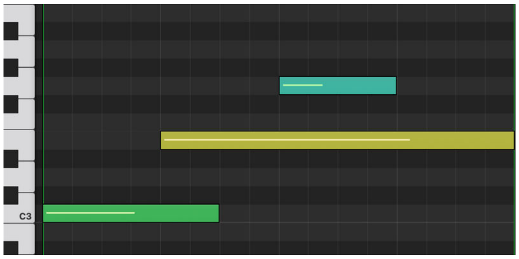
`DBG` is JUCE's debug macro. The `%02hhX` format specifier will turn the byte into a hexadecimal number. For now, you're leaving `render` empty because there is no audio to output yet, but you'll remedy that before the chapter is over.

Build the plug-in and try it out! Make sure your MIDI controller is enabled in AudioPluginHost by double-clicking the MIDI Input block and then selecting it under Active MIDI Inputs. You can also test the Standalone Plugin version. Click the Options button in the top-left corner of the window and select your MIDI controller in the Audio/MIDI Settings panel.

Press some keys on your MIDI keyboard or the on-screen virtual piano and you'll see the following kinds of messages being printed in the IDE's debug console:

```
90 3C 41
90 40 5A
80 3C 22
90 43 2D
80 43 48
80 40 46
```

You should recognize these as MIDI messages. Each has three bytes. There are three Note On messages here (status 90) and three Note Off (status 80), all in channel 0. The notes I played are middle C (3C hex), E (40 hex), and G (43 hex). I released the C after playing the E but before the G note. As a MIDI region in a DAW it might look something like this:



If you feel adventurous, you can test the plug-in in your DAW with a MIDI region to see what happens. However, you'll need to capture the debug output from the DAW somehow to see the `DBG` messages. On macOS you can do this by running the DAW from the Terminal.

> **Important:** You're not really supposed to print things to the debug console from the audio thread! Recall that `processBlock` runs in a special high-priority thread and `handleMIDI` is therefore on this audio thread as well. The one thing you are not allowed to do is stall the audio thread, and printing something to the console is a slow and potentially blocking operation. If you print a lot of stuff then `processBlock` might miss its deadline. It's a bad idea to do this kind of thing in production code, but `DBG` is OK for occasional debugging. Keep in mind that using something like `DBG` may change the timing of your audio code and can therefore introduce new bugs.

Creating the synthesizer objects

`JXAudioProcessor` is the main class for the plug-in and as such it's tempting to put all the audio processing code into this source file. However, this synth is going to end up with a lot of code and it will be much clearer if you already start putting that into their own objects.

In this section you will create a `Synth` class that represents the synthesizer. You will let the `JXAudioProcessor` handle things such as parameters and presets, and communication with the host, but `Synth` makes the actual sound. In fact, you won't put everything in `Synth` either but create a helper object `Voice` — and many more in the next chapters: `Oscillator`, `Envelope`, `Filter`, and others.

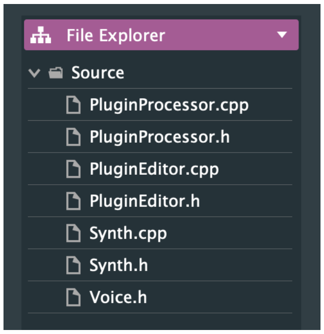
In **Projucer**, add three source files to the project:

- Synth.h
- Synth.cpp
- Voice.h

You do this by expanding the **File Explorer** section in the sidebar. Click the round + button and choose **Add New CPP & Header File…** from the popup.

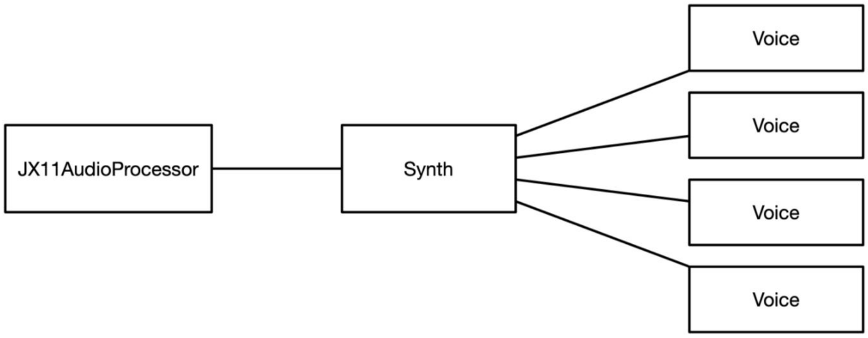
For **Voice.h**, only add a header file. The code for `Voice` is simple enough to fit in a header file, but `Synth` is fairly complex and therefore gets a cpp file as well.

Put these new files inside the **Source** group. Then press the export button again to update the project in your IDE.



Adding new files to Projucer

`Synth` is the main synthesizer class, while `Voice` is an object that manages a single playing note. In monophonic mode there will be only one note playing at time and therefore the synth just needs one `Voice` instance, but in polyphonic mode there will be a `Voice` instance for every note that is playing. It's a very common design to have this kind of split between the synthesizer as a whole and the individual sounds that it plays.



The Synth will eventually have multiple Voices, one for each note playing

Let's start with the `Voice` object. Put the following code into **Voice.h**:

```
#pragma once

struct Voice
{
    void reset()
    {
        // do nothing yet
    }
};
```

The job of `Voice` is to produce the next output sample for a given note. For now, the `Voice` object won't do anything yet, but you'll implement something here soon.

Put the following into **Synth.h**:

```
#pragma once

#include <JuceHeader.h>
#include "Voice.h"

class Synth
{
public:
    Synth();

    void allocateResources(double sampleRate, int samplesPerBlock);
    void deallocateResources();
    void reset();
    void render(float** outputBuffers, int sampleCount);
    void midiMessage(uint8_t data0, uint8_t data1, uint8_t data2);

private:
    float sampleRate;
    Voice voice;
};
```

Just like the `JX11AudioProcessor` class, `Synth` has methods to reset its state, to render the current block of audio, and to handle MIDI messages.

The `allocateResources` and `deallocateResources` functions are called right before the host starts playing audio and after it finishes playing audio. They are analogous to `prepareToPlay` and `releaseResources` from the audio processor. If necessary, the synthesizer can use this to allocate / deallocate any memory or other resources it needs.

For the time being, `Synth` has a single `Voice` object, so it will only be able to play one note at a time. In chapter 10 you'll make JX11 into a proper polyphonic synth that can play multiple notes at once, such as chords.

Add the implementations to **Synth.cpp**:

```
#include "Synth.h"

Synth::Synth()
{
    sampleRate = 44100.0f;
}
```

```
void Synth::allocateResources(double sampleRate_, int /*samplesPerBlock*/)
{
    sampleRate = static_cast<float>(sampleRate_);
}

void Synth::deallocateResources()
{
    // do nothing
}

void Synth::reset()
{
    // do nothing yet
}

void Synth::render(float** outputBuffers, int sampleCount)
{
    // do nothing yet
}

void Synth::midiMessage(uint8_t data0, uint8_t data1, uint8_t data2)
{
    // do nothing yet
}
```

These methods mostly don't do anything yet, except for `allocateResources`, which stores the sample rate in a member variable.

In **PluginProcessor.h**, change the includes to:

```
#include <JuceHeader.h>
#include "Synth.h"
```

And add a new private member variable to the `JX11AudioProcessor` class:

```
private:
    Synth synth;
```

There are also a couple of changes to make in **PluginProcessor.cpp**. Change `prepareToPlay` and `releaseResources` to the following. This lets the `Synth` object to react to changes in the sample rate or maximum block size, so that it can reallocate any data it needs. In JX11 these methods don't actually perform allocations but it's useful to set this up anyway.

```
void JX11AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    synth.allocateResources(sampleRate, samplesPerBlock);
}

void JX11AudioProcessor::releaseResources()
{
    synth.deallocateResources();
}
```

Put that MIDI to work!

All right, let's do something useful with these MIDI messages. In the next section you'll write code to output a sound, which will be triggered by the user holding down a key on their MIDI controller. This means the synth will need to track whether a key is down or not by handling the Note On and Note Off messages.

In **PluginProcessor.cpp**, change `handleMIDI` to the following:

```
void JX11AudioProcessor::handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2)
{
    synth.midiMessage(data0, data1, data2);
}
```

For now, this simply passes the MIDI message to the `Synth` class. Later you will also handle certain MIDI messages directly in `handleMIDI`, such as Program Change to select a different factory preset.

Switch to **Synth.cpp** and implement `midiMessage` like so:

```
void Synth::midiMessage(uint8_t data0, uint8_t data1, uint8_t data2)
{
    switch (data0 & 0xF0) {
        // Note off
        case 0x80:
            noteOff(data1 & 0x7F);
            break;

        // Note on
        case 0x90: {
            uint8_t note = data1 & 0x7F;
            uint8_t velo = data2 & 0x7F;
            if (velo > 0) {
                noteOn(note, velo);
            } else {
                noteOff(note);
            }
            break;
        }
    }
}
```

Recall that the first byte of the MIDI message is the status byte. This consist of two parts: the command and the channel number. By doing `switch(data0 & 0xF0)` you only look at the four highest bits that make up the command while skipping the four lowest bits that have the channel number.

If you wanted to also find the channel this message applies to, you'd write the following:

```
  uint8_t channel = data0 & 0x0F;
```

For our synth, you will simply ignore the MIDI channel. A more advanced synth could play different sounds for messages arriving on different channels.

If the command is `0x80`, this is a Note Off message. In that case, call the `noteOff` method. This is a new method that you'll add shortly. Similarly, if the command is `0x90` or Note On, you call `noteOn`.

The first data byte is the note number. The second data byte is the velocity. Both should be values between 0 – 127 but just to make sure you do `& 0x7F`. This is probably a bit paranoid but defensive programming never hurt anyone.

If the velocity for the Note On event is zero, you treat it as a Note Off event. Recall that this is an optimization that can happen in MIDI because of the running status feature.

Note Off messages can have a velocity, but many MIDI controllers don't support release velocity, and JX11 doesn't use it for anything. That's why the velocity is passed into `noteOn` but not into `noteOff`.

To keep track of the note that is being played, you need some way to register the note number and velocity of the most recently pressed key. You could put this in some variables in `Synth` but as a note is associated with a voice, it makes more sense to keep track of it there.

Open **Voice.h** and add the following two variables to the struct.

```
struct Voice
{
    int note;
    int velocity;

    // ...
```

Switch to **Synth.h** and add the following private methods:

```
private:
    void noteOn(int note, int velocity);
    void noteOff(int note);
```

In **Synth.cpp**, add the implementations for these methods. `noteOn` simply registers the note number and velocity of the most recently pressed key.

```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;
    voice.velocity = velocity;
}

void Synth::noteOff(int note)
{
    if (voice.note == note) {
        voice.note = 0;
        voice.velocity = 0;
    }
}
```

In `noteOff` the `voice.note` variable is cleared but only if the key that was released is for the same note. For example, if the user plays C followed by E and then releases C, that C is no longer the current note and `noteOff` should ignore the event. Only when the E is released will this set the note number back to zero.

> **Note:** Here, note number 0 means “no note”. However, note 0 is technically a valid MIDI note number. It has such a low frequency that it’s hardly audible, about 8 Hz, and so it’s unlikely anyone would ever play it. Alternatively, you could have used a special value such as –1 to mean “no note playing”.

It’s a good idea to make sure the `note` and `velocity` variables are zero when the plug-in is initialized. That’s why `Synth` and `Voice` have a `reset` method. Implement this as follows, first in **Voice.h**.

```
void reset()
{
    note = 0;
    velocity = 0;
}
```

And then in **Synth.cpp**:

```
void Synth::reset()
{
    voice.reset();
}
```

You must call `Synth`'s `reset` method from somewhere in `JX11AudioProcessor`. Its base class `juce::AudioProcessor` has itself a reset method, and that's a good place to do it.

In **PluginProcessor.h**, add the following below the line for `releaseResources`:

```
void reset() override;
```

In **PluginProcessor.cpp**, add its implementation:

```
void JX11AudioProcessor::reset()
{
    synth.reset();
}
```

And in `prepareToPlay` call this `reset` method:

```
void JX11AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    synth.allocateResources(sampleRate, samplesPerBlock);
    reset();
}
```

You're passing on the "reset yourself" message to the `Synth` object, which passes it on to the `Voice`, which can then clear out the variables. This may seem like a bit of unnecessary indirection at this point but it is necessary to keep things manageable as you make the synth more and more complex.

Excellent, you have now implemented some very basic MIDI handling that keeps track of the most recent key pressed and its velocity. In the next section you'll use this to play a simple sound. It's humble beginnings, but over the next chapters you'll expand `midiMessage` to also handle MIDI Control Change messages, Pitch Bend messages, the sustain pedal, and much more.

Make some noise (literally!)

MIDI is of course only half the picture. The other half is writing sample values into the `AudioBuffer` to generate sound.

Let's start by making the simplest possible sound there is: noise. The most basic form of noise is so-called white noise, which has equal intensity at all possible frequencies. We're not just doing this as a silly example: noise is an important sound source in a virtual analog synth and you'll find a noise generator in many synthesizers. You're actually going to be making the first real part of the synth here!

Generating white noise is as simple as outputting a bunch of random numbers between –1 and +1. JUCE even has a `Random` class for this. However, the original MDA synth that JX11 is based on used its own pseudo random number generator and so we'll keep that one.

In **Projucer**, open the File Explorer on the left and press the + button to add a new header file named **NoiseGenerator.h**. Then click the export button to update your Xcode or Visual Studio project.

Open **NoiseGenerator.h** and put the following inside it:

```
#pragma once

class NoiseGenerator
{
public:
    void reset()
    {
        noiseSeed = 22222;
    }

    float nextValue()
    {
        // Generate the next integer pseudorandom number.
        noiseSeed = noiseSeed * 196314165 + 907633515;

        // Convert the integer to a float, to get a number between 2 and 4.
        unsigned int r = (noiseSeed & 0x7FFFFF) + 0x40000000;
        float noise = *(float*)&r;

        // Subtract 3 to get the float into the range [-1, 1].
        return noise - 3.0f;
    }

private:
    unsigned int noiseSeed;
};
```

How this works: the `noiseSeed` variable starts out with the value 22222. Every time you call the `nextValue` method, `noiseSeed` is updated using a very simple formula. This creates a unique sequence of integers that's somewhat random.

To use `noiseSeed` as an audio sample value, it needs to be turned into a floating point number between –1 and +1. To do this, a little trick is used. 32-bit floating point numbers between 2.0 and 4.0 have the hexadecimal representation 0x40000000 – 0x407fffff.

Writing `(noiseSeed & 0x7FFFFF) + 0x40000000` puts the integer value in that range. Then you cast it to a float to get a number between 2 and 4. Finally, to turn it into a number between –1 and +1, you simply subtract 3 from it.

> **Note:** To be fair, this is not a particularly good random number generator but it will do for this synth. The problem with pseudo random numbers is that they will eventually repeat. If the sequence repeats too quickly, the cycle might actually become a noticeable part of the sound. For this particular random generator, I verified that the sequence of integers does not repeat for at least the first million numbers, so it's not too bad. Feel free to substitute your own random number generator here (but note that not all random generators are safe for use in the audio thread).

Go to **Synth.h** and add the following include:

```
#include "NoiseGenerator.h"
```

In the section where the private members are, add a new member variable:

```
NoiseGenerator noiseGen;
```

Then in **Synth.cpp**, change `reset` to the following to set the noise seed to its initial value:

```
void Synth::reset()
{
    voice.reset();
    noiseGen.reset();   // add this line
}
```

Making sound happens in `Synth`'s `render` method, which is currently empty. This function takes a pointer-to-a-pointer argument that might warrant some explanation.

The method signature for `render` is:

```
void Synth::render(float** outputBuffers, int sampleCount)
```

You will need to call this from `JX11AudioProcessor`'s own `render` method. In **PluginProcessor.cpp**, implement `render` like so:

```
void JX11AudioProcessor::render(
    juce::AudioBuffer<float>& buffer, int sampleCount, int bufferOffset)
{
    float* outputBuffers[2] = { nullptr, nullptr };
    outputBuffers[0] = buffer.getWritePointer(0) + bufferOffset;
    if (getTotalNumOutputChannels() > 1) {
        outputBuffers[1] = buffer.getWritePointer(1) + bufferOffset;
    }

    synth.render(outputBuffers, sampleCount);
}
```

My goal when writing this book was to keep `Synth` as independent from JUCE as possible, so that it would be relatively easy to hook into other audio APIs for readers who don't want to use JUCE. That's why `Synth` does not render into a `juce::AudioBuffer` object but gets a naked `float*` pointer.

Because there may be two channels, you pass an array of two `float*` pointers to `Synth`, one for the left channel and one for the right channel. That's why its `outputBuffers` argument is `float**`. In case the `Synth` is configured to run on a mono bus instead of a stereo bus, only the first pointer is used and the second one will be `nullptr`.

To get a pointer to the audio data inside an `AudioBuffer` object, you call `buffer.getWritePointer()` and pass in the channel number. The samples for the `AudioBuffer` are stored as a contiguous block in memory. In some audio APIs the samples for the left and right channels are interleaved but in JUCE each channel is stored independent of the others.

Because earlier you split up the `AudioBuffer` based on the timestamps of the MIDI events, the call to `JX11AudioProcessor::render` does not necessarily start at the beginning of the `AudioBuffer`'s memory area, so you add `bufferOffset` to the pointer. That is the number of samples that have already been rendered for this block because there were MIDI messages.

If the plug-in runs in stereo, you do this twice: once for the left channel and once for the right channel. Finally, you call `synth.render` and pass it these two pointers.

> **Note:** To reiterate, the reason we're using naked pointers instead of `juce::AudioBuffer`, is to show that JUCE itself isn't really needed inside the `Synth` class. However, don't feel like you have to do this the same way in your own synths. There are a number of advantages to keep using `AudioBuffers`. For example, `AudioBuffer` has convenient methods for adding two buffers together, for crossfading between two buffers, and so on. By using naked pointers, you don't have access to this functionality in the `Synth` class.

Now, to make some sound, in **Synth.cpp** fill in the `render` function like so:

```
void Synth::render(float** outputBuffers, int sampleCount)
{
    float* outputBufferLeft = outputBuffers[0];
    float* outputBufferRight = outputBuffers[1];

    // 1
    for (int sample = 0; sample < sampleCount; ++sample) {
        // 2
        float noise = noiseGen.nextValue();

        // 3
        float output = 0.0f;
        if (voice.note > 0) {
            output = noise * (voice.velocity / 127.0f) * 0.5f;  // 4
        }

        // 5
        outputBufferLeft[sample] = output;
        if (outputBufferRight != nullptr) {
            outputBufferRight[sample] = output;
        }
    }
}
```

How this works:

1. Loop through the samples in the buffer one-by-one. `sampleCount` is the number of samples you need to render. If there were MIDI messages, this will be less than the total number of samples in the block.
2. Grab the next output from the noise generator.
3. First check if `voice.note` is not 0, meaning a key is pressed. In other words, the synth received a Note On message but did not receive a Note Off message for this key yet. Then calculate the new sample value.

4. The noise value is multiplied by the velocity. Since the velocity is a number between 0 and 127, you need to divide it by 127 otherwise the output is way too loud (guess who forgot to do this when he was writing this chapter!).

   Also multiply by 0.5 to make the gain a bit more reasonable since noise isn't particularly pleasant to listen to. Multiplying the output by 0.5 means making a 6 dB reduction in the volume level.

5. Write the output value into the audio buffers. As both channels always get the same value, the sound will appear mono, even if the plug-in is on a stereo bus. When the plug-in runs in mono, `outputBufferRight` will be `nullptr` and you only write to the `outputBufferLeft` buffer.

That's it! Run the plug-in to try it out. When you press a key on your MIDI controller, you'll hear white noise coming out of your speakers. Release the key and the noise should stop again.

The noise isn't pitched, so it doesn't matter which key you play, but the loudness should change with the velocity of your keypress. In the virtual keyboard of AudioPluginHost, clicking the top of the key creates less velocity than clicking near the bottom of the key.

Congrats, you've made a synth! It's not a very impressive synthesizer yet but it accepts MIDI input and it generates audio output — everything else is a matter a refinement. In the next chapter you'll add the oscillators that will create the sawtooth waveform. You'll be making some real music soon!

Protect your ears at all times!

I just mentioned that I had made a mistake and forgot to divide velocity by 127, so instead of being between –1 and +1, my audio output was between –127 and +127. That's the kind of thing that makes you spill your drink, or worse, blow out your eardrums.

When writing audio code, it's way too easy easy to make a small mistake that has disastrous consequences for the sound. Especially screaming feedback is the kind of thing you want to avoid (ask me how I know).

This is why it's a good idea not to prototype any new ideas while wearing headphones. It's better to damage your speakers than your ears. However, with a simple trick you can avoid both. You can write some code that checks that the values that are being written into the `AudioBuffer` are reasonable, and if not, sets the entire `AudioBuffer` to silence.

Use the **Projucer** to add a new header file to the project named **Utils.h**. Add the following code to this file.

```
#pragma once

inline void protectYourEars(float* buffer, int sampleCount)
{
    if (buffer == nullptr) { return; }
    bool firstWarning = true;
    for (int i = 0; i < sampleCount; ++i) {
        float x = buffer[i];
        bool silence = false;
        if (std::isnan(x)) {
            DBG("!!! WARNING: nan detected in audio buffer, silencing !!!");
            silence = true;
        } else if (std::isinf(x)) {
            DBG("!!! WARNING: inf detected in audio buffer, silencing !!!");
            silence = true;
        } else if (x < -2.0f || x > 2.0f) {  // screaming feedback
            DBG("!!! WARNING: sample out of range, silencing !!!");
            silence = true;
        } else if (x < -1.0f) {
            if (firstWarning) {
                DBG("!!! WARNING: sample out of range, clamping !!!");
                firstWarning = false;
            }
            buffer[i] = -1.0f;
        } else if (x > 1.0f) {
            if (firstWarning) {
                DBG("!!! WARNING: sample out of range, clamping !!!");
                firstWarning = false;
            }
            buffer[i] = 1.0f;
        }
        if (silence) {
            memset(buffer, 0, sampleCount * sizeof(float));
            return;
        }
    }
}
```

Add an include for this file in **Synth.cpp**. Then at the end of `render`, add the lines:

```
    protectYourEars(outputBufferLeft, sampleCount);
    protectYourEars(outputBufferRight, sampleCount);
```

Now whenever the sound is way too loud or the buffer contains “bad” values such as `nan` or `inf`, the buffer will be silenced. If the sound is only slightly too loud, it’s hard clipped to –1 or +1 and a warning is printed to the debug console. That warning is useful because it lets you know the sound is going outside the safe limits and your DSP code should be preventing this.

With this function in place, you can rest a bit safer, knowing that hurting your ears is much less likely now. Still, it's a good idea to turn your volume down when trying out new ideas — I often have one hand on the mute button just in case.

In the IDE you can set breakpoints on the lines that print "WARNING: nan detected" and "WARNING: inf detected", so that if you ever receive that message, the debugger immediately stops the plug-in. This allows you to inspect the state of your objects to try and figure out where this `nan` (not a number) or `inf` (infinity) came from.

Both are floating point values that shouldn't occur if all is well, so if they do occur there's a serious issue with your DSP code somewhere. For example, I got into this situation a few times when implementing the filter for JX11 and it was due to certain variables being initialized with junk values instead of zeros.

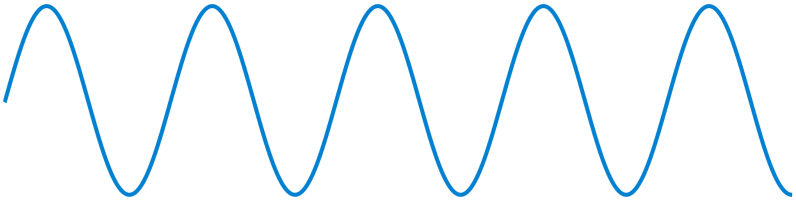
You'll probably want to remove `protectYourEars` from release builds, once you've made sure the DSP code is 100% correct and won't result in blowing someone's eardrums out. When testing your plug-in in a DAW, it's also a good idea to place a limiter at the end of the signal path for the same reason.

Chapter 5: Introduction to oscillators

At this point you have an extremely basic synth that plays white noise when it receives a MIDI Note On event. In this chapter you'll make the synth play actual notes. Making pitched sounds is the job of the oscillator, and JX11 will use not one but two sawtooth oscillators. Let's start simple and learn how to create a sine wave first.

The humble sine wave

I'm sure you're familiar with what a sine wave looks like but here it is again:



The plot of sin(x)

Sine waves are super important in audio programming because all sounds, no matter how complex, can be deconstructed into sine waves. Splitting up a sound into its component sine waves is called Fourier analysis and is done using the FFT (Fast Fourier Transform). The reverse is also true: by adding up different sine waves you can create any possible sound imaginable. This is known as additive synthesis.

Whenever someone says something like, "This sound has a 100 Hz frequency in it," they mean that one of the components that makes up this sound is a sine wave with a frequency of 100 Hz. In this context, a "frequency" really means: a sine wave with that frequency. For example, when applying a low-pass filter set to 5 kHz, what happens is that the filter will remove from the sound all sine waves with frequencies larger than 5000 Hz.

Sine waves are the fundamental building blocks of sound. The waveform of the function `sin(x)` as shown in the picture above, may not look particularly simple but it truly is the most basic sound you can make. Any other waveform, even if it appears to have a simpler shape — such as a square wave or a sawtooth wave — will have a more complex sound than the sine wave.

Since every audio programmer should be able to work with sine waves, that's the first oscillator you will build.

The graph above was made by plotting the mathematical function `y = sin(x)`. Here, `x` is also known as the angle and is measured in radians. For audio programming, we modify this formula a little so that it becomes:

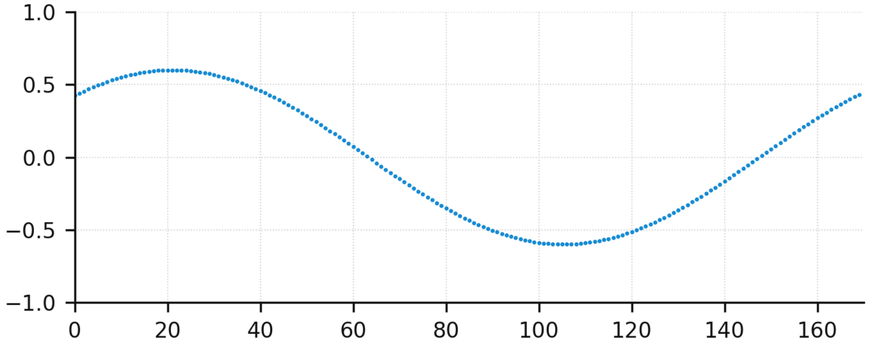
$$y = \text{amplitude} \times \sin(2\pi \times t \times \text{frequency}/\text{sampleRate} + \text{phase})$$

That looks a lot more complicated but it's still the same sine wave, except now its argument is a frequency in Hz, and it produces a digital audio signal consisting of samples.

For example, if you fill in these variables with some real numbers,

$$y = 0.6 \times \sin(2\pi \times t \times 261.63/44100 + 0.785)$$

the resulting sine wave looks like this:



The sampled sine wave

The horizontal axis in this plot is the time axis, measured in samples. In the formula, t is the index of the sample and increases from left to right with the time. Notice how it takes approximately 168 samples to complete one cycle of the sine wave and then it repeats.

There are a few things you need to know about sine waves. They have the following properties:

- **Frequency.** A sine wave has a frequency measured in hertz or Hz. This is how often per second the sine wave repeats. In the example above it is 261.63 Hz. Why that particular number? Well, a sound that vibrates 261.63 times per second has the pitch of the middle C note on the piano, according to the “equal temperament” method of tuning used in Western music.

- **Period.** The inverse of the frequency is the period. This is the length of a single cycle of the sine wave. In the example, the period is 1 / 261.63 Hz = 3.82 milliseconds. Since we're working with sampled sounds, usually we will express the period in number of samples rather than seconds.

  At a sample rate of 44100 Hz, the period of this sine wave is 44100 / 261.63 = 168.56 samples. It might seem odd that this is a fractional number, but dealing with fractional samples is a common situation in audio programming. (Rounding off to a whole number is a bad idea, as that will make the frequency slightly lower or higher, and the instrument will be out of tune.)

- **Amplitude.** The amplitude describes how high the sine wave goes. In the example above, the amplitude is 0.6 because that is the maximum value the sine wave reaches (and –0.6 negative). In audio programming, it's important to not create waveforms that go over 1.0 or below –1.0, as that will horribly mess up the sound.

- **Phase.** The least well-understood part of the sine wave it is phase. This is by how much the sine wave is shifted on the time axis. If the phase of the sine wave is 90 degrees (or π/2 radians), then it's actually a cosine wave. In practice, the phase can be anywhere between 0 and 360 degrees (or 0 and 2π radians). Phase is expressed in degrees or radians because a sine wave is created by going round a circle, and after 360 degrees you're back at the beginning of the sine wave.

  In the image above, the phase is 0.785, which is π/4 radians or 45 degrees, so this waveform is halfway between a sine and cosine. Phase is an essential concept in audio programming, and it occurs anywhere from filtering to spectrum analysis to making special effects, but remember it's nothing more than by how much a particular sine wave is shifted in time.

To make a sine wave oscillator, all you need to do is use the above formula. First you choose the amplitude, the frequency, and the initial phase. Then you keep incrementing the sample index t and fill it into the formula to get the sine value. That's all there is to it — let's write the code!

Building the sine oscillator

In **Projucer**, add a new file to the project and name it **Oscillator.h**. Put the following code into this new file.

```
#pragma once

const float TWO_PI = 6.2831853071795864f;

class Oscillator
{
public:
    float amplitude;
    float freq;
    float sampleRate;
    float phaseOffset;
    int sampleIndex;

    void reset()
    {
        sampleIndex = 0;
    }

    float nextSample()
    {
        float output = amplitude * std::sin(
                        TWO_PI * sampleIndex * freq / sampleRate + phaseOffset);

        sampleIndex += 1;
        return output;
    }
};
```

This is quite literally the mathematical formula for the sine wave as shown in the previous section: $\text{amplitude} \times \sin(2\pi \times t \times \text{frequency}/\text{sampleRate} + \text{phase})$. Instead of `t`, here the time variable is named `sampleIndex`.

In **Voice.h**, include this new source file and add an `Oscillator` instance to the struct. Call `osc.reset()` from the `reset` method, and add a new method `render` that gets the next sample from the oscillator. Like so:

```
#include "Oscillator.h"

struct Voice
{
    int note;
    Oscillator osc;   // this is new

    void reset()
    {
        note = 0;
        osc.reset();   // this is new
    }
    // add this method
    float render()
    {
        return osc.nextSample();
    }
};
```

Also make sure to remove the `velocity` variable from the `Voice` struct, it's no longer needed.

You will need to call the new `render` method from somewhere. Go to **Synth.cpp** and inside its `render` method, change the `if` statement to the following:

```
if (voice.note > 0) {
    output = voice.render();
}
```

Instead of using the output from the noise generator, you now ask the `Voice` object to provide the next value for the sine wave.

Before you can test this out, you have to tell the oscillator what the amplitude, frequency, and phase offset are for the sine wave. Do that in `noteOn`:

```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;

    voice.osc.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc.freq = 261.63f;
    voice.osc.sampleRate = sampleRate;
    voice.osc.phaseOffset = 0.0f;
    voice.osc.reset();
}
```

The amplitude is given by the velocity of the MIDI note. Like before, you multiply by 0.5 to lower the output level by 6 dB, just to make sure it's not annoyingly loud.

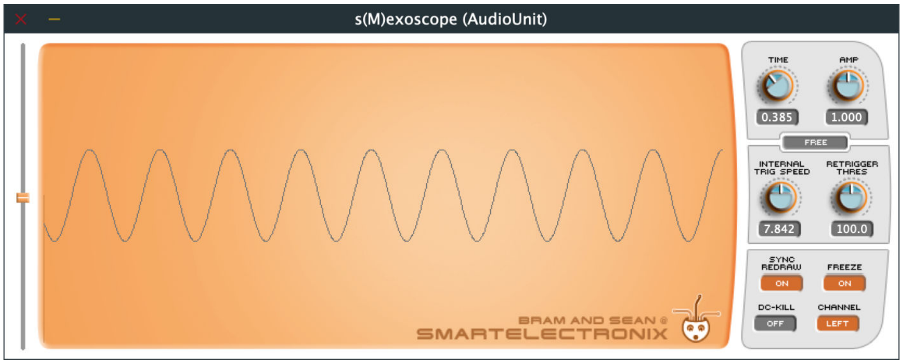
The frequency of the sine wave is fixed to 261.63 Hz, which is the pitch of middle C on the piano. Soon you'll set the frequency based on the MIDI note that was pressed, but for now we'll keep it simple and focus on getting the oscillator to work.

You need to tell the oscillator what the current sample rate is, so that it can calculate after how many samples long one cycle of the sine wave is for the given frequency. Fortunately, `Synth` already knows what the sample rate is, since `JX11AudioProcessor` told the synth about this when it called `allocateResources`.

Since you now use the velocity to set the amplitude of the oscillator, `Voice` no longer needs to have its own `velocity` variable. Also remove the line `voice.velocity = 0` in `Synth::noteOff`.

Try it out! Build the plug-in and open it in AudioPluginHost or your DAW. Play any note and you'll hear the soothing sound of a sine wave at 261.63 Hz.

In a previous chapter I recommended using an oscilloscope and frequency analyzer to help with debugging the plug-in and understanding what's going on. I'm using s(M)exoscope[39] as the oscilloscope and it shows the following:

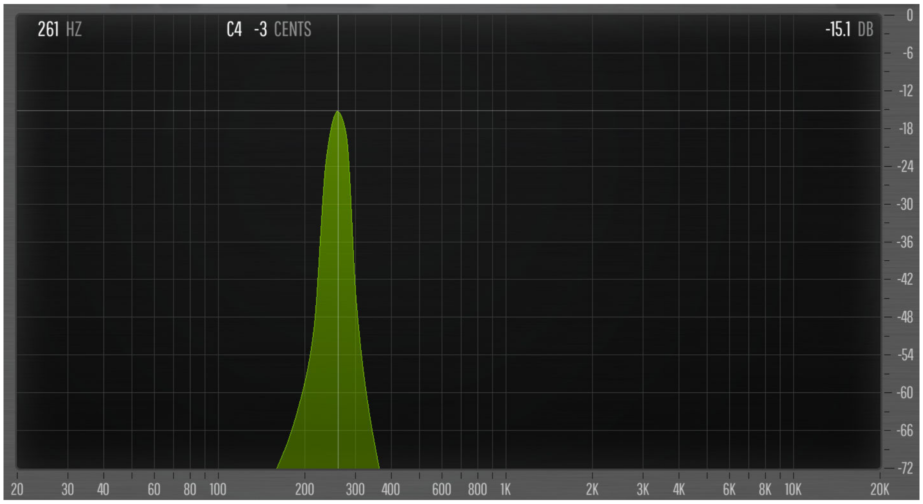

s(M)exoscope (AudioUnit)

TIME 0.385
AMP 1.000
FREE
INTERNAL TRIG SPEED 7.842
RETRIGGER THRES 100.0
SYNC REDRAW ON
FREEZE ON
DC-KILL OFF
CHANNEL LEFT

The oscilloscope shows a sine wave

That looks like a sine wave indeed! You may need to fiddle with the time knob to "zoom in" on the signal before it shows the sine wave, but this is what it should look like.

In the frequency analyzer[40], there should be a single peak around 261.63 Hz. Here's the proof that a sine wave describes only a single frequency:

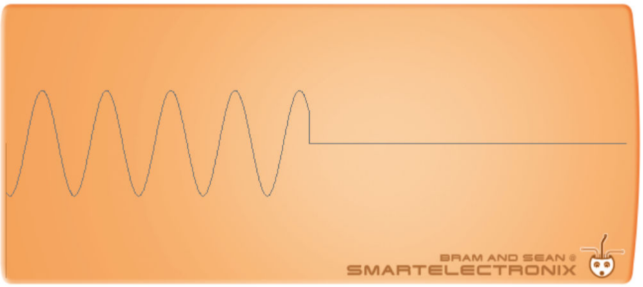

261 HZ
C4 -3 CENTS
-15.1 DB

The sine wave in the frequency analyzer

[39] http://armandomontanez.com/smexoscope/
[40] https://www.voxengo.com/product/span/

> **Note:** Maybe you're wondering why the peak is so wide — in the above image it covers the area between approximately 200 Hz and 350 Hz. This is a side effect of how these frequency analyzers work. For higher notes, the peak will get narrower. The lower the note, the fatter the peak becomes. For our purposes, the important part is the top of the peak, and here it's centered nicely on 261 Hz.

One thing that's not ideal is that you may hear a clicking or popping sound whenever the note stops. This is because the last output from the oscillator is usually a value other than zero, which creates a jump in the signal. Such jumps manifest as audible clicks or pops. You might be able to see these jumps in the oscilloscope:



The sound stops abruptly, causing an audible glitch

You will fix this issue in chapter 8 by adding the envelope to the sound. In audio programming we usually want signals to change smoothly, not abruptly. One way to hide these discontinuities is to fade them in and out, which is exactly what an envelope does. But for now, you'll have to live with these clicks and pops. They won't hurt your speakers, they're just not pleasant to listen to.

Feel free to experiment with different frequencies. Humans can hear frequencies between 20 and 20000 Hz, so try setting `voice.osc.freq` to a few different values in this range. Also try something low like 10 Hz and see if you can hear it. Pump up the volume on a big set of speakers and you might feel rather than hear such a low tone.

It might be fun (or depressing if you're old) to find out what the highest frequency is that you can hear. However, you will find that the oscillator starts acting weird for high frequencies such as 15000 Hz. It no longer sounds like a regular tone but more like noise. That isn't a problem with your ears but a limitation of how the `std::sin()` function is implemented. You'll fix this in the next section.

If you can't hear this high, you can still see in the frequency analyzer that these very high pitches start to become noisy, because additional peaks will appear in the spectrum. The higher the frequency, the worse it gets. This is another good reason for testing your audio code using a frequency analyzer: it shows things that you may not be able to hear — but other people might!

Try experimenting by changing the `voice.osc.phaseOffset` value. The phase offset is usually a number between 0 and 6.28 (or 2π), although it can be larger or even negative. You'll find that changing the phase offset won't make any difference! That's because human ears can't hear the phase on a static sound. However, that doesn't mean phase is not important. In fact, phase is a key feature in many DSP algorithms such as filtering.

Phase is also important when you have more than one oscillator. If two oscillators are playing a sine wave with the same frequency, with the second oscillator being 180 degrees (or π radians) out of phase with the first oscillator, the two waveforms cancel each other out and the result is silence. You'll encounter this exact same situation with JX11's two sawtooth oscillators in chapter 9.

Improving the sine oscillator

To be honest, no audio programmer would ever implement the sine oscillator the way we just did. It's not very efficient and, as you may have found by experimenting, doesn't work well for high frequencies.

In the formula that you used,

```
amplitude * std::sin(TWO_PI * sampleIndex * freq / sampleRate + phaseOffset)
```

the only thing that changes over time is the `sampleIndex` variable, all the other values are fixed. You can simplify the math by combining the parts that don't change into a single value. It will make the code simpler, faster, and better behaved.

The expression `TWO_PI * sampleIndex * freq / sampleRate` converts the frequency in Hz given by `freq`, into an angle in radians, making it possible to call `std::sin` to calculate the sine value. But `std::sin` only works well if the angle is somewhere between –2π and +2π radians.

In our current implementation, `sampleIndex` is continuously being incremented — 44100 times per second! — and will quickly become very large. As a result, the angle becomes too large for `std::sin` to handle and it starts outputting inaccurate results. It's better to restrict the angle to a smaller range.

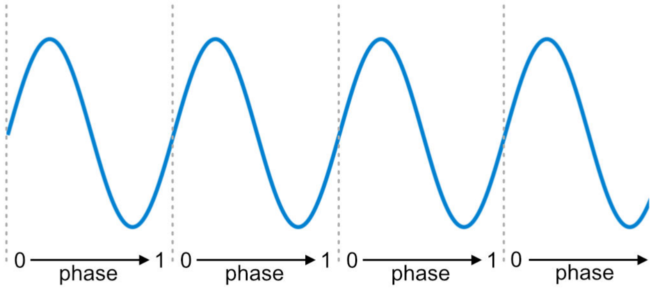
In **Oscillator.h**, change the definition of the `Oscillator` class to the following.

```
class Oscillator
{
public:
    float amplitude;
    float inc;
    float phase;

    void reset()
    {
        phase = 0.0f;
    }

    float nextSample()
    {
        phase += inc;
        if (phase >= 1.0f) {
            phase -= 1.0f;
        }

        return amplitude * std::sin(TWO_PI * phase);
    }
};
```

The `freq`, `sampleRate`, `phaseOffset`, and `sampleIndex` variables have been replaced by two new variables: `inc` and `phase`.

The new `phase` variable is nothing more than a floating-point number that counts from 0 to 1 in small steps, and then wraps back around to 0. This is also known as a modulo counter or a phasor. How quickly `phase` counts up is determined by the value of `inc`, the increment. The larger the desired frequency of the sine wave, the larger `inc` will be, and the bigger steps `phase` takes.

What this means is that `freq` times per second, the `phase` variable counts from 0 to 1. Or put another way, it takes `sampleRate/freq` samples for `phase` to count from 0 to 1.



For every cycle of the sine wave, the phase goes from 0 to 1

The `std::sin` function takes an argument in radians. To render one cycle of the sine wave, the angle should go from 0 to 2π, which is the same as 0 to 360 degrees. After 360 degrees, we're back at the beginning of the sine wave and the cycle repeats. This is why you multiply `phase` by `TWO_PI` before calling `std::sin`. Now the oscillator will output `freq` complete cycles of a sine wave per second, which to our ears sounds like a `freq` Hz tone.

Go back to **Synth.cpp** and change `noteOn` to the following:

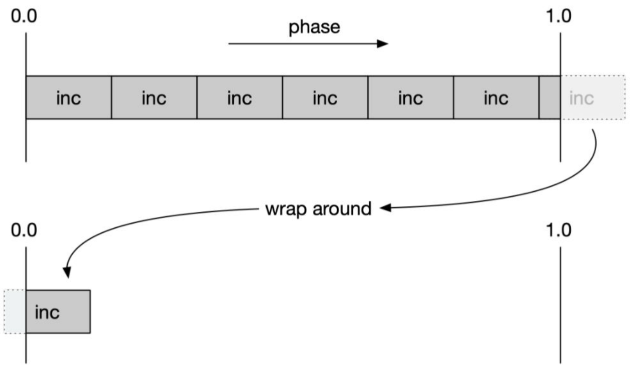
```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;

    float freq = 261.63f;

    voice.osc.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc.inc = freq / sampleRate;
    voice.osc.reset();
}
```

Try it out. The synth should still produce the same sound as before. This time, however, if you change `freq` to a high pitch, such as 14000 Hz or higher, the sound stays a clean sine wave and doesn't turn into noise. How high can you — or your dog! — hear now? If you're young you might be able to hear all the way up to 20 kHz.

Before you continue, a few more words about the `phase` variable. Note that you don't do this to reset the phase counter:

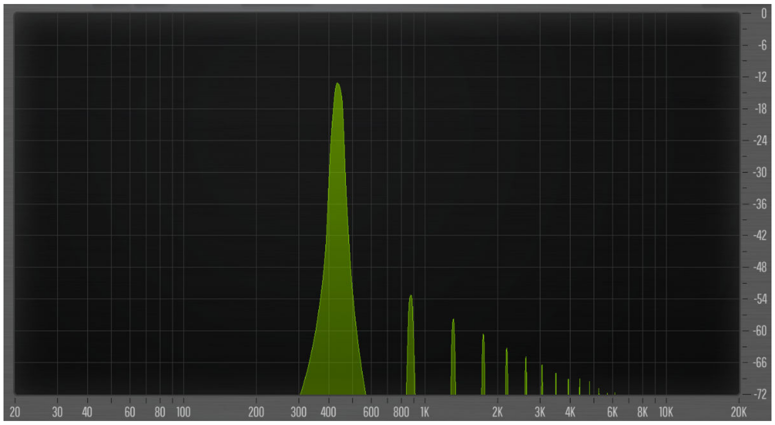
```
        phase += inc;
        if (phase >= 1.0f) {
            phase = 0.0f;      // wrong!
        }
```

When `phase` goes over 1.0, it should not be reset to 0, but it keeps the "extra bit" that it went too far:



The phase wraps around

If you always set `phase` back to 0, you're not using the correct number of samples for each cycle of the sine wave, and different cycles will have different lengths. In the oscilloscope the waveform will still mostly look like a sine, but in the frequency analyzer it's clear that there are now additional peaks. (If you don't see these extra peaks, make sure the analyzer's vertical axis goes down to at least –60 dB.)



What happens when the period is not exact

That waveform no longer is a clean sine wave. I mention this because it's quite easy to make a small error in these kinds of calculations. The tone still sounds all right (try it out) so you might not immediately hear this error, but it's obviously not correct — those extra peaks are an unwanted harmonic distortion. That's why it's a good idea to look at both the oscilloscope and the frequency analyzer to verify that your code is really doing what you expect it to.

In some books or source code, you might see that `phase` counts up to `TWO_PI` instead of `1.0`, like so:

```
phase += inc;
if (phase >= TWO_PI) {
    phase -= TWO_PI;
}
return amplitude * std::sin(phase);
```

Or it might count to `PI` and then subtract `TWO_PI`, so that phase goes from –π to +π instead of 0 to 2π. Doing it that way also works fine, but for many other oscillators, such as the sawtooth oscillator you'll create in the next chapter, it makes more sense to count from 0 to 1. By the way, if you're going to count to 2π, the calculation of `inc` should include this factor 2π as well: `osc.inc = TWO_PI * freq / sampleRate`.

> **Note:** The JUCE DSP module has an `Oscillator` class that basically does the same thing as the simple class you've just written. It manages the phase counter for you, which in JUCE goes from –π to +π. All you have to do is provide your own function or lambda that takes this phase value and uses it to compute the next waveform value. You can certainly use this `juce::dsp::Oscillator` class to build your own oscillators, but as you've seen in this chapter, it's pretty simple to manage this logic yourself.

What happens at the Nyquist limit?

In the previous chapter I mentioned that there is a maximum frequency that can be in a signal, the so-called Nyquist limit. This is always half the sample rate. If the sample rate in AudioPluginHost or your DAW is set to 44100 Hz, then the Nyquist limit is 22050 Hz and higher frequencies are not possible. If the sample rate is set to 48000 Hz, the Nyquist limit is 24000 Hz.

In **Synth.cpp** in `noteOn`, set `freq = 22050` or `freq = 24000`, depending on the sample rate you're using. What happens now? You won't hear anything and there is nothing appearing in the oscilloscope or frequency analyzer.

At the Nyquist limit there are only 2 samples per period of the waveform, and the sine wave goes through zero at exactly these two sample points. As a result, the entire signal is always zero. A sine wave with a frequency that is the same as the Nyquist limit therefore isn't audible — at least not if the phase is zero.

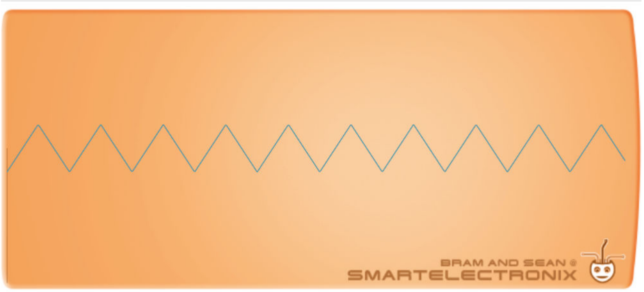
Let's shift the sine wave so that it becomes a cosine wave. Do this by changing the initial value of the `phase` variable in **Oscillator.h**:

```
void reset()
{
    phase = 1.5707963268f;
}
```

This gives the sine wave a 90-degree (or π/2 radians) shift, making it a cosine wave. Earlier you used a separate `phaseOffset` variable for doing this time shift. In this improved version of the oscillator, that initial phase offset has been combined with the `phase` variable. Instead of starting the count from 0 degrees, you now start it from 90 degrees, which in radians is `2π * 90 / 360 = 1.5707963268` radians or π/2.

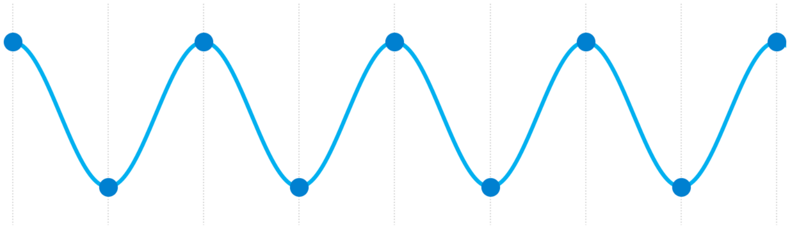
> **Note:** Sometimes in audio programming we use the same word but with slightly different meanings. The "phase of a sine wave" refers to the amount by which the sine wave is shifted. In other words, the initial phase offset. In the oscillator code, the `phase` variable keeps track of where we are in the sine wave. This is also known as the instantaneous phase. Both are really the same thing, as we can get any time shift we want by starting the `phase` from a value that is not 0. Another term for this is the angle of the sine wave, but we don't use that term so much in audio programming as it has more of a geometric interpretation.

Try it out now. You'll still not be able to hear this super high frequency, but in the oscilloscope you'll see the following after zooming in:



What the cosine wave looks at the Nyquist frequency

Interestingly enough, this shows up as a triangle wave instead of a cosine. That happens because there are only two samples in every cycle of the cosine wave. The oscilloscope plug-in connects these samples by straight lines because drawing the true waveform requires slow calculations. Recall that we can always reconstruct exactly what the continuous signal looks like from a set of samples, and if we do this here the result truly is a cosine:



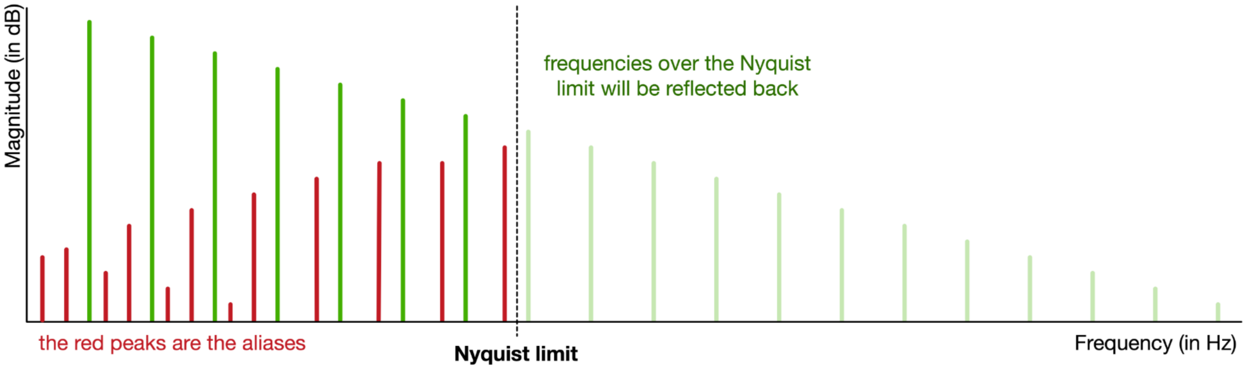
The only possible signal through these points is a cosine

The reason for doing this exercise is to prove that indeed the highest possible frequency a digital signal can have is the sample rate divided by two. That makes sense because it takes at least 2 samples to describe one period of a waveform — you can't describe up-and-down movement with only one sample. That's why the Nyquist limit is `sampleRate / 2`.

There's something else I want to show you. What if you use a frequency that's higher than 22050 or 24000 Hz? Try it out: set `freq = 44000` and see what happens. Don't worry, nothing will blow up.

Instead of getting an extremely high tone that only bats can hear, the synth will play a much lower tone. If the sample rate is 44100 Hz, you'll hear a 100 Hz tone. At a sample rate of 48 kHz, you'll get a 4000 Hz tone. What you're hearing is called an alias.

Frequencies that are higher than the Nyquist limit, will be reflected in the Nyquist limit as lower frequencies. These lower frequencies are "aliases" of the frequencies that are too high to be described by these samples.



Where aliases come from

For example, if the Nyquist limit is 24000 (sample rate 48 kHz), then playing a 25000 Hz tone will actually give a 23000 Hz tone. The original frequency is 1000 Hz higher than the Nyquist limit. This frequency cannot be represented by a 48 kHz sampled signal since that would require less than two samples per period, and so it will be reflected as a signal that *can* be represented, 1000 Hz lower than the Nyquist limit.

This limit and the fact that aliases exist, is one of the fundamental issues when dealing with digital audio. It's not just some theoretical oddity. In fact, you'll run smack-bang into these aliases in the next chapter where you're going to make the sawtooth waveform.

Now, I know there's some mad scientist among my readers who is wondering what happens if you make `freq` a *negative* number. Go on, try it out. Negative frequencies are possible and you'll more than likely run into them in your audio programming journey.

Just like frequencies that are too high get reflected in Nyquist, frequencies that are less than 0 Hz get reflected around zero. Setting `freq = -100` simply results in a 100 Hz tone. As a side effect, it shifts the phase by 180 degrees, which means this gives an upside-down sine wave.

Play a melody!

There's not a lot of music that only uses middle C, so let's make the synth able to play other notes as well. It only requires you to change a single line of code, but to understand how this works there is a bit more math involved. Take a deep breath and here we go!

In **Synth.cpp**, change `noteOn` to the following:

```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;

    float freq = 440.0f * std::exp2(float(note - 69) / 12.0f);  // this changed

    voice.osc.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc.inc = freq / sampleRate;
    voice.osc.reset();
}
```

Most of the code is unchanged, except for the line that calculates the frequency. Rather than using a fixed number such as 261.63, this has been replaced with the following formula:

$$\text{freq} = 440 \times 2^{(\text{note}-69)/12}$$

The `std::exp2` function computes 2-to-the-power-of.

If you're not very strong at math this formula may look intimidating but it's not so bad once we pick it apart.

The goal here is to turn a MIDI note number into a pitch — that is, into the fundamental frequency of the note we wish to play. As you've seen in the MIDI chapter, there are 128 note numbers. Number 60 is middle C.

Western musical instrument tuning is based on the frequency of the A above middle C. This note has MIDI number 69 and a frequency of 440 Hz. That's where the number 440 at the front of the formula comes from. All the other notes have a pitch that is relative to this A note.

In this system of twelve-tone equal temperament tuning, the difference between the frequencies of two neighboring notes is $2^{1/12}$. Since A is 440 Hz, the note immediately above it is $440 \times 2^{1/12} = 466.16$ Hz. And the note two semitones above A is $440 \times 2^{1/12} \times 2^{1/12} = 493.88$ Hz. You can also write this as $440 \times 2^{2/12}$ Hz. The note three semitones higher is $440 \times 2^{3/12} = 523.25$ Hz, and so on.

Where does this magic number $2^{1/12}$ come from? For two notes that are an octave apart, the frequency doubles. The A note an octave higher is 880 Hz, while the A an octave below is 220 Hz. The *two-to-the-power-of* describes this doubling with each octave. Since an octave has twelve notes or semitones, the doubling of the frequency is split up into 12 equal parts, making the distance between any two semitones $2^{1/12}$.

The distance between multiple semitones is then $2^{N/12}$. For example, if N = 12, meaning the note one octave up, we get a multiplier of $2^{12/12} = 2$, which indeed doubles the frequency. And if N = 24, meaning two octaves up, this gives the multiplier $2^{24/12} = 4$, which quadruples the original frequency.

As a software developer you're probably used to working in powers of two (256, 512, 1024 and so on). The same is true for music when it comes to the pitches of notes.

It also works the other way: The note immediately below the A is $440 \times 2^{-1/12} = 415.30$ Hz. The note below that has a frequency of $440 \times 2^{-2/12} = 392.00$ Hz, and so on. Middle C is nine semitones below A, which is $440 \times 2^{-9/12} = 261.63$ Hz as expected. You can verify these numbers are correct by looking up the MIDI note chart in chapter 3.

To summarize: Given a certain note, here the A 440 Hz, to find the frequency of the note N semitones higher, you multiply by $2^{N/12}$. To find the frequency of the note N semitones lower, you multiply by $2^{-N/12}$.

This works for any starting note, by the way. To find the frequency of the B below middle C, you can write $261.63 \times 2^{-1/12}$ or you can do $440 \times 2^{-10/12}$. Both formulas give the same result: 246.94 Hz. The reason we use this particular A note as the starting point is merely a convention.

Hopefully the formula $\text{freq} = 440 \times 2^{(\text{note}-69)/12}$ makes sense now:

1. First you do $(\text{note} - 69)$ to get the number of semitones this note is up or down from the A with note number 69.
2. Then you calculate $2^{(\text{note}-69)/12}$ to get the multiplier.
3. Finally, apply this multiplier to the frequency of that A note, 440 Hz, to get the pitch of the note you're looking for.

Try it out. You should now be able to play all your favorite melodies!

In the next chapters you'll add different tuning options and write code to allow playing more than one note at a time, but at least for now you can use the synth to actually play a tune.

Using a digital resonator

Is the current version of Oscillator the best we can do? Sine waves are a big deal in audio programming, so it's no surprise people have come up with faster ways to create them. The problem is that std::sin is relatively expensive to compute and using a good-enough approximation gives the same result but faster.

One of these approximations is the so-called direct-form digital resonator. This isn't a general-purpose replacement for the std::sin function, since you can't call it with arbitrary values such as std::sin(3.14). It's an algorithm that, once you set it going, will iteratively produce a sine wave sample-by-sample.

In **Oscillator.h**, add three private variables to the class:

```
private:
    float sin0;
    float sin1;
    float dsin;
```

Add the following lines to the bottom of reset:

```
sin0 = amplitude * std::sin(phase * TWO_PI);
sin1 = amplitude * std::sin((phase - inc) * TWO_PI);
dsin = 2.0f * std::cos(inc * TWO_PI);
```

Replace nextSample with the following:

```
float nextSample()
{
    float sinx = dsin * sin0 - sin1;
    sin1 = sin0;
    sin0 = sinx;
    return sinx;
}
```

It doesn't get much simpler than that. Instead of calling the expensive std::sin function for every sample, all you do now is some simple arithmetic.

The important part is that the variables are initialized properly in reset. This does call std::sin twice and std::cos once, but that is done only the one time when the note starts. From then on, the algorithm automatically keeps outputting a sine wave at the desired frequency.

Try it out! This direct oscillator algorithm gives very accurate results even at high frequencies. Over time it might drift away from the true sine shape somewhat, but this takes many seconds before it becomes noticeable. Most notes are much shorter than that, and so this drift won't be a problem in practice.

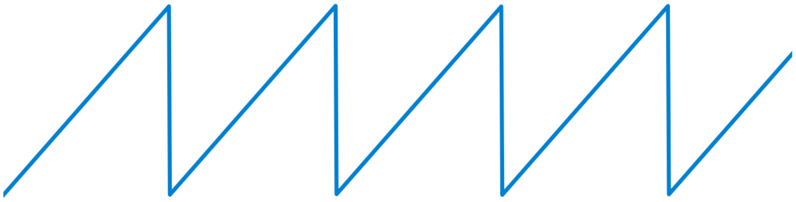
I'm not going to explain in detail how this algorithm works because that involves a lot of background information about filters and the z-plane that we don't have room to cover in this book. For the time being, file it away as a clever method for making a sine wave oscillator.

All right, enough of those sine waves. In the next chapter you'll learn how to make more interesting sounds!

Chapter 6: The sawtooth oscillator

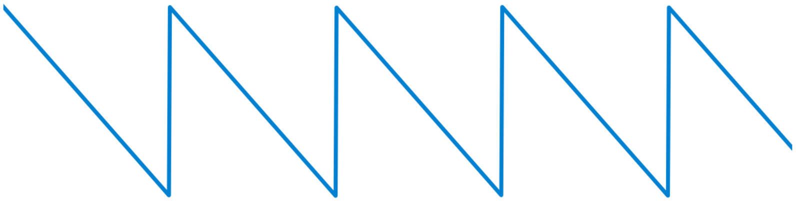
One of the most common oscillators in analog synths is the sawtooth waveform. As you'll see shortly, whereas the sine wave is made up of only a single frequency (yawn), the sawtooth has loads (yay!).

As a quick reminder, the sawtooth waveform looks like this:



The sawtooth waveform

The sawtooth is a straight line that consistently rises over the duration of one cycle or period. On the beginning of the next cycle, it sharply drops and then starts ramping up again. It's also possible to do it the other way around, where the value always falls rather than rises. You can consider this a 180-degree phase-shifted version, which is a fancy way of saying it has been flipped upside down.



Also a sawtooth waveform

It would seem that this oscillator is extremely easy to make. Since you're already counting up the `phase` variable from 0 to 1 on each cycle and then wrapping it around, you can just output the `phase` value!

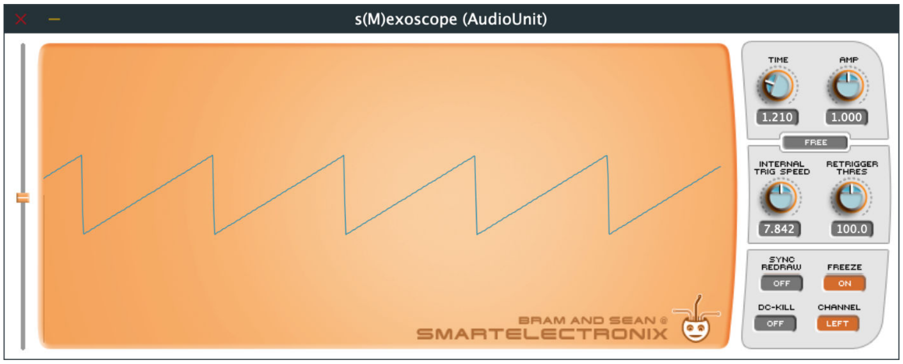
Of course, sample values are supposed to be between –1 and +1, so you'll use the formula `2*phase - 1` instead. Finally, you multiply this number by the desired amplitude.

Change the code in **Oscillator.h** to the following:

```
class Oscillator
{
public:
    float amplitude;
    float inc;
    float phase;

    void reset()
    {
        phase = 0.0f;
    }

    float nextSample()
    {
        phase += inc;
        if (phase >= 1.0f) {
            phase -= 1.0f;
        }
        return amplitude * (2.0f * phase - 1.0f);
    }
};
```

That's all you need to do! That was easy enough… Build the plug-in and try it out. You should hear the wonderful old school sound of a sawtooth wave now.

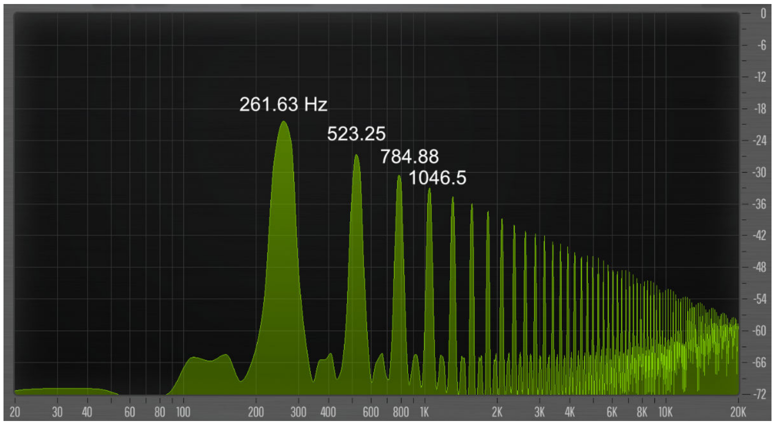
It looks like this in the oscilloscope:


s(M)exoscope (AudioUnit)

TIME 1.210
AMP 1.000
FREE
INTERNAL TRIG SPEED 7.842
RETRIGGER THRES 100.0
SYNC REDRAW OFF
FREEZE ON
DC-KILL OFF
CHANNEL LEFT

The sawtooth oscillator in the oscilloscope

By looking at the frequency analyzer, it's clear there are a lot more frequencies in the signal now! Where there was only a single peak for the sine wave, here there are many. The leftmost peak is still the same. This is the lowest frequency in the sound and is known as the fundamental frequency. Here is it 261.63 Hz, the middle C note.



The sawtooth oscillator in the frequency analyzer

Note: Often we will use the term pitch when we mean the fundamental frequency. We might say the pitch of middle C is 261.63 Hz, but it's really the frequency of that first peak we're referring to. Using the word "pitch" is not 100% technically correct because pitch is a psychoacoustic phenomenon and it's possible to hear a pitch even when the tone's fundamental frequency is completely missing. But this is a book about synthesizers, not psychology, and for our purposes when we refer to the pitch of a sound, it means the fundamental frequency.

The other frequencies that are in the sawtooth waveform are the harmonics. These harmonics exist at multiples of the fundamental frequency. In the frequency spectrum shown by the analyzer above, you can see that the first four peaks are at 261.63 Hz, 523.25 Hz (= 2× the fundamental), 784.88 Hz (3×), and 1046.5 Hz (4×).

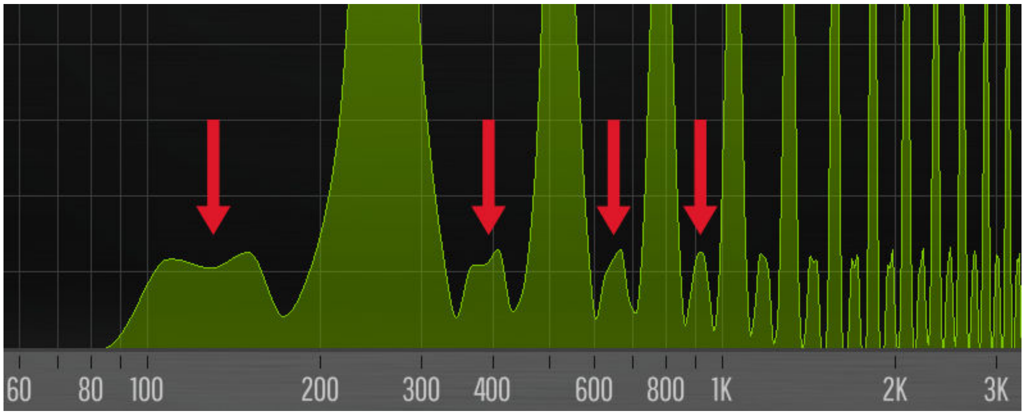
For a sawtooth wave, all the possible harmonics are present. For other types of waveforms, such as a square wave, only the harmonics that are even multiples of the fundamental (2×, 4×, 6×, …) are present and the frequencies for the odd harmonics are silent.

The timbre of a sound is determined by which harmonics it has, and what their heights or magnitudes are relative to the fundamental frequency (also known as the first harmonic).

Just as importantly, the timbre is affected by how the frequency spectrum changes over time. The sound coming out of the oscillator is static at the moment, but in later chapters you'll add new features such as a resonant filter that can be modulated. This allows the synth to gradually change the magnitudes of these harmonics, which is what gives the sound its character.

Doctor, I think I have aliases

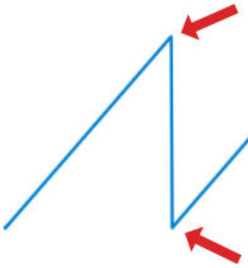
However… there is a problem that you may have noticed if you looked closely at the frequency analyzer. Besides the harmonics, which we actually expect to be present, there's a lot of other stuff in the frequency spectrum that isn't supposed to be there:



What are these things?

These extra frequencies that show up in between the peaks for the harmonics are the dreaded aliases! Also notice in the frequency analyzer that the shapes of these aliases change when you play different notes. What exactly is going on here?

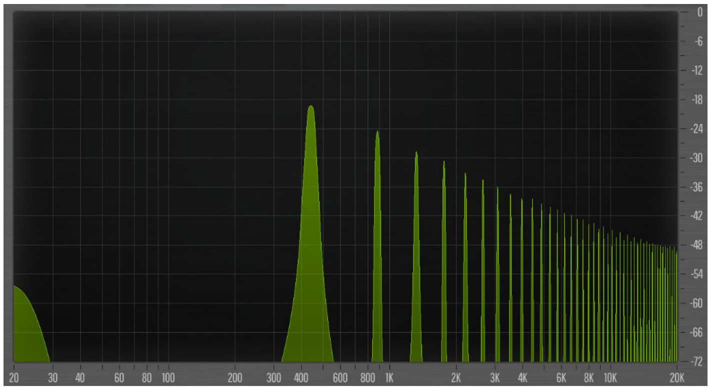
When you create the sawtooth waveform like you just did, which is often referred to as a "naive" sawtooth, it has a sharp corner in it where the output jumps back from +1 to –1 to start the next cycle:



The naive sawtooth has sharp transitions

Remember how I said that all signals can be considered to be made up by adding multiple sine waves together? Well, it takes a *lot* of sine waves to make sharp corners like this, infinitely many in fact, and most of these sine waves will have very high frequencies that easily go over the Nyquist limit. Those frequencies will be mirrored back as lower frequencies, creating aliases.

This isn't always horrible. Try setting `freq = 441.0` (sample rate 44.1 kHz) or `freq = 480.0` (sample rate 48 kHz) in `Synth::noteOn`. The aliases will appear here too, but in this case the reflected frequencies will fall exactly on the original harmonics, as a 441 Hz sound fits an integer number of times in a 44.1 kHz signal, and likewise for 480 Hz in a 48 kHz signal. The spectrum looks clean, as the aliases always coincide with the peaks from the harmonics.



The spectrum without any aliases

Notice how there is nothing in between the peaks now. That's what we want to see. It happens to work out well for 441 Hz or 480 Hz, but for most other pitches that you'd want to use, things don't line up so nicely and the reflected harmonics show up as unwanted frequencies in the sound. In the frequency spectrum this appears as "muck" between the harmonics.

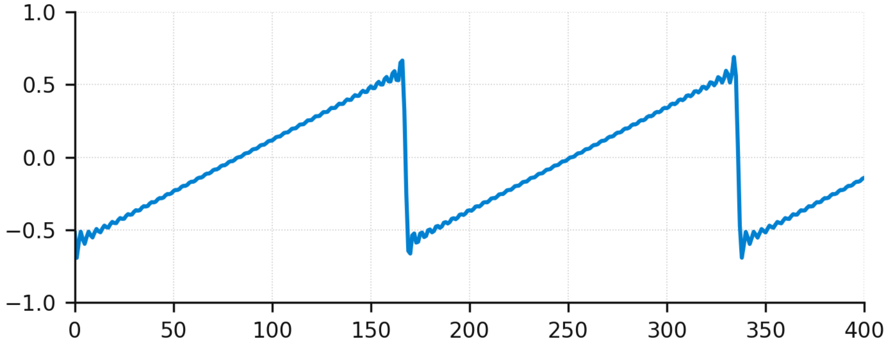
> **Note:** Wondering about that blob on the far left of the spectrum? The distance between harmonics is 441 Hz, and since the aliases fall exactly on the harmonics, there is an alias frequency every 441 Hz. The blob happens because some of the aliases will end up at 0 Hz. You can't directly hear a 0 Hz frequency, but it does affect the sound by shifting it up or down, giving it a so-called DC offset.

In summary: Any frequencies that go over the Nyquist limit and that are not multiples of the fundamental frequency will create "inharmonic" sounds. You can certainly create such sounds on purpose, but in the case of aliases you don't have any control over where they will happen — this solely depends on the sample rate. The 441 Hz tone has no aliases when the sample rate is 44.1 kHz but it does have aliases when the sample rate is 48 kHz.

Since you can't control them, and they are different for every note that is played, you definitely want to get rid of those aliases. To prove that aliases are not just a theoretical problem, I'm going to let you hear them!

Making a bandlimited sawtooth

Instead of the naive sawtooth waveform, we want something that looks more like this:



A bandlimited sawtooth

Notice how the corners are no longer sharp but are a little wobbly? This kind of “ringing” is typical for bandlimited waveforms. The term bandlimited means that the frequencies in the signal are restricted to a certain maximum — in this case no frequencies higher than the Nyquist limit.

This bandlimited sawtooth waveform is made by adding up many sine waves:

$$\frac{2}{\pi}\left(\sin(\phi) - \frac{\sin(2\phi)}{2} + \frac{\sin(3\phi)}{3} - \frac{\sin(4\phi)}{4} + \frac{\sin(5\phi)}{5} - \frac{\sin(6\phi)}{6} + \ldots\right)$$

Each of these sine waves adds a new harmonic to the spectrum. In this formula, ϕ represents the fundamental frequency. The second harmonic therefore has frequency 2ϕ, the third harmonic is at 3ϕ, and so on.

The amplitude of these sine waves becomes smaller for higher harmonics, since you divide each term by an increasing number. Also note that to create a sawtooth, the even harmonics are subtracted rather than added. The $\frac{2}{\pi}$ is a scaling factor to keep the values between –1 and +1.

This method of making waveforms is called additive synthesis. Any kind of sound can be created by adding the right sine waves together in the right amounts. Adding up lots of sine waves can be slow, especially if you need hundreds of them, but for the sake of experimentation let’s change the `Oscillator` class to use additive synthesis. In the next section you’ll implement a much faster way to achieve the same result.

Change **Oscillator.h** to the following.

```
class Oscillator
{
public:
    float amplitude;
    float inc;
    float phase;

    // add these lines:
    float freq;
    float sampleRate;
    float phaseBL;

    void reset()
    {
        phase = 0.0f;
        phaseBL = -0.5f;   // important!
    }

    float nextBandlimitedSample()
    {
        phaseBL += inc;
        if (phaseBL >= 1.0f) {
            phaseBL -= 1.0f;
        }

        float output = 0.0f;
        float nyquist = sampleRate / 2.0f;
        float h = freq;
        float i = 1.0f;
        float m = 0.6366197724f;  // this is 2/pi
        while (h < nyquist) {
            output += m * std::sin(TWO_PI * phaseBL * i) / i;
            h += freq;
            i += 1.0f;
            m = -m;
        }
        return output;
    }

    float nextSample()
    {
        return amplitude * nextBandlimitedSample();
    }
};
```

At first sight this may look a bit intimidating, but you’ve already dealt with most of these pieces.

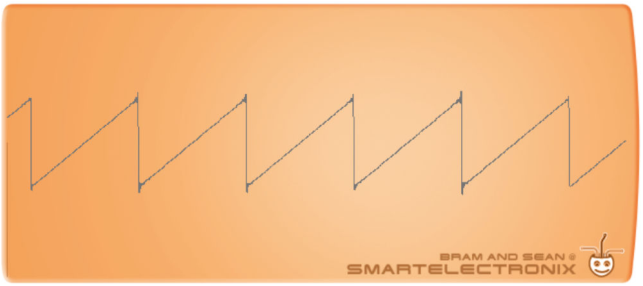
- There is a new phase counter `phaseBL` for the bandlimited oscillator. This works just like `phase` from before except it starts at a different offset of `-0.5`. That's not important right now but it will be for demonstrating the aliases.
- The while loop adds up sine waves until it reaches the Nyquist limit. The sine values are accumulated in the variable `y`.
- The variable `h` starts out at the fundamental frequency `freq`, and in each iteration of the loop this is incremented to get the frequency of the next harmonic. The loop repeats until `h` exceeds Nyquist.
- `i` counts which harmonic we're currently at. As before, you use the phase value as the argument to `std::sin`, but this time multiplied by `i` to get the sine value for this harmonic.
- `m` provides the scaling factor 2/π from the formula, and is also used to flip the sign so that even harmonics are subtracted instead of added.

For a tone at 261.63 Hz and a 44.1 kHz sample rate, this loop adds up 84 sine waves. That is the fundamental frequency plus 83 harmonics. How many harmonics are used depends on the pitch — the lower the tone, the more harmonics it takes to reach the Nyquist limit.

In `Synth::noteOn`, add the following two lines to allow the oscillator to calculate how many harmonics are needed:

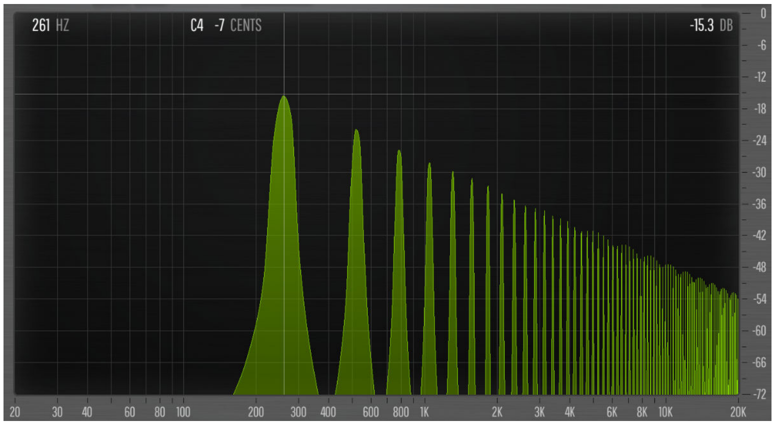
```
voice.osc.freq = freq;
voice.osc.sampleRate = sampleRate;
```

Try it out! You now have a properly bandlimited sawtooth oscillator, albeit a rather inefficient one. It looks like this in the oscilloscope:



The bandlimited sawtooth in the oscilloscope

The spectrum for this bandlimited version of the sawtooth shows no aliases since this waveform does not contain frequencies over the Nyquist limit. There is also nothing at 0 Hz. The spectrum doesn't get any cleaner than this!



The bandlimited sawtooth in the frequency analyzer

You can clearly see that the higher the harmonic, the smaller its magnitude is. This happens because in the formula the amplitude of the sine wave for the n-th harmonic is divided by n. It is this particular arrangement of harmonics that gives the sawtooth waveform its shape.

> **Note:** If you're using Voxengo SPAN as the frequency analyzer and the harmonics do not gradually fall off like in the above picture, open the settings dialog and set the SLOPE control to 0.

Bandlimited vs. aliases

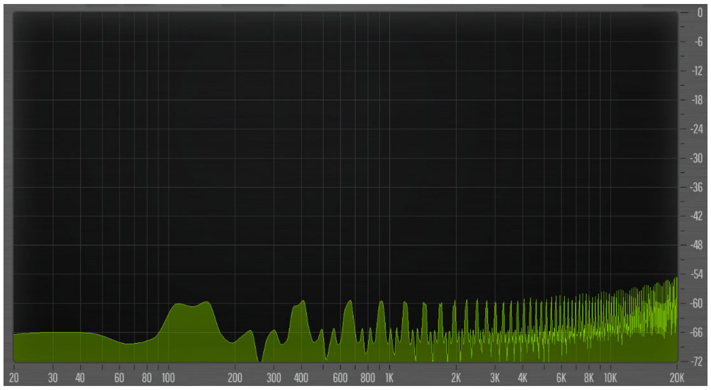
I promised I'd prove that aliases were bad. Change the `Oscillator`'s `nextSample` method to:

```
float nextSample()
{
    phase += inc;
    if (phase >= 1.0f) {
        phase -= 1.0f;
    }
    float aliased = 2.0f * phase - 1.0f;

    return amplitude * (aliased - nextBandlimitedSample());
}
```

This creates the naive sawtooth with aliases like before, and subtracts the clean bandlimited version from that. What remains is only the alias frequencies.

Make sure that `phaseBL` is set to `-0.5` in `reset` and that `osc.reset()` is called from `Synth::noteOn`. This is necessary, so that each new note you play has both waveforms starting from the exact same position.

Try it out! The frequency analyzer now shows only the aliases. You might need to set the range of the frequency analyzer to –96 dB to see all of them, and the FFT size to 4096 or larger to see the small peaks clearly.



The alias frequencies by themselves

If you don't hear anything, you may need to turn up the volume to hear the aliases. All this extra stuff in the frequency spectrum sounds like buzzy, inharmonic crap.

The image above was made using a 261.63 Hz tone at a 44.1 kHz sample rate. I mentioned there were 84 harmonics in the bandlimited version of this sound, including the fundamental. The 85th harmonic would have a frequency of 261.63 × 85 = 22238.55 Hz. That is 188.55 Hz over the Nyquist limit, so this gets reflected as 22050 – 188.55 = 21861.45 Hz. And then all the next harmonics get reflected too, at multiples of 261.63 below that number.

In the spectrum of the aliases, you should see peaks at 931.05 Hz, 669.42 Hz, 407.79 Hz, and 146.16 Hz. And then the next alias gets reflected around 0 Hz again, which shows up as 115.47 Hz. The one after that at 377.10 Hz, and so on. It's a mess. Verify in your frequency analyzer that you indeed have peaks at those frequencies.

Your ears may not have been able to isolate these extra buzzy sounds while the sawtooth was playing, but they're definitely there and they are nasty. Some of the aliases are not going to be audible. It's hard to pick out an alias that is quieter than –60 dB unless it's the only thing playing. And you probably won't hear aliases close to 20 kHz. Still, there are plenty of aliases that remain — by itself, the buzzing is clearly audible!

Also try this with a 441 Hz sound at 44.1 kHz (or 480 Hz at 48 kHz). Since there were no aliases visible in the frequency spectrum for these tones previously, you might expect to hear nothing when subtracting the bandlimited waveform from the naive one. However, recall that the aliases really were there — they just happened to coincide with the real harmonics. These aliases didn't add new frequencies to the sound but they did create harmonic distortion, which you can now hear in isolation.

> **Note:** Another interesting exercise is to use a DAW to first record the naive sawtooth and then record the bandlimited sawtooth. Align these two tracks so that both waveforms are playing at the same time. Pan one track fully to the left and the other fully to the right. Now listen with headphones. With some practice you should be able to hear the extra buzzing noise from the aliases in one ear.

I hope this little demonstration has convinced you that it's easy for aliases to slip into your synthesized sounds, and not for the better. Even something as simple as the sawtooth waveform already creates lots of them. Now that you've seen how *not* to make a sawtooth oscillator, let's do it properly.

BLIT (bandlimited impulse trains)

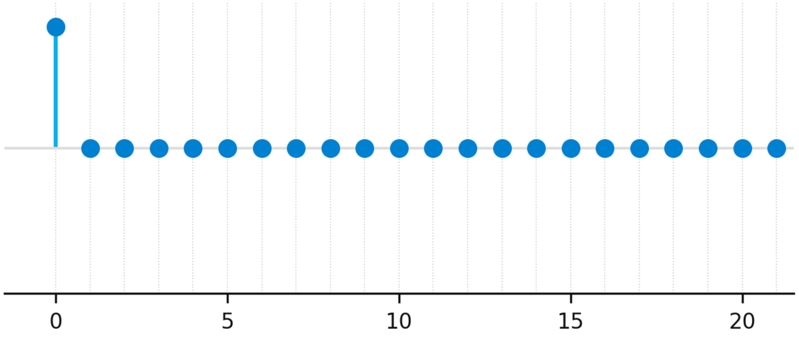
While it's possible to create any type of waveform imaginable by adding up sine waves, the process of calculating many individual sines can be slow, which is why people have been trying to find ways to create bandlimited oscillators using less expensive techniques. Often the resulting oscillators still have some aliases but these may be quiet enough that they are not audible.

Since aliases are frequencies that are reflected around the Nyquist limit, it's OK to have a few small aliases in the higher regions of the spectrum, as they're essentially inaudible there anyway — especially if you also filter out the highest frequencies using the synth's filter stage.

Of all the methods I could have chosen to implement the sawtooth oscillator, I picked an older method named BLIT, which stands for BandLimited Impulse Trains. Even though these days there are better ways to make sawtooth waves, I chose BLIT for two reasons: MDA JX10 used this method too, plus it's instructive to learn what an impulse train actually is, since impulses are a foundational idea in DSP.

Let's pick the acronym apart. The "BL" stands for bandlimited, which means that the signal has no frequencies over the Nyquist limit of 22050 Hz (assuming a sampling rate of 44.1 kHz). So far, so good. The "IT" stands for impulse train, what is that?

An impulse is almost the simplest digital signal imaginable. It is zero everywhere except a single sample has the maximum value, as shown in the following figure.



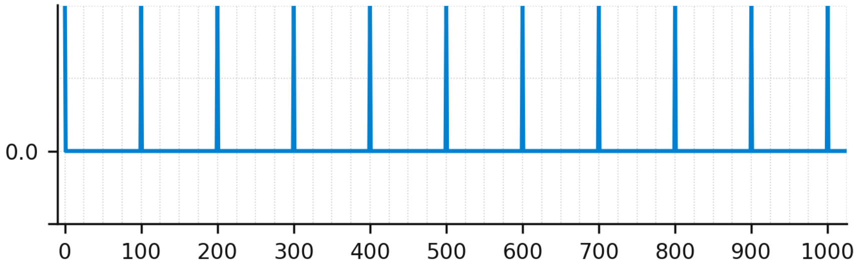
The impulse signal

Don’t let its simplicity fool you: this basic-looking signal contains all possible frequencies. That’s right, this teeny tiny impulse contains all frequencies from 0 Hz up to 22050 Hz. How is this even possible? You know by now that a frequency is just a single sine wave. Well, if you add up all sine waves — or in this case cosines, a sine that has been shifted by 90 degrees — from 0 Hz to 22050 Hz, what you get is the impulse signal. All these cosine waves cancel out each other everywhere, except at one point where they reinforce each other.

This property makes the impulse a very important signal in DSP. Among other things, it is used to measure the frequency response of a system such as a filter. When the impulse signal is sent through a filter, this outputs a new signal known as the impulse response.

We can take the FFT (Fast Fourier Transform) of this impulse response to find the frequency response. The frequency spectrum of the original impulse is flat, as it contains all frequencies in equal measure. The frequency spectrum of the impulse response, however, shows how each of these frequencies has been affected by the filter: some frequencies will be made louder, others may have been made quieter.

For our purposes of making a bandlimited waveform, you will use not use just one impulse but a whole bunch of them. An impulse train is nothing more than repeating impulses spaced out in time. The time between each impulse determines the period of the sound. An impulse train signal looks like this:



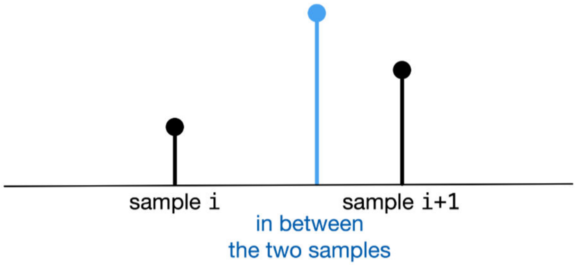
An impulse train consists of repeated impulses

The waveform in the picture has a pitch of 441 Hz because there are exactly 100 samples from one impulse to the next, and 44100 / 100 = 441. It's almost, but not quite, the pitch of the A above middle C. This impulse train outputs a real tone, but it doesn't sound particularly pleasing. Shortly, you'll hear this for yourself. If you have experience with synths that can do PWM, or pulse width modulation, then an impulse train is almost like a pulse wave with an extremely short duty cycle.

In the image above, the impulse train looks simple enough to create: output a value of 1.0 once in every period, and a value of 0.0 for the rest of the period. That works fine indeed, but only if the period is an integer number of samples. This, unfortunately, is not true for most notes you'll want to play — usually the period is some fractional number of samples.

For a period that is a fractional number, the impulse will usually fall somewhere in between two samples. There are a couple of ways to deal with this:

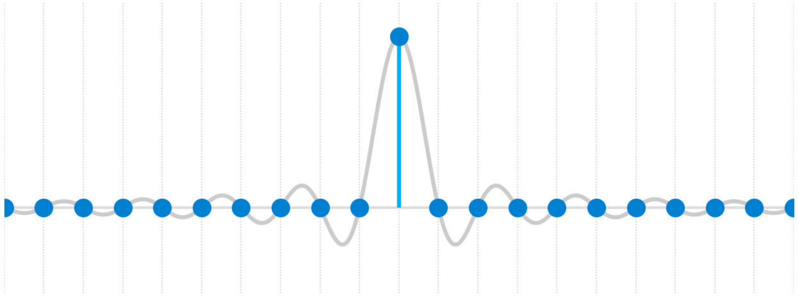
- Round the position of the impulse up or down to the nearest sample. This is the solution that takes the least effort, but it creates a noisy signal with a lot of aliases in it.
- Divide the impulse value between the two nearest samples. For example, if the fractional part is 0.6, the impulse can be divided into a sample of height 0.4 and a sample of height 0.2:

  This is very similar to the technique called linear interpolation. The advantage of linear interpolation is that it's fast, which is why a lot of audio code uses it. The downside is that it isn't accurate at all and it still adds a lot of aliases.

Both these methods are quick but neither gives a satisfactory result.

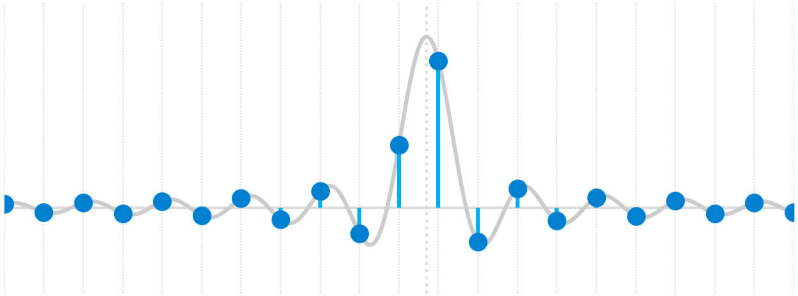
The correct way to create an impulse train is to realize that there's more to an impulse than just a single 1.0 surrounded by zeros. If you would be able to see what happens in between the samples, it looks like the following.



The impulse is really the sinc function

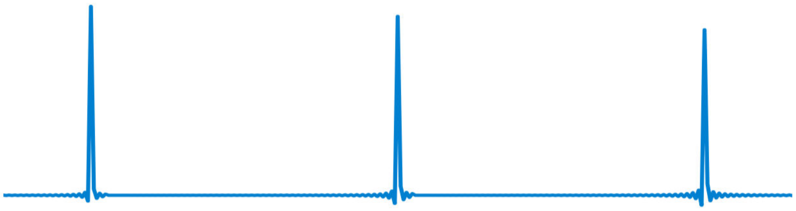
When the impulse is sent through the computer's DAC and out through its speakers, the resulting analog signal is exactly what wobbly line. That is called the "sinc" function and has the formula $\sin(\pi x)/(\pi x)$. Previously I claimed that, given a properly bandlimited signal, we can always predict exactly what happens between two successive samples. That is done using this sinc function.

It just so happens that when the impulse lands exactly on a sample, the sinc function is 1.0 at that sample and it goes through zero everywhere else. However, if the impulse falls in between two samples, the sinc is shifted and now all the surrounding samples have non-zero values:



An impulse that falls between two samples

Long story short, a properly bandlimited impulse train is made up of these sinc shapes. The signal may not look very clean with all those wobbles but remember that you shouldn't just look at the sample values — it's what happens in between the samples that matters!

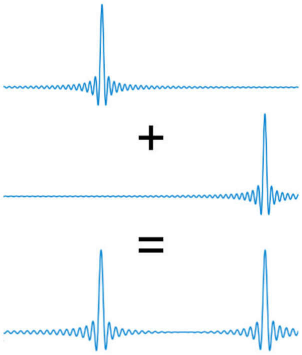


The impulse train made of sinc pulses

Everywhere an impulse falls in between two samples — which is pretty much always — to get the bandlimited version of the impulse, you'll need to "sample" this ideal sinc function. The problem is that doing this 100% correctly is expensive. The sinc function is infinitely long and you obviously can't take an infinite amount of time to fill up the audio buffer.

So instead, you're going to have to fake it. This is true for a lot of audio programming: something might be too time-consuming to compute in real-time, and you'll have to find a smart method to approximate what happens and hope it sounds good.

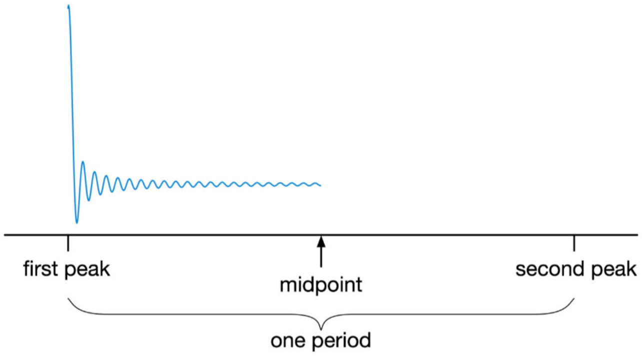
To create a perfect bandlimited impulse train, you'd calculate the full-length sinc function centered on the first peak, then calculate another sinc centered on the second peak, and add them up:



Creating the impulse train by adding up individual sinc pulses

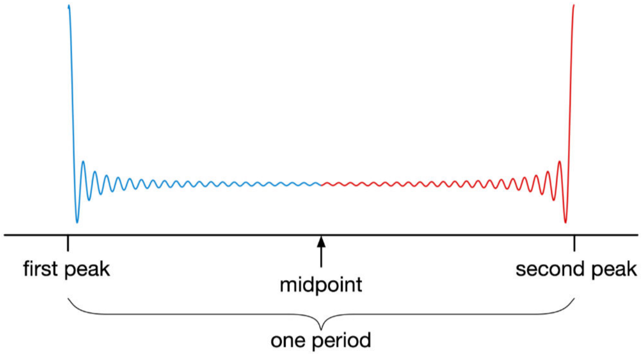
That is the result we want to approximate, but without calculating full sinc functions for all the peaks. Notice in the image that both sincs extend beyond the other peak, and therefore also will affect all the other peaks in the impulse train, especially since the sinc function is infinitely long.

Instead, what you will do is sample from the sinc function starting at the first peak until the midpoint between the two peaks, as illustrated in the following figure.



Start by drawing the first half of the pulse

And then mirror that same shape until the peak from the next impulse:



And then draw this same shape but in reverse

It's not perfect but it comes pretty close. The main difference is that in the mathematically correct version, the various pulses interfere with each other (since they're infinitely long), which doesn't happen here. But the approximation is good enough that this difference is neglectable. If you look at the frequency spectrum of an impulse train made using this method, there are still a few alias frequencies present but they mostly happen near Nyquist, out of the hearing range of humans. We can live with a few aliases like that.

Let's implement this algorithm step-by-step. First, replace the contents of **Oscillator.h** with the following:

```cpp
#pragma once

#include <cmath>

const float PI_OVER_4 = 0.7853981633974483f;
const float PI = 3.1415926535897932f;
const float TWO_PI = 6.2831853071795864f;

class Oscillator
{
public:
    float period = 0.0f;
    float amplitude = 1.0f;

    void reset()
    {
        inc = 0.0f;
        phase = 0.0f;
    }

    float nextSample()
    {
        // TODO
    }
private:
    float phase;
    float phaseMax;
    float inc;
};
```

Previously you chose the pitch of the oscillator by setting a frequency or a phase increment, but here you're supposed to set the `period` in samples. For this algorithm it makes more sense to think in terms of the period than the frequency, as the period gives you the number of samples between this impulse peak and the next.

The period of a sampled waveform is simply `sampleRate / freq`. For a 261.63 Hz tone at a 44.1 kHz sample rate, the period is `44100 / 261.63 = 168.56` samples.

Like the improved sine oscillator, this uses `phase` and `inc` variables but now they are private. They still serve the same function, though: `phase` keeps track of where you are in the sine wave, and `inc` determines how quickly the phase variable changes.

Wait, did I just say sine wave? The sinc function is made up of two parts:

$$\text{sinc} = \frac{\sin(\pi x)}{\pi x}$$

To make the sinc shape, you still need to calculate the sine function that goes into the numerator. Told you these sine waves are everywhere! However, this time the frequency of the sine wave is independent of the pitch of the sound — it always has a period of only 2 samples. (You guessed it: that means the frequency of this sine wave is the Nyquist frequency, so that it is zero on every sample.)

Also notice that this says πx. In this formula, x is the sample index. For the sinc function to go through zero in the right places, you must multiply the sample index by π. That's why in this new version of the oscillator, `phase` is not measured in samples but in "samples times π". A bit weird maybe, but this is done to avoid having to multiply by π all the time.

Note: Previously, `phase` counted up from 0 to 1, and `inc` was how fast it counted. Here it's slightly different: `phase` counts from 0 up to `(period/2)*π` to render the first half of the sinc function, and then counts back down to 0 again to render the second half in reverse. That's why the period is divided by two, because you'll render each impulse in two halves. The increment `inc` is π samples when counting up, and –π samples when counting down.

Now let's fill in the `nextSample` method. This is quite a chunk of code, but I'll explain what each part does below.

```
float nextSample()
{
    float output = 0.0f;

    phase += inc;  // 1

    if (phase <= PI_OVER_4) {  // 2

        // 3
        float halfPeriod = period / 2.0f;
        phaseMax = std::floor(0.5f + halfPeriod) - 0.5f;
        phaseMax *= PI;

        inc = phaseMax / halfPeriod;
        phase = -phase;

        // 4
        if (phase*phase > 1e-9) {
            output = amplitude * std::sin(phase) / phase;
        } else {
            output = amplitude;
        }
    } else {  // 5

        // 6
        if (phase > phaseMax) {
            phase = phaseMax + phaseMax - phase;
            inc = -inc;
        }
        // 7
        output = amplitude * std::sin(phase) / phase;
    }

    return output;
}
```

This is the most mathy piece of code in this book. It's OK if you don't quite get how it works, but don't worry, it's easier to understand than it may appear at first.

Just keep in mind that there are three phases to drawing the waveform. At any given time, you may be starting a new impulse; you may be drawing the first half of the sinc function; or you may be drawing the second half, which is the same as the first half but in reverse.

Step-by-step this is what the code does:

1. Update the phase. In the first half of the sinc function, `inc` is positive and `phase` is incremented. When drawing the second half, `inc` is negative and `phase` is decremented.

2. When we get here, it means the oscillator should start a new impulse. Recall that `phase` is measured in samples times π. If the phase is less than π/4, the end of the previous impulse is reached and you need to start a new one. The oscillator enters this `if` statement once per period, and also immediately when a new note starts.

3. Find where the midpoint will be between the peak that was just finished and the next one. This depends on the period. If `period` is 100 samples, the next midpoint will be 50 samples into the future.

   It's possible that the period changes while the oscillator is active, for example because the user plays a new note or by applying a modulation effect such as vibrato. To keep things simple, the oscillator ignores any changes to `period` until it starts the next cycle.

   The `phaseMax` variable holds the position of the midpoint between the two impulse peaks, and again is measured in samples times π. The floor function is used to "fudge" the halfway point a bit, which is a trick that helps to reduce aliasing. That also means `inc` isn't going to be exactly π but a value close to it.

4. Calculate sin$(\pi x)/(\pi x)$, where `phase` is the variable that holds the current πx. This is multiplied by the `amplitude` and becomes the output value for this sample.

   One small problem: sin$(0)/0$ gives a division by zero. In that case, the sinc function should output the value 1. Because floating point values are not always precise, you can't simply check if `phase == 0.0`. Very small values such as `1e-20` might still give problems with the division.

   Instead, take the square of `phase` and make sure this is a large enough number. The reason for squaring `phase` is that this value may be negative and `phase*phase` always makes it positive. An alternative way of writing this would be: `if (phase < -1e-9 || phase > 1e-9)`.

5. When we get here, it means the current sample is somewhere between the previous peak and the next. This is where the oscillator spends most of its time.

6. If the `phase` counter goes past the half-way point, set `phase` to the maximum and invert the increment `inc`, so that now the oscillator will begin outputting the sinc function backwards.

7. Calculate the current value for the sinc function. Here you don't have to check for a possible division by zero because `phase` will always be large enough.

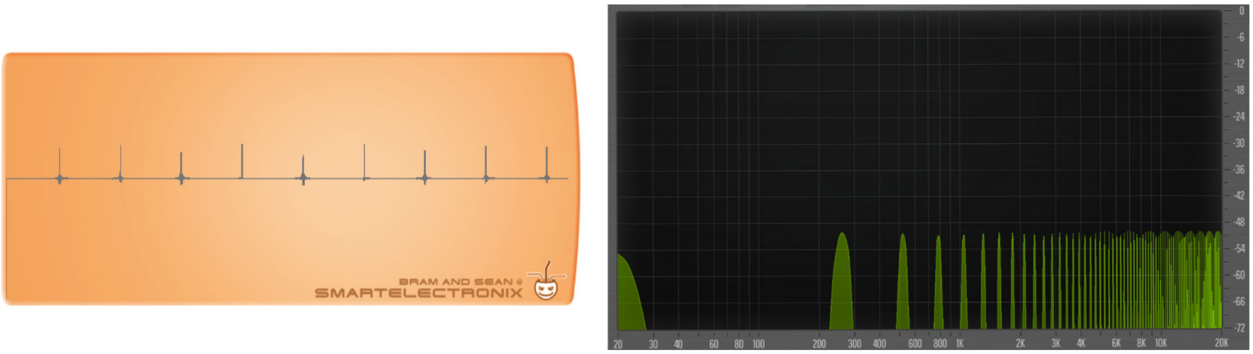
Phew! That was a lot to take in. It took me a few hours to make sense of this too the first time I saw it. If you're still not entirely sure how this algorithms works, don't worry about it for now. Let's make a few more changes to the code and then you can see it in action.

Tip: Whenever I encounter an algorithm that I don't understand, I implement it in Python and experiment with it in a Jupyter notebook. C++ is a great language for implementing fast DSP algorithms but it's not the most convenient way to do experiments. In a Python notebook, it's much easier to take apart the algorithm line-by-line and plot the results in a graph to see what happened. Another language commonly used for this is MATLAB.

In **Synth.cpp**, change the code in `noteOn` to the following:

```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;

    float freq = 440.0f * std::exp2(float(note - 69) / 12.0f);

    voice.osc.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc.period = sampleRate / freq;
    voice.osc.reset();
}
```

That's enough to get the oscillator going. Try it out! Play a note and you should hear a kind of thin sound. In the oscilloscope you'll see the sinc pulses happen at regular intervals. If you zoom in by changing the time knob, the sinc shapes are clearly visible for each pulse.



The impulse train in the oscilloscope and frequency analyzer

The frequency spectrum also kind of resembles an impulse train. There is equal spacing of 261.63 Hz between the peaks in the spectrum. (You can see this even better by setting the frequency axis in the spectrum analyzer to a linear scale instead of logarithmic.)

Interestingly, all the harmonics have the same height. This is different than before with the sawtooth, where the magnitude of the harmonics gradually dropped off. So even though this spectrum has the harmonics in the exact same place as the sawtooth wave, it sounds quite dissimilar because the harmonics have different magnitudes! Hence, the sound has a different timbre.

Also note the absence of aliases. There is no “junk” between the harmonics. That means you were successful in creating a bandlimited version of the impulse train!

To keep the algorithm somewhat simple to explain, I made use of the `std::sin` function again. However, for a more optimal implementation you can replace these with the digital resonator approximation.

Add the following private variables to the `Oscillator` class:

```
float sin0;
float sin1;
float dsin;
```

In `reset`, set these to zeros:

```
sin0 = 0.0f;
sin1 = 0.0f;
dsin = 0.0f;
```

In `nextSample`, inside the `if` statement, replace the following code,

```
if (phase*phase > 1e-9) {
    output = amplitude * std::sin(phase) / phase;
} else {
    output = amplitude;
}
```

by these lines:

```
sin0 = amplitude * std::sin(phase);
sin1 = amplitude * std::sin(phase - inc);
dsin = 2.0f * std::cos(inc);

if (phase*phase > 1e-9) {
    output = sin0 / phase;
} else {
    output = amplitude;
}
```

This initializes the digital resonator and uses it to output the sample for the peak of the sinc function. (In last chapter when we discussed the digital resonator, the initialization used an additional factor TWO_PI. You don't need that here because phase and inc already include PI. Also, phase only counts up to the halfway point which is why the factor 2 disappears.)

Inside the else clause, replace the line,

```
output = amplitude * std::sin(phase) / phase;
```

with:

```
float sinp = dsin * sin0 - sin1;
sin1 = sin0;
sin0 = sinp;

output = sinp / phase;
```

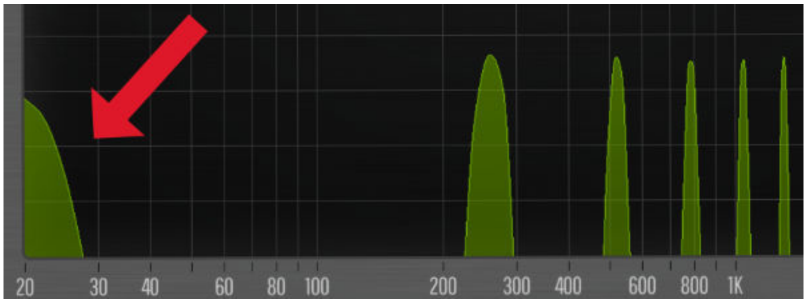
Here the sinp variable calculates the same thing as std::sin(phase). Note that you don't need to multiply by amplitude because the correct amplitude was already used in the initialization of the digital resonator.

Try it out. You still get the same output as before, but the code should run faster!

Note: Oscillator.h defines the constants PI and TWO_PI, for the values of π and 2π, respectively. The C standard library already defines M_PI, so why not use that? We're using floats but M_PI is a double, and that would mean any computations with M_PI would be performed as doubles. Is this bad? Not really, on modern architectures there isn't much difference between using float and double for computations. In fact, some plug-in authors exclusively use doubles. But since we're using float everywhere, we might as well keep PI as a float too. (In addition, Visual Studio does not define M_PI in the header cmath.)

Removing the DC offset

In the spectrum of the impulse train there is a peak all the way over on the left at 0 Hz. It looks more like a blob than a clean peak because spectrum analyzers are based on the FFT, which has less resolution on the low end, meaning that it can't tell apart low frequencies very well. Still, the BLIT sound appears to have a frequency below the fundamental frequency — how come?



The DC offset shows up as a frequency around 0 Hz

What you're seeing there is the so-called DC offset, which you can think of as the 0 Hz frequency. It is a constant value that is added to all the samples in the signal. Put differently, this DC offset happens because the average value of the samples in the signal is not zero.

Note: The term DC or "direct current" is taken from electrical engineering. It describes a current or voltage with a fixed, unchanging value. This in contrast to AC, or "alternating current", which describes the moving parts of the signal. In audio programming, we don't worry too much about AC and DC, but at least you now know where the terminology comes from.

Recall that the samples in the digital audio signal must be values between –1 and +1, which implies that 0 is the center. If no sound is happening, the output should be 0 for silence. But technically speaking, silence just means there is no movement of the speaker cones, so if you were to output 0.5 or +1 or even –1 all the time, you'd still hear only silence.

However, for those signals the average value is not zero, and we say that they have a DC offset. You can't hear a DC offset while sound is playing. However, when you stop such a sound you'll hear a "pop" as the speaker suddenly moves from the DC offset back to zero. That's why DC offsets are generally undesirable, since clicks and pops are sounds we want to avoid. In a frequency spectrum, if the DC offset is not zero, it shows up as a peak at 0 Hz.

It’s simple to remove the DC offset from the BLIT signal: subtract the signal’s average value. Because you’re doing this in real-time on a sample-by-sample basis, the average value needs to be divided by the number of samples to get the contribution for just one sample.

You’re not only doing this to get rid of any pops and clicks, but also because this step is necessary to turn the BLIT into a sawtooth wave.

Add a new private variable to the `Oscillator` class to hold the average amplitude:

```
float dc;
```

Set it to zero in the `reset` method:

```
dc = 0.0f;
```

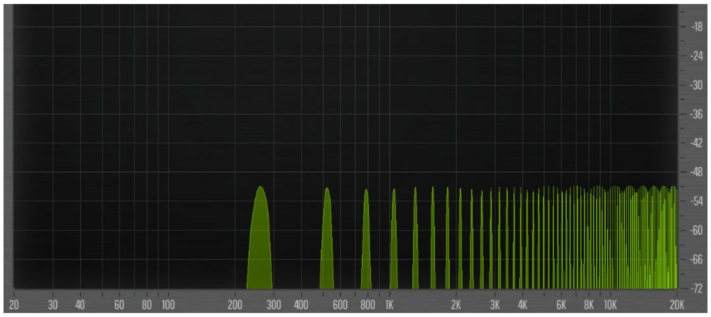
Then in `nextSample`, insert the code to calculate `dc` in the following place:

```
phaseMax = std::floor(0.5f + halfPeriod) - 0.5f;
dc = 0.5f * amplitude / phaseMax;      // insert this!
phaseMax *= PI;
```

Finally, change the return statement to:

```
return output - dc;
```

That’s all you need to do. The `dc` variable contains the DC offset per sample, so if you subtract this from each sample value, the average of the total signal will become zero and the DC offset disappears. Try it out! The frequency analyzer no longer shows the blob around 0 Hz:



There is no more peak around 0 Hz

Turning the BLIT into a sawtooth wave

By itself the impulse train doesn’t sound very nice. You did all this work because, with a small modification, the impulse train can be turned into a bandlimited sawtooth wave. Other types of classic waveforms are also possible, such as a square wave and a triangle wave.

In technical jargon, you will be integrating the BLIT, which just means that you add up the values coming out of `nextSample` over time.

Switch to **Voice.h** and add a new member variable to the struct:

```
float saw;
```

Set this value to zero in `reset`:

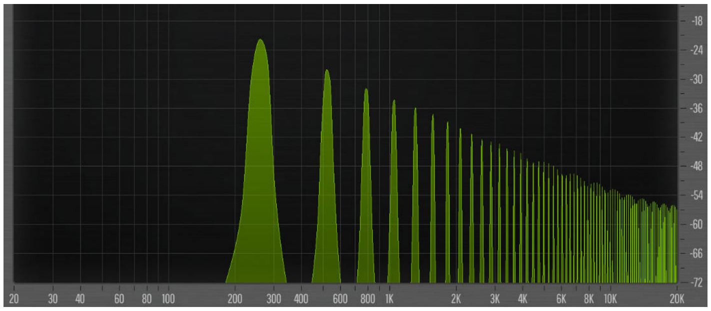
```
saw = 0.0f;
```

Then change `render` to the following:

```
float render()
{
    float sample = osc.nextSample();
    saw = saw * 0.997f + sample;
    return saw;
}
```

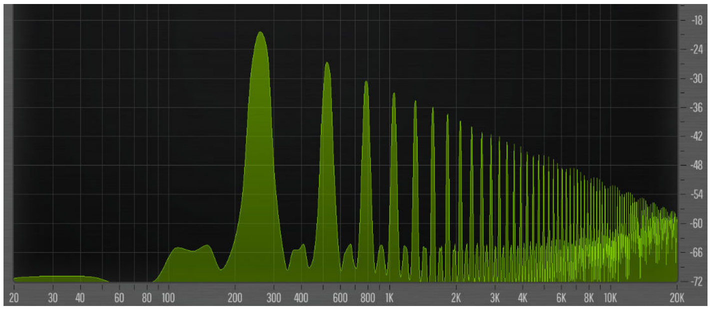
This simply keeps adding each next sample to the `saw` variable, and then returns that as the output sample. The multiplication by 0.997 is what makes this a so-called “leaky” integrator. This acts as a simple low-pass filter that prevents an offset from building up.

Try it out! The sawtooth wave sounds pretty good and doesn’t have a lot of aliasing. Here is its frequency spectrum:



The sawtooth in the frequency analyzer

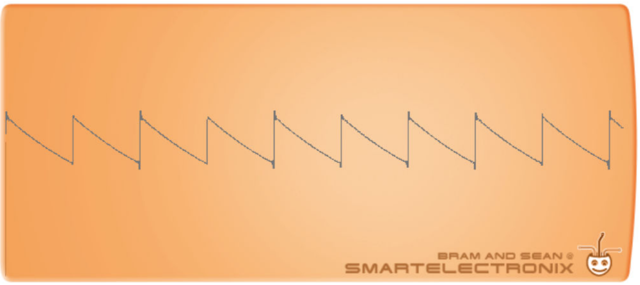
That’s a lot better than the naive sawtooth you had earlier. For comparison, it looked like this:



The sawtooth with the aliases

All that mess is gone now. Well, almost all. There still may be a few aliases present in the improved sawtooth but only in the higher frequency regions and they are quite small. It’s unlikely you’ll be able to hear them. For all intents and purposes, the spectrum of the BLIT sawtooth is indistinguishable from the true bandlimited version.

By the way, maybe you noticed that this sawtooth is the other way around from the ones you made before. Now it ramps down instead of up:



The sawtooth in the oscilloscope

This makes absolutely no difference in how it sounds, as the only difference is that signal has been flipped upside down. In other words, the phase has been shifted by 180 degrees, and you can’t hear phase shifts on signals that aren’t changing.

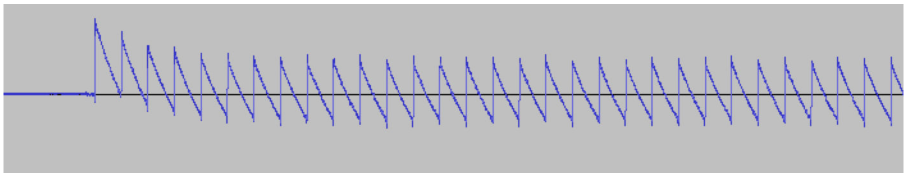
Try it out for yourself. Replace the line that calculates saw with the following:

```
saw = saw * 0.997f - sample;
```

You won’t be able to hear any difference and the spectrum still looks exactly the same. How interesting is that! Waveforms that look different can still sound exactly alike.

If you look carefully at the oscilloscope, you’ll see that the sawtooth shape has a little “bend” in it. This is due to the leaky integrator. If you change the 0.997 to something like 0.975 you’ll see that the bend becomes more pronounced and the sound becomes thinner. By making this factor small enough, the waveform will look more and more like the original impulse train.

One final observation: At the start of a note the sawtooth wave is not neatly centered around zero but starts off higher. It might be hard to see in the oscilloscope but you can see it by recording the output in a DAW:



The sawtooth starts with an offset

This is another example of a DC offset. It quickly gets reduced after only a few periods, and is a side effect of this particular method of sound synthesis. This is why you multiply by 0.997. Without this “leak”, the integrator wouldn’t be able to filter out this offset. You might even be able to see this DC offset in your frequency analyzer. It shows up as a small peak around 0 Hz at the start of the note, and then quickly fades away as you hold down the note.

Conclusion

Excellent, you’ve got a pretty decent sawtooth oscillator now without too much aliasing! It took a while to get here, but I think the journey was worth it!

As I said, the BLIT algorithm is no longer the state-of-the-art and a modern synth probably wouldn’t use it. But studying it should have given you some insight into the lengths that audio programmers have to go through to implement DSP code that sounds good.

If you’re wondering why I’m teaching you about some old algorithm, the quality of the algorithm is not the point. The reason I explained it in so much detail here is to give you an appreciation for what it’s like to work with sampled audio and a relatively complex oscillator algorithm to create a bandlimited signal.

Now that you’ve seen how to implement something like the BLIT algorithm, you’ll be able to go out and find better — and possibly easier! — algorithms and implement them yourself.

Some examples:

- There is an algorithm known as DPW, which stands for Differentiated Polynomial Waveforms. To make a bandlimited sawtooth wave, you first generate the simple but naive sawtooth, then you multiply each sample by itself, and finally subtract the previous sample value. This creates a sawtooth waveform that has more aliases than the BLIT-based method, but the algorithm is also a lot simpler.

- You could store one cycle of the bandlimited sawtooth waveform in a lookup table, for example by doing additive synthesis. If you make this lookup table one second long, e.g. 44100 samples if the sample rate is 44.1 kHz, then the table describes a 1 Hz signal. To output a signal at a higher pitch of `freq` Hz, loop through this table taking steps of size `freq`, and wrap around when you get to the end. This is known as wavetable synthesis.
- The most popular methods are based on BLEP, which stands for BandLimited stEP function. This is an enhancement of the BLIT method you've used in this chapter. BLEP already does the integration ahead of time. There are a few different variations of BLEP. If you're up for some homework, google "MinBLEP" or "PolyBLEP" and see if you can implement these algorithms in the `Oscillator` class.
- Another way to get rid of aliasing is to use oversampling, meaning that everything happens at a higher sample rate (2×, 4×, or even higher). If you do everything at 88200 Hz instead of the usual 44100 Hz, the aliases won't happen until 44100 Hz instead of 22050 Hz. You can simply filter them out and then downsample again to the regular sample rate. This is a valid technique and is used by a lot of synths, but it also takes up 2× the processing resources — you literally have to render twice as many samples.

If you're wondering where to find these kinds of algorithms, you can certainly find many of them in the various books and blog posts that have been written about sound synthesis, but a lot of this knowledge comes from academic papers. For example, the paper that introduced the BLIT technique is Alias-Free Digital Synthesis of Classic Analog Waveforms[41] by Stilson and Smith.

Unfortunately, papers are often written for other academics and experienced practitioners. Take the BLIT paper: there is a lot of math there! In fact, the PDF I linked to has typography errors inside some of the math formulas, making them hard to read for beginners. Ugh… That said, some papers are nicer than others and getting familiar with reading them is a useful skill to develop for someone who's interested in DSP programming.

If you want to have a go at reading papers, I suggest starting with Oscillator and Filter Algorithms for Virtual Analog Synthesis[42]. This is a very readable paper that describes the DPW algorithm, among other things.

[41] https://ccrma.stanford.edu/~stilti/papers/blit.pdf

[42] https://www.researchgate.net/publication/220386519_Oscillator_and_Filter_Algorithms_for_Virtual_Analog_Synthesis

Note: For the `Oscillator` class you wrote in this chapter, you worked with the period rather than the frequency. For other oscillator algorithms, it may make more sense to use the frequency instead. Frequencies are often more natural to think about than periods. Due to this choice, the coming chapters may appear to do things the “other way around”.

For example, to go up in pitch, you would multiply the frequency by a larger number. That seems very logical. However, because we’re using the period, to go up in pitch the period must be multiplied by a smaller number. If in the next chapters you find yourself wondering why we’re making something smaller when it’s supposed to be larger, or vice versa, remember that for this particular synth we do everything in terms of the period, not frequency.

Chapter 7: Plug-in parameters

I hope you're having fun so far! You've written code to handle MIDI and generate noise, as well as a pretty good sawtooth oscillator. Before you can build the rest of the synth, there is some bookkeeping to take care of. This chapter covers an essential part of modern plug-in development: managing the plug-in's parameters.

Every synthesizer has knobs that allow the user to change the sound that is produced. JX11 will have knobs for oscillator tuning, envelope attack, decay and release times, the filter cutoff and resonance, the LFO rate, and so on. For each of these you will define a parameter, which exposes these controls to the host application. Giving the host access to the synth's settings makes it possible for the user to do things like record automation, for example to automatically sweep the filter.

Each plug-in format has its own unique method for declaring parameters. Fortunately JUCE hides these details from you. JUCE handles all communication with the host, so that when the user turns a knob the host is notified of this. Conversely, when the host plays back automation, JUCE makes sure the plug-in is notified of any changes.

JX11 has 26 parameters in total. Let's add them!

Parameter identifiers

Parameters in JUCE are instances of the class `AudioProcessorParameter`. There are different subclasses for floating-point parameters, integer parameters, boolean parameters, and so on. In JX11 you'll be using `AudioParameterFloat` for settings that have a continuous range and `AudioParameterChoice` for letting the user choose from a number of options.

Each parameter needs to have a unique name that identifies it. In JUCE this identifier is a string, or more precisely, a `juce::String` object. Rather than hardcoding these strings, you'll use a C++ preprocessor trick to automatically generate them. That saves some typing and makes it harder for mistakes to creep in.

As of JUCE 7, the full parameter identifier is an instance of `juce::ParameterID`. This is a small class that combines the string identifier with a version number. This version number is necessary in particular for Audio Unit plug-ins. It ensures that you can safely add new parameters in updates to the plug-in, so that future versions of the plug-in will remain backwards compatible with existing music projects.

Tip: Since a lot of the source code in this chapter is repetitive, I suggest that you copy-paste the majority of this code from the GitHub repo, folder `Chapter 7`.

In **PluginProcessor.h**, between the include statements and the class definition, add the following lines:

```
namespace ParameterID
{
    #define PARAMETER_ID(str) const juce::ParameterID str(#str, 1);

    PARAMETER_ID(oscMix)
    PARAMETER_ID(oscTune)
    PARAMETER_ID(oscFine)
    PARAMETER_ID(glideMode)
    PARAMETER_ID(glideRate)
    PARAMETER_ID(glideBend)
    PARAMETER_ID(filterFreq)
    PARAMETER_ID(filterReso)
    PARAMETER_ID(filterEnv)
    PARAMETER_ID(filterLFO)
    PARAMETER_ID(filterVelocity)
    PARAMETER_ID(filterAttack)
    PARAMETER_ID(filterDecay)
    PARAMETER_ID(filterSustain)
    PARAMETER_ID(filterRelease)
    PARAMETER_ID(envAttack)
    PARAMETER_ID(envDecay)
    PARAMETER_ID(envSustain)
    PARAMETER_ID(envRelease)
    PARAMETER_ID(lfoRate)
    PARAMETER_ID(vibrato)
    PARAMETER_ID(noise)
    PARAMETER_ID(octave)
    PARAMETER_ID(tuning)
    PARAMETER_ID(outputLevel)
    PARAMETER_ID(polyMode)

    #undef PARAMETER_ID
}
```

This defines a new namespace containing one `juce::ParameterID` object for each of the twenty six parameter definitions. To get the identifier for the "Oscillator Mix" parameter, for example, you can now write `ParameterID::oscMix`. Thanks to C++ preprocessor magic, the actual string value for this identifier is `"oscMix"`. Since this is version 1.0 of JX11, the version number of each of the parameters is set to 1.

Still in **PluginProcessor.h**, add the following to the private variables inside the `JX11AudioProcessor` class:

```
juce::AudioParameterFloat* oscMixParam;
juce::AudioParameterFloat* oscTuneParam;
juce::AudioParameterFloat* oscFineParam;
juce::AudioParameterChoice* glideModeParam;
juce::AudioParameterFloat* glideRateParam;
juce::AudioParameterFloat* glideBendParam;
juce::AudioParameterFloat* filterFreqParam;
juce::AudioParameterFloat* filterResoParam;
juce::AudioParameterFloat* filterEnvParam;
juce::AudioParameterFloat* filterLFOParam;
juce::AudioParameterFloat* filterVelocityParam;
juce::AudioParameterFloat* filterAttackParam;
juce::AudioParameterFloat* filterDecayParam;
juce::AudioParameterFloat* filterSustainParam;
juce::AudioParameterFloat* filterReleaseParam;
juce::AudioParameterFloat* envAttackParam;
juce::AudioParameterFloat* envDecayParam;
juce::AudioParameterFloat* envSustainParam;
juce::AudioParameterFloat* envReleaseParam;
juce::AudioParameterFloat* lfoRateParam;
juce::AudioParameterFloat* vibratoParam;
juce::AudioParameterFloat* noiseParam;
juce::AudioParameterFloat* octaveParam;
juce::AudioParameterFloat* tuningParam;
juce::AudioParameterFloat* outputLevelParam;
juce::AudioParameterChoice* polyModeParam;
```

There is one variable for each parameter. All of them are `juce::AudioParameterFloat`, except for `glideModeParam` and `polyModeParam`, which are `juce::AudioParameterChoice`.

Note that these variables are merely pointers. The actual parameter objects are owned by something called the `AudioProcessorValueTreeState`, which you'll add next.

AudioProcessorValueTreeState

There are several ways to create the parameter objects and add them to the audio processor, but the recommended approach is to use the `AudioProcessorValueTreeState` object or APVTS for short. That's quite the mouthful!

JUCE has a powerful class called `ValueTree`, a tree structure that can contain objects of any type. The APVTS uses such a `ValueTree` to hold the plug-in's parameters.

One of its useful features is that the `ValueTree` allows you to connect listeners to nodes in the tree, for receiving notifications when something in the tree changes. It can also serialize the tree to XML, supports undo/redo functionality, automatically manages the lifetimes of nodes through reference counting, and more.

For JX11, you're going to ignore most of this and just treat the APVTS as the thing that owns the plug-in's parameters. The advantage over managing the parameters by hand is that the APVTS makes it really easy to hook up the parameters to GUI controls, and to save the plug-in's state and restore it later.

To use the APVTS, you need to add an instance to the audio processor. Add the following line inside the class definition in **PluginProcessor.h**. It's usually added to the `public` section, so that the editor class that handles the UI can also access it.

```
juce::AudioProcessorValueTreeState apvts { *this, nullptr, "Parameters",
                                           createParameterLayout() };
```

This calls a new the method `createParameterLayout`. Inside this method is where you will instantiate all the `AudioParameterFloat` and `AudioParameterChoice` objects.

Add the following declaration to the header. This method can be private:

```
juce::AudioProcessorValueTreeState::ParameterLayout createParameterLayout();
```

For now, add a minimal implementation of this method to **PluginProcessor.cpp**:

```
juce::AudioProcessorValueTreeState::ParameterLayout JX11AudioProcessor::createParameterLayout()
{
    juce::AudioProcessorValueTreeState::ParameterLayout layout;

    // you will add the parameters here soon

    return layout;
}
```

This is a good point to build the project to make sure you're not getting any error messages. If all is well, next you're going to add the parameter definitions to the `createParameterLayout` method.

Adding the parameters

The parameters reflect the design of the synth. As you're adding these parameters and you're not exactly sure what each one is for, please refer back to chapter 1 for an overview of the design of the synth. I'll also add a short explanation of each parameter, and in the subsequent chapters you'll revisit them in detail as you implement the remaining functionality of the synth.

Inside `createParameterLayout`, add the following code:

```
layout.add(std::make_unique<juce::AudioParameterChoice>(
    ParameterID::polyMode,
    "Polyphony",
    juce::StringArray { "Mono", "Poly" },
    1));
```

This is the same pattern you'll use for all parameters. The `layout` object is an instance of `juce::AudioProcessorValueTreeState::ParameterLayout`. This is a helper object used to construct the APVTS. It's a temporary container for passing around a set of `AudioParameter` objects. The `ParameterLayout` assumes ownership of the `AudioParameter` object, which is why the parameter is constructed using `std::make_unique`.

The parameter object you're creating here is a `juce::AudioParameterChoice`. Its constructor requires at least four arguments:

1. the identifier, here `ParameterID::polyMode`
2. the human-readable name of the parameter, **Polyphony**. This is what the DAW will show to the user if they inspect the plug-in's parameters.
3. a `juce::StringArray` object containing a list of choices. The `polyMode` parameter has two choices: **Mono** and **Poly**.
4. the index of the default choice, here 1 meaning the Poly option

That's it. The synth now has a parameter for choosing between the monophonic and polyphonic playing modes.

> **Note:** Since there are only two choices, instead of `AudioParameterChoice` you could have used `AudioParameterBool`, for polyphony on/off. Either will do the job.

Adding the other parameters is just as straightforward. Let's add the second parameter, for the amount of oscillator detuning in semitones, which is an `AudioParameterFloat`. Add the following code below the first parameter:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::oscTune,
    "Osc Tune",
    juce::NormalisableRange<float>(-24.0f, 24.0f, 1.0f),
    -12.0f,
    juce::AudioParameterFloatAttributes().withLabel("semi")));
```

The code is very similar, although `AudioParameterFloat` takes slightly different arguments:

1. the identifier, here `ParameterID::oscTune`
2. the human-readable name, **Osc Tune**. You could have called this "Oscillator Tuning" but these names shouldn't become too long and "Osc" is a well-known abbreviation in the world of synths.
3. a `juce::NormalisableRange` object, which describes the allowed values this parameter can have — notice the British spelling!
4. the default value, which you set to –12
5. an attributes object that describes the label for the units. The oscillator tuning is expressed in semitones, so this says "semi". You can put anything here you like, it's just a hint for the user.

In older plug-in formats such as VST2, parameters were always treated as floats between 0.0 and 1.0, and for compatibility reasons JUCE still treats them that way internally. However, it's more convenient to describe these parameters in the actual ranges they can accept. The `NormalisableRange` object exists to do that mapping from your preferred range to 0 – 1 and back.

The `oscTune` parameter can have values from –24 to +24. The third number in the `NormalisableRange` is the step size. Here this is 1.0 because we want the tuning to be in steps of whole semitones.

Creating a generic UI

Before you continue adding the rest of the parameters, it would be good to verify that what you have so far actually works. However, the synth does not have a user interface yet and it's a lot of work to build one. Fortunately, JUCE comes with a handy class that can auto-generate a UI from the plug-in's parameter definitions. It's very easy to use.

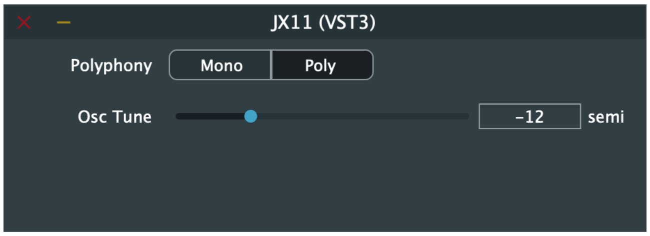
In **PluginProcessor.cpp**, go to `createEditor`. That method currently has the following code:

```
juce::AudioProcessorEditor* JX11AudioProcessor::createEditor()
{
    return new JX11AudioProcessorEditor (*this);
}
```

Replace the code inside the method with:

```
auto editor = new juce::GenericAudioProcessorEditor(*this);
editor->setSize(500, 1050);
return editor;
```

Now build and run the plug-in. Double-click the JX11 block inside AudioPluginHost to open the UI, or right-click and choose **Show plugin GUI**. Instead of the "Hello World!" message, the window now has controls for the two parameters you've defined.



Osc Tune -12 semi

The Polyphony and Osc Tune parameters in the generic editor

You can click the buttons and move the slider around but obviously they don't do anything, since the plug-in does not listen to changes to the parameters yet. That's something you will add in a later section. Double-clicking a slider will restore it to its default position. Alt-clicking does the same.

If the window is too big to fit on your screen, feel free to change the height from 1050 into something smaller. JUCE will automatically add a scrollbar when necessary.

> **Tip:** If you save the filtergraph from AudioPluginHost with the plug-in editor showing, it will automatically open the editor the next time you run AudioPluginHost. However, the plug-in does not remember the position of the slider handle, it always gets reset to –12. Doing so requires the plug-in to save and restore its state, which you'll add later in the chapter.

The skew factor

Let's continue adding the parameters… two down, twenty four to go.

Put the following below the previous definitions in `createParameterLayout`:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::oscFine,
    "Osc Fine",
    juce::NormalisableRange<float>(-50.0f, 50.0f, 0.1f, 0.3f, true),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("cent")));
```

Recall that JX11 will have two sawtooth oscillators running at the same time. The `oscFine` parameter is for fine-tuning the second oscillator. Where the previous parameter, **Osc Tune**, is used to tune the second oscillator up or down in steps of a whole semitone at a time, **Osc Fine** works with much smaller steps in cents, or 1/100th of a semitone.

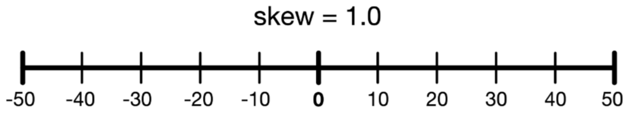
The only new thing here is that the `NormalisableRange` constructor takes five arguments instead of just three. This is what they mean:

1. the minimum value for the parameter, –50 cents
2. the maximum value for the parameter, +50 cents
3. the smallest step size by which the parameter can be adjusted, 0.1 cents
4. a skew factor, which changes the parameter curve from linear to something more exponential
5. if `true`, the skew happens from the center out

What is this skew thing about? For many parameters, it's fine if dragging the slider or turning the knob always affects the parameter by the same amount. This is what happens with **Osc Tune**: the slider goes from left to right in steps of one semitone.

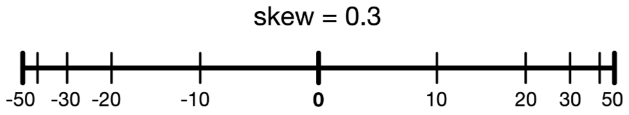
But for certain parameters you want to have more precise control over smaller values. Try out the plug-in and you'll find that the **Osc Fine** slider moves in tiny steps when the handle is close to the center but in larger steps the further away from the center you go. This makes fine-grained adjustments easier when the parameter is close to zero.

The skew factor, which here is 0.3, changes the way the possible values are distributed across the entire range. Instead of this,



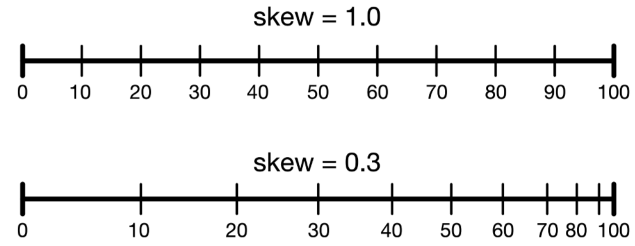
A slider without skew

the slider looks more like this:



Slider with symmetric skew

A skew factor of 1.0 has no effect. I picked 0.3 because it felt right, but go ahead and experiment with it. For the **Osc Fine** parameter, skew is relative to the center of the slider, but it's also possible to have it go from left to right:



Examples of non-symmetric skew

The skew factor is often used for parameters that represent something that is logarithmic in nature, such as a frequency. Skew is common with time-based parameters too. For example, the difference between 0.1 and 0.2 seconds can be much more noticable than the difference between 3.1 and 3.2 seconds, and so you'd want more precision on the small end of the scale. In general, use skew whenever it makes sense to offer the user more control over small values than over large values. Tip: Holding Ctrl or Command while dragging a slider will also let you make fine-grained adjustments.

Displaying customized values

Onwards to the next parameter. Add the following below the other parameter definitions:

```
auto oscMixStringFromValue = [](float value, int)
{
    char s[16] = { 0 };
    snprintf(s, 16, "%4.0f:%2.0f", 100.0 - 0.5f * value, 0.5f * value);
    return juce::String(s);
};

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::oscMix,
    "Osc Mix",
    juce::NormalisableRange<float>(0.0f, 100.0f),
    0.0f,
    juce::AudioParameterFloatAttributes()
            .withLabel("%")
            .withStringFromValueFunction(oscMixStringFromValue)));
```

JX11 has two independent oscillators. The **Osc Mix** parameter sets the amount they will be mixed together. This is expressed as a percentage. Oscillator 1 is always 100% present but Oscillator 2 can be added in from 0 – 100%.

Although there is more code here, what happens is very similar to what you did before. No step size is provided to the `NormalisableRange` constructor, meaning that the value can vary freely between `0.0` and `100.0`, in as small steps as the UI allows.

The last part is new. Previously you used `juce::AudioParameterFloatAttributes` to set the label for the units. The attributes object is used to describe additional behavior of the parameter. Besides the label, it has other configuration options too.

For the **Osc Mix** parameter, you use `withStringFromValueFunction` to supply a string-from-value function, a lambda that converts the current value of the parameter to a `juce::String`. It is this string that will be displayed in the UI and be provided to the host application. If you want to display the parameter's value in a non-standard way, you can do this by providing such a lambda.

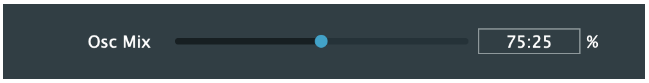
> **Note:** There is also a value-from-string lambda that goes the other way around: it takes a string and converts it back to a valid parameter value. This is useful for when users can type in the value of the parameter. You set this on the attributes object using `withValueFromStringFunction`. We won't have this feature in JX11.

The **Osc Mix** parameter will be shown as a ratio: from “100:0” if Oscillator 2 is turned off, to “50:50” if it’s fully turned on. Creating that string is what happens in the lambda, highlighted here for clarity:

```
[](float value, int)
{
    char s[16] = { 0 };
    snprintf(s, 16, "%4.0f:%2.0f", 100.0 - 0.5f * value, 0.5f * value);
    return juce::String(s);
}
```

This lambda is just like a function but specified inline. It takes two parameters: a `float` with the parameter’s current value and an `int` with the maximum allowed length for the string. Inside this lambda you convert the value into a `juce::String` object. Here, that’s done by `snprintf` into a temporary character buffer, but you could also use various methods on `juce::String` to achieve the same.

Try it out. The UI shows the mix amount as a ratio of two parameters:



The parameter with its custom value display

Adding the remaining parameters

The rest of the parameter definitions happen much in the same vein. Feel free to copy the code from the GitHub repo, I wouldn’t type this all in by hand either.

First up are the glide options that control the portamento effect:

```
layout.add(std::make_unique<juce::AudioParameterChoice>(
    ParameterID::glideMode,
    "Glide Mode",
    juce::StringArray { "Off", "Legato", "Always" },
    0));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::glideRate,
    "Glide Rate",
    juce::NormalisableRange<float>(0.0f, 100.f, 1.0f),
    35.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::glideBend,
    "Glide Bend",
    juce::NormalisableRange<float>(-36.0f, 36.0f, 0.01f, 0.4f, true),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("semi")));
```

The **Glide Mode** parameter is a choice box that offers three options: off, glide when legato-style playing, and always glide.

The **Glide Rate** determines how long it takes to glide from one note to the next. In some synths you can set a specific time for this, but here it is expressed as percentage. It's not clear exactly what it is a percentage of, but it's somewhere between "a short time" and a "long time". This is often true for synth parameters: you set them by feel rather than by worrying about the exact number of milliseconds.

The **Glide Bend** parameter lets you add an additional glide to any note that gets played, anywhere between –36 and +36 semitones. This again is a skewed parameter.

Next are the parameters for the resonant filter. Besides the oscillators, the filter is the most important tool that an analog synth uses to shape the sound. That's why there are nine parameters for the filter alone.

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterFreq,
    "Filter Freq",
    juce::NormalisableRange<float>(0.0f, 100.0f, 0.1f),
    100.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterReso,
    "Filter Reso",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    15.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

The first filter parameters are for the cutoff frequency and resonance settings. These also are expressed as percentages. Exactly how the cut-off frequency of the filter is determined is quite complicated and depends on many different factors — the **Filter Freq** parameter is only one of them.

The next filter settings determine by how much the filter envelope will be modulated:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterEnv,
    "Filter Env",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 0.1f),
    50.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterLFO,
    "Filter LFO",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

The **Filter Env** parameter sets the intensity for the filter envelope and **Filter LFO** does the same for the LFO modulation.

```
auto filterVelocityStringFromValue = [](float value, int)
{
    if (value < -90.0f)
        return juce::String("OFF");
    else
        return juce::String(value);
};

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterVelocity,
    "Velocity",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes()
            .withLabel("%")
            .withStringFromValueFunction(filterVelocityStringFromValue)));
```

The **Velocity** parameter has a custom string-from-value function. It changes the text that is displayed to “OFF” when the slider is dragged all the way to the left. The range of this parameter is from –100% on the left, to +100% on the right, with 0% in the center.

When set to 0%, this disables velocity sensitivity for the filter modulation but velocity will still be used to determine the amplitude of the notes. Setting it to OFF will disable velocity altogether and all notes will be equally loud.

The final four filter parameters are for the filter envelope:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterAttack,
    "Filter Attack",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterDecay,
    "Filter Decay",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    30.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterSustain,
    "Filter Sustain",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::filterRelease,
    "Filter Release",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    25.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

Often **Attack**, **Decay**, and **Release** are specified in milliseconds but in JX11 they are percentages. Just like with the glide rate, 0% means fast and 100% means slow, and anything else is in between. Exactly how fast or slow this is, that's something you'll have to find out by playing with the synth. **Sustain** is a percentage of the amplitude level.

Next up are four parameters for the amplitude envelope. They are defined exactly like the ones for the filter envelope, but they have different default values.

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::envAttack,
    "Env Attack",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::envDecay,
    "Env Decay",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    50.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::envSustain,
    "Env Sustain",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    100.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::envRelease,
    "Env Release",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    30.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

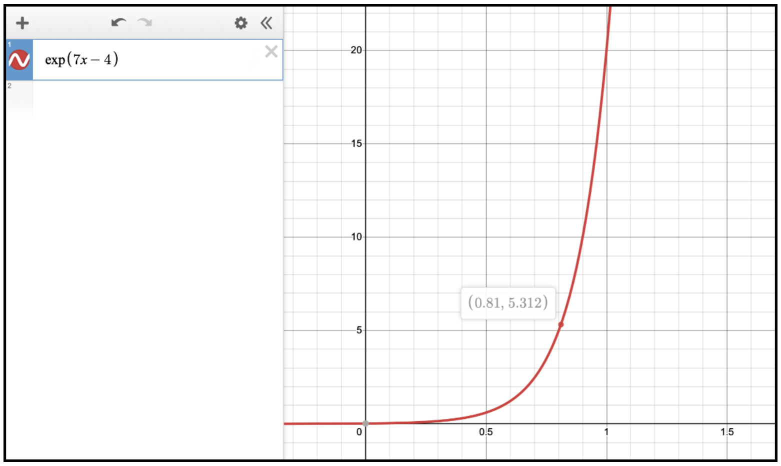
Besides using envelopes to modulate the filter and oscillators, JX11 also has a basic LFO. The **LFO Rate** parameter lets the user set the frequency of the LFO.

```
auto lfoRateStringFromValue = [](float value, int)
{
    float lfoHz = std::exp(7.0f * value - 4.0f);
    return juce::String(lfoHz, 3);
};

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::lfoRate,
    "LFO Rate",
    juce::NormalisableRange<float>(),
    0.81f,
    juce::AudioParameterFloatAttributes()
            .withLabel("Hz")
            .withStringFromValueFunction(lfoRateStringFromValue)));
```

The `NormalisableRange` here takes no arguments, which means this parameter is a float between 0.0 and 1.0. However, if you drag the slider you'll see this parameter goes between 0.018 Hz and 20.086 Hz. LFOs generally don't need to go much faster than 20 times per second; if they do, the LFO itself may become audible as an additional pitch in the sound.

This mapping from the range 0 – 1 to a value in Hz is done by the formula `std::exp(7.0 * value - 4.0)`. If you plot this formula using the excellent Desmos[43] online graphing calculator you'll get the following graph.

[43] https://www.desmos.com/calculator



The curve for the LFO Rate parameter

The area of interest is between `x = 0` and `x = 1`. As you can see this is an exponential curve that starts low and gradually becomes higher. The default value of 0.81 for this parameter means an LFO rate of `std::exp(7.0 * 0.81 - 4) = 5.312` Hz.

Using a curve like this is similar to the skew parameter, which works more like the formula x^{skew}. The reason you're not using skew here is that I wanted to demonstrate you can also implement your own curves. You will use similar exponential curves in other parts of the synth too, for example to convert the envelope attack, decay, and release percentages into actual times.

The **Vibrato** parameter tells the LFO how much modulation should be applied:

```
auto vibratoStringFromValue = [](float value, int)
{
    if (value < 0.0f)
        return "PWM " + juce::String(-value, 1);
    else
        return juce::String(value, 1);
};

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::vibrato,
    "Vibrato",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 0.1f),
    0.0f,
    juce::AudioParameterFloatAttributes()
            .withLabel("%")
            .withStringFromValueFunction(vibratoStringFromValue)));
```

This slider has two functions: it sets the modulation depth as a percentage between 0 and 100%, but when the slider is dragged to the left it switches from regular vibrato to PWM (pulse width modulation). This is a fairly common technique in synthesizer UIs when two effects are mutually exclusive: turn a knob one way to enable one effect, turn it the other way to enable the other effect. If the **Vibrato** parameter is a negative number, the synth should perform PWM instead of vibrato.

Almost there. The next parameter determines the amount of noise to mix into the sound. You'll be playing with this particular parameter throughout the rest of the chapter.

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::noise,
    "Noise",
    juce::NormalisableRange<float>(0.0f, 100.0f, 1.0f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("%")));
```

There are also two parameters for master tuning. Where the `oscTune` and `oscFine` parameters determine the relative tuning of the two oscillators, these master tuning parameters set the overall tuning of the synth. You can tune two octaves up or down, which is useful for bass sounds, and fine-tune by 100 cents or one semitone.

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::octave,
    "Octave",
    juce::NormalisableRange<float>(-2.0f, 2.0f, 1.0f),
    0.0f));

layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::tuning,
    "Tuning",
    juce::NormalisableRange<float>(-100.0f, 100.0f, 0.1f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("cent")));
```

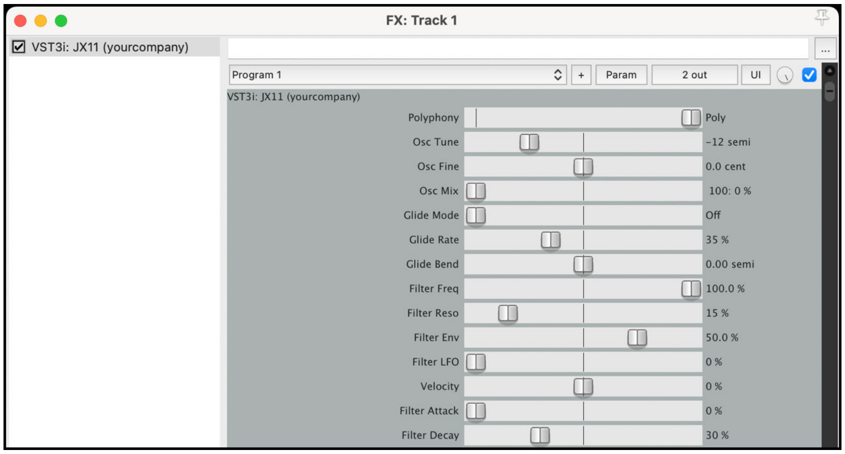
Finally, there is a parameter that lets you change the overall output volume of the synth from –24 to +6 dB:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::outputLevel,
    "Output Level",
    juce::NormalisableRange<float>(-24.0f, 6.0f, 0.1f),
    0.0f,
    juce::AudioParameterFloatAttributes().withLabel("dB")));
```

Phew! That's all of them. Try it out, you should now have a window filled with sliders.

In the generic UI, anything that's an `AudioParameterFloat` gets a horizontal slider, while `AudioParameterChoice` gets toggle buttons or a combo box. In chapter 13 you'll see how to make proper knobs for these parameters, but for now the sliders will have to do.

You should also be able to view these parameters in your DAW. For example, load JX11 into REAPER and then press the **UI** button. The display switches to REAPER's own generic view:

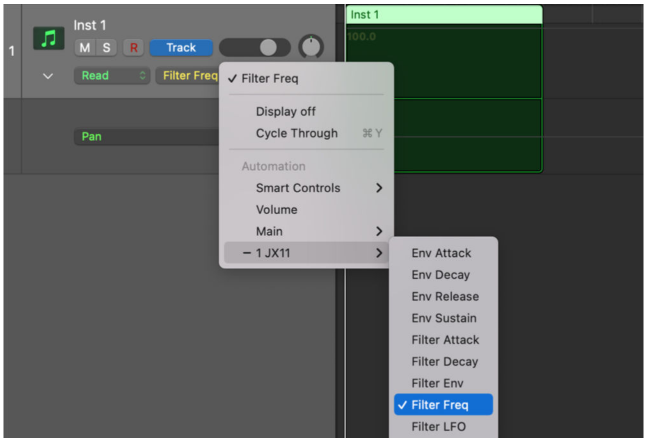


Polyphony	Poly
Osc Tune	-12 semi
Osc Fine	0.0 cent
Osc Mix	100: 0 %
Glide Mode	Off
Glide Rate	35 %
Glide Bend	0.00 semi
Filter Freq	100.0 %
Filter Reso	15 %
Filter Env	50.0 %
Filter LFO	0 %
Velocity	0 %
Filter Attack	0 %
Filter Decay	30 %

REAPER's generic UI for the parameters

For the VST3 version of JX11, REAPER shows a lot of additional parameters for all the possible MIDI CC codes. This is a workaround for an unfortunate design choice with the VST3 plug-in format. VST3 does not support MIDI CC messages unless they are added as "fake" parameters. JUCE handles this behind the scenes so you don't have to worry about this, but some DAWs like REAPER will show these extra parameters to the user.

The plug-in's parameters should appear in the automation menu for your DAW. For example, in Logic Pro:



The parameters in Logic Pro's automation menu

> **Tip:** If the compiler ever gives an error in the `<memory>` header file when you try to build a plug-in, you probably made a mistake in one of these parameter definitions, by providing too few or too many arguments. The error message isn't very clear, but the error happens because the C++ compiler doesn't understand how to construct the parameter object using `std::make_unique` if the number of arguments does not match any of the constructors for `AudioParameterFloat` or the other subclasses.

Threads and avoiding data races

Defining the parameters is only half the story, the plug-in must also react to changes to these parameters. A parameter gets new value when the user drags a slider or turns a knob in the UI, or because the host is doing something like playing back a recorded automation event or loading a user preset.

There are different ways to handle this sort of thing, and in this book you'll be using a basic method that works quite well. Before we get to the code, there is an important issue to address: how to deal with the different threads in your program.

I've mentioned that the audio processing code — the `processBlock` method — runs in a special high-priority thread that should never be blocked. The plug-in's UI, however, runs in a lower priority thread, usually called the UI thread or the main thread, or in JUCE parlance, the message thread.

You don't have to worry about creating these threads, they are managed by the host application. However, you do have to worry about which thread is being used for what.

When the user interacts with a knob or slider, the APVTS will post a notification on the UI thread to let both the host and plug-in know about it. When it is the host that informs the plug-in that a parameter has changed, this may happen on the UI thread, on the audio thread, or on some other thread.

Because of this, the plug-in can't assume which thread receives the "parameter has been updated" message. This is important, because the audio thread must share the parameter objects with these other threads! If the audio thread is reading a parameter's value while some other thread is modifying the parameter at the same time, that's a race condition happen to waiting.

As you hopefully know, two threads should not attempt to access the same variable at the same time, if at least one of them is writing to the variable. In regular programming this is an easy fix: just put a lock around the critical section. If two threads try to use the variable at the same time, one of them will have to wait. But us audio programmers can't use this solution: the audio thread should never be made to wait! This is one of the cardinal rules of audio programming — never block the audio thread.

There are a few ways around this issue and JUCE already provides one solution: the `AudioParameterFloat` object stores its value in an atomic variable. If a thread tries to read this atomic variable, it can safely do so without being interrupted. And if another thread tries to write the variable, it can safely do so without anyone currently trying to read it.

For datatypes such as `float`, atomics are implemented without requiring a lock, which makes them ideal for our purposes. If you haven't used atomics before, think of them as really lightweight synchronization primitives.

Thanks to these atomics, the `AudioParameterFloat`, `AudioParameterChoice`, and other parameters objects are thread-safe to use. Calling `yourParameter->get()` will safely read from the atomic variable and return the result as a `float` or `int`.

> **Note:** If you're not using JUCE, and your wrapper or plug-in SDK does not use atomic variables to store the parameter values, you'll have to make your own atomic variables. This is the case with Apple's AUv3 framework, for example. Sample code for these SDKs does not always use atomics, even when it should!

Reading the parameters in the audio callback

When should the audio thread try to read the parameter values? It's quite possible that parameters change in between calls to `processBlock` or even while `processBlock` is running, since this may happen on other threads.

You can pick up any changes that were made to the parameters simply by reading the parameters at the start of `processBlock`. The APVTS class has a convenient method `getRawParameterValue()` for this.

For example, in **PluginProcessor.cpp** in `processBlock`, you can add the following logic.

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                        juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;

    // Read the parameters here:
    const juce::String& paramID = ParameterID::noise.getParamID();
    float noiseMix = apvts.getRawParameterValue(paramID)->load() / 100.0f;
    noiseMix *= noiseMix;
    synth.noiseMix = noiseMix * 0.06f;

    // ... rest of the method ...
}
```

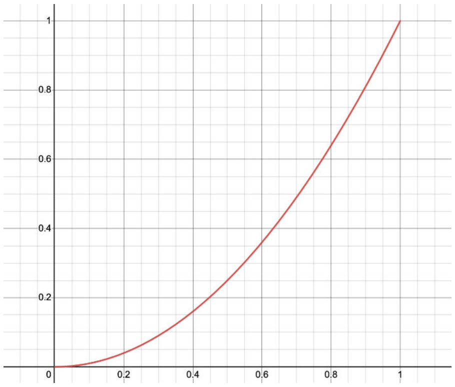
This reads the value of the parameter with the identifier `ParameterID::noise` at the start of the block, does a few computations on it, and then passes it to the `Synth` object.

The `apvts.getRawParameterValue()` function returns a `std::atomic<float>*`, a pointer to the atomic variable that holds the parameter's value. Then you call `load()` to get the actual `float` value. This is thread-safe because of the atomic.

The **Noise** parameter is shown in the UI as a percentage from 0 – 100%, which you turn into a value from 0 to 1 by dividing it by 100, and then it gets squared (by multiplying it by itself). Taking the square is similar to setting the skew factor on the parameter to 0.5, except doing that would change how the slider gets drawn.

Here, we want the slider to be linear, so as you drag it from 0 – 100% the slider moves in equal steps, while the amount of noise that gets mixed into the sound increases slowly at first but ever faster the closer you get to 100%. Again, this is done because loudness is logarithmic in nature and this curve roughly approximates the loudness increasing evenly the way we perceive it. It also gives the user more control over small amounts of noise.

The curve of x^2 looks like the following figure, for values of x between 0 and 1. As you can see, the value of x^2 does not grow as quickly as x (the slider position).



The curve of x squared

Finally, before it is passed to the `synth` object, the `noiseMix` variable is made significantly smaller (times 0.06), so that the maximum noise level is roughly –24 dB. That is loud enough for noise because we want the noise to be just the flavoring, not the entire dish.

The value of `noiseMix` is passed to the `Synth` object because that's where the actual audio rendering happens. In **Synth.h**, add a new public member variable to the class:

```
float noiseMix;
```

Then in **Synth.cpp**, update the `render` method to mix the output from the noise generator with the sawtooth:

```
void Synth::render(float** outputBuffers, int sampleCount)
{
    // ...

    for (int sample = 0; sample < sampleCount; ++sample) {
        float noise = noiseGen.nextValue() * noiseMix;  // change this line

        float output = 0.0f;
        if (voice.note > 0) {
            output = voice.render() + noise;  // change this line
        }
```

Try it out! Play a note and it should sound clean. Now drag the **Noise** slider up to 10% or so and play a note again — this time the sound has noise in it. You can even change the noise amount while the note is playing.

Using parameters through their pointers

What you just did is certainly a valid approach but getRawParameterValue performs a relatively inefficient parameter lookup by comparing strings, since parameter identifiers are juce::String objects under the hood. You'll be doing this lookup hundreds of times per second, since it happens at the start of processBlock.

It's more efficient use the AudioParameterFloat object directly, like so:

```
float noiseMix = noiseParam->get() / 100.0f;   // this line changed
noiseMix *= noiseMix;
synth.noiseMix = noiseMix * 0.06f;
```

You already added this noiseParam variable to the class definition earlier, as well as variables for all the other parameters, but these pointers are still uninitialized. If you try to run the plug-in with the above modification, it will immediately crash since noiseParam is either null or some garbage value.

Inside the JX11AudioProcessor constructor, add the following line:

```
noiseParam = dynamic_cast<juce::AudioParameterFloat*>(
                  apvts.getParameter(ParameterID::noise.getParamID()));
```

This grabs the parameter with the identifier ParameterID::noise from the APVTS, casts it to an AudioParameterFloat*, and then assigns it to the noiseParam variable.

One small problem is that you'll need to repeat the above line of code for all twenty six parameters. I don't find the dynamic_cast<> very pretty, so I prefer to use a small helper function for this. Add this function into the **Utils.h** file:

```
template<typename T>
inline static void castParameter(juce::AudioProcessorValueTreeState& apvts,
                                 const juce::ParameterID& id, T& destination)
{
    destination = dynamic_cast<T>(apvts.getParameter(id.getParamID()));
    jassert(destination);  // parameter does not exist or wrong type
}
```

Now go back to **PluginProcessor.cpp** and add an include for the utils source file.

```
#include "Utils.h"
```

Replace the contents of the constructor with the following:

```
castParameter(apvts, ParameterID::oscMix, oscMixParam);
castParameter(apvts, ParameterID::oscTune, oscTuneParam);
castParameter(apvts, ParameterID::oscFine, oscFineParam);
castParameter(apvts, ParameterID::glideMode, glideModeParam);
castParameter(apvts, ParameterID::glideRate, glideRateParam);
castParameter(apvts, ParameterID::glideBend, glideBendParam);
castParameter(apvts, ParameterID::filterFreq, filterFreqParam);
castParameter(apvts, ParameterID::filterReso, filterResoParam);
castParameter(apvts, ParameterID::filterEnv, filterEnvParam);
castParameter(apvts, ParameterID::filterLFO, filterLFOParam);
castParameter(apvts, ParameterID::filterVelocity, filterVelocityParam);
castParameter(apvts, ParameterID::filterAttack, filterAttackParam);
castParameter(apvts, ParameterID::filterDecay, filterDecayParam);
castParameter(apvts, ParameterID::filterSustain, filterSustainParam);
castParameter(apvts, ParameterID::filterRelease, filterReleaseParam);
castParameter(apvts, ParameterID::envAttack, envAttackParam);
castParameter(apvts, ParameterID::envDecay, envDecayParam);
castParameter(apvts, ParameterID::envSustain, envSustainParam);
castParameter(apvts, ParameterID::envRelease, envReleaseParam);
castParameter(apvts, ParameterID::lfoRate, lfoRateParam);
castParameter(apvts, ParameterID::vibrato, vibratoParam);
castParameter(apvts, ParameterID::noise, noiseParam);
castParameter(apvts, ParameterID::octave, octaveParam);
castParameter(apvts, ParameterID::tuning, tuningParam);
castParameter(apvts, ParameterID::outputLevel, outputLevelParam);
castParameter(apvts, ParameterID::polyMode, polyModeParam);
```

This looks up all the parameters in the APVTS structure and makes pointers to them, so that you can now use these pointers directly rather than having to do a slow lookup every time.

Listening to parameter changes

There is another improvement to make. For some of these parameters, the synth may need to perform expensive calculations when they are changed. The `noiseMix` parameter is simple enough: all it needs is a division by 100 to get it into the range 0 – 1. No point in trying to optimize that. But for some of the other parameters you will be calling `pow` or `exp` functions, which aren't known for being fast.

For example, there are six envelope parameters for which durations will be calculated and these each perform `std::exp` two times. It's a little silly to recalculate these things over and over at the top of `processBlock`, especially when these parameters have not changed. Wouldn't it be better to recalculate stuff only if it's needed?

Fortunately, that's fairly easy to accomplish in JUCE because the parameters are stored in the APVTS, and one of its features is that you can listen to changes. You will make the audio processor such a listener and call a special `update` method whenever the parameters have changed. If no parameters change, `update` isn't called and nothing gets recalculated.

In **PluginProcessor.h**, make `JX11AudioProcessor` also inherit from `juce::ValueTree::Listener`, like so:

```
class JX11AudioProcessor  : public juce::AudioProcessor,       // don't forget the comma
                            private juce::ValueTree::Listener   // add this
{
public:
    // ...
```

Add a new private method. The implementation is small enough that it can go into the header file:

```
private:
    void valueTreePropertyChanged(juce::ValueTree&, const juce::Identifier&) override
    {
        DBG("parameter changed");
    }
```

In **PluginProcessor.cpp**, add the following line to the bottom of the constructor. This hooks up the new `valueTreePropertyChanged` method to the APVTS:

```
apvts.state.addListener(this);
```

And remove the listener again in the destructor:

```
JX11AudioProcessor::~JX11AudioProcessor()
{
    apvts.state.removeListener(this);
}
```

Now build and run the plug-in. Every time you drag a slider or choose an option in the UI, you should see the message "parameter changed" in the debug console.

You will now use this `valueTreePropertyChanged` listener method to read from the parameters and recalculate anything the synth needs.

First, remove the lines that compute `noiseMix` from `processBlock`, as you no longer want to do this all the time. To only update the value of `noiseMix` whenever its parameter changes, you might be tempted to do this as follows:

```
void valueTreePropertyChanged(juce::ValueTree&, const juce::Identifier&) override
{
    float noiseMix = noiseParam->get() / 100.0f;
    noiseMix *= noiseMix;
    synth.noiseMix = noiseMix * 0.06f;
}
```

However, this would ignore all my good advice about threading! `valueTreePropertyChanged` might be called on who knows what thread. If the user is manipulating the UI, it will probably be called on the main thread, but it might also be called on some background thread.

That's a problem because `synth.noiseMix` should only ever be accessed by the audio thread, as this is not an atomic variable! Clearly this is the wrong approach.

The solution is to use `valueTreePropertyChanged` to signal the audio thread that one or more parameters have been updated, and then let the audio thread perform the calculations.

To do this, add the following variable to the private members in **PluginProcessor.h**:

```
std::atomic<bool> parametersChanged { false };
```

And change `valueTreePropertyChanged` to:

```
void valueTreePropertyChanged(juce::ValueTree&, const juce::Identifier&) override
{
    parametersChanged.store(true);
}
```

Whenever the APVTS notifies the listener that a parameter has received a new value, all this does is set the `parametersChanged` boolean to true. This is thread-safe, since it's an atomic variable.

Also add the following private method declaration to the class. This `update` method is where you'll do all the necessary calculations using the new parameter values.

```
void update();
```

In **PluginProcessor.cpp**, change `processBlock` to the following:

```
void JX11AudioProcessor::processBlock (juce::AudioBuffer<float>& buffer,
                                       juce::MidiBuffer& midiMessages)
{
    juce::ScopedNoDenormals noDenormals;
    auto totalNumInputChannels = getTotalNumInputChannels();
    auto totalNumOutputChannels = getTotalNumOutputChannels();

    for (auto i = totalNumInputChannels; i < totalNumOutputChannels; ++i) {
        buffer.clear(i, 0, buffer.getNumSamples());
    }

    // add these lines:
    bool expected = true;
    if (parametersChanged.compare_exchange_strong(expected, false)) {
        update();
    }

    splitBufferByEvents(buffer, midiMessages);
}
```

The new code does a thread-safe check to see whether `parametersChanged` is true. If so, it calls the `update` method to perform the parameter calculations. It also immediately sets `parametersChanged` back to false.

This “check if true and set back to false” is a single atomic operation. It either succeeds or it fails. If the operation fails, some other thread was trying to write to `parametersChanged` while `processBlock` was trying to read it. This is not a big deal. You’ll miss the parameter update in this block, but most likely the operation will succeed in one of the next blocks and `update` will eventually get called. The design pattern is that the UI thread is the only one that sets `parametersChanged` to true, and the audio thread is the only one that sets it to false.

The `update` method is as follows:

```
void JX11AudioProcessor::update()
{
    float noiseMix = noiseParam->get() / 100.0f;
    noiseMix *= noiseMix;
    synth.noiseMix = noiseMix * 0.06f;
}
```

This is the sole place where you’ll read from the `noiseParam` parameter. Its value is processed and stored in `Synth`. This is safe because `noiseParam->get()` is an atomic operation, and `update` is guaranteed to always run on the audio thread. Over the course of the next chapters, you will add a lot more calculations to this method.

One more small thing to add. Change prepareToPlay to the following:

```
void JX11AudioProcessor::prepareToPlay (double sampleRate, int samplesPerBlock)
{
    synth.allocateResources(sampleRate, samplesPerBlock);
    parametersChanged.store(true);
    reset();
}
```

This forces update() to be executed the very first time processBlock is called, so that the synth will be properly initialized with the initial state of the parameters.

To recap:

1. When the user or the host changes a parameter, valueTreePropertyChanged sets the parametersChanged boolean to true.
2. In processBlock, it first checks to see if parametersChanged is set. If so, update is called to read all the parameters and put their values into Synth's member variables.
3. Reading the parameters in update is thread-safe because JUCE's AudioParameter objects use atomics to store their values.
4. Any of the other functions called from processBlock, such as Synth::render, do not access the parameters, only their own variables.
5. The variables used by the audio thread, such as synth.noiseMix, are not written or read by anyone else. They are off-limits for the UI.

This simple approach keeps a clear separation between the audio thread and the UI thread and any other threads. Of course, the update method currently doesn't do much yet but this will change when you implement the rest of the synth.

If you're making a plug-in that has many parameters, you could go even further and exclusively update the variables for parameters that actually did change, unlike our update method that always recalculates everything. For this synth it is overkill but in your own projects it may be worth having different listeners for different (groups of) parameters.

In your career as an audio programmer, from time to time you will probably find yourself having to come up with clever ways to communicate between the audio thread and the UI thread in your plug-in. It's a tricky problem, but there are well-established methods for doing so, and using atomic variables is a key technique.

Saving and loading state

Try the following experiment: Make some changes to the sliders, choose **File** > **Save** in AudioPluginHost and then quit it. Restart AudioPluginHost and reload the graph. You'll notice all the sliders have been reset to their default values.

That's a problem because it means users of your plug-in will lose their carefully designed sounds once they quit the DAW. Fortunately, there is a way for the plug-in to save its current state and load it back the next time the plug-in is used. You want to keep those users happy!

The audio processor class has two methods, `getStateInformation` and `setStateInformation`, that are used to serialize and deserialize the plug-in's state.

In **PluginProcessor.cpp**, fill in these methods:

```
void JX11AudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    copyXmlToBinary(*apvts.copyState().createXml(), destData);
}

void JX11AudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<juce::XmlElement> xml(getXmlFromBinary(data, sizeInBytes));
    if (xml.get() != nullptr && xml->hasTagName(apvts.state.getType())) {
        apvts.replaceState(juce::ValueTree::fromXml(*xml));
        parametersChanged.store(true);
    }
}
```

That's all you have to do. Run the same test again, and now you'll find that the plug-in does remember its settings. Nice!

The `getStateInformation` method is used to save the plug-in's current state. You are supposed to serialize anything that is important into the given `juce::MemoryBlock`. For our synth, this is the current values of all the parameters, contained in the APVTS. One of the convenient features of the APVTS is that it can serialize its contents to an XML document using the `createXml` method. And then `copyXmlToBinary` is used to put the XML into the memory block.

In case you're curious about the XML that is generated, add the following line to `getStateInformation` and run the plug-in again.

```
DBG(apvts.copyState().toXmlString());
```

When you save in AudioPluginHost, the `DBG` statement will write something like the following to the debug console.

```
<?xml version="1.0" encoding="UTF-8"?>

<Parameters>
  <PARAM id="envAttack" value="0.0"/>
  <PARAM id="envDecay" value="50.0"/>
  <PARAM id="envRelease" value="30.0"/>
  <PARAM id="envSustain" value="100.0"/>
  <PARAM id="filterAttack" value="0.0"/>
  ...
</Parameters>
```

This is how the plug-in's state gets saved to an XML document.

In `setStateInformation` it goes the other way around. You are provided with a block of binary data. First, this calls `getXmlFromBinary` to parse the binary data into an XML document. Then it verifies this XML contains the `<Parameters>` tag, and finally it calls `apvts.replaceState()` to update the values of all the parameters to the new values.

After loading and restoring the plug-in's state, you set the `parametersChanged` boolean to true to signal `processBlock` that it should call `update` again. There is no way to know exactly when `setStateInformation` may be called in the lifetime of the plug-in, so you should assume it could happen at any time, even if the plug-in is busy processing audio. Hence, it's necessary to recalculate anything that depends on these parameter values the very next time `processBlock` is called.

The APVTS `replaceState` method is thread-safe, by the way. That's a good thing too, because there is no guarantee that `getStateInformation` or `setStateInformation` will be called from any particular thread, it doesn't have to be the UI thread.

You may be wondering exactly where this XML gets saved. This depends on the host application; it will typically end up in the project file. In the case of AudioPluginHost, the XML is saved in the **.filtergraph** file, which itself is XML. Open the filtergraph file in a text editor, and you'll see something like this:

```
<PLUGIN name="JX11" format="VST3" category="Instrument|Synth"
        manufacturer="yourcompany" .../>
<STATE>Ea0cVZtMEcgQWY9vSRC8Vav8lak4Fc9DCL3HiKV0jZLcFUq3hKt3xSqX1UgIWPnM1ZI....</STATE>
</PLUGIN>
```

The `<STATE>` field contains the plug-in's serialized state. Note that this doesn't show the XML from before but some sort of encoded version. That's because the serialization format does not need to be XML. JUCE doesn't care and neither do host applications. The only thing that matters is that your plug-in can serialize and deserialize its state to a blob of binary data. If you're not a fan of XML, feel free to use something else. However, the big advantage of using XML is that APVTS has built-in support for it already.

For more advanced plug-ins you may want to serialize additional settings that are not parameters. Maybe the state of the UI, for example whether a certain panel was visible. You can store this kind of information inside the plug-in's serialized state, even if you're not keeping it in the APVTS. Just add it to the XML yourself using JUCE's `XmlDocument` class.

Factory presets

The original MDA JX10 synth included fifty or so preset patches that show off the capabilities of this synth. You'll add those factory presets to JX11 too. In JUCE terminology the factory presets are called programs. Each program is basically a collection of parameter settings.

To round off this chapter, let's add these factory presets in the synth already. Having a good set of sounds available will make testing the synth a lot more fun!

Use **Projucer** to add a new source file to the project and name it **Preset.h**. Put the following into that new file. This is a basic struct that holds the values for the twenty six parameters.

```
#pragma once

#include <cstring>

const int NUM_PARAMS = 26;

struct Preset
{
    Preset(const char* name,
           float p0,  float p1,  float p2,  float p3,
           float p4,  float p5,  float p6,  float p7,
           float p8,  float p9,  float p10, float p11,
           float p12, float p13, float p14, float p15,
           float p16, float p17, float p18, float p19,
           float p20, float p21, float p22, float p23,
           float p24, float p25)
    {
        strcpy(this->name, name);
        param[0]  = p0;   // Osc Mix
        param[1]  = p1;   // Osc Tune
        param[2]  = p2;   // Osc Fine
        param[3]  = p3;   // Glide Mode
        param[4]  = p4;   // Glide Rate
        param[5]  = p5;   // Glide Bend
        param[6]  = p6;   // Filter Freq
        param[7]  = p7;   // Filter Reso
        param[8]  = p8;   // Filter Env
        param[9]  = p9;   // Filter LFO
        param[10] = p10;  // Velocity
        param[11] = p11;  // Filter Attack
        param[12] = p12;  // Filter Decay
        param[13] = p13;  // Filter Sustain
        param[14] = p14;  // Filter Release
        param[15] = p15;  // Env Attack
        param[16] = p16;  // Env Decay
        param[17] = p17;  // Env Sustain
        param[18] = p18;  // Env Release
        param[19] = p19;  // LFO Rate
        param[20] = p20;  // Vibrato
        param[21] = p21;  // Noise
        param[22] = p22;  // Octave
        param[23] = p23;  // Tuning
        param[24] = p24;  // Output Level
        param[25] = p25;  // Polyphony
    }

    char name[40];
    float param[NUM_PARAMS];
};
```

In **PluginProcessor.h**, add an include for this file:

```
#include "Preset.h"
```

In the private declarations for this class, add the following:

```
void createPrograms();

std::vector<Preset> presets;
int currentProgram;
```

The `presets` variable is a vector with all the `Preset` objects. `currentProgram` is the index of the currently selected preset. The new method `createPrograms` will be used to fill up the presets vector.

In **PluginProcessor.cpp** add the following two lines to the constructor:

```
createPrograms();
setCurrentProgram(0);
```

Important: put this below the lines that grab the parameter pointers from the APVTS, as `setCurrentProgram` will use these pointers.

`JX11AudioProcessor` has a number of built-in methods for managing these factory presets. They have already been added to the .cpp file. Implement these methods as follows.

```
int JX11AudioProcessor::getNumPrograms()
{
    return int(presets.size());
}

int JX11AudioProcessor::getCurrentProgram()
{
    return currentProgram;
}

const juce::String JX11AudioProcessor::getProgramName (int index)
{
    return { presets[index].name };
}

void JX11AudioProcessor::changeProgramName (int /*index*/, const juce::String& /*newName*/)
{
    // not implemented
}
```

JX11 does not support renaming the programs, so `changeProgramName` does nothing. The reason for commenting out the argument names of this method, is to stop compiler warnings about unused arguments.

> **Note:** The `Preset` struct stores the preset's name as a character array, but `getProgramName` returns a `juce::String` object. The JUCE `String` class has a constructor that accepts `const char*`, and so you can use C-style character arrays wherever a `juce::String` object is needed.

The final method to implement is `setCurrentProgram`, which is also the largest:

```
void JX11AudioProcessor::setCurrentProgram (int index)
{
    currentProgram = index;

    juce::RangedAudioParameter *params[NUM_PARAMS] = {
        oscMixParam,
        oscTuneParam,
        oscFineParam,
        glideModeParam,
        glideRateParam,
        glideBendParam,
        filterFreqParam,
        filterResoParam,
        filterEnvParam,
        filterLFOParam,
        filterVelocityParam,
        filterAttackParam,
        filterDecayParam,
        filterSustainParam,
        filterReleaseParam,
        envAttackParam,
        envDecayParam,
        envSustainParam,
        envReleaseParam,
        lfoRateParam,
        vibratoParam,
        noiseParam,
        octaveParam,
        tuningParam,
        outputLevelParam,
        polyModeParam,
    };

    const Preset& preset = presets[index];

    for (int i = 0; i < NUM_PARAMS; ++i) {
      params[i]->setValueNotifyingHost(params[i]->convertTo0to1(preset.param[i]));
    }

    reset();
}
```

This simply loops through the 26 elements in the `Preset` object's `params` array and assigns their values to the appropriate `AudioParameterFloat` and `AudioParameterChoice` objects.

The important thing here is that you call `setValueNotifyingHost` on the audio parameter object, so that JUCE will let the host application know the parameter has changed.

Because JUCE uses the values 0.0 – 1.0 internally for all parameter values, `setValueNotifyingHost` also expects to get a value between 0 and 1, and you must call the `convertTo0to1` method to convert the parameter's real value into that range. A bit awkward but that's how the JUCE API works.

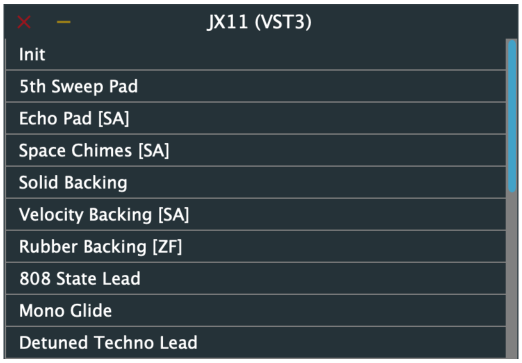
You also call `reset` to tell the synth to stop playing. This is necessary because if any notes are still playing, changing presets may lead to parameter values making big jumps, which can cause filter states to be invalid, resulting in `nan` (not-a-number) and `inf` (infinity) values in the audio buffer. Not something you'd want to happen! In debug mode, `protectYourEars` will stop this from blowing up your speakers, but it's even better to reset the synth state to avoid it in the first place.

The final thing to do is to implement the `createPrograms` method. It looks something like the following, with many lines of code just like that. My suggestion is that you copy-paste this method from the GitHub repo instead of typing in all these numbers.

```
void JX11AudioProcessor::createPrograms()
{
    presets.emplace_back("Init", 0.00f, -12.00f, 0.00f, 0.00f, 35.00f, 0.00f, 100.00f,
        15.00f, 50.00f, 0.00f, 0.00f, 0.00f, 30.00f, 0.00f, 25.00f, 0.00f, 50.00f,
        100.00f, 30.00f, 0.81f, 0.00f, 0.00f, 0.00f, 0.00f, 0.00f, 1.00f);

    presets.emplace_back("5th Sweep Pad", 100.00f, -7.00f, -6.30f, 1.00f, 32.00f, 0.00f,
        90.00f, 60.00f, -76.00f, 0.00f, 0.00f, 90.00f, 89.00f, 90.00f, 73.00f, 0.00f,
        50.00f, 100.00f, 71.00f, 0.81f, 30.00f, 0.00f, 0.00f, 0.00f, 0.00f, 1.00f);

    presets.emplace_back("Echo Pad [SA]", 88.00f, 0.00f, 0.00f, 0.00f, 49.00f, 0.00f,
        46.00f, 76.00f, 38.00f, 10.00f, 38.00f, 100.00f, 86.00f, 76.00f, 57.00f, 30.00f,
        80.00f, 68.00f, 66.00f, 0.79f, -74.00f, 25.00f, 0.00f, 0.00f, 0.00f, 1.00f);

    // ...
}
```

Each of these lines will construct a new `Preset` object, fills in its name and the twenty six parameter values, and then appends it to the end of the `presets` vector.

Try it out! The AudioPluginHost app shows the presets if you right-click the JX11 block and choose **Show all programs**.



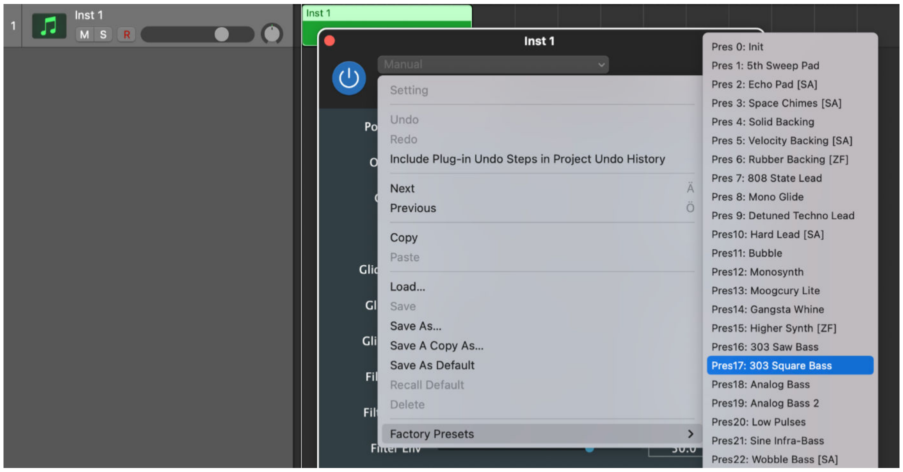
Init
5th Sweep Pad
Echo Pad [SA]
Space Chimes [SA]
Solid Backing
Velocity Backing [SA]
Rubber Backing [ZF]
808 State Lead
Mono Glide
Detuned Techno Lead

The list of factory presets in AudioPluginHost

However, at least at the time of writing, this screen does not let you select a program. Instead, right-click and choose **Show all parameters**. This pops up a small window using the same kind of generic UI that you're currently using for the editor.

Scroll all the way down, where you'll find a slider for choosing the program. As you select a different program, you'll notice the parameters get different values. Running the plug-in as a standalone app unfortunately does not let you choose presets.

If you're loading the plug-in in a DAW, use the DAW's factory presets menu to switch between presets. Where this is located depends on your DAW. The following image shows JX11's presets in Logic Pro.



Using the factory presets in Logic Pro

There is also a MIDI message for changing the sound that's being used on a given MIDI channel. This is the Program Change message. You can add support for this in JX11 too.

In **PluginProcessor.cpp**, change `handleMIDI` to the following:

```cpp
void JX11AudioProcessor::handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2)
{
    // Program Change
    if ((data0 & 0xF0) == 0xC0) {
        if (data1 < presets.size()) {
            setCurrentProgram(data1);
        }
    }

    // ...the rest here...
}
```

This checks if the command is `0xC`, which is Program Change. It ignores the channel number. The `data1` byte contains the number of the program, `data2` is not used. To test this, you'll need a MIDI controller that can send Program Change messages. You can also create MIDI events for this in your DAW.

User presets

Besides factory presets, most commercial synths allow users to save and load their own presets. This can be done in two ways:

1. The host application saves / loads the presets.
2. The plug-in saves / loads the presets.

You don't need to write any additional code for the first case, it uses the existing serialization mechanism. When using the host application to save a preset file, the host will ask the plug-in to serialize its state using `getStateInformation` and then it puts the binary blob into its own file format.

For example, Logic Pro uses **.aupreset** files, which again are XML. If you look inside this XML file, you'll find the plug-in's state encoded in some form. When the host loads such a preset, it will call `setStateInformation` on the plug-in.

The downside of letting the host handle the presets is that these presets cannot be shared with users who run other hosts. If you're using REAPER and I'm using Logic Pro, and you want to share your presets with me, Logic won't be able to read the REAPER preset files.

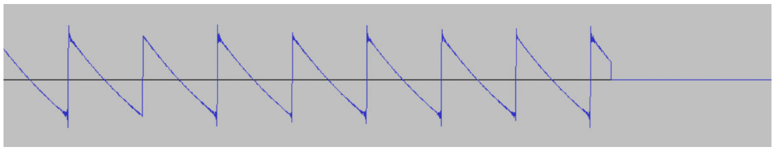
This is why many synths have their own method for saving and loading user presets. Again, you can use the existing serialization mechanism for this, but you will have to code additional file handling yourself as now the plug-in needs to put its XML into a file somewhere.

Writing a system to manage user presets is beyond the scope of this book. There is a great tutorial video[44] on The Audio Programmer's YouTube channel. Jatin Chowdhury of ChowDSP has a nice open source implementation[45].

[44] https://www.youtube.com/watch?v=YwAtWuGA4Cg
[45] https://github.com/Chowdhury-DSP/chowdsp_utils

Chapter 8: The ADSR envelope

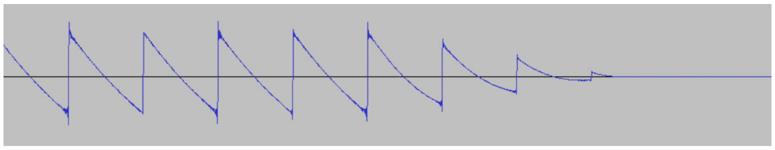
When playing a note using the current version of the plug-in, you might hear a popping sound when the note is released and the synth stops playing. This is an audio glitch that happens because the sound ends suddenly:



The sound ends with a jump

A big jump like that, from the waveform to all zeros, creates a clicking or popping sound. Don't feel bad if you hadn't noticed these glitches, the sawtooth wave is fairly complex and has a lot of movement in it already. Try playing a few notes in quick succession and you'll hear that the transition between the notes is kind of rough. The clicks are definitely audible when using the basic sine wave oscillator.

Audio works best when changes are smooth. The popping effect can be eliminated by fading out the sound when it ends:



The sound fades out when the note ends

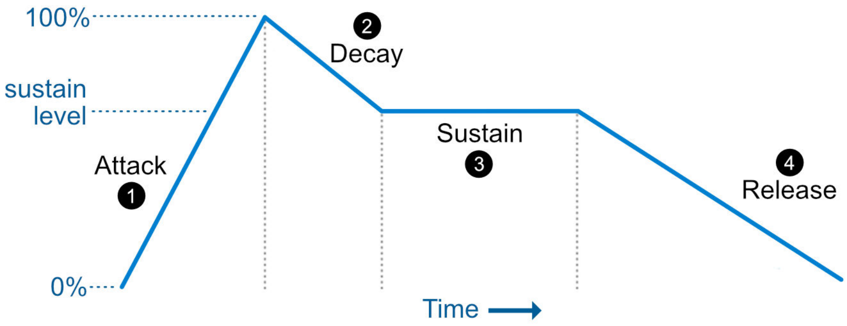
Such a fade-out doesn't have to be very long, anywhere between 10 and 100 samples is usually enough to get rid of the glitches. These kinds of jumps can also happen at the start of the note, so it's a good idea to quickly fade in the sound as well.

Writing code to perform fade-ins and fade-outs is simple enough, but we can do even better: most synths let you define an amplitude envelope that consists of attack (fade in) and release (fade out) stages, but also has additional decay and sustain stages.

Using the envelope not only gets rid of the glitches, it also serves to change the shape of the sound over time, thereby making it more interesting to listen to. The ADSR envelope is a classic design that allows you to realize different envelope shapes that emulate what real instruments do.

What is ADSR?

The first chapter already introduced the envelope used by JX11 but here we'll look at it in more detail. The ADSR envelope consists of four stages:

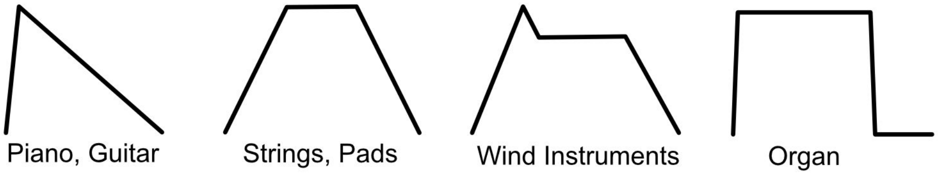


The ADSR stages of the envelope

The attack stage fades in the sound, which is immediately followed by the decay stage dropping the sound level again. Together, these two stages determine a lot of the character of the instrument. To get aggressive, punchy sounds, you'd make the attack and decay very short. Pads or strings, on the other hand, often have a slow attack and little to no decay.

After going through attack and decay, the envelope enters the sustain stage where the sound level stays constant. The synth remains in this stage until a Note Off event is received, at which point the envelope goes into the release stage and the sound fades out. Often the release stage is relatively long as it's natural for tones to fade out rather than stop abruptly.

Some common envelope shapes you can make with an ADSR envelope:



Examples of envelopes

The JX11 envelope uses the following four plug-in parameters that you created in the previous chapter:

- Env Attack – `envAttackParam`
- Env Decay – `envDecayParam`

- Env Sustain – `envSustainParam`
- Env Release – `envReleaseParam`

Env Attack is the time it takes to go from silence to the full amplitude for this note. The amplitude is determined by things such as the note velocity, the oscillator mix, the output level, and possibly other parameters. The envelope doesn't care about any of that. At each time step, it just returns a number between 0 and 1, meaning 0% to 100%, and you'll scale the output from the oscillator by this percentage. No matter what the true amplitude of the tone is, whether it's loud or quiet, the envelope always reaches 100% of that amplitude at the end of the attack stage.

The decay can be specified as the time it takes to go from 100% amplitude down to the sustain level, or as the time from 100% all the way down to 0%. You might think the first option is the most logical, but here the decay speed becomes faster or slower when you change the sustain level. This is why many synths including JX11 use the second option, where **Env Decay** is the rate of change rather than the absolute time. Now the decay speed is always the same, but the length of the decay depends on the sustain level.

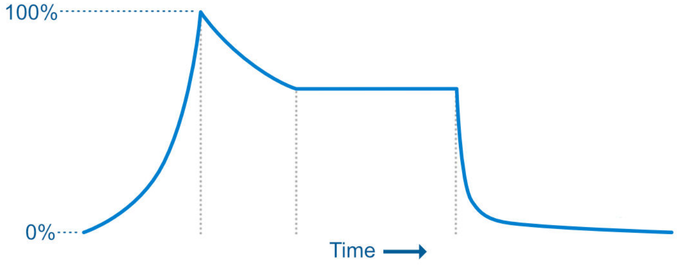
The same is true for **Env Release**. In JX11, this parameter sets the time for the sound level to drop from the maximum amplitude down to zero, not from the sustain level. As a result, the release time will change if you raise or lower the sustain level, but the release speed always stays the same.

I just said attack, decay, and release are times. On many synths, you indeed specify these parameters in milliseconds or seconds. With JX11, however, attack, decay, and release are percentages between 0% and 100%, where 0% represents a "short time" and 100% represents a "long time" — whatever that may mean.

The actual times used by JX11 are on an exponential scale between 0.03 and 68 seconds, but this doesn't really matter. The idea is that as a user, you simply move the sliders for these settings until you like the sound you're getting. Exactly what a setting such as 29% means is not very important. It's quite common for synths to have control knobs that simply go from 1 – 10, which is equally vague. The goal here isn't to be scientific, but to give musicians tools for creating interesting sounds. Ultimately, music is more about feeling than about getting the numbers exactly right.

Unlike the other three envelope parameters, **Env Sustain** is not a time, but a level. It determines how loud the sound is during the steady part. It's important to realize that the sustain level does not change based on the note velocity. For every note you play, no matter how loud or how quiet it is, the sustain level is always the same relative to the note's amplitude. That's why sustain is expressed as a percentage as well.

In the images above, the ADSR sections were drawn with straight lines. However, different curves can be used for each of the stages. For example, something like this:

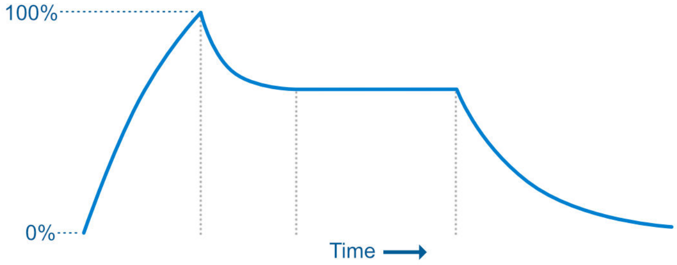


The envelope segments do not have to be straight lines

Recall that our hearing is logarithmic. For loudness, this means that humans are more sensitive to volume changes in quiet sounds than to changes in loud sounds. To make the sound fade in evenly, so that we perceive it as steadily increasing by the same amount, the attack curve should be the opposite of logarithmic, which is exponential. That's what the above image shows.

Likewise, for an even fade-out, the release should be exponential too. In the above image, the release curve has been made extra steep for dramatic effect.

The envelope you'll use for JX11 is a traditional analog ADSR that looks like this:



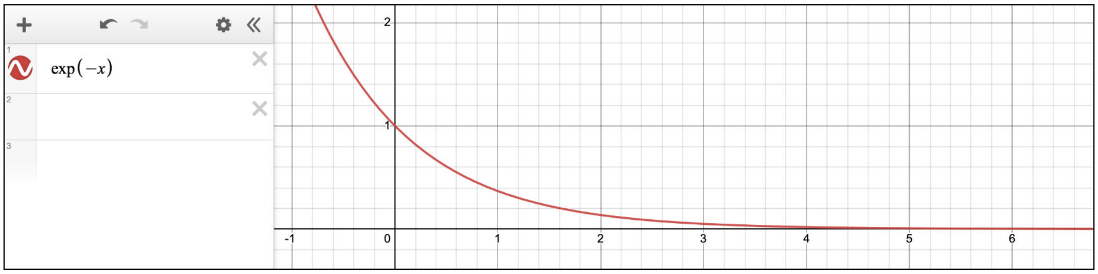
The analog-style ADSR envelope used in JX11

Both decay and release are exponential. The attack curve is also exponential but the "wrong way around". It's not a gradual fade-in but rather more violent. This is a common attack shape for analog synths and it mimics what happens in many acoustic instruments too: the attack of a note is often quite intense — it's called "attack" for a reason!

The practical reason why the curves of analog synth envelopes have this exponential shape, is because they are realized in hardware using capacitors, and this is what the charging (attack) and discharging (decay and release) curves for a capacitor look like. As a bonus, it also sounds good!

The exponential function

Let's start by building a simplified version of the envelope that just does the release part. In math, you can use the function e^x or exp$()$ to describe a simple exponential decay. For example, this is the graph of exp$(-x)$:

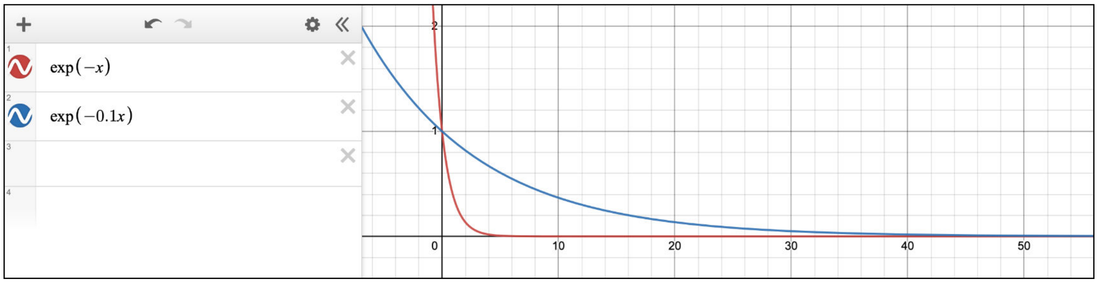


The plot of $y =$ exp$(-x)$

Notice how this graph goes through $y = 1$ at $x = 0$ and then slowly drops down as x increases. This is like an envelope that starts at 100% and then decays towards zero. Interestingly, exp$()$ will never actually become zero — it gets closer and closer but can never quite reach it.

If we say x is the sample index then the above graph looks like it reaches $y = 0$ after about 5 or 6 samples. That's a bit quick for the kind of fade-out that we're after. If we want to hear the sound gradually fading out, the exponential curve should be at least a few hundred or thousand samples long. You can make the exponential curve less steep by writing exp$(-0.1x)$.

I made the above graph using the online graphing calculator Desmos[46]. Try it out, it's a great tool for playing with math equations. Open the Desmos calculator and type in `exp(-0.1x)`. You should now see a shallower version of the exponential curve that takes way longer to reach zero. On macOS there is a built-in app named Grapher you can use for this.

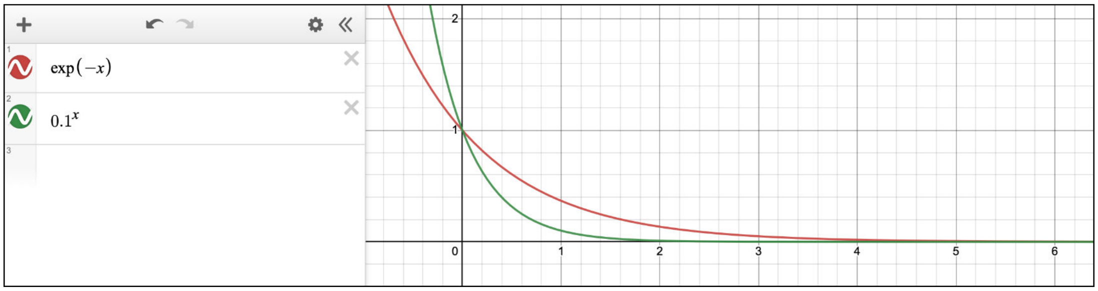


The blue curve is $y =$ exp$(-0.1x)$**, which decays slower than** exp$(-x)$**.**

[46] https://www.desmos.com/calculator

One way you could implement the envelope in code, is to calculate `std::exp(-a * x)` on every time step, where `a` is some small number. The smaller `a`, the longer the decay is. However, calling `std::exp()` is relatively expensive and there is a much simpler method that only involves doing a multiplication.

In the Desmos calculator, add a new curve: `0.1^x`. This creates a similar curve to `exp(-x)`, although it's somewhat steeper:



The plot of $y = -0.1^x$ **is similar to** $y = \exp(-x)$

For the mathematically uninclined, the formula 0.1^x means that on the first timestep $x = 0$, the output value is $y = 1$. That's great because we want to start the envelope at 1 or 100%. On the second timestep, $x = 1$, the result is $y = 0.1$. On the third timestep, the result is 0.1×0.1. On the fourth timestep, it is $0.1 \times 0.1 \times 0.1$. And so on. With each step, you multiply the value by 0.1 again so that it becomes progressively smaller.

Is this really the same as doing $\exp(-x)$? Desmos to the rescue! Change the formula from `0.1^x` to `0.3679^x`. Now instead of multiplying by 0.1 on each timestep, you multiply by 0.3679. You should see that this gives the exact same curve as $\exp(-x)$. That means you can replace the expensive `std::exp()` function using a multiplication with an appropriate number to get an identical curve.

The key here is to choose the correct multiplier. This number needs to be less than 1 and more than 0. If it's exactly 1, nothing happens because multiplying something by one has no effect. If it's more than 1, the envelope level would increase instead of decay. (Try it out in Desmos!)

By the way, if you're wondering where the number 0.3679 came from, this is $\exp(-1)$. That's why it matches the curve of $\exp(-x)$ exactly. To get the curve of $\exp(-0.1x)$, you would write `0.9048^x`, since $\exp(-0.1) = 0.9048$.

Let's put this theory into practice and write some code. You'll use a multiplier of 0.9999, which is equivalent to the curve $\exp(-0.0001x)$. At a sample rate of 44100 samples per second, it takes about two seconds for this curve to drop down from 100% to almost 0%, which makes for a clearly audible demonstration.

Adding the Envelope class

In **Projucer**, add a new file to the project, **Envelope.h**, and replace its contents with the following:

```
#pragma once

class Envelope
{
public:
    float nextValue()
    {
        level *= 0.9999f;
        return level;
    }

    float level;
};
```

The `level` variable holds the current envelope level. It will start at 1.0 or 100%. Every time `nextValue()` is called, `level` is multiplied by the value 0.9999.

Using the `Envelope` class is very straightforward. In **Voice.h**, first include it:

```
#include "Envelope.h"
```

Then, add a member variable to the struct:

```
Envelope env;
```

And change the `render` method to the following:

```
float render()
{
    float sample = osc.nextSample();
    saw = saw * 0.997f + sample;

    // these lines are new
    float envelope = env.nextValue();
    return saw * envelope;
}
```

For every sample that gets rendered, this also asks the envelope what its current level is, and you multiply the oscillator output with the envelope level. That’s all there is to it.

In **Synth.cpp**, add to the bottom of `noteOn`:

```
voice.env.level = 1.0f;
```

This sets the initial level of the envelope to 100% when the new note starts.

Try it out! Play a note and hold it down. Unlike before, where the note sound was sustained indefinitely, now it fades out over the course of about two seconds.

Experiment with smaller and larger multipliers. Note that the difference between 0.999 and 0.9999 is massive! For example: 0.9995 is about half a second, while 0.9998 is roughly one second.

If you release the key before the fade-out has finished, the sound immediately stops. It doesn't keep fading out like you'd expect. You'll fix this properly later, but for now you can comment out the code inside `Synth::noteOff`, so that a MIDI Note Off event no longer ends the note.

Like many analog synths, JX11 allows the user to mix noise into the sound. Drag the slider for the **Noise** parameter to a value larger than 0% and play a new note. Notice that the noise doesn't fade out. That's because you're currently not applying the envelope to the noise, only to the sawtooth. To fix this, in **Voice.cpp**, change `render` to the following:

```
float render(float input)
{
    float sample = osc.nextSample();
    saw = saw * 0.997f + sample;

    float output = saw + input;

    float envelope = env.nextValue();
    return output * envelope;
}
```

The method now accepts an `input` argument, which will receive the current noise sample. You add the oscillator output to this input sample, and then apply the envelope to both.

To make this work, in **Synth.cpp**, change the following lines inside `render`:

```
if (voice.note > 0) {
    output = voice.render(noise);
}
```

Instead of adding the noise to the output from the voice, you let the voice decide how to handle the noise. Try it out. Now the noise will fade out with the rest of the sound.

Calculating the multiplier

It's possible to calculate exactly how large the multiplier should be to achieve a given decay length. First, add the following variable to **Envelope.h**, above the class definition:

```
const float SILENCE = 0.0001f;
```

When the envelope level drops below 0.0001, we will consider the sound to be too quiet to be audible. Recall that an exponential function never drops to exactly zero, so you'll have to set a threshold that you consider to be low enough.

I've mentioned a few times that human hearing is logarithmic. That's why usually a logarithmic scale is used for loudness measurements, the so-called decibel or dB scale. The value 0.0001 corresponds to –80 dB, according to the standard formula $20 \log_{10}(0.0001)$. For all intents and purposes, a sound level of –80 dB is so quiet that it counts as silence.

Our goal now is to calculate the multiplier that is needed to make the function exp$(-x)$ go from 1.0 down to 0.0001 in a certain amount of time.

In the digital audio domain, time is measured in samples. Let's say we want the decay to happen in two seconds. At a sample rate of 44.1 kHz, one second is 44100 samples, so two seconds is 2.0 × 44100 = 88200 samples. Then the multiplier is:

```
multiplier = exp(log(0.0001) / 88200) = 0.9998955798
```

Rounded off that is pretty close to the 0.9999 you used previously, so that explains why indeed each note lasts about two seconds.

The `exp` function you've seen before; the `log` function is the natural logarithm and is the opposite of `exp`. If you're wondering where the minus sign from exp$(-x)$ went, `log(0.0001)` is a negative number, so it's still there.

> **Note:** In equations sometimes `log` is meant as the logarithm with base 10. That is different than the natural logarithm you're using here, which has base e. You can't always tell from the name which `log` it is, so if you see `log` used in a formula make sure that you understand whether it's the log of e, the log of 10, the log of 2, or something else. Otherwise the answers you'll get will be wrong. Considerate authors will write $\log_{10}$ to make it obvious which one they mean, or `ln` for the natural logarithm.

More generally, you can write the multiplier formula as:

```
decayTime = 2.0
decaySamples = sampleRate * decayTime
multiplier = exp(log(0.0001) / decaySamples)
```

It's also common to specify the time in milliseconds rather than in seconds, in which case the calculation becomes:

```
decayTimeInMilliseconds = 2000
decaySamples = sampleRate * decayTimeInMilliseconds / 1000
multiplier = exp(log(0.0001) / decaySamples)
```

What this formula says is: at the given sampling rate, after 2000 milliseconds, the exponential curve has dropped from its starting point at $y = 1.0$ down to $y = 0.0001$.

> **Tip:** Try this out in Desmos. Plot the equation `exp(x * ln(0.2)/10)`. Using big numbers like 2000 milliseconds and 44100 Hz makes it hard to see what's going on, so I've substituted smaller numbers. Don't forget to use `ln` instead of `log` in Desmos You'll get an exponential curve that goes through the value 0.2 at the point $x = 10$. See how that works? The envelope is exactly the same, except that goes through the value 0.0001 at the point `x = decaySamples`.

Let's put these formulas into the program, since the synth already has a parameter for setting the decay time. In **Envelope.h**, add a new variable to the class:

```
float multiplier;
```

and change `nextValue` to the following:

```
float nextValue()
{
    level *= multiplier;
    return level;
}
```

Now you need to fill in this `multiplier` variable somewhere. Since you'll be using the plug-in parameters for this, the correct place the calculate the multiplier is in `JX11AudioProcessor`'s `update` method. Recall that this method is called from `processBlock` whenever a parameter has been changed.

In **PluginProcessor.cpp**, change `update` to the following.

```
void JX11AudioProcessor::update()
{
    float sampleRate = float(getSampleRate());  // 1

    float decayTime = envDecayParam->get() / 100.0f * 5.0f;  // 2
    float decaySamples = sampleRate * decayTime;             // 3
    synth.envDecay = std::exp(std::log(SILENCE) / decaySamples);  // 4

    float noiseMix = noiseParam->get() / 100.0f;
    noiseMix *= noiseMix;
    synth.noiseMix = noiseMix * 0.06f;
}
```

You already added the `noiseMix` lines in the previous chapter. New is the calculation of the decay time and the envelope multiplier. Step-by-step this is what happens:

1. Get the current sample rate. The `getSampleRate` function is part of JUCE's `AudioProcessor` class.
2. Read the value from the **Env Decay** parameter. This returns a percentage, which you'll divide by 100 because we want 100% to be 1.0, not 100. For the sake of this example, we'll say that a setting of 100% corresponds to a decay duration of five seconds, so you multiply by 5 to get the time in seconds.
3. Calculate the number of samples the decay time corresponds to. Since you know the time in seconds, the number of samples is simply the sample rate multiplied by that duration. Note that the time in samples does not have to be a whole number. It's perfectly possible to have a decay time of 9167.83 samples, for example.
4. Apply the `exp()` formula to get the multiplier, and store the result into a new `envDecay` variable in the `Synth` class. You can't set the multiplier on the voice's envelope directly, since the voice is a private member of `Synth`.

In **Synth.h**, add the following public member variable:

```
float envDecay;
```

In **Synth.cpp**, in `noteOn`, add the following line:

```
voice.env.multiplier = envDecay;
```

Try it out! You can now change the length of the fade-out using the **Env Decay** slider. The decay time goes from anywhere between zero and five seconds.

This is already a pretty good envelope, even though it's kind of limited. It only has the decay stage, not attack, sustain, or release. This kind of decaying envelope can be found in specialized synths such as drum synths where sounds do not need to be sustained for a long time.

You may also have found that if you quickly play several notes in succession with different velocities, there is still a clicking or popping sound. This happens because you reset the envelope to 1.0 on every new note so there can still be sudden jumps. You'll fix this soon.

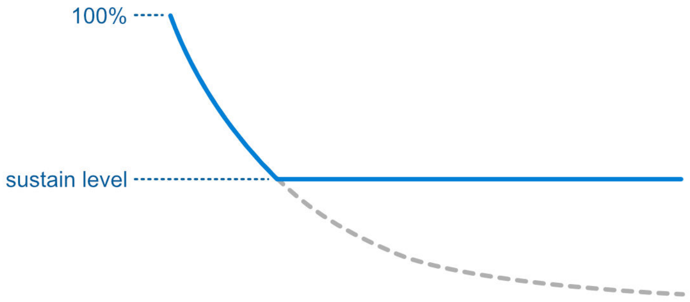
Tip: Because `log(0.0001)` is a constant, you might be tempted to replace `std::log(SILENCE)` with its actual value:

```
synth.envDecay = std::exp(-9.210340372f / decaySamples);
```

That certainly works, but a modern C++ compiler will automatically turn the `log(SILENCE)` into a constant anyway. This generates the exact same code, so you might as well keep the `log` in there. I thought I would point this out since you may come across formulas with `exp()` that use such "magic" constants and you may be wondering what they mean. It's likely the result of calling `log()` on some other, more meaningful number.

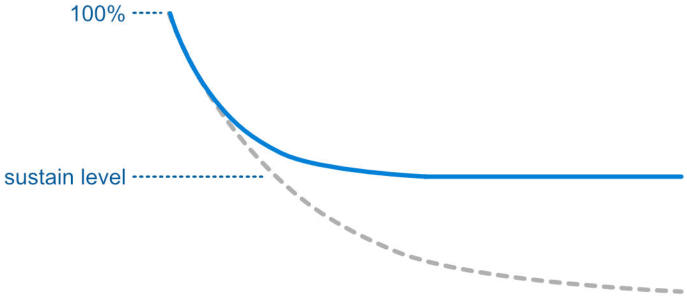
One-pole filter for smoothing

With an ADSR envelope, the decay stage doesn't drop all the way from 100% to zero, it should stop at the sustain level set by the user. There are a few ways to handle this. You could calculate the decay curve as before and simply stop once it has reached the sustain level, like so:



The decay curve abruptly stops when it crosses the sustain level

This is indeed what certain synths do, but remember that sharp transitions can create audible artifacts such as aliases. We'll be a bit smoother about it and make a curve like in the following picture.



The decay curve smoothly transitions into sustain

The actual decay time is still set with respect to a full 100% – 0% drop (the dotted line in the image), so that the decay speed is the same no matter how high or low the sustain level is. But the closer the envelope gets to the sustain level, the more it flattens out. The transition is no longer abrupt but becomes gradual.

These kinds of transitions can be created by a so-called one-pole filter. The one-pole filter naturally outputs exponential curves, making it ideal for the kind of envelope shape we're after. The synth's main filter is covered in more detail in chapter 12, but since filtering is such an important part of audio programming and DSP in general, it won't hurt to do a crash course here.

A filter is a DSP building block that can make certain frequencies in a signal louder and other frequencies quieter. Since each frequency is really a sine wave, all the filter does is make the amplitudes for certain sine waves become larger or smaller — but only if the sine waves with these frequencies are already present in the signal.

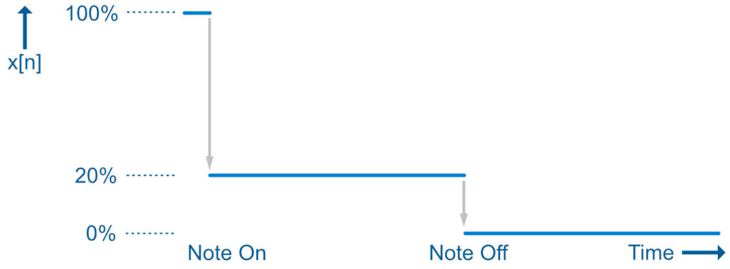
"Louder" and "quieter" are of course only applicable if the thing you're filtering is sound. Here, you'll be filtering the envelope, which itself isn't sound, it's merely used to affect the loudness of the oscillator output. Still, you can think of the curve made by the envelope as a digital signal. After all, you calculate a new value for the envelope level on every time step, so at a sample rate of 44.1 kHz there is also an envelope signal with 44100 numbers per second.

The filter you're going to be using here is an extremely simple low-pass filter that can only decrease frequencies, not increase them. Applying this filter to the envelope will lower the contribution of certain frequencies in the envelope signal. By using the one-pole filter to interpolate between the different envelope levels — 0%, 100%, the sustain level, and back to 0% — you automatically get a signal that is made up of exponential curves.

Maybe that all sounds rather vague, so it might be useful to look at an example to see how this works. For now, we'll ignore the attack stage of the envelope and focus only on the decay and release stages.

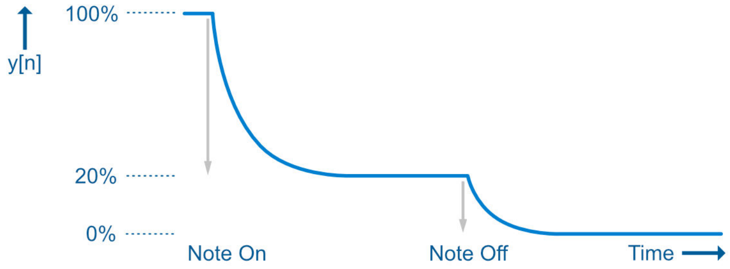
Let’s say we want to decay from 100% down to 20%. The current envelope level is 1.0 and the target level is 0.2. Then after a while, the note stops playing and the release drops the envelope level from 20% all the way to zero.

If you did this without filtering, the envelope signal would look as follows.



The envelope levels without a smooth transition

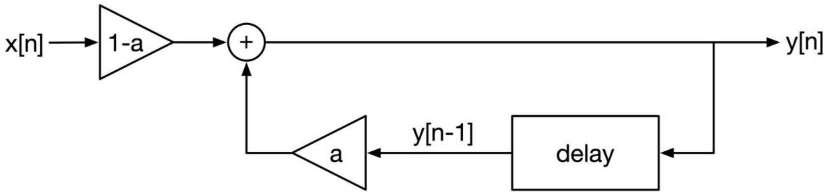
In the chapter on oscillators you’ve seen that these steep transitions require a lot of high frequencies. If you apply a low-pass filter to remove such high frequencies, the transition naturally becomes smoother. The lower you set the cutoff frequency of this filter — in other words, the more frequencies get filtered out — the longer it takes for the signal to decay.



The envelope levels after filtering out the high frequencies

That looks exactly like the sort of envelope shape we’re after! All you need to do to get this kind of shape is create an input signal with discrete envelope levels and apply a low-pass filter, and voila, instant exponential curves.

The one-pole filter you’ll be using is such a low-pass filter. It has a block diagram that looks like this:



The block diagram of the one-pole filter

The input signal is x[n] and describes the discrete envelope levels as shown in the first picture above. This gets mixed with the output of a delay element in a feedback loop, to create the filtered signal y[n] from the second picture.

The length of the delay is one sample, meaning that the delay block outputs the filtered value from the previous timestep. Using delays in feedback loops is a common method for building digital filters, and the design presented here is the simplest possible version.

When describing filters, often a difference equation is given. For this filter that would be:

```
y[n] = (1 - a) * x[n] + a * y[n - 1]
```

Here, x[n] represents the current input value and y[n] is the current output value. That makes y[n - 1] the previous output value. The mixing coefficient is called a, and is a number between 0 and 1. Note that a and (1 - a) will always add up to 1. This difference equation is nothing more than the block diagram expressed in math.

How does this work? The filter will gradually move from whatever its current value y[n] is, to the new value given by x[n]. How fast this happens depends on a. The easiest way to figure out exactly how is to look at some examples.

- If the multiplier a = 0, then y[n] = x[n] since the term with y[n - 1] will not be used. This means the filter output immediately jumps to the new value. This is what happened in the first picture above, and it's the same as not using a filter at all.
- If we say a = 1, then y[n] = y[n - 1]. Now (1 - a) equals zero and any new input is always ignored. This keeps the envelope at whatever level it started out with, it will never change.
- If a is somewhere in the middle, say a = 0.5, then y[n] = 0.5*x[n] + 0.5*y[n - 1]. The computation of the new filter output now uses a little bit from the new input value and also a little bit from the previous output value. With every timestep, the output from the filter will gradually move closer to the target value given by x[n].

The smaller a is, the more the filter uses from x[n] and the less it uses from y[n], so it will take larger steps and will reach the target faster. Conversely, for larger values of a, the slower the filter moves because it takes smaller steps.

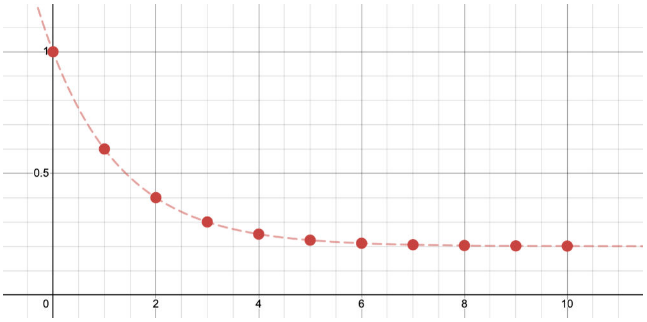
Let's make this example more concrete. After the attack stage, which we're ignoring for now, the envelope level is 1.0. This is the current output y[n]. The user has dialed in the **Env Decay** knob so that the multiplier a = 0.5, and put the **Env Sustain** knob to 20%. With these settings, the envelope should decay from its current level 1.0 down to 0.2. You do this by setting the target level for the envelope x[n] to 0.2.

In all these equations, the symbol `n` represents an arbitrary timestep. Both the signals `x[n]` and `y[n]` are 1.0 initially. As soon as the envelope moves into the decay stage, `x[n]` is set to 0.2 to begin the exponential curve for the decay stage. Now let's examine what these filter calculations do over the next several timesteps.

Timestep	`x[n]`	`y[n - 1]`	`y[n]`
0	1.0	1.0	0.5 × 1.0 + 0.5 × 1.0 = 1.0
1	0.2	1.0	0.5 × 0.2 + 0.5 × 1.0 = 0.6
2	0.2	0.6	0.5 × 0.2 + 0.5 × 0.6 = 0.4
3	0.2	0.4	0.5 × 0.2 + 0.5 × 0.4 = 0.3
4	0.2	0.3	0.5 × 0.2 + 0.5 × 0.3 = 0.25
5	0.2	0.25	0.5 × 0.2 + 0.5 × 0.25 = 0.225
6	0.2	0.225	0.5 × 0.2 + 0.5 × 0.225 = 0.2125
7	0.2	0.2125	0.5 × 0.2 + 0.5 × 0.2125 = 0.20625
8	0.2	0.20625	0.5 × 0.2 + 0.5 × 0.2125 = 0.203125
9	0.2	0.203125	0.5 × 0.2 + 0.5 × 0.203125 = 0.2015625
10	0.2	0.2015625	0.5 × 0.2 + 0.5 × 0.2015625 = 0.20078125

Initially the steps are quite big but then they become smaller and smaller. In particular, with the multiplier `a = 0.5`, the next step is always half the size of the previous step. With every new timestep, the value of `y[n]` gets closer and closer to the target value 0.2.

Plotted as a graph over time it looks like this:



The output of the filter with `a = 0.5`

With each timestep, the distance between the current value and the target value is made smaller. But because the current value itself changes, as represented by `y[n - 1]`, the steps that are taken to close the distance also become smaller over time. This is what creates the exponential shape.

The larger `a` is, the longer it takes for the filter to reach the target value. A larger multiplier therefore means more smoothing. It makes the cutoff frequency of the filter lower — the fewer high frequencies are in the signal, the slower it moves. That's also what happened in the previous section: the multiplier 0.9999 took more time to fade out than 0.9995.

Let's implement this new logic in the `Envelope` class. In **Envelope.h**, add a new public member variable to the class:

```
float target;
```

Change `nextValue` to the following:

```
float nextValue()
{
    level = multiplier * (level - target) + target;
    return level;
}
```

This is the same formula from earlier, written in a more compact way. If you work out the math, you'll see that it's the same as: `level = (1 - multiplier) * target + multiplier * level`. In the signal processing notation used above, `x[n]` is the target and `y[n]` is the newly computed level. The value of `level` on the right-hand side of the statement is the previous value, `y[n - 1]`.

In **Synth.cpp**, add the following line to `noteOn`:

```
voice.env.target = 0.2f;
```

This sets the sustain level to 20%. Try it out. Play a note and immediately release it. The sound still fades out but not all the way to zero. The note keeps playing at 1/5th its full loudness, which is a drop of about 14 dB in volume. You can enable the code in `noteOff` again, so that the note stops playing completely when you release the key.

Nice, you have an envelope that can exponentially decay from an arbitrary starting level down to any target level. This is enough to implement both the decay and release stages of the envelope, but also the attack stage. For a quick preview of how the attack will work, set `env.level = 0.001f` and `env.target = 1.0f` in `noteOn`. Now when you press a key, you'll hear the note fade in!

Note: In the literature you might find the difference equation for the one-pole filter given as `y[n] = a * x[n] + (1 - a) * y[n - 1]` instead of the equation we used, which was `y[n] = (1 - a) * x[n] + a * y[n - 1]`. The difference is that `a` and `(1 - a)` are swapped around. The filter is exactly the same, just formulated differently. The implementation in code becomes `level += multiplier * (target - level)`.

Reading the ADSR plug-in parameters

You’ve already added the `envDecay` variable to `Synth`, so in **Synth.h** add member variables for the other stages too:

```
float envAttack;
float envDecay;   // already have this one
float envSustain;
float envRelease;
```

The values for these properties need to be derived from their respective plug-in parameters. In **PluginProcessor.cpp**, change `update` to the following:

```
void JX11AudioProcessor::update()
{
    float sampleRate = float(getSampleRate());
    float inverseSampleRate = 1.0f / sampleRate;

    synth.envAttack =
        std::exp(-inverseSampleRate * std::exp(5.5f - 0.075f * envAttackParam->get()));

    synth.envDecay =
        std::exp(-inverseSampleRate * std::exp(5.5f - 0.075f * envDecayParam->get()));

    synth.envSustain = envSustainParam->get() / 100.0f;

    float envRelease = envReleaseParam->get();
    if (envRelease < 1.0f) {
        synth.envRelease = 0.75f;  // extra fast release
    } else {
        synth.envRelease = std::exp(-inverseSampleRate * std::exp(5.5f - 0.075f * envRelease));
    }

    float noiseMix = noiseParam->get() / 100.0f;
    noiseMix *= noiseMix;
    synth.noiseMix = noiseMix * 0.06f;
}
```

The sustain level is simply a percentage so you divide that by 100 to get a value from 0 – 1. However, the formulas for the multipliers are a little different than before. Previously, you did the following to calculate the multiplier:

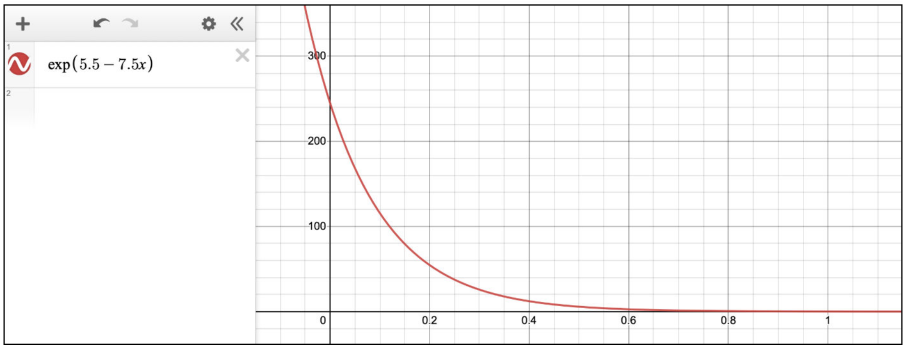
```
multiplier = exp(log(SILENCE) / decaySamples)
```

But now the formula is:

```
multiplier = exp(-inverseSampleRate * exp(5.5 - 0.075 * parameter))
```

Attack, decay, and release all use the same formula, except release has an extra fast multiplier of 0.75 if the slider is set to 0%. This setting will fade out the sound over 32 samples, since $(0.75)^{32} = 0.0001$, which is the `SILENCE` level. Recall that a short fade-out is better than having no release at all, because a sudden jump is likely to create a glitch.

How does this new formula work? Well, it's still basically the same formula but rewritten somewhat. The parameter is a value from 0% to 100%, so we can rewrite the inner part as exp$(5.5 - 7.5x)$ where x is a value from 0 to 1. Plotting this curve with Desmos:



The plot of exp$(5.5 - 7.5x)$

The part that's interesting to us is between $x = 0$ and $x = 1$. This is a very steep exponential curve. At $x = 0$, the value is exp$(5.5) = 244.691925$. At $x = 1$, or 100%, the value is exp$(-2) = 0.135335$. But what do these values mean? If we fill them into the formula, using a sample rate of 44.1 kHz, we get:

```
multiplier = exp(-244.691925 / 44100) = 0.994467
multiplier = exp(-  0.135335 / 44100) = 0.999997
```

This makes sense: the larger the multiplier, the shallower the exponential curve is, and the longer it takes to fade out. However, we can also do some math to calculate exactly how long this decay time is. Feel free to skip the next page if seeing all this math makes you nauseous.

Recall that the original formula that we used was,

```
multiplier = exp(log(SILENCE) / (sampleRate * decayTime))
```

while here we have:

```
multiplier = exp(-exp(5.5 - 7.5 * x) / sampleRate)
```

Therefore, we know that:

```
log(SILENCE) / decayTime = -exp(5.5 - 7.5 * x)
```

If you're having trouble seeing this, we're trying to find out which parts both formulas have in common, and which are different. Both formulas do the following:

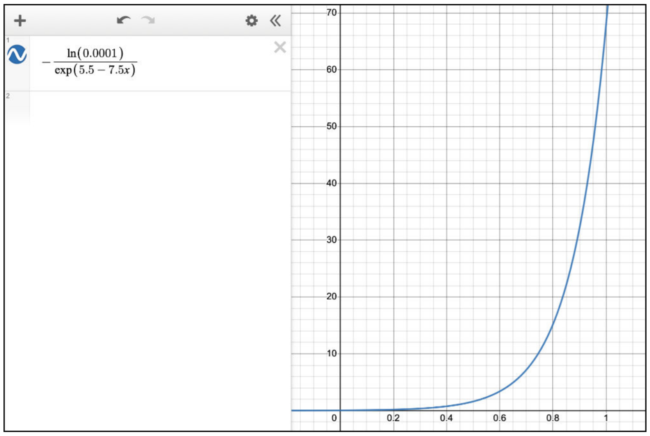
```
multiplier = exp(something / sampleRate)
```

In the new formula that "something" is `-exp(5.5 - 7.5 * x)`. The same spot in the first formula says `log(SILENCE)`, but notice that the denominator hasn't just `sampleRate` but also `decayTime`. Therefore, the "something" in the first formula is really `log(SILENCE) / decayTime`.

The goal here is to figure out how all of this relates to the `decayTime`. We can reorder the terms to get an equation, and then fill in different values for `x`:

```
decayTime = -log(SILENCE) / exp(5.5 - 7.5 * x)
```

From this, we can conclude that the decay time ranges from 0.03764 seconds when the parameter is 0% (or `x = 0`), to 68.056 seconds when the parameter is 100% (or `x = 1`). This is how to long it takes to decay from full amplitude to the silence level (0.0001) for the minimum and maximum parameter settings.



$$-\frac{\ln(0.0001)}{\exp(5.5-7.5x)}$$

The possible range of decay time values plotted in Desmos

Notice that at a setting of 0%, the decay time is not actually zero seconds, which is a good thing because we want to at least have short fade-in and fade-outs to avoid clicks and pops.

Why go through all this trouble with this exp$(5.5-7.5x)$ formula? This again is an example of skewing the parameter so that smaller values are easier to choose. The slider goes in equal steps but 50% does not mean half the time of 100%:

```
slider at 0% = 0.0376 seconds
slider at 25% = 0.2454 seconds
slider at 50% = 1.601 seconds
slider at 75% = 10.437 seconds
slider at 100% = 68.056 seconds
```

I'm not sure how the original author of this synth arrived at these particular values, but they probably picked something by hand that seemed like a decent curve. Still, 68 seconds seems like a very long decay time. I probably would have picked something between 0 and 30 seconds at most, and even that seems long. In this sense, synthesizer design is more art than science.

Could you have achieved this by changing how the parameter was defined, using a skew factor? Something like the following perhaps:

```
layout.add(std::make_unique<juce::AudioParameterFloat>(
    ParameterID::envDecay,
    "Env Decay",
    juce::NormalisableRange<float>(30.0f, 30000.0f, 0.01f, 0.25f),
    1500.0f,
    "ms"));
```

Absolutely! This would define a parameter that ranges from 30 milliseconds to 30 seconds, with a default of 1.5 seconds. Its skew factor is set to 0.25 so that the default is roughly in the center position of the knob. In your own synths, please feel free to implement it like this. After reading the time in milliseconds directly from the parameter, you can calculate the multiplier for the one-pole filter using the formulas given earlier in this chapter.

So why did we do it the hard way? The formulas with `exp` and `log` show up a lot in DSP and it's good to get your hands dirty with the math from time to time. I don't want you to be afraid of the math! If you're still not entirely sure what we did here, please give it some time to sink in and play around with the curves in Desmos. You don't need to be a math wizard to write synths, but understanding how these exponential curves work will get you a long way.

Implementing the new formula

With the multipliers sorted, you can now implement the rest of the envelope. For the time being, we'll skip the attack stage and just deal with the other three stages.

In **Envelope.h**, add the following member variables. These are all public:

```
float attackMultiplier;
float decayMultiplier;
float sustainLevel;
float releaseMultiplier;
```

Also add a `reset` function to the `Envelope` class:

```
void reset()
{
    level = 0.0f;
    target = 0.0f;
    multiplier = 0.0f;
}
```

In **Voice.h**, inside `Voice`'s own `reset`, add the following line to reset the envelope:

```
env.reset();
```

In **Synth.cpp**, in `noteOn`, do the following:

```
Envelope& env = voice.env;
env.attackMultiplier = envAttack;
env.decayMultiplier = envDecay;
env.sustainLevel = envSustain;
env.releaseMultiplier = envRelease;

env.level = 1.0f;
env.target = env.sustainLevel;
env.multiplier = env.decayMultiplier;
```

This first copies the multiplier values and sustain level into the `Envelope` object, and then triggers the decay stage. Try it out. The decay stage works as before but now you can also change the sustain level. At 100%, the default setting for **Env Sustain**, the decay stage is completely turned off. The lower you set the sustain level, the longer the decay is because the envelope needs to drop further.

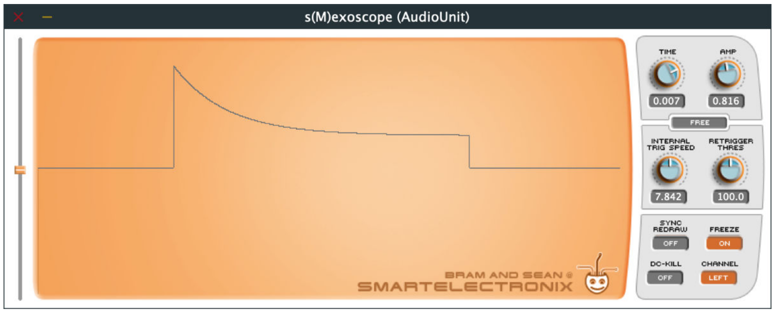
There technically isn't a separate sustain "stage" in this envelope implementation. Since exponential decay never truly reaches its target, as long as you hold down the note, the envelope remains in the decay stage. We think of it as the sustain stage, but it's nothing more than the output from the one-pole filter being approximately the same as the sustain level.

> **Note:** In this synth, we assign the envelope properties to the voice when the note starts playing. As a result, changing the envelope parameters while a voice is playing will have no effect on that voice. Try it out: play a note and while holding it down, change the sustain level. The note will not become louder or quieter. Different synths handle this in different ways — in some synths you can even change the attack, decay, or release time while the note is in that state.

At this point I'd like to show you a little trick for debugging the envelope, just in case it's giving you problems. In `Voice::render`, change the return statement to:

```
return envelope;  // instead of output * envelope
```

Instead of applying the envelope to the output of the oscillator, you output the envelope value itself. This will not play any sound (except for clicks and pops because technically you're now glitching the audio), but you can still capture it on the oscilloscope or in your DAW and inspect it:



TIME 0.007
AMP 0.816
INTERNAL TRIG SPEED 7.842
RETRIGGER THRES 100.0
SYNC REDRAW OFF
FREEZE ON
DC-KILL OFF
CHANNEL LEFT

Debugging the envelope curves in the oscilloscope

I find this works best if you zoom out the oscilloscope, both in time and in amplitude. By the way, if the popping sounds bother you when doing this, simply disconnect the plug-in from the audio output in AudioPluginHost or mute the track in your DAW. You can look at the output that the plug-in generates, without necessarily having to hear it. Until they get sent to the DAC, these audio buffers are nothing more than a bunch of numbers.

The release stage

Use the **Env Release** slider to set the release time of the envelope to one second or so. Play a note and immediately let go. The note will abruptly stop playing. That's no good… we want the sound to properly fade out.

`Synth::noteOff` currently sets `voice.note` to 0, so that the loop in `render` stops rendering the voice. The solution is to no longer do this and instead tell the envelope to go into the release stage when a Note Off message is received.

In **Envelope.h**, add the following method to `Envelope`:

```
void release()
{
    target = 0.0f;
    multiplier = releaseMultiplier;
}
```

Initially, the `target` is the `sustainLevel` and the multiplier is the one for the decay time. As soon as `release()` is called, this changes the `target` to 0 and uses the multiplier for the release time.

In **Voice.h**, give the `Voice` struct its own `release` method. For now, this merely passes on the request to the envelope:

```
void release()
{
    env.release();
}
```

In **Synth.cpp**, change `noteOff` to:

```
void Synth::noteOff(int note)
{
    if (this->note == note) {
        voice.release();
    }
}
```

Try it out! Now when you release the key, the note does not immediately stop but slowly fades out.

There's something else you should do. The release curve will never reach zero, but it will eventually cross the value 0.0001 (the level we called SILENCE). That's a good opportunity to stop the voice from playing, since you won't be able hear it anymore anyway. There's no point calculating the next value for the oscillator or any other work we might be doing to produce the sound if it's inaudible.

In **Envelope.h**, add a function that returns true if the envelope level is still high enough:

```
inline bool isActive() const
{
    return level > SILENCE;
}
```

Then in **Synth.cpp**, change the code inside render to the following:

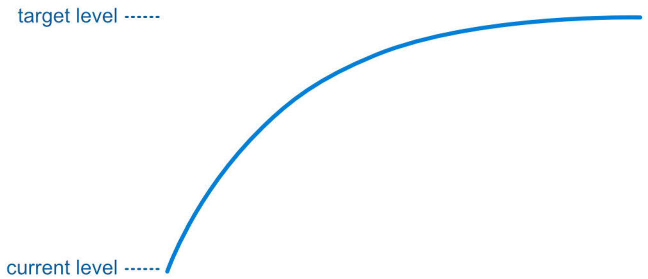
```
for (int sample = 0; sample < sampleCount; ++sample) {
    float noise = noiseGen.nextValue() * noiseMix;
    float output = 0.0f;

    if (voice.env.isActive()) {   // change this line
        output = voice.render(noise);
    }
    // ...
}

// add these lines
if (!voice.env.isActive()) {
    voice.env.reset();
}
```

This only renders the voice if the envelope is still active. Knowing when a voice is no longer used is important for when you implement polyphony in chapter 10. The voice stealing logic will look at which voices are not currently playing and it uses the envelope's active state for this.

> **Note:** Another reason to stop the voice when its envelope finishes is denormals. If floating point values become too small, they turn into what's known as denormal or subnormal numbers. On many CPUs such denormal values are handled in software and are therefore very slow. This can result in a plug-in suddenly spiking in CPU usage when the sound is fading out, as eventually the output from the envelope will be so small the numbers get into denormal territory. Fortunately, the JUCE ScopedNoDenormals object stops this from happening, so for our synth these denormal numbers won't be a problem.

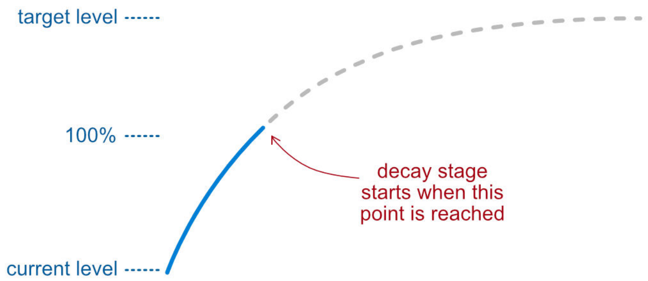
The attack stage

And now for the final part: the A in ADSR, the attack stage. Revisiting the one-pole filter, if you set the target value `x[n]` higher than the current value `y[n]`, the following kind of curve will be created by the filter:



The one-pole filter can also create rising curves

That already kind of looks like the desired attack curve but doing it this way sounds too tame. We want the attack of the sound to have a bit of punch. JX11 applies a simple trick: instead of setting the target to 1.0, set it to 2.0. That makes the curve a lot steeper. Of course, we can't let the envelope actually reach 2.0 since 1.0 is the maximum.



A better attack

One new thing to consider is that the attack proceeds on its own accord until the envelope level reaches 100% and then the envelope should switch to the decay stage automatically. So far, you've started the envelope in decay and let it run until the user stopped the note, where you used the MIDI Note Off event to trigger the release state. But with the attack-to-decay transition there is no user event. Instead, the envelope will need to keep track of being in the attack stage and when the level exceeds 1.0, switch to the decay stage.

If you've been programming for a while, this might sound suspiciously like a job for a state machine. And you'd be right: it's common to implement an envelope as a state machine where each stage corresponds to a state, and let the state machine handle the transitions between the stages. It's definitely a suitable design pattern for an ADSR. However, we can get by without having to make an explicit state machine for our envelope.

To create the steeper attack, you set the target level to 2.0. So, you know that if the target is 2.0, the envelope is still in the attack stage. The decay stage's target level is 1.0 or less (equal to the sustain level), and the release stage's target level is always 0. To check whether the attack is finished, you could write:

```
if (target == 2.0f && level >= 1.0f) {
    /* switch to decay */
}
```

That would work, but an even quicker way is the following. Change `Envelope`'s `nextValue` method to:

```
float nextValue()
{
    level = multiplier * (level - target) + target;

    if (level + target > 3.0f) {
        multiplier = decayMultiplier;
        target = sustainLevel;
    }

    return level;
}
```

Once the combined value of the target and the current level is greater than 3.0, switch to the decay multiplier and change the target to the sustain level. (Why 3.0? The target is 2.0 and the level should be more than 1.0, so summed up that is 3.0 or more.)

Also add the following two methods:

```
inline bool isInAttack() const
{
    return target >= 2.0f;
}

void attack()
{
    level += SILENCE + SILENCE;
    target = 2.0f;
    multiplier = attackMultiplier;
}
```

This sets the target level to 2.0 and activates the multiplier for the attack time. The `isInAttack` method isn't used right now but will come in handy later.

The statement `level += SILENCE + SILENCE` is perhaps less obvious. This gives the envelope a little boost so that the initial envelope level is always greater than `SILENCE`. In the render loop, `isActive()` will now return true, guaranteeing that the voice will be seen as active and be rendered. Without this line, new notes would never play!

Note that this does `+=` instead of a regular assignment. This enables legato-style playing: it's possible that this voice is told to play a new note even though the previous note hasn't finished yet. In that case, we want to continue from the current envelope level, not restart it from zero.

Make the `target` and `multiplier` variables private, since they should no longer be accessed from outside this class:

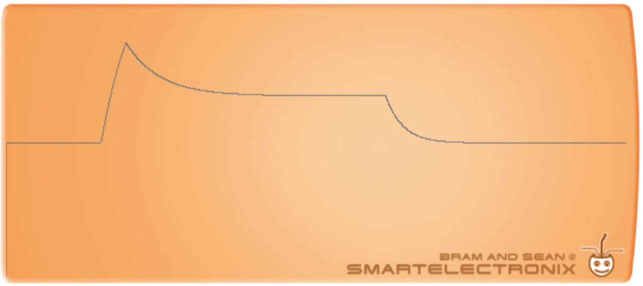
```
private:
    float target;
    float multiplier;
```

Users of the `Envelope` class don't need to know that a target of 2.0 means the envelope is in the attack stage. So it's good to hide these inner workings. However, `level` should stay public, so other objects can read the current envelope level without having to call `nextValue()`.

Back in **Synth.cpp**, in `noteOn`, change the code that fires off the envelope to just these lines:

```
Envelope& env = voice.env;
env.attackMultiplier = envAttack;
env.decayMultiplier = envDecay;
env.sustainLevel = envSustain;
env.releaseMultiplier = envRelease;
env.attack();
```

Try it out! The envelope should have a proper attack now. If you use the debugging trick again and output only the envelope value, the oscilloscope should show something like this:



The full ADSR curve in the oscilloscope

That's exactly the shape we were aiming for. Note that for the attack, the duration is shorter than for decay and release, even though you used the same formula to calculate the multiplier. This is of course because the attack curve is steeper.

Conclusion

Great! That's another building block for the synth complete. Thanks to the envelope you've eliminated click and pops from the sound, and you've given the user an important tool for shaping the sounds coming out of the oscillator.

The analog-style envelope you've built in this chapter works well, but it's a basic design. These days many synths allow you to construct envelopes with an arbitrary number of stages and curve types. That may seem a lot more complicated than what you've done here, but in the end an envelope is just some object that outputs a value between 0 and 1 at each timestep.

If you're thinking about designing your own envelope, keep in mind that there are all kinds of edge cases to deal with. For example, what happens when you play a new note while the previous one is still fading out?

The JX11 envelope always starts the attack from the current level — in other words, it continues from where it left off — so that there's no big jump when this happens. There are synths that will reset the envelope to zero on the new note, but that requires a much more complicated envelope: to avoid glitching, you first need to rapidly fade out the old note to zero before you can start the new note.

Another edge case: What happens when a Note Off event is received when the envelope is still in the attack or decay stages? In that case, the JX11 envelope will immediately jump to the release stage and begin its fade-out from whatever the current level is. Other synths may decide to first finish the attack and decay and only then go into release.

Or what happens if the decay time is zero? Then the synth should immediately go into the sustain stage after the attack. What if both attack and decay are zero? And so on… The advantage of the "analog" implementation that simulates the charging and discharging of capacitors, is that we don't have to worry about these situations!

Fair enough, the analog-style envelope isn't perfect. If you use a very short attack the sound can still glitch because of that DC offset at the beginning of the sawtooth oscillator. Plus, the envelope doesn't know anything about the velocity of the notes. If you play a note that is really quiet and interrupt it with a new note that is loud, there will still be a jump in the waveform.

This isn't the fault of the envelope — there's no jump in the envelope's curve — but occurs because the amplitudes of the two notes are so different. A possible solution is to quickly crossfade between these two different amplitude levels.

I should also mention that JUCE has its own built-in envelope class, `juce::ADSR`. This is a simple envelope with linear curves, but its source code is a good example of how to build an envelope using a state machine. For bonus points, check out the source of the JUCE class[47] and compare it to the envelope from this book.

[47] https://github.com/juce-framework/JUCE/blob/master/modules/juce_audio_basics/utilities/juce_ADSR.h

Chapter 9: Combining oscillators

What’s better than one oscillator? Two oscillators! Adding a second oscillator to the synth vastly expands the palette of sounds that JX11 can produce.

In **Voice.h**, replace the line:

```
Oscillator osc;
```

with these two lines:

```
Oscillator osc1;
Oscillator osc2;
```

Likewise, in `reset` write:

```
osc1.reset();
osc2.reset();
```

And in `render` do the following:

```
float render(float input)
{
    float sample1 = osc1.nextSample();
    float sample2 = osc2.nextSample();
    saw = saw * 0.997f + sample1 - sample2;

    float output = saw + input;
    float envelope = env.nextValue();
    return output * envelope;
}
```

This method is mostly still the same as before, except now there are two oscillators that each produce their own sample value. The output from the second oscillator is subtracted from the first.

JX11 has an **Osc Mix** parameter that determines how much the second oscillator is mixed into the final sound. This parameter ranges from 0% to 100%. To mix the two oscillators together, you will make the amplitude of `osc2` a percentage of `osc1`'s amplitude. This happens in the `Synth` class.

First, in **Synth.h**, add a new variable to the class. This goes into the public section because it will be filled in by the audio processor.

```
float oscMix;
```

In **Synth.cpp**, edit `noteOn` with the following changes:

```
void Synth::noteOn(int note, int velocity)
{
    voice.note = note;

    float freq = 440.0f * std::exp2(float(note - 69) / 12.0f);

    // activate the first oscillator
    voice.osc1.period = sampleRate / freq;
    voice.osc1.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc1.reset();

    // activate the second oscillator
    voice.osc2.period = voice.osc1.period;
    voice.osc2.amplitude = voice.osc1.amplitude * oscMix;
    voice.osc2.reset();

    Envelope& env = voice.env;
    env.attackMultiplier = envAttack;
    env.decayMultiplier = envDecay;
    env.sustainLevel = envSustain;
    env.releaseMultiplier = envRelease;
    env.attack();
}
```

The second oscillator gets the same period as the first one, but a potentially smaller amplitude that is scaled by `oscMix`. Naturally, you must still give this `oscMix` variable the value from the plug-in parameter, which happens in `update`.

In **PluginProcessor.cpp**, in `update`, add the following line. Since this parameter is a percentage, you divide it by 100 to put it into the range 0 – 1.

```
synth.oscMix = oscMixParam->get() / 100.0f;
```

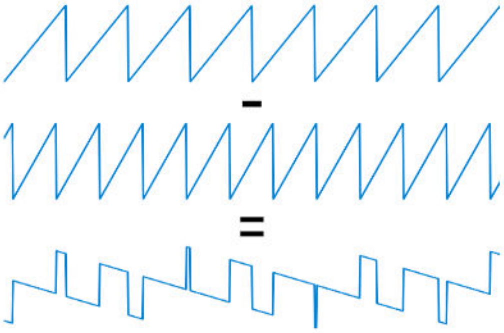
Try it out! Set the **Osc Mix** parameter to 100:0% and play a note. You should hear the usual sawtooth wave. Now slide the parameter to 75:25% and play a new note. What you hear is… a quieter sawtooth wave?! Drag the slider all the way to the right, to 50:50%. Play a note and you'll hear absolutely nothing. Uh, that doesn't seem like a very useful feature!

Perhaps what happened here doesn't surprise you in the slightest but I thought it was odd when I first experienced the sound disappearing. However, it makes perfect sense. You created two oscillators with the same period, so both `osc1` and `osc2` output exactly the same sawtooth wave, especially if they also have the same amplitude. Because you subtract the `osc2` waveform from `osc1`, nothing is left over. Both oscillators cancel each other out… doh!

One way to fix this is to give the second oscillator a different starting phase. That way, `osc2` is shifted in time and the two sawtooth waves interact differently. Sometimes they boost each other and sometimes they cancel out. Adjusting the `phase` variable of the second oscillator would work, but instead I want you to comment out the following two lines from `noteOn`:

```
//voice.osc1.reset();
//voice.osc2.reset();
```

This by itself won't fix the issue, but in combination with the next step, it will help to make the phase interactions less predictable. That next step is to detune the second oscillator so that it runs at a slightly different pitch than the first oscillator. This will create a much more interesting interference pattern. Here is a somewhat exaggerated example:



Combining two oscillators at different pitches creates more complex waveforms

Since the synth no longer resets the phase of the oscillators when a new note is started, the oscillators will first finish their current cycle before they switch to the pitch of the new note. This was a feature of the oscillator you built in the sawtooth chapter — it can't change the phase increment until it reaches the next impulse peak.

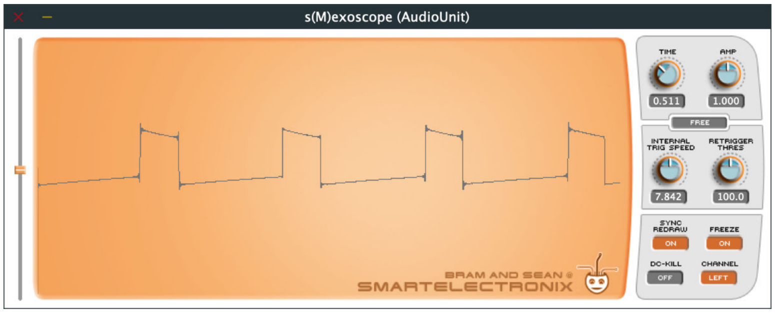
Because the second oscillator is detuned, it runs a bit faster or slower than the first oscillator, and therefore it will activate its new period some time before or after the first oscillator does so. As a result, the oscillators will always have a certain amount of phase shift relative to each other, and the amount changes with every new note you play.

We want this phase difference! It makes the sound fuller and less predictable. Although, as you'll find, sometimes it can be a little too unpredictable and make the phases align in such a way that the sound will completely disappear like before. When I mentioned this synth has quirks, that's one of them!

Let's experiment with this idea before doing the full implementation. In `Synth::noteOn`, change the line that sets the second oscillator's period to:

```
voice.osc2.period = voice.osc1.period * 0.994f;
```

This makes the frequency of `osc2` higher than `osc1` by 1/10th of a semitone or 10 cents. The period is shorter, so the pitch goes up. Try it out! As you slide **Osc Mix** from 100:0% towards 50:50%, you'll notice that the second oscillator starts to affect the sound. At 50:50% it gives a pretty wild PWM (pulse width modulation) type of effect:



TIME 0.511
AMP 1.000
FREE
INTERNAL TRIG SPEED 7.842
RETRIGGER THRES 100.0
SYNC REDRAW ON
FREEZE ON
DC-KILL OFF
CHANNEL LEFT

The two sawtooth waves combine to create a modulated square wave

Just for fun, uncomment the lines that reset the oscillators and try out the difference this makes. Now whenever you play the same note a few times in a row, the sound will be exactly the same too. Remove those lines again so that the oscillators do not get reset. Repeat the note and notice that the sound is somewhat different each time.

Which one you prefer is a matter of taste, and this is one of the many design decisions you need to make as a synth designer. For a virtual analog synth like JX11, it's perfectly acceptable to never reset the phase. In analog hardware, the oscillators are always running, whether a note is being played or not. Some synths have an option whether to reset the oscillators on a new note. Personally, I like JX11 better when it never resets the phase, but feel free to experiment for yourself and see what you prefer.

By the way, the **Osc Mix** parameter only affects new notes. If you press and hold a note and then drag the slider, the sound does not change. This is also a design decision. The same is true for the envelope parameters, such as **Env Sustain**. Changing this does not make the currently playing note any louder or quieter. Different synths do this in different ways.

JX11 does have certain parameters that affect the playing notes. The detuning settings, which you're going to look at next, are an example of that.

Detuning the oscillator

Rather than always using a hardcoded value of 0.994 like you did in the previous section, JX11 lets the user choose the amount of detuning for the second oscillator. There are two parameters for this:

- **Osc Tune**, for detuning in semitones
- **Osc Fine**, for detuning in cents

As I am sure you are aware, a semitone is the distance between two notes: from A to Bb is one semitone up, while A to G# is one semitone down. An octave is 12 semitones. The **Osc Tune** parameter goes from –24 to +24, so it can detune the second oscillator by two octaves. The default is –12 semitones, meaning one octave down. A cent is 1/100th of a semitone, which allows **Osc Fine** to make small adjustments to the tuning.

In **Synth.h**, add a new public member variable to the `Synth` class:

```
float detune;
```

This combines both **Osc Tune** and **Osc Fine** parameters into a single value. You will multiply the period of the second oscillator with this `detune` variable. Tuning down gives a value for `detune` that is greater than 1, while tuning up makes the value smaller than 1. If no detuning is applied, `detune` should be exactly 1, so that multiplying by this number will not change the period.

In **PluginProcessor.cpp**, in `update`, add the following lines to read the parameters:

```
float semi = oscTuneParam->get();
float cent = oscFineParam->get();
synth.detune = std::pow(1.059463094359f, -semi - 0.01f * cent);
```

The first two lines are simple enough but the last one needs some explanation. **Osc Tune** is already in semitones but **Osc Fine** is in cents. To get the total amount of detuning in semitones, you do `semi + cent/100`, or written slightly differently, `semi + 0.01*cent`.

In the oscillators chapter you saw that you can calculate the pitch of any note by taking a starting pitch and multiplying it by $2^{N/12}$, where N is the number of (fractional) semitones. That is exactly what is being calculated here, even though this might not be obvious at first sight.

Another way to write this calculation is as follows,

```
synth.detune = std::pow(2.0f, (-semi - 0.01f * cent) / 12.0f);
```

or like what you did in chapter 5, using `std::exp2` instead of `std::pow`:

```
synth.detune = std::exp2((-semi - 0.01f * cent) / 12.0f);
```

Since $2^{1/12} = 1.059463094359$, you can write $2^{N/12}$ as 1.059463094359^N. All these formulas calculate exactly the same answer. We're merely simplifying the math to avoid doing this extra division by 12.

You may be wondering why the formula says `-semi - 0.01*cent` instead of `semi + 0.01*cent`. Recall that in JX11, we don't specify the oscillator's frequency but its period. To make the pitch lower, you must increase the period. Conversely, to get a higher pitch, the period should become smaller. If the frequency must go up, the length of the period must go down, and vice versa.

You can do this either by dividing the period by $2^{N/12}$, or alternatively, by multiplying the period by $2^{-N/12}$. Mathematically these two operations come down to the same thing, but in software we prefer multiplying over dividing because it's often faster. Notice that there is a minus sign in front of N when multiplying. That's why the number of semitones in the formula for `synth.detune` is specified as a negative amount.

Whenever I come across such a formula I always find it useful to calculate what the maximum and minimum possible values are that it can produce. The maximum to detune downwards is –24 semitones and –50 cents, or 24.5 semitones in total, which is $2^{24.5/12} = 4.12$. The maximum to detune upwards is +24 semitones and +50 cents, or $2^{-24.5/12} = 0.242$.

Therefore, `synth.detune` is some value between 0.242 and 4.12. It roughly makes the period of the second oscillator four times larger or four times smaller than the period of the first oscillator. Makes sense, since that's indeed about two octaves up or down.

In **Synth.cpp**, in `noteOn`, change the line that sets `osc2`'s period, to the following:

```
voice.osc2.period = voice.osc1.period * detune;
```

Try it out! You can now use the **Osc Tune** and **Osc Fine** sliders to change tuning for the second oscillator and create all kinds of exciting new timbres.

One thing that doesn't work yet is changing the tuning while a sound is playing. Let's fix this. You already read the values of all the plug-in parameters in `update`. This happens at the top of the audio processor's `processBlock` method, so whenever the user changes **Osc Tune** or **Osc Fine**, the synth sees these new values in the very next block that gets rendered.

However, right now the `detune` variable is only read when a Note On event occurs, so it's purely used for new notes. Instead, you will have to update the period of the oscillators at the start of each new block, so that if the detuning parameters have changed in the meantime, the new value of `detune` will get picked up by the currently playing voices too.

To make this a little easier to manage, in **Voice.h** add a new property to the `Voice` struct:

```
float period;
```

In **Synth.cpp**, in `noteOn`, make the following changes to the lines that set the oscillator properties. You're no longer directly setting `osc1.period` and `osc2.period` here, only the new `voice.period` property.

```
float freq = 440.0f * std::exp2(float(note - 69) / 12.0f);
voice.period = sampleRate / freq;
voice.osc1.amplitude = (velocity / 127.0f) * 0.5f;
voice.osc2.amplitude = voice.osc1.amplitude * oscMix;

// remove these lines:
voice.osc1.period = sampleRate / freq;
voice.osc2.period = voice.osc1.period;
```

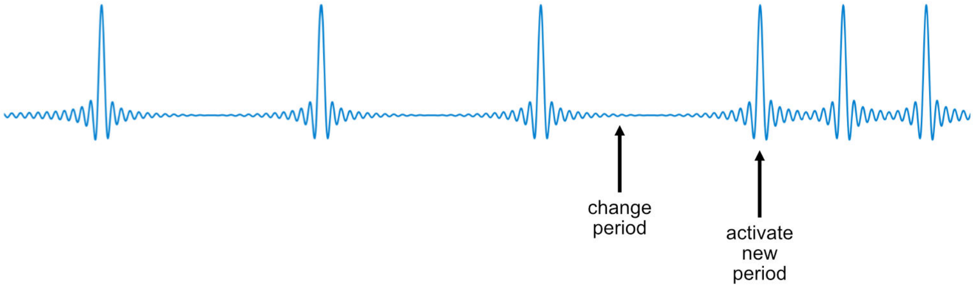
Then in `Synth`'s `render`, just before the `for` loop, add these two lines:

```
voice.osc1.period = voice.period;
voice.osc2.period = voice.osc1.period * detune;
```

This is where you put the period into the actual oscillators and apply the detuning to the second oscillator. Now any changes to `detune` will be automatically picked up by the voice if it's already playing. The value of `detune` won't change during the current block — it can only change at the start of the next block in `update` — so reading this value once before the render loop is sufficient.

Note: The value of `voice.period` doesn't change when the note is playing, so it's a little silly to keep assigning the same value over and over to `voice.osc1.period`. However, later in this chapter you will add pitch bending, which will modify the period of `osc1` too.

Remember, due to the way `Oscillator` is implemented, setting its `period` property does not immediately change the pitch of the oscillator. Only after the current cycle is complete does `Oscillator` look at the `period` property again to calculate the new phase increment for the next cycle. So you can safely set the new period at any time and `Oscillator` will pick it up on the next cycle.



The old period continues until the end of the cycle

If the second oscillator is detuned, updating the phase increment happens at a slightly different time than for the first oscillator, which causes the phases of both oscillators to constantly change relative to each other.

Try it out! Start the plug-in but don't play any notes yet. First, make both **Osc Tune** and **Osc Fine** zero, so there's no detuning. Set the **Osc Mix** to 50:50%, then play a note. You should not hear anything as both oscillators are cancelling each other out. Keep holding down the note and slowly drag the **Osc Fine** slider. Now you will start to hear sound as the two waveforms no longer have the same period.

Put the **Osc Fine** slider back to 0 and play a new note. Even though these are the exact same settings that you started out with, and which previously produced no sound, this time you will keep hearing the tone. That happens because the two oscillators have gone out-of-phase with each other — they literally have different phase values — and they no longer cancel out. With every new note you play, the effect of this will be different.

However, if you were to always reset the oscillators in `noteOn`, then this wouldn't happen: as soon as the new note starts,resetting puts both oscillators back in phase again. I'm not saying either approach is better than the other; I merely wanted to point out this behavior because understanding how your oscillators interact is key to making a good synth!

With JX11's approach of not resetting the phase, sometimes the sound can disappear. It's hard to predict when, because it depends on what notes you play and in what order, and how the two phases end up relative to each other. Whether this is a problem or not depends on your point of view: sometimes these kinds of quirks are desirable features, sometimes they are bugs. Up to you whether you want to "fix" this or not.

Setting the overall tuning

Besides letting you detune the second oscillator, JX11 also has two parameters for changing the overall tuning of all voices:

- **Octave**: 2 octaves up or down
- **Tuning**: –100 to +100 cents

These two parameters get combined into a single variable again. In **Synth.h**, add this to the `Synth` class as a public member:

```
float tune;
```

This variable will hold the total amount of tuning, expressed in semitones. In **PluginProcessor.cpp**, add the following lines to the `update` method:

```
float octave = octaveParam->get();
float tuning = tuningParam->get();
synth.tune = octave * 12.0f + tuning / 100.0f;
```

This formula is easy enough: there are 12 semitones in an octave and 100 cents in a semitone. Then in **Synth.cpp**, in `noteOn`, you calculate the pitch and period of the note using:

```
float freq = 440.0f * std::exp2((float(note - 69) + tune) / 12.0f);
voice.period = sampleRate / freq;
```

Recall that `note - 69` calculates how many semitones the MIDI note is away from the reference A note at 440 Hz. If the overall tuning is changed, you can simply add this amount from the `tune` variable, as that is also expressed in semitones.

Suppose you tune everything down by one octave, then `tune` is –12. Now for any new note that is played, you take the MIDI note number and subtract 12 for the tuning, so that it appears the note is playing an octave lower. Makes sense, right?

By the way, there is no reason the number of semitones in `tune` has to be a whole number — even if you change the tuning by a fraction of a semitone, the formula still works.

Try it out. You should be able to change the overall tuning of the synth.

Since the fine-tuning in cents goes up or down by 100%, this lets you adjust the tuning by a semitone. Play a B note. Then set tuning to 100%. Play the same B note, you should now hear a C note. To verify this, put tuning back to 0% and play a C note. This should sound the same as the detuned B you just played.

The **Octave** and **Tuning** parameters do not apply to any notes that are already playing, only to new notes. Fair enough, tuning isn't something you might typically change in the middle of playing.

Optimizing the math

While the above works fine, you can do something clever to reduce the number of calculations that are needed. Remember how earlier I mentioned that writing $2^{N/12}$ is the same as 1.059463094359^N? You're going to do something similar here but take it to the extreme. Hold on to your propeller hat!

Add a new function to `Synth` named `calcPeriod`. Put its signature in the **Synth.h** file, with the private members.

```
float calcPeriod(int note) const;
```

The goal of this function is to calculate the period for a given MIDI note number. You're putting that logic in its own function because eventually you'll be calling this from more than one place.

Add the implementation to **Synth.cpp**:

```
float Synth::calcPeriod(int note) const
{
    float period = tune * std::exp(-0.05776226505f * float(note));
    return period;
}
```

I'll explain how this works in a second. First, in `noteOn`, remove the line that calculates `freq`, and change the line that sets the period to:

```
float period = calcPeriod(note);
voice.period = period;
```

In **PluginProcessor.cpp** change the code in `update` to the following:

```
float octave = octaveParam->get();
float tuning = tuningParam->get();
float tuneInSemi = -36.3763f - 12.0f * octave - tuning / 100.0f;
synth.tune = sampleRate * std::exp(0.05776226505f * tuneInSemi);
```

This looks more complicated, but believe it or not, together these formulas do the exact same thing as the formula you've used in the previous section, and they do it more efficiently. But how?! To be honest, the first time I looked at this code I was also perplexed as to what it was doing. When that happens, the trick to figuring it out, is to take the formula apart piece-by-piece until it makes sense.

You know by now that the regular formula for calculating a pitch in Hz is:

$$\text{freq} = 440 \times 2^{(\text{note}-69)/12}$$

This does `(note - 69)` because 440 Hz is the pitch for the A above middle C, which is note 69. If we pull out that constant factor $2^{-69/12}$, the formula becomes:

$$\text{freq} = 440 \times 2^{-69/12} \times 2^{\text{note}/12} = 8.1758 \times 2^{\text{note}/12}$$

Now the reference frequency is no longer 440 Hz but 8.1758 Hz. This is the pitch of the lowest possible MIDI note with note number 0. Verify for yourself that using the same value for `note` gives the same result in both formulas.

What we want is a period in samples, not a frequency in Hz, so we take the reciprocal:

$$\text{period} = \frac{\text{sampleRate}}{\text{freq}} = \frac{\text{sampleRate}}{8.1758 \times 2^{\text{note}/12}}$$

Rearranging this a little gives:

$$\text{period} = \left(\frac{\text{sampleRate}}{8.1758}\right) \times 2^{-\text{note}/12}$$

Because previously it was in the denominator, there is now a minus sign in front of `note`. If we move the first term into a new variable,

$$\text{tune} = \text{sampleRate}/8.1758$$

then we can write:

$$\text{period} = \text{tune} \times 2^{-\text{note}/12}$$

The term $2^{-\text{note}/12}$ can replaced by $(0.9438743127)^{\text{note}}$, because $2^{-1/12} = 0.9438743127$. So you could calculate the period as follows:

$$\text{period} = \text{tune} \times (0.9438743127)^{\text{note}}$$

However, another way to write it is using the exp function, as $\exp(-0.05776226505 \times \text{note})$. This is allowed because x^y is the same as $\exp(\log(x) \times y)$, and $\log(2^{-1/12}) = -0.05776226505$. Note: here I'm using the natural logarithm, the log with base e.

Thus, the formula for calculating the period becomes:

$$\text{period} = \text{tune} \times \exp(-0.05776226505 \times \text{note})$$

That is exactly what `calcPeriod` does, so hopefully this explains that part of the puzzle. If your eyes are glazing over at this point, don't worry. All I'm doing here is showing different ways that these formulas can appear in audio code, and how they're really all equivalent.

Tip: It may seem like we're going through a lot of trouble to rewrite a `std::pow` statement into `std::exp`. Arguably, this even has made the code harder to read. When you see `std::pow(2, (note - 69)/12)`, it's easy to recognize this pattern as a MIDI note number-to-frequency conversion. However, there's a good reason! `std::exp` is a lot faster than `std::pow` because it doesn't need to handle as many special cases. Mathematically speaking, you can always replace `pow(x, y)` with `exp(y * log(x))`, at least when `x > 0`. Therefore, `exp` is preferable in audio code. Every little bit of speed counts!

Let's look again at `update`, which does:

```
float tuneInSemi = -36.3763f - 12.0f * octave - tuning / 100.0f;
synth.tune = sampleRate * std::exp(0.05776226505f * tuneInSemi);
```

When rewriting the formulas above, I said `tune = sampleRate / 8.1758`. You can see in `update` that indeed `synth.tune` equals the sample rate times something, but the factor 8.1758 is missing and instead there is another call to `std::exp`. Let's explore this further.

Recall that 8.1758 Hz is the pitch of MIDI note number 0, our new reference pitch. If no tuning is applied then the term `std::exp(0.05776226505f * tuneInSemi)` should result in the value 1 / 8.1758, because in that situation we want to get `tune = sampleRate / 8.1758`.

We can find what the value of `tuneInSemi` should be to make this happen, by solving the following equation.

$$\text{exp}(0.05776226505 \times \text{tuneInSemi}) = 1/8.1758$$

The standard mathematical trick to get rid of the exp is to take the natural logarithm of both sides of the equation.

$$\begin{aligned}
\text{log}(\text{exp}(0.05776226505 \times \text{tuneInSemi})) &= \text{log}(1/8.1758) \\
0.05776226505 \times \text{tuneInSemi} &= -2.1011785713 \\
\text{tuneInSemi} &= -2.1011785713/0.05776226505 \\
\text{tuneInSemi} &= -36.3763
\end{aligned}$$

So that's where that factor –36.3763 comes from! If no additional tuning is applied, then `tuneInSemi = -36.3763`. A bit weird maybe, since intuitively you might expect that "no tuning" means `tuneInSemi = 0`, but that's not the case. Don't worry too much about what the number –36.3763 means. The only reason we have this, is to get rid of the division by 8.1758.

When tuning is used, you simply subtract the additional number of octaves and cents from `tuneInSemi`, after converting them to semitones. Why subtracting? Because tuning higher means the period becomes smaller, and vice versa.

Phew! It was kind of complicated to get here, but the math actually has been simplified: `synth.tune` combines the sample rate, the tuning in octaves and cents, and the reference pitch of 8.1758 Hz, into a single number. And the formula that calculates the period in `noteOn` now only uses one constant value and one call to `std::exp`.

Try it out! It shouldn't sound any different than before but the math is more efficient.

To be honest, I think the formula we used before,

```
float freq = 440.0f * std::exp2((float(note - 69) + tune) / 12.0f);
```

is much easier to understand. And performance wise, I doubt it's going to be any slower in practice. This formula gets computed once on every Note On event, and these events happen relatively rarely, no more than a few per second in most reasonable music. It's up to you to decide what you like better: readable code or faster equations.

The reason I included the more complicated version in this book is to show that you shouldn't be afraid of math formulas that may not be immediately obvious. With a bit of sleuthing you can probably figure out what is going on, just like we did here. Plus, it's good practice in audio code to use `std::exp` over `std::pow`, and now you know how to convert `pow` to `exp`.

Oh, there's one more thing you should add to calcPeriod. The BLIT-based oscillator may not work reliably if the period is too small. In **Synth.cpp**, add the following line to calcPeriod, before the return statement:

```
while (period < 6.0f || (period * detune) < 6.0f) { period += period; }
```

This ensures that the period, or the detuned period for the second oscillator, is at least six samples long. If not, the while loop lowers the pitch an octave at a time — doubling the period means reducing the frequency in half — until period is at least 6 samples long.

At a sampling rate of 44100 Hz, this means the highest tone our oscillators can produce is 44100 / 6 = 7350 Hz. That's higher than the highest note on the piano, but lower than MIDI note 127, which is about 12.5 kHz. You can experience this pitch drop for yourself by playing notes in the two highest octaves of the MIDI scale.

Bend that pitch!

Since we're messing with the pitch of the notes, you might as well add support for the pitch bend wheel that is on most MIDI controllers. It's pretty easy since all the pieces are in place already. Pitch bend is not controlled by a plug-in parameter, but by a MIDI message.

First, in **Synth.h**, add a new member variable to the Synth class. This can be a private variable, since only Synth needs to use it:

```
float pitchBend;
```

Just like detune, this is a multiplier for the period. In **Synth.cpp**, set pitchBend to a default value of 1.0 in the reset method:

```
void Synth::reset()
{
    voice.reset();
    noiseGen.reset();
    pitchBend = 1.0f;
}
```

It is necessary to give this variable a known value, because if the user does not touch the pitch wheel, you will never get a MIDI message for it. There is no way to query the host for the current pitch wheel position. When the synth starts, you must therefore assume that the pitch wheel is in its center position.

The MIDI message for the pitch bend is “Ex” in hexadecimal, where the x stands for the channel number. As with the other MIDI messages, in JX11 we ignore the channel number. Add the following to the switch in Synth’s midiMessage method:

```
// Pitch bend
case 0xE0:
    pitchBend = std::exp(-0.000014102f * float(data1 + 128 * data2 - 8192));
    break;
```

Oh no, more math! Most MIDI controls receive their value in one data byte, which goes from 0 – 127. Pitch bend, however, uses both data bytes. Each of these still only has 7 bits worth of data in it, but both bytes combined make a 14-bit number. This is done to give the pitch wheel extra precision. A 14-bit integer can contain the values 0 – 16383 if unsigned, or –8192 to 8191 if signed.

Since the pitch wheel rests in the center and can go up and down, we want to have the position as a signed number. That’s why the code first turns the two data bytes into a number between –8192 and 8191, by doing:

```
float(data1 + 128 * data2 - 8192)
```

Let’s do quick mental check to see this makes sense: If data1 and data2 are both 0, this gives –8192. If they are both 127, this gives 8191. In the center position, where there is no pitch bend, data1 will be 0 and data2 will be 64, and the formula returns 0. (The pitch bend data is in little endian order, meaning that data2 contains the most significant bits and data1 the least significant bits.)

Next, we need to decide what these numbers mean. In many synths the user can choose the pitch bend range, but in JX11 it is fixed to two semitones up and down. Creating a parameter that lets the user set this range is left as an exercise for the reader.

When changing the pitch, you always want to express the amount in semitones first and then use the formula $2^{N/12}$, where N is the number of semitones. It’s important to do the calculations using semitones, because that is how music works.

For example, for a 440 Hz tone, the next tone up is an A#/Bb at 466.16 Hz, so that’s 26.16 Hz away. But the previous tone is G#/Ab at 415.30 Hz, which is 24.70 Hz away. These distances are not the same, because the frequency scale is logarithmic. An octave lower, at 220 Hz, the next tone is only 13.08 Hz away, at 233.08 Hz. As you go lower in pitch, the distances in Hz become smaller. As you go up in pitch, they become larger.

Since this is a logarithmic scale, where each frequency doubles or halves every twelve notes, you need to compensate for this using an exponential function, in this case $2^{N/12}$.

The pitch bend range is two semitones. To make the pitch rise by two semitones, you multiply it by $2^{2/12} = 1.12$. Conversely, the multiplier for lowering the pitch by 2 semitones is $2^{-2/12} = 0.89$. However, since you'll be changing the period it's exactly the other way around: multiply by 1.12 to shift the pitch down, and by 0.89 to shift the pitch up.

Therefore, you must map the range –8192 to 8191 into 1.12 – 0.89. If the pitch wheel is at rest in the center position, the multiplier must be 1, so that the period remains unchanged.

You could do this by writing:

$$\text{pitchBend} = 2^{\frac{-2\times(\text{data}/8192)}{12}}$$

Or you can combine all these constants into one value,

$$\log\left(2^{(-2/8192)/12}\right) = -0.000014102$$

and then stick it into an `exp` function, as was done in the code that you just added:

$$\text{pitchBend} = \text{exp}(-0.000014102 \times \text{data})$$

Verify for yourself that filling in `data = -8192` into this formula gives 1.12, `data = 0` gives 1.0, and `data = 8191` gives 0.89.

Applying the pitch bend is easy. In `render`, change the line that sets the period of the first oscillator to:

```
voice.osc1.period = voice.period * pitchBend;
```

And that's all you need to do. Try it out, go bend some notes! You should be able to bend each note up or down by 2 semitones.

Note: When the user rolls the pitch wheel, the synth receives this as a sequence of MIDI events. As you'll recall, the `AudioBuffer` is split into different chunks when these MIDI events happen. So whenever you're in `Synth::render`, you know that `pitchBend` can't change in the meantime. That's why it's enough to set the oscillator periods at the top of `render`, as they are guaranteed not to change during the rest of this block.

Going stereophonic

So far, the audio has always been rendered in mono, not stereo. The synth puts the same sample value into both the left and right audio buffer, which makes the sound appear in the center of the stereo field. To pan the sound away from the center and more to the left speaker or to the right speaker, all you have to do is reduce the volume on one of the channels.

For example, if in `Synth`'s `render` method you were to make the following change, then the audio coming out of the right speaker will be louder than the left speaker, and the sound will appear to be panned to the right.

```
outputBufferLeft[sample] = output * 0.5f;  // make left channel quieter
if (outputBufferRight != nullptr) {
    outputBufferRight[sample] = output;
}
```

One way to turn the synthesizer output into something more stereophonic is to pan lower notes towards the left-hand speaker, higher notes towards the right-hand speaker, and have middle C smack in the middle. That corresponds to the layout of the piano keyboard, where low notes are to the left and high notes are to the right.

Let's make JX11 stereo by assigning the voice a place in the stereo field when it starts playing. In **Voice.h** add two new properties:

```
float panLeft, panRight;
```

And give them an initial value in `reset`:

```
panLeft = 0.707f;
panRight = 0.707f;
```

By default, `panLeft` and `panRight` have the same value, meaning they create the same mono sound as before. You might be wondering why 0.707 instead of 1.0, but that's how the math for the panning formula works out. More about that shortly.

Switch to **Synth.cpp** and make the following changes in `render`.

```
void Synth::render(float** outputBuffers, int sampleCount)
{
    // ...existing code...

    for (int sample = 0; sample < sampleCount; ++sample) {
        float noise = noiseGen.nextValue() * noiseMix;

        // 1
        float outputLeft = 0.0f;
        float outputRight = 0.0f;

        if (voice.env.isActive()) {
            // 2
            float output = voice.render(noise);
            outputLeft += output * voice.panLeft;
            outputRight += output * voice.panRight;
        }

        // 3
        if (outputBufferRight != nullptr) {
            outputBufferLeft[sample] = outputLeft;
            outputBufferRight[sample] = outputRight;
        } else {
            outputBufferLeft[sample] = (outputLeft + outputRight) * 0.5f;
        }
    }

    // ...
```

Step-by-step this is what these changes do:

1. Instead of having a single `output` variable there are separate output variables for the left and right channels.

2. The `voice` still renders in mono, returning a single sample value at a time, but now this sample is mixed into the left channel output using the `panLeft` amount, and into the right channel output using the `panRight` amount.

3. In stereo mode, this writes the sample values for the left and right channels to their respective audio buffers. However, if the synth is placed on a mono bus in the DAW, the left and right output values need to be combined into a mono sample again, undoing the stereo effect.

Finally, to make the sound louder in one channel than in the other and get the stereo effect, you must give `voice.panLeft` and `voice.panRight` appropriate values. That happens when the voice starts playing a new note.

Add the following line to `noteOn`, right below where you set `voice.note`.

```
voice.updatePanning();
```

You still need to add this method. Go to **Voice.h** and add it inside the `Voice` struct:

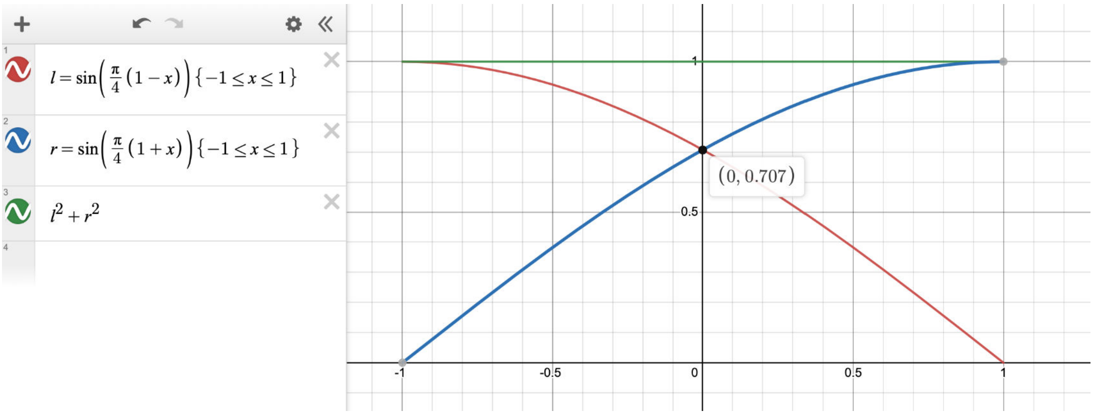
```
void updatePanning()
{
    float panning = std::clamp((note - 60.0f) / 24.0f, -1.0f, 1.0f);
    panLeft = std::sin(PI_OVER_4 * (1.0f - panning));
    panRight = std::sin(PI_OVER_4 * (1.0f + panning));
}
```

First, this converts the voice's MIDI note number into a `panning` value from –1 to +1:

- A panning value of –1 means the sound is fully panned left. This happens when the note is 36 or lower, which is the C two octaves below middle C.
- A panning value of +1 means the sound is fully panned right. This is for note 84 and up, the C two octaves above middle C.
- The panning value for middle C, or note 60, is 0.

I chose these note numbers because this is the range of a 49-key MIDI controller. Any lower notes are clamped to the minimum panning value of –1, any higher notes to the max of +1.

Next, the code applies the constant power panning formula to compute how large `panLeft` and `panRight` should be. It's easiest to explain what's going on here by plotting these curves in Desmos.



1 $l = \sin\left(\frac{\pi}{4}(1-x)\right)\{-1 \le x \le 1\}$

2 $r = \sin\left(\frac{\pi}{4}(1+x)\right)\{-1 \le x \le 1\}$

3 $l^2 + r^2$

The panning curves for the left (red) and right (blue) channels

In these plots, `x` is the equivalent of the `panning` value. At `x = 0`, when the sound should be centered in the stereo field, you can see that the red line (for the left channel) and the blue line (for the right channel) have the same value of 0.707, or `1 / sqrt(2)`. This corresponds to a perceived loudness of –3 dB, which is why this is also called the "–3 dB panning law".

As the sound is panned to the left, the red line increases towards 1.0 while the blue line drops towards zero. The opposite happens as the sound is panned to the right. This means both `panLeft` and `panRight` are values between 0 and 1, and when one goes up, the other goes down.

The reason these particular curves are used rather than straight lines, is that this way the perceived loudness will always stay the same, no matter how the signal is panned. This principle is expressed by the green line, which is the combined power (the square of the amplitude) of both left and right channels. As is clear from the plot, the power always has a constant value, hence the term constant power panning. If we didn't use the constant power formula, the sound would appear louder when panned to the sides than in the middle, which is undesirable.

Try it out! As you alternate playing low and high notes, you should hear the sound move between the two speakers.

Also notice that, if you're using an oscilloscope such as s(M)exoscope that only shows one channel at a time, the amplitude of the displayed waveform depends not just on the velocity but also on the amount of panning. For example, a very high note that is panned all the way to the right, will not show up on the oscilloscope if it's set to the left channel.

Maybe you feel like the current stereo spread is too wide — or not wide enough. As an exercise, try adding a plug-in parameter to control the width of the stereo field. When this parameter is 0%, all notes sound like mono again. The larger the parameter, the further away low notes and high notes should be from the center.

Note: Another way to use panning to give the synth a stereo feel, is to put each new voice in a random position in the stereo field, again with a parameter that determines how wide this stereo field is. JUCE has a `Random` class you can use for generating random numbers. Or perhaps put lower tones closer to the center, where they fit more naturally, and place higher tones randomly left or right.

Besides using panning, you can make the synth more stereo by adding some kind of special effect at the end of the pipeline that is naturally stereophonic, such as a reverb. There are also dedicated stereo widening effects that use techniques such as comb filtering and the Haas effect to suppress certain frequencies from the left channel and boost them in the right channel.

A note on plug-in architecture

To wrap up this chapter, some higher-level thoughts about the architecture of the plug-in.

In the past few chapters, you've steadily been adding public member variables to the `Synth` and `Voice` objects. If you're a seasoned software developer you might balk at my use of public variables in the interfaces of these classes. After all, best practices say you should hide such variables behind methods.

When making audio software there is a trade-off between writing "proper" code and writing fast code. If you abstract everything away and add many layers of indirection, the code might become slower than it needs to be.

In the case of JX11, it probably doesn't matter much if these handful of variables are accessed through methods instead of directly — the compiler can inline that kind of stuff anyway — but the point I'm trying to make is that best practices for writing enterprise software or web apps or mobile apps may not always apply to audio software. Execution efficiency matters more than a clean architecture — although it's obviously best if you can achieve both!

One extreme is to put all the code into a single class. That is what the original MDA JX10 did. This approach has the advantage that you don't have to pass around data between different objects, as everything is accessible from one place. For DSP code that needs to be super efficient, that is probably the way to go.

The architecture used in this book, with `Synth`, `Voice`, `Oscillator`, and `Envelope` classes, was largely chosen to make the inner workings of the synth easier to understand and explain. I'm not claiming this is the One True Architecture. For your own plug-ins, feel free to architect them differently. Just keep in mind that doing things efficiently is more important for audio software than for most other software — if you need to structure your code in way that developers from other fields frown upon, let them frown.

That said, there is one software design pattern you should avoid at all costs when writing plug-ins: global variables. I'm sure you're aware that globals have a bad rap, but for plug-ins they can be deadly! When the user puts your plug-in on multiple tracks in their DAW, each track gets its own instance of your `AudioProcessor` class. If these different instances attempt to use and modify the same global variable, things will go horribly wrong in all kinds of interesting ways. You can't even protect the global with a lock if you need to use it from the audio thread.

Global data is OK if it's read-only, but using globals as a convenient place to store your plug-in's state, or to communicate between the audio processor and the GUI, is guaranteed to flood your inbox with support email requests when it inevitably breaks.

Chapter 10: Polyphony & voice management

The synth you've built up to this point is monophonic: it can only play one note at a time. This might seem like a severe limitation, but many of the classic analog hardware synths are monophonic too. To support more than one voice in hardware, all the electronics need to be duplicated, which is expensive. In addition, sometimes you actually want a monophonic sound, for example to play bass lines or leads.

In software there are no such limitations, and JX11 will also work in polyphonic mode, allowing the user to play chords. In this chapter you'll add polyphony to the synth so that it can play up to eight notes at once. The user can switch between mono and poly modes using the **Polyphony** parameter.

More than one voice

Currently, the `Synth` class has a variable `voice` that handles only a single voice. You will now replace this by an array of `Voice` objects.

Open **Synth.h** and add the following to the public member section:

```
static constexpr int MAX_VOICES = 8;
int numVoices;
```

The `numVoices` variable will be either 1 for mono, or `MAX_VOICES` for polyphony. The maximum amount of polyphony is hardcoded to be eight voices. There are synths that let you choose the amount of polyphony, but for JX11 it is always either one voice or eight. There isn't any reason you couldn't make this higher: simply change `MAX_VOICES` to a different number. Just keep in mind that more active voices means more CPU time will be used.

In **PluginProcessor.cpp**, in `update`, add the following line to fill in the new `numVoices` variable:

```
synth.numVoices = (polyModeParam->getIndex() == 0) ? 1 : Synth::MAX_VOICES;
```

The other parameters you've used were `AudioParameterFloat` objects, but the **Polyphony** parameter is an `AudioParameterChoice`. To get the value of an `AudioParameterChoice` object, call `getIndex()`. Here, index 0 means **Mono** and index 1 means **Poly**.

In the private section of **Synth.h**, remove the declaration of the `voice` variable and replace it by an array:

```
std::array<Voice, MAX_VOICES> voices;
```

This allocates room for all eight voices. In monophonic mode the synth will only use the first object, `voices[0]`.

This change breaks a bunch of code in **Synth.cpp**, so let's fix those errors. Everywhere it used the `voice` variable now becomes a loop through the `voices` array.

Change the `reset` method to the following:

```
void Synth::reset()
{
    for (int v = 0; v < MAX_VOICES; ++v) {
        voices[v].reset();
    }

    noiseGen.reset();
    pitchBend = 1.0f;
}
```

The changes in `render` are more extensive:

```
void Synth::render(float** outputBuffers, int sampleCount)
{
    float* outputBufferLeft = outputBuffers[0];
    float* outputBufferRight = outputBuffers[1];

    // 1
    for (int v = 0; v < MAX_VOICES; ++v) {
        Voice& voice = voices[v];
        if (voice.env.isActive()) {
            voice.osc1.period = voice.period * pitchBend;
            voice.osc2.period = voice.osc1.period * detune;
        }
    }

    for (int sample = 0; sample < sampleCount; ++sample) {
        const float noise = noiseGen.nextValue() * noiseMix;

        float outputLeft = 0.0f;
        float outputRight = 0.0f;
        // 2
        for (int v = 0; v < MAX_VOICES; ++v) {
            Voice& voice = voices[v];
            if (voice.env.isActive()) {
                float output = voice.render(noise);
                outputLeft += output * voice.panLeft;
                outputRight += output * voice.panRight;
            }
        }

        if (outputBufferRight != nullptr) {
            outputBufferLeft[sample] = outputLeft;
            outputBufferRight[sample] = outputRight;
        } else {
            outputBufferLeft[sample] = (outputLeft + outputRight) * 0.5f;
        }
    }

    // 3
    for (int v = 0; v < MAX_VOICES; ++v) {
        Voice& voice = voices[v];
        if (!voice.env.isActive()) {
            voice.env.reset();
        }
    }

    protectYourEars(outputBufferLeft, sampleCount);
    protectYourEars(outputBufferRight, sampleCount);
}
```

It still does the same thing as before, except anything to do with the voices now happens in a loop:

1. Assign the period, plus any detuning and pitch bend, to the voice's oscillators, so that they pick up any changes to the tuning parameters at the start of the block. This is done for all voices that are currently being rendered.

2. Render all the active voices. The sample value produced by each `Voice` object is added to the `outputLeft` and `outputRight` variables, which are initially 0. This is how multiple voices get mixed together, simply by adding them up. The noise oscillator is shared by the voices.

3. After the loop, if any of the voices has its envelope dip below the `SILENCE` level, the voice is disabled so that the synth doesn't waste resources trying to render it. Strictly speaking, this check isn't necessary because a voice already doesn't get rendered if its `env.isActive()` is false. But later you'll also reset other properties of the voice here.

This is a good point to do a quick refactoring of `noteOn`. First, copy the contents of `noteOn` into a new method named `startVoice`.

```
void Synth::startVoice(int v, int note, int velocity)
{
    float period = calcPeriod(note);

    Voice& voice = voices[v];  // this line is new
    voice.period = period;
    voice.note = note;
    voice.updatePanning();

    voice.osc1.amplitude = (velocity / 127.0f) * 0.5f;
    voice.osc2.amplitude = voice.osc1.amplitude * oscMix;

    Envelope& env = voice.env;
    env.attackMultiplier = envAttack;
    env.decayMultiplier = envDecay;
    env.sustainLevel = envSustain;
    env.releaseMultiplier = envRelease;
    env.attack();
}
```

Also add its prototype to the header file:

```
void startVoice(int v, int note, int velocity);
```

The reason you're putting this into a separate method is that `noteOn` will get too complicated otherwise when you add the new functionality in this chapter. Note that `startVoice` has an argument `v` that receives the index of the voice to use.

Change `noteOn` to the following. For now, this simply calls the new `startVoice` method, always using the first voice object:

```
void Synth::noteOn(int note, int velocity)
{
    startVoice(0, note, velocity);
}
```

Finally, add the following line at the top of `noteOff`, so that it always uses the first voice too:

```
Voice& voice = voices[0];
```

Before you continue, compile and run the project. Does the synth still work? Great!

Voice stealing

The synth currently only uses the first `Voice` object from the `voices` array to play the notes. The idea behind polyphony is that it will assign each new note to a new `Voice` object. If you're playing four notes at the same time, four `Voice` objects will be active, one for each note.

Suppose you let go of those notes and play four other notes, then four new voices will be activated while the previous voices will go into the release stage and fade out. Temporarily all eight voices are in use, until the original four notes have faded to the `SILENCE` level and their voices become inactive.

It doesn't really matter which `Voice` is playing which note, the order of the voices in the array doesn't have any inherent meaning. The idea is simply: when a new note needs to be played, the synth finds the next free `Voice` object in the array and uses that voice for the new note.

Of course, this leads to a problem. There are only eight possible voices in the array, so what happens if eight notes are already playing and a new note must be started? You may think that with eight voices this is unlikely to happen — that would be a massive chord — but keep in mind that some of these voices may be playing notes that have already been released but didn't completely fade out yet.

Later in this chapter you'll add support for the sustain pedal. When the sustain pedal is pressed, all previously played notes need to keep ringing. As it turns out, it's not that hard to use up all the voices! This is also why JX11 has a maximum of merely 8 simultaneous voices, which is very small by today's standards: a limited amount of polyphony is good for demonstrating what happens when the synth runs out of voices.

When all voices are in use, the synthesizer needs to perform voice stealing. It has to decide which voice should stop playing its old note and start playing the new one. There are many algorithms for this, but in this book we'll use a simple set of rules.

The JX11 voice stealing algorithm works as follows:

1. loop through all the voices
2. find the quietest voice, based on the envelope level
3. don't steal a voice that is currently in the attack stage

This will first use any voices that are not playing (with envelope level = 0). If all voices are in use, it steals the voice with the lowest envelope value.

Let's implement this algorithm. Add the following private method declaration to **Synth.h**:

```
int findFreeVoice() const;
```

And add its implementation to **Synth.cpp**:

```
int Synth::findFreeVoice() const
{
    int v = 0;
    float l = 100.0f;  // louder than any envelope!

    for (int i = 0; i < MAX_VOICES; ++i) {
        if (voices[i].env.level < l && !voices[i].env.isInAttack()) {
            l = voices[i].env.level;
            v = i;
        }
    }
    return v;
}
```

The `findFreeVoice` method returns the index of the voice to use. The loop finds the voice with the lowest envelope level, which includes voices that are inactive since their envelope level is zero, but ignores voices that are in the attack stage as their envelope is increasing.

If no notes are playing yet, `findFreeVoice` returns 0. Otherwise, it returns the index of the next free voice. If all voices are in use, it returns the index of the voice with the smallest envelope level.

Call this new method from `noteOn`:

```
void Synth::noteOn(int note, int velocity)
{
    int v = 0;  // index of the voice to use (0 = mono voice)

    if (numVoices > 1) {  // polyphonic
        v = findFreeVoice();
    }

    startVoice(v, note, velocity);
}
```

In mono mode, this always uses voice 0. In polyphonic mode, this calls the `findFreeVoice` method to find a free voice or steal one if necessary.

Also make the following change to `noteOff`, so that the correct voice stops playing when you release a key.

```
void Synth::noteOff(int note)
{
    for (int v = 0; v < MAX_VOICES; v++) {
        if (voices[v].note == note) {
            voices[v].release();
            voices[v].note = 0;
        }
    }
}
```

Whenever a MIDI Note Off event is received, `noteOff` checks all the voices to see which one is playing that note, and then the voice is put into release.

This also sets `voice.note` to 0, which registers that this voice is no longer used for a key that is being held down. Previously, `voice.note = 0` meant that the voice wasn't playing at all. Now it means that a Note Off event was received for this voice, i.e. the corresponding key has been released, but the voice will keep playing until the note is done fading out.

Try it out! Make sure the **Polyphony** parameter is set to **Poly**. You can now play chords. Each note in the chord receives its own MIDI Note On event, and so if you play a 3-note chord, `noteOn` is called three times, and three different voices will be playing at the same time. Also see what happens when you attempt to have more than 8 active notes. This is easiest if you set **Env Release** to 50% or longer. For fun, try out the pitch wheel with multiple notes playing!

> **Note:** Even if you press multiple keys on your MIDI controller at what seems the same time, there will probably be timing differences between the Note On events, and the voices won't start playing at the same moment. Likewise for Note Off events. When you manually create a MIDI region in a DAW, or record and quantize the notes, the Note On and Note Off events can be made to happen at exactly the same moment.

JX11's voice stealing rules seem quite reasonable, but they don't cover all edge cases. What happens if all voices are in use and they all have the same envelope value, for example when they are all sustained? This simple algorithm has no solution for that and so it picks voice 0 in that case.

JX11 looks at the envelope level to decide which voice to steal, but the envelope level does not necessarily correspond to the actual loudness of the voice. Two voices can have the same envelope level but have different velocities and therefore different amplitudes. It might be better to steal the quieter voice — the one with the smaller amplitude — before the louder one.

Much more advanced voice stealing algorithms can be devised, for example some of the rules you could use are:

- steal the oldest playing note
- steal the oldest note that isn't the lowest or highest pitched note currently being played
- steal the note with the smallest velocity (or amplitude)
- first try to steal notes that have been released already
- if the same note was already playing, re-use that voice
- … and many others.

Feel free to add some of these more advanced rules into JX11's voice stealing logic!

> **Note:** With our current system of voice stealing, it's possible for the same note to be played by more than one voice. Put the synth in **Poly** mode, set the **Env Release** to 100%. Now play a C and release it. Then play the same C again a few times. Each time you press the key, a new voice is used to play the same note. This is another example of an instrument design decision: should this be allowed or not? If you don't like this feature, you could change `findFreeVoice` to first see if the note is already playing and then reuse that voice — even if there are still free voices available.

The design of JX11 includes another small polyphony enhancement. Add the following constant at the top of **Synth.cpp**:

```
static const float ANALOG = 0.002f;
```

And change `calcPeriod` to the following:

```
float Synth::calcPeriod(int v, int note) const
{
    float period = tune * std::exp(-0.05776226505f * (float(note) + ANALOG * float(v)));
    // the rest stays the same
}
```

This method now has an extra argument `v` for the voice index. Also change the declaration in **Synth.h** to match this.

The formula adds the voice index times 0.002 to the note number, which is a tiny amount of somewhat random detuning, expressed as usual in semitones. This emulates oscillator drift, since no two analog hardware oscillators sound exactly alike. Plus they can go slightly out of tune because of changes in temperature or other external factors. The reason it's "random" is that you never exactly know which note gets assigned to which voice.

Finally, in `startVoice`, change the call to `calcPeriod` to include the voice index:

```
float period = calcPeriod(v, note);
```

Try it out! The effect is very subtle, so you probably won't hear it but it adds a bit of that analog synth charm.

To verify this actually does something, temporarily change `ANALOG` to `1.0f`. Then set **Env Release** to 100% and press the same key several times in a row. It will use a new voice for each key press, and you'll hear each subsequent voice go up by a semitone, even though you're playing the same note each time. So that definitely works!

> **Note:** Many synths have a unison mode, also known as voice stacking, where multiple voices are used to play a single note, but each voice is slightly detuned and has a different starting phase. This makes the sound thick and full but it does use additional voices, which limits the amount of polyphony that can be achieved. Another way to implement unison is to give each voice an array of oscillators, so that it doesn't use additional voices but each voice simply does more work. JX11 doesn't have a unison feature but it's something you might try adding yourself.

Exercise: Instead of having a **Mono** / **Poly** parameter, make this a slider from 1 – 8 where 1 means monophony and any number greater than 1 means polyphony. Keep in mind that the synth will need to have some way to handle the user changing this parameter while voices are playing. Some synths immediately stop all playing notes, others gracefully let the existing notes ring out. Currently, if you switch from Poly to Mono mode, the synth keeps playing active voices but once they're released that voice stays inactive, and any new notes only use voice 0.

The sustain pedal

Piano players will no doubt be familiar with the sustain pedal, also known as the damper pedal. When this pedal is pressed, any playing notes will be sustained even after their keys are released. This makes smooth transitions between different chords possible, as now the player will have ample time to move their hands from one chord position to the next.

It's possible to connect a sustain pedal to most MIDI controllers (check for a SUSTAIN input around the back), and they're inexpensive — worth getting if you're serious about making synths. In this section you will add sustain pedal support to JX11.

The most common type of sustain pedal only registers the pressed and not pressed states, but there are also sustain pedals that can measure how far the pedal is pressed down. In JX11, we only care about the pedal being up or down.

In a more advanced synth, one way to use the extra position information could be to lengthen the release time of the envelope by how far the pedal is pressed, allowing the player to determine with the pedal how much the released notes will be blended with still playing notes.

The plug-in is informed of changes to the position of the sustain pedal through a MIDI message. Let's begin by writing the code to handle this message.

In **Synth.h**, add a new private member variable to `Synth`:

```
bool sustainPedalPressed;
```

You haven't been giving default values to `Synth`'s member variables, since they are derived from the plug-in parameters and are guaranteed to be filled in by `update` before they get used. However, there is no parameter for `sustainPedalPressed`, and the synth only receives a MIDI message when the pedal position changes, so this variable must get an initial value.

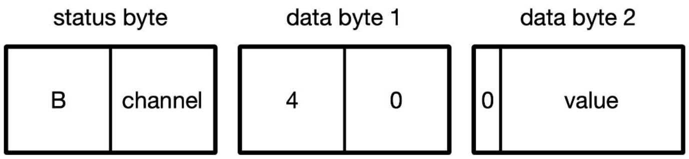
It's reasonable to assume the sustain pedal is not pressed by default. Therefore, add the following line to `reset` in **Synth.cpp**:

```
sustainPedalPressed = false;
```

> **Note:** Even though it's theoretically possible that the sustain pedal is pressed down when `Synth` gets the order to reset itself, in which case `sustainPedalPressed` should be `true` initially, you don't really know anything for sure about the state of the MIDI controller at this point. Maybe the pedal is down, maybe it is not. There is no way to query the host to find out what the position of the pedal truly is, and the plug-in can't rely on old state from before the call to `reset`. Therefore, we play it safe and reset the state of any variables that come from MIDI controllers to reasonable defaults — this is also why `pitchBend` is set to 1.0.

When the sustain pedal is pressed or released, the MIDI controller or DAW sends a MIDI Control Change message to the plug-in, often abbreviated to MIDI CC.

Recall from the MIDI chapter that such messages look like this:



The MIDI message for the sustain pedal

The command is `Bx` where `x` is the channel number. The first data byte describes the type of controller that produced this event. For the sustain pedal the controller code is `0x40` or 64 decimal. The second data byte has the new value for the controller and is 0 – 127.

In `Synth::midiMessage`, add the following to the switch statement:

```
// Control change
case 0xB0:
    controlChange(data1, data2);
    break;
```

To keep the code clean, this calls a new method `controlChange`, because soon enough JX11 will be handling other MIDI CC messages as well.

Add the method declaration to **Synth.h**:

```
void controlChange(uint8_t data1, uint8_t data2);
```

And put its implementation on the next page in **Synth.cpp**.

```
void Synth::controlChange(uint8_t data1, uint8_t data2)
{
    switch (data1) {
        // Sustain pedal
        case 0x40:
            sustainPedalPressed = (data2 >= 64);
            break;
    }
}
```

The code for handling the sustain pedal is straightforward enough: if the controller number in `data1` equals `0x40`, then set `sustainPedalPressed` to true if `data2` is 64 or greater. Many sustain pedals output only two values: 0 for not pressed and 127 for pressed. To also support pedals that output a continuous value, we check for the middle of the range, so that 0 – 63 counts as not pressed, and 64 – 127 as pressed.

Now that you have a way to read the state of the sustain pedal, you'll need to consider how this affects the notes that are currently playing. The point of the sustain pedal is to keep notes alive even after their keys are released. One way to do this, is to keep the voice's envelope in the sustain stage while the pedal is down, whereas previously the envelope would have gone into the release stage.

The sustain pedal affects notes in two ways:

1. Upon a Note Off event, if the sustain pedal is pressed, the note should keep playing.
2. When the sustain pedal is released, any previously sustained notes should now go into the release stage and fade out.

The sustain pedal merely delays the inevitable. This means you need a way to keep track of notes that have been released but are being kept alive by the sustain pedal. Make the following changes to `noteOff`:

```
void Synth::noteOff(int note)
{
    for (int v = 0; v < MAX_VOICES; v++) {
        if (voices[v].note == note) {
            if (sustainPedalPressed) {
                voices[v].note = SUSTAIN;
            } else {
                voices[v].release();
                voices[v].note = 0;
            }
        }
    }
}
```

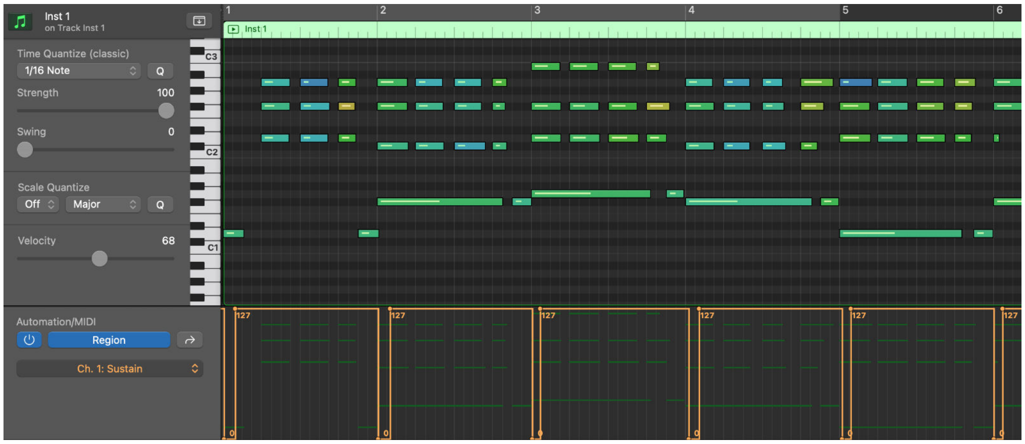
If the sustain pedal is pressed, this changes the voice's note number to a special value, `SUSTAIN`. Where `note == 0` means the key was released and the note is fading out, `note == SUSTAIN` means the key was released but the voice will keep playing for now.

You must still define this special `SUSTAIN` value, so put it somewhere near the top of **Synth.cpp**:

```
static const int SUSTAIN = -1;
```

This is only half the solution but you can already try it out. Play some notes without the sustain pedal and they should stop as usual. Now play some notes with sustain down and they will keep ringing after you let the keys up. What doesn't work yet is stopping these notes when the sustain pedal goes up again.

If you don't have a sustain pedal, you can still test this in your DAW by making a MIDI region and drawing MIDI CC events for the sustain pedal into it. Your DAW may show this as controller number 40 in hex, or 64 in decimal, or as "Sustain".



Time Quantize (classic): 1/16 Note
Strength: 100
Swing: 0
Scale Quantize: Off, Major
Velocity: 68
Automation/MIDI: Region
Ch. 1: Sustain

A MIDI region in Logic Pro with sustain pedal events at the bottom

To stop the `SUSTAIN`ed notes, you can use a little trick. As soon as you detect that the sustain pedal has gone up in the MIDI handling code, you can call `noteOff` yourself — but instead of a real MIDI note number, pass in the special value `SUSTAIN`.

Then `noteOff` will loop through the voices as usual, match any voice whose note number is `SUSTAIN`, and call `release()` on them as `sustainPedalPressed` is now false.

To make this happen, change the MIDI event handling in `controlChange` to the following.

```
switch (data1) {
    // Sustain pedal
    case 0x40:
        sustainPedalPressed = (data2 >= 64);

        if (!sustainPedalPressed) {   // add this
            noteOff(SUSTAIN);
        }
        break;
}
```

This is all you need to do. The existing `noteOff` method will automatically handle the rest. Any voices marked as `SUSTAIN` now get released.

It's quite clever to reuse the `note` property for this, so that the same `noteOff` logic can handle normal notes as well as notes that were kept alive by the sustain pedal. Of course, if you think this is too clever, you could add a separate `sustainedByPedal` property to `Voice`, and then call `release()` on any voices with that property set. There are many ways to implement this kind of logic.

Try it out, you now have a working sustain pedal!

One more thing I want to handle at this point is the All Notes Off or PANIC! message. When this MIDI message is received, the synth should stop playing its voices immediately. There are several different variations of this message, but JX11 will treat anything with controller ID `0x78` (= 120) or higher (up to `0x7F` or 127) as an instruction to stop making sound.

In `controlChange`, add the following default case statement to the switch:

```
// All notes off
default:
    if (data1 >= 0x78) {
        for (int v = 0; v < MAX_VOICES; ++v) {
            voices[v].reset();
        }
        sustainPedalPressed = false;
    }
    break;
```

When such a message is received, all voices are immediately reset, which completely turns them off. These MIDI messages aren't used a lot, but they are useful when a note gets stuck for some reason — for example, if the plug-in didn't receive the appropriate Note Off message somehow. Some MIDI controllers have a Panic button for this reason.

Automatic gain adjustment

You may have found that, when playing big chords with many notes, the sound started to crackle and the debug console printed the message:

```
!!! WARNING: sample out of range, clamping !!!
```

This message comes from the `protectYourEars` function that checks whether the samples you're writing into the audio buffers become too loud, where "too loud" means numbers larger than 1.0 or smaller than –1.0. It's not a good idea for a plug-in to output values outside this range, as such sounds will get horribly distorted when they're played back to the user.

One way to avoid the output becoming too loud is to add a volume control to the plug-in, allowing the user to dial the volume back if necessary. But the plug-in can also try to compensate for loud sounds automatically. JX11 does both.

> **Note:** Inside a DAW, going outside the range [–1, 1] isn't really an issue for plug-ins that are not on the master bus. The user can always turn down the overall volume level in the DAW. Giving the user the option to manage the output levels inside the plug-in is a good idea regardless.

In **Synth.h** add the following to the public variable:

```
float volumeTrim;
```

Then in **Synth.cpp**, in `startVoice`, change the line that sets the amplitude of the first oscillator to the following:

```
voice.osc1.amplitude = volumeTrim * velocity;
```

Previously, the amplitude was determined only by the velocity. Now it also uses the new `volumeTrim` variable to keep the output gain constant after changing parameters.

In **PluginProcessor.cpp**, add the following line to the `update` method. Since this code depends on both `oscMix` and `noiseMix`, make sure to put it below the lines that calculate those values.

```
synth.volumeTrim = 0.0008f * (3.2f - synth.oscMix - 25.0f * synth.noiseMix) * 1.5f;
```

When using two oscillators instead of one, the gain of the output signal increases. Likewise as more noise is added. The `volumeTrim` variable tries to compensate for that. Intuitively, the larger `oscMix` and `noiseMix` are, the smaller `volumeTrim` should be.

Let's say **Osc Mix** and **Noise** are both 0%, then:

```
volumeTrim = 0.0008 * (3.2) * 1.5 = 0.00384
```

In other words, if just one oscillator is used and no noise, `volumeTrim` is a smallish number. Why this particular number?

As you know, `velocity` is a value between 1 and 127. The calculation to set the amplitude used to be `(velocity / 127) * 0.5`. The division by 127 was necessary to turn `velocity` into a number between 0 and 1, and the factor 0.5 was to lower the output volume by 6 dB.

The value of 0.5/127 = 0.00394, which is pretty close to the 0.00384 we're getting here. It's not exact, but for all intents and purposes, the factor 0.5/127 has been folded into the value for `volumeTrim`.

Therefore, with `volumeTrim = 0.00384`, the maximum amplitude of a single voice is –6 dB or 0.5. With one oscillator playing, no noise, and **Env Sustain** set to 100%, you should theoretically see in the oscilloscope that a single note at maximum velocity takes up one-half of the vertical space. In practice, since the integrated sawtooth has a somewhat smaller amplitude than the BLIT peaks, it's more like one-third. Try it out to confirm this for yourself. (Keep in mind that the panning of the note also affects the amplitude.)

Now let's suppose that **Osc Mix** is at 100%, then:

```
volumeTrim = 0.0008 * (3.2 - 1) * 1.5 = 0.00264
```

The volume will be dropped by roughly 3 decibels, since $20 \times \log_{10}(0.00264/0.00384) = -3.25$ dB. That makes sense. When combining two sawtooth waveforms with the same amplitude, you'd expect something like a 3 dB gain increase, so it's reasonable to lower the total volume by that amount.

When both **Osc Mix** and **Noise** are at 100%, then:

```
volumeTrim = 0.0008 * (3.2 - 1 - 1.5) * 1.5 = 0.00084
```

This is a volume reduction of about 13 dB. Adding noise doesn't make the amplitude of the waveform any larger, but it does increase how loud we perceive the sound to be, as there is more frequency content in the spectrum. The volume is reduced to compensate for this extra perceived loudness.

Where does the formula for `volumeTrim` come from? I have no clue. The original author of this synth must have thought this gave a good balance, and it seems to work well enough in practice. Writing synths is not an exact science. You'll add another term to this formula when you implement filter resonance in a later chapter.

Give it a try. Play a note with one oscillator and no noise. Then play the same note at the same velocity with two oscillators. Is the total amplitude that you see in the oscilloscope still roughly the same? s(M)exoscope lets you click in the scope display to set a line for easy comparison.

Now add in the white noise using the **Noise** slider — how does this affect the volume? What happens if you hardcode `synth.volumeTrim` to always be 0.00384 and repeat the experiment?

The output level control

The `volumeTrim` property exists to prevent changes to the oscillator or noise parameters from creating an unwanted boost in the loudness of the voice, but it applies only to a single voice. However, the total output volume also depends on how many notes are playing, their envelopes, filter resonance settings, etc. That's why JX11 has a manual output gain control as well.

The user can change the total amount of output gain using the **Output Level** parameter. Implementing a gain control is the typical "Hello, World!" of audio programming, so I'm sure you've seen this before — but nothing ever is as easy as it seems.

Add a new public variable to **Synth.h**:

```
float outputLevel;
```

Then in **Synth.cpp**, in `render`, after accumulating all the voices into the variables `outputLeft` and `outputRight`, simply multiply the output values with this gain level.

```
outputLeft *= outputLevel;
outputRight *= outputLevel;
```

In **PluginProcessor.cpp**, in `update`, add the following line.

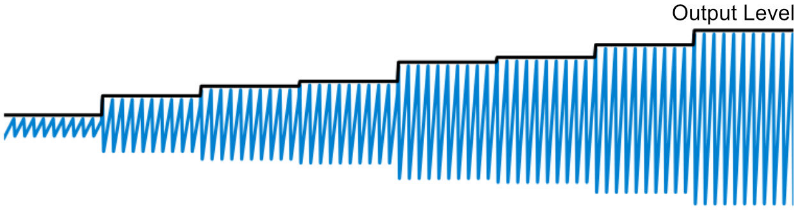
```
synth.outputLevel = juce::Decibels::decibelsToGain(outputLevelParam->get());
```

The parameter is specified in decibels but `outputLevel` should be a linear gain variable, so you use the JUCE helper method `decibelsToGain` to make this conversion. In case you're curious, or if you're not using JUCE, the formula for this is: $10^{\text{decibels}/20}$ or `std::pow(10.0f, decibels/20.0f)`.

Try it out, you can now adjust the output level if the total sound is too loud or too quiet.

The output gain control is very basic, but this does let me touch on an important feature of modern synths and plug-ins in general: parameter smoothing. What happens if you rapidly drag the **Output Level** slider back and forth? If you listen carefully you will hear glitches, also known as zipper noise.

These glitches occur because moving the slider happens at a relatively low speed compared to the sampling rate of the audio. Even if you drag the slider up and down like a maniac, between each parameter update the audio callback will render many samples. If we plot the parameter update over time versus the audio signal, it will look something like this:



The output level (thick black line) moves in steps, which makes the signal jump in steps too

Those steps are what creates the zipper noise. The amplitude level of the audio signal makes the same kind of discrete jumps whenever the parameter is changed. Those jumps introduce extra frequency content in the sound, much like the aliases you saw in the chapter about oscillators.

Note: In our synth this zipper issue is made slightly worse by the way we're handling the parameter updates. Recall that you use a listener to be notified of changes to the APVTS. This is more laggy than reading the parameters directly (it runs off a timer). As an experiment, change the code in `processBlock` to always call `update`, regardless of the `parametersChanged` boolean. Then try dragging the **Output Level** slider again. It's more responsive now and the glitches are less noticeable — but they're still there, the steps between updates are just a bit smaller. Ideally, we wouldn't have such steps at all. Also note that the audio buffer size affects this: a larger buffer size means it takes longer before `update` is called again.

Making it smooth

To avoid the glitches from zipper noise, you can smoothen the parameter changes. There are different ways to do this. In this chapter you'll use one of JUCE's built-in smoother objects. In chapter 12 you'll use a one-pole filter — the same thing you used to make the envelope — to smooth out things.

In **Synth.h**, remove the `float outputLevel` variable declaration and replace it with:

```
juce::LinearSmoothedValue<float> outputLevelSmoother;
```

In **Synth.cpp**, in `render`, grab the output level from the smoother before applying it to the left and right output samples:

```
float outputLevel = outputLevelSmoother.getNextValue();   // add this
outputLeft *= outputLevel;
outputRight *= outputLevel;
```

The `juce::LinearSmoothedValue` is a simple smoother that performs a linear interpolation between the previous value and the new value.

For example, if the previous value is 5 and the new value is 11, the difference between them is 6 units. Suppose the smoothing time is 3 seconds. The smoothed value starts at 5 and is incremented by 2 units every second, until after three seconds it ends up at the target value of 11. In audio code, time is measured in samples, of course, so at a sample rate of 44.1 kHz, the smoothed value is incremented by 2 / 44100 = 0.000045 on each timestep.

In practice, you want smoothing to happen relatively quickly, in say 50 milliseconds. In this example, the increment per timestep then becomes ((11 – 5) / 0.05) / 44100 = 0.00272. Verify for yourself this makes sense: if you start at 5 and keep adding 0.00272, how many steps does it take to get to 11?

Setting the smoothing time is done in `Synth`'s `reset` method. Add the following line to set the duration of the linear interpolation to 50 milliseconds:

```
outputLevelSmoother.reset(sampleRate, 0.05);
```

Back in **PluginProcessor.cpp**, in `update`, replace the line that sets `synth.outputLevel` with the following.

```
synth.outputLevelSmoother.setTargetValue(
            juce::Decibels::decibelsToGain(outputLevelParam->get()));
```

This converts the **Output Level** parameter from decibels into a linear gain value like before, and then tells the smoother object to use this as the new target value. If the smoother is already at this target value, nothing will happen. But if it's not there yet, it will do a linear interpolation over 0.05 seconds from the current value to this new target.

By the way, those 0.05 seconds are measured in audio time, not wall clock time. There is no timer or anything like that: the `LinearSmoothedValue` object simply counts audio samples until it has seen 0.05 seconds worth of them. At a sample rate of 44100 Hz, the interpolation is therefore performed over 2205 samples. Time in this sense only advances when you call `getNextValue()` on the smoother.

There's one more thing you need to do and that is assign an initial value to the smoother. If you don't, when it starts playing, the synth will always do a quick fade-in from zero to the current output level, which sounds odd.

To avoid this, change the `reset` method to the following:

```
void JX11AudioProcessor::reset()
{
    synth.reset();
    synth.outputLevelSmoother.setCurrentAndTargetValue(
            juce::Decibels::decibelsToGain(outputLevelParam->get()));
}
```

This tells the smoother what the initial setting of **Output Level** is without ramping up to it.

Try it out. Any zipper noise you heard before should be gone now. If you're not sure, change the duration from 0.05 to 0.5 or longer, and look in the oscilloscope what happens when you drag the **Output Level** slider.

> **Note:** Perhaps you're wondering why you're using a linear smoother when sound levels are logarithmic in nature… shouldn't the smoothing be exponential so that it remains linear in perception? Good question! To do it "properly", you can keep the parameter in decibels and smoothen the decibel value, then do the `decibelsToGain` calculation in the render loop.
>
> But I'm not sure it's worth it: we just want something that gets rid of the discrete steps that an unsmoothed parameter has. The idea is to make smoothing happen fast enough that the brain doesn't consciously notice it, and so users may not be able to tell the difference between linear and exponential smoothing curves anyway.

In principle, you should think about applying smoothing to most of the plug-in's parameters. The main candidates for smoothing are parameters that give audible zipper noise when you move the slider while playing a sound, especially if they are parameters that are likely to be automated.

However, there are also parameters that don't need smoothing. Some things are already smoothed as a result of the DSP code in which they're used, such as the envelope. These do not need to be smoothed again.

You generally only smoothen parameters that have continuous values, not parameters that act like switches or that have discrete values. For example, the **Octave** tuning parameter is supposed to make discrete jumps. Smoothing this parameter would create an unexpected glide effect.

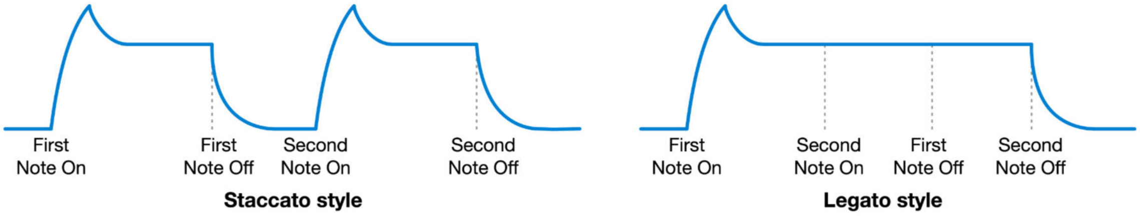
Changing discrete parameters will typically result in glitches, but these are not the kinds of parameters that get automated. After all, it's very unusual for someone to change the **Octave** parameter while sound is playing.

Parameters whose values are only used when starting a new note, such as **Osc Mix**, also do not have to be smoothed. Dragging the **Osc Mix** slider does not affect sounds that are already playing.

Legato playing

With legato-style playing, the key for the new note is already pressed down before the old key is released. When this happens, the synth should create a smooth transition between notes.

In polyphonic mode this isn't an issue since each note gets assigned to a different voice. But in monophonic mode there is just one voice and it should switch to the new note. Rather than completely restarting the envelope, legato playing should continue the existing envelope, keep the same amplitude, and only change the pitch of the tone.



Legato style playing does not restart the envelope

To implement this, instead of calling `startVoice` like before, you'll use a new method `restartMonoVoice`. The difference between these is that now only the pitch of the sound changes, but not the envelope.

In **Synth.h**, add the following private member declaration:

```cpp
void restartMonoVoice(int note, int velocity);
```

Put the implementation in **Synth.cpp**:

```cpp
void Synth::restartMonoVoice(int note, int velocity)
{
    float period = calcPeriod(0, note);

    Voice& voice = voices[0];
    voice.period = period;

    voice.env.level += SILENCE + SILENCE;
    voice.note = note;
    voice.updatePanning();
}
```

This is a simplified version of `startVoice`, used in mono mode when playing legato-style. It does not restart the envelope or calculate new amplitudes for the oscillators.

To use this new method, change `noteOn` to the following:

```cpp
void Synth::noteOn(int note, int velocity)
{
    int v = 0;  // index of the voice to use (0 = mono voice)

    if (numVoices == 1) {  // monophonic
        if (voices[0].note > 0) {  // legato-style playing
            restartMonoVoice(note, velocity);
            return;
        }
    } else {  // polyphonic
        v = findFreeVoice();
    }

    startVoice(v, note, velocity);
}
```

In mono mode, if we receive a Note On event when the voice's `note` is not 0, it means the key for the previous note is still being held down — the synth did not get a Note Off event for it yet. In that case, the user is playing legato-style and you call `restartMonoVoice`. Otherwise, the user is playing staccato style (notes don't overlap) or is in polyphonic mode, and `noteOn` proceeds as before.

Try it out! Put the synth in **Mono** mode and set **Env Attack** to 80%. If you play separate notes, each new note begins with a fade-in from the attack. But if you play a new note while holding down the previous one, a new attack does not happen and the amplitude stays the same.

There is another fun feature we can add for legato playing, called "last note priority". In **Poly** mode, if you play a C note followed by E followed by G, and hold them all down, it sounds like a chord. In **Mono** mode, it sounds like three different notes in a row. So far, so good. But what happens if you let go of G while still holding down C and E?

The answer again varies from synth to synth. On some synths, after you release G it will play the previous note E. And if you let go of E, it will play C — provided you're still holding down that C key, of course.

To implement this feature in JX11 will require the synth to keep track of which keys are currently pressed down, even if they are not making any sound. Recall that in monophonic mode, only voice 0 is rendered. Since the other seven voice objects are not used for anything, they can queue up the previous notes. This allows JX11 to remember up to seven notes, which should be plenty.

How this will work: When a Note On message is received, and voice 0 is already active, copy the MIDI note number from voice 0 into voice 1, copy voice 1's note number into voice 2, and so on. This shifts all the queued notes one position down. You'll write a new method for this, `shiftQueuedNotes`.

Example: The user plays a C. Now voice 0 contains the C note:

```
voice:   0  |   1  |   2  |   3
note:    C  |   0  |   0  |   0
```

Next, they play an E while holding down the C. Voice 0 gets the new E note and the C shifts into voice 1:

```
voice:   0  |   1  |   2  |   3
note:    E  |   C  |   0  |   0
```

The next note is a G, and everything shifts another place down:

```
voice:   0  |   1  |   2  |   3
note:    G  |   E  |   C  |   0
```

Let's consider what happens when a Note Off event is received. If that is the currently playing note, we should look at the queue to see if there are any previous keys being held down. You'll add a new method `nextQueuedNote` that loops through voices 1 – 7 and finds the first voice whose `note` is not 0. Voice 0 is then told to switch to this new note.

After the user has played C–E–G, the voices array looks like this:

```
voice:   0  |   1  |   2  |   3
note:    G  |   E  |   C  |   0
```

Now if the user releases the G note, the others shift one position down and E becomes the new note that's playing:

```
voice:   0  |   1  |   2  |   3
note:    E  |   C  |   0  |   0
```

After the E is released, the next oldest note is the C:

```
voice:   0  |   1  |   2  |   3
note:    C  |   0  |   0  |   0
```

This is the reverse of what happened during the Note On events. However, it's possible that the Note Off message is for a different key than the currently active note. For example, if you play C–E–G and release the E note, then E's place in the queue will be set to 0 — or to `SUSTAIN` if the pedal is down.

Example: Assume the current state is this:

```
voice:   0  |   1  |   2  |   3
note:    G  |   E  |   C  |   0
```

Once the E note is released, the note for voice 1 is set to 0.

```
    voice:   0  |   1  |   2  |   3
    note:    G  |   0  |   C  |   0
```

Now if the G is released, the C will shift down two places and becomes the playing note.

Let's write the code to handle all this. In **Synth.h** add the following private methods.

```
void shiftQueuedNotes();
int nextQueuedNote();
```

In **Synth.cpp**, add the implementation for `shiftQueuedNotes`:

```
void Synth::shiftQueuedNotes()
{
    for (int tmp = MAX_VOICES - 1; tmp > 0; tmp--) {
        voices[tmp].note = voices[tmp - 1].note;
    }
}
```

The loop copies the note number from voice 6 into voice 7, from voice 5 into voice 6, and so on, all the way down to voice 0. It doesn't touch any other part of the voice, only the `note` property.

Next up is the implementation of `nextQueuedNote`:

```
int Synth::nextQueuedNote()
{
    int held = 0;
    for (int v = MAX_VOICES - 1; v > 0; v--) {
        if (voices[v].note > 0) { held = v; }
    }

    if (held > 0) {
        int note = voices[held].note;
        voices[held].note = 0;
        return note;
    }

    return 0;
}
```

First, this loops through the voices backwards to find the most recent voice whose `note` is not 0 or `SUSTAIN`. The index of this voice is placed in the `held` variable. If such a voice is found, its `note` property is set to 0 to mark that element of the queue as used, and the original note number is returned. If there are no voices in the queue that have an active note, i.e. no keys are still pressed, this returns 0.

> **Note:** We could have used another data structure for the note queue, perhaps a `std::vector<int>` that holds just the queued note numbers. But be aware that appending values to the end of a `std::vector` may result in a memory allocation when the vector's internal storage becomes full and needs to be resized. For this reason, `std::vector` should not be used in code that runs on the audio thread. Using a `std::array` or a plain C array would be a better choice. However, by reusing the `Voice` objects for this, the existing logic in `noteOff` already handles removing notes from the queue when their keys are released, as this sets `voice.note` for any matching voices to 0 or `SUSTAIN`.

In **Synth.cpp**, in `noteOn`, call `shiftQueuedNotes` right before `restartMonoVoice`:

```
if (voices[0].note > 0) {
    shiftQueuedNotes();     // add this line
    restartMonoVoice(note, velocity);
    return;
}
```

Finally, add the following to the top of `noteOff`:

```
void Synth::noteOff(int note)
{
    if ((numVoices == 1) && (voices[0].note == note)) {
        int queuedNote = nextQueuedNote();
        if (queuedNote > 0) {
            restartMonoVoice(queuedNote, -1);
        }
    }

    for (int v = 0; v < MAX_VOICES; v++) {
        // as before
    }
}
```

The new code checks whether the synth is in mono mode and that the key that got released is for the note that is currently playing. If so, it means we need to look at the queue to see if the player is still holding down any other keys, and then restart the voice. Again this will use `restartMonoVoice` so that the pitch changes but the envelope keeps going undisturbed.

Try it out! Put the synth in **Mono** mode and play and hold several notes, then release them one-by-one. The notes for the keys that are still pressed should be restored in the order you played them.

Conclusion

This concludes the voice management code for JX11. If you look back over the code for `Synth`, you'll find that the voice management takes up a significant portion of this class and it can be complicated to understand what's going on sometimes. The voice logic handles quite a few things: monophonic and polyphonic modes, the sustain pedal, legato-style playing, and last note priority.

As with anything else in the synth, this is not the only way to implement voice management. If you read through the source code of other synths, you'll find that each of them implements it in their own way. Different synths use different rules, there is no one-size-fits-all solution for voice management.

For example, if the same note is repeated while the sustain pedal is down, one synth may re-use the existing voice for that note, so that at most one voice at a time will be playing that note. Other synths, including JX11, will simply play the same note again but using the next free voice. Both are valid choices, but require different logic.

When writing your own synth, you will find that implementing the voice management logic is one of the trickier parts. It's easy for subtle bugs to creep into this code. I'm sure there are some obscure issues with the voice handling in JX11. In fact, I will demonstrate one now.

Set the synth to **Poly** mode and play a chord. While holding down the chord, switch to **Mono**. The chord keeps playing, which is not a problem in itself. Release the keys. At least one note will keep ringing now and you may not be able to get it to stop! Granted, switching between monophonic and polyphonic isn't something that you'd normally do while playing, but it's still a bug.

Fortunately, the fix is straightforward. In `shiftQueuedNotes` add the following inside the loop:

```
voices[tmp].release();
```

If any voices other than voice 0 happen to be playing in mono mode — which can only happen when you switch from poly to mono — then this line forces these other voices to fade out. This has no effect if the voices are not doing anything, so it's safe to always call it.

I did extensive testing on this synth but there may be other edge cases like this. One way to find these is to write automated unit tests. Creating a suite of tests is always a good idea, even for audio code. I didn't want to cover unit testing in detail in this book, but consider writing some tests for the voice management part of this synth. To make this easier, you might even want to extract the logic for the voice management into its own class. Chapter 14 mentions a few more things about testing.

Maybe you don't want to implement all this stuff yourself for your own synths. Fair enough! You'll be happy to learn that JUCE has built-in synthesizer functionality. We are not using those classes in this book because I feel it's important that you get a sense of everything that's involved in building a synth. Still, you may decide that you'd like to use JUCE's built-in synthesizer classes, so here's a quick overview.

The `juce::Synthesiser` (note the British spelling!) class handles MIDI events, the audio buffer, and the voices. To use this, you create a subclass of `juce::SynthesiserVoice`. Just like our own `Voice` class, this object knows how to create the sound for a single voice. You then call `addVoice` on the `Synthesiser` class — the more voice instances you add, the higher the polyphony for the synth.

Oddly enough you also need to make a subclass of `juce::SynthesiserSound`. This is mostly useful for samplers, where multiple voices may be playing the same sample. JUCE comes with built-in `SamplerVoice` and `SamplerSound` classes for building such sample-based instruments.

However, `juce::Synthesiser` and friends are older classes on the verge of deprecation. If you're going to use JUCE's built-in synthesizer support, I suggest using `juce::MPESynthesiser`, which is more modern and more capable.

In the chapter on MIDI, I briefly mentioned MPE already, which stands for MIDI Polyphonic Expression. This is an extension of MIDI that changes how the synth should handle the notes and channels. An MPE controller will put each note on a new channel, so that messages that previously affected the whole channel — such as pitch bend — now work on individual notes.

If you feel like extending JX11 to make it handle MPE, I suggest that you look into JUCE's `MPESynthesiser` class. You can also use this class if you're not specifically interested in MPE, it's fully compatible with "regular" MIDI. Just make a subclass of `juce::MPESynthesiser` and `juce::MPESynthesiserVoice` and implement the callback functions.

The big advantage of using JUCE's `Synthesiser` or `MPESynthesiser` is that they already implement the voice management for you. It lets you focus on the DSP code for making the actual sounds, not on voice bookkeeping.

Chapter 11: Modulation

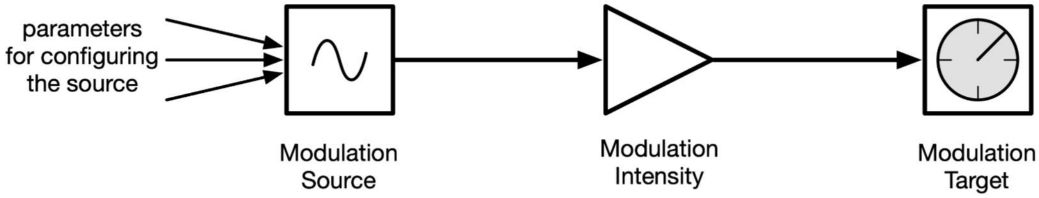
Modulation is using one signal to change another signal. It is an "invisible hand" that automatically twiddles the knobs and sliders of the synth.

For example, to create a vibrato effect you could move the pitch wheel back-and-forth to rhythmically vary the pitch of the playing notes. That works, sort of, but it leaves you with one less hand free to play. With modulation you can let the synth automatically perform that vibrato effect.

One type of modulation you've already seen is the amplitude envelope. The envelope automatically changes the height of the waveform coming out of the oscillators. The envelope is modulating the amplitude, without requiring manual intervention on your end.

Modulation always consists of a source that is changing a target. Often the target is one of the plug-in parameters but it can also be some internal property of the synthesis process, such as the amplitude or the pitch of a note.

The modulation source can be configured using its own parameters, such as the attack, decay, and release times for the envelope. Usually you can also set a modulation amount, known as the intensity or depth, to determine how much the modulation source should affect the target parameter.



How the modulation source is connected to the target

The value coming from the modulator is multiplied by the target's set value to create the final modulated result, although sometimes they're added together. For example: modulating the pitch of the note creates a vibrato effect. The pitch is already set by the note the user is playing, and so the modulator's signal needs to change the pitch relative to its initial position.

A single modulation source can modulate multiple targets at a time, each with its own intensity. In JX11, the LFO can be used both for vibrato and for the filter cutoff frequency.

A target may be modulated by multiple sources. For example, the amplitude of the sound is determined by the envelope and by the velocity of the note. In JX11, the filter cutoff frequency is the most complicated modulation target. It is affected by up to seven different modulation sources. Usually the values from the different sources are added together before they are applied to the target.

In advanced synths, sometimes the parameters of one modulation source may be modulated by a second source. The possibilities are endless and advanced synths can make complex modulation arrangements. Often these are expressed in the form of a so-called modulation matrix that is fully configurable by the user.

For JX11 we're keeping it simpler. Our synth uses the following modulation sources: the envelope, LFO, velocity, MIDI note number, and various MIDI CCs. In JX11 all the modulation routings are fixed. For example, an LFO could be used to add a tremolo effect by making it periodically change the amplitude of the sound. However, in JX11 that modulation routing doesn't exist and the user cannot make their own routings.

Modulation techniques are very powerful and massively expand the kinds of sounds the synth can make.

Velocity

It seems kind of obvious that velocity would change the loudness of the note, but the velocity can also be used for other things, as you'll see in the next chapter where it's one the modulation sources that influences the filter.

For now, you'll refine how the velocity affects the amplitude. Currently, the amplitude of the oscillator is set using the following code:

```
voice.osc1.amplitude = volumeTrim * velocity;
```

Here, `velocity` is the raw MIDI velocity value, which goes from 1 – 127 (velocity 0 is not included, as that means Note Off). Recall that `volumeTrim` includes the factor 1/127 that is used to turn the velocity into a value between 0 and 1.

However, velocity isn't necessarily linear. Velocity is related to the loudness of the sound, and loudness as you know by now is logarithmic in nature. Perhaps a velocity of 64 shouldn't create a sound that has 50% of the full amplitude. Instead, you can define a velocity curve that maps the linear velocity into something else.

There are many ways to do this mapping, and there are synths that let the user choose what kind of velocity curve they prefer. In JX11 the velocity is mapped onto a parabolic curve.

The formula you'll be using is:

$$0.004(\text{velocity} + 64)^2 - 8$$

To study an equation like this in more detail, type it into Desmos or another graphical calculator and replace the variable, here `velocity`, with `x`:

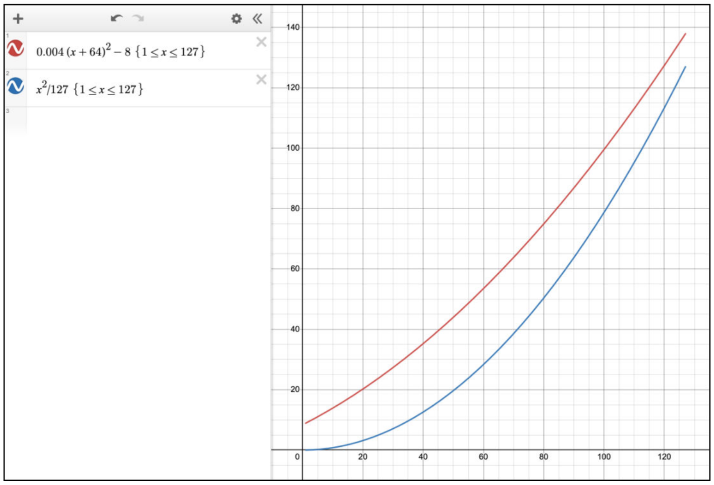
```
0.004 (x + 64)^2 - 8 { 1 <= x <= 127 }
```

Since `x` is a value between 1 and 127, you'll have to zoom out to see the entire curve. It's not a very steep curve but it does map the velocity values into something that feels a bit more natural to play. The part between curly braces tells Desmos to limit the input values to just those numbers.

There is also an "official" curve as suggested by the MIDI Association for doing this mapping:

```
x^2 / 127 { 1 <= x <= 127 }
```

Here are the two curves plotted side-by-side. The red one is the velocity curve used in JX11, the blue one is the mapping suggested by the MIDI Association.



$0.004(x+64)^2 - 8 \; \{1 \le x \le 127\}$

$x^2/127 \; \{1 \le x \le 127\}$

The two velocity curves

The difference between these curves, and not using such a curve at all, is that smaller velocities result in a relatively lower amplitude, making it easier to play quieter notes.

Let's put this velocity curve into the plug-in. In **Synth.cpp**, in `startVoice`, change the line that sets the amplitude of the first oscillator.

```
float vel = 0.004f * float((velocity + 64) * (velocity + 64)) - 8.0f;
voice.osc1.amplitude = volumeTrim * vel;
```

The velocity curve used by JX11 turns the velocity values 1 – 127 into a value in the range 8.9 – 137.9. Don't worry too much about these numbers, it just happens to be what the parabolic curve looks like. The `volumeTrim` variable divides this by 127 and all is well. If you were to use the curve suggested by the MIDI Association, the output range would be 1 – 127.

The smallest amplitude this curve gives, for velocity = 1, is $20 \log_{10}(8.9/127) = -23$ dB. The largest amplitude, for velocity 127, is $20 \log_{10}(137.9/127) = 0.72$ dB. These numbers describe the amount of amplification or attenuation relative to the full amplitude of the waveform.

In other words, if you play with full velocity, the sound level is boosted a tiny bit by 0.72 dB. If you play with the softest possible velocity, the sound level is lowered by 23 dB.

Therefore, the total dynamic range of this velocity curve is approximately 24 dB. The term dynamic range describes the difference between the loudest and quietest possible sounds. Here, the loudest sound is 24 dB louder than the softest sound. (For the MIDI Association curve, the dynamic range is about 42 dB, meaning it can play even quieter sounds.)

In many synths, there is a velocity sensitivity setting that determines how much the MIDI note velocity truly affects the loudness. The dynamic range just described is for 100% sensitivity. At 50% sensitivity, the dynamic range would get cut in half, so that the lowest velocity is only 12 dB quieter than the loudest.

The lower the sensitivity, the less the velocity affects the amplitude. In modulation terms, velocity sensitivity determines the intensity at which the velocity modulation source modulates its target, in this case the amplitude.

JX11 has a parameter named **Velocity** for the velocity sensitivity. However, this is not used to modulate the amplitude. This sensitivity parameter goes from –100% to +100% and will be used in the next chapter to set the intensity at which the velocity affects the filter's cutoff frequency.

When you drag the **Velocity** slider all the way to the left, it changes to say "OFF" rather than a percentage. This was done in `createParameterLayout` in **PluginProcessor.cpp** with a `withStringFromValueFunction`. In JX11, the **Velocity** parameter normally only changes how the filter reacts to velocity, but when the parameter is set to OFF, it also affects the amplitude of the sound.

Setting **Velocity** to OFF effectively changes the dynamic range of the sound to 0 dB. The velocity no longer matters, and all notes play at the same volume regardless of how hard you bang the keys or how softly you caress them.

In **Synth.h** add the following public member variables.

```
float velocitySensitivity;
bool ignoreVelocity;
```

In **PluginProcessor.cpp**, add the following to the `update` method:

```
float filterVelocity = filterVelocityParam->get();
if (filterVelocity < -90.0f) {
    synth.velocitySensitivity = 0.0f;
    synth.ignoreVelocity = true;
} else {
    synth.velocitySensitivity = 0.0005f * filterVelocity;
    synth.ignoreVelocity = false;
}
```

This sets `ignoreVelocity` to `true` when the parameter says OFF. It also sets the `velocitySensitivity` value already but you won't use this until next chapter.

Next, in **Synth.cpp** add the following line to the start of `noteOn`:

```
void Synth::noteOn(int note, int velocity)
{
    if (ignoreVelocity) { velocity = 80; }

    // ...the rest...
}
```

This setting effectively disables the velocity modulation source by forcing the velocity to a fixed value, which is chosen here to be 80.

Try it out! When the **Velocity** parameter is set to OFF, all the notes you play will always be equally loud.

Exercise: Velocity sensitivity for the amplitude is not an official part of this synth but I encourage you to experiment with it: use the `velocitySensitivity` variable in `noteOn` to change the velocity. You've already seen how to turn the velocity completely off. The `velocitySensitivity` variable goes between –0.05 and +0.05. Since it's intended for modulating the filter, this value can be negative. I suggest doing `float sens = std::abs(velocitySensitivity) / 0.05;` to turn it into a number between 0 and 1, and then use this to make the velocity less important for setting the amplitude when the sensitivity is lower than 100%. Good luck!

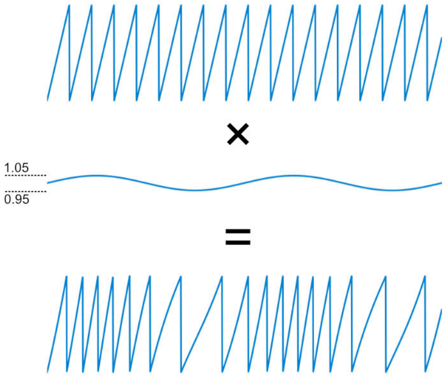
LFO

No synth is complete without an LFO. The LFO, which stands for Low Frequency Oscillator, is an oscillator that isn't used for making sound directly. Instead, it changes one or more parameters of the sound-making process in a periodic fashion. The frequency, or rate, of an LFO is usually less than 20 Hz and can be as low as 0.01 Hz, which means one cycle of the oscillator takes 100 seconds.

Because the frequency is less than 20 Hz, you can't hear the LFO waveform, as our ears do not interpret such vibrations as sound. But you can use this waveform to modify some aspect of the sound that's being produced. In this section you'll implement the basic LFO mechanisms to perform vibrato, which uses the LFO to change the pitch of the sound.

In JX11 the LFO always is a basic sine wave. You've already seen how to create such an oscillator so this should be a piece of cake. In more advanced synths the user may be able to choose between different waveform shapes, such as triangle, sawtooth, square wave, or random values (also known as sample-and-hold).

Vibrato is an effect where the pitch of the oscillator is varied in a subtle way. The figure below starts out with a sawtooth waveform at a given frequency. It's clear that each period is exactly the same. Then we multiply the pitch by a sine wave that slowly oscillates between 0.95 and 1.05, to get the modulated sawtooth wave. Now you can see that the periods vary from larger to smaller and back again. On average the periods have the same length as before, so you still hear the original pitch but with some additional movement.



Multiplying the pitch by a sine wave creates a vibrato effect where the periods expand and contract

There are a few things you must do to accomplish this. First, the `Oscillator` class has to be modified to allow for such modulations. It's been a while since we looked at `Oscillator` but recall that it has a public variable `period`. If this is changed, for example by using pitch bending, then the next cycle is rendered using the new period.

One way to achieve vibrato is to change the `period` variable using the LFO. However, it's cleaner to not apply the modulations to the period directly but use a new variable for this.

In **Oscillator.h**, add a new public variable to `Oscillator`:

```
float modulation = 1.0f;
```

This is a multiplier that describes the current amount of modulation to be applied to the period. If it is 1.0, there is no modulation.

In `nextSample`, change the line that calculates the `halfPeriod` to the following:

```
float halfPeriod = (period / 2.0f) * modulation;
```

Whenever a new cycle happens, the modulation amount is applied. Since this is a multiplier for the period — as opposed to the frequency — a value of `modulation` larger than 1 will make the note lower, while a value smaller than 1 will make the note higher. If you want to try this out now, temporarily change the declaration to `float modulation = 0.5f;` and any note you play will be an octave higher.

Next you need to create the LFO and put the output of its sine wave into this new `modulation` variable. But where does the LFO live?

Again, this is a synthesizer design choice. You could give each voice a separate LFO instance. The result of this choice is that all voice LFOs run at the same frequency, but each has its own phase. For JX11 that is not very important, so we keep it simple and just have one instance of the LFO that is owned by `Synth` and is shared by all the voices. With this choice, each voice always performs the exact same vibrato.

Next, you must decide how often to run the LFO calculations. You could run the LFO at audio rate, which means once for every sample that gets rendered, 44100 or more times per second. That may be overkill. After all, if the maximum frequency of the LFO is 20 Hz or so, it takes over 2000 samples for a single cycle of the LFO to complete — and lower frequencies take even longer.

If you look at any two neighboring samples in a 2000-point sine wave, you'll notice that the difference between them is very small. You can safely update the LFO at a slower rate and save some CPU time.

In JX11, the update rate for the LFO is once every 32 samples. At this rate, each LFO cycle at the maximum frequency of 20 Hz is made up of 69 steps, which is still sufficient to draw a clean sine wave. Looking at it another way: at a 44.1 kHz sample rate, updating once per 32 samples means the LFO gets recalculated 1378 times per second. That's plenty!

> **Note:** Calculating things at a slower rate than audio rate is something you're already doing in the synth. Any properties from `Synth` that depend on parameter values are recalculated only when the parameter changes, and never more than once per audio block. Likewise for applying pitch bend: that happens at the rate that MIDI events are received. Typically, only the actual rendering of the audio takes place at audio rate, everything else can run slower.

To recap, whereas the regular `Oscillator` outputs new values 44100 times per second, you only calculate the next output value of the LFO after every 32 samples of audio. Anything that depends on the LFO, such as the amount of vibrato modulation, is then also updated 32 times slower than the audio rate.

Recall that for parameter updates you applied smoothing, to make sure the changes weren't too "jumpy" and avoid zipper noise. For the LFO this is not a problem, as it still gets updated a few thousand times per second, which is plenty smooth.

In **Synth.h**, add the following to the public members:

```
const int LFO_MAX = 32;
float lfoInc;
```

The constant `LFO_MAX` determines how often the LFO is updated, measured in audio samples. This needs to be public because you'll use it to calculate several other properties later. The `lfoInc` variable is the phase increment for the LFO. This is the same thing you've seen before when you were learning how to make oscillators. Since this is for a sine wave, `lfoInc` will be a value between 0 and 2π.

The `lfoInc` variable is derived from the **LFO Rate** parameter that the user can set. Let's write the code that calculates this value.

In **PluginProcessor.cpp**, add the following lines to `update`:

```
const float inverseUpdateRate = inverseSampleRate * synth.LFO_MAX;

float lfoRate = std::exp(7.0f * lfoRateParam->get() - 4.0f);
synth.lfoInc = lfoRate * inverseUpdateRate * float(TWO_PI);
```

Recall from the oscillators chapter that the phase increment is `2π * freq / sampleRate`. That's exactly what is calculated here: the variable `lfoRate` sets the frequency and `inverseSampleRate` is `1 / sampleRate`. However, the sample rate for the LFO is 32 times lower than the audio's sample rate, which is why this is placed in its own variable, `inverseUpdateRate`. You will use this lower update rate for most of the other modulation parameters too, not just the LFO.

The `lfoRate` variable does some stuff with `std::exp` that you may recognize by now as a way to skew the parameter. It's an exponential curve that maps the 0 – 1 parameter value to 0.0183 Hz – 20.086 Hz, or roughly 0.02 Hz to 20 Hz. Plot the curve `exp(7x - 4)` in Desmos if you're not sure what it looks like. (This could also have been done without an `exp` function by setting the minimum of the parameter to 0.02 and the maximum to 20, with an appropriate skew factor.)

Great, you now have a phase increment for the LFO that describes how to make a sine wave with the given frequency, at a sample rate that is 32 times smaller than the audio rate.

In **Synth.h**, add the following to the private members:

```
void updateLFO();

int lfoStep;
float lfo;
```

The `lfo` variable has the current phase of the sine wave. `lfoStep` is a counter that counts down from `LFO_MAX` to 0, to keep track of when to move the LFO forward, which happens in the method `updateLFO`.

In **Synth.cpp**, add the following two lines to `reset`, to set the LFO state back to zero:

```
lfo = 0.0f;
lfoStep = 0;
```

Then in `render`, call `updateLFO`. This is done before anything else inside the loop:

```
void Synth::render(float** outputBuffers, int sampleCount)
{
    // ...

    for (int sample = 0; sample < sampleCount; ++sample) {
        // add this line
        updateLFO();

        // ...
    }
```

Note that `updateLFO` is called on every sample, meaning at the audio rate. It also handles the counter that determines whether on this sample the LFO should be advanced and the modulations should be calculated.

Next, add this new `updateLFO` function:

```
void Synth::updateLFO()
{
    // 1
    if (--lfoStep <= 0) {
        lfoStep = LFO_MAX;

        // 2
        lfo += lfoInc;
        if (lfo > PI) { lfo -= TWO_PI; }

        // 3
        const float sine = std::sin(lfo);

        // 4
        float vibratoMod = 1.0f + sine * 0.2f;

        for (int v = 0; v < MAX_VOICES; ++v) {
            Voice& voice = voices[v];
            if (voice.env.isActive()) {
                voice.osc1.modulation = vibratoMod;
                voice.osc2.modulation = vibratoMod;
            }
        }
    }
}
```

Here's how this works:

1. Decrement `lfoStep` every time this function is called. When `lfoStep` reaches 0, the logic inside the `if` block is executed. Because you always set `lfoStep` back to `LFO_MAX`, this means the `if` statement is entered after every 32 samples. And as `lfoStep` is initially made 0 in `reset`, this logic is also guaranteed to be run the very first time.

2. Increment the LFO's phase variable `lfo` with the step size `lfoInc`. When this exceeds `PI`, which is the halfway point of the sine wave, subtract `TWO_PI` to put `lfo` back to `-PI`. How this works was covered in chapter 5, so check that out if you need to refresh your knowledge. The important part is that the phase is kept somewhere between ±π, so that it doesn't grow too large as `std::sin` doesn't work so well with a large argument.

3. Calculate the sine value. There's no need to bother with optimizations such as using the direct-form resonator, as `std::sin` is called only once every 32 samples and is fast enough for that.

4. Use the output value from the LFO to calculate a vibrato amount and then assign this to the `modulation` property of the two oscillators.

Try it out! For any note you play, you should hear the sound go up and down in pitch. Use the **LFO Rate** slider to vary the speed at which this happens. In the frequency analyzer you should see the peaks moving back and forth.

Right now the vibrato amount is hardcoded — and it's a bit extreme! It sounds more like an air raid siren than proper vibrato. That's why JX11 also has a **Vibrato** parameter that lets you change the intensity of this particular modulation.

In **Synth.h**, add a public property to the class:

```
float vibrato;
```

And then change the line in `updateLFO` to the following. Instead of multiplying with a fixed value 0.2, you multiply with the new `vibrato` variable:

```
float vibratoMod = 1.0f + sine * vibrato;
```

The final step is to fill in this vibrato value with the parameter, which happens as usual in `update`, in **PluginProcessor.cpp**:

```
float vibrato = vibratoParam->get() / 200.0f;
synth.vibrato = 0.2f * vibrato * vibrato;
```

The **Vibrato** parameter goes between –100% and +100%, although we'll ignore the negative part for now. The formula above calculates a parabolic curve — because the value is squared — that goes between 0.0 and 0.05. Plot it in Desmos to see for yourself.

Because the maximum value for `synth.vibrato` is 0.05, and `std::sin` goes between –1 and +1, the value of `sine * vibrato` goes periodically up and down between –0.05 and +0.05. To get the final value of `vibratoMod`, you add 1.0 to this, which makes `vibratoMod` vary between 0.95 and 1.05. That's roughly the same as one semitone up, since $2^{-1/12} = 0.9439$, or one semitone down, $2^{1/12} = 1.0594$. That's more than enough for vibrato.

When the **Vibrato** parameter is set to 0%, the vibrato effect is disabled because `vibrato` is 0, and so `vibratoMod = 1.0f + sine * 0 = 1.0f`. Since `vibratoMod` is used to multiply the period, nothing changes now and the sound always keeps its original period.

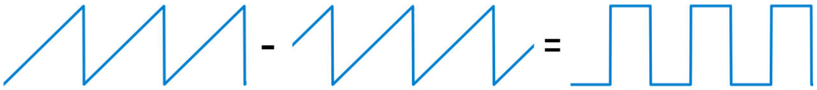
Try it out! Set **LFO Rate** to something like 5 Hz and **Vibrato** to 30% and you'll hear a gentle vibrato effect. Switch a few times between 0% and 30% and notice how even a little added vibrato can make the sound feel much more alive.

Note: In our implementation, the amount by which the pitch is multiplied upwards is the same as downwards. When the **Vibrato** parameter is at 100%, the modulated pitch hovers between 0.95× and 1.05× the original pitch. That's a difference of 0.05 in either direction. Technically speaking this isn't entirely correct. The amount to swing down must be smaller than the amount to swing up because of the logarithmic nature of frequencies.

If we were to do vibrato by exactly one semitone up and down, the pitch would vary between 0.9439× and 1.0594×. Note that the difference downward is 0.0561 and the difference upward is 0.0595. These numbers are not the same, but of course no one is going to notice this in practice. If you wanted to do it properly and burn some extra CPU cycles, you'd have to take the `sine * vibrato` value and treat it as the number of semitones and then do $2^{N/12}$ to calculate the correct multiplier.

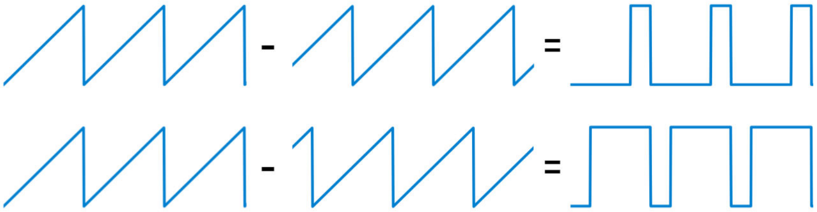
PWM

In JX11 the LFO is also used for PWM or Pulse Width Modulation. You’ve seen that two sawtooth waves can be combined into a square wave:



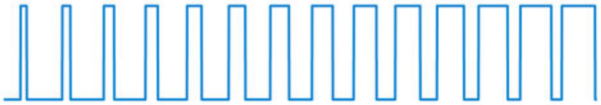
Subtracting a shifted sawtooth from another sawtooth makes a square wave

The pulse width of the above square wave is 50%, meaning that the waveform is positive 50% of the time and negative 50% of the time. By shifting the second sawtooth wave relative to the first, you can change the pulse width:



Shifting the second sawtooth changes the pulse width

By changing the shift amount over time, the width of the pulse continuously changes. This is PWM.



PWM is when the pulse width is varied

Interestingly enough, `Voice` already subtracts `osc2` from `osc1`, and while playing with the synth you’ve probably been able to create square waves with different pulse widths, as well as other kinds of shapes, depending on the amount of detuning and the difference in phase between the two sawtooths.

For example, restart the synth and set it up as follows:

- **Osc Mix**: 50:50%
- **Osc Tune**: 0 semi
- **Osc Fine**: 5.0 cents

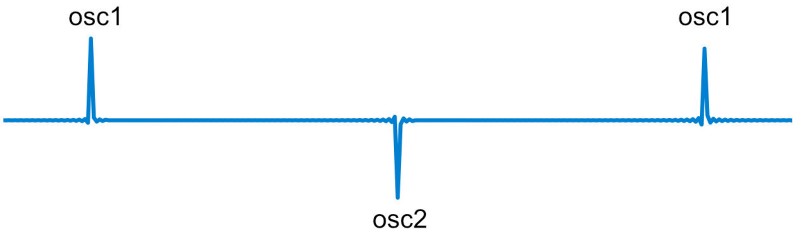
Now play a note. You will hear the kind of pulsating motion in the sound that is typical for PWM even though you're not using an LFO yet. This is simply the two sawtooth waves interacting with each other. On the oscilloscope you can clearly see the waveform is a pulse wave and that the width of the pulse changes over time.

Play a few notes and you'll hear the effect is different every time. The oscilloscope shows that the pulse width is almost random. The relative tuning of the two waves stays the same, but their phase isn't. In this synth, the phase of the oscillators is not reset when new notes are played, and so the effect is unpredictable.

So, if you can already make square waves and something that resembles PWM, then why does this synth have a special PWM mode? As just demonstrated the waveform isn't very reliable to create and the sound you get depends on whatever the phases of the two oscillators happen to be. In fact, by playing an unlucky sequence of notes you can make the sound completely disappear because the oscillators will cancel each other out if their phases align.

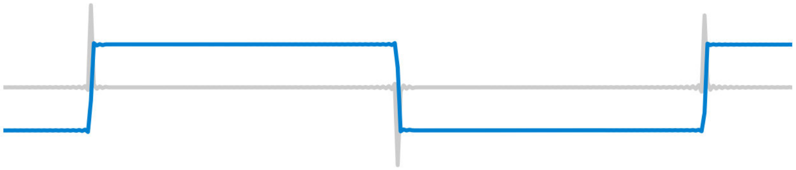
JX11's PWM mode exists to create the square wave in a more reliable fashion. This is done by making the phase of the second oscillator "locked" to the phase of the first oscillator, and shifting it by 180 degrees, i.e. by half a cycle.

What happens now is that the BLIT pulse from `osc2` is placed exactly in between the peaks from `osc1`, but is negative:



The negative peak of oscillator 2 is placed halfway the peaks from oscillator 1

Integrating this signal creates the following square wave. It transitions between high and low on every sinc peak:



The square wave resulting from the arrangement of the sinc peaks

By moving the position of the negative peak closer to or away from the positive peaks, you change the duty cycle of the wave. If you do this smoothly over time, the result is pulse width modulation.

One way to achieve this is to change the phase of the second oscillator, so it shifts in time relative to the first. Alternatively, you can slightly alter the period of the second oscillator, essentially performing a vibrato on only one oscillator. That will also move the positions of the sinc peaks from `osc2` relative to the peaks from `osc1`. This is the method JX11 uses.

To set this up, in **Oscillator.h**, add a new method to `Oscillator`:

```
void squareWave(Oscillator& other, float newPeriod)
{
    // 1
    reset();

    // 2
    if (other.inc > 0.0f) {
        phase = other.phaseMax + other.phaseMax - other.phase;
        inc = -other.inc;
    } else if (other.inc < 0.0f) {
        phase = other.phase;
        inc = other.inc;
    } else {
        phase = -PI;
        inc = PI;
    }

    // 3
    phase += PI * newPeriod / 2.0f;
    phaseMax = phase;
}
```

Normally the two oscillators have their own phase that is never “synced up” anywhere. However, to make a reliable square wave, the `squareWave` function is used to set the phase of one oscillator based on the phase of the other oscillator. This method is intended to be called on `osc2` when the voice starts playing a new note.

The logic in `squareWave` sets the starting phase of the second oscillator so that its sinc peaks fall exactly halfway in between two sinc peaks from the first oscillator:

1. Reset this oscillator in case it had been used already, so that the digital resonator that makes the sinc curve doesn’t try to continue from stale values.
2. Figure out what the phase of the other oscillator is and in which direction it’s going. If this math doesn’t make any sense to you, don’t worry about it. It’s specific to the BLIT algorithm that you’re using to make these sinc pulses.

The final `else` clause is somewhat of a hack: if `other.inc` equals zero, the other oscillator has not started yet, and there's no way of knowing where its peaks will be. In that case we must guess… usually `inc` is a value close to π, so that'll have to do.

3. Shift the phase by half a period so that we're halfway in between the peaks of the other oscillator. Recall that `phase` is measured in samples times π, which is why you multiply by `PI`. The `newPeriod` variable used here is the unmodulated period for the note to play.

Before you can put this into action, first define some new variables. In **Synth.h**, add the following public member to `Synth`:

```
float pwmDepth;
```

This is the modulation intensity for the PWM effect. In **PluginProcessor.cpp**, in `update`, add the following lines below the lines that set `synth.vibrato`:

```
synth.pwmDepth = synth.vibrato;
if (vibrato < 0.0f) { synth.vibrato = 0.0f; }
```

This first sets the new `pwmDepth` variable to the value of `synth.vibrato`. Recall that the **Vibrato** parameter goes from –100% to +100%. If it's negative, PWM mode should be used instead of the vibrato effect. In that case, `synth.vibrato` is made 0 to turn off regular vibrato.

Note that `pwmDepth` itself never becomes negative, because `synth.vibrato` contains the squared value of the parameter, and a negative number squared is always positive.

In **Synth.cpp**, in `updateLFO`, change the following lines:

```
float vibratoMod = 1.0f + sine * vibrato;
float pwm = 1.0f + sine * pwmDepth;       // add this line
```

This sets the intensity for the PWM modulation in the same way that you did for vibrato. In the loop that assigns these values to the voice oscillators, make the following change:

```
voice.osc1.modulation = vibratoMod;
voice.osc2.modulation = pwm;   // change this line
```

Previously, `vibratoMod` was used for both oscillators but now `osc2` gets the `pwm` value.

How this works: If the **Vibrato** parameter is 0 or positive (slider moved to the right), the vibrato effect should be used. In that case, `vibrato` and `pwmDepth` have the exact same value, and so `vibratoMod` and `pwm` are the same too. Both oscillators will modulate their pitch using the LFO.

If the **Vibrato** parameter is less than 0 (slider moved to the left), `pwmDepth` still works as before but `vibrato` is 0, which means `vibratoMod` is zero too, and only the second oscillator gets modulated.

Therefore, the main difference between vibrato and PWM — at least how it's implemented in JX11 — is that with vibrato both oscillators are modulated by the same amount, but with PWM, `osc1` is not modulated and only `osc2`'s pitch swings back and forth at the LFO rate.

Additionally, in PWM mode the starting phase of `osc2` is linked to that of `osc1`, which is what happens in the `squareWave` method. Speaking of, there is one more step to make all this happen, and that is to call `squareWave` on the second oscillator when the note starts.

In **Synth.cpp**, in `startVoice`, below the lines that set the oscillator amplitudes, add this:

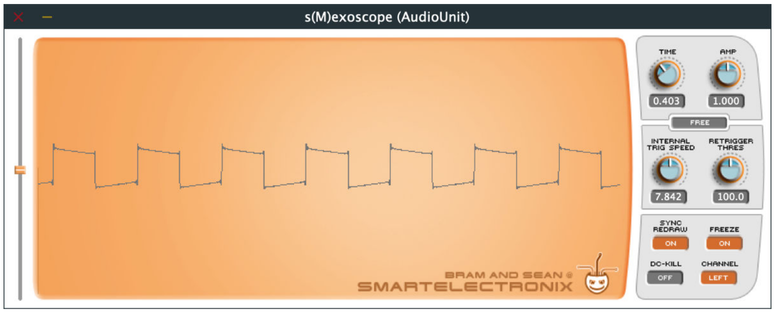
```
if (vibrato == 0.0f && pwmDepth > 0.0f) {
    voice.osc2.squareWave(voice.osc1, voice.period);
}
```

If the synth is in PWM mode, the second oscillator will now be phase-locked to the first one to create a proper square wave.

Try it out! Set the parameters as follows:

- **Osc Mix** to 50:50%
- **Osc Tune** and **Osc Fine** to 0
- **LFO Rate** to 0.3 Hz
- **Vibrato** to something small like PWM 1.0% (drag to the left)

Play a note and you should see a stable square wave with a 50% pulse width:



s(M)exoscope (AudioUnit)

TIME 0.403
AMP 1.000
FREE
INTERNAL TRIG SPEED 7.842
RETRIGGER THRES 100.0
SYNC REDRAW ON
FREEZE ON
DC-KILL OFF
CHANNEL LEFT

A square wave with 50% duty cycle or pulse width

As you drag the **Vibrato** slider to the left, the pulse width will start to vary with the speed of the LFO. Verify for yourself that this sounds quite different than when the **Vibrato** slider is to the right. Keep in mind that PWM is only useful when Osc Mix is not 100:0%, since it needs both oscillators to make sound.

To see what happens under the hood, change `Voice`'s `render` method to do `saw = sample1 - sample2`. This removes the integrator part that turns the sinc pulses into saw waves. Now it will just draw the sinc pulses and you can see the negative peak moving back and forth relative to the positive one in the oscilloscope.

This technique of making a square wave that can be modulated, by subtracting two sawtooth waves that are shifted in phase, is not limited to the BLIT oscillator you're using here. If you implemented a different sawtooth oscillator, for example using the DPW algorithm, the same trick still works. Simply make the second oscillator start with a phase that is 180 degrees shifted with respect to the first oscillator. And then the modulation of the period does the rest.

> **Note:** If you're playing mono notes legato-style, or queued mono notes, the `squareWave` method is not called, as you only added it to `startVoice`, not to `restartMonoVoice`. The result is that phase differences in mono legato may cause the two oscillators to cancel each other out. Try this by putting the synth in **Mono** mode and alternating between very low and high notes in legato. In the oscilloscope you can clearly see this gives a different waveform than playing these notes separated. Whether this is a problem or not depends on your perspective. Personally, if I was designing a new synth from scratch I might make a different choice, but quirks like this do give JX11 some character.

Exercise: If you feel like a challenge, add a few of the other common LFO waveforms such as triangle, sawtooth, and square wave, with a parameter to choose between them. Unlike the oscillators that are used to produce sound, LFO waveforms should not be bandlimited. This is because the LFO isn't used to directly output sound, only to modulate other parameters. In fact, using a bandlimited sawtooth will sound weird at low rates because of the extra wobbles in it that might become audible.

To add a sawtooth LFO, simply use the naive form that was shown in chapter 6. An LFO square wave is also trivial to implement: half its period it should be 1.0, the other half it should be –1.0 (no need to combine two sawtooths). Notice that a square wave LFO makes sudden jumps, which you might hear in the sound as glitches. Such is the nature of a square wave LFO.

The modulation wheel

If you have a MIDI controller, chances are it comes with a round control that is dedicated to modulation: the mod wheel. Usually this is located next to the pitch bend wheel.

The difference between these two is that the pitch bend wheel will always return to the center and is mostly useful for making temporary changes to the sound, primarily bending the pitch, while the mod wheel stays where you leave it and can be used for many different kinds of modulation jobs.

In JX11 the mod wheel is used for additional vibrato. The mod wheel sends a regular MIDI CC message, and so it can be in 128 possible positions.

In **Synth.h**, create a private variable to store the current value of the mod wheel:

```
float modWheel;
```

In **Synth.cpp**, add the following line to `reset`:

```
modWheel = 0.0f;
```

It is possible that the mod wheel isn't in the zero position when the synth is started, but there is no way for us to find out what position it is in. There is no "ask the MIDI controller for the mod wheel position" message. The only way to know the mod wheel position is when it changes and the MIDI controller or the DAW sends you a MIDI CC message.

You've already added support for handling MIDI CC messages coming from the sustain pedal. Handling the mod wheel is very similar. Just add a new case to the `controlChange` method:

```
// Mod wheel
case 0x01:
    modWheel = 0.000005f * float(data2 * data2);
    break;
```

As with all MIDI CC messages, the value of the controller is in the `data2` byte and is between 0 – 127. Here the possible values are converted into a parabolic curve so that you have more control over small values than large ones. It's the same idea as adding skew to a parameter, except here you're not dealing with a parameter.

When the modulation wheel is in its lowest position, `modWheel` is 0. When the wheel is fully open, `modWheel` is 0.0806. This amount will be added to the LFO intensity for vibrato / PWM. That happens in `updateLFO`.

Change the lines that calculate the `pwm` and `vibratoMod` values to the following:

```
float vibratoMod = 1.0f + sine * (modWheel + vibrato);
float pwm = 1.0f + sine * (modWheel + pwmDepth);
```

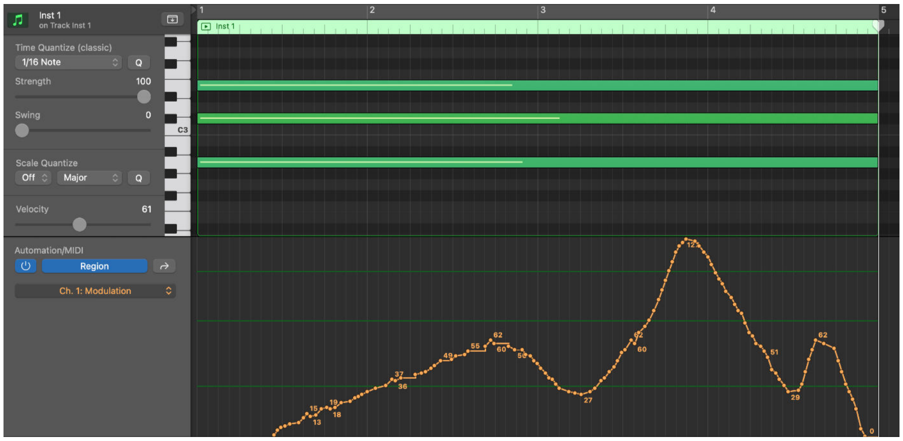
Recall that `vibrato` and `pwmDepth` have values between 0 and 0.05. Here, you add the contribution of `modWheel`, with is roughly on the same scale. I'm not exactly sure why the modulation wheel has a slightly larger range than what you get from the **Vibrato** parameter. Another design choice, there is no written standard for any of this.

Combined with the mod wheel, the `vibratoMod` and `pwm` multipliers now range between 0.869 and 1.131, so the total amount of vibrato is a little more than two semitones up and down: one semitone from the **Vibrato** parameter and another from the mod wheel.

Try it out! Play a note without vibrato, then slowly roll in the mod wheel to increase the amount of vibrato. If the **Vibrato** parameter is set to PWM mode, this lets you do vibrato and PWM at the same time.

This is an example of where one modulation source, the mod wheel, affects the intensity of another modulator, the LFO. So here you have a modulator modulating another modulator!

If you don't have a MIDI controller with a mod wheel, you can still test this in your DAW. Consult your DAW's manual for details, but in general you can add MIDI CC messages to MIDI regions. Alternatively, you may be able to add automation for MIDI controllers such as the mod wheel. Here, I recorded some mod wheel vibrato in a MIDI region in Logic Pro:



Inst 1
on Track Inst 1

Time Quantize (classic)
1/16 Note
Strength 100
Swing 0
Scale Quantize
Off Major
Velocity 61
Automation/MIDI
Region
Ch. 1: Modulation

MIDI CC events for the mod wheel in a Logic Pro MIDI region

Glide (portamento)

The MIDI note number is also a modulation source. Obviously, the note number's main job is to set the pitch of the note being played. But it can be used for other things too: in the next chapter you'll use it to set the cutoff frequency of the filter.

Glide is a common feature of synths that lets you, well, glide between different notes. If you play a C followed by a G, the note will smoothly glide from the pitch of C to the pitch of G with a certain speed. This is also called portamento.

This feature can be implemented in two ways: glide between MIDI note numbers, where it is possible to have fractional MIDI note numbers, or glide directly between the pitches. In JX11 you'll use MIDI note numbers to determine the starting pitch and destination pitch, and the actual glide is between these two pitches.

I'm not sure if glide counts as a kind of modulation, but in JX11 the glide logic will be performed inside the `updateLFO` method, so this chapter seems like an appropriate place to cover it.

When should glide happen? There are three possibilities:

- Never. The synth does not glide from one note to the other. This is the same as setting the glide time to zero so that the glide is instantaneous.
- When playing legato-style. Glide is only applied when the previous key is still held down while the new note is played.
- Always. This glides even if the previous key has already been released.

The user can choose one of these options using the **Glide Mode** parameter. The **Glide Rate** parameter sets the speed of the glide. And finally, there is a **Glide Bend** parameter that always adds a slide up or down by a fixed number of semitones, and is used regardless of the glide mode.

Let's add variables for these parameters to `Synth`. In **Synth.h**, add the following public member variables:

```
int glideMode;
float glideRate;
float glideBend;
```

You will fill these in from the usual place, the `update` method in **PluginProcessor.cpp**. Add this somewhere below the LFO stuff because it needs the `inverseUpdateRate` variable, as the glide calculations are also done 32 times slower than the audio rate.

```
synth.glideMode = glideModeParam->getIndex();

float glideRate = glideRateParam->get();
if (glideRate < 2.0f) {
    synth.glideRate = 1.0f;  // no glide
} else {
    synth.glideRate = 1.0f - std::exp(-inverseUpdateRate * std::exp(6.0f - 0.07f * glideRate));
}

synth.glideBend = glideBendParam->get();
```

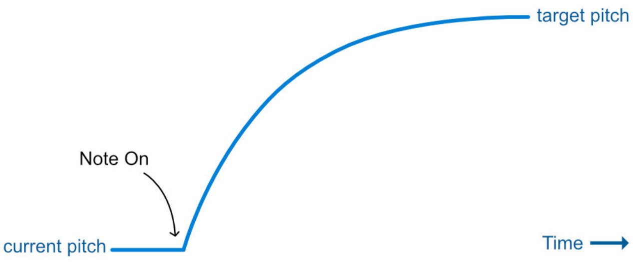
The **Glide Mode** parameter is an `AudioParameterChoice` object, and so `glideMode` is a simple integer where 0 = off, 1 = when legato-style playing, 2 = always.

Just like the envelope, glide is implemented using a one-pole filter, except one that is updated every 32 samples instead of on every sample. The user specifies the glide speed as a percentage using the **Glide Rate** parameter. Here, you use that to calculate `glideRate`, which is the coefficient for this one-pole filter. A smaller coefficient means the glide takes longer. Refer back to the envelope chapter if you need a refresher on how this works.

The **Glide Bend** parameter goes from –36 semitones to +36 semitones, making `glideBend` simply a value between –36 and +36.

JX11 can perform glide both in mono and poly modes. In mono mode it's obvious which note to glide from, but in poly mode there are again options. Some synths glide from the note that was previously used for the selected voice, other synths glide from whatever note was played most recently overall. JX11 does the latter.

As you've seen in the envelope chapter, a one-pole filter smoothly transitions from its current value to a target value over a certain amount of time in an exponential fashion:



Using a one-pole filter to glide between two pitches

For glide, the current and target values are the pitch, or rather the period, of the voice. This means you'll need to add a new property to `Voice` for this target value.

In **Voice.h**, add the following member variable for the desired period:

```
float target;
```

In `Synth::startVoice`, rather than setting `voice.period` to the correct value for the new note, you'll set `voice.target` and then let the one-pole filter interpolate from the current period to the new target period.

First, in **Synth.h**, add a new private member variable to `Synth`:

```
int lastNote;
```

This will keep track of the most recent MIDI note number that was played, which gives you the period to start gliding from. In **Synth.cpp**, in `reset`, set this to zero so that no glide will happen for the very first note played:

```
lastNote = 0;
```

Next, set the target period in `startVoice`. There are a bunch of things to change here:

```
void Synth::startVoice(int v, int note, int velocity)
{
    float period = calcPeriod(v, note);

    // 1
    Voice& voice = voices[v];
    voice.target = period;

    // 2
    int noteDistance = 0;
    if (lastNote > 0) {
        noteDistance = note - lastNote;
    }

    // 3
    voice.period = period * std::pow(1.059463094359f, float(noteDistance) - glideBend);

    // 4
    if (voice.period < 6.0f) { voice.period = 6.0f; }

    // 5
    lastNote = note;
    voice.note = note;
    voice.updatePanning();

    // ...rest of the method...
```

Let's take this step-by-step:

1. Instead of assigning `voice.period = period;` this now sets `voice.target` to the desired period.

2. Calculate how far away the new note is in semitones. This uses the `lastNote` variable, except on the very first note that's played when `lastNote` is zero. There is nothing to glide from if there was no previous note.

3. Set `voice.period` to the period to glide from. This is necessary because in polyphony mode, the voice may not have the period of the most recent note that was played, if that note was handled by another voice.

   You should recognize this formula as period$\times 2^{N/12}$ where N is the number of semitones to go up or down from the starting period, except here again $2^{N/12}$ is written as 1.059463094359^N.

   The number of semitones is given by the `noteDistance` that you just calculated, minus any additional semitones from the **Glide Bend** parameter. It's minus instead of plus because to glide from high to low — a positive value of `glideBend` — the period must become longer.

4. Make sure the starting period does not become too small. Recall that you did something similar in `calcPeriod`. Unlike the target period, this doesn't need to be exact, so you can simply limit it to the minimum of 6 samples.

5. Assign the new note number to `voice.note` like before, but also to `lastNote`.

Since gliding is something that happens per-voice, it makes sense that the logic for performing the glide happens in the `Voice` object.

In **Voice.h**, add a new variable to the struct. This is a copy of `Synth`'s `glideRate` variable so that the voice will be able to access it.

```
float glideRate;
```

Also add a new method to `Voice`:

```
void updateLFO()
{
    period += glideRate * (target - period);
}
```

This is the one-pole filter formula that creates an exponential transition curve between the two pitches. If the voice's current `period` is not yet equal to the `target` value, this equation brings it a little closer with every update step. It doesn't matter whether `period` is larger or smaller than `target`, it works in both directions.

The voice will always perform this calculation, even if glide is disabled. In that case, the `glideRate` is 1.0, and so the voice's period is immediately set to the target value.

You need to call this function from `Synth`'s `updateLFO`, so head back to **Synth.cpp** and do this in the loop for the voices in `updateLFO`:

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        voice.osc1.modulation = vibratoMod;
        voice.osc2.modulation = pwm;

        // add these lines
        voice.updateLFO();
        updatePeriod(voice);
    }
}
```

Because `voice.updateLFO()` possibly changes `voice.period`, you have to assign the new period to the oscillators again. That's done in the new method `updatePeriod`. For efficiency's sake, add this as a private inline method to **Synth.h**:

```
inline void updatePeriod(Voice& voice)
{
    voice.osc1.period = voice.period * pitchBend;
    voice.osc2.period = voice.osc1.period * detune;
}
```

This does exactly the same thing as what happens at the top of `Synth::render`, so to avoid code duplication replace those two lines with a call to `updatePeriod(voice);` too.

The reason you have to do `updatePeriod` following `voice.updateLFO()`, is that otherwise the oscillators would only have their period updated at the start of the block. If the block size is relatively large, the glide no longer sounds smooth but happens in audible steps — making it more of a glissando than a portamento.

One more thing to do before you can test it out, and that is to set the `glideRate` on the voice object. Since this is something that is derived from a plug-in parameter, this should happen at the top of `render`.

In `Synth::render` add the following line to the `for` loop at the start:

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        voice.osc1.period = voice.period * pitchBend;
        voice.osc2.period = voice.osc1.period * detune;
        voice.glideRate = glideRate;              // add this line
    }
}
```

All right, let's try it out! Don't bother choosing a **Glide Mode** yet, currently the synth will always glide. Put the synth in **Poly** mode, set the **Glide Rate** to 60%, and play a few notes. It's easiest to hear if you play notes that are far apart. Also try the **Glide Bend** parameter. The effect of this is clearest if you play the same note a few times in a row.

The synth shouldn't glide if **Glide Mode** is "Off" and it should always glide when **Glide Mode** is "Always". Those two situations are easy to handle. However, for "Legato" mode, you need to detect whether the user is playing legato-style.

Let's add a helper method to `Synth` for that:

```
bool Synth::isPlayingLegatoStyle() const
{
    int held = 0;
    for (int i = 0; i < MAX_VOICES; ++i) {
        if (voices[i].note > 0) { held += 1; }
    }
    return held > 0;
}
```

For a new Note On event to count as being legato-style, at least one key must still be held down for a previous note. This method simply counts how many of the currently playing voices are for keys that are still held down, i.e. how many voices did not get a Note Off event yet. Recall that if the `note` property is 0, this voice is not playing; if it's –1, or `SUSTAIN`, the voice is still playing but is only sustained by the pedal (which doesn't count as legato).

Don't forget to add the declaration for `isPlayingLegatoStyle` to **Synth.h** as well. This should go in the private section:

```
bool isPlayingLegatoStyle() const;
```

Now in `startVoice`, make the following change to the code that calculates `noteDistance`:

```
int noteDistance = 0;
if (lastNote > 0) {
    if ((glideMode == 2) || ((glideMode == 1) && isPlayingLegatoStyle())) {
        noteDistance = note - lastNote;
    }
}
```

If `glideMode` is 0 (off), then all of this is skipped and `noteDistance` will be 0, meaning that there is no glide except for any additional glide set by the **Glide Bend** parameter.

If `glideMode` is 2 (always), then the glide is always performed. But if `glideMode` is 1 (legato), the glide only happens if `isPlayingLegatoStyle` is true.

Almost done, but there's something you should fix for monophonic mode. When playing legato-style in mono mode, the `restartMonoVoice` function is called instead of `startVoice`. This should also set the `voice.target` value to the new period.

Change `restartMonoVoice` to the following:

```
void Synth::restartMonoVoice(int note, int velocity)
{
    float period = calcPeriod(0, note);

    Voice& voice = voices[0];
    voice.target = period;   // change this

    if (glideMode == 0) { voice.period = period; }  // add this

    voice.env.level += SILENCE + SILENCE;
    voice.note = note;
    voice.updatePanning();
}
```

There's no need to use `lastNote` here, since there is ever only one voice playing, and so the synth can glide from whatever the voice's current period is. If gliding is disabled, it sets `voice.period` to the same target value, which will skip the glide effect. Note that this does not use the additional **Glide Bend** amount.

That's it for glide. Try it out! All the glide modes should work now, both in Poly and Mono modes. Also try using the pitch wheel and vibrato during a glide. Each of these modulations should work together without a hitch since they're described by their own independent variables, and the final oscillator period is made by multiplying them all together. Happy gliding!

> **Note:** What happens to the panning when you glide between two notes that are far apart? While the pitch smoothly glides from the first note to the next, the panning immediately jumps to the new value. Exercise for the reader: also glide the panning amount, so you can hear the notes slide left-to-right too. You'll need to call the voice's `updatePanning()` method from `updateLFO` now. Good luck!

Changing a parameter using MIDI CC

Depending on how fancy your MIDI controller is, it might have a master volume slider. This control is mapped to MIDI CC `0x07`. One thing you could do with this, is read the value from the MIDI CC message and put it into a new variable `volumeCtl` that goes between 0 and 1.5 or so, and then multiply that with the other variables that affect the amplitude:

```
voice.osc1.amplitude = volumeTrim * vel * volumeCtl;
```

The master volume control now modulates the amplitude of the voices by making them quieter (values between 0 and 1) or louder (between 1 and 1.5). This is very similar to the other MIDI CC modulations you've seen so far and is an acceptable solution. If you feel up to a challenge, try implementing this yourself.

However, for this book I thought it would be useful to show an example of using MIDI CC to change a plug-in parameter, in this case the **Output Level** parameter. You'll make it so that the MIDI CC for master volume will do the same as dragging the **Output Level** slider by hand, and can be used to boost or cut the overall sound level.

This requires a different approach than before. Until now the parameters were changed by the user or the host, and the plug-in was notified about this in the `update` method. But here it's the other way around: the plug-in will be the one changing the parameter in response to a MIDI message and it should notify the host about this.

The parameters are owned by the audio processor, so it's easiest to handle this feature there. Add the following code to **PluginProcessor.cpp** in the `handleMIDI` method.

```
void JX11AudioProcessor::handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2)
{
    // Control Change
    if ((data0 & 0xF0) == 0xB0) {
        if (data1 == 0x07) {  // volume
            float volumeCtl = float(data2);
            DBG(volumeCtl);
        }
    }

    // ...rest of the method...
}
```

If the MIDI message is a Control Change event for controller `0x07`, then this prints out the data value to the debug console. Try it out, to verify this actually works with your MIDI controller.

It’s possible the master volume control may not work in your host. Or perhaps there is no master volume on your MIDI controller, but you have other sliders or knobs. If so, simply replace the `0x07` with the identifier number for the slider or knob.

To find out what the MIDI CC identifier is, use a `DBG` statement to print out the first data byte of the MIDI CC message and see what this says when you twiddle the knobs on your controller. Keep in mind that `DBG` prints out the number as decimal, not hex.

```
if ((data0 & 0xF0) == 0xB0) {
    DBG(data1);
}
```

In the chapter on user interfaces you’ll learn how to make this more flexible with a MIDI Learn function, so that users can assign their own MIDI CCs to the plug-in parameters. For now, a hardcoded MIDI CC will suffice.

Next, you need to do two things:

1. Map the `volumeCtl` value, which goes from 0 to 127, into the range of the **Output Level** parameter, which goes from –24 dB to +6 dB. Since a slider on the decibel scale should be linear, this is a straightforward conversion.
2. Change the parameter and tell the host about this change. Notifying the host is important, so that the host can update its own UI in response. For example, the user might be recording automation for this parameter, and notifying the host lets it know there is new automation data.

You can do both of these things with a single JUCE command, `setValueNotifyingHost`. In the code that handles the MIDI CC message, change the following:

```
float volumeCtl = float(data2) / 127.0f;
outputLevelParam->beginChangeGesture();
outputLevelParam->setValueNotifyingHost(volumeCtl);
outputLevelParam->endChangeGesture();
```

Internally, JUCE stores parameters as a floating point value between 0 and 1. This is for compatibility with certain older plug-in formats and host programs. Normally you don't have to worry about this and you'd use the range you set on the `AudioParameterFloat` object, which here is –24 to +6. However, `setValueNotifyingHost` expects a value in the range 0 – 1.

The `AudioParameterFloat` class has `convertTo0to1` and `convertFrom0to1` helper functions to convert between this 0 – 1 value and the more human-friendly range. However, if you divide the value from the MIDI CC message by 127, it's already between 0 and 1, and you can directly pass it into `setValueNotifyingHost`. No need to convert to decibels and back again.

> **Note:** In addition to `setValueNotifyingHost`, the audio parameter object has a `setValue` method. You shouldn't call this yourself; it's intended for use by hosts when they want to change the value of the parameter. Also note that you did use `convertTo0to1` once before already, in `setCurrentProgram` when loading a new preset.

The `beginChangeGesture` and `endChangeGesture` methods are there to let the host know that a parameter is about to be changed. This is mostly intended for things like knobs and sliders in the GUI: When the user clicks on the knob to start dragging it, you'd send a `beginChangeGesture`. As the knob is being dragged, you'd send `setValueNotifyingHost`. And when the knob is released, you'd send `endChangeGesture`.

That situation doesn't apply here, since all we get is a MIDI CC message whenever the control has moved, and we don't know when the user stops interacting with it. But we call these methods anyway for good measure.

Try it out! Moving the master volume slider on your MIDI controller will now directly change the **Output Level** parameter. You should see this parameter immediately jump to its new value in the plug-in's UI. Also try it out in your DAW by recording some animation.

There is a lot more to say about modulation, and in the next chapter you'll learn all about the synth's filter and how that gets modulated too.

Note: In this section we were a little naughty. I've mentioned a few times already that the audio thread should never ever be blocked, as that may cause the plug-in to miss its deadline and give nasty glitches. It's also not safe to modify UI things from the audio thread, and vice versa. Fortunately for us, `setValueNotifyingHost` is thread safe. However, it does have a lock in it… Whoops! This lock is taken while notifying any listeners — such as the host — about the change to the parameter's value.

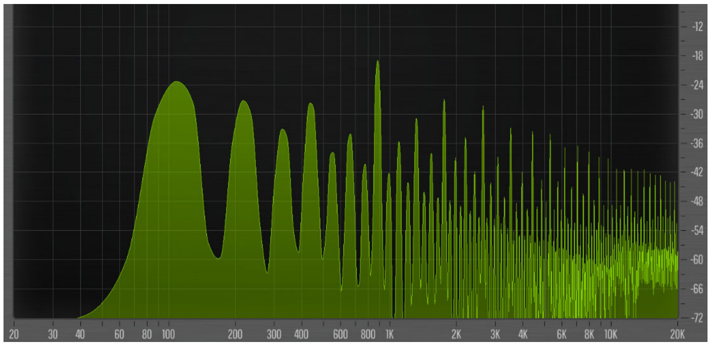
The question is: how big of a problem is this lock? Most of the time, there is no contention for the lock and so the audio thread will not have to wait while trying to obtain it. The only time that the audio thread may fail to get the lock, and will therefore be stalled for an indeterminate amount of time, is when the user opens or closes the plug-in's editor window.

If opening or closing the editor were to happen exactly when a MIDI CC message is received for the master volume control, the UI thread and the audio thread might end up fighting over the lock, and there's a tiny chance the audio will glitch. Since that is such an unlikely occurrence, we can get away with calling `setValueNotifyingHost` from the audio thread, even though technically it's not 100% correct.

Chapter 12: The filter

The most exciting part of an analog synth is the filter. The oscillators are fine and good, but the real character of the sound comes from careful application of the filter.

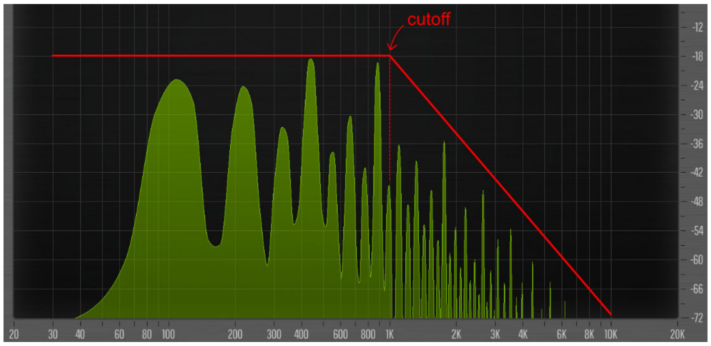
The purpose of a filter is to make certain frequencies louder and others quieter. This works best if there are many frequencies in the sound to begin with. Given that we have a nice sawtooth oscillator and can now play multiple notes at a time, the frequency spectrum is lush and full of activity:



The spectrum with lots of lovely harmonics

The filter in JX11 is a resonant low-pass filter, meaning that it keeps all the frequencies up to a certain point intact but any higher frequencies are removed. This point is known as the cutoff. The frequencies around the cutoff point are boosted if the resonance setting of the filter is made high enough.

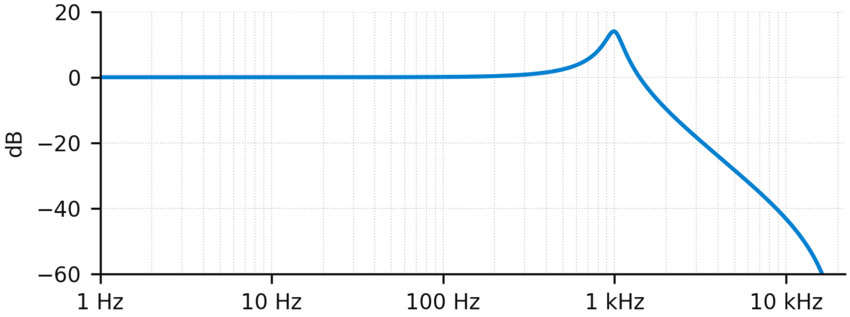
Here is the same spectrum again but now with the filter applied:



The spectrum with a low-pass filter applied

The frequencies that are higher than the cutoff point do not completely disappear but fall off at certain rate. This rate is also known as the slope of the filter. For the filter used in JX11, which is a second order filter, this slope is 12 dB per octave. For example, if the cutoff is at 2000 Hz, the frequencies one octave higher at 4 kHz are 12 dB quieter. Another octave higher (at 8 kHz) they are 24 dB quieter, and so on.

To describe the behavior of a filter we usually plot the frequency response of the filter. It lets us view how the filter affects all the possible frequencies.



The frequency response of a second-order filter

This is the frequency response of the JX11 filter with a cutoff frequency of 1000 Hz, and a resonance or Q setting of 5. When the line is at 0 dB, it means there is no change in loudness. This area of the curve is known as the passband, because the frequencies are passed through unchanged. Since this is a low-pass filter, the passband is made up of the frequencies below the cutoff point.

When the line in the frequency response plot becomes larger than 0 dB, it means the frequencies are made louder. When the line dips below 0 dB, it means those frequencies are made quieter. As a quick rule of thumb: +6 dB means the amplitude is two times as high, –6 dB means the amplitude is cut in half. The area of the curve where the frequencies are reduced in amplitude, here anything to the right of 1000 kHz, is known as the stopband.

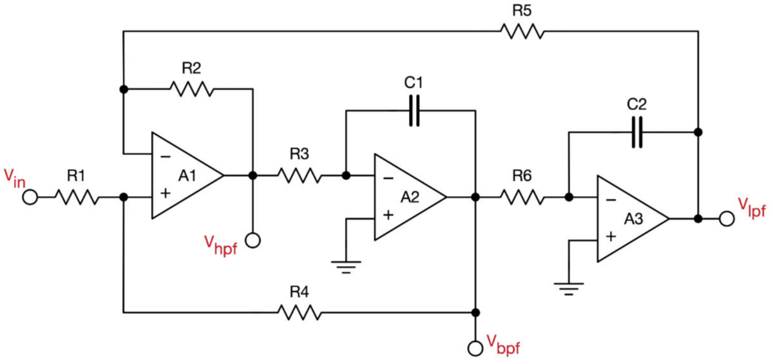
Do you remember impulses from the sawtooth chapter when you used them to build the oscillator? Fun fact: The above frequency response was made by sending a single impulse through the filter to get the impulse response, and then an FFT or Fast Fourier Transform was used to draw the frequency spectrum of this impulse response. From this frequency response plot you can easily tell what frequencies get amplified and which ones are attenuated.

There are several ways to create a resonant second-order low-pass filter. In most DSP books you will find many a chapter dedicated to filter theory and the so-called biquadratic or biquad filter structure. For many non-audio DSP filtering tasks, the biquad is a perfectly adequate filter. However, biquads have unfortunate side-effects when combined with the kind of thing music producers love to do: modulation.

If the biquad filter's cutoff frequency is modulated too quickly, the filter may "blow up" and suddenly output extremely large values, resulting in screaming feedback coming out of the speakers. Yikes! Needless to say, this is something you'd want to avoid at all costs.

Unfortunately, when using a biquad and modulation together, you cannot guarantee this kind of blow-up won't happen. That's not to say biquad filters are not useful — in fact, the one-pole filter you've used previously to make the envelope and glide feature is a simplified biquad — but they're no good as "musical" filters whose main purpose is shaping the synthesizer sound, as that generally involves modulation.

Instead, JX11 uses a different filter design called the State Variable Filter or SVF. Like most filter designs, it is based on an analog electronic circuit. The original SVF filter schematic looks like this:



Electronics diagram for the State Variable Filter

The triangles are opamps or operational amplifiers, a staple building block of analog circuits. The diagram also has resistors and capacitors in it. Instead of sampled data, an analog circuit works with voltages, which are continuous signals. The first amplifier is used to sum up the input voltage and the voltages from the feedback loops. The other two op-amps acts as integrators.

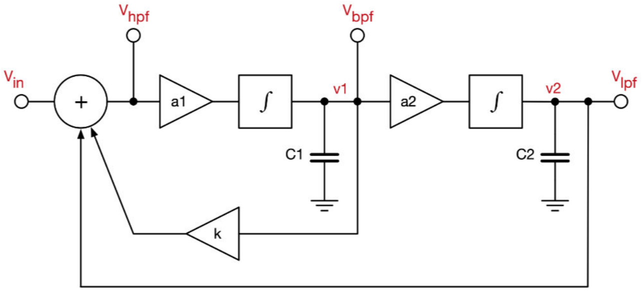
A question you might be asking yourself is: Why bother with an analog schematic? We're doing DSP, where the D stands for *digital* signal processing, not analog. The reason for starting with analog filter designs is that these filters have been around for a long time, they work great in analog gear, sound good, and are well understood — so it makes sense to turn them into digital versions.

One advantage of the SVF design is that it does three types of filtering at the same time: it produces a high-pass filtered output V_{hpf}, a band-pass filtered version V_{bpf}, and finally a low-pass filtered version V_{lpf}. The low-pass output is what you will use in JX11.

So how does one convert such an analog filter design into a digital filter that can be implemented in software? There are many ways but they all involve fancy math of some kind. You're not really expected to be able to do this math yourself, although you could certainly learn how to, if you're interested in this kind of thing.

Most texts about DSP start out from an analog schematic such as the one shown here, and use math such as the bilinear Z-transform (also known as BLT or BZT) to convert an analog transfer function into a digital transfer function, which can then be implemented in software. This technique is primarily used with biquads. For the SVF a different technique must be used, due to the way its feedback loops are constructed.

For JX11 you will use the digital SVF filter that was designed by Andrew Simper of Cytomic, commonly referred to as the Cytomic SVF. What Andrew did was first convert the analog schematic into something simpler:



The simplified SVF diagram

The first op-amp is replaced by a summer. The other two op-amps are replaced by integrators. For the purposes of this method, we're assuming the op-amps have ideal behavior, which means that, if there is a voltage difference between their + and – terminals, the output voltage of the opamp changes to compensate for this. In other words, the op-amp continually tries to keep its + and – terminals at the same voltage, and it does this using the feedback loop.

We can therefore assume that the voltage on both terminals of the op-amp will always be the same. Because these op-amps have their + terminal connected to ground, in this simplified schematic the capacitors will be connected to ground as well. To find the V_{bpf} and V_{lpf} outputs — for band-pass and low-pass filtering, respectively — we then only need to calculate what the voltages over these capacitors are and how they change over time. V_{hpf} can be derived from a combination of V_{bpf}, V_{lpf}, and the input voltage.

This technique is called nodal analysis because it uses formulas about the behavior of voltages and currents in electrical circuits — such as the famous Ohm's Law that you may have heard of — to calculate what the voltages at different points in the circuit must be. One of those voltages represents the low-pass filtered output that we're looking for.

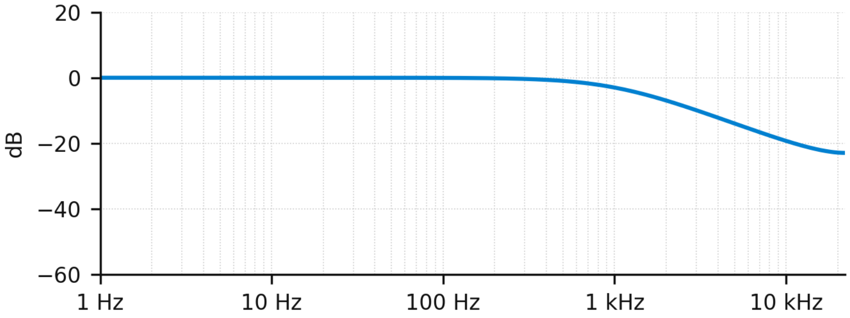
Besides nodal analysis there are several other techniques for analyzing the voltages inside electronic circuits. This kind of thing is done a lot in audio programming, not just for filters but also for other kinds of circuits such as guitar effects pedals. This practice is called circuit modeling and it's popular because people love the sound of old analog gear. The more powerful computers become, the more accurately we can simulate these old analog circuits.

We're not going to cover the derivation of the formulas used in the nodal analysis of this filter. I just wanted to give you a taste of where these filter designs come from. If you're interested, you can read Andrew's SVF paper[48]. I suggest reading this one[49] first, as it explains the method in more detail. Warning: these documents are quite dense and the programming language used is Mathematica, not C++. If you try to read these papers and feel completely overwhelmed, don't worry, that's a normal first reaction.

By the way, the Cytomic SVF filter will sound exactly the same as a low-pass biquad filter. Both use the bilinear Z-transform in their derivations, and so they have the exact same frequency response. However, the SVF has the wonderful property that it's stable under heavy modulation, so that's one worry less about hurting someone's eardrums.

You might be wondering how different the one-pole filter is from the SVF. The one-pole filter is really more of a utility filter than a sound filter. For the envelope you used it because it's a convenient and fast way to make an exponential curve; for glide you used it to smoothen the change between two pitches. None of these jobs was about changing the frequencies that are present in the sound.

You can definitely use a one-pole filter to change the frequencies in the sound but it's not very capable. Since it's a first-order filter it has a slope of 6 dB/octave compared to 12 dB/octave of the SVF, so it doesn't drop off as steeply. It also doesn't have a nice frequency response. Compare the frequency response of the SVF shown above with that of a one-pole filter, also set to 1000 Hz:



The frequency response of the one-pole filter

As you can see, the one-pole filter is much less effective at removing the frequencies above the cut-off point, and this only becomes worse as the cutoff gets higher. There is

[48] http://cytomic.com/files/dsp/SvfLinearTrapOptimised2.pdf
[49] http://cytomic.com/files/dsp/OnePoleLinearLowPass.pdf

no resonant boost, since first-order filters can’t do that. The one-pole filter is mostly useful for smoothing things out with an exponential curve — which happens at a relatively low cutoff frequency — but not for filtering sound.

Adding the filter

Use **Projucer** to add new header file to the project and name it **Filter.h**. I’m going to give you the entire filter source code at once. Here it is:

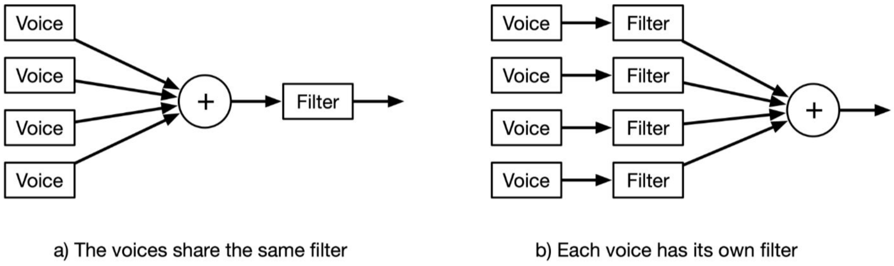
```
#pragma once

class Filter
{
public:
    float sampleRate;

    void updateCoefficients(float cutoff, float Q)
    {
        g = std::tan(PI * cutoff / sampleRate);
        k = 1.0f / Q;
        a1 = 1.0f / (1.0f + g * (g + k));
        a2 = g * a1;
        a3 = g * a2;
    }

    void reset()
    {
        g = 0.0f;
        k = 0.0f;
        a1 = 0.0f;
        a2 = 0.0f;
        a3 = 0.0f;

        ic1eq = 0.0f;
        ic2eq = 0.0f;
    }

    float render(float x)
    {
        float v3 = x - ic2eq;
        float v1 = a1 * ic1eq + a2 * v3;
        float v2 = ic2eq + a2 * ic1eq + a3 * v3;
        ic1eq = 2.0f * v1 - ic1eq;
        ic2eq = 2.0f * v2 - ic2eq;
        return v2;
    }
private:
    const float PI = 3.1415926535897932f;

    float g, k, a1, a2, a3;  // filter coefficients
    float ic1eq, ic2eq;      // internal state
};
```

That’s the whole thing. Not so bad after all, eh? This is what the different methods do:

- `reset`. Like every other DSP building block you’ve used so far, `Filter` has a reset method. This sets the coefficients and internal state back to zero.
- `updateCoefficients`. Whenever the cutoff frequency or the amount of resonance changes, the filter’s coefficients must be recalculated. Previously for the one-pole filter there was a single coefficient, but here there are five: `g`, `k`, `a1`, `a2`, `a3`. The `g` coefficient is derived from the cutoff frequency and the sample rate; the `k` coefficient is derived from the resonance; `a1`-`a3` are used by the various calculations later on.
- `render`. This takes an input sample value and applies the filter to it. Understanding the math used here isn’t very important. Suffice to say that these magical equations are what perform the filter operations. The `v1`, `v2`, and `v3` variables correspond to the voltages at the different nodes in the circuit diagram shown before. Important for us is that `v2` contains the low-pass filtered “voltage”. Of course, you’re not working with voltages but with samples, but these samples ultimately do represent voltage levels once the data goes into the DAC and drives the loudspeaker.

The other important thing that happens in `render` is that it updates the filter’s internal state, `ic1eq` and `ic2eq`, which keep track of the charge on the capacitors. Essentially the formulas in `render` perform a simulation of the original electronic circuit.

To use this `Filter` object, you must call `updateCoefficients` at an opportune moment and pass the sample values through the `render` method. There are generally two places where filtering can happen in a synth. It can be done once, after all the voices have been mixed together, or each voice can have its own filter instance.



One filter per voice or one filter total?

In JX11 you will give each voice its own filter because the synth will be doing heavy modulation of the filter's cutoff frequency and this needs to happen independently for each voice. For a synth that doesn't do such modulations, using a single overall filter might be sufficient and is less computationally expensive.

> **Note:** JX11 renders its voices in mono, so each voice only needs one filter. To make a synth with stereo voices, the left and right channel both must have their own filter instances. This is a common mistake made by people new to audio programming. If you ever find yourself using a filter and the sound keeps breaking up or there's a lot of crackling, chances are you're using the same filter on more than one channel. That's a problem because the filter needs to keep track of state — in the SVF this is the `ic1eq` and `ic2eq` variables — and you don't want the state from the right channel to overwrite the state of the left channel. Hence the need for two separate filter objects if you're filtering stereo sound.

In **Voice.h**, add an include for the new filter source file:

```
#include "Filter.h"
```

Also add a new member variable to the struct:

```
Filter filter;
```

In `reset`, make sure to reset the filter:

```
filter.reset();
```

And add the following line into `updateLFO`:

```
filter.updateCoefficients(1000.0f, 0.707f);
```

This will set the cutoff frequency to 1000 Hz and the Q value for resonance to 0.707. You're doing this just for testing that the filter works correctly, later you'll make these settings configurable.

The number 0.707 is a value you should become familiar with. It's `1/sqrt(2)` or `sqrt(1/2)`, both are the same thing. This number shows up a lot in DSP. In the case of Q it means there is no resonance. To get resonance, you'd make Q higher than 0.707. Setting Q lower than

0.707 also moves the cutoff point, so we'll say 0.707 is the minimum allowed value for Q in this synth.

The reason you call `updateCoefficients` in `updateLFO` is that the math for setting the coefficients is usually kind of heavy, in this case involving a `std::tan` function. Keep in mind that each playing voice will have to do this for its own filter, so that's up to eight times the amount of work. But it's not such a big deal when you do these calculations only once every 32 samples.

The final change in **Voice.h** is to perform the filtering inside `render`:

```
float render(float input)
{
    float sample1 = osc1.nextSample();
    float sample2 = osc2.nextSample();
    saw = saw * 0.997f + sample1 - sample2;

    float output = saw + input;

    output = filter.render(output);   // add this line

    float envelope = env.nextValue();
    return output * envelope;
}
```

Everything is as before except now the sample value gets sent through the filter too. It doesn't matter if you do this before or after the envelope. This is another important DSP principle: if two systems are linear time-invariant or LTI — and both the filter and envelope are — then the order in which you send the signal through these two systems is not important. The result will be exactly the same whether the filter comes before the envelope, or the other way around.

What you have here is the completed version of `Voice::render`, you won't be adding anything more to it in this book. Notice how simple and clean it is, even though it does a lot? It renders the two oscillators, combines them into sawtooth waves or a square wave, filters the sound, and applies the envelope. You could replace each of these building blocks with different ones, for example a different type of envelope or a different filter, but the `render` function would remain the same.

There are a couple of small things you need to fix elsewhere in the code before you can try it out. In **Synth.cpp**, change `allocateResources` to the following.

```
void Synth::allocateResources(double sampleRate_, int /*samplesPerBlock*/)
{
    sampleRate = static_cast<float>(sampleRate_);

    // add these lines
    for (int v = 0; v < MAX_VOICES; ++v) {
        voices[v].filter.sampleRate = sampleRate;
    }
}
```

To be able to calculate the coefficients, `Filter` needs to know the sample rate. Here, you loop through the voices and tell each filter object what the current sample rate is.

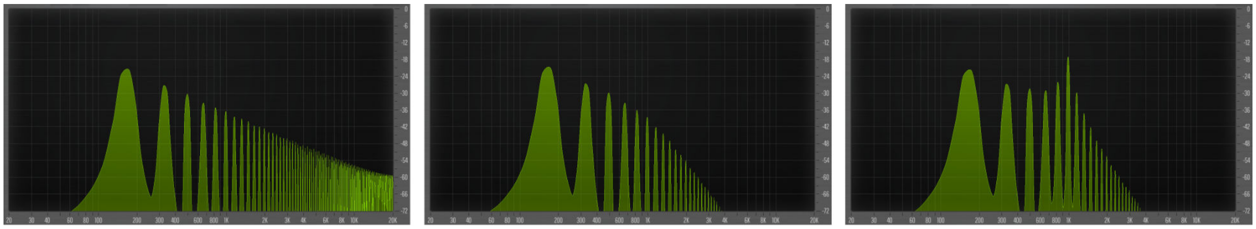
> **Note:** When writing audio code, you always find yourself having to pass the sample rate to the various DSP building blocks. It might be tempting to put the current sample rate into a global variable, so that all objects can easily access it, but this is asking for trouble. If your plug-in is used on more than one track in the DAW, each track may be running at a different sample rate — for example because the user decides to oversample a track, running it at 2× or 4× the project rate — and then the global variable will be wrong for some of the plug-in instances. One way to avoid passing around the sample rate everywhere is to make the DSP building blocks independent of the sample rate, for example the filter could use the normalized frequency (a value from 0 – 1) instead of a value in Hz.

Finally, at the end of `Synth`'s `render` function, add this to the loop that turns off the voices whose envelope has dropped below the minimum level:

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (!voice.env.isActive()) {
        voice.env.reset();
        voice.filter.reset();    // add this line
    }
}
```

It's important to reset the filter, since you don't want it to use its old internal state when the voice gets reused later. That's almost guaranteed to give nasty glitches at the start of the new note.

Try it out! You should now see — and hear! — the higher frequencies roll off in the frequency analyzer.

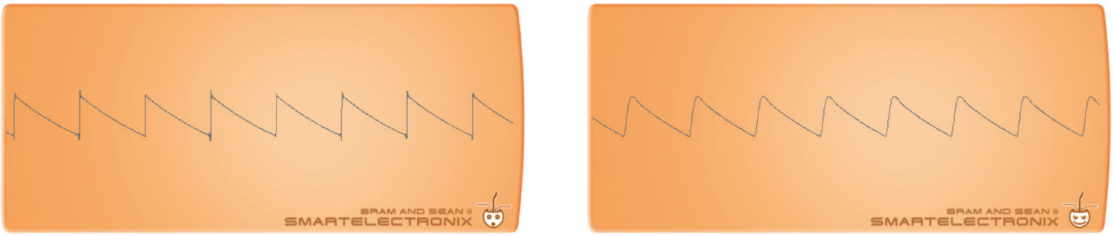


No filter Filter 1 kHz Q = 10

The spectrum of a note with and without filter

Play with different values for the cutoff frequency (anything between 20 and 20000 Hz) and Q (between 0.707 and 50). For a low enough cutoff value, high notes may not be audible at all. The higher Q is, the more you'll get a peak at the cutoff frequency, but be warned that high values of Q can create very loud output. Note that a cutoff frequency higher than the Nyquist limit (half the sample rate) will blow up the filter.

You can also see the effect of the filter in the oscilloscope. The lower the cutoff is, the more the sharp edges will be rounded off.



No Filter Filter 1 kHz

A low-pass filter rounds off the edges

> **Note:** If sharp edges create aliases and the filter rounds off these sharp edges, then why can't you just use a filter to get rid of any aliases from the oscillator? Good question! The problem is that aliases will also occur at frequencies that are lower than the filter's cutoff point. To verify this for yourself, replace the BLIT oscillator with the naive one from the sawtooth chapter, and see if you can still detect aliases even after filtering. That said, a common technique to avoid aliases is to render the audio at a much higher sample rate, followed by a low-pass filter to get rid of frequencies that are too high to be represented by the project's true sample rate. You can use this oversampling technique in JUCE with the `juce::dsp::Oversampling` class.

Key tracking

For some synths, having a single filter with a fixed cutoff frequency is sufficient. However, JX11 is more flexible and uses key tracking: it sets the filter's cutoff based on the pitch of the note that the voice is playing. In effect, this uses the MIDI note number to modulate the cutoff frequency. This is why each voice has its own filter, since different voices will be playing different notes, and therefore each voice has a different cutoff point for its filter.

In **Voice.h**, add a new member variable:

```
float cutoff;
```

In `Voice`'s `updateLFO`, change the line that sets the filter coefficients to the following:

```
filter.updateCoefficients(cutoff, 0.707f);
```

The `cutoff` variable describes the frequency of the cutoff point in Hz. This variable will be filled in when the voice starts, since that's where you have the MIDI note number.

In **Synth.cpp**, somewhere in `startVoice`, add the following line:

```
voice.cutoff = sampleRate / period;
```

This sets the cutoff frequency to exactly the note's pitch. Earlier in this method, `period` was calculated using the MIDI note number, expressed in samples, so dividing this through the sample rate will give a frequency in Hz.

Try it out! No matter how low or high you play the notes, you can always see four or five harmonics above the fundamental because the cutoff frequency moves along with the note.

Now, change the line in `startVoice` to the following:

```
voice.cutoff = sampleRate / (period * PI);
```

Dividing by π makes the cutoff frequency about 3 times lower than the note's pitch. Why are we doing this? This is a choice, possibly inadvertently, made by the original author of the MDA JX10 synth. That version of the synth used a slightly different SVF implementation with a different way to calculate the coefficients, and I think maybe that factor π fell through somehow. Or perhaps it was on purpose, who knows.

If you play some notes now, you'll hardly be able to hear them because the cutoff point is way lower than the note's pitch. That seems a bit pointless, were it not that JX11 has a **Filter Freq** parameter that lets you modify the intensity of the key tracking modulation.

The **Filter Freq** parameter goes from 0 to 100%. Despite its name, it does not directly set the cutoff frequency, as that is done through the note's pitch. But it does let you move the cutoff up or down relative to that point. The higher you set **Filter Freq**, the more harmonics will escape unharmed.

In **Synth.h**, add a new public member variable to `Synth`. This will store the amount by which the key tracking frequency will be shifted up or down:

```
float filterKeyTracking;
```

In **PluginProcessor.cpp**, in `update`, add the following line:

```
synth.filterKeyTracking = 0.08f * filterFreqParam->get() - 1.5f;
```

This converts the percentage from 0 – 100% into the range –1.5 to 6.5. Why these numbers? `synth.filterKeyTracking` acts as a multiplier, but to modify the cutoff frequency you will take the exponential first. That means it can multiply the cutoff anywhere from `exp(-1.5)` = 0.22×, which makes the cutoff even lower than the initial point, all the way up to `exp(6.5)` = 665×, which effectively turns off key tracking because it pushes the filter cutoff far beyond the maximum frequency.

The default setting for **Filter Freq** is 100%, giving a multiplier of 665×. This seems excessively high… If the pitch of the note is 440 Hz, `synth.cutoff` is 140 Hz (the frequency divided by π). Applying the multiplier would set the actual cutoff point of the filter to 93 kHz — much higher than the highest harmonics in the sound, so nothing gets filtered. However, there are going to be other modulators that affect the cutoff as well, and they can modulate downwards. So even if the key tracking is disabled because the multiplier is very high, the cutoff frequency can still be modulated.

To be honest, these numbers by themselves aren't important. The key thing to understand is that `synth.filterKeyTracking` gives us a way to adjust the cutoff point relative to the pitch of the current note. The lower **Filter Freq**, the more the sound gets filtered. These particular numbers were chosen by the original author of MDA JX10. You or I maybe would have chosen something different. What makes instrument design interesting is that for a lot of these features you can't find an "official answer" anywhere, you just have to try things to see what sounds good.

The filter coefficients are calculated in `Voice`, so you need to pass the `filterKeyTracking` value from `Synth` to the active voices. In **Synth.cpp**, in `updateLFO`, change the following.

```
float filterMod = filterKeyTracking;    // add this line

for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        voice.osc1.modulation = vibratoMod;
        voice.osc2.modulation = pwm;
        voice.filterMod = filterMod;    // add this line
        voice.updateLFO();
        updatePeriod(voice);
    }
}
```

The `filterMod` variable has the amount of key tracking. Soon you'll add other things to this variable, which is why it's outside the loop. Just like the other voice-level modulation parameters, you pass the filter modulation amount to each `voice` object. This is done in a new variable, `voice.filterMod`.

In **Voice.h**, first add this new member variable to the struct:

```
float filterMod;
```

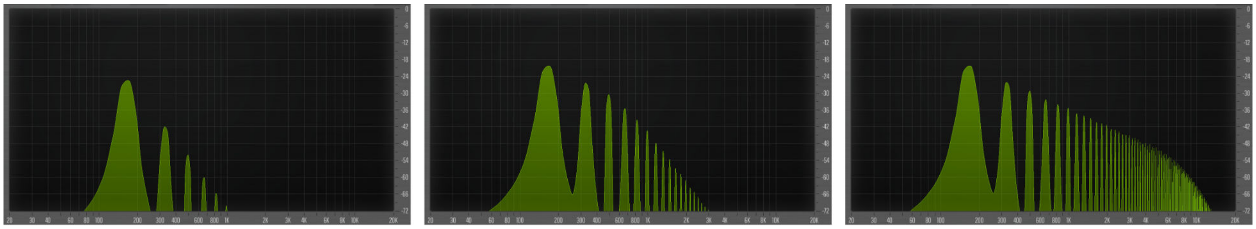
And in **Voice.h**, in `updateLFO`, replace the existing filter code with the following:

```
float modulatedCutoff = cutoff * std::exp(filterMod);
modulatedCutoff = std::clamp(modulatedCutoff, 30.0f, 20000.0f);
filter.updateCoefficients(modulatedCutoff, 0.707f);
```

Instead of directly using `cutoff`, you first calculate a new variable, `modulatedCutoff`. This takes the exponential of the modulation amount to turn it into a multiplier that makes `cutoff` higher or lower. You use the exponent because frequencies are logarithmic, and so modulating them works best exponentially. This is similar to the formula you've seen several times before: freq $\times 2^{N/12}$ to make the pitch higher or lower, except here you're not expressing the modulation amount in semitones.

Because the `std::exp` can make `modulatedCutoff` very large, you use `std::clamp` to restrict the cutoff frequency between 30 Hz and 20000 Hz. Filters tend to do strange things if you set the cutoff frequency to a negative number or to something larger than the Nyquist limit. So you'll play it safe and limit the cutoff to a reasonable range.

Try it out! Play a note and drag the **Filter Freq** slider back and forth. You'll find that the higher the percentage, the more of the sound you can hear.



Filter Freq 30%

Filter Freq 50%

Filter Freq 80%

Spectrum of a note played with different Filter Freq settings

It may not be immediately obvious to understand what cutoff frequencies the **Filter Freq** percentages correspond to. Here are a few guidelines:

- When you set **Filter Freq** to 18.75%, this parameter has no effect, because it makes `synth.filterKeyTracking = 0` and `exp(0) = 1`. The cutoff frequency is now equal to `voice.cutoff`, the note's pitch divided by π.
- At 33%, the cutoff frequency is exactly on the pitch of the note.
- At roughly 90%, the cutoff frequency is so high that you can no longer hear the influence of the filter. At what percentage this happens depends on the note. For very low notes, the **Filter Freq** percentage may need to be a bit higher to keep all the harmonics.

Great, you now have a way to set the cutoff frequency of the filter relative to the note that's playing, and a parameter that determines how intense the filter is. And this is only the beginning of the filter fun!

By the way, if you drag the **Filter Freq** parameter back and forth at a medium pace, you may hear odd transitions. This is another example of zipper noise. Because the parameter update is relatively slow compared to the audio rate, the cutoff frequency can make sudden jumps. As the filter wasn't expecting such a jump, this can result in glitches. You'll fix this later in the chapter by smoothing the frequency changes.

Note: There is an important subtlety hidden in the way the filter's coefficient updates are calculated. This happens in `updateLFO`, which runs 32× slower than the audio sample rate. However, the moment at which a new voice starts playing is not necessarily aligned on a 32-sample boundary. Therefore, the new voice may start anywhere from 1 to 31 samples before the next `updateLFO` is called — and thus before the synth had a chance to set the cutoff frequency of the filter to the correct value! 31 samples (at most) is not a lot and no one would notice this… unless it created a glitch. And glitches can easily happen when the filter is used with old state on new input data — especially if it also suddenly makes a jump from the old cutoff to the new one.

There are different ways to solve this conundrum: you could call `filter.updateCoefficients` from `startVoice` to make sure the coefficients are immediately set to the new value. Another solution is to wait until the next `updateLFO` to start the voice. Or, as we do it in JX11, reset the filter when the voice finishes playing the previous note, so that the coefficients are all zeros again. Now the first samples of the voice are simply not audible because the filter's cutoff is at 0 Hz, until the next `updateLFO` properly sets the filter coefficients. As you can tell, designing a synth — especially when you're trying to optimize certain parts of it — is full of small decisions and edge cases that can have audible consequences.

Velocity sensitivity

The note velocity is currently used to modulate the loudness of the sound. This makes a lot of sense: the harder you press down on a piano key, the louder a tone you get.

But with acoustic instruments something else happens too. The harder you strike the piano, the brighter the tone becomes. In a synth you can achieve the same by using the velocity to modulate the filter, since a higher cutoff frequency means a brighter sound.

In the last chapter you already wrote the code to set `synth.velocitySensitivity`, but you haven't used it for anything yet. This is a small value between –0.05 and +0.05. If the **Velocity** parameter was set to OFF, the `velocity` was fixed to 80 and `velocitySensitivity` became 0.

To modulate the filter cutoff using velocity, add a new line in **Synth.cpp** in `startVoice`, below where you set `voice.cutoff`:

```
voice.cutoff = sampleRate / (period * PI);
voice.cutoff *= std::exp(velocitySensitivity * float(velocity - 64));  // this is new
```

This multiplies the base cutoff frequency with some value to make it higher or lower. Here the multiplier is made from a combination of the velocity and the velocity sensitivity. As before, `exp` is used because frequencies are logarithmic. The reason you do this calculation here and not in `updateLFO`, is because `startVoice` has access to the velocity and it needs to be done just once.

Let's say `velocitySensitivity` is 0, either because the **Velocity** parameter is OFF or because the slider is perfectly centered at 0%. Then the multiplier is 1.0 and `velocity` has no effect on the cutoff frequency. (Recall that setting **Velocity** to 0% is different than OFF, which additionally affects the scaling of the amplitude envelope.)

If velocity is 64, the middle of the velocity range, then the multiplier is also 1.0. When you play quieter than average, the multiplier will be smaller than 1. And when you play louder than average, the multiplier will be larger than 1.

The smallest value this multiplier can be is approximately 1/24, the largest it can be is 24. This suspiciously looks like two octaves up or down — but these are not semitones. It's more like ±55 semitones because of the exp. So, by playing very softly or very loudly, you can shift the cutoff frequency up or down by four-and-a-half octaves.

Try it out! Set **Filter Freq** to 50% and **Velocity** to 100%. Press a key softly and you won't get many harmonics. Gradually press the key with more force, and you'll hear the sound increase in brightness. In the spectrum analyzer, the number of harmonics should increase the louder you play.

Here's a cool trick: the intensity value for a modulation source can be negative. Normally a low velocity means that not much should happen, the amplitude of the sound is small and the filter cutoff doesn't let many frequencies through. By setting the **Velocity** parameter to a negative value, the opposite happens: a low velocity will have a large impact on the filter cutoff, while a high velocity will hardly move the cutoff from its initial position.

Give that a try too! Set the **Velocity** to –80% and notice the difference between playing loudly and playing softly. This upside-down frequency response isn't how most real instruments behave, but it can create cool synth sounds!

Before you're done with this topic, there's a loose end to take care of. You currently fill in voice.cutoff inside startVoice, but there is also restartMonoVoice that is used for legato-style playing in **Mono** mode. This method needs to update voice.cutoff with the pitch of the new note and the velocity too.

Add the following lines to restartMonoVoice in **Synth.cpp**:

```
voice.cutoff = sampleRate / (period * PI);
if (velocity > 0) {
    voice.cutoff *= std::exp(velocitySensitivity * float(velocity - 64));
}
```

This uses the exact same formula as in startVoice. However, when restoring a queued note, you do not have the velocity of the old note anymore, so in that case it ignores the velocity sensitivity part.

Resonance / Q

The amount of resonance or Q is determined by the **Filter Reso** parameter. Unlike the cutoff frequency, Q is the same for all voices and it doesn't get modulated in this synth.

Add the following public member variable to **Synth.h**:

```
float filterQ;
```

As usual you'll fill this in from **PluginProcessor.cpp**'s `update` method:

```
float filterReso = filterResoParam->get() / 100.0f;
synth.filterQ = std::exp(3.0f * filterReso);
```

This creates an exponential curve that starts at `filterQ` = 1 and goes up to `filterQ` = 20.

In the same method, change the code that calculates `volumeTrim` to the following. Naturally, you must put this behind the above two lines or it won't find the `filterReso` variable.

```
synth.volumeTrim = 0.0008f * (3.2f - synth.oscMix - 25.0f * synth.noiseMix)
                           * (1.5f - 0.5f * filterReso);
```

Since resonance can make the sound a lot louder, especially when the cutoff frequency falls exactly on the fundamental or one of the harmonics, `volumeTrim` makes everything quieter as the resonance goes up. But keep in mind that `volumeTrim` only applies when a new note is played — if you change **Filter Reso** while a note is playing, the volume level isn't automatically compensated for.

Also add a new member variable to **Voice.h**:

```
float filterQ;
```

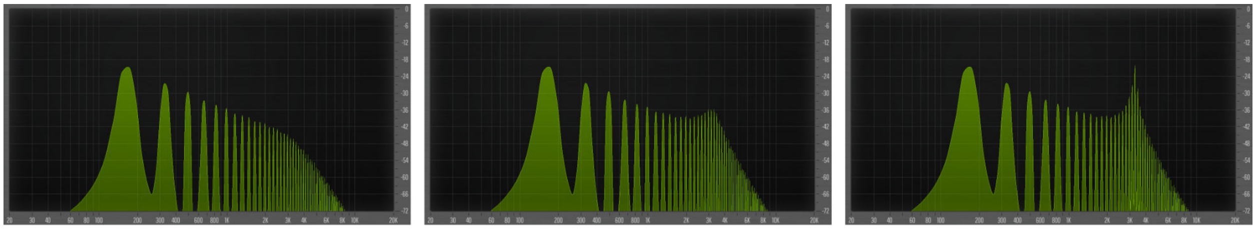
In `Voice`'s `updateLFO`, change the call to `updateCoefficients` into the following:

```
filter.updateCoefficients(modulatedCutoff, filterQ);
```

Finally, in **Synth.cpp**, in `render`, add the following line to the loop that sets the properties on the voice objects.

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        updatePeriod(voice);
        voice.glideRate = glideRate;
        voice.filterQ = filterQ;      // add this line
    }
}
```

Try it out! Use the **Filter Reso** setting to change the amount of resonant peaking at the cutoff frequency. For example, here is a note with **Filter Freq** set to 70%, and different values for **Filter Reso**:



Filter Reso 0% Filter Reso 40% Filter Reso 100%

Spectrum of a note with different Q settings

Q usually goes from 0.707 (the square root of 2) until infinity. If Q = 0.707, there is no resonant peak and the sound level at the cutoff frequency is precisely 3 dB lower than in the passband, which is why the cutoff frequency is also called the –3 dB point. For small values of Q, the resonant peak is fairly wide and rather flat. As Q becomes larger, the peak becomes taller but thinner.

In JX11, Q starts at 1.0, which means there's always a tiny resonant bump at the cutoff point, even if **Filter Reso** is dialed all the way back. We don't let Q get higher than 20. When Q is large and the cutoff point falls exactly on a harmonic, this harmonic gets a massive boost. Especially on low tones you may find that this creates rather loud, and perhaps unpleasant, sounds.

Note: There are different ways to avoid this kind of extreme resonance, for example by using a modified version of the filter that reduces the height of the peak relative to the passband level as Q increases, usually through some kind of feedback. Another technique is to apply some kind of soft limiter that will distort the signal when it gets too loud, but in a pleasing way. Interesting as they are, those topics are out of scope for this book.

If you're up for a fun experiment, change the formula for calculating `filterQ` in `update` to the following:

```
synth.filterQ = 1.0f / ((1.0f - filterReso + 1e-9) * (1.0f - filterReso + 1e-9));
```

Now `filterQ` can go up to almost infinity. At Q = infinity, or close to it, the filter will self-oscillate. It needs only a tiny impulse to be excited and then it produces an additional sine wave at the cutoff frequency. In practice, this means you can tune **Filter Freq** so that a single voice actually plays two different notes. But be careful, with Q extremely high you can easily create super loud sounds. (When you're done with this experiment, I suggest changing the `synth.filterQ` line back to what it was.)

It may be handy to control the amount of resonance with a knob or slider on your MIDI controller. Add the following private member variable to **Synth.h**:

```
float resonanceCtl;
```

In **Synth.cpp**, in `reset`, set this value to 1.0 initially so that it has no effect:

```
resonanceCtl = 1.0f;
```

Then in `Synth::controlChange`, add the following case:

```
// Resonance
case 0x47:
    resonanceCtl = 154.0f / float(154 - data2);
    break;
```

This reacts to MIDI CC `0x47`, which is commonly used for this feature. However, feel free to use your own MIDI CC number here. Recall from last chapter that you can easily find those with a `DBG` statement. In the next chapter you'll learn how to make this sort of thing more flexible with a MIDI Learn function.

The formula maps the position of the controller (0 – 127) to a linear curve between 1.0 and 5.7, which will be used to add an extra boost to the value from the **Filter Reso** parameter.

Finally, in `Synth`'s `render`, change the line that sets the voice's `filterQ` to the following:

```
voice.filterQ = filterQ * resonanceCtl;
```

Try it out! You can now easily adjust the amount of resonance while you're playing.

Filter cutoff LFO

In the last chapter you used the LFO to control the pitch of the oscillators to create a vibrato or PWM effect. JX11 only has one LFO but it can do double duty and also modulate the cutoff frequency of the filter.

Add the following public member variable to **Synth.h**:

```
float filterLFODepth;
```

This variable holds the LFO modulation intensity for the filter cutoff. In **PluginProcessor.cpp**'s `update` method, add the following lines:

```
float filterLFO = filterLFOParam->get() / 100.0f;
synth.filterLFODepth = 2.5f * filterLFO * filterLFO;
```

As with most parameters for this synth, the **Filter LFO** parameter goes from 0 to 100%. Because the percentage is squared, this is turned into a parabolic curve. The possible values go between 0 and 2.5.

In **Synth.cpp**, in `updateLFO`, change the line that calculates `filterMod` to the following:

```
float filterMod = filterKeyTracking + filterLFODepth * sine;
```

Previously this only used the `filterKeyTracking` variable but now it adds `filterLFODepth * sine`. When multiple modulation sources all affect the same target, the different modulation variables are added up. Recall that `sine` is the output value from the LFO and oscillates between +1 and –1.

That's all you need to do to use the LFO for modulating the cutoff. Try it out! Set **Filter Freq** to 50%, **LFO Rate** to 2 Hz, and slowly drag **Filter LFO** from left to right as you play a note. The filter will now create a kind of wah-wah effect. Also experiment with changing the **LFO Rate**, **Filter Freq**, and **Filter Reso** parameters while the note plays. All of a sudden the kinds of sounds this synth can create have gotten a lot more possibilities!

Hold up, you're not done yet… If your MIDI controller supports aftertouch, this can be used to give additional depth to the LFO. For vibrato and PWM this is done with the mod wheel, but for the filter this can be done with aftertouch. These are not hard-and-fast rules for synth design, it's simply the choice that JX11 goes with. If you want to use aftertouch for something else in your own synth, by all means.

Aftertouch is what happens when, after you press a key to play a note, you press it down even more. JX11 only supports channel aftertouch, otherwise known as channel pressure. There's at most one key sending aftertouch data — the key with the highest pressure applied — and this pressure affects all notes currently being played. The MIDI standard also defines polyphonic aftertouch, where each key transmits its own independent aftertouch value.

In **Synth.h** add a private member variable:

```
float pressure;
```

In **Synth.cpp**, in `reset`, set this new variable to zero so that it doesn't have any effect if the synth never receives any aftertouch events:

```
pressure = 0.0f;
```

The MIDI message for channel pressure is `Dx` where `x` is the channel. Unlike most other MIDI messages it only has one data byte. In the `midiMessage` method, add the following case statement:

```
// Channel aftertouch
case 0xD0:
    pressure = 0.0001f * float(data1 * data1);
    break;
```

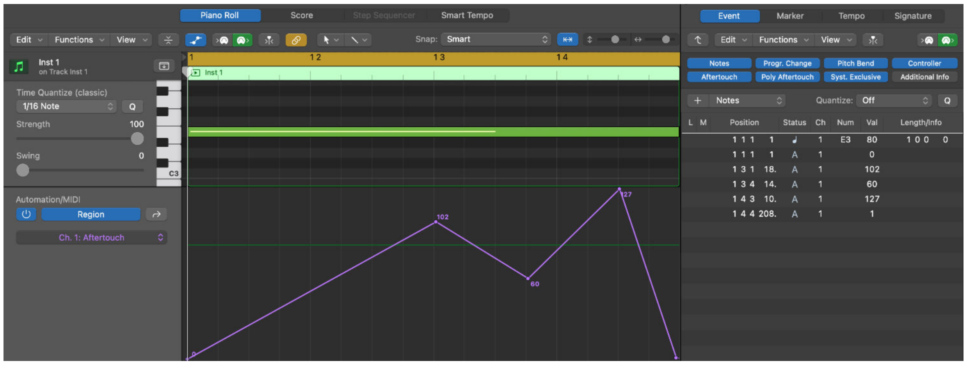
Make sure you're adding this to `midiMessage`, not `controlChange`, as this is not a MIDI CC message. The formula maps the pressure value to a parabolic curve starting at 0.0 (position 0) up to 1.61 (position 127).

In `updateLFO`, change the calculation of `filterMod` to the following:

```
float filterMod = filterKeyTracking + (filterLFODepth + pressure) * sine;
```

The amount of aftertouch is added to the intensity from the **Filter LFO** parameter. One way to use this is to set **Filter LFO** to zero or a small value, and then apply aftertouch to increase the intensity of the LFO while you're playing.

Try it out! If your MIDI controller doesn't have aftertouch (which is quite likely) you can test this feature in your DAW by drawing automation for the Channel Pressure MIDI events. It may also be called Channel Aftertouch, Mono Pressure, Mono Aftertouch, or just Aftertouch.



Time Quantize (classic): 1/16 Note
Strength 100
Swing 0
Snap: Smart
Notes
Quantize: Off

L	M	Position	Status	Ch	Num	Val	Length/Info
		1 1 1 1	♩	1	E3	80	1 0 0 0
		1 1 1 1	A	1		0	
		1 3 1 18.	A	1		102	
		1 3 4 14.	A	1		60	
		1 4 3 10.	A	1		127	
		1 4 4 208.	A	1		1	

Aftertouch automation in Logic Pro

MIDI CC and pitch bend

Another useful feature is to allow the user to change the amount of filter cutoff modulation using knobs or sliders on their MIDI controller. To allow this, you just need to intercept some MIDI CC messages. You've already seen how to do this kind of thing before, so this should be a piece of cake.

In **Synth.h**, add a new private member variable:

```
float filterCtl;
```

And in **Synth.cpp**, in `reset`, set this variable to zero:

```
filterCtl = 0.0f;
```

Then in `controlChange`, add the following case statements:

```
// Filter +
case 0x4A:
    filterCtl = 0.02f * float(data2);
    break;

// Filter -
case 0x4B:
    filterCtl = -0.03f * float(data2);
    break;
```

This uses not one but two controls: one for changing `filterCtl` in the positive direction, and one for changing it in the negative direction. The MIDI CC code `0x4A` is typically used for the filter cutoff value; `0x4B` is a generic controller code. As before, feel free to change these numbers if your MIDI controller doesn't send these particular codes.

Finally, in `updateLFO` change the calculation of `filterMod` to include the new variable:

```
float filterMod = filterKeyTracking + filterCtl + (filterLFODepth + pressure) * sine;
```

The `filterCtl` variable serves the same purpose as `filterKeyTracking`, so you can use these knobs to adjust the initial cutoff point set by **Filter Freq** while playing. Try it out! Set **Filter Freq** to 50%, play a note, and twiddle some knobs! One knob makes the cutoff higher, the other makes the cutoff lower.

Another small tweak is that the pitch bend wheel changes the pitch of the note by ±2 semitones. Since the base cutoff frequency of the filter depends on the note's pitch, you should include the amount of pitch bend in the filter modulation too. Perfection is about getting the small details right.

The pitch bend amount is something that applies to all playing voices, which is why it's handled by `Synth`. However, for our purposes it's better to apply the pitch bend modulation in `Voice`, and so you need to tell the voices about the current pitch bend amount.

Add the following member variable to **Voice.h**:

```
float pitchBend;
```

Then in **Synth.cpp**, in the `render` method, copy the `pitchBend` value into the active voice objects:

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        updatePeriod(voice);
        voice.glideRate = glideRate;
        voice.filterQ = filterQ * resonanceCtl;
        voice.pitchBend = pitchBend;              // add this line
    }
}
```

Finally, in **Voice.h**, in `updateLFO`, apply the amount of pitch bend as follows:

```
float modulatedCutoff = cutoff * std::exp(filterMod) / pitchBend;
```

The reason this happens in `Voice` but the other filter modulations take place in `Synth`, is that `pitchBend` is already a multiplier based on the number of semitones, and so it should not be part of `filterMod` as then it would get exponentiated twice.

Try it out! Set **Filter Freq** to 75%, play a note and roll the pitch bend wheel. The cutoff frequency should move back and forth with the pitch. To make this more audible, set **Filter Reso** to 100% and in the spectrum analyzer look at the resonant peak moving along with the pitch wheel.

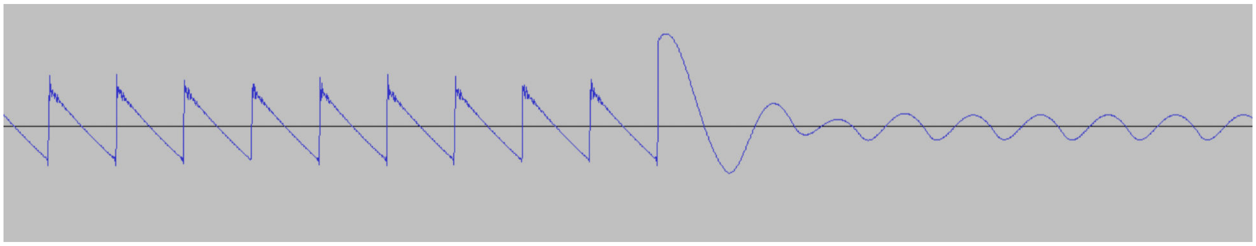
> **Note:** The `pitchBend` multiplier was derived from the number of semitones using the formula $2^{N/12}$. But for `filterMod` you apply `std::exp`. Both roughly come down to the same thing because 2^x and $\exp(x)$ are both exponential curves, and by choosing the exponent carefully you can make both curves the same. That said, it might be nicer to express all modulation amounts in semitones, as they have more musical meaning than seemingly arbitrary numbers stuck into `std::exp`. So rather than specifying **Filter Freq** as a percentage, it could be something like ±48 semitones. Perhaps something to consider for your own synths.

Smoothing the filter updates

There are quite a few things that affect the filter cutoff modulation: the note's pitch and velocity, the **Filter Freq** parameter, the LFO, aftertouch, MIDI CC, and the pitch wheel.

The combination of all these means the filter cutoff can jump around a fair bit from one `updateLFO` call to the next. Drag the **Filter Freq** slider up and down for a while and hear for yourself that it sometimes gives popping or crackling sounds. This is another example of zipper noise, where the jumps between different states are too big.

The following recording shows a typical example:



When the cutoff point makes a big jump, the sound may glitch

There clearly is a discontinuity in the sound where the cutoff changed. To reproduce this, alternate between clicking on the 30% and 90% positions of the **Filter Freq** slider while a note is playing. This occasionally gives a nasty glitch — not great.

Previously, you used a linear smoother to remove zipper noise on parameter changes. You can do the same thing on the filter modulation amount. However, frequencies are logarithmic, not linear, and so an exponential smoothing curve would sound more natural. Fortunately, you already know an excellent method for transitioning between two values in an exponential manner: the one-pole filter.

In this section you'll use a one-pole filter to smoothen the modulation amount for the cutoff frequency. In a way you're filtering the filter!

In **Synth.h**, add a new private member variable to the class:

```
float filterZip;
```

In **Synth.cpp**, in `reset`, set this value to zero:

```
filterZip = 0.0f;
```

Finally, in `Synth`'s `updateLFO` make the following changes:

```
float filterMod = filterKeyTracking + filterCtl + (filterLFODepth + pressure) * sine;

filterZip += 0.005f * (filterMod - filterZip);    // add this line

for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        voice.osc1.modulation = vibratoMod;
        voice.osc2.modulation = pwm;
        voice.filterMod = filterZip;         // change this line
        voice.updateLFO();
        updatePeriod(voice);
    }
}
```

The new `filterZip` variable holds the smoothed version of `filterMod`. On every LFO update step, this moves `filterZip` a little closer to the value of `filterMod`. The reason for dezippering the filter modulation in `Synth` and not `Voice`, is that all voices share this value, so you only have to do it once.

Try it out! Dragging the **Filter Freq** slider now sounds a lot more polished. As a bonus, this smoothing applies to the other filter modulations as well.

To keep things simple, the filter coefficient is hardcoded to 0.005, but that does mean the speed of the smoothing depends on the sample rate: the higher the sample rate, the quicker the one-pole filter reaches its target. As an exercise for the reader, use the formulas from the envelope or glide parameters to calculate a filter coefficient that is independent of the sample rate.

> **Note:** The one-pole filter is very handy to have in your toolbox of DSP building blocks. You've used it for the envelope, for glide, and now for evening out changes to the filter modulation. The one-pole filter can also be used for smoothing plug-in parameters.
>
> You used a linear smoother for the Output Level parameter. That begs the question: which kind of smoother should you choose? The one-pole filter makes an exponential curve, which may not be appropriate for the parameter in question. Another potential downside is that the one-pole filter never lands on its target exactly, it merely approximates it. A linear smoother is guaranteed to hit its target and only runs for the time you set.
>
> That said, smoothing is supposed to happen fast enough for the user not to notice. With a smoothing time of just a few milliseconds, the difference between the two methods probably doesn't matter too much. As with anything, experiment, and see which one you like best!

Smoothing is a kind of interpolation, where the synth doesn't immediately transition from the old value to the new one, but gradually in small steps. Audio programming involves a lot of this kind of interpolation.

Not all of JX11's parameters have smoothing applied to them. A few other candidates for smoothing are the **Filter Reso** and **LFO Rate** parameters. Big jumps in the latter can create unwanted discontinuities in the LFO waveform. Recall that it only makes sense to smoothen parameters with continuous values that affect the voices that are currently playing.

For this book, I sustained some notes and vigorously dragged the sliders for the various parameters back and forth. If this didn't result in audible zipper noise, I decided not to dezipper or smoothen the parameter. However, it's a good exercise to try and smoothen some of these parameters anyway. Use either `juce::LinearSmoothedValue` or a one-pole filter, or perhaps even build your own `ParameterSmoother` class.

Do you need to smoothen the mod wheel and pitch bend wheel? When you manipulate these wheels by hand they change gradually, but there is always a minimum step size — a decent 14-bit precision for the pitch bend, a measly 7 bits for the mod wheel.

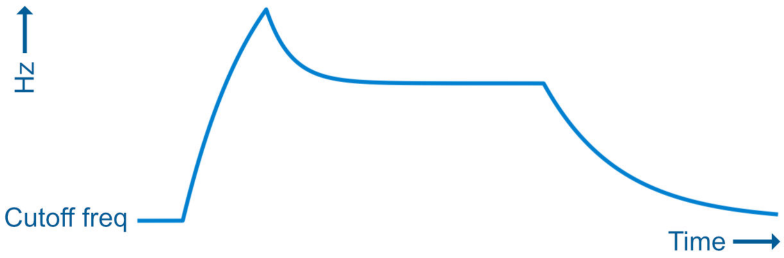
There’s also no telling how often the MIDI controller actually sends MIDI events, and so the step size may still be large, even for the pitch wheel. Plus, the user can automate these controls in their DAW, in which case they can make as big jumps as they like. So yes, you should smoothen these if you find they create zipper noise.

I don’t hear anything particularly egregious for this synth but feel free to use JUCE’s linear smoother or a one-pole filter to smoothen out the wheels, especially the mod wheel since it has relatively low precision.

Keep in mind that smoothing might still make glitches if automation causes the parameter to make *very* big jumps, since the smoothing time is so short. If the distance for the parameter change is large, the smoothing steps will be too. On the other hand, you can’t make the smoothing time too long or the smoothing itself becomes audible as a glide between two parameter values. Getting the smoothing to feel right is a matter of trial and error.

Filter envelope

The final modulation you’ll apply to the cutoff frequency of the filter is an envelope. This envelope will have the exact same shape as the one used for the amplitude, except now it changes the filter’s cutoff:

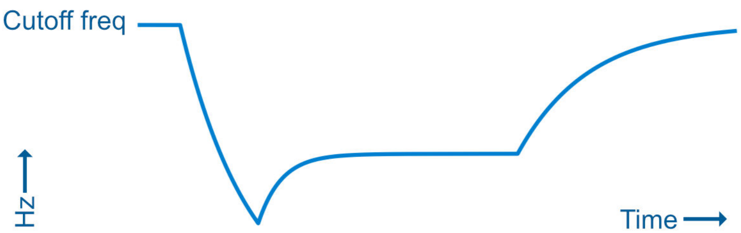


The ADSR envelope applied to the cutoff frequency

In the attack phase, the frequency of the cutoff point increases. In the decay stage, the frequency decreases again. In the sustain stage, the cutoff frequency is not modulated by the envelope and stays in the same position. When the key is released, the release stage will decrease the cutoff frequency back to its starting point.

For the amplitude envelope, the intensity at which the envelope is applied is always 100%, although the amplitude does get scaled by the note velocity. For the filter envelope, the user can set the intensity manually. The larger the intensity, the more the envelope will shift the cutoff frequency away from its starting point.

It’s possible to set the intensity to a negative number. Now the envelope works the other way around, with attack lowering the cutoff frequency, and decay and release raising it again. This makes the sound duller at first (fewer high frequencies) and brighter near the end. Weird but interesting!



The filter envelope with a negative intensity

Since this envelope works in exactly the same way as the amplitude envelope, you’ll reuse the `Envelope` class for this.

First, add some new public member variables in **Synth.h**:

```
float filterAttack, filterDecay, filterSustain, filterRelease;
float filterEnvDepth;
```

The first four are the multipliers that will be used by the `Envelope` class. The `filterEnvDepth` variable holds the modulation intensity for the envelope.

Add the following lines to **PluginProcessor.cpp**, in `update`:

```
synth.filterAttack = std::exp(-inverseUpdateRate *
                                std::exp(5.5f - 0.075f * filterAttackParam->get()));

synth.filterDecay = std::exp(-inverseUpdateRate *
                                std::exp(5.5f - 0.075f * filterDecayParam->get()));

float filterSustain = filterSustainParam->get() / 100.0f;
synth.filterSustain = filterSustain * filterSustain;

synth.filterRelease = std::exp(-inverseUpdateRate *
                                std::exp(5.5f - 0.075f * filterReleaseParam->get()));
```

These are the same equations you used for the amplitude envelope, except that sustain is squared. I’m not 100% sure why this was done, perhaps to account for the logarithmic nature of frequencies or perhaps to make it easier to set small sustain values, like a parameter skew.

Another important difference is the use of inverseUpdateRate rather than inverseSampleRate, because the modulations are updated only once every 32 samples, and therefore this envelope can run at the slower rate too.

Also add the following line to update:

```
synth.filterEnvDepth = 0.06f * filterEnvParam->get();
```

The **Filter Env** parameter sets the intensity of the envelope modulation. This parameter goes from –100% to 100%. If it's negative, the envelope gets flipped upside down as illustrated above. Here, you set synth.filterEnvDepth to a value between –6.0 and +6.0.

Next, you need to add the filter envelope to the voices. In **Voice.h**, add the following member variables:

```
Envelope filterEnv;
float filterEnvDepth;
```

In Voice's reset method, add this line:

```
filterEnv.reset();
```

Change Voice's updateLFO to add the filter envelope to the modulation calculations:

```
void updateLFO()
{
    period += glideRate * (target - period);

    // add this line:
    float fenv = filterEnv.nextValue();

    // change this line:
    float modulatedCutoff = cutoff * std::exp(filterMod + filterEnvDepth * fenv) / pitchBend;

    modulatedCutoff = std::clamp(modulatedCutoff, 30.0f, 20000.0f);
    filter.updateCoefficients(modulatedCutoff, filterQ);
}
```

First this gets the filter envelope's current level, a value between 0 and 1, then multiplies it by the intensity, and finally adds it to the amount of cutoff modulation. If the **Filter Env** parameter is set to 0%, the envelope has no effect on the cutoff frequency but the other modulations still work.

It may seem strange that the envelope modulation happens inside the `std::exp`, since the envelope itself is already exponential. Agreed, but this is how the original MDA synth did it. Feel free to experiment with this formula and see what the effect is when you put the envelope modulation outside the exponential.

The final change for `Voice` is in the `release` method, which is called after a Note Off event is received. Make this put the filter envelope in the release stage too:

```
void release()
{
    env.release();
    filterEnv.release();    // add this line
}
```

There are still a few things to implement before this works. In **Synth.cpp**, in `render`, you need to pass the `filterEnvDepth` value to the voice:

```
for (int v = 0; v < MAX_VOICES; ++v) {
    Voice& voice = voices[v];
    if (voice.env.isActive()) {
        updatePeriod(voice);
        voice.glideRate = glideRate;
        voice.filterQ = filterQ * resonanceCtl;
        voice.pitchBend = pitchBend;
        voice.filterEnvDepth = filterEnvDepth;    // add this line
    }
}
```

Note: This loop is copying quite a few things into `Voice` now at the start of every audio block. This is simple and easy but not necessarily the most efficient approach. Another way to do this, is to give `Voice` a pointer back to `Synth` so it can access all of `Synth`'s variables. You can also put all the values that need to be shared between `Synth` and `Voice` into a struct, and give the `Voice` objects a pointer to this struct upon construction. Whatever approach you take, don't use global variables for this purpose!

You must also set up the filter envelope when the voice starts playing. Add the following lines to `startVoice`. This is identical to what you did for the amplitude envelope, except it uses the filter attack, decay, sustain, and release values.

```
Envelope& filterEnv = voice.filterEnv;
filterEnv.attackMultiplier = filterAttack;
filterEnv.decayMultiplier = filterDecay;
filterEnv.sustainLevel = filterSustain;
filterEnv.releaseMultiplier = filterRelease;
filterEnv.attack();
```

Note that in monophonic mode, the filter envelope is not retriggered in `restartMonoVoice`, just like the amplitude envelope.

That's it! Try it out. For a great example of what the filter envelope can do, check out the "5th Sweep Pad" preset. If you're not sure what to listen for, set **Filter Env** to 0% to turn off the envelope and then put it back to –76% and play the note again. You can definitely hear the difference! The filter sweep is clearly visible in the spectrum analyzer.

Also try the envelope in combination with different velocity settings, including a negative velocity sensitivity. The filter envelope is lots of fun to play with!

Conclusion

Phew! That was a lot of stuff. Here's a quick recap of everything that's used to modulate the filter cutoff:

- the note's pitch
- the note velocity
- the Filter Freq parameter
- the LFO
- aftertouch
- MIDI CC
- the pitch wheel
- filter envelope shape and intensity

The way these modulators interact can be quite complex but it also allows for all kinds of exciting sounds. You should now be able to hear all the built-in presets in their full glory. Try them out if you haven't already.

This concludes the audio processing part of the synth. It still lacks a decent user interface, but otherwise JX11 is complete. You've built an entire virtual analog synthesizer, and hopefully learned a thing or two along the way!

Bonus section: Moog ladder filter

Filters are a lot of fun and I don't want to end this chapter just yet. The Cytomic SVF filter works well but it's not the most musical filter available. It sounds like a biquad — exactly the same, in fact — and while biquads and SVFs give very clean results, musicians don't always want their filters to be too clean. True musical audio filters also color, or distort, the sound in ways we find pleasing.

I encourage you to rip out the filter and replace with something else, to experiment with the effect on the sound that different filters have. JUCE's DSP module comes with a few built-in filters. This bonus section gives a quick overview of how to put one of these JUCE filters into JX11.

Go to **Projucer** and activate the **Modules** section in the sidebar. Then click the round + button. From the popup choose **Add a module** > **Global JUCE modules path** > **juce_dsp**. Then export the project again to your IDE.

This adds a new JUCE module to your project that isn't there by default. The `juce_dsp` module contains a variety of DSP building blocks[50] such as a delay line, convolution, a limiter and compressor, oversampling, waveshaping, and many other functions — and filters, naturally.

You will be using the `LadderFilter` class, which is based on the classic Moog ladder filter. The easiest way to use `LadderFilter` is to wrap it in the existing `Filter` class, so that the rest of the code doesn't have to change a lot.

The JUCE DSP building blocks are intended to be applied to an entire audio block at once, described by a `ProcessContext` object. However, our synth renders the voices sample-by-sample. The `LadderFilter` object can do this too but unfortunately the required `processSample` method has the access level `protected`. To work around this issue, you'll make `Filter` extend from `LadderFilter`.

Replace the contents of **Filter.h** with the following code (you may want to backup the previous version first).

[50] https://docs.juce.com/master/namespacedsp.html

```
#pragma once

#include <JuceHeader.h>

class Filter : public juce::dsp::LadderFilter<float>
{
public:
    void updateCoefficients(float cutoff, float Q)
    {
        setCutoffFrequencyHz(cutoff);
        setResonance(std::clamp(Q / 30.0f, 0.0f, 1.0f));
    }

    float render(float x)
    {
        updateSmoothers();
        return processSample(x, 0);
    }
};
```

Filter now extends juce::dsp::LadderFilter. This is a templated class, so you need to specify the datatype, float. For a plug-in that does its audio processing in double precision, you'd use double here.

In the updateCoefficients method, you call LadderFilter's own setCutoffFrequencyHz and setResonance methods. The resonance in LadderFilter goes from 0 to 1, and dividing Q by 30 sounded reasonable to me. Any values outside this range will trigger an assert.

The render function calls the processSample method. I guess the reason this method is protected in LadderFilter, is that it doesn't work without calling updateSmoothers first, which you also do here. There is no need to define a reset method, as LadderFilter already has one of its own.

Before you can use any of the JUCE DSP building blocks, you must call prepare on them. Let's do this from Synth's allocateResources method in **Synth.cpp**:

```
void Synth::allocateResources(double sampleRate_, int samplesPerBlock)
{
    sampleRate = static_cast<float>(sampleRate_);

    // add these lines
    juce::dsp::ProcessSpec spec;
    spec.sampleRate = sampleRate;
    spec.maximumBlockSize = samplesPerBlock;
    spec.numChannels = 1;
    for (int v = 0; v < MAX_VOICES; ++v) {
        // replace the code with these lines
        voices[v].filter.setMode(juce::dsp::LadderFilterMode::LPF12);
        voices[v].filter.prepare(spec);
    }
}
```

This tells the `LadderFilter` what sample rate to expect and how many channels to use. JX11 always uses one channel because the voice renders the sound in mono, not stereo. Don't forget to uncomment the `samplesPerBlock` argument.

Try it out! The synth will sound quite a bit different because the Moog ladder filter has a distinctively different frequency response than a standard bilinear Z-transform-based filter such as the SVF. Another nice feature of this filter is that the gain drops as Q increases, which helps to avoid the sound from becoming extremely loud when the cutoff happens to fall on one of the note's harmonics — something you may have noticed happening with the SVF filter.

Experiment with the different filter modes: Right now, the filter is configured in the `LPF12` mode, which is a low-pass filter with 12 dB/octave roll-off. That's similar to the SVF. However, you can also choose `LPF24`, which is twice as steep with 24 dB/octave roll-off. Or even HPF for a high-pass filter and BPF for a band-pass filter.

The `LadderFilter` has a `setDrive` method. Give this a value of 1.0 or higher to add saturation to the sound. Good stuff! You may even want to add a new parameter for this.

Many other filter designs exist, and part of synthesizer design is picking great filters. The SVF we discussed in this chapter is an excellent all-round filter for performing basic filtering tasks, but if you really want your synth to stand out, it's worth diving deeper into other filter designs — there are plenty!

Note: The `juce_dsp` module has a `StateVariableTPTFilter` class. This is also an SVF and it will give the same frequency response as the Cytomic SVF filter. However, its internal implementation is quite different. The Cytomic SVF was made by doing nodal analysis, where equations were used to simulate the charging of capacitors to predict the different voltages in the circuit. JUCE's SVF was designed in a more mathematical way using something called the Topology Preserving Transform. In the end, both approaches give a filter that sounds exactly the same, and both implementations work equally well.

Chapter 13: User interface

So far we've mostly ignored the user interface of the synth and relied on the auto-generated UI, which is fully functional but won't win any prizes for looks or usability. The synth certainly sounds good but it could do with a lick of paint!

In the early stages of development or prototyping, using the automatic UI is more than sufficient, especially if you're still adding and removing parameters while trying to figure out exactly how to design and build the synth.

But make no mistake: the UI is just as important to the synth as the sounds it can produce. I don't think it's a controversial opinion to state that many existing synths are unnecessarily difficult to use. In the case of hardware synths, some awkwardness in the design of the control panel cannot always be avoided due to physical limitations, but in software there is no such excuse!

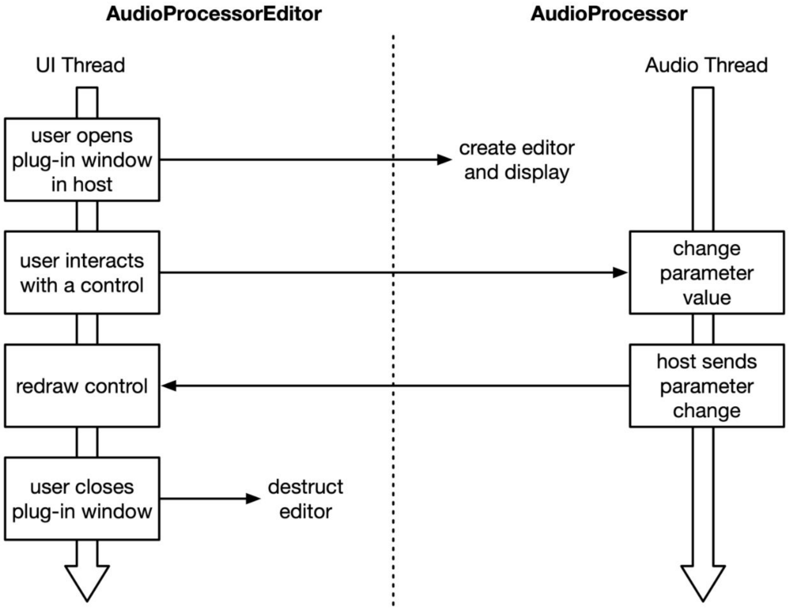
Designing an intuitive user interface is key to letting people get the most of your synth. Besides, the eye wants to look at something pretty, and plug-ins with shiny UIs are easier to sell.

This chapter will briefly cover how to get a basic UI up and running with JUCE. To fully do justice to the topic of synthesizer user interface design would require a book all of its own (maybe a sequel to this one?). Making JX11 look nice and easy to use is left as an exercise for the reader, but this chapter should get you well underway.

The editor and the processor

The `AudioProcessor` is the plug-in's main object. It exists for as long as the plug-in is being used by the host application. In a typical DAW, the `AudioProcessor` is instantiated when the user puts the plug-in on a track, and it's destroyed when the plug-in gets removed from the track again. If the plug-in is placed on more than one track, each of these gets its own `AudioProcessor` instance that is independent from the others.

When the user opens the UI of the plug-in, the audio processor object creates an instance of the `AudioProcessorEditor` class and the host displays this editor object on the screen, usually in its own window. Exactly how this happens depends on the host, and the plug-in doesn't need to worry about it.


AudioProcessorEditor
AudioProcessor
UI Thread
Audio Thread
user opens plug-in window in host
create editor and display
user interacts with a control
change parameter value
redraw control
host sends parameter change
user closes plug-in window
destruct editor

The division of labor between the UI thread and audio thread

The user interacts with the UI through the `AudioProcessorEditor` object: clicking on buttons, rotating the knobs and dials, typing in values, and so on. These events are handled by the host's UI thread. Typically this is the application's main thread, or in JUCE parlance, the message thread. Just like all audio processing should be done exclusively on the audio thread, any UI-related activities must only happen on the UI thread.

In general, each of the controls in the UI corresponds to a plug-in parameter. JX11 has a parameter for the filter resonance, for example, and so you'd also expect there to be a knob for this in the UI. Not all parameters necessarily appear in the UI, and not everything in the UI needs to be a parameter. However, most of the UI controls will correspond 1:1 to the plug-in's parameters.

Note: If you're not sure what to make a plug-in parameter and what not, the rule-of-thumb is that anything you want to expose to the host so that it can be automated should be a parameter. Anything else can remain private state inside the plug-in that the host doesn't need to know about.

Suppose you've decided to put the synth's UI controls into two different panels that the user can switch between. To keep track of which panel is visible, you wouldn't use a plug-in parameter. The visible panel merely represents the state of the UI, not the sounds that come out of the synth. However, you do want to save this state inside the XML when the plug-in is serialized, so that the next time the user loads their project the UI can be restored to exactly where they left off.

Whenever the user turns a knob or moves a slider in the plug-in's editor, the UI must tell the audio processor that the value for the corresponding parameter has changed. And whenever the host plays back automation, the audio processor must tell the UI that the parameter value has changed, so that the editor can redraw the knobs accordingly.

Clearly there needs to be some way for the audio processor and the editor to communicate, and this method has to be thread-safe since the parameters are used both by the UI thread and by the realtime audio thread. What makes this more complicated than regular concurrent programming is that the audio thread should never be blocked.

There are other situations too where the audio processor and the editor need to talk to each other. For example, you might want to show visualizations of the audio waveform or frequency spectrum, similar to what the oscilloscope and frequency analyzer plug-ins do. To enable this, the audio processor would periodically send blocks of audio data to the editor — from the audio thread to the UI thread — so this data can be displayed in the editor. Again, this needs to happen in a thread-safe manner without blocking.

Finally, when the user closes the plug-in window, the host deletes the `AudioProcessorEditor` object (although different hosts may do this at different times). The plug-in itself will stay alive until the user stops using it, typically by removing it from the track in the DAW. If the editor window was still open, it will also be deleted then.

The important thing to take away from this, is that the `AudioProcessor` class will always exist for as long as the plug-in is in use, but you have no guarantee that there is an `AudioProcessorEditor` instance. Your code can never depend on the editor existing!

The editor can safely assume the audio processor is present, but this is not true the other way around: the audio processor must be able to operate fully even if there is no active `AudioProcessorEditor` instance. It's a common error to put important variables that the audio processor needs inside the editor class — a big no-no.

This was a quick overview of the lifecycle of the plug-in and the editor, and how and when the processor and editor need to communicate. In the rest of this chapter you'll see a number of examples of how to do this in practice.

The PluginEditor source files

The user interface for the synth is managed by the class `JX11AudioProcessorEditor`, which lives in the **PluginEditor.h** and **PluginEditor.cpp** source files. Those files were created by Projucer when you made the project, but so far you haven't used them in JX11. Let's do something about that!

First, we'll examine the header file, which looks like the following:

```
#include <JuceHeader.h>
#include "PluginProcessor.h"

class JX11AudioProcessorEditor  : public juce::AudioProcessorEditor
{
public:
    JX11AudioProcessorEditor (JX11AudioProcessor&);
    ~JX11AudioProcessorEditor() override;

    void paint (juce::Graphics&) override;
    void resized() override;

private:
    JX11AudioProcessor& audioProcessor;

    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (JX11AudioProcessorEditor)
};
```

The base class for the editor is `juce::AudioProcessorEditor`. As always, the JUCE documentation[51] is a good reference to keep open. In the documentation you'll see that `juce::AudioProcessorEditor` in turn is derived from the `juce::Component` class. All the elements in your user interface are such `Component` objects. Once you understand how to use these `Components`, the JUCE UI framework will hold no more secrets for you.

The header file shows that the editor has `paint` and `resized` methods. The `paint` method is for drawing the contents of the UI, `resized` is for doing the layout of those contents. Usually you'll add several `juce::Component` objects to the editor — or at least objects that are subclasses of `Component` such as `juce::Slider` and `juce::TextButton` — and `resized` will position them inside the editor window.

The editor also has a reference back to the `JX11AudioProcessor` object. And like all JUCE classes, it uses the `JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR` macro to disable the copy constructor and register the instance with JUCE's memory leak detector.

[51] https://docs.juce.com/master/classAudioProcessorEditor.html

Let's see what happens inside **PluginEditor.cpp**. Projucer filled in this file with the following code. I've removed some of the comments to save space.

```
#include "PluginProcessor.h"
#include "PluginEditor.h"

JX11AudioProcessorEditor::JX11AudioProcessorEditor (JX11AudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    setSize (400, 300);
}

JX11AudioProcessorEditor::~JX11AudioProcessorEditor()
{
}

void JX11AudioProcessorEditor::paint (juce::Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour (juce::ResizableWindow::backgroundColourId));

    g.setColour (juce::Colours::white);
    g.setFont (15.0f);
    g.drawFittedText ("Hello World!", getLocalBounds(), juce::Justification::centred, 1);
}

void JX11AudioProcessorEditor::resized()
{
}
```

The job of the constructor is to add any `Component` objects the UI needs, and to set the size of the editor window, here 400×300 pixels. The destructor is empty by default but you'd use this to clean up when the editor window closes.

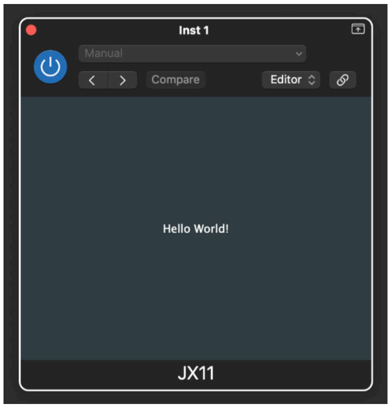
Inside `paint`, it first fills the background of the editor with a solid color and then draws the text "Hello World!" in the center. `resized` is currently empty because there are no components yet, but this is where you'd move them into their correct positions based on the current size of the editor window.

Great, now that you've got a sense of what the editor class looks like, let's enable it in the plug-in. In **PluginProcessor.cpp**, change the method `createEditor` to the following:

```
juce::AudioProcessorEditor* JX11AudioProcessor::createEditor()
{
    return new JX11AudioProcessorEditor(*this);
}
```

Instead of creating the `GenericAudioProcessorEditor` like you did before, it instantiates `JX11AudioProcessorEditor` and passes it a reference to the `JX11AudioProcessor`, so that the two can communicate. You don't need to worry about the memory management for this editor object. JUCE will automatically delete the instance when the user closes the editor window.

Try it out, opening the UI of the synth now shows the default "Hello World!" message. The `JX11AudioProcessorEditor` is the top-level component of the UI. It's usually placed into its own window inside the host program. Here is what the UI looks like in Logic Pro:



Hello World!

The plug-in editor as shown in Logic Pro

The round blue button and the controls at the top are window chrome that Logic Pro adds. The window will look somewhat differently in every DAW, but the main content always comes from the plug-in editor.

> **Note:** Even though the UI no longer shows the plug-in parameters, remember that you can still access them in AudioPluginHost by right-clicking on the JX11 block and choosing **Show all parameters**. Most DAWs also have an option to show the plug-in parameters without any adornment. Very useful for debugging the synth while you're building its user interface.

Your first rotary knob

There are many ways to design plug-in UIs but you can't go wrong with rotary knobs. In this section you'll go through all the steps it takes to add this kind of control to the editor. JX11 has 26 parameters but you'll just create knobs for a few of them in this book — once you've seen how to do one, you can do any number of them.

In JUCE the rotary knob is implemented by the `juce::Slider` class. It's a little odd that they called this a slider, but the same class can also make true horizontal and vertical slider controls in addition to the round ones we're after. A quick glance at the documentation[52] shows that this is a large class with lot of options and features. Like all UI controls, it derives from `juce::Component`.

In **PluginEditor.h** add the following declaration to the private members. This will be a knob for controlling the synth's **Output Level** parameter.

```
juce::Slider outputLevelKnob;
```

By making the slider object a regular member variable of the editor, it is instantiated when the editor window is opened. You don't have to worry about destructing the slider, it automatically goes away when the editor window is closed.

Go to **PluginEditor.cpp** and change the constructor to the following:

```
JX11AudioProcessorEditor::JX11AudioProcessorEditor (JX11AudioProcessor& p)
    : AudioProcessorEditor (&p), audioProcessor (p)
{
    outputLevelKnob.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    outputLevelKnob.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 100, 20);
    addAndMakeVisible(outputLevelKnob);

    setSize(600, 400);
}
```

The constructor is where you set up and configure the slider object. `setSliderStyle` makes it a rotary slider and `setTextBoxStyle` adds an edit box below the rotary part.

After configuring the slider, you call `addAndMakeVisible` to put it into the editor window. Finally, don't forget `setSize` to give the editor window its initial size. I made it 600×400 to give us some space to work with.

The slider itself doesn't have a size yet. You could set this size inside the constructor but it's better to implement the editor's `resized` method to do this kind of layout logic.

Change `resized` to the following:

```
void JX11AudioProcessorEditor::resized()
{
    outputLevelKnob.setBounds(20, 20, 100, 120);
}
```

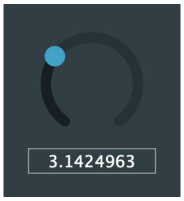
[52] https://docs.juce.com/master/classSlider.html

This makes the size of the rotary knob 100×100 points, with an additional 20 points at the bottom for the text box. The slider component is placed 20 points away from the left and top edges of the editor.

Change `paint` to remove the “Hello World!” message. It merely fills in the background now:

```
void JX11AudioProcessorEditor::paint(juce::Graphics& g)
{
    g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));
}
```

Try it out, run the plug-in and you’ll get a UI that looks like this:



The slider / rotary knob

This uses the standard JUCE look, which in my opinion is rather ugly, and you’ll change this to something prettier later in the chapter. The knob works like a typical rotary control that you find in many synth designs: click and drag up/down or left/right to change the value, or use the mouse wheel. If you hold down Ctrl while dragging, it goes in smaller increments.

After clicking on the slider to give it keyboard focus, you can also use the arrow keys to set its value. Normally you can Alt-click or double-click on a JUCE slider to reset it to its default position, but this slider isn’t hooked up to a parameter yet so it doesn’t know what the default position is. There is a text box below the slider showing the current value, which goes from 0 to 10. You can type into this text box to set the value by hand.

With only a few lines of code you created a rotary knob with all the essential features, it just doesn’t do anything useful yet. For that, it needs to be connected to the synth’s **Output Level** parameter.

You might think that you’d need to set a listener on the knob and handle the notifications in the `JX11AudioProcessorEditor` class, then get access to the audio processor’s `outputLevelParam` somehow and change its value in a thread-safe way, perhaps by calling `setValueNotifyingHost`. Plus, you’d need to tell the `Slider` object that this parameter goes from –24 dB to +6 dB, so that it can show the correct range instead of 0 – 10.

All of that sounds like a lot of work, which is why JUCE makes this easy. Because all the parameters live inside the `AudioProcessorValueTreeState` object, JUCE lets you create a so-called attachment object between a parameter from the APVTS and a slider, that automatically handles the necessary logic. How nice!

Using these attachments is very simple. Open **PluginEditor.h** and add the following lines.

```
using APVTS = juce::AudioProcessorValueTreeState;
using SliderAttachment = APVTS::SliderAttachment;

SliderAttachment outputLevelAttachment { audioProcessor.apvts,
            ParameterID::outputLevel.getParamID(), outputLevelKnob };
```

The attachment is an object of type `juce::AudioProcessorValueTreeState::SliderAttachment`. The `using` statements are for convenience, as the names of these types tend to get rather long.

Constructing the attachment requires a reference to the APVTS. Way back when you added this to the `JX11AudioProcessor`, you made `apvts` a public member variable for exactly this reason: so that the editor can access it. The attachment also needs the ID for the parameter to look up the corresponding `AudioParameterFloat` in the APVTS, and a reference to the `Slider` object.

It’s important that these new lines go below the declaration for the `outputLevelKnob`. When the editor window closes, all the member objects are destroyed in reverse order of declaration, and the attachment must be destroyed before the slider it’s attached to. If your plug-in crashes when the editor window is closed, chances are you got the order of the declarations wrong.

And that’s all you need to do, nothing more to it!

Try out the plug-in. The knob is set to 0 dB initially. Dragging it all the way to the left puts it at –24 dB, all the way to the right puts it at +6 dB. And it works too: play some notes and drag the knob to hear the sound level changing.

- In AudioPluginHost, if you do **Show all parameters** and scroll down to the slider for **Output Level** you’ll see that it gets updated when you drag the knob.
- And vice versa: move the slider from the auto-generated UI and the rotary knob updates in response.
- Typing in a new value in the text box works.
- Alt-clicking or double-clicking should reset the parameter to its default of 0 dB.
- Changing the preset will reload the knob with the preset’s setting.

- Recall that the **Output Level** parameter is connected to a MIDI CC for the master volume control. If you have such a slider on your MIDI controller, move it up and down and you'll see the knob respond in realtime too.

It all works as if by magic! These attachments are a great solution and you will use them all the time when writing UIs with JUCE. Besides rotary knobs / sliders, other components such as buttons and combo box controls also work with attachments.

Adding a few more controls

That knob looks a little lonely there all by itself, so in this section you'll add a few other controls. First up is another knob, this time for the **Filter Reso** parameter. The steps are very similar to what you did before.

In **PluginEditor.h** add the following lines. Again, the declaration for the attachment should be below that of its associated slider object.

```
juce::Slider filterResoKnob;
SliderAttachment filterResoAttachment { audioProcessor.apvts,
          ParameterID::filterReso.getParamID(), filterResoKnob };
```

In **PluginEditor.cpp**, add the next lines to the constructor. It's the same code as before but this time for the `filterResoKnob`.

```
filterResoKnob.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
filterResoKnob.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 100, 20);
addAndMakeVisible(filterResoKnob);
```

You also need to give the new knob a position and size in the editor window, so change `resized` to the following:

```
void JX11AudioProcessorEditor::resized()
{
    outputLevelKnob.setBounds(20, 20, 100, 120);
    filterResoKnob.setBounds(120, 20, 100, 120);
}
```

Try it out, you now have a UI with two rotary knobs side-by-side!

JX11 has a couple of parameters that aren't suitable for a rotary knob, such as the **Polyphony** parameter. Fortunately, JUCE comes with other types of UI controls as well, such as the `juce::TextButton`, which can be used as a toggle button. Let's use one of those to make a toggle between Poly and Mono mode.

In **PluginEditor.h**, add the following lines:

```
using ButtonAttachment = APVTS::ButtonAttachment;

juce::TextButton polyModeButton;
ButtonAttachment polyModeAttachment { audioProcessor.apvts,
          ParameterID::polyMode.getParamID(), polyModeButton };
```

This time the attachment is of type `juce::AudioProcessorValueTreeState::ButtonAttachment` and it's used just like the slider attachment.

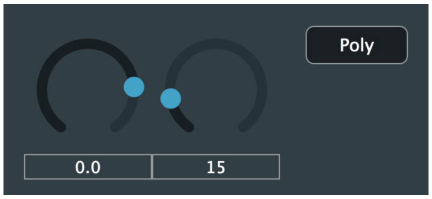
In **PluginEditor.cpp** in the constructor, add these lines:

```
polyModeButton.setButtonText("Poly");
polyModeButton.setClickingTogglesState(true);
addAndMakeVisible(polyModeButton);
```

And in `resized`, add the following:

```
polyModeButton.setBounds(240, 20, 80, 30);
```

Try it out, you now have a button that can toggle Poly on and off. As you can tell from the handful lines of code this required, building up a UI in JUCE doesn't take much effort.



The toggle button is positioned next to the sliders

To be fair, with this toggle button it's not very clear to see whether Mono mode or Poly mode is selected, but you'll fix that in a later section where you'll be styling the button. At least you should see the **Polyphony** parameter change between the two states in the **Show all parameters** window — and you can hear it, of course — so the button does work, it's just not immediately obvious which state it is in.

Instead of a `TextButton` you could also use a `juce::ToggleButton` which draws a tick-box next to the label, or a `juce::ComboBox` that shows multiple choices, or any of the other UI components that come with JUCE. And if all fails, you can always write your own by subclassing `juce::Component`. For the purposes of this chapter, we'll stick with the `TextButton`.

Tip: To use Unicode symbols on the button, you cannot pass a regular string to `setButtonText`. Instead, wrap the string into a `CharPointer_UTF8` object, like so:

```
button.setButtonText(juce::CharPointer_UTF8("I like π"))
```

The reason I added these two extra components was so I could show you some layout tricks. In the version of `resized` above, you called `setBounds` on each component with hardcoded sizes and coordinates. That method certainly works but it can be a bit finicky if you need to make changes. Fortunately, JUCE has a handy `juce::Rectangle` class that can make layouts more convenient. Here's a small taste of what it can do:

```
void JX11AudioProcessorEditor::resized()
{
    juce::Rectangle r(20, 20, 100, 120);
    outputLevelKnob.setBounds(r);

    r = r.withX(r.getRight() + 20);
    filterResoKnob.setBounds(r);

    polyModeButton.setSize(80, 30);
    polyModeButton.setCentrePosition(r.withX(r.getRight()).getCentre());
}
```

This defines the rectangle object `r` as having the size and position of the first knob. To position the `filterResoKnob`, it does `r = r.withX(r.getRight() + 20)` to shift that rectangle to the right by the width of the `outputLevelKnob` plus 20 points padding. You do the same for the `polyModeButton`, but center it vertically.

Try it out to see what the plug-in looks like now. Or, to place the button below the knobs, you'd rewrite the layout code to something like the following.

```
r = r.withY(r.getBottom());
polyModeButton.setCentrePosition({ r.getX(), r.getCentreY() });
```

With this approach, you don't have to worry so much about hardcoded sizes and positions but you can lay out the components relative to each other by defining `Rectangle` objects and manipulating them. This also makes it possible to create resizable plug-ins where the sizes of the UI components grow larger or smaller with the window.

Creating a custom component

One obvious issue with the UI is that it's not really clear which knob does what, since they are not labelled. The `juce::Slider` object does have a `setTitle` method but that's for accessibility purposes, it doesn't draw that title on the screen anywhere.

You can fix this by creating a `juce::Label` component and placing that in the UI above or below the knob. Or you can do it by implementing the `paint` method and using `drawText` to draw the label text in the desired position. In this section, you'll create a `RotaryKnob` subclass that combines the `juce::Slider` object with a label into a custom component.

Creating custom components is easy enough. You make a subclass of `juce::Component` and implement the necessary methods. It's quite likely that for your own UIs you'll be writing custom components too — even though JUCE comes with many built-in controls, there's always the need to do things in a way that the JUCE developers didn't think of.

In **Projucer**, go to the File Explorer to add a new file. From the + button, choose **Add New Component Class (split between a CPP & header)** and name it **RotaryKnob**. This adds two files to your project containing the basic template for making a custom component.

The header file **RotaryKnob.h** looks like this (minus the comments):

```
class RotaryKnob  : public juce::Component
{
public:
    RotaryKnob();
    ~RotaryKnob() override;

    void paint (juce::Graphics&) override;
    void resized() override;

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (RotaryKnob)
};
```

That looks very much like the class definition of the plug-in editor, which is no surprise since that's also a `juce::Component` under the hood.

Add the following two variables to the public section in the header:

```
juce::Slider slider;
juce::String label;
```

The RotaryKnob component consists of a juce::Slider as a child component, as well as a juce::String label that will be drawn inside the bounds of the RotaryKnob by the paint method.

The file **RotaryKnob.cpp** has empty implementations for the constructor, destructor, and resized method, and some default drawing code in paint.

Change the constructor to the following:

```
static constexpr int labelHeight = 15;
static constexpr int textBoxHeight = 20;

RotaryKnob::RotaryKnob()
{
    slider.setSliderStyle(juce::Slider::SliderStyle::RotaryHorizontalVerticalDrag);
    slider.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 100, textBoxHeight);
    addAndMakeVisible(slider);

    setBounds(0, 0, 100, 120);
}
```

This uses similar code as before to create the slider object and add it as a child component. Previously, when you did addAndMakeVisible, it added the slider directly to the JX11AudioProcessorEditor. Here, it adds it to the RotaryKnob component, which in turn will be added to the JX11AudioProcessorEditor. That's typical for a JUCE UI: it's a hierarchy of component objects containing other component objects.

The size of the RotaryKnob component is initially set to 100×120 points, but this can be overridden by the layout code in its parent component. To give the RotaryKnob a flexible size, implement resized as follows:

```
void RotaryKnob::resized()
{
    auto bounds = getLocalBounds();
    slider.setBounds(0, labelHeight, bounds.getWidth(), bounds.getHeight() - labelHeight);
}
```

You first call getLocalBounds to get a juce::Rectangle object with the current width and height of the RotaryKnob component, as set by the resized method from the parent component, which is the JX11AudioProcessorEditor. Then you move the slider into position so that it is exactly as wide as the RotaryKnob, but not as high. The extra space at the top is to make room for the label. This is why you defined the labelHeight constant earlier.

Next, implement paint to draw the text for the label.

```
void RotaryKnob::paint(juce::Graphics& g)
{
    g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));

    g.setFont(15.0f);
    g.setColour(juce::Colours::white);

    auto bounds = getLocalBounds();
    g.drawText(label, juce::Rectangle<int>{ 0, 0, bounds.getWidth(), labelHeight },
               juce::Justification::centred);
}
```

Again this uses `getLocalBounds` to find out how large the component is, in particular how wide, and then it draws the text from the `label` variable centered at the top.

To use the new `RotaryKnob` class, go to **PluginEditor.h** and add an include:

```
#include "RotaryKnob.h"
```

Where the member variables are, replace the declaration for `outputLevelKnob` and `filterResoKnob` and their attachments with:

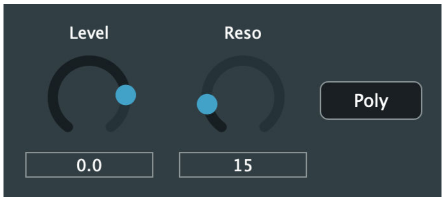
```
RotaryKnob outputLevelKnob;
SliderAttachment outputLevelAttachment { audioProcessor.apvts,
          ParameterID::outputLevel.getParamID(), outputLevelKnob.slider };

RotaryKnob filterResoKnob;
SliderAttachment filterResoAttachment { audioProcessor.apvts,
          ParameterID::filterReso.getParamID(), filterResoKnob.slider };
```

Note that the attachment always expects a reference to a `juce::Slider` object, which is why you now pass it `outputLevelKnob.slider` instead of just `outputLevelKnob`.

In **PluginEditor.cpp**, in the constructor, replace the code for configuring the sliders with the following. You no longer need to call `setSliderStyle` and `setTextBoxStyle` here.

```
outputLevelKnob.label = "Level";
addAndMakeVisible(outputLevelKnob);

filterResoKnob.label = "Reso";
addAndMakeVisible(filterResoKnob);
```

Try it out, the knobs have labels drawn above them with their names.



Now you know what the knobs are for

When creating custom components it can be useful to determine exactly how large they are and how much space each child component takes up. A nice trick is to make the `paint` method draw some colored rectangles. For example, add the following to the bottom of `paint` in **RotaryKnob.cpp** to draw an outline around the entire component, which is useful to see where it begins and ends:

```
g.setColour(juce::Colours::red);
g.drawRect(getLocalBounds(), 1);
```

Or in even more detail, if you want to see where the label and slider are. Drawing such rectangles is an indispensable aid when debugging your layout and drawing code.

```
g.setColour(juce::Colours::yellow);
g.drawRect(0, labelHeight, bounds.getWidth(),
           bounds.getHeight() - labelHeight - textBoxHeight, 1);

g.setColour(juce::Colours::green);
g.drawRect(0, 0, bounds.getWidth(), labelHeight, 1);
```

Go ahead and change the layout code in **PluginEditor.cpp** to use different sizes for the knobs and everything should automatically adapt. For example, set the initial rectangle to the following to make the sliders twice as large:

```
juce::Rectangle r(20, 20, 200, 220);
```

No matter how big or small you make this rectangle, the rotary slider grows (or shrinks) in size and the label is always centered above it. The only thing that doesn't resize is the text box, but I think that's fine.

Great, you've seen how to take an existing JUCE UI component and improve on it by adding it inside a custom component. Now it's time to make it look good!

Styling the controls

JUCE uses the concept of a “look-and-feel” to determine what the UI elements look like. The sliders don’t do their own drawing, they use a separate look-and-feel or LNF object for this. That way you can simply swap in a different LNF without having to change the code in `juce::Slider` or having to make a subclass, and all your sliders immediately look different.

There are also look-and-feel definitions for the other UI components, as well as for colors and fonts. You’ve already seen some of this in action. In `JX11AudioProcessorEditor`’s `paint` method, it does the following:

```
g.fillAll(getLookAndFeel().findColour(juce::ResizableWindow::backgroundColourId));
```

This asks the active look-and-feel object for the color with the identifier `backgroundColourId`. To fill the window with a different color you could change this code to something like:

```
g.fillAll(juce::Colours::blue);         // use a predefined color
g.fillAll(juce::Colour(60, 120, 240));  // or create your own color
```

But the preferred method is to create a new look-and-feel object that changes the color for `backgroundColourId`. All the components in your code that ask for the `backgroundColourId` will automatically get the new color. The big advantage of this is that you only have to define your colors in one place.

Since JUCE’s default style is some of the ugliest UI I’ve ever seen, let’s write our own look-and-feel classes and make the synth look less dreadful. In **Projucer**, add a C++ source file & header file to the project named **LookAndFeel**. Add the following code to **LookAndFeel.h**:

```
#include <JuceHeader.h>

class LookAndFeel : public juce::LookAndFeel_V4 {
public:
    LookAndFeel();

private:
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR(LookAndFeel)
};
```

This new class extends from `juce::LookAndFeel_V4`. There are several other versions of this look-and-feel system, but V4 is the latest. It’s worth having a look at the documentation for that class[53], to get an idea of how extensive this system is. All the `draw...` methods are things you can override to customize how things look.

In **LookAndFeel.cpp**, put the following code:

```
#include "LookAndFeel.h"

LookAndFeel::LookAndFeel()
{
    setColour(juce::ResizableWindow::backgroundColourId, juce::Colour(30, 60, 90));

    setColour(juce::Slider::rotarySliderOutlineColourId, juce::Colour(0, 0, 0));
    setColour(juce::Slider::rotarySliderFillColourId, juce::Colour(90, 180, 240));
    setColour(juce::Slider::thumbColourId, juce::Colour(255, 255, 255));
}
```

For now this just overrides the `backgroundColourId` property with a color I chose myself, plus the colors for the different parts of the slider. Feel free to fill in your own RGB values here.

To use this new look-and-feel, go to **PluginEditor.h** and add an include:

```
#include "LookAndFeel.h"
```

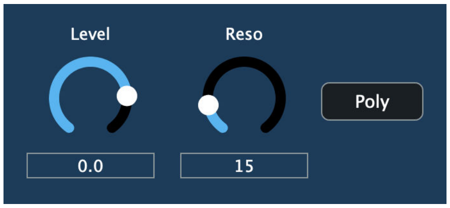
Also add a new private variable:

```
LookAndFeel globalLNF;
```

The final step is to enable this look-and-feel. This happens in **PluginEditor.cpp**. Add the following line to the constructor:

```
juce::LookAndFeel::setDefaultLookAndFeel(&globalLNF);
```

Try it out! The background color has changed to a dark shade of blue and the sliders are drawn in light blue. That already looks a lot better. Notice that the background color for the `RotaryKnob` changed to this color too, since that uses the same `backgroundColourId` color.



New colors for the sliders

[53] https://docs.juce.com/master/classLookAndFeel__V4.html

This may seem like a roundabout way to change some colors, but it's nice that in one swoop you were able to set the colors of all the sliders in the UI. And there's a lot more we can do! Now let's customize how the slider gets drawn.

Add the following public method declaration to the class in **LookAndFeel.h**.

```
void drawRotarySlider(juce::Graphics& g, int x, int y, int width, int height,
                      float sliderPos, float rotaryStartAngle,
                      float rotaryEndAngle, juce::Slider& slider) override;
```

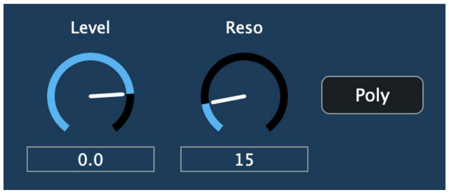
Put the implementation in **LookAndFeel.cpp**. Feel free to copy-paste this code from the GitHub repo instead of typing it all in by hand.

```
void LookAndFeel::drawRotarySlider(
    juce::Graphics& g, int x, int y, int width, int /*height*/, float sliderPos,
    float rotaryStartAngle, float rotaryEndAngle, juce::Slider& slider)
{
    auto outlineColor = slider.findColour(juce::Slider::rotarySliderOutlineColourId);
    auto fillColor = slider.findColour(juce::Slider::rotarySliderFillColourId);
    auto dialColor = slider.findColour(juce::Slider::thumbColourId);

    auto bounds = juce::Rectangle<int>(x, y, width, width).toFloat()
                          .withTrimmedLeft(16.0f).withTrimmedRight(16.0f)
                          .withTrimmedTop(0.0f).withTrimmedBottom(8.0f);

    auto radius = bounds.getWidth() / 2.0f;
    auto toAngle = rotaryStartAngle + sliderPos * (rotaryEndAngle - rotaryStartAngle);
    auto lineW = 6.0f;
    auto arcRadius = radius - lineW / 2.0f;

    auto arg = toAngle - juce::MathConstants<float>::halfPi;
    auto dialW = 3.0f;
    auto dialRadius = arcRadius - 6.0f;

    auto center = bounds.getCentre();
    auto strokeType = juce::PathStrokeType(lineW, juce::PathStrokeType::curved,
                                           juce::PathStrokeType::butt);

    juce::Path backgroundArc;
    backgroundArc.addCentredArc(center.x, center.y, arcRadius, arcRadius, 0.0f,
                                rotaryStartAngle, rotaryEndAngle, true);
    g.setColour(outlineColor);
    g.strokePath(backgroundArc, strokeType);

    if (slider.isEnabled()) {
        juce::Path valueArc;
        valueArc.addCentredArc(center.x, center.y, arcRadius, arcRadius, 0.0f,
                               rotaryStartAngle, toAngle, true);
        g.setColour(fillColor);
        g.strokePath(valueArc, strokeType);
    }

    juce::Point<float> thumbPoint(center.x + dialRadius * std::cos(arg),
                                  center.y + dialRadius * std::sin(arg));
    g.setColour(dialColor);
    g.drawLine(center.x, center.y, thumbPoint.x, thumbPoint.y, dialW);
    g.fillEllipse(juce::Rectangle<float>(dialW, dialW).withCentre(thumbPoint));
    g.fillEllipse(juce::Rectangle<float>(dialW, dialW).withCentre(center));
}
```

I'm not going to explain in detail what happens here, since the specifics aren't that important. It's just a bunch of drawing code, which in JUCE is handled by the `juce::Graphics` class. We draw two arcs, one in black that goes all the way around, and a shorter arc with the active part of the slider in light blue. Then we also draw a line in the middle that acts as the dial.

To get the colors, this doesn't do `getLookAndFeel().findColour()` like you saw before. Instead, it asks the `juce::Slider` object through `slider.findColour()`. That in turn will go out and get the colors from the look-and-feel. This is done so that you can override the LNF colors on sliders that you want to look different.

Try it out, you now have knobs with a customized look-and-feel:



Finally they look like synthesizer knobs

How did I know how to write this drawing code for the slider? Simple, I went into the JUCE source code and looked up how `juce::LookAndFeel_V4` implemented this. Sometimes you need to look at one of the superclasses such as `LookAndFeel_V3` to find the actual drawing code. Then I copied it into my own class and started messing around with it until I got something that looked nice.

One more thing you can customize is the starting and ending angles of the knob, i.e. how many degrees it takes to go from left to right. I think the default is a bit too extreme, so add the following line to the constructor in **RotaryKnob.cpp**.

```
slider.setRotaryParameters(juce::degreesToRadians(225.0f),
                           juce::degreesToRadians(495.0f), true);
```

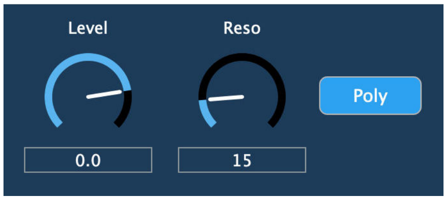
Let's adjust the colors of the `TextButton` so that it's more obvious whether Poly mode is enabled or not. Add the following lines to the `LookAndFeel` constructor:

```
setColour(juce::TextButton::buttonColourId, juce::Colour(15, 30, 45));
setColour(juce::TextButton::buttonOnColourId, juce::Colour(90, 180, 240));
setColour(juce::TextButton::textColourOffId, juce::Colour(180, 180, 180));
setColour(juce::TextButton::textColourOnId, juce::Colour(255, 255, 255));
setColour(juce::ComboBox::outlineColourId, juce::Colour(180, 180, 180));
```

Key to figuring out how to do this, is to pick the correct "ColourIds" used by the drawing code for this component. You can find these in the documentation for the component, although it's not always obvious and sometimes they use the color ID from another component.

For example, to change the outline color of the text button, you needed to use `ComboBox::outlineColourId`, because that happens to be the color the default look-and-feel picks for drawing it. You may sometimes need to dig through JUCE's look-and-feel code to understand exactly what thing to customize.

Now the knobs and button look like this, nice! When polyphonic mode is on, the button turns light blue.



After styling the knobs and button

You can also completely override how every part of the button gets drawn. Check out the look-and-feel methods `drawButtonBackground`, `getTextButtonFont`, and `drawButtonText` for this.

> **Tip:** You don't need to put all the drawing code into a single `LookAndFeel` class. It's possible to create different LNF objects for different purposes, for example if your synth needs different styles of rotary knobs.

Embedding binary data into the plug-in

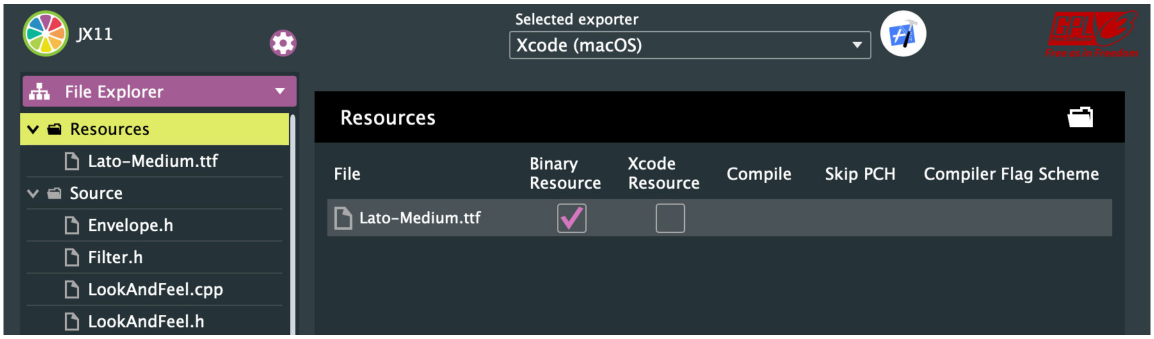
This was only a whirlwind tour through JUCE's functionality for building user interfaces but there is one more essential feature I need to show and that is binary data. While it is possible for your plug-in to load things like fonts and images from external files, it's customary to embed all this data into the plug-in DLL itself. JUCE makes this easy.

The repo for this book includes a **Resources** folder containing the Lato font. This font is freely redistributable under the SIL Open Font License (see the file **OFL.txt** for the exact terms). You will now put this font into the plug-in and use it from the look-and-feel.

Adding binary data is a piece of cake in Projucer. Go to the **File Explorer**, click the + button and choose **Add New Group**. Name this group **Resources**. Using a new group will keep the data files separate from the source files.

Click + again and choose **Add Existing Files...**. Select the **Lato-Medium.ttf** file from the book repo's Resources folder. If you're the drag-and-droppy type, you can also drag this file into Projucer.

If you select the **Resources** header, it shows a summary of the different files that are added inside this group. Notice how **Lato-Medium.ttf** has a checkmark under **Binary Resource**. This tells JUCE to treat it as a special binary file instead of a source file.



File	Binary Resource	Xcode Resource	Compile	Skip PCH	Compiler Flag Scheme
Lato-Medium.ttf	☑	□			

The Lato font in Projucer

Click the export button to make the changes in your IDE.

Even though the Lato-Medium.ttf file will be listed in the IDE as one of the project's files, it's not actually added to the build. Instead, go to the **JUCE Library Code** group and expand it. There are two files in this group that we're interested in: **BinaryData.h** and **BinaryData.cpp**. These files were generated by Projucer when you exported the project, and contain the contents of Lato-Medium.ttf, plus any other binary data that you may add to this project.

Let’s have a quick look at **BinaryData.h**. The important parts for us are the two definitions near the top.

```
namespace BinaryData
{
    extern const char*   LatoMedium_ttf;
    const int            LatoMedium_ttfSize = 663564;

    // ...other stuff...
}
```

You can access the font through `BinaryData::LatoMedium_ttf`, which is a pointer to the raw data from the Lato-Medium.ttf file. The size of that file in bytes is given by `LatoMedium_ttfSize`.

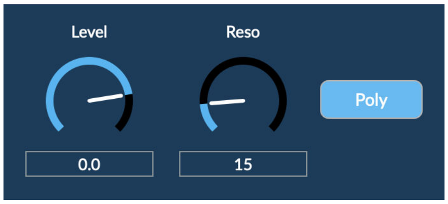
The file **BinaryData.cpp** has the actual contents from the file, encoded as a large C array:

```
namespace BinaryData
{
//================== Lato-Medium.ttf ==================
static const unsigned char temp_binary_data_0[] =
{ 0,1,0,0,0,17,1,0,0,4,0,16,71,80,79,83,120,98,80,99,0,8,171,72,0,1,75,... };

const char* LatoMedium_ttf = (const char*) temp_binary_data_0;
```

To use the font from this binary blob in your plug-in, add the following lines to the constructor in **LookAndFeel.cpp**:

```
juce::Typeface::Ptr typeface = juce::Typeface::createSystemTypefaceFor(
                BinaryData::LatoMedium_ttf, BinaryData::LatoMedium_ttfSize);

setDefaultSansSerifTypeface(typeface);
```

This constructs a new `juce::Typeface` object from the pointer to the font data, and then makes it the standard font. Try it out, all the text in the plug-in now uses Lato.



Using a different font for the text labels

Another typical use of JUCE's binary data feature is embedding images into the plug-in. Let's say you added an image called Logo.png to Projucer. You can load this image with the following code snippet, and then draw it inside the `paint` function using `drawImage`. Try this out as an exercise. You can look up the exact arguments for `drawImage` in the `juce::Graphics` documentation.

```
auto image = juce::ImageCache::getFromMemory(BinaryData::Logo_png,
                                             BinaryData::Logo_pngSize);
graphics.drawImage(image, ...);
```

You have now tasted a bit of what it takes to create a more appealing UI for JX11. Of course, JUCE has many other UI elements besides the `Slider` and the `TextButton`. For example, you could use a `GroupComponent` to place relaced knobs together inside their own area, or add a `ListBox` that lets the user select presets, or… I'll let your imagination do the work here. Have fun!

MIDI Learn

Several of the synthesizer's sound-making properties can be adjusted by a MIDI CC message, such as the filter cutoff and the resonance. So far the MIDI CC numbers for this have been hardcoded in `Synth`'s `controlChange` method, but these numbers won't work for everyone's MIDI controller. Even though MIDI has been standardized, not all manufacturers make the same choices when it comes to assigning CC codes to knobs and sliders.

Many synths therefore offer a MIDI Learn function that can assign any arbitrary MIDI CC message to the synth's functions. This works by choosing the function you want to configure and then putting the synth in MIDI Learn mode. The synth waits until the user twiddles one of the knobs or sliders on their MIDI controller, and grabs the MIDI CC number from the first message it receives. From that point on, that will be the MIDI CC number the synth will use for this function.

In this section you'll build a basic MIDI Learn function for the filter resonance setting. Currently this is hardcoded to use MIDI CC `0x47` but when you're done here, users can assign any MIDI CC number to it that they want.

First we need to decide on a user interface for the MIDI Learn feature. For this book you're going to keep it simple and just add a single button to the UI. Clicking it puts the synth into MIDI Learn mode. The text on the button will change to indicate the synth is waiting for the user to send a MIDI CC event. Once that is received, the button returns to its original state.

This is the sequence of events that needs to take place:

1. When the button is clicked, notify the audio processor that that we're in MIDI learn mode. (UI thread)
2. The audio processor waits for a MIDI CC message to arrive. (Audio thread)
3. Once a MIDI CC message is received, use its CC number for modulating the filter resonance from now on. (Audio thread)
4. Tell the editor that the learning session is over and restore the button. (UI thread)

It's clear that some of these responsibilities happen in the UI and some in the audio processor, so we need to be careful about synchronizing this and not getting into any threading issues.

First, let's make `Synth` flexible enough so it can check for any MIDI CC number for the resonance modulation. In **Synth.h** add a new public variable:

```
uint8_t resoCC = 0x47;
```

This is a `uint8_t` since the MIDI CC number is only one data byte long. It gets the default MIDI CC number of `0x47`.

In **Synth.cpp**, in `controlChange`, remove the case block that checks for code `0x47`, and add the following lines at the bottom of the method. You need to do it like this because a switch statement in C++ cannot check case values that are not constant expressions.

```
// Resonance
if (data1 == resoCC) {
    resonanceCtl = 154.0f / float(154 - data2);
}
```

That's all the changes for `Synth`. The rest of the MIDI Learn code happens in the audio processor and the editor.

The MIDI Learn button

There needs to be a variable that keeps track of whether the MIDI Learn function is currently active and waiting for a MIDI CC event. A good place for this variable is inside `JX11AudioProcessor`, as that is where you'll be scanning for the MIDI CC messages.

We have to be careful: this variable needs to be set in response to a UI event, but must also be read from the audio thread since that's where incoming MIDI events are handled. Since this is a variable that is accessed by two threads at once, and we can't use any kind of mutex, it should be an atomic variable.

In **PluginProcessor.h** add the following declaration to the public members section:

```
std::atomic<bool> midiLearn;
```

The `midiLearn` boolean determines whether the plug-in is currently in learning mode or not. In **PluginProcessor.cpp**, in `reset`, set this variable to false:

```
midiLearn = false;
```

Next you'll write the code to handle the MIDI Learn button. When this button is pressed, it will set `midiLearn` to true and the audio processor will start scanning for a MIDI CC event.

In **PluginEditor.h** add the following private variable declaration:

```
juce::TextButton midiLearnButton;
```

This time there is no need for an attachment as this button is not linked with any of the plug-in parameters. But we do need to respond to clicks on the button, so make the editor a button listener by also deriving from `juce::Button::Listener`.

```
class JX11AudioProcessorEditor  : public juce::AudioProcessorEditor,
                                  private juce::Button::Listener
```

This requires you to override the `buttonClicked` method, so add its declaration to the private members:

```
void buttonClicked(juce::Button* button) override;
```

Switch to **PluginEditor.cpp** and add the following lines to the constructor:

```
midiLearnButton.setButtonText("MIDI Learn");
midiLearnButton.addListener(this);
addAndMakeVisible(midiLearnButton);
```

The listener must be removed when the editor window is closed. This happens in the destructor.

```
JX11AudioProcessorEditor::~JX11AudioProcessorEditor()
{
    midiLearnButton.removeListener(this);
}
```

And of course you need to put the button somewhere inside the editor window. Add these lines to `resized`:

```
midiLearnButton.setBounds(400, 20, 100, 30);
```

For now, let's see if the button works. Implement the `buttonClicked` listener method like so:

```
void JX11AudioProcessorEditor::buttonClicked(juce::Button* button)
{
    DBG("button clicked");
}
```

Try it out. There should be a new **MIDI Learn** button in the UI. Every time you click the button, it prints a message to the IDE's debug output pane.

Once the user clicks the MIDI Learn button, it should be disabled until a MIDI CC code is received so that the user can't click it more than once. Change `buttonClicked` to the following:

```
void JX11AudioProcessorEditor::buttonClicked(juce::Button* button)
{
    button->setButtonText("Waiting...");
    button->setEnabled(false);
    audioProcessor.midiLearn = true;
}
```

That's all you have to do here. A click on the button sets `midiLearn` to true, which means it's the audio processor's turn to do some work.

Scanning for the MIDI CC message

Go to **AudioProcessor.cpp**. At the top of the `handleMIDI` function add the following logic:

```
void JX11AudioProcessor::handleMIDI(uint8_t data0, uint8_t data1, uint8_t data2)
{
    if (midiLearn && ((data0 & 0xF0) == 0xB0)) {
        DBG("learned a MIDI CC");
        synth.resoCC = data1;
        midiLearn = false;
        return;
    }

    // ...existing code...
```

While MIDI Learn is active, `handleMIDI` intercepts any MIDI CC event and it uses the CC number (in `data1`) from the message that arrives first, ignoring the channel it was sent on. However, it lets any other MIDI events through, so you can do the MIDI Learn process while you're playing music.

Try it out, put the plug-in into MIDI Learn mode by clicking the button and fiddle with a knob or slider on your MIDI controller, even the mod wheel will do. You should get the message "`learned a MIDI CC`" in the debug console, and from then on you can change the filter resonance using that particular knob or slider.

After learning the new MIDI CC number, put **Filter Freq** to around 50% and play a big chord. Now interact with the learned control on your MIDI controller and in the frequency analyzer you should see the resonant peak increase and decrease.

> **Note:** Unlike the MIDI CC for the **Output Level** parameter, the MIDI CC for the resonance does not directly change the **Filter Reso** parameter but adds a bit of extra resonance on top. So when you use your MIDI controller to change the resonance, you won't see the **Reso** knob move in the UI. Some synths draw an extra bar around the knob to indicate the active amount of modulation. As an exercise, perhaps you could try something similar with the extra resonance amount coming from the MIDI CC.

Of course, there are some obvious flaws. Most notably is that the button in the UI stays disabled even after the learning finishes. This is because no one told it that we are done. It might be tempting to do this from the audio processor. You could get a reference to the editor using `getActiveEditor()`, and maybe make a method that tells the button to enable itself again. However, handleMIDI runs in the audio thread and UI stuff should happen in the UI thread. So that is not the right approach.

A better solution is for the editor to recognize that the user is done with MIDI Learning when midiLearn becomes false again. It's OK to access midiLearn from multiple threads, since it's atomic. However, there is no concept of a "listener" for atomic variables. There are different ways to go about this but a simple approach is to make the editor periodically check whether midiLearn is still true. When it sees that midiLearn has become false, it can enable the button again and stop checking.

The way to implement this in JUCE is with a timer object. Open **PluginEditor.h** and also make the class inherit from juce::Timer.

```
class JX11AudioProcessorEditor  : public juce::AudioProcessorEditor,
                                  private juce::Button::Listener, juce::Timer
```

In the private members, declare the method that will handle the timer:

```
void timerCallback() override;
```

Go to **PluginEditor.cpp**. We want to start the timer in the buttonClicked method, so add the following line to that method:

```
startTimerHz(10);
```

The timer will start running and call the timerCallback function ten times per second. Fortunately for us, this callback happens on the JUCE message thread, so we can safely access any UI components from there. Implement it as follows:

```
void JX11AudioProcessorEditor::timerCallback()
{
    if (!audioProcessor.midiLearn) {
        stopTimer();
        midiLearnButton.setButtonText("MIDI Learn");
        midiLearnButton.setEnabled(true);
    }
}
```

Once midiLearn is set back to false (by the audio thread, but that does not matter since the variable is atomic), this stops the timer again and restores the button.

Periodically polling the state of the audio processor is a common pattern when communicating between the UI and the audio thread, and it's safe because the only variable they have in common is an atomic boolean. The UI thread sets this boolean to true, the audio thread sets it to false again. A very simple communications protocol, but effective.

When using timers there's always the edge case that the user closes the editor window while the timer is still going. In JUCE there is nothing special you need to handle here. The timer will automatically stop when the editor window is destroyed.

However, it's a good idea to cancel the MIDI Learning when the editor window closes, so add the following line to the `JX11AudioProcessorEditor` destructor:

```
audioProcessor.midiLearn = false;
```

Awesome, try it out. You now have a working MIDI Learn function and it wasn't even that complicated to write!

Remembering the learned MIDI CC number

For a good user experience, the plug-in ought to remember the MIDI CC number that it has learned, so that the next time the user opens their project they won't have to use MIDI Learn again. There are different strategies for how and where to store this information, but we'll add it to the plug-in's state when it gets saved by the host.

Recall that saving and restoring the plug-in's state happens in the methods `getStateInformation` and `setStateInformation` in the audio processor. Currently they serialize the contents of the APVTS to XML, but you can also add your own information into this XML data. Let's add the MIDI CC code in there too.

Currently the XML that gets written starts with a `<Parameter>` top-level element that encodes the contents of the APVTS inside `<PARAM>` tags. We'll have to reorganize this a little so that there's a new top-level element that contains separate child elements for the `<Parameter>` section as well as the learned MIDI CC number.

Let's do that refactoring first and then worry about how we're going to store the MIDI CC data. In **PluginProcessor.cpp** add a new declaration. This will be the new name of the top-level XML element.

```
static const juce::Identifier pluginTag = "PLUGIN";
```

Then rewrite `getStateInformation` to the following.

```
void JX11AudioProcessor::getStateInformation (juce::MemoryBlock& destData)
{
    auto xml = std::make_unique<juce::XmlElement>(pluginTag);

    std::unique_ptr<juce::XmlElement> parametersXML(apvts.copyState().createXml());
    xml->addChildElement(parametersXML.release());

    copyXmlToBinary(*xml, destData);

    DBG(xml->toString());
}
```

If you now run the plug-in and save the session from within the host, the `DBG` statement will print the following XML structure:

```
<?xml version="1.0" encoding="UTF-8"?>

<PLUGIN>
  <Parameters>
    <PARAM id="envAttack" value="0.0"/>
    <PARAM id="envDecay" value="50.0"/>
    ...
```

All the APVTS data is still in there, except it's one level deeper under the new `<PLUGIN>` tag. You also have to modify `setStateInformation` to match this new structure, or else it won't be able to restore the saved plug-in state anymore.

```
void JX11AudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    std::unique_ptr<juce::XmlElement> xml(getXmlFromBinary(data, sizeInBytes));
    if (xml.get() != nullptr && xml->hasTagName(pluginTag)) {
        if (auto* parametersXML = xml->getChildByName(apvts.state.getType())) {
            apvts.replaceState(juce::ValueTree::fromXml(*parametersXML));
            parametersChanged.store(true);
        }
    }
}
```

This is not so different from the previous code, except it first checks if the `<PLUGIN>` tag exists, and then loads the parameter values back into the APVTS.

Try it out, you should be able to load and save plug-in state again. Unfortunately, since you changed the format in which to serialize the synth's state, it can no longer restore state from the old format. So if you made some cool sounding patches and saved them as presets in your DAW, I'm sorry to inform you that these patches can no longer be loaded.

> **Note:** This is why it's a good idea to add a version number to the XML data, so that the plug-in will be able to migrate from an older version to the current one. It's inevitable that you'll be changing the serialization format at some point in the plug-in's lifetime, if only to add support for new features. For example, the `<PLUGIN>` tag could have a version attribute so that it looks like `<PLUGIN version="1">`. How to do this kind of version migration is beyond the scope of this book, but worth thinking about right from the start.

You still need to add the MIDI CC number to the XML, that's why you're doing all this in the first place. But there's a catch: JUCE does not guarantee that `getStateInformation` and `setStateInformation` will be called on any thread in particular, not even the UI thread. Therefore, you cannot access the `synth.resoCC` variable from within these methods. One solution would be to make `synth.resoCC` into an atomic variable but that seems smelly, so we're going to do this in a somewhat roundabout way.

Add a new member variable to **PluginProcessor.h**:

```
std::atomic<uint8_t> midiLearnCC;
```

In **PluginProcessor.cpp**, in `reset`, set this to the same default value as is used in `Synth`:

```
midiLearnCC = synth.resoCC;
```

In `handleMIDI`, instead of writing the learned MIDI CC value directly to `synth.resoCC`, put it in `midiLearnCC`:

```
midiLearnCC = data1;
```

Then in `processBlock`, after the loop that clears the audio buffers, copy the value from `midiLearnCC` into `synth.resoCC`:

```
synth.resoCC = midiLearnCC;
```

The main reason for doing this, is that we can access this `midiLearnCC` variable from any thread, since it's atomic. However, only the audio thread will ever access `synth.resoCC`. It's OK to do this assignment on every call to `processBlock`, even if the value of `midiLearnCC` almost never changes, since atomic variables are fast.

Back to saving and loading the XML. Add the following two new declarations inside **PluginProcessor.cpp**.

```
static const juce::Identifier extraTag = "EXTRA";
static const juce::Identifier midiCCAttribute = "midiCC";
```

Then in `getStateInformation`, before the call to `copyXmlToBinary`, insert these lines:

```
auto extraXML = std::make_unique<juce::XmlElement>(extraTag);
extraXML->setAttribute(midiCCAttribute, midiLearnCC);
xml->addChildElement(extraXML.release());
```

This creates a new XML element `<EXTRA>` and gives it an attribute named `midiCC` with the value of the MIDI CC code that the user has learned (or the default `0x47`, which is 71 in decimal). Try it out, run the plug-in and save the session. The `DBG` statement now prints out the following XML:

```
<PLUGIN>
  <Parameters>
    <PARAM id="envAttack" value="0.0"/>
    <PARAM id="envDecay" value="50.0"/>
    ...
  </Parameters>
  <EXTRA midiCC="71"/>
</PLUGIN>
```

The final step is to load this state back into the plug-in in `setStateInformation`. Inside the outer `if` statement, add the following code:

```
if (auto* extraXML = xml->getChildByName(extraTag)) {
    int midiCC = extraXML->getIntAttribute(midiCCAttribute);
    if (midiCC != 0) {
        midiLearnCC = static_cast<uint8_t>(midiCC);
    }
}
```

This looks for the `<EXTRA midiCC="..."/>` element, reads the attribute and puts it into `midiLearnCC`. Excellent, that concludes our implementation of MIDI Learn! Try it out. Whenever you save the session, it remembers what MIDI CC number it learned. You can add additional settings to this `EXTRA` element too, such as UI state or anything you'd want to save and restore that isn't a plug-in parameter.

Of course this is only a barebones implementation of MIDI Learn but we've covered all the important parts. In a real product, you'd want to streamline the UI for this feature. For example, in some synths you can right-click on a knob and choose the MIDI Learn option from a menu, or immediately assign the control to a specific MIDI CC number.

Accessibility

Before concluding this chapter on user interfaces, we should explore the topic of accessibility. In terms of software design, accessibility means that the software can also be used by people with special needs. With a few small tweaks the UI can become a lot more convenient to use, not just for visually impaired users or people with mobility issues, but for everyone. A good product is designed for people of all abilities.

The JUCE UI components have the following methods to support accessibility:

- `setTooltip`. A tooltip is a small bubble that is displayed when the mouse hovers over the component. You can use this to show a short help text of what the UI control is for, making the features of the synth more discoverable. (Note: to make this work, add a `TooltipWindow` object to the plug-in editor.)
- `setTitle`, `setDescription`, `setHelpText`. These methods provide text descriptions of the purpose of the component to screen reader software, such as VoiceOver on macOS and Narrator on Windows. Good descriptions here make it possible for visually impaired users to navigate the UI.
- `setExplicitFocusOrder`. This determines the order in which UI elements are given keyboard focus when you're using the `<tab>` key to move between them. If you don't set this order, JUCE may not always choose something that is logical. The focus order is used by screen readers but also for keyboard navigation. Making it possible to use the plug-in from the keyboard is a good accessibility feature as not everyone is able to use a mouse or trackpad.
- `setFocusContainerType`, `createFocusTraverser`. These are more advanced methods for defining parent/child relationships between the UI components from a navigation point of view, and are useful for UIs with complex hierarchies.

These same accessibility features can be added to custom components with a little bit of work. It's worth doing! A large part of user interface development is about polishing the product until it looks good and feels good to use, and adding accessibility support is one of those things.

The key thing in making a UI usable for people who may not have 100% functional bodies is to make the navigation between the different UI components as clear and simple as possible.

For more info about JUCE's accessibility features, see this note in the JUCE repo[54].

[54] https://github.com/juce-framework/JUCE/blob/develop/docs/Accessibility.md

A common form of visual impairment is color blindness. When designing your UI and choosing colors, it's a good idea to use a color blindness simulation tool that will show what the UI looks like for people with different forms of this condition. Two colors that may look great together to you may blend into the same color for someone else, but fortunately with these tools it's an easy mistake to avoid.

> **Tip:** If you're on macOS, enable VoiceOver. If you're on Windows, enable Narrator. (Be sure to read the instructions on how to disable this again first!) Now try using the plug-in with your eyes closed. It's tough! However, it gives you a good idea of how hard — or maybe easy? — it is to navigate through your UI for someone who doesn't have full use of their eyes. Be sure to have actual visually impaired users test your plug-in too, their perspective on how they interact with the world can be very illuminating.

Conclusion

In this chapter you've learned how to put together a basic UI for the synth using JUCE's `Component` hierarchy. Be sure to check out the JUCE documentation because we've only scratched the surface of what is possible here.

To communicate between the UI thread and the audio thread you used the APVTS attachment objects that hide all the underlying complexity, and you've also used atomic variables directly. Most of the time, this is all you need.

To send more complex data between the audio and UI threads, such as the result of an FFT that contains the frequency spectrum of the current audio buffer, you'd need to use a lock-free FIFO. This is a special queue data structure that can work between different threads, without requiring blocking synchronization primitives such as a mutex.

Make sure to read this JUCE tutorial on building a frequency analyzer[55] to learn more about FIFOs. (The tutorial is not for making a plug-in but it is easy enough to adapt.)

The advantage of using JUCE is that you write the UI code once and it will work on all the different platforms — macOS, Windows, Linux, iOS, Android. The JUCE UI and graphics libraries are fairly extensive and can do almost everything you can think of. And if they don't, you can create your own custom components.

One downside, however, is that because of the cross-platform abstractions it provides, the JUCE graphics code isn't always the most performant. If you were to rewrite the UI for

[55] https://docs.juce.com/master/tutorial_spectrum_analyser.html

every platform using native libraries, the interface would redraw faster and feel more natural to that platform. However, that's a ton of work and to most audio developers using JUCE's abstractions is an acceptable trade-off between optimal graphics performance and development effort.

That said, if you absolutely need the fastest graphics possible, JUCE lets you drop down to the OS native libraries with `HWNDComponent` or `NSViewComponent`. It also supports OpenGL or Metal for drawing into components, for those plug-ins that need to squeeze graphics performance to the max. But that's a story for another time…

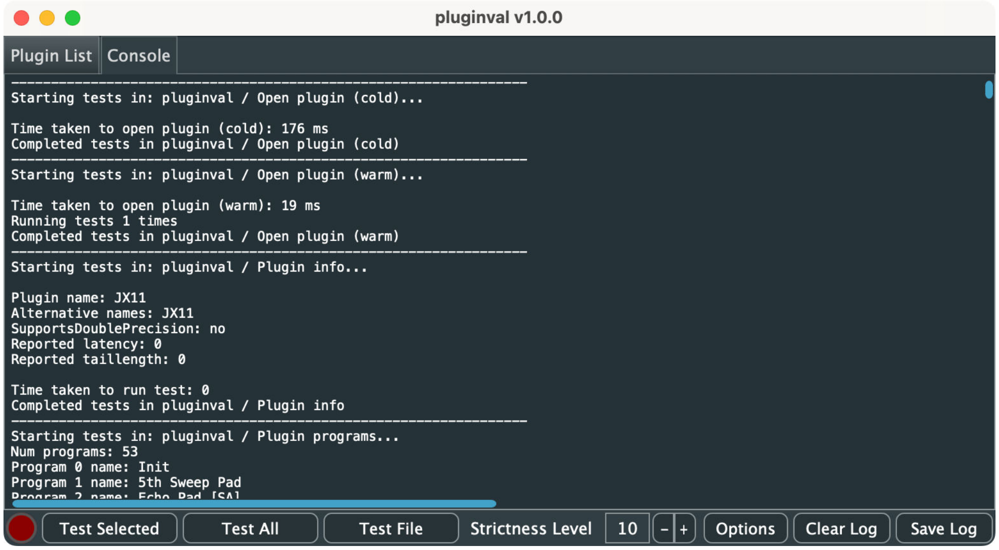
Chapter 14: Where to go from here

Congrats, you've made it all the way to the end of the book! Thanks for joining me on this journey — I hope you had fun writing this cool synthesizer plug-in. In this chapter I'm going to give some ideas for things you can try next, and resources you can use to learn more about audio programming and making synths.

Testing the plug-in

Naturally, before releasing any new product to the public, you'll need to give it a proper round of testing. No matter how good a programmer you are, there will probably be bugs. It is better if it's you that finds these bugs and not your users.

The first thing you should do is run the plug-in through pluginval[56]. This is a handy tool that performs a suite of tests to make sure your plug-in behaves properly. The document Testing plugins with pluginval[57] explains how to install pluginval and how to use it. Give it a go now on JX11!



```
-----------------------------------------------------------------
Starting tests in: pluginval / Open plugin (cold)...

Time taken to open plugin (cold): 176 ms
Completed tests in pluginval / Open plugin (cold)
-----------------------------------------------------------------
Starting tests in: pluginval / Open plugin (warm)...

Time taken to open plugin (warm): 19 ms
Running tests 1 times
Completed tests in pluginval / Open plugin (warm)
-----------------------------------------------------------------
Starting tests in: pluginval / Plugin info...

Plugin name: JX11
Alternative names: JX11
SupportsDoublePrecision: no
Reported latency: 0
Reported taillength: 0

Time taken to run test: 0
Completed tests in pluginval / Plugin info
-----------------------------------------------------------------
Starting tests in: pluginval / Plugin programs...
Num programs: 53
Program 0 name: Init
Program 1 name: 5th Sweep Pad
Program 2 name: Echo Pad [SA]
```

Testing the plug-in with pluginval

[56] https://github.com/Tracktion/pluginval

[57] https://github.com/Tracktion/pluginval/blob/develop/docs/Testing%20plugins%20with%20pluginval.md

Specifically for Audio Units there is also `auval`, which performs many of the same tests as pluginval. This utility is already installed on your Mac. You run it from the Terminal.

```
$ auval -v aumu JX11 Manu
```

The first argument after `auval -v` is the type of the plug-in. `aumu` means it's a synth. For an effect plug-in you'd use `aufx`. The next argument is the 4-character code for the plug-in (JX11). The last argument is the 4-character code for the manufacturer (Manu). These values are defined in Projucer in the main project settings, under **Plugin Manufacturer Code** and **Plugin Code**.

`auval` should finish with the message `AU VALIDATION SUCCEEDED.` If not, you've got some debugging to do.

Another useful test is to run the plug-in with Thread Sanitizer enabled. This is a compiler feature that inserts additional code into your project to verify that variables are not being modified by more than one thread at a time. Thread Sanitizer will show a detailed error message if your code has a data race. This is a good tool for hunting down thread-safety issues.

In Xcode, edit the scheme for the target, for example **JX11 - VST3**. Go to the **Diagnostics** tab and select the **Thread Sanitizer** checkbox. Then run the plug-in in AudioPluginHost. If there are any data races in your code, the debug pane will print `WARNING: ThreadSanitizer: data race` followed by a stacktrace of where the race happened.

Don't forget to turn Thread Sanitizer off again when you're done testing, as some hosts may crash when this is enabled. (At the time of writing, Visual Studio does not ship with a thread sanitizer.)

Unit tests

Writing unit tests is a common practice in software development, and it should be for audio developers too.

The logic inside a synth can get complicated and it's easy to break something in some subtle way without realizing it. For example, managing voice polyphony can get so intricate that it becomes scary to make any changes to this code.

In the original MDA JX10 source code, everything was stuck together in a single source file. That makes it pretty much impossible to add unit testing. In JX11, we separated out the different building blocks into their own files: `Oscillator`, `Filter`, `Envelope`. These structs and classes really should get unit tests.

For something like the polyphonic voice management, a typical test case would be to start playing one or more notes and check that the correct number of voices is active. For the oscillator, test cases could verify that the signal has the correct amplitude and that it repeats after every "period" samples. For the envelope, its level should always be rising when in the attack stage and be constant during the sustain stage. And so on…

For more advanced tests, you can create a set of MIDI files to use as input and also a set of WAV files containing the expected output. The tests read the MIDI files and produce new audio files, and you compare these to the expected output to see if they are identical.

If this seems like a lot of work, it is. However, it's worth the peace of mind. Audio code consists of many pieces of logic that interact in all kinds of complex ways. It's easy to inadvertently break something. What you want to get in that case is a failing unit test, not angry support emails because you shipped a broken plug-in.

Besides, audio code has a lot of weird edge cases — for example, what happens if the attack time on the envelope is set to zero? Writing unit tests forces you to think about such edge cases and cover them with tests.

For C++ tests, I prefer to use the Catch2 testing framework[58]. Try to write some test cases for JX11. Suggestion: begin with `NoiseGenerator` and make sure it outputs values between –1 and +1, and that the next value is always different from the previous one.

Manual tests

Unit tests are great but they won't catch any problems that may happen because of disagreements between the plug-in and the host program. Tools like pluginval and auval help to minimize the risk of running into such issues, but you should still test your plug-in in a DAW. Or even better, in multiple DAWs. Just because your plug-in works OK in one DAW doesn't mean it will work in all of them — they all have their own quirks. Ideally, you should test on all the major DAWs, both on Mac and Windows.

Here's a checklist of things to test:

- Try the plug-in at different sample rates: 44100, 48000, 96000, etc.
- Try the plug-in with different buffer sizes, from smallest to largest.
- Try the plug-in on mono and stereo tracks.
- Put the plug-in on multiple tracks at the same time.

[58] https://github.com/catchorg/Catch2

- Add / remove the plug-in repeatedly from a track while sound is and isn't playing.
- Test that automation of all parameters works. The controls should change values when you play back recorded automation.
- Make sure the plug-in's state is saved and restored correctly when the project is closed and reopened.
- Verify that exporting / bouncing / offline rendering works.

Tip: Make a number of test projects in the DAW. Create a track with some MIDI regions, put your plug-in on the track, and bounce the project to a WAV file. When you've made changes to the plug-in, bounce the project again and compare the files by listening to them. Or use a program such as Audacity to subtract the second WAV from the first one, to see if they null out. If not, then something might be wrong with the plug-in.

There is actually an interesting "bug" in JX11 that only shows up when you use it in a DAW. Try it out: create a MIDI region with a note that starts exactly at time zero. Set this MIDI region to loop and press play. The very first note sounds a little weird, but as it gets repeated the note sounds fine. Any guesses why this happens?

Answer: It's due to the filter smoothing. The `filterZip` variable starts out at `0.0` when the synth is reset, which happens right before playback, and so it will do an exponential fade from zero towards the actual filter modulation settings. That unintentional filter sweep is what you're hearing. This problem does not happen when you're playing live, or when the first note does not start at time zero. Make the note begin a little later and notice the problem going away. How would you fix this issue?

Performance testing

Users like their plug-ins to not eat up too much CPU time. After all, the less CPU usage, the more plug-ins they can run at the same time! As developers, it's tempting to obsess over using as little CPU as possible. Whether that describes you or not, at the very least it's good to get an idea of how efficient or inefficient the plug-in is.

For a rough indication of the CPU usage of your plug-in, you can look at the Activity Monitor or Task Manager. For more detailed results, use the profiling tools that come with your IDE, such as Xcode's Instruments or Visual Studio's Diagnostic Tools. The downside of these approaches is that they measure the CPU usage of the entire host application, not just of your plug-in. Make sure to disable all other plug-ins, then attach the profiling tool to the host application and compare how much CPU it uses with the plug-in enabled vs. disabled.

> **Tip:** Always do these tests with the plug-in compiled in release mode, not debug mode. It's easy to forget to switch to release mode, but performance measurements in debug mode are meaningless and you may end up optimizing the wrong thing. At this point it's good to disable the `protectYourEars` routine, as it's not intended to be part of release builds of the plug-in.

The DAW may have its own CPU meter. Logic Pro has a CPU meter in the **Custom** mode of the time display at the top. REAPER has a detailed meter that you can access by choosing **Performance Meter** from the **View** menu. The FX window in REAPER also shows two CPU usage numbers: the first is the CPU usage of the selected plug-in, the second is for the entire plug-in chain. Other DAWs will have their own ways to measure performance.

Make sure to test this while the plug-in is actually producing sound — in some hosts if the track isn't playing, the plug-in may be paused.

You can also add timing code to the plug-in itself. For example, use `std::chrono::high_resolution_clock` to measure how long each call to `processBlock` takes. The question is how to report this elapsed time, since printing to the debug console from `processBlock` will affect the timing. One solution is to add up the measurements over the course of one second, then print out how much time was taken up during that second. JUCE has a class for this, `AudioProcessLoadMeasurer`.

For automated performance testing, create your own lightweight host program that loads the plug-in and then calls `processBlock` in a continuous loop for a certain amount of time. Creating such a test host is easy enough with JUCE since you can borrow the source code from AudioPluginHost.

Running the plug-in in a loop is similar to what a DAW does when it bounces the project in offline mode, and it lets you measure exactly how efficient your code is. If you make the host produce 100 seconds of audio and it takes the loop 1 second to do this, then the plug-in runs at 100× real-time speed and you can expect it to use only 1% CPU time in real-world usage.

Another useful test is the stress test: In your DAW, put the plug-in on 10 tracks at the same time and give all these tracks MIDI regions so all the plug-in instances will be rendering audio simultaneously. Does this use 10× the CPU time or is it suddenly much worse? What if you use 20 tracks, or more?

A downside of doing the performance test in a "how fast can this run?" loop is that your plug-in may benefit from all its data being in the cache. But when you use ten or more instances at the same time, they may start getting cache misses, which is a more realistic usage pattern.

There are also special applications that can be used to analyze and benchmark plug-ins, such as DSP Testbench[59] and Plugindoctor[60].

Improving real-time performance

What if your plug-in is too slow and burns more CPU time than you'd like?

It depends. Let's say you made a synth that can render extremely realistic vocals, which is something that no synth on the market can currently do, and it takes ten minutes to render one minute of singing. That's pretty slow, 10× slower than real-time, and so this wouldn't even work as a plug-in. However, it takes a lot longer than ten minutes to work with a real vocalist, so to your users it might just be worth the wait.

More realistically, if your plug-in is a CPU hog, you'll probably want to try optimizing it. Here are some suggestions:

- Before you do anything else, use a profiler to determine what part of your plug-in is responsible for making it slow. Don't guess, measure. Create a test environment that runs your `processBlock` in a loop and use a time profiler tool to measure how long everything takes.
- If the profiler shows you're calling a lot of expensive mathematical functions, you might consider replacing these with approximations or look-up tables. There are several fast math approximation libraries available.
- You may have a memory access problem. When a modern CPU loads a variable from RAM, it actually grabs a larger chunk of memory to put into the cache. Key to getting good performance is taking advantage of this caching mechanism because cache memory is much faster than regular RAM. Usually this means being smart about how you're setting up your loops. Code that has a lot of "cache misses" will be unnecessarily slow.
- Use SIMD, which stands for Single Instruction Multiple Data. Your CPU comes with a set of instructions that can perform multiple floating point operations at once. On Intel chips these are SSE/AVX instructions, on Arm chips it is known as Neon. These so-called vector instructions let you perform operations such as addition and multiplication on four or eight floats at once, which gives a 4 - 8× speed up.

[59] https://github.com/AndrewJJ/DSP-Testbench
[60] https://ddmf.eu/plugindoctor/

There is some overhead involved in using SIMD, since data needs to be copied into and out of these special registers, but it's the premier way to speed up floating point stuff. You can use SIMD from C++ using so-called compiler intrinsics, which are functions with names like `_mm256_add_ps`. JUCE has `SIMDRegister` and `FloatVectorOperations` classes to make this easier.

Vectorizing your audio code using SIMD is great for performance, but it does require you to rethink how the code is structured, as now operations need to be done in groups. You can vectorize code "in time", meaning that instead of processing just the current sample, you will process the next four samples at once. In a synth, you can also vectorize across voices, so that it performs the logic for four voices at once.

JX11 processes the audio sample-by-sample. There literally is a loop that goes:

```
for (int sample = 0; sample < sampleCount; ++sample) {
    // update the LFO
    // generate the next noise sample

    for (int v = 0; v < MAX_VOICES; ++v) {
        // get next sample from osc1
        // get next sample from osc2
        // combine osc1 and osc2 into a sawtooth wave
        // apply the next envelope value
        // apply the filter
    }

    // apply gain to this sample
}
```

Another approach is to process the entire block at once for each individual operation and then combine the results. In pseudo code:

```
// render the LFO for sampleCount steps into a temporary buffer
// write sampleCount noise samples into the output buffer

for (int v = 0; v < MAX_VOICES; ++v) {
    // render sampleCount samples of osc1 into a temporary buffer
    // subtract sampleCount samples of osc2 from this buffer
    // take a cumulative sum over the temporary buffer

    // render sampleCount values of the envelope into a temporary buffer
    // multiply the envelope buffer with the oscillator buffer

    // apply the filter to the entire temporary buffer
    // add the temporary buffer to the output buffer
}
```

The benefit of this approach is that it can make better use of the cache, since the synth is not jumping between different tasks all the time. Plus, this kind of structure makes it easier to use SIMD. Even better, modern C++ compilers already can apply auto-vectorization if the loop is simple enough, so you might not even have to write the SIMD code by hand. It requires more memory for a few temporary buffers, but that's a small price to pay.

I prefer to start by writing the plug-in in a sample-by-sample fashion, just like what we did in the book. That makes it easier to understand what the synth is doing and to get the logic right. But once the plug-in does what I want, I'll figure out a SIMD way to make it faster.

> **Tip:** If you care about performance, learn to read assembly code. Your IDE can show the assembly it generates from the C++ code. By inspecting the assembly you can already see if a loop was auto-vectorized by the compiler, and if not, you can experiment to see how to rewrite to loop so that the compiler does the right thing. You should still always measure the performance because longer assembly code doesn't always mean it runs slower, but getting some insight into what the compiler does under the hood is really useful for writing high performance code.

Suggestions for new features

Here are some suggestions for new things that you could add to the JX11 synth.

Since the synth has two oscillators, they can be combined in different ways. Currently `osc2` is subtracted from `osc1`, but that's only one possibility. You could add a new parameter to the synth that lets the user choose how the two oscillators are combined:

- Subtract `osc2` from `osc1`, as it is now.
- Add the two oscillators together.
- Multiply `osc1` by `osc2`. This is known as ring modulation.
- Use `osc2` to determine the phase of `osc1`. This does the kind of phase modulation that is performed by FM synths.
- Hard sync: set `osc1` to a higher rate than `osc2`, but whenever `osc2` ends a cycle also reset the phase of `osc1`.

Or add a third oscillator, which gives even more possibilities for combining them.

By the way, to learn more about FM synthesis, check out the MDA DX10 synth at my GitHub[61]. This is very similar to the synth you've built in this book, but instead of using sawtooth oscillators it combines two sine waves using the FM synthesis method. Worth checking out just to see how it's different from JX11.

You can also allow the user to choose between different oscillator waveforms. JX11 has two sawtooths that can be combined into one square wave. Other common oscillator waveforms are:

- sine wave
- triangle wave
- arbitrary shape from a look-up table (wavetable)
- distorted sine wave

You've already seen how to make a sine oscillator. Many synths add a so-called sub bass oscillator that plays a sine wave one octave lower than the note. You could add a parameter that determines how much to mix this sub-sine wave into the sound, which is great for adding a bit of depth.

Distorted sine waves are fun too. There are many formulas for making them. For example, the following code creates a sine wave approximation between `x` = –1 and `x` = +1:

```
y = x + x * x * x * (r * x * x - 1.0f - r);
```

The value of `r` determines how distorted the shape is. If `r` is 0.5, the above formula creates a sine wave-like shape. For negative values of `r`, for example –2.5, this looks more like a sawtooth shape. You could vary `r` using an LFO.

If you're up for a challenge, try making a supersaw. This combines seven sawtooth oscillators per voice, each slightly detuned and with a different starting phase. The phase is important: If all oscillators were to start at the exact same phase, the small amount of detuning between them creates a "beating" type of effect where the sound appears to fade in and out. By giving each sawtooth a slightly different phase offset, this beating effect doesn't happen.

More is better, so why limit this synth to only one filter, or one filter type? The JX11 filter is a low-pass filter but other common filter types are high-pass, band-pass, or notch. With two filters, you could let the user choose between different routings. For example, each

[61] https://github.com/hollance/mda-plugins-juce

oscillator could have its own filter (each with its own filter envelope), both filters could be put in series, or both filters could be put in parallel.

When designing your synth, there are many choices to make. Synth programming isn't just about knowing how to write the DSP code, it's also about knowing how to design an instrument. And the only way to get experience is to do it. So get cracking, and take what you've learned in this book to design and build your own synth experiments!

If you're looking for inspiration, then study your favorite synth plug-ins or let yourself be inspired by old school hardware synth designs. You can find demos of many of these synths on YouTube and often their manuals are online[62] too. Another good place to look for inspiration is the Eurorack scene of modular hardware synths.

Tip: Also have a look at the other MDA plug-ins[a] that I converted to JUCE. Most of these are FX plug-ins but there are three other synths included:

- MDA Piano: a virtual piano that uses sampled sounds
- MDA EPiano: like the piano but using samples of a Rhodes electronic piano
- MDA DX10: an FM synthesizer

Reading other people's source code is great practice, and after reading this book I'm sure you'll be able to decypher how these other synths work too!

[a] https://github.com/hollance/mda-plugins-juce

Further reading

There aren't that many approachable books on the market that give practical advice about building synthesizer plug-ins but I've found a few that I can recommend. There are also a number of good books on audio programming and DSP in general.

Not all of these books are about writing plug-ins and the source examples are not always in C++. Many DSP books use MATLAB, Python, or even BASIC. If you don't have any experience with MATLAB, it's not that hard to convert MATLAB code to Python since many of the Python numerical programming libraries such as NumPy, SciPy and matplotlib are inspired by MATLAB.

[62] http://www.synthmanuals.com

If you don't know Python, I can recommend learning it. Using Python together with Jupyter notebooks is an excellent way to prototype audio algorithms. In the end, all programming languages are pretty much the same and if you learn the basics of MATLAB or Python syntax, you shouldn't have too much trouble converting the code to C++ or your language of choice.

Audio programming is a form of DSP, so it's a good idea to get a solid grounding in how digital signal processing works. Here is a short selection of quality DSP books:

Think DSP by Allen B. Downey. Available as a printed book but also online as a free PDF[63]. This is a great introduction to the basic principles of digital signal processing. It's written for beginners and doesn't have lots of math. Especially nice is that most of the examples use audio signals. If you haven't studied any DSP at all, I highly recommend starting with this book. It uses Python and Jupyter notebooks, so it's also a good place to learn about those.

The Scientist and Engineer's Guide to Digital Signal Processing by Steven W. Smith. This is a fantastic book that is written for people who are new to DSP. While I recommend that you start with Think DSP because it's more relevant to audio and is a shorter read, this is also a must-read book. It is all about being practical so there is not too much math, and it goes deeply into the subject. This book is about DSP in general, not audio, but it's all stuff you'll need to know anyway. The code examples are in BASIC, but you can simply treat this as pseudocode. This book can be read for free online, although I suggest that you download the PDFs[64] and read those.

Understanding Digital Signal Processing by Richard G. Lyons. Another well-regarded DSP beginner book. This has more math than the other two books I listed but at some point you'll have to get dirty with the math, it's unavoidable. A good book to read after the above two. Available on O'Reilly online.

Introduction to Signal Processing by Sophocles J. Orfanidis. Another good free online textbook[65] that covers all the basics of DSP. Written for university students, so it has more math than the others.

Books that specifically cover sound synthesis:

Designing Software Synthesizer Plugins in C++ and **Designing Audio Effect Plugins in C++** by Will C. Pirkle. These books contain a wealth of relevant information, although the explanations are sometimes confusing (you may want to read a regular DSP book first) and I'm not a fan of the coding style. There are two editions of these books. Personally, I found the first edition of the synth book to be easier to read, but both editions contain useful information. Pirkle's books are among the few that talk about making plug-ins, although

[63] https://greenteapress.com/wp/think-dsp/
[64] http://www.dspguide.com/pdfbook.htm
[65] http://www.ece.rutgers.edu/~orfanidi/intro2sp

they do not use JUCE. The first editions use the AU and VST3 SDKs directly while the second editions use Pirkle's own framework, RackAFX / ASPiK, which is suitable for educational purposes but not for making real plug-ins. The first editions are available on O'Reilly online. I learned a lot from Pirkle's books and they're definitely worth getting.

BasicSynth by Daniel R. Mitchell. A short but enjoyable book that explains the basics of making a software synthesizer in C++. The author does not explain how to make plug-ins and roughly half the book is about building a sequencer that allows you to actually use the synth. Still, it has useful information and it's inexpensive.

The Computer Music Tutorial by Curtis Roads. This is a massive book that discusses all facets of computer music, including a big section on synthesis methods. This is not a programming book but it does describe the algorithms in a fair amount of detail. A great reference for anything related to computer audio and very readable. Every time I read it, I learn something new.

The Theory and Technique of Electronic Music by Miller Puckette. This is an online textbook that covers many synthesis techniques. The language used is Pure Data (Pd), which is more of a prototyping environment than a real language but that should not matter. It's free and worth checking out[66].

Real Sound Synthesis for Interactive Applications by Perry R. Cook. This was one of the first books I bought about sound synthesis and I found it quite confusing. The text is rather terse and not really written for a complete beginner. But once I had a better understanding of DSP, I found this to be a useful little book that explores a variety of synthesis techniques. The code is in C and is outdated, but you can still make it work.

Books about audio processing and effects:

Hack Audio by Eric Tarr. An accessible introduction to audio processing algorithms. The code examples are in MATLAB but Python versions can be found on GitHub. This book is available on O'Reilly online. You can also find useful video tutorials on Eric's website[67].

Audio Effects: Theory, Implementation and Application by Joshua D. Reiss and Andrew McPherson. A good practical book that explains a variety of audio effects. The code is in C++. The final chapter of the book explains how to make plug-ins in JUCE, but as the book is almost a decade old it may no longer be completely up-to-date with the latest versions of JUCE. Available at O'Reilly online.

DAFX: Digital Audio Effects, Second Edition by Udo Zölzer. Another excellent book for learning about audio programming. The examples are in MATLAB. Worth getting for your collection or read it at O'Reilly online.

[66] http://msp.ucsd.edu/techniques/v0.11/book.pdf

[67] https://www.hackaudio.com/

The Art of VA Filter Design by Vadim Zavalishin. This is a free book[68] about designing filters using the Topology Preserving Transform (TPT), which is a mathematical method for taking an analog filter design and making it work in software. I'm not going to lie: this is a tough book to get into. The first chapters are a terse summary of many decades worth of DSP theory and are math-heavy. If you can look past the math, this book is a fascinating look at how filters can be designed. Not a book I would recommend to a beginner but worth a read if you're curious about filters.

Julius O. Smith's publications: The JOS publications[69] at Stanford's CCRMA (Center for Computer Research in Music and Acoustics) are legendary. There are several online books about topics such as filters and physical modeling synthesis, and many other relevant publications. You need to be comfortable with the math, but there's enough here to keep you coming back for a long time.

There is a book about JUCE, *Getting Started with JUCE* by Martin Robinson. Much as I like to support my fellow authors, I cannot recommend getting this book. It's old, it does not describe latest JUCE best practices, and doesn't cover much about audio programming anyway. If you have an account, read it at O'Reilly online and make up your own mind.

Math books:

You can't avoid it, DSP and audio programming involves math. You don't have to be a math wizard to be able to write plug-ins, but it's nice not to panic when you encounter some equations.

Math Overboard! (Basic Math for Adults) by Colin W. Clark. This is a set of two books that explain how math works from the ground up, written for adults who want to learn math, even if you flunked it in school. These are the books I used to brush up on my math skills.

An Introduction to the Mathematics of Digital Signal Processing by F. R. Moore. This is a set of two tutorials published in the Computer Music Journal in 1978 but they're still relevant. The PDFs are behind a paywall but you can easily find them with a bit of googling.

Online materials:

The JUCE website has tutorials[70] and API documentation. If you haven't looked at the tutorials yet, you definitely should. They don't just explain how to use JUCE, but also how to do practical audio programming. Also check out the JUCE forums[71]. And don't forget the example projects that come with JUCE in the examples folder (also accessible from Projucer).

[68] https://www.native-instruments.com/fileadmin/ni_media/downloads/pdf/VAFilterDesign_2.1.0.pdf

[69] https://ccrma.stanford.edu/~jos/pubs.html

[70] https://juce.com/learn/tutorials

[71] https://forum.juce.com/

The Audio Programmer If you learn best by watching videos, I can heartily recommend Joshua Hodge's YouTube channel[72]. He explains how to use JUCE to make audio effects and synths. Here you can also find the videos and live streams of monthly meetups where Josh and his co-host interview experts from the audio development world.

GitHub There is a massive amount of audio code on GitHub and some of it is very high quality. In particular, check out the source code of the Vital[73] and Surge[74] synthesizers. These are two top-notch synths and you can see exactly how they are made, how awesome is that!

Musicdsp.org A collection of algorithms, thoughts, and snippets, gathered from the now defunct Music-DSP mailing list. There are a lot of useful source code snippets[75] here. Just keep in mind that some of these techniques are now considered outdated.

KVR Forums The KVR website is all about audio gear and software, but they also have an audio development forum[76]. As is typical for forums, not everyone is always on their best behavior, but you can also find great discussions here and insights shared by well-known audio developers. Worth keeping an eye on.

A lot of the knowledge of the field is in the form of academic papers. You can often find these papers in PDF format online by googling their title. Sometimes the papers are published by a journal and are behind a paywall.

Journals and organizations for audio development and computer music:

- AES, the Audio Engineering Society[77]. This is a professional society of audio engineers, artists, scientists, producers, and anything else having to do with audio and sound. It covers a lot more than software development. Not only can you learn a lot from being in an organization like this but you can also connect with other members, who might just turn out to be the future users of your plug-ins.
- DAFX conference[78]: An international conference that specializes in digital audio effects. The DAFX book mentioned above is a collection of works that originated here. You can read the papers from previous conferences at this website.
- Computer Music Journal[79]. A quarterly publication that goes back all the way to the 1970s.

[72] https://www.youtube.com/c/TheAudioProgrammer
[73] https://github.com/mtytel/vital
[74] https://github.com/surge-synthesizer/surge
[75] https://www.musicdsp.org/en/latest/
[76] https://www.kvraudio.com/forum/viewforum.php?f=33
[77] https://aes.org
[78] http://www.dafx.de
[79] https://direct.mit.edu/comj

- International Computer Music Association[80]
- Journal of the Acoustical Society of America[81]
- IEEE signal processing society[82]. This covers all DSP, not just audio.

A lot of conferences and organizations swing towards the academic side of things more than the practitioner's side. However, there is a great conference[83], the **Audio Developer Conference** or ADC, that focuses on the people writing the code. It is organized by the people behind JUCE. You can find videos of previous ADC talks on the JUCE YouTube channel[84].

The Audio Programmer Community This is a Discord server with lots of smart and fun people (and me) who love to chat about audio development all day long. This Discord is very welcome to newbies but we also have a number of people who work for audio companies that you know and love. Come and join[85], it's free and you will learn tons.

I hope to see you online or in person at a conference or meetup some time!

[80] http://computermusic.org
[81] https://asa.scitation.org/journal/jas
[82] https://signalprocessingsociety.org
[83] https://audio.dev
[84] https://www.youtube.com/c/JUCElibrary/videos
[85] https://theaudioprogrammer.com/community