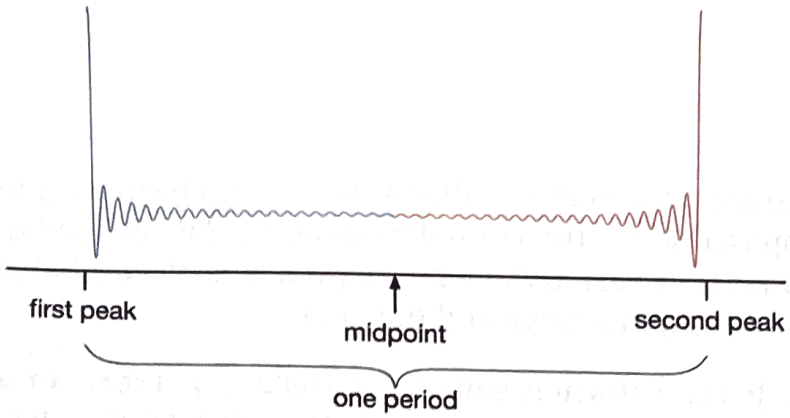
And then mirror that same shape until the peak from the next impulse:



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:

```
#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 $-\pi$ 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.

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.

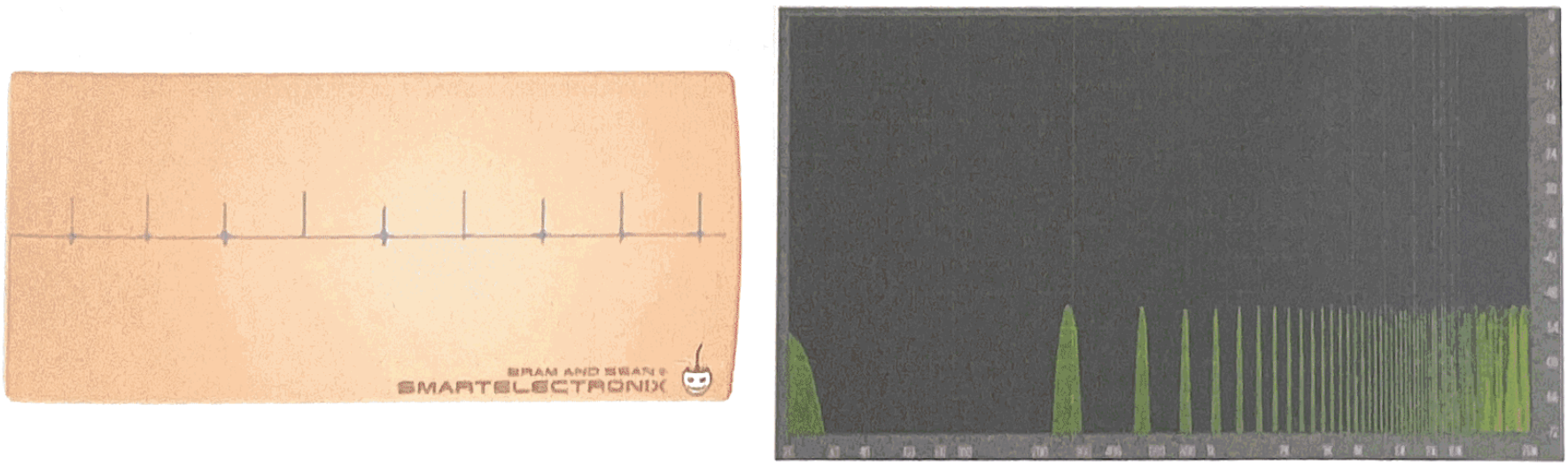
In **Synth.cpp**, change the code in `noteOn` to the following:

```
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. 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.)

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;
```

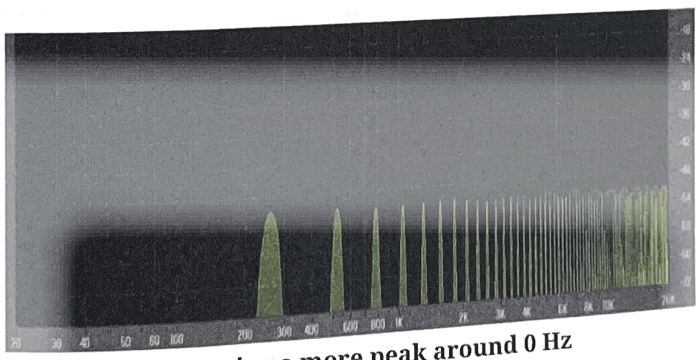
Then in `nextSample`, insert the code to calculate `dc` in the following place:

```
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`:

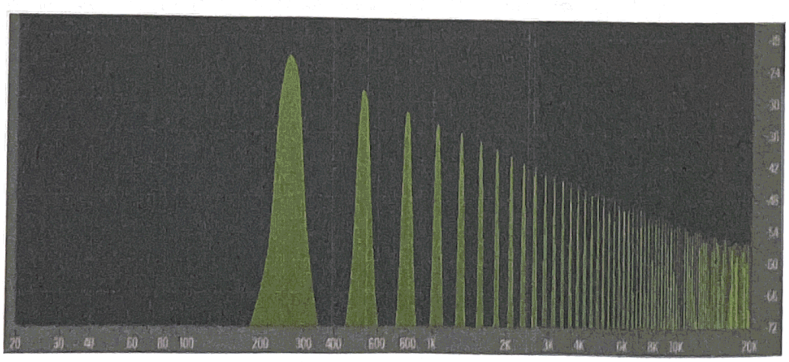
```
saw = 0.0f;
```

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