Additive Synthesis using the CORDIC algorithm

Cesare Ferrari
https://github.com/cesaref/ADC2021

What we’ll cover

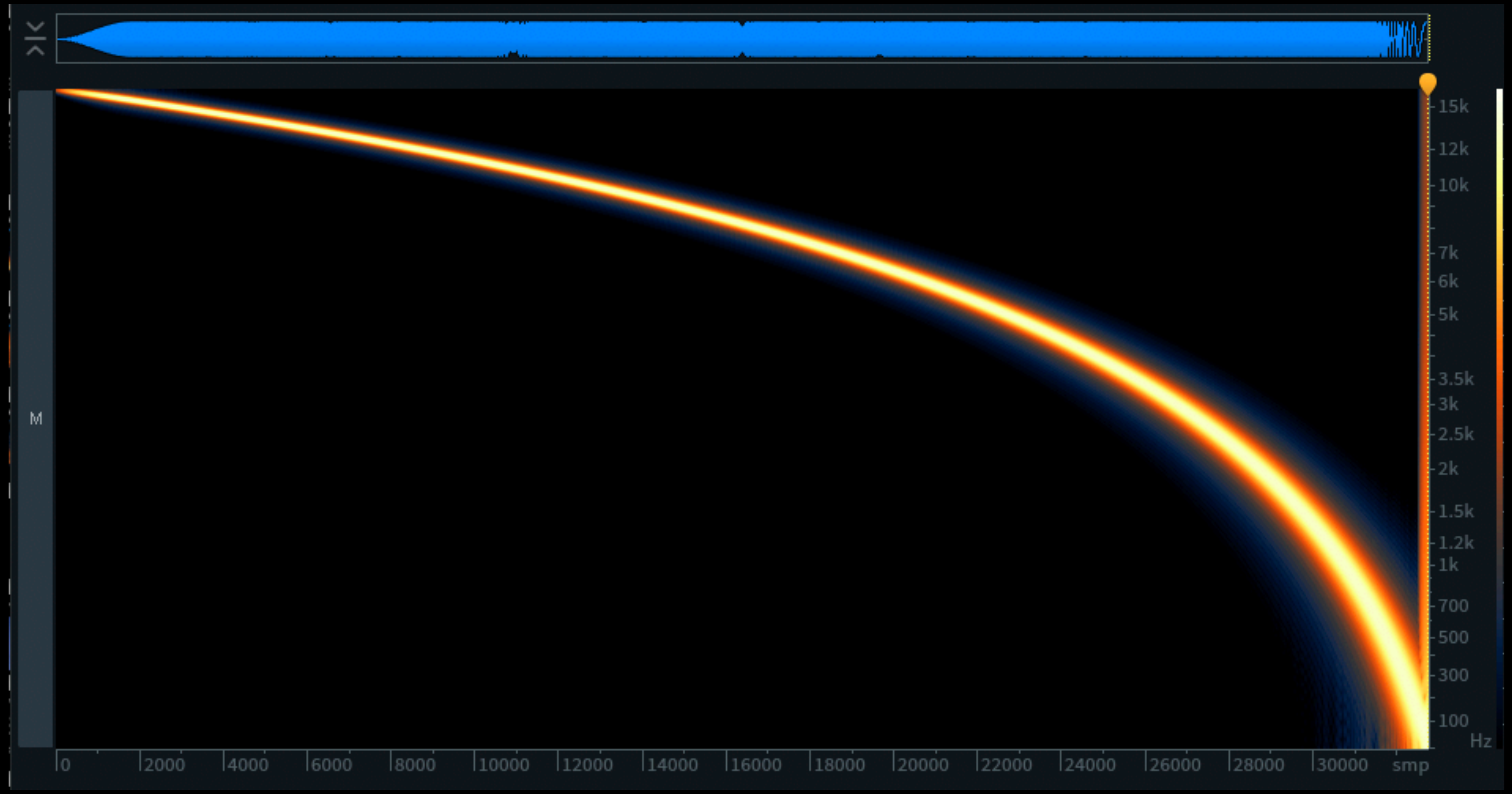
- What is Additive Synthesis?
- The CORDIC algorithm
- Implementing Additive Synthesis efficiently
- Generating harmonic envelopes
- Creative opportunities

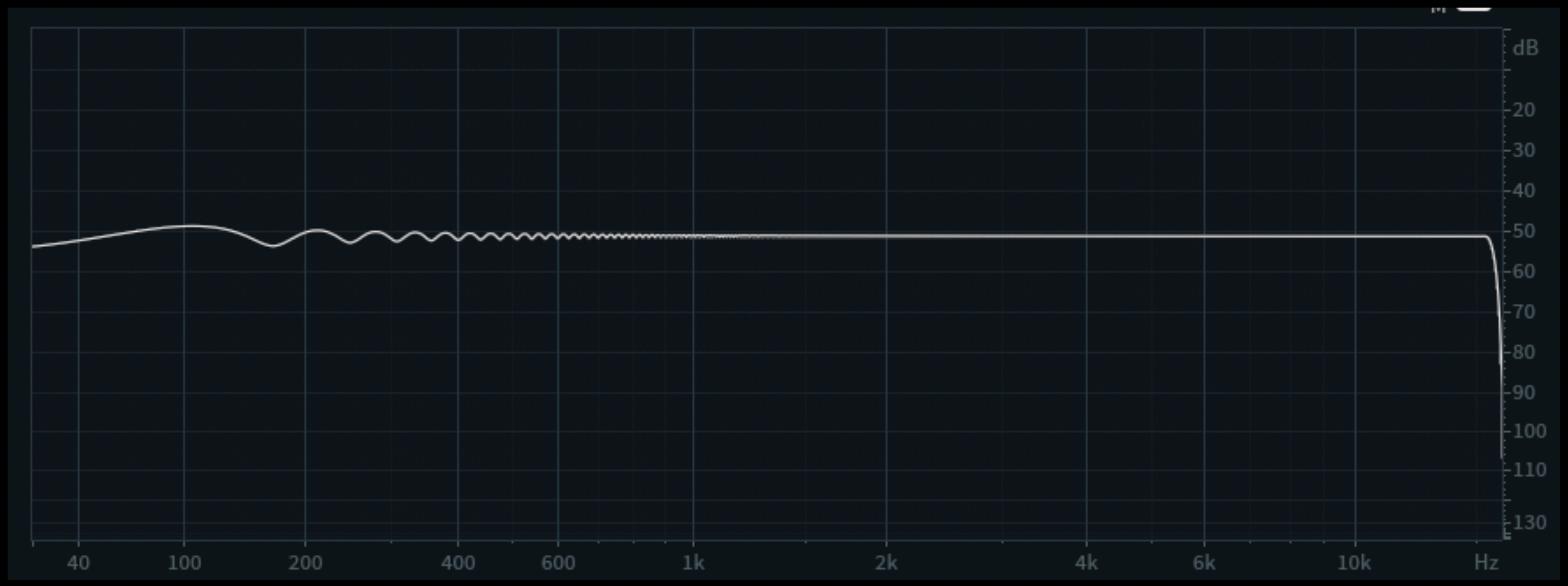
What is Additive Synthesis?

Additive Synthesis

- Model our sound as the sum of sine waves
- Apply envelopes to the pitch and amplitude of each sine over the course of the note
- It is common for oscillator pitches to be fixed and to form the harmonic series of a played note but this is not necessary

Additive Synthesis





- This is **not** fourier analysis - we can however use fourier analysis to build our additive model
- Consider a swept sine wave. In additive terms, it's a single sine oscillator changing in pitch. In fourier analysis, it's flat in frequency response

Additive Synthesis

- Hammond Organ (1935-)
- Uses tone wheels to generate harmonics
- sliders allow the harmonics to be configured for each manual
- 9 harmonics per note

Additive Synthesis

- Kawai K5000 (1996)
- Combined PCM (Sample) and Additive oscillators, 64 harmonics, Formant Filter, Morphing, with a ghastly digital filter implementation

Additive Synthesis

vintagesynth.com

Kai: ‘I sold mine, horrible machine to program.’

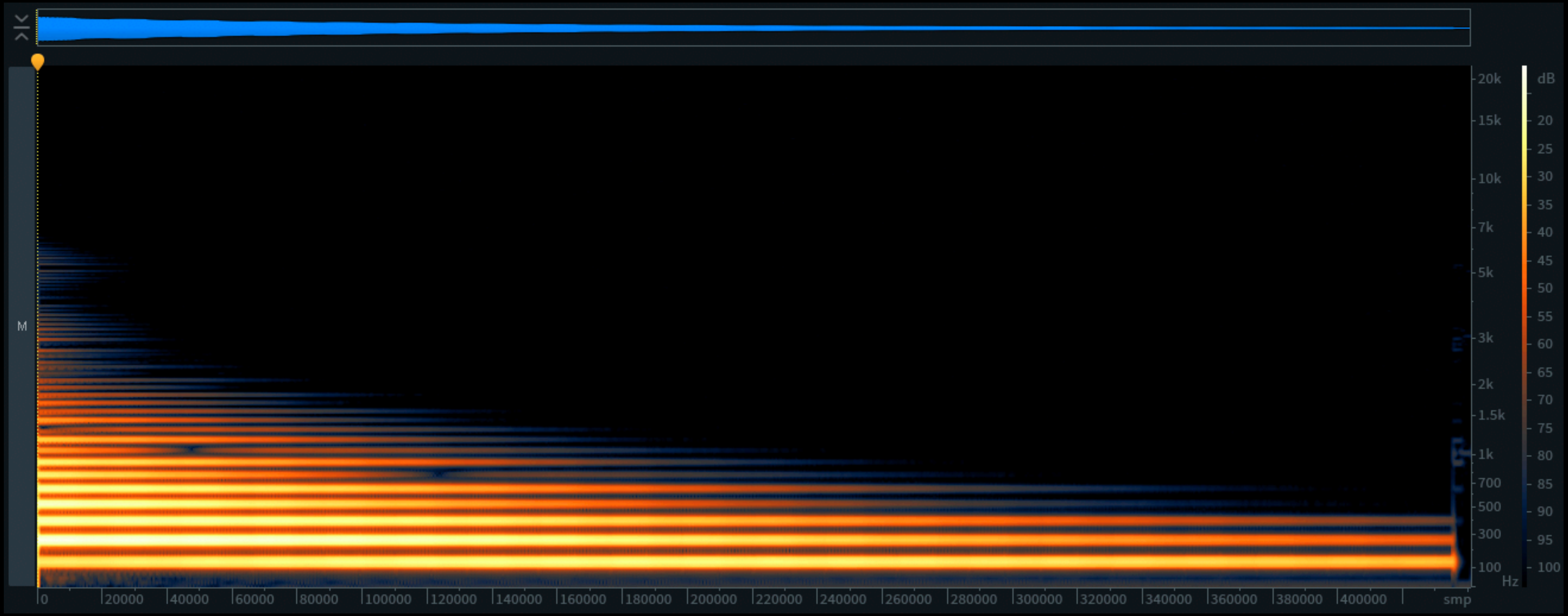
Dmitry: ‘Simply. The. Best. Synthesizer. Ever.’

Additive Synthesis

Modern plugins:

- NI's Razor
- Apple's Alchemy

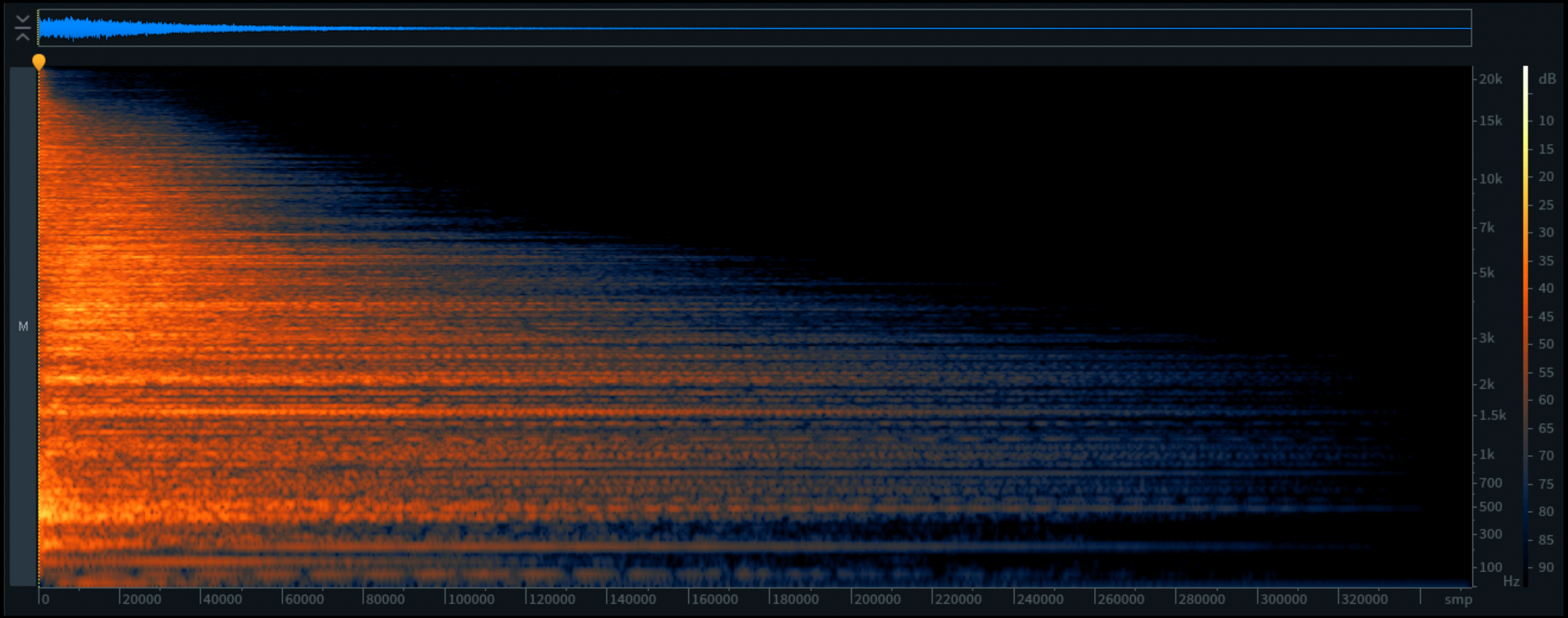
Additive Synthesis



Spectrogram of Rhodes Piano sample

Strong horizontal lines occurring at overtones of the fundamental

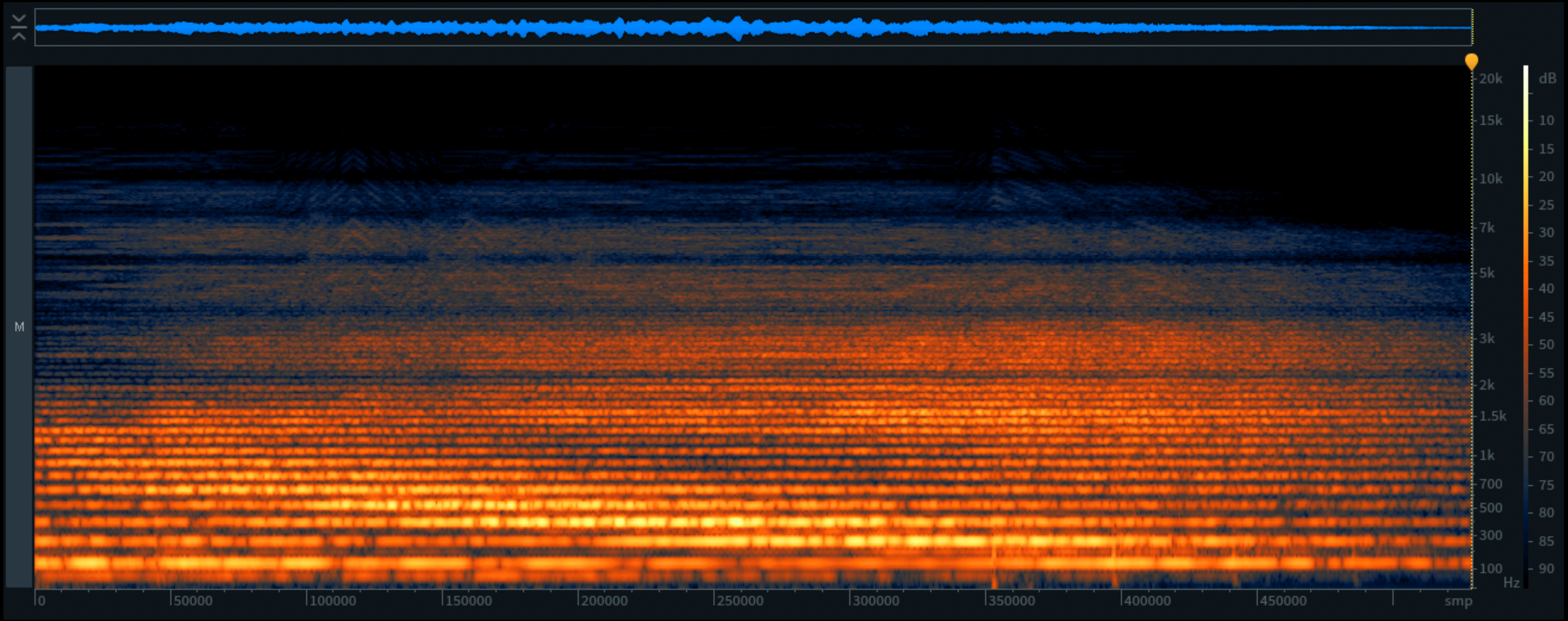
Additive Synthesis



Spectrogram of Crash Cymbal

No harmonic structure - horizontal lines do not form a harmonic series

Additive Synthesis

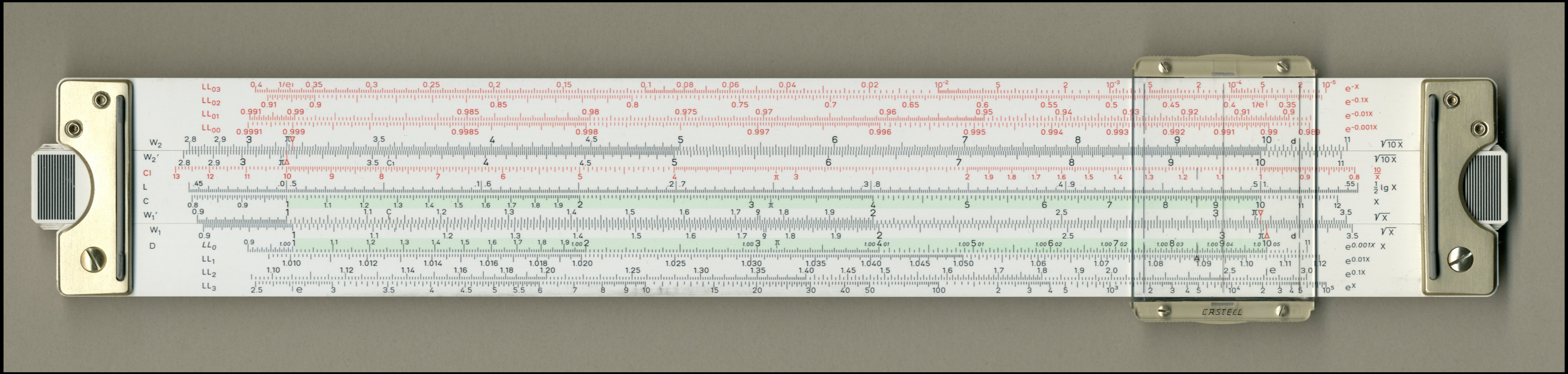


Spectrogram of Equator Patch

A mixture of strong harmonic lines and additional ‘noise’

The CORDIC algorithm

How did the sin() button on calculators work?



Early calculators had a very simple processor supporting integer arithmetic, and limited memory

Operations like add and multiply were performed digit by digit in decimal

How were the trigonometric functions implemented
using add and multiply?

Complex numbers

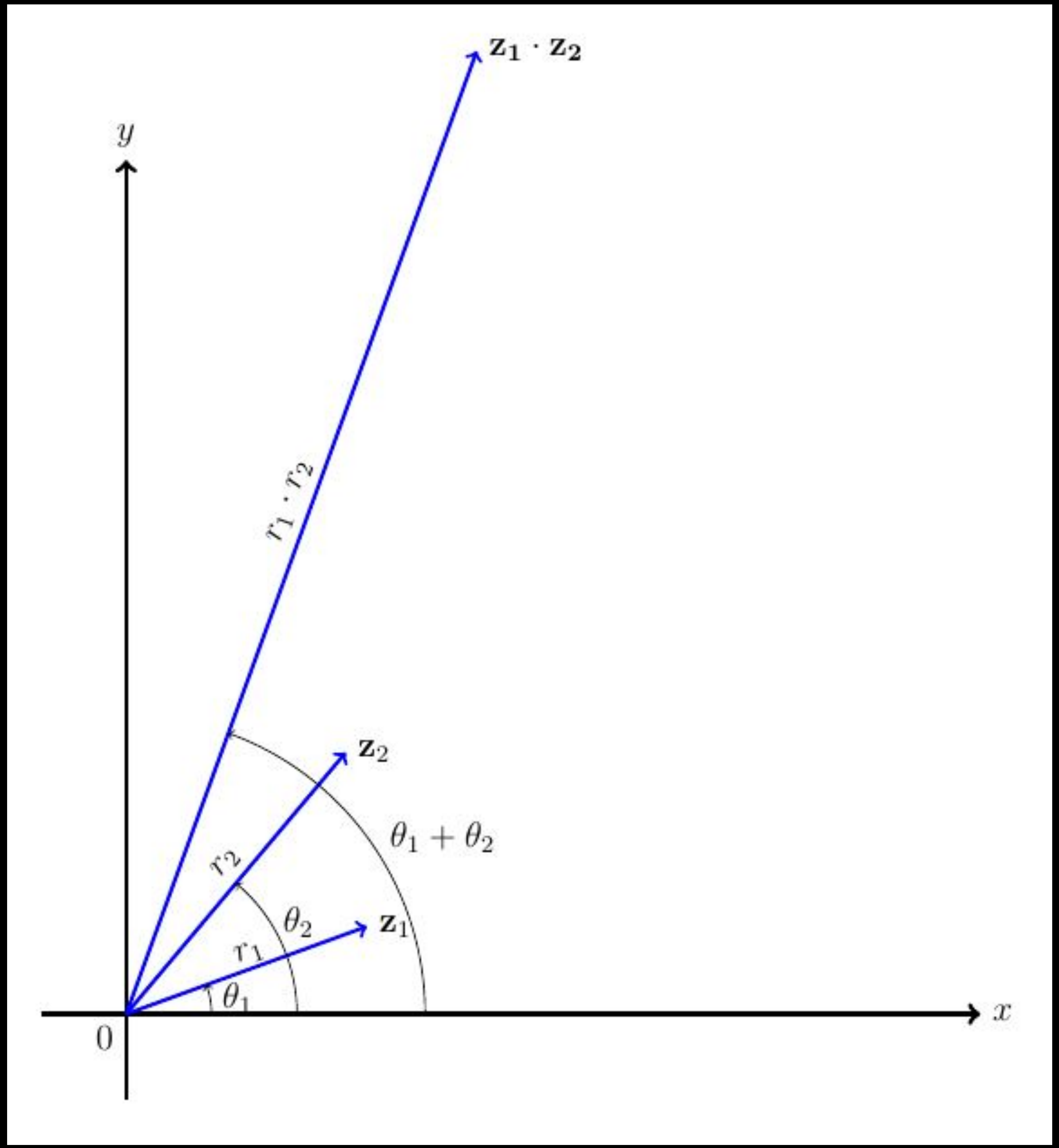
- Complex numbers are of the form a + bi, with the property that $i^2 = -1$

- Multiplying two complex numbers gives:

  $(a + bi) * (c + di) = ac + adi + bci + bdi^2$

  $= (ac - bd) + (ad + bc)i$

- If expressed in polar coordinates, this is equivalent to multiplying the magnitudes, and adding the angles

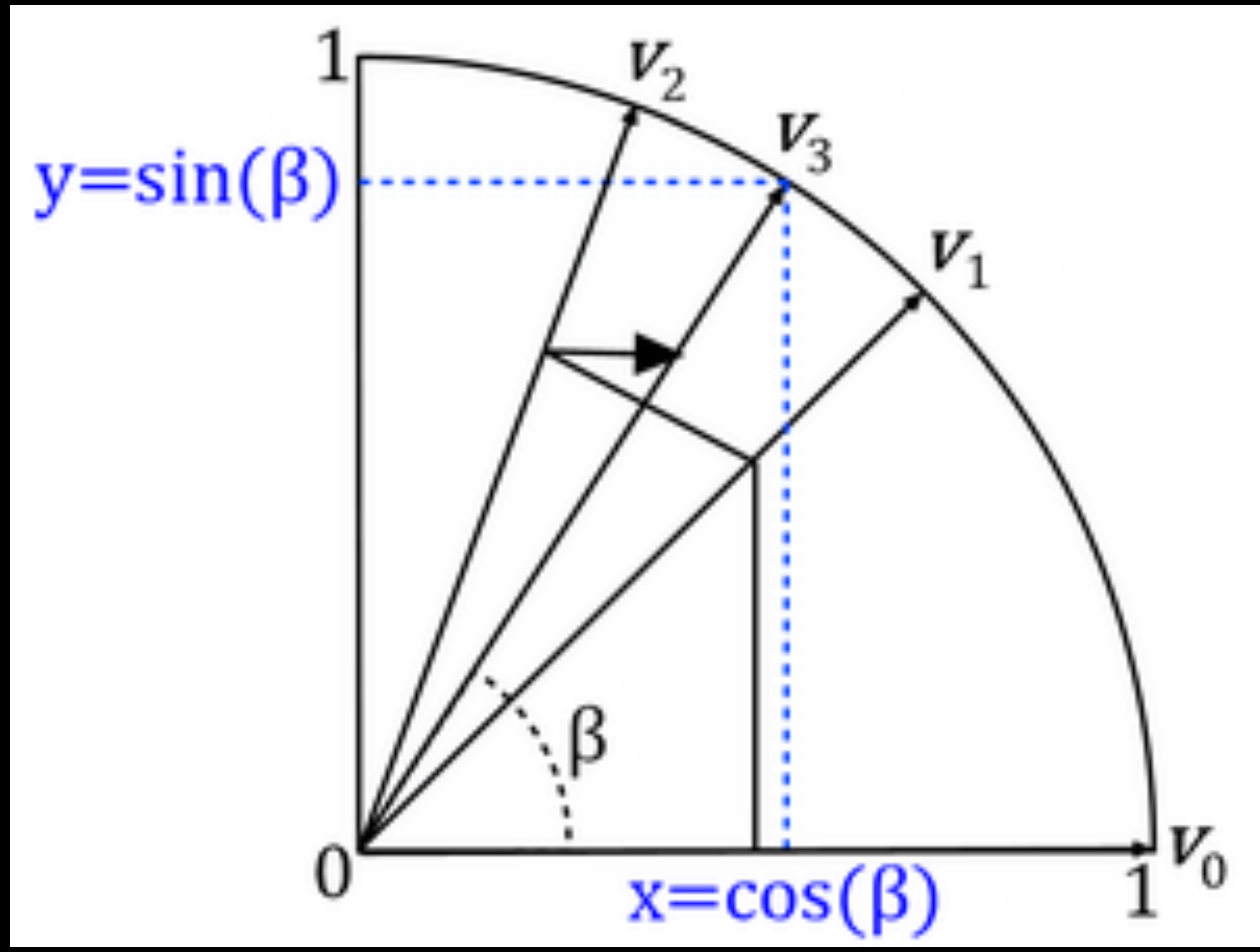


Complex numbers

- Imagine multiplying a complex number by the unit vector cos(ϑ) + sin(ϑ)i
- This vector has a magnitude of 1, and angle of ϑ. The result is rotated by ϑ, but it's magnitude is unchanged.
- Repeated multiplication by this vector causes the result to rotate by the angle a each time

CORDIC algorithm (Volder's algorithm)

- The CORDIC algorithms explore this analysis, and produce efficient ways of breaking down functions like sin(ϑ) to being a series of operations on known values
- Rather than repeatedly applying the same rotation, the algorithm applies either clockwise or counterclockwise rotations with angles following a power series to converge on the result. The values to apply are taken from a lookup table
- By always applying the same rotations (clockwise or counterclockwise) the exact error can be pre-calculated for a given number of operations, so the required number of stages to reach a given precision can be calculated
- By using rotation vectors of non-unit length (noticing that cos(ϑ) approaches 1 for small ϑ) the operations are further reduced, at the cost of adding a single multiply at the end to correct the vector length



The CORDIC Computing Technique (1959)
http://home.citycable.ch/pierrefleur/Jacques-Laporte/Volder_CORDIC.pdf

HP 9100A (1968)

Sinclair scientific calculator (1974)

- Sinclair had only 320 words of memory for the entire calculator logic. Instead of using CORDIC, the sin() function used repeated rotation by 0.001 radians using the operation:

  C = C - S/1000
  S = S + C/1000

- So to work out sin(0.5) the above would be applied 500 times. It wasn't fast, but it got the job done (to 3 digits of precision)

- http://files.righto.com/calculator/sinclair_scientific_simulator.html

So what?

So what?

- For additive synthesis, we need to calculate the sin() function a lot (say, 64 sine oscillators per voice, 32 voices, 48k samples per second = 98 million per second)
- The sin() function is accurate, but it isn’t fast
- The majority of the time, we are not calculating arbitrary sin() values, but the next phase increment from a previous sin() value.
- But since complex multiplication can generate rotation with carefully chosen values, we can generate the series of sine values we need for a sine oscillator just by multiplying a unit vector by an appropriate rotation value
- Complex multiplication requires 4 multiplies and 2 adds, and there are no conditional code paths. Fused multiply/add is heavily optimised in modern FPUs

Implementing the algorithm

Calculating a sin wave with CORDIC

```
processor Sine
{
    output stream float32 out;
    input event float32 frequency;

    event frequency (float32 noteFrequency)
    {
        phaseIncrement = float (noteFrequency * twoPi * processor.period);
    }

    float phase, phaseIncrement;



    // ------------------------
    void run()
    {
        loop
        {
            phase += phaseIncrement;

            if (phase > twoPi)
                phase -= float (twoPi);

            out << sin (phase);
            advance();
        }
    }
}
```

```
processor CordicSine
{
    output stream float32 out;
    input event float32 frequency;

    event frequency (float32 noteFrequency)
    {
        let phaseIncrement = float (noteFrequency * twoPi * processor.period);

        multiplier.real = cos (phaseIncrement);
        multiplier.imag = sin (phaseIncrement);
    }

    complex value = 1, multiplier = 1;


    // ------------------------
    void run()
    {
        loop
        {
            value = value * multiplier;
            out << value.imag;
            advance();
        }
    }
}
```

Demo

Calculating a sin wave with CORDIC

- The CORDIC implementation is stable, doesn’t gain or loose amplitude and is stable in pitch
- It diverges from the equivalent sin() implementation in phase, but very slowly
- The cause of the divergence is the limitations of float32 accuracy - it’s not clear which is moving away from which, but it’s accurate enough for synthesis purposes
- I guess calling it CORDIC is a stretch, and maybe ‘vector rotation’ would be a more accurate description, but i wouldn’t have had the chance to mention the cool work by Jack Volder

Calculating a bank of sin waves with CORDIC

```
processor CordicSine
{
    output stream float32 out;
    input event float32 frequency;

    event frequency (float32 noteFrequency)
    {
        let phaseIncrement = float (noteFrequency * twoPi * processor.period);

        multiplier.real = cos (phaseIncrement);
        multiplier.imag = sin (phaseIncrement);
    }

    complex value = 1, multiplier = 1;

    // -------------------------
    void run()
    {
        loop
        {
            value = value * multiplier;
            out << value.imag;
            advance();
        }
    }
}
```

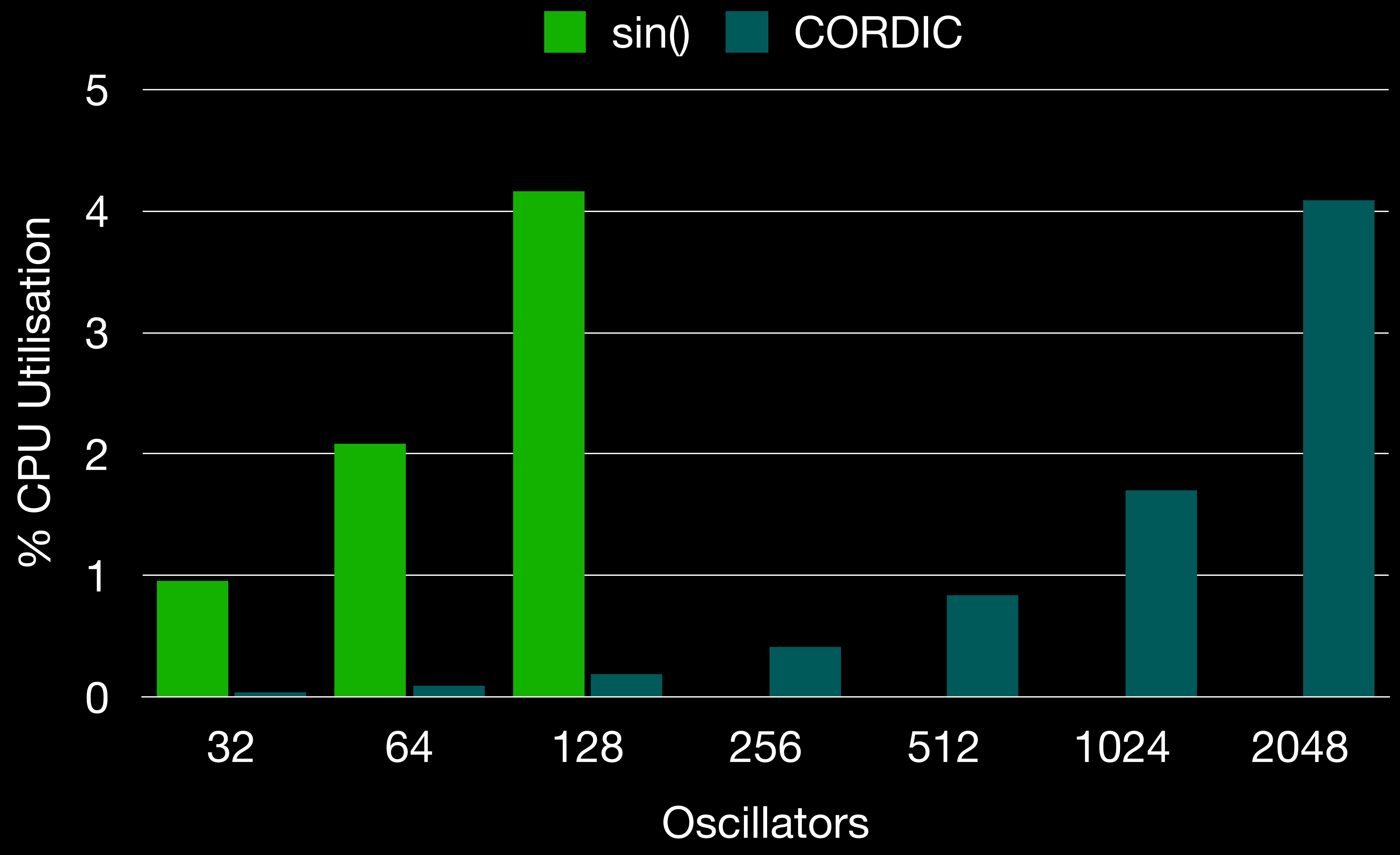
```
processor SawOsc [[ main ]]
{
    output stream float32 out;
    input event float32 frequency [[ name: "Frequency", min: 20, max: 10000, init: 500 ]];

    let harmonics = 64;

    event frequency (float32 noteFrequency)
    {
        amplitudes = 0;

        for (wrap<harmonics> i)
        {
            let harmonicFrequency = noteFrequency * float (i+1);
            let phaseIncrement = float (harmonicFrequency * twoPi * processor.period);

            multiplier[i].real = cos (phaseIncrement);
            multiplier[i].imag = sin (phaseIncrement);

            if (harmonicFrequency < (processor.frequency /2))
                amplitudes[i] = 1.0f / float(i+1);
        }
    }

    complex<harmonics> value = 1, multiplier = 1;
    float<harmonics> amplitudes;

    // -------------------------
    void run()
    {
        loop
        {
            value = value * multiplier;
            out << sum (value.imag * amplitudes);
            advance();
        }
    }
}
```

Demo

Calculating a bank of sin waves with CORDIC

- Scaling to multiple harmonics is simple - it’s just a vector of complex numbers.
- Additional cost of summing across the harmonic values, and applying scaling factors which again vectorises well
- Anti-aliasing is trivial!
- Easy to implement different waveform shapes - trades off CPU use vs memory use (compared to say, wavetable synthesis)

Performance figures



The CORDIC algorithm is around 16-20 times faster than the sin() algorithm
(intel i7 with AVX)

Building an instrument

Build a subtractive synth

We could use the sine bank oscillator directly in a subtractive synth architecture

Pros

- Free control of the harmonics present in the timbre, supporting user configurable harmonic content
- Reasonably efficient to implement - heavier use of compute vs more memory intense approaches (e.g. wavetable)
- Alias free by design
- Interesting tone shaping options hard to achieve otherwise (see later), and hence could be a useful different waveform flavour to add to an existing design

Cons

- Why bother? MinBLEP rocks!
- High harmonic count needed for low notes
- Hard to implement common subtractive effects e.g sync, pulse width modulation

Build a pure additive synth

Let’s go for it, additive all the way!

Pros

- Total freedom to program envelopes for both pitch and amplitude for hundreds of harmonics
- Ability to model unique sounds not achievable any other way
- Very conducive to mapping performance parameters to create an organic feel

Cons

- OMG, how do you program this thing?

Build a hybrid synth

Ok, so maybe there’s a middle ground

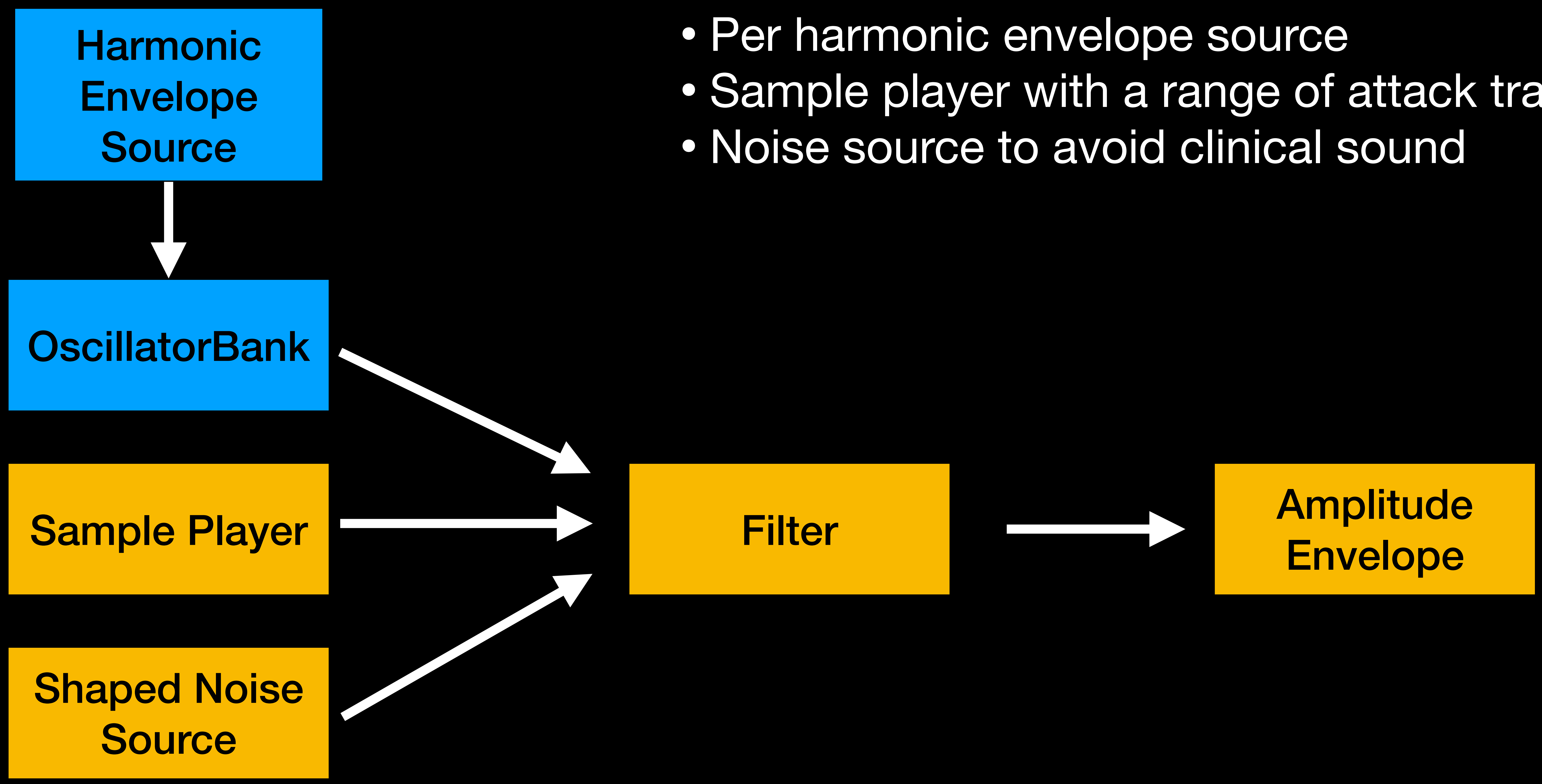
Pros

- Keep the synth architecture familiar, so oscillator banks, filters, amplitude envelopes etc
- Incorporate standard filter designs rather than relying solely on harmonic manipulation
- Provide tooling to import and generate harmonic content from samples, and graphical representations to massage the generated envelopes (e.g a spectrogram view and ‘dodge/burn’ manipulation tools)
- Use other generators to reinforce the sound, e.g sample playback for attack transients, noise sources etc

Cons

- Will upset purists

Build a hybrid synth



- Oscillator bank with fixed harmonic series
- Per harmonic envelope source
- Sample player with a range of attack transients
- Noise source to avoid clinical sound

Envelope Source

```
processor AmplitudeSource (int size = 64, int stepRate = 1024)
{
    input event soul::note_events::NoteOn noteOn;
    output stream float<size> out;

    float<size>[] amplitudes = ( … );

    event noteOn (soul::note_events::NoteOn e)
    {
        value = amplitudes[0];
        increment = (amplitudes[1] - amplitudes[0]) / stepRate;
        nextSlot = 1;
        steps = stepRate;
    }

    void next()
    {
        increment = 0;
        steps = stepRate;

        if (nextSlot < amplitudes.size - 1)
        {
            value = amplitudes.at (nextSlot);
            increment = (amplitudes.at (nextSlot + 1) - amplitudes.at (nextSlot)) / stepRate;
            nextSlot++;
            steps = stepRate;
        }
    }

    float<size> value, increment;
    int nextSlot, steps;
```

```
    void run()
    {
        loop
        {
            out << value;
            value += increment;
            steps--;
            if (steps == 0) next();
            advance();
        }
    }
}
```

Generating Harmonic Envelopes

Generating harmonic envelopes

- Analyse a sample, determining the fundamental frequency of the tone
- Generate envelopes for each harmonic at different positions within the sample
- Build a table of envelope values for each harmonic
- Use correlation and a suitable window to reduce artefacts
- Use a limited dynamic range to avoid sample noise being interpreted as harmonic energy

Generating harmonic envelopes

For each harmonic frequency

For each time slice

Build a windowed slice of the sample around the time point

Correlate the sample slice with the harmonic frequency

The harmonic amplitude is the correlation value at this time point

Generating harmonic envelopes

- We analyse the sample in blocks of framesPerSample
- We analyse the sample every framesBetweenSamples
- Assuming the amplitude is above a threshold, we use this (otherwise use 0)

```
std::vector<float> getEnvelope (float frequency)
{
    std::vector<float> result;

    int offset = 0;

    float phaseIncrement = frequency / sampleRate;
    const float twoPi = 3.14159265f * 2.0f;

    while (offset < sample.getNumFrames())
    {
        float phase = 0.0f;
        float real = 0.0f, imag = 0.0f;

        // Correlate the signal with our frequency
        for (int i = 0; i < framesPerSample; i++)
        {
            float s = window[i] * sample.getSampleIfInRange (0, offset + i);
            real += s * sin (phase * twoPi);
            imag += s * cos (phase * twoPi);

            phase += phaseIncrement;

            if (phase > 1.0f)
                phase -= 1.0f;
        }

        float v = sqrt ((real * real) + (imag * imag)) / framesPerSample;

        v = v * 4.0f;   // Correct amplitude

        if (v < threshhold)
            v = 0.0f;

        result.push_back (v);

        offset += framesBetweenSamples;
    }

    return result;
}
```

Demo
Putting it all together

Creative opportunities

Creative effects

There are various interesting effects which can be applied to an additive model that are hard to do with other techniques:

Odd/Even harmonic balance

By applying a scaling factor to either the even or odd harmonics, the relative amplitude of these harmonics can be changed, leading to a change in tone, from a nasal character to a hollow tone. It's similar to the change you hear moving from Saw to Square waveforms

Randomising effects

You can apply random modifications to the amplitudes, frequencies, or initial phase to generate variation when repeating notes, to avoid artificially similar sounds

Morphing

By transitioning between two or more envelopes, interesting morphing and layering effects can be created. Unlike sample based systems, there are no phase related cancellations

Creative effects

Rotating envelopes

Who says that harmonic envelopes need to be applied to the original harmonic number? By mixing things up, weird and unique sounds can be achieved

Slow down/speed up envelopes

We can move through the envelopes at a different rate than they were captured. Slow evolving sounds emerge from otherwise recognisable envelopes.

Questions?