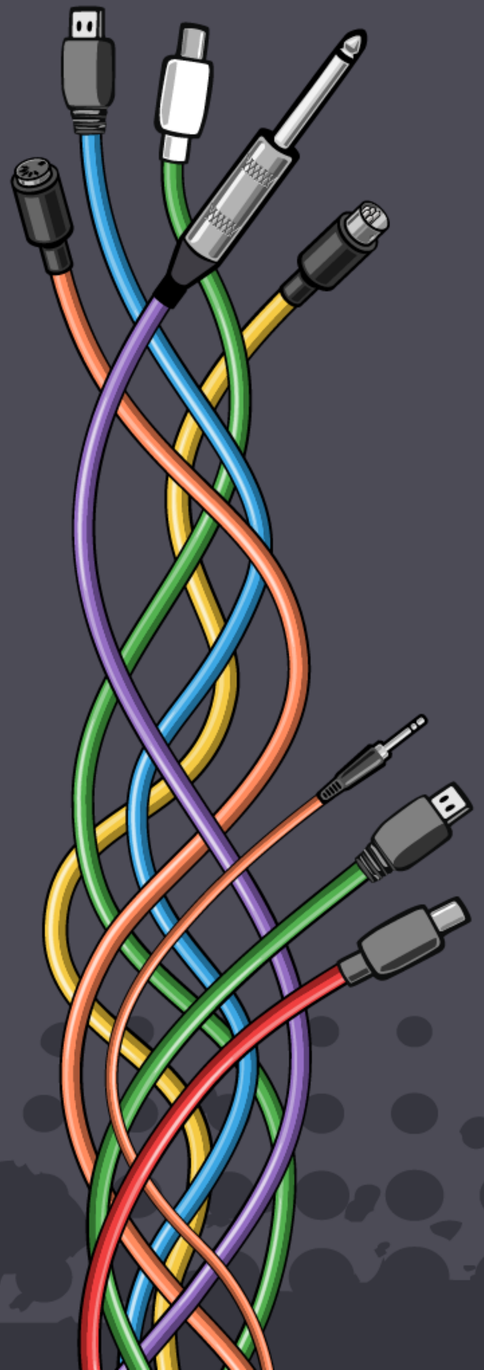
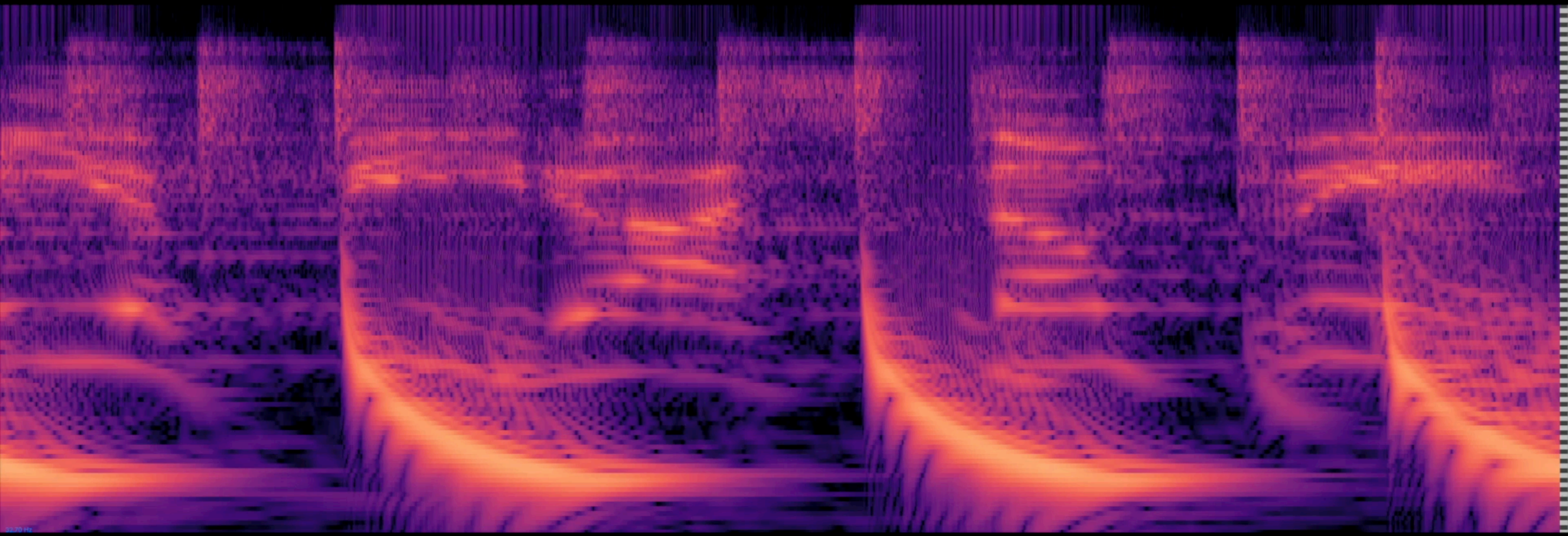


REAL-TIME, LOW LATENCY AND HIGH TEMPORAL RESOLUTION SPECTROGRAMS

ALEXANDRE R.J. FRANCOIS

ADC25 Bristol



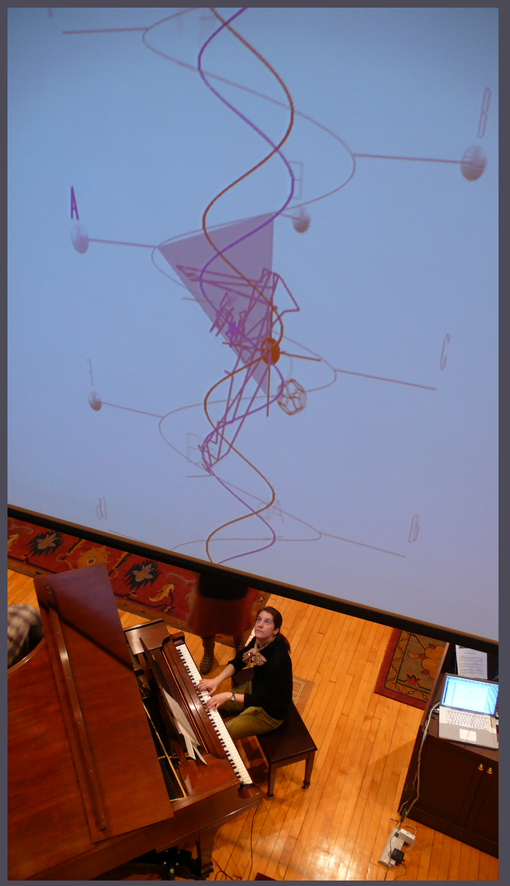
Pump Up The Jam -Technotronic

Alexandre R.J. François

- Computer scientist, software engineer, etc.
- Computer vision, interactive systems, computer graphics, **music computing**
- Academia, startups, larger companies
- Interactive systems, design, user experience (with computers)
  - For **brains**

MIMI (2007+) with E. Chew

MuSA.RT (2003+) with E. Chew



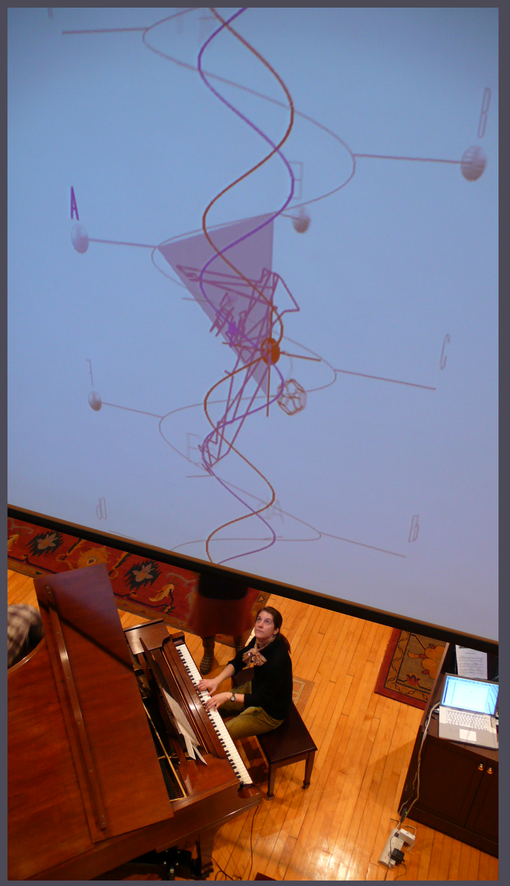
www.alexandrefrancois.org

Context and Motivation

Which/how much of the frequencies of interest are present in the input signal?

- Input to interactive music analysis systems for visualization, improvisation, etc.
  - MIDI
- Perceptually relevant
- Low latency
- Computationally efficient

MuSA.RT (2003+) with E. Chew



MIMI (2007+) with E. Chew

The Fast Fourier Transform (FFT)

But… is it really the best solution?

- Fast, but… buffering imposes:
  - Latency
  - Linearly distributed frequency bins
  - Time / frequency resolution trade-off
- Mitigation techniques can be computationally expensive
  - Sliding window
  - Constant-Q transform (log frequency scale)

Buffer size (samples)	Duration (ms)
1	0.023
8	0.181
16	0.363
32	0.726
64	1.451
128	2.903
256	5.805
512	11.610
1024	23.220
2048	46.440
4096	92.880

Sampling rate: 44.1 kHz

A Different Approach: Resonate

- Simple resonator model, assembled into banks
  - Efficient iterative formulation and implementation, no buffering
- Analogous to a bank of non-linear filters
- Similar/related/identical? to Goertzel algorithm, SWIFT
- Not a better FFT - not trying to compute the DFT
- Not aware of this approach being used in a practical way for music/audio

- Reference publication:
  Alexandre R.J. François, “Resonate: Efficient Low Latency Spectral Analysis of Audio Signals,” in Proceedings of the 50th Anniversary of the International Computer Music Conference 2025, pp. 251-258, Boston, MA, USA, 8-14 June 2025.

Resonate Properties

- Direct computation of spectrum with arbitrary frequency distribution
- High temporal resolution and low latency: one measurement per input sample
- Resonators independently tuned: frequency and dynamics
- Works across whole frequency range with same efficiency
  - Including low frequencies
- …and the resulting data look reasonable…

This Talk

- Goal: share what I have done, no matter how imperfect, so people can use it, improve on it, dream up and build applications aligned with their own interests…

- This is a personal project rooted in my past academic work, developed on week-ends, holidays and other breaks, over a number of years
  - Related open source material online since 2022
  - Does not represent the views of past or present employers

Resources

- *noFFT* open source module: python and C++ implementations, Jupyter notebooks with code used to generate the figures in the paper (and more…)
- Oscillators open source Swift package: reference implementations in Swift and C++
- **Oscillators app for iOS/iPadOS/MacOS (Apple Silicon, designed for iPad)**
- Resonate Youtube playlist

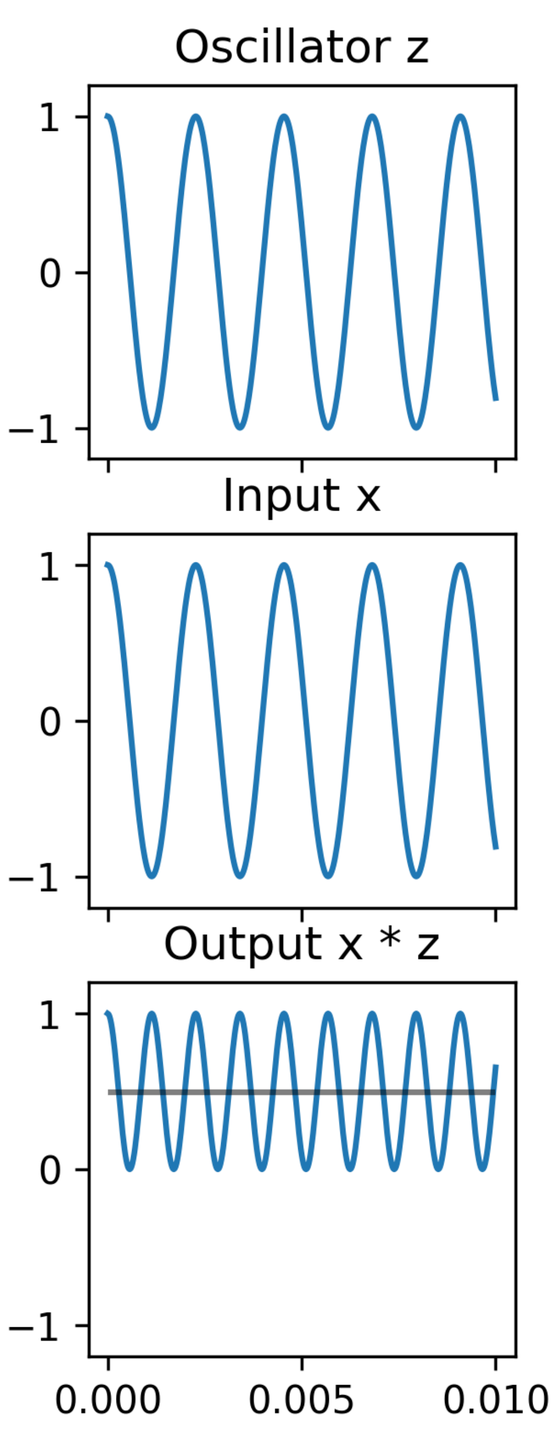
www.alexandrefrancois.org/Resonate

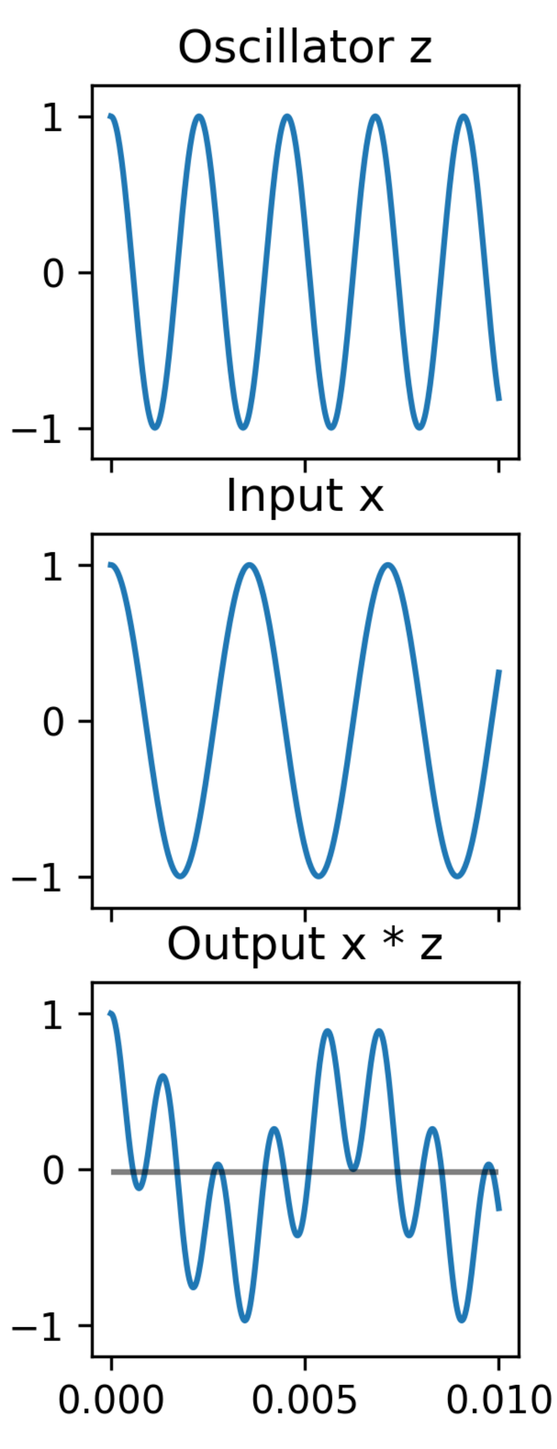
Outline

- Intro
- **Single resonator**
- Resonator banks
- Apps and applications

Resonator

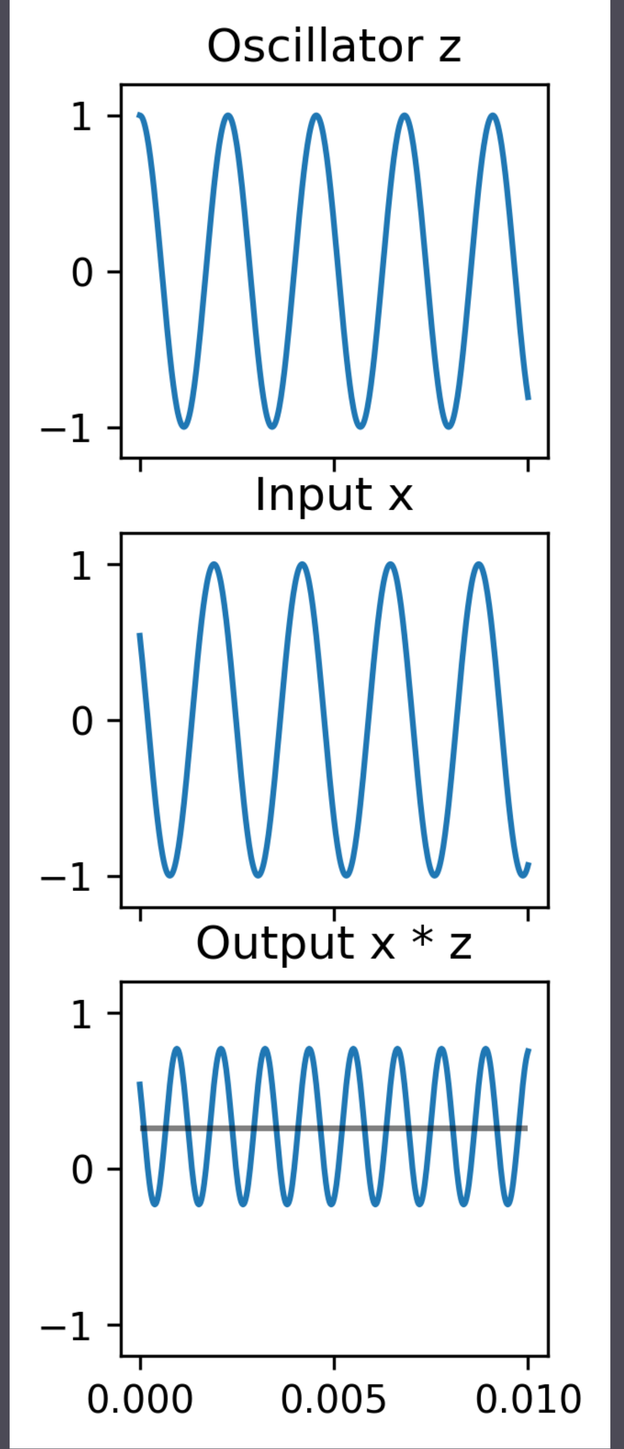
- A resonator is a device or system that exhibits resonance, meaning it oscillates with greater amplitude at certain frequencies than others
- Simple example:
  - Oscillator with sine wave z at known frequency (amplitude 1)
  - Input sine wave x (amplitude 1)
  - Output: product of input and oscillator, averaged over time
  - Intuitively, output will be maximal when input and resonator have same frequency… and phase

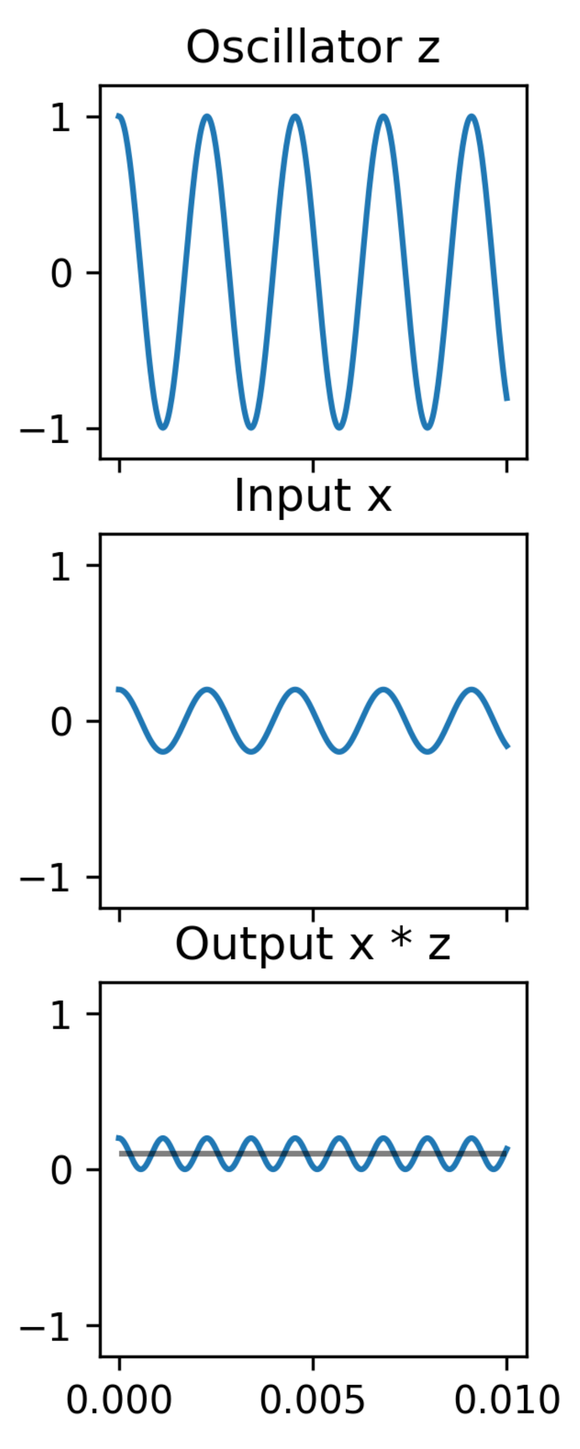




Phase and Amplitude

- Given resonant frequency
- Input signal of same frequency
- Two parameters to compute:
  - Phase
  - Amplitude





Phase and Amplitude

- 2 degrees of freedom
- Need 2 measurements per input sample
- Use cos and sin
  - cos^2(x) + sin^2(x) = 1

```
// input signal X

// oscillator waveforms Zc (cosine)
// and Zs (sine)

// for each input sample x
c = x * Zc
s = x * Zs

power = c*c + s*s
amplitude = sqrt(c*c + s*s)

phase = atan2(s,c)
```

Phasor

Iterative implementation of sine oscillator

```
class Phasor {
  var frequency: Float
  var sampleRate: Float

  // Phasor variables
  // Phasor: Z = Zc + j Zs
  // Multiplier: W = Wc + j Ws

  var Zc : Float = 1.0
  var Zs : Float = 0.0

  var Wc : Float = 0.0
  var Ws : Float = 0.0
  // pre-computed Wc + Ws
  var Wcps : Float = 0.0

  […]
}
```

```
// pre-compute Wc + Ws
let omega = twoPi * frequency / sampleRate
Wc = cos(omega)
Ws = sin(omega)
Wcps = Wc + Ws
```

```
// Compute next value of the phasor
// Z <- Z * W
// complex multiplication
// with 3 real multiplications

let ac = Wc*Zc
let bd = Ws*Zs
let abcd = Wcps * (Zc+Zs)
Zc = ac - bd
Zs = abcd - ac - bd
```

Phasor

Iterative implementation of sine oscillator

```
class Phasor {
  var frequency: Float
  var sampleRate: Float

  // Phasor variables
  // Phasor: Z = Zc + j Zs
  // Multiplier: W = Wc + j Ws

  var Zc : Float = 1.0
  var Zs : Float = 0.0

  var Wc : Float = 0.0
  var Ws : Float = 0.0
  // pre-computed Wc + Ws
  var Wcps : Float = 0.0

  […]
}
```

```
// Apply re-normalization correction
// to compensate for numerical drift
// Use Taylor expansion of 1/sqrt(x)
// around 1 to reduce computational cost

// This can be applied every n? samples

let k = (3.0 - Zc*Zc - Zs*Zs) / 2.0
Zc *= k
Zs *= k
```

Exponentially Weighted Moving Average

aka Low-Pass Filter

```
// input signal X

// smoothed signal xx
// update at each sample x

xx = (1-alpha) * xx + alpha * x


// parameter alpha relates to time constant:
// time for system response to a unit step function
// to reach ~62% of the original signal

sampleDuration = 1.0 / sampleRate

tau = -sampleDuration / log(1.0 - alpha)
```

Resonator

EWMA

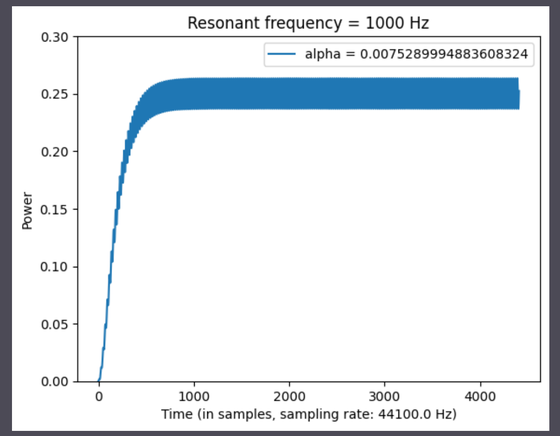
```
class Resonator : Phasor {
  // complex: r = c + j s
  var c: Float = 0.0
  var s: Float = 0.0

  […]
}
```

```
// Update with sample x


c = (1-alpha) * c + alpha * x * Zc
s = (1-alpha) * s + alpha * x * Zs
```

Resonator

EWMA

```
class Resonator : Phasor {
  // complex: r = c + j s
  var c: Float = 0.0
  var s: Float = 0.0

  […]
}
```




```
// Update with sample
alphaSample = alpha * sample
omAlpha = 1.0 - alpha
c = omAlpha * c + alphaSample * Zc
s = omAlpha * s + alphaSample * Zs
```

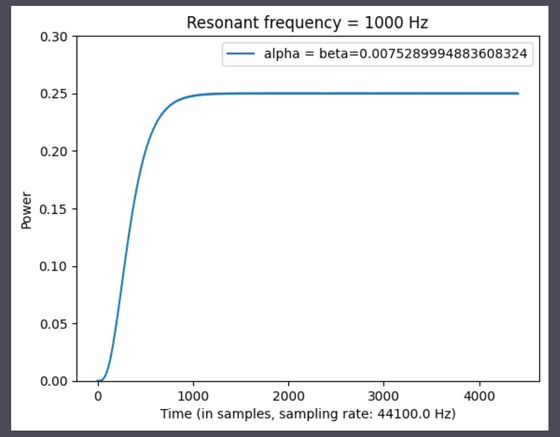
```
// Compute next value of the phasor
// Z <- Z * W
// complex multiplication
// with 3 real multiplications

ac = Wc*Zc
bd = Ws*Zs
abcd = (Wcps) * (Zc+Zs)
Zc = ac - bd
Zs = abcd - ac - bd
```

Resonator

Smooth out output oscillations

```
class Resonator : Phasor {
  // complex: r = c + j s
  var c: Float = 0.0
  var s: Float = 0.0

  // Smoothed resonator output
  var cc: Float = 0.0
  var ss: Float = 0.0

  […]
}
```

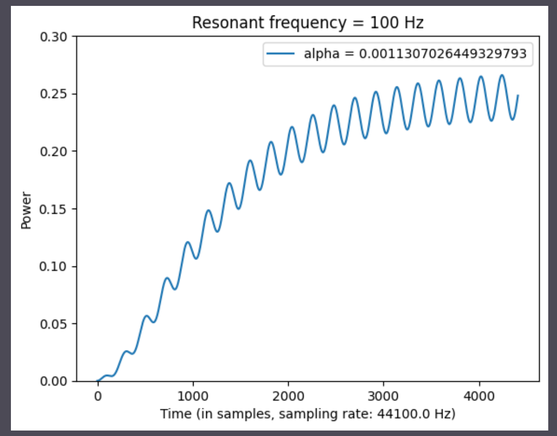
```
// Update with sample
alphaSample = alpha * sample
omAlpha = 1.0 - alpha
c = omAlpha * c + alphaSample * Zc
s = omAlpha * s + alphaSample * Zs


omBeta = 1.0 - beta
cc = omBeta * cc + beta * c
ss = omBeta * ss + beta * s
```

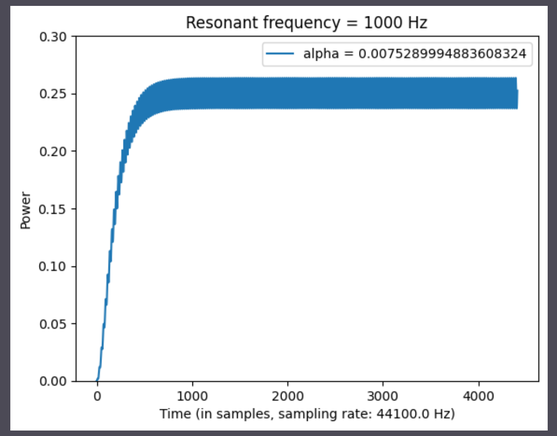


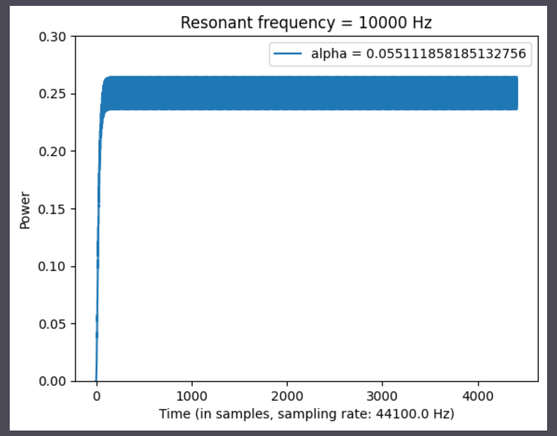
Parameter: Time Constant

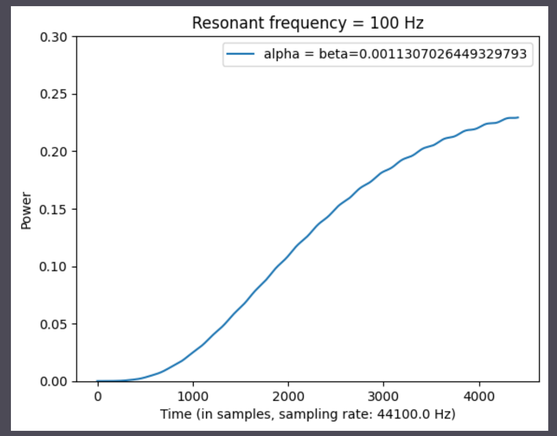
- Single parameter alpha, set heuristically
- Relate to resonator frequency
- Intuitively, lower frequencies require more time to converge
  - Time constant inversely proportional to frequency

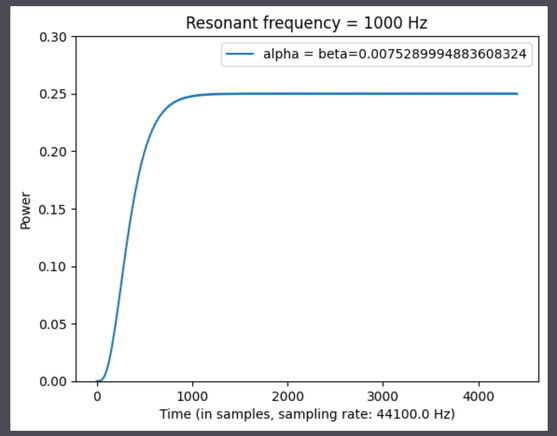
```
tau = k * log10(1+frequency) / frequency

// parameter alpha heuristic
alpha = 1 - exp(-frequency / (sampleRate * k * log10(1+frequency)))
```

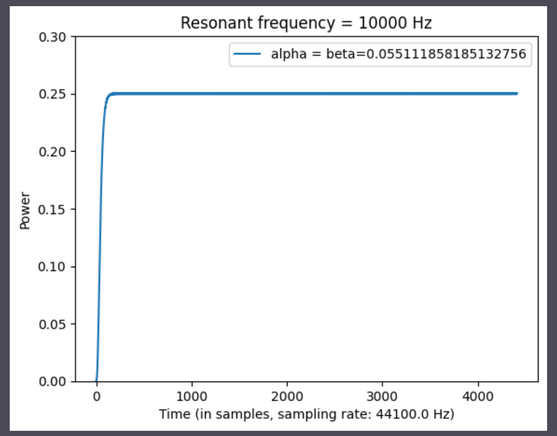


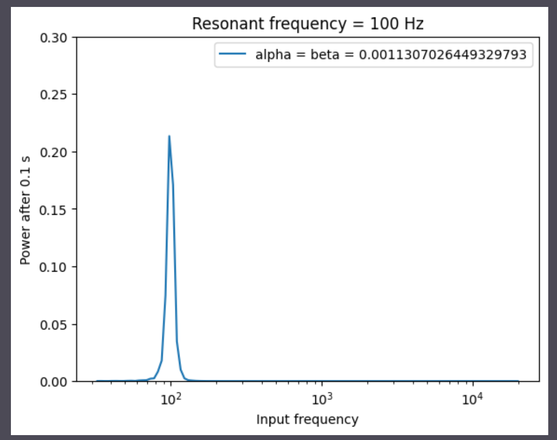


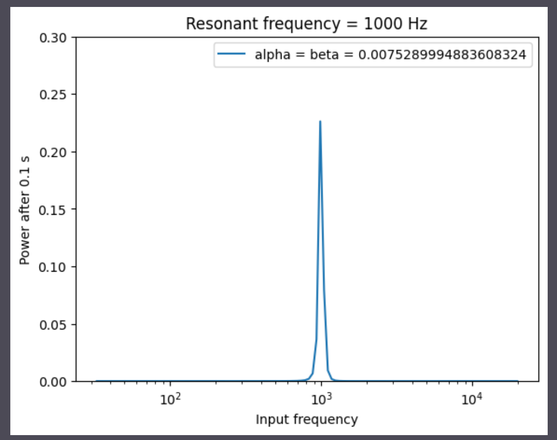


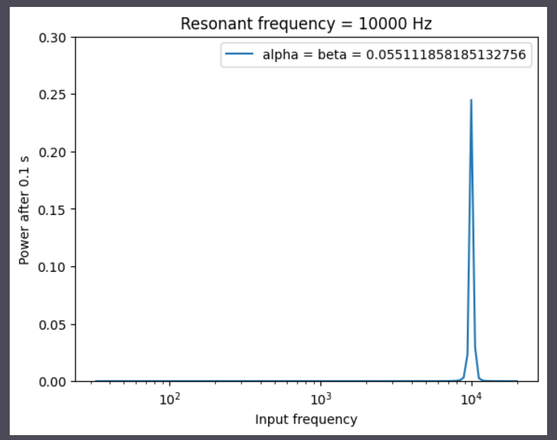










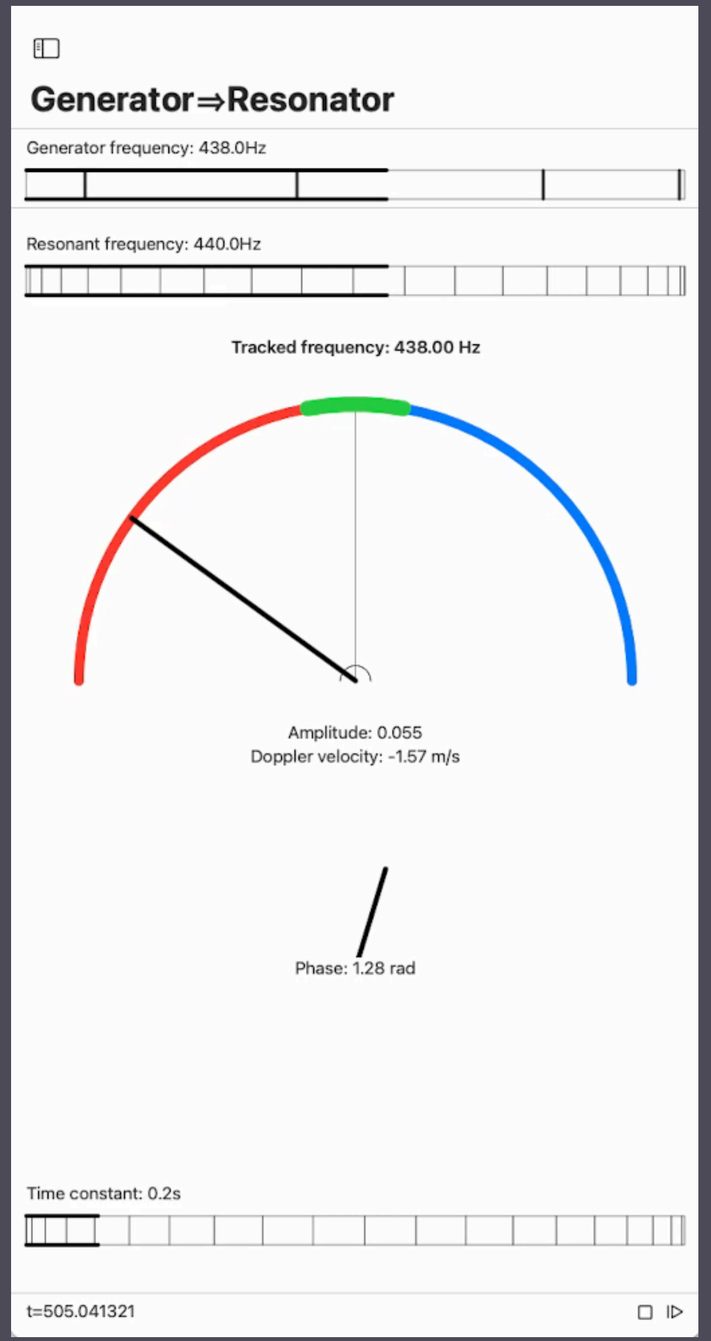


Computational Cost

- Memory: 6 Floats
- Computations **per sample** update:
  - a handful of multiplications and additions
  - Some occasional overhead to compensate for numerical approximations
  - Complexity is constant
- Overall complexity is linear in the number of samples processed

Some Fun Applications

- Frequency tracking
  - From phase drift (aperture effect)
- Doppler velocity
  - From measured frequency shift
- Stereo directional estimation
  - From stereo phase difference

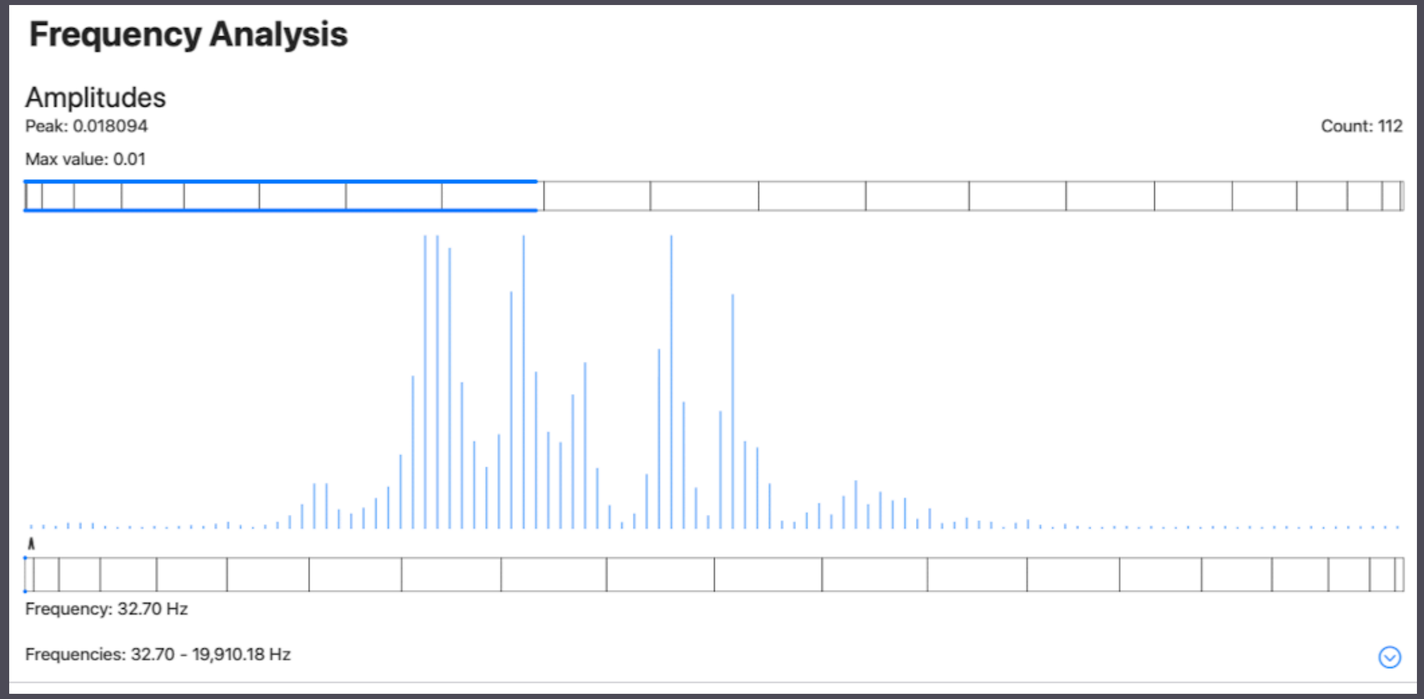


Outline

- Intro
- Single resonator
- **Resonator banks**
- Apps and applications

Resonator Banks

- Arbitrary number of resonators
- Directly tuned at frequencies of interest, arbitrarily distributed
  - Log frequency scale (like CQT), mel scale, etc.
  - Dynamics
- Resonators are independent
  - Parallel processing of each sample
- Analogous to bank of non-linear band pass filters



Resonator Banks

- Arbitrary number of resonators
- Directly tuned at frequencies of interest, arbitrarily distributed
  - Log frequency scale (like CQT), mel scale, etc.
  - Dynamics
- Resonators are independent
  - Parallel processing of each sample
- Analogous to bank of non-linear band pass filters

```
// Array of independent resonator instances
class ResonatorBankArray {
  var resonators = [Resonator]()
  […]
}

// A bank of independent resonators
// implemented as a single array
// computations use the Accelerate framework
// with manual memory management
// (unsafe pointers)
class ResonatorBankVec {
  […]

  // Accumulated resonance values
  // non-interlaced real (cos) | imaginary (sin) parts
  var rPtr : UnsafeMutableBufferPointer<Float>

  // Smoothed accumulated resonance values
  // non-interlaced real (cos) | imaginary (sin) parts
  var rrPtr : UnsafeMutableBufferPointer<Float>

  […]
}
```

Resonator bank with Accelerate (SIMD)

```
// Update with sample

// compute alphas * sample
vDSP_vsmul(alphas, 1,
           &sample,
           alphasSample.baseAddress!, 1,
           vDSP_Length(twoNumResonators))

// resonator update
vDSP_vmma(rPtr.baseAddress!, 1,
          omAlphas, 1,
          zPtr.baseAddress!, 1,
          alphasSample.baseAddress!, 1,
          rPtr.baseAddress!, 1,
          vDSP_Length(twoNumResonators))

// smoothing using beta
vDSP_vmma(rrPtr.baseAddress!, 1,
          omBetas, 1,
          rPtr.baseAddress!, 1,
          betas, 1,
          rrPtr.baseAddress!, 1,
          vDSP_Length(twoNumResonators))
```

```
// Single resonator
// Update with sample

alphaSample = alpha * sample


omAlpha = 1.0 - alpha


c = omAlpha * c + alphaSample * Zc
s = omAlpha * s + alphaSample * Zs



omBeta = 1.0 - beta
cc = omBeta * cc + beta * c
ss = omBeta * ss + beta * s
```

Resonator bank with Accelerate (SIMD)

```
// Update with sample

// compute alphas * sample
vDSP_vsmul(alphas, 1,
           &sample,
           alphasSample.baseAddress!, 1,
           vDSP_Length(twoNumResonators))

// resonator update
vDSP_vmma(rPtr.baseAddress!, 1,
          omAlphas, 1,
          zPtr.baseAddress!, 1,
          alphasSample.baseAddress!, 1,
          rPtr.baseAddress!, 1,
          vDSP_Length(twoNumResonators))

// smoothing using beta
vDSP_vmma(rrPtr.baseAddress!, 1,
          omBetas, 1,
          rPtr.baseAddress!, 1,
          betas, 1,
          rrPtr.baseAddress!, 1,
          vDSP_Length(twoNumResonators))
```

```
// all vectors of size 2 * numResonators
// cosine and sine


// Advance phasor
// Z <- Z * W
// Z and W are DSPSplitComplex
vDSP_zvmul(&Z, 1,
           &W, 1,
           &Z, 1,
           vDSP_Length(numResonators),
           1)


// Re-normalization
vDSP.squareMagnitudes(Z, result: &smPtr)
// use reciprocal square root
vForce.rsqrt(smPtr, result: &rsqrtPtr)
vDSP.multiply(Z, by: rsqrtPtr, result: &Z)
```

Performance

	Array of resonators		Vectorized (SIMD)
Swift	Frame Size 32 Implementation Swift Update Model Sequential Processing time per sample (ns): 19,736 Max samples per second: 50,670 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz	Frame Size 32 Implementation Swift Update Model Concurrent Processing time per sample (ns): 4,842 Max samples per second: 206,508 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz	Frame Size 32 Implementation Vectorized Update Model Processing time per sample (ns): 396 Max samples per second: 2,526,249 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz
C++	Frame Size 32 Implementation C++ Update Model Sequential Processing time per sample (ns): 934 Max samples per second: 1,071,130 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz	Frame Size 32 Implementation C++ Update Model Concurrent Processing time per sample (ns): 1,404 Max samples per second: 712,425 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz	Frame Size 32 Implementation Vectorized (C++) Update Model Processing time per sample (ns): 116 Max samples per second: 8,629,989 Input Device com.rogueamoeba.Loopb...82-A89B-99E57AC1FFFB Sample rate: 44,100 Hz

Processing time per sample on MacBookPro M4 Pro - debug mode! (higher overhead for Swift versions)

Things NOT to do at Every Sample

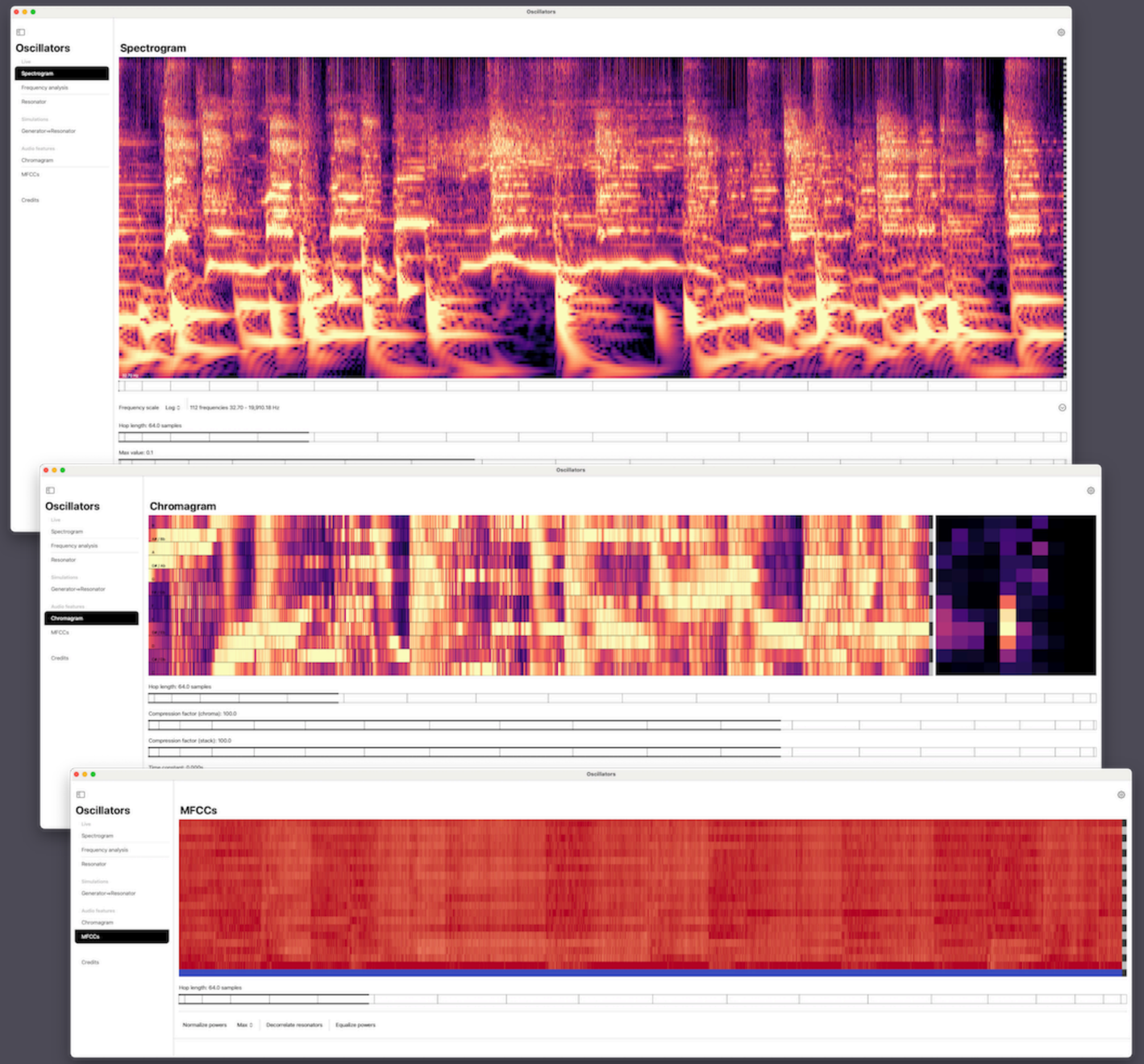
- Allocate buffer/arrays
  - Including those used in intermediary computations
- Copy/duplicate data
  - Most Accelerate functions can operate in-place (but not all…)
- Perform unnecessary computations
- Leak memory!

Outline

- Intro
- Single resonator
- Resonator banks
- **Apps and applications**

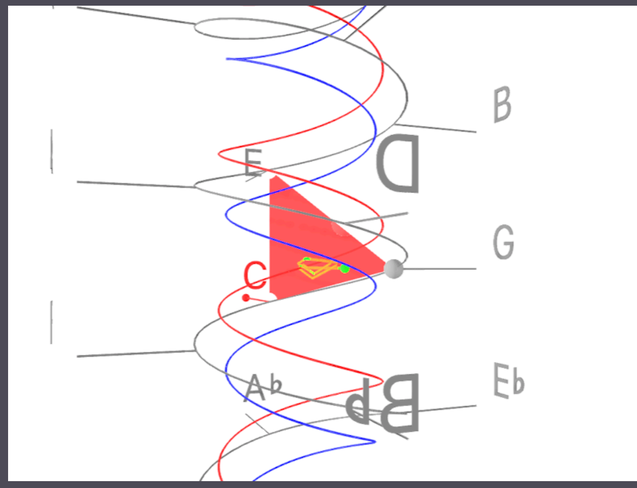
Oscillators App

- A playground for Oscillators Swift package
  - Single resonator
  - Resonator banks
  - **Spectrograms, chromagrams, MFCCs**
- AudioKit
- Swift UI
- Resonate Swift package
- Accelerate



Applications

- MuSA_RT?



MuSA_RT

- JUCE plugins?
- Optimized C++ implementation using Eigen for portability?
- ML
  - Replace Constant-Q transform: save memory and time; reduce inference latency (possibly not significantly compared to overall model latency, but it’s a start…)
  - Efficient tokenization

Python wrapper (noFFT module)

- Resonate algorithm is auto regressive: cannot be parallelized across time
  - Usual Python vectorization does not work out of the box
  - Implement iteration loop in lower-level language (e.g. C++)
- noFFT module
  - C++ implementation with Python wrappers and utility functions
  - Jupyter notebooks
  - Uses Accelerate -> Eigen implementation for portability?
  - GPU implementation?

Thank you!

www.alexandrefrancois.org/Resonate