ADC23

JAN WILCZEK

BUG-FREE AUDIO CODE:

LEVERAGE SIMPLE DSP PRINCIPLES TO WRITE ROCK-SOLID MUSIC SOFTWARE EVERY TIME

Where do my knowledge and experience come from?

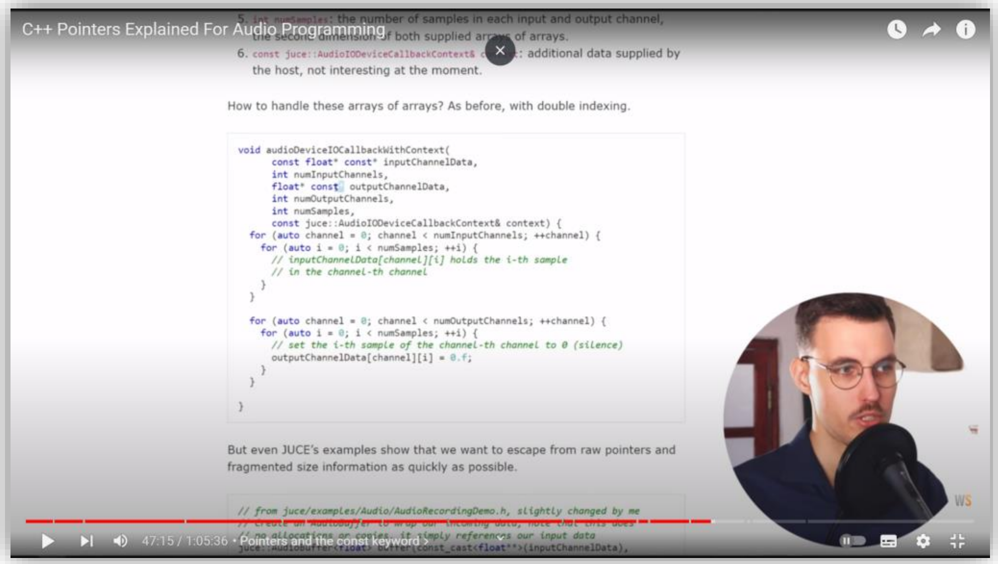
- C++ audio developer for Android and Windows

Music Maker JAM

Loudly

Where do my knowledge and experience come from?

- Educator and coach in the area of audio programming
  - thewolfsound.com
  - youtube.com/c/WolfSoundAudio



Where do my knowledge and experience come from?

- Worked on JUCE audio plugins and a game audio engine
- DSP research with prototyping in Python
- Studied advanced DSP @ Uni Erlangen
- Huge interest in C++/audio best and common practices

Personal sources of information ☺

- Christoph Guttandin
- Ian Hobson
- Oliver Larkin
- Russell McClellan
- Roth Michaels
- Joe Noel
- Sam Tarakajian
- Stefano D’Angelo
- Ken Bogdanowicz
- Sebastian Freund
- Anna Wszeborowska
- Aleksandra Korach
- Moritz Schaller
- Stefan Gretscher

Outline

- Motivation
- What to do for bug-free audio code
- How to do it
  - Prerequisites
  - 2 approaches
- Practical examples
- More resources

Motivation

- No one asks me
  - “How to write correct audio code?”
  - “How to write unit tests for audio code?”
- Instead, I hear
  - “Why is my audio code glitching?”

#1 most common programming mistake that we see on the forum

Audio Plugins

jules Jan 2018

This post is here so that we have something we can link to when this same issue comes up, because the JUCE team is getting very bored of spending time replying again and again to the same damned problem!

This isn’t a JUCE bug - there’s nothing we can do in our codebase that would stop people making it, which makes it painful for us to watch beginner after beginner come along and fall headfirst into the same trap!

Goal

- Write bug-free, real-time audio code
  - in C++ using JUCE
  - in C++ for an Android application
  - in C++ for a game audio engine
  - in … for …

Why is bug-free audio code crucial to your business's success?

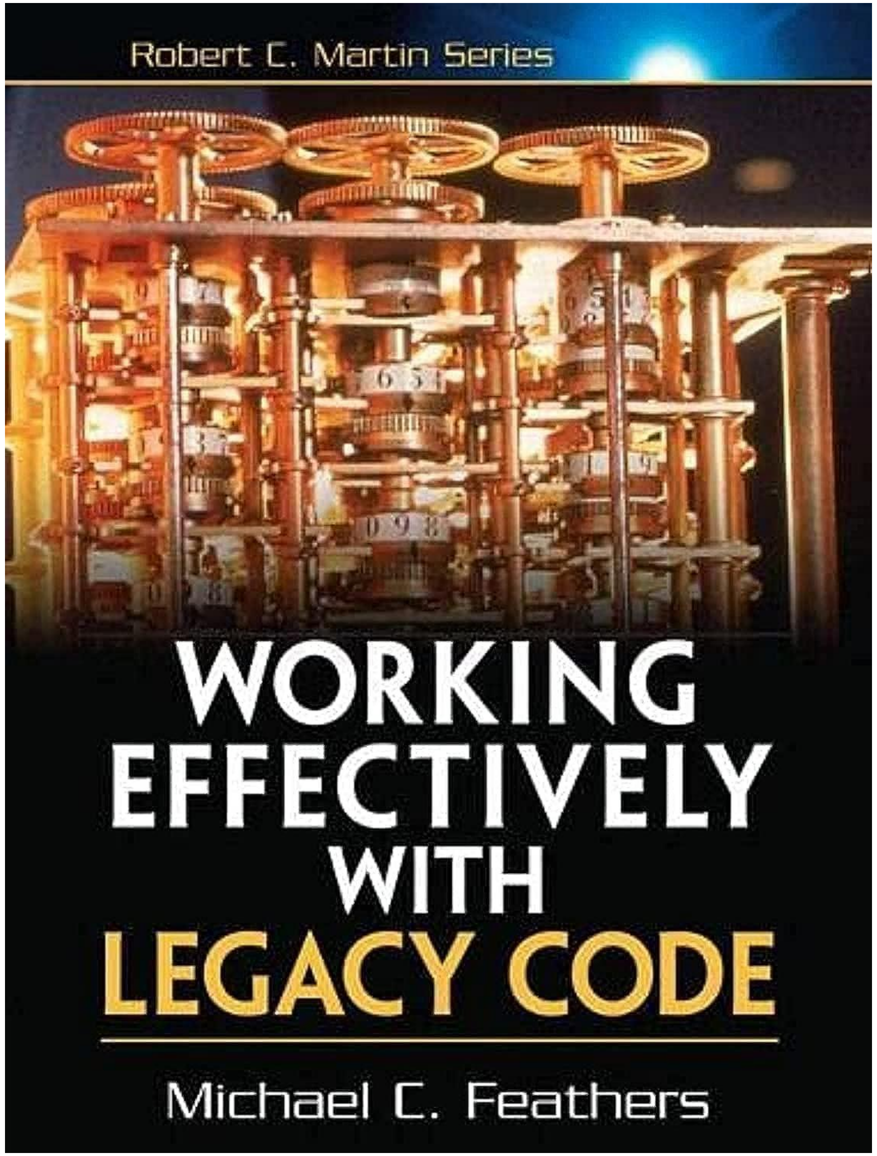
- Better UX and higher user satisfaction → better reviews

> ★★☆☆☆ April 15, 2021
>
> I use this app to record instruments in real life and put them together in the app. This app does it just fine but when I record the entire track, I end up with only 30 or 50 seconds of it which is very frustrating.

- Developer’s confidence
- Reduced QA workload
- Team’s morale

Popular but flawed approaches to verifying the correctness of audio software:

- Listening
- Re-read your own code
- Code review
- Mathematical proof of correctness
- QA testing
- End-user testing


Robert C. Martin Series
WORKING EFFECTIVELY WITH LEGACY CODE
Michael C. Feathers

Problems of unit testing audio code in C++

- Audio I/O requires dependencies
- Audio analysis functions require dependencies
- Insufficient visualization tools
- Large code overhead
- Unclear compiler messages

Why use Python with audio C++?

- Easy and fast package management
- Powerful visualization tools
- A lot of DSP algorithms and numerical operations already implemented
- Deep learning libraries
- Easy networking
- Easy OS-agnostic scripting
- Fast prototyping

How to combine Python with C++?

- Through C++ output (.wav, .csv)

```
ret = os.system(f'cd {root_repo_path} && cmake -S . -B build && cmake --build build &&'
                f' {test_executable_path} --gtest_filter="OutputTest.*"')

data, fs = sf.read(root_repo_path / 'sine.wav')
```

- Through C++ bindings (pybind11, cppyy)

```
cppyy.add_include_path(os.path.dirname(os.path.abspath(sys.argv[0])))
cppyy.include("gain.h")
ones = np.ones((10,))

ones_vector = std.vector[float](ones.tolist())

# call C++ implementation
cppyy.gbl.apply_gain(ones_vector)
```

2 testing approaches you need

- Reference comparison testing
- Unit tests with DSP principles

Reference comparison testing

Leverage the power of what's already done to do more

Reference audio file (regression tests)

1. Render a reference audio file
   - e.g., a fixed test project
2. Check into git
3. Refactor/extend the code
4. Render the audio file again
5. Compare sample-wise against the reference

Reference implementation

- Use the output of already written and tested code as the desired output of your code.
- Sources of implementation
  - Python libraries
  - MATLAB
  - GNU Octave
  - 3rd-party libraries

```cpp
TEST(ButterworthFilterTest, SecondOrderCoefficientsAreCorrect) {
    auto filter = ButterworthHighpassFilter(2);
    constexpr auto cutoffFrequency = 100.f / 44100.f;
    filter.cutoffFrequency(cutoffFrequency);
    const auto frequencyWarpingConstant = std::tan(pi * cutoffFrequency);

    const auto numerator = filter.numerator();
    const auto denominator = filter.denominator();

    const auto frequencyWarpingConstantSquared = frequencyWarpingConstant * frequencyWarpingConstant;

    const auto expecteda0 = 1.f + 2 * frequencyWarpingConstant * std::cos(pi / 4.f) + frequencyWarpingConstantSquared;
    ASSERT_EQ(numerator.size(), 3u);
    ASSERT_EQ(denominator.size(), 3u);
    ASSERT_EQ(numerator[0], 1.f / expecteda0);
    ASSERT_EQ(numerator[1], -2.f / expecteda0);
    ASSERT_EQ(numerator[2], 1.f / expecteda0);
    ASSERT_EQ(denominator[0], 1.f);
    ASSERT_EQ(denominator[1], (2.f * frequencyWarpingConstantSquared - 2.f) / expecteda0);
    ASSERT_EQ(denominator[2], (1.f - 2.f * frequencyWarpingConstant * std::cos(pi / 4.f) + frequencyWarpingConstantSquared)
                          / expecteda0);
}
```

```cpp
TEST(ButterworthFilterTest, ThirdOrderCoefficientsAreCorrect) {
    auto filter = ButterworthHighpassFilter(3);

    const auto cutoffFrequency = 200.f / 44100.f;
    filter.cutoffFrequency(cutoffFrequency);

    const auto numerator = filter.numerator();
    const auto denominator = filter.denominator();

    ASSERT_EQ(numerator.size(), 4u);
    ASSERT_EQ(denominator.size(), 4u);

    // These values were obtained by running scipy.signal.butter(3, 200, 'highpass', fs=44100) in Python.
    EXPECT_FLOAT_EQ(numerator[0], 0.97190605f);
    EXPECT_FLOAT_EQ(numerator[1], -2.91571815f);
    EXPECT_FLOAT_EQ(numerator[2], 2.91571815f);
    EXPECT_FLOAT_EQ(numerator[3], -0.97190605f);
    EXPECT_FLOAT_EQ(denominator[0], 1.f);
    EXPECT_FLOAT_EQ(denominator[1], -2.94301158f);
    EXPECT_FLOAT_EQ(denominator[2], 2.88763545f);
    EXPECT_FLOAT_EQ(denominator[3], -0.94460137f);
}
```

Prototype implementation

- 4 tasks when implementing an audio algorithm:
  - Algorithm design (with tweaking)
  - Implementing the correct algorithm
  - Writing correct C++ code
  - Integrating the algorithm into the existing codebase/framework
- Python implementation as the baseline
- Unoptimized/inaccurate C++ implementation as the baseline

Baseline vs advanced algorithm

- Zero-crossings algorithms against an advanced pitch tracker
- Time-domain autocorrelation vs FFT-based autocorrelation
- FIR filter implementation: plain (sample-wise) vs SIMD-optimized vs fast convolution

```python
def apply_gain(samples):
    samples *= 0.5
```

```python
import numpy as np

def apply_gain(samples):
    samples *= 0.5

def generate_reference():
    ones = np.ones((10,))

    apply_gain(ones)

    # save output
    # ...
```

```cpp
// gain.h
#pragma once
#include <vector>


void apply_gain(std::vector<float>& samples) {
  for (auto& sample : samples) {
    sample *= 0.5f;
  }
}
```

```python
import cppyy
from cppyy.gbl import std
import numpy as np
import os, sys

def test_cpp():
    cppyy.add_include_path(os.path.dirname(os.path.abspath(sys.argv[0])))
    cppyy.include("gain.h")
    ones = np.ones((10,))

    ones_vector = std.vector[float](ones.tolist())

    # call C++ implementation
    cppyy.gbl.apply_gain(ones_vector)

    # test against saved output
    # ...
```

```python
import cppyy
from cppyy.gbl import std
import numpy as np
import os, sys

def test_cpp():
    cppyy.add_include_path(os.path.dirname(os.path.abspath(sys.argv[0])))
    cppyy.include("gain.h")
    ones = np.ones((10,))

    ones_vector = std.vector[float](ones.tolist())

    # call C++ implementation
    cppyy.gbl.apply_gain(ones_vector)

    # test against saved output
    # ...
```

```python
import cppyy
from cppyy.gbl import std
import numpy as np
import os, sys

def test_cpp():
    cppyy.add_include_path(os.path.dirname(os.path.abspath(sys.argv[0])))
    cppyy.include("gain.h")
    ones = np.ones((10,))

    ones_vector = std.vector[float](ones.tolist())

    # call C++ implementation
    cppyy.gbl.apply_gain(ones_vector)

    # test against saved output
    # ...
```

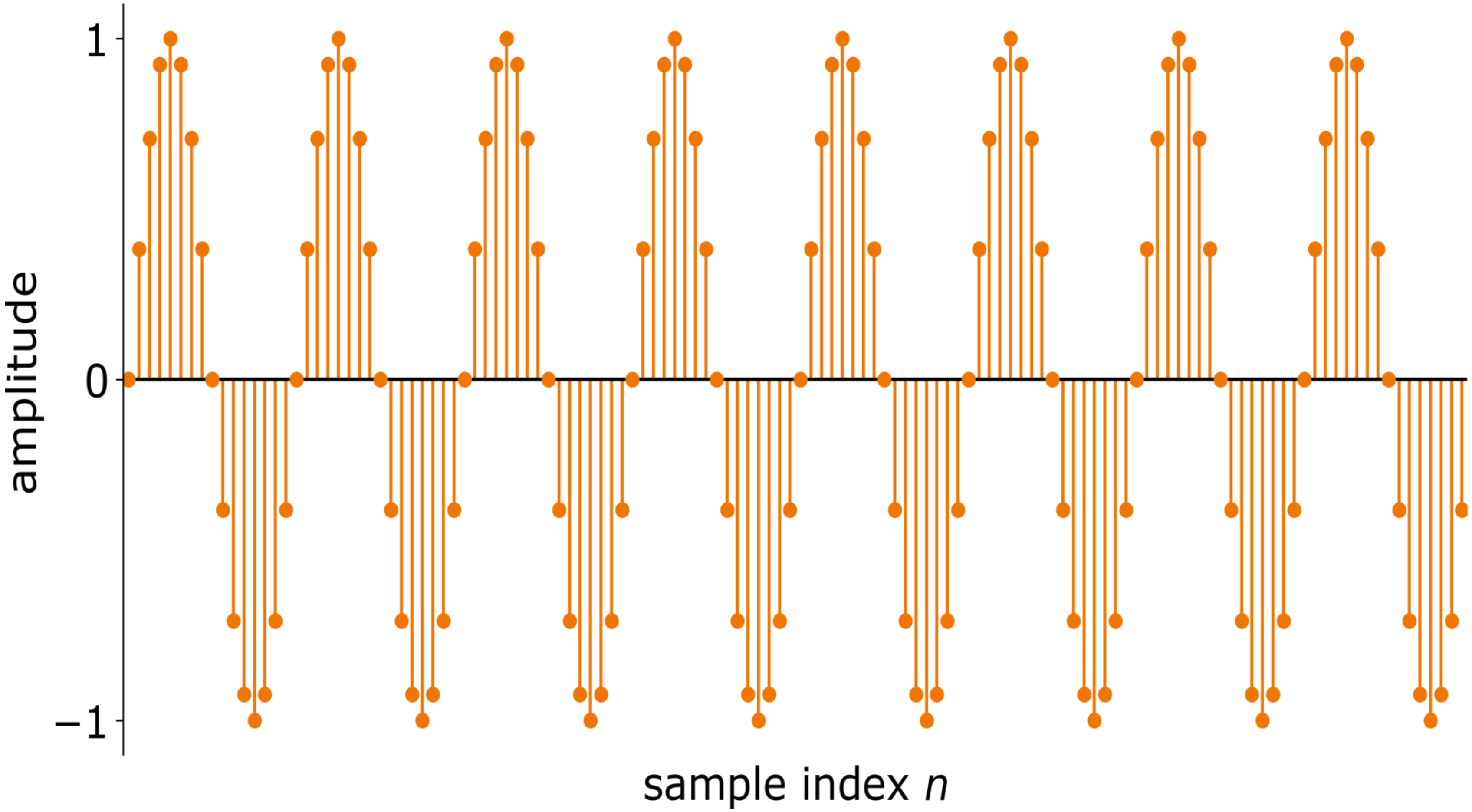
Unit tests with DSP principles

Top 7 DSP principles to make writing your unit tests easy

#1: Known-input, known-output

- Most DSP systems are “boxes” with input and output.
- Test for equality or acceptable error, e.g., mean squared error (MSE).
- Examples:
    - 1st-order Butterworth filter should attenuate a sine at the cutoff frequency by 3 dB.
    - Pitch shifter with ratio of 2 should pitch-shift a sine at the DFT bin no. 1 to DFT bin no. 2.

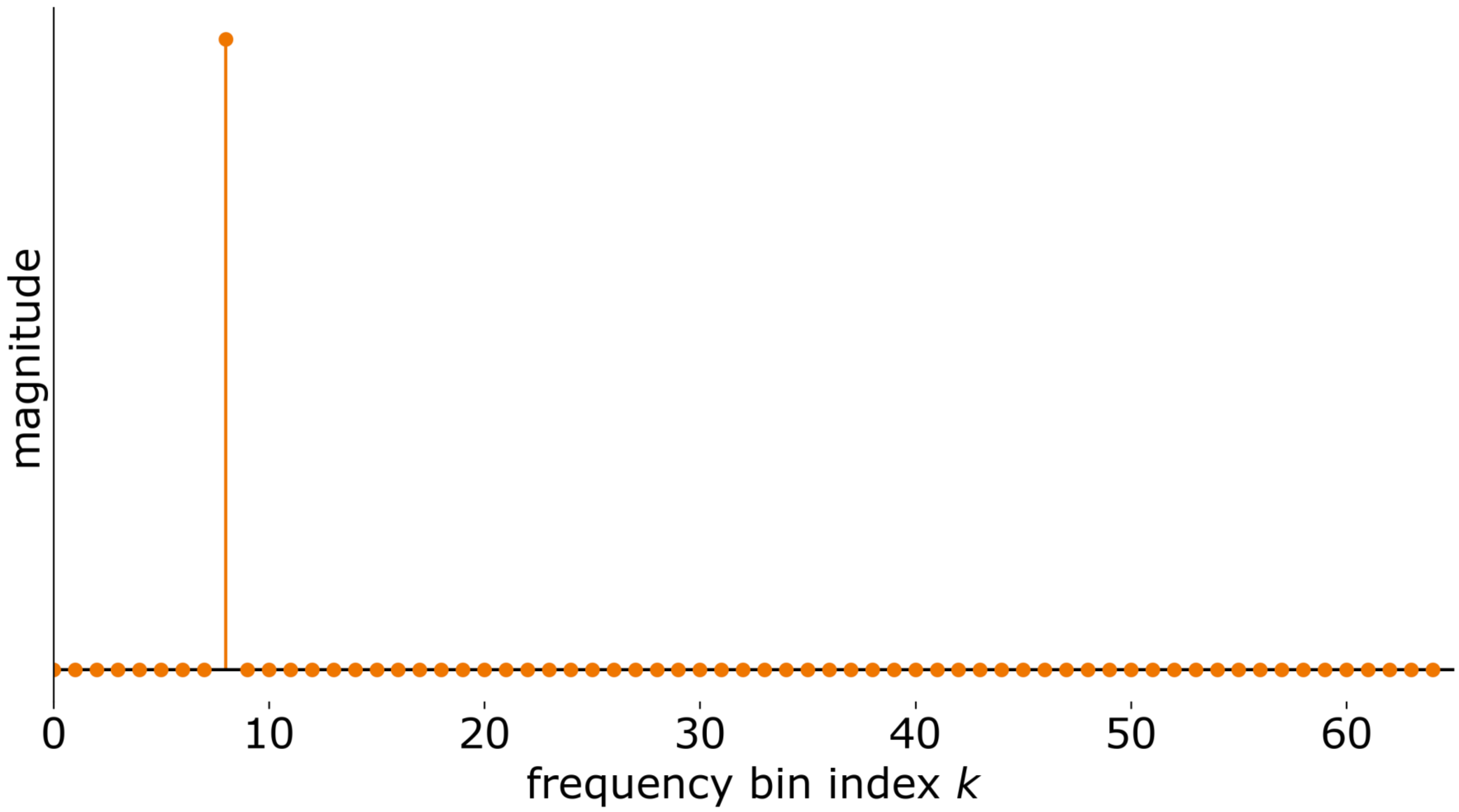
Sine hitting a DFT bin



DFT of size 128

8 full periods

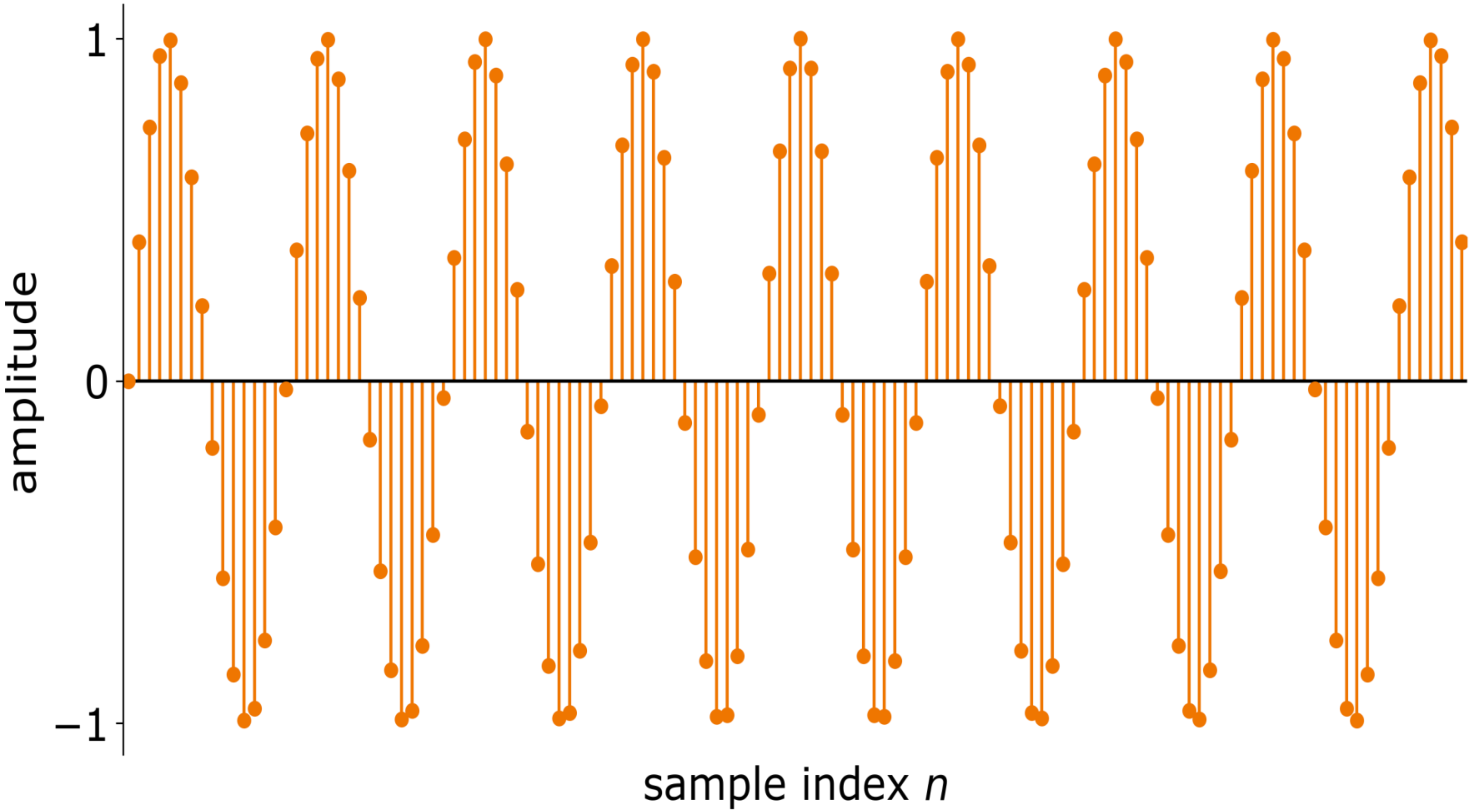
Sine hitting a DFT bin



DFT of size 128

8 full periods

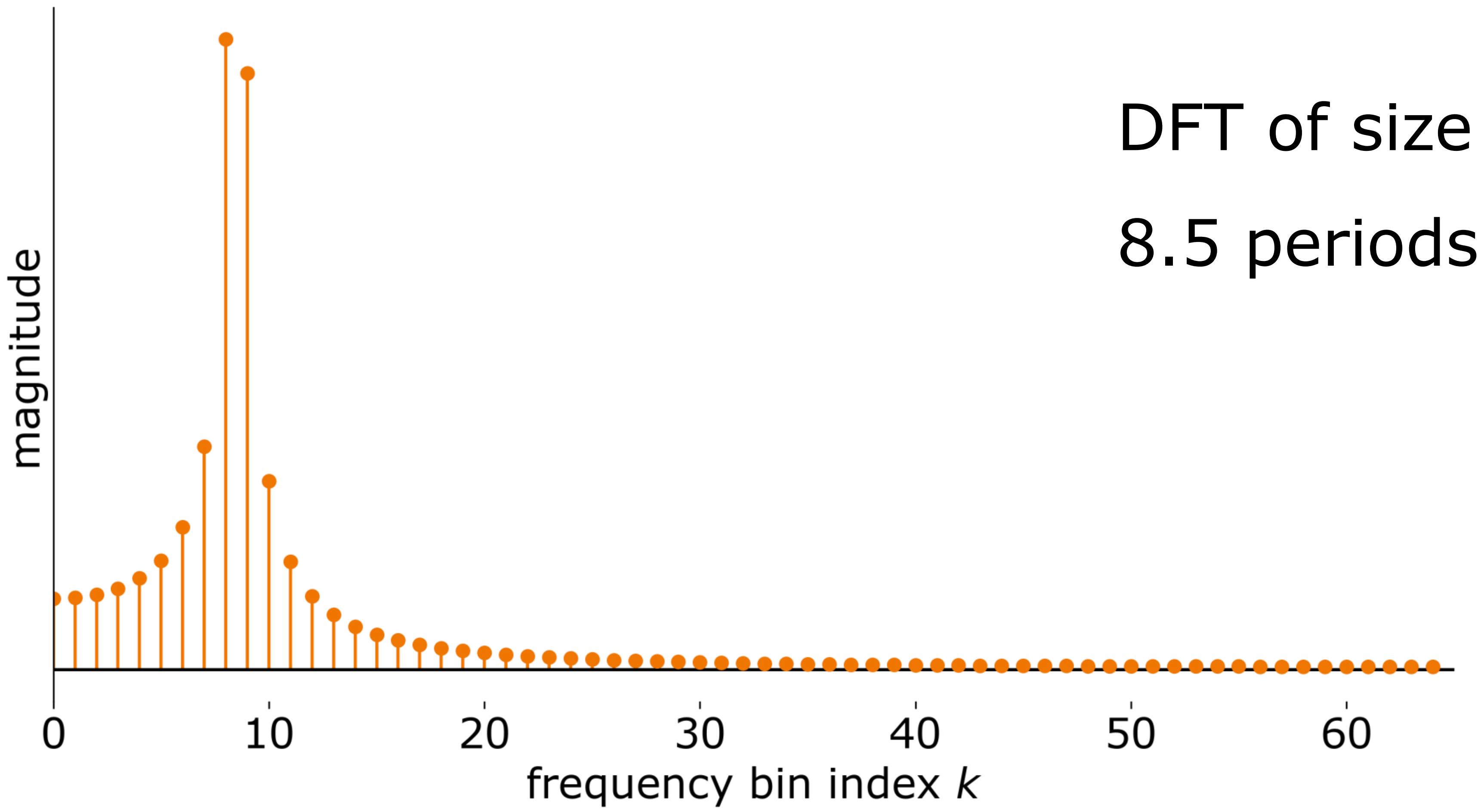
Sine exactly between two DFT bins (spectral leakage)



DFT of size 128

8.5 periods

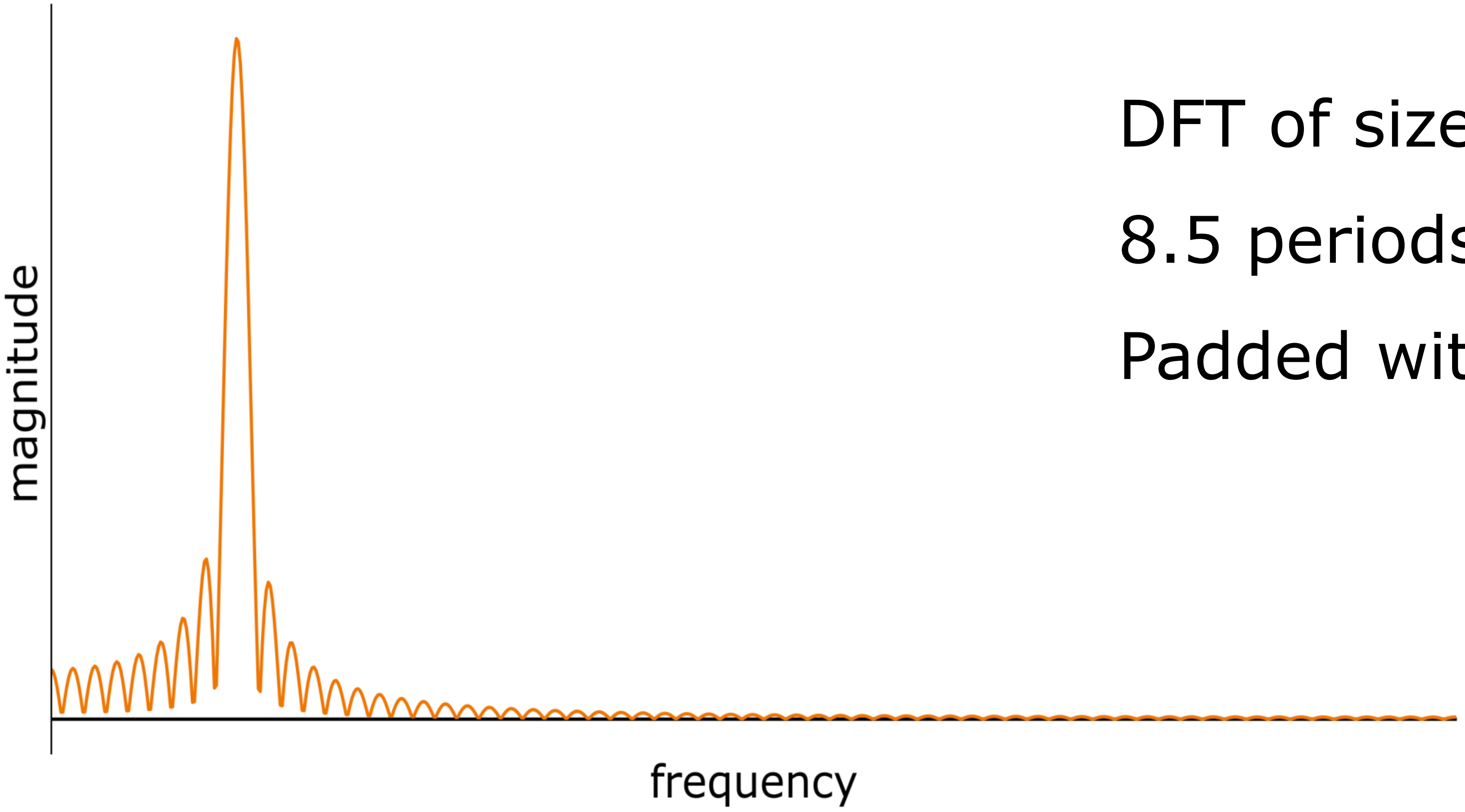
Sine exactly between two DFT bins (spectral leakage)



DFT of size 128

8.5 periods

Sine exactly between two DFT bins (spectral leakage)



DFT of size 128

8.5 periods

Padded with 1280 zeros

#2: Inverse operation

- Use an inverse operation to the tested one to obtain back input data
- Example: FFT and IFFT

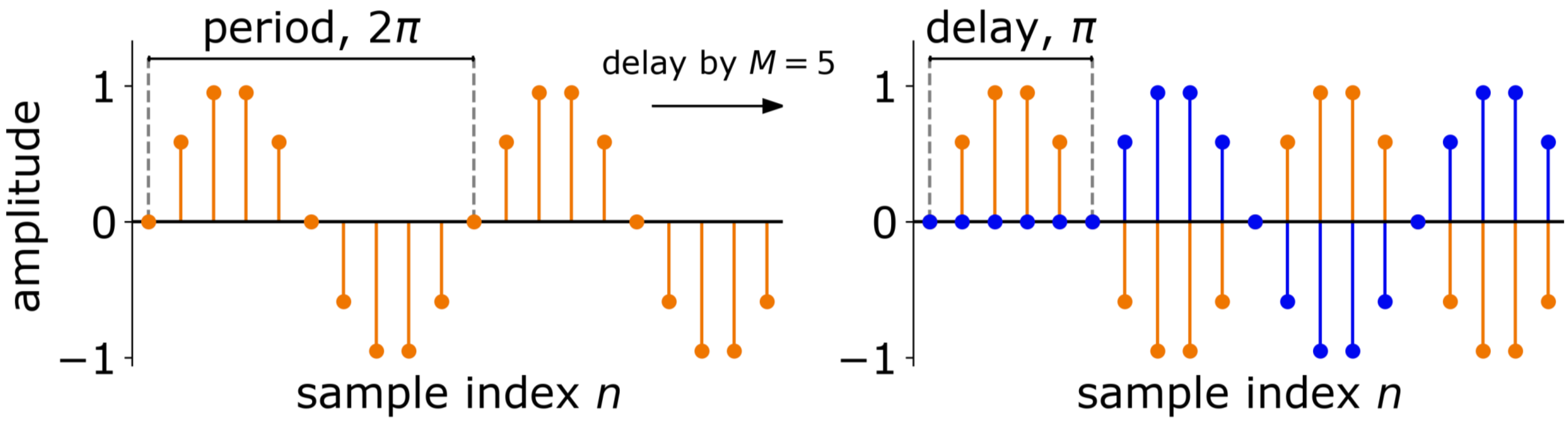
```cpp
TEST(FFTTest, ForwardAndInverseTransformDoesNotScaleSignal) {
  constexpr auto FFT_SIZE = 512;
  FFT fft{FFT_SIZE};

  auto unitStep = fft.alignedBuffer();
  std::fill(unitStep.begin(), unitStep.end(), 1.f);
  auto dft = std::vector<std::complex<float>>(FFT_SIZE);
  auto output = fft.alignedBuffer();

  fft.forward(unitStep, dft.data());
  fft.inverse(dft.data(), output);

  for (const auto el : output) {
    EXPECT_FLOAT_EQ(1.f, el);
  }
}
```

```cpp
TEST(FFTTest, ForwardAndInverseTransformDoesNotScaleSignal) {
  constexpr auto FFT_SIZE = 512;
  FFT fft{FFT_SIZE};

  auto unitStep = fft.alignedBuffer();
  std::fill(unitStep.begin(), unitStep.end(), 1.f);
  auto dft = std::vector<std::complex<float>>(FFT_SIZE);
  auto output = fft.alignedBuffer();

  fft.forward(unitStep, dft.data());
  fft.inverse(dft.data(), output);

  for (const auto el : output) {
    EXPECT_FLOAT_EQ(1.f, el);
  }
}
```

```cpp
TEST(FFTTest, ForwardAndInverseTransformDoesNotScaleSignal) {
  constexpr auto FFT_SIZE = 512;
  FFT fft{FFT_SIZE};

  auto unitStep = fft.alignedBuffer();
  std::fill(unitStep.begin(), unitStep.end(), 1.f);
  auto dft = std::vector<std::complex<float>>(FFT_SIZE);
  auto output = fft.alignedBuffer();

  fft.forward(unitStep, dft.data());
  fft.inverse(dft.data(), output);

  for (const auto el : output) {
    EXPECT_FLOAT_EQ(1.f, el);
  }
}
```

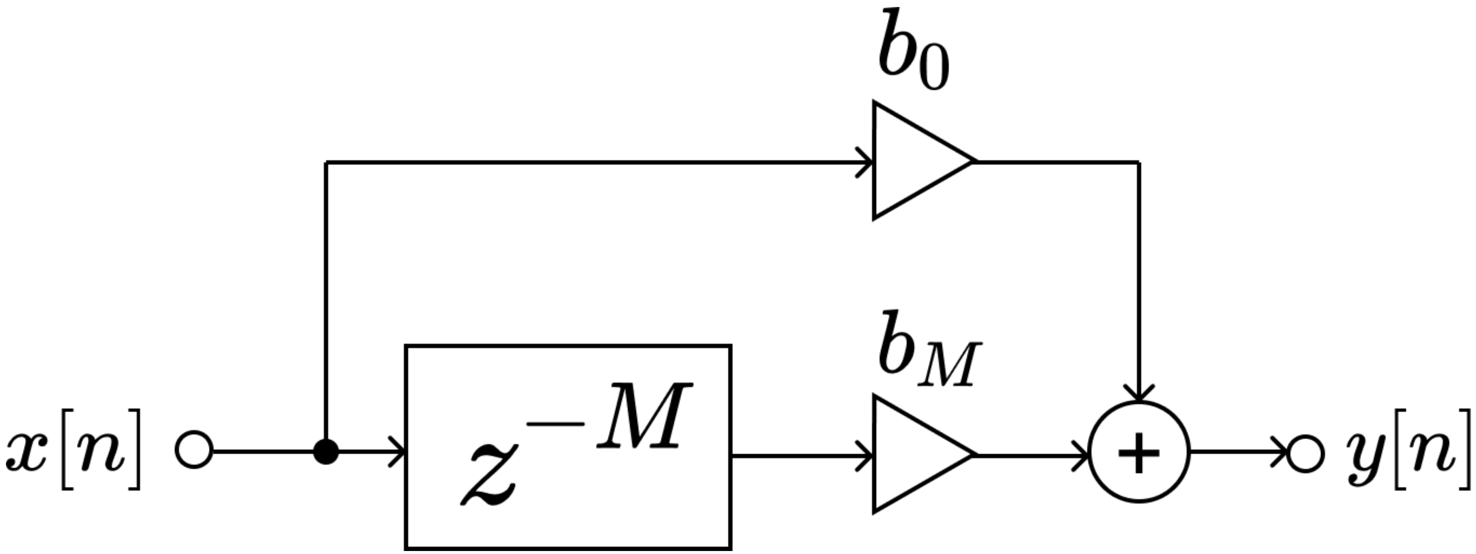
```cpp
TEST(FFTTest, ForwardAndInverseTransformDoesNotScaleSignal) {
  constexpr auto FFT_SIZE = 512;
  FFT fft{FFT_SIZE};

  auto unitStep = fft.alignedBuffer();
  std::fill(unitStep.begin(), unitStep.end(), 1.f);
  auto dft = std::vector<std::complex<float>>(FFT_SIZE);
  auto output = fft.alignedBuffer();

  fft.forward(unitStep, dft.data());
  fft.inverse(dft.data(), output);

  for (const auto el : output) {
    EXPECT_FLOAT_EQ(1.f, el);
  }
}
```

#3: Enforce phase cancellation



#3: Enforce phase cancellation

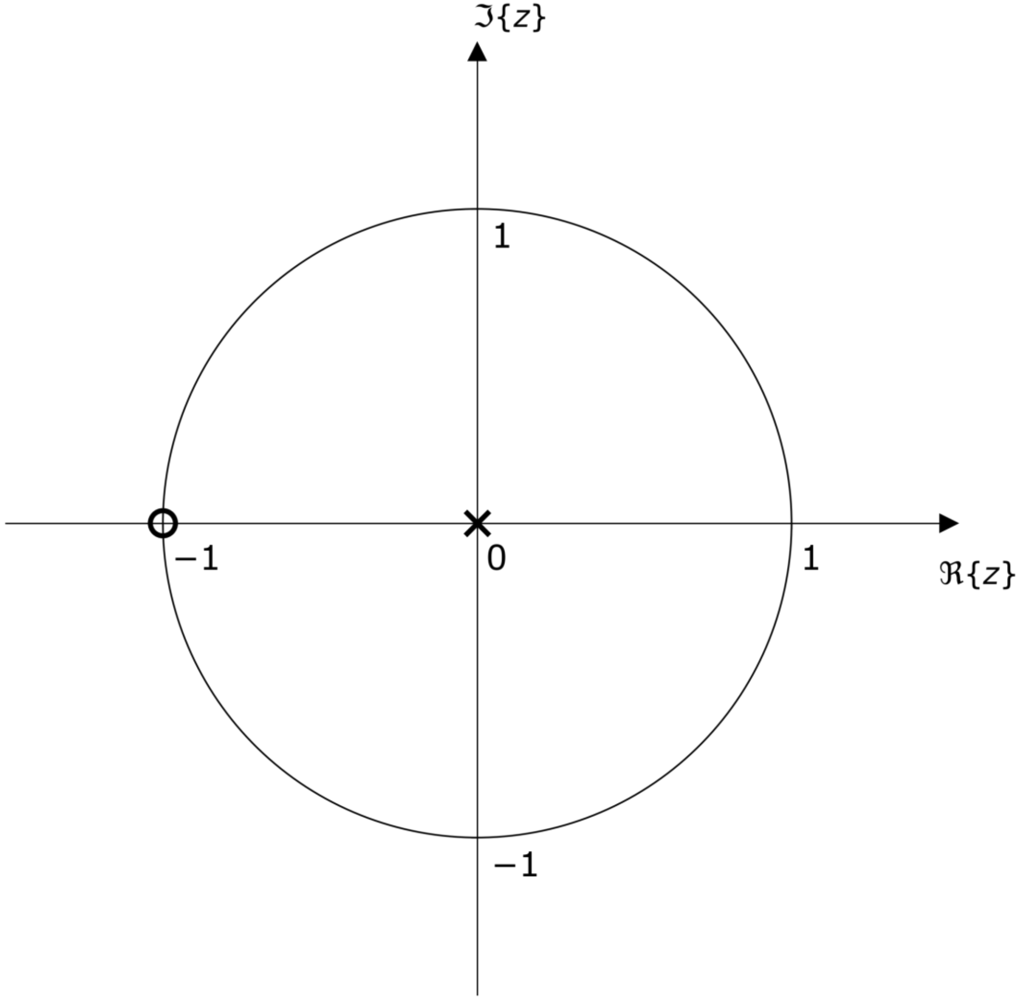
- Feedforward comb filter



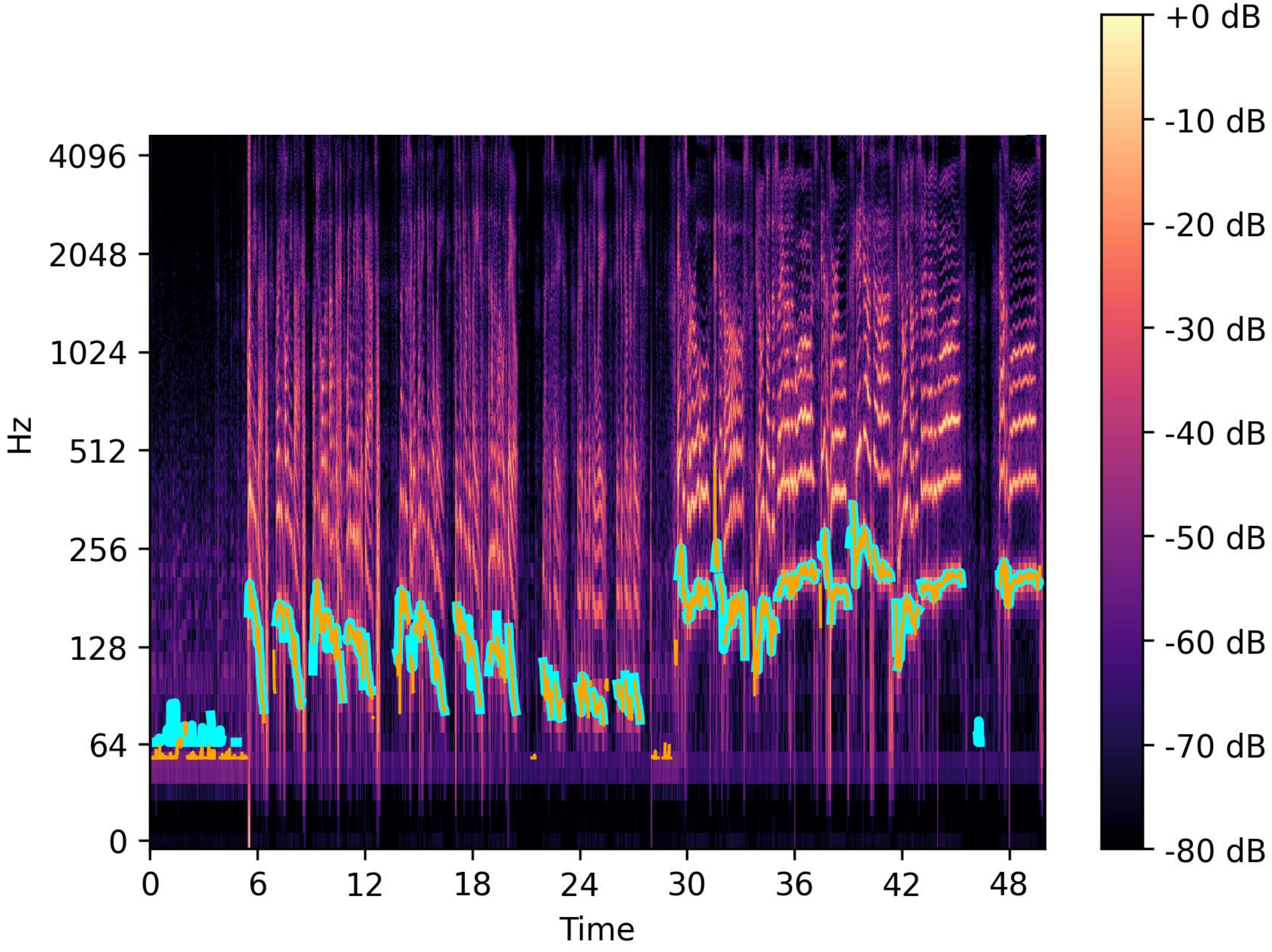
```cpp
TEST(FeedforwardCombFilter, DelayByHalfPeriodResultsInPhaseCancellation) {
  CF::FeedForwardCombFilter ffcf;
  auto signal = generateSineWithPeriodOf(10.f);

  ffcf.setDelay(5);
  ffcf.setDelayedGain(1.f);
  ffcf.setDirectGain(1.f);

  // filter the sine
  for (int i = 0; i < std::ssize(signal); i++) {
    signal[i] = ffcf.process(signal[i]);
  }

  // the output after the first half of the period should be all zeros
  for (int i = 5; i < std::ssize(signal); i++) {
    ASSERT_NEAR(signal[i], 0.0, 1e-6);
  }
}
```

#4: Visual inspection

- Magnitude response
- Pole-zero plot
- Magnitude spectrum
- Spectrogram



Spectrogram inspection

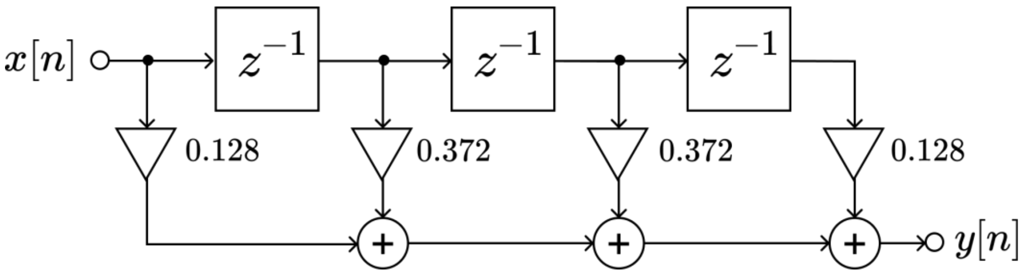


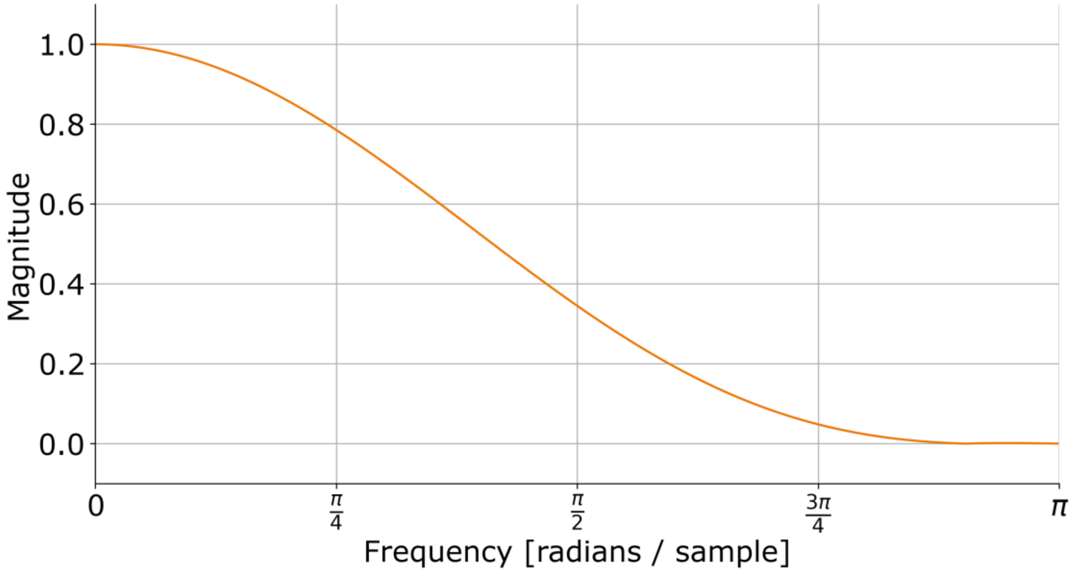
#5: Use signal measures

- Total Harmonic Distortion (THD) to verify that your system has (not) introduced distortion at the output.
- Crest factor, or peak-to-average-power ratio (PAPR), or root mean square value to ensure that a momentary signal has been properly captured.
  - E.g., verify that a loudspeaker-microphone loopback has been properly recorded

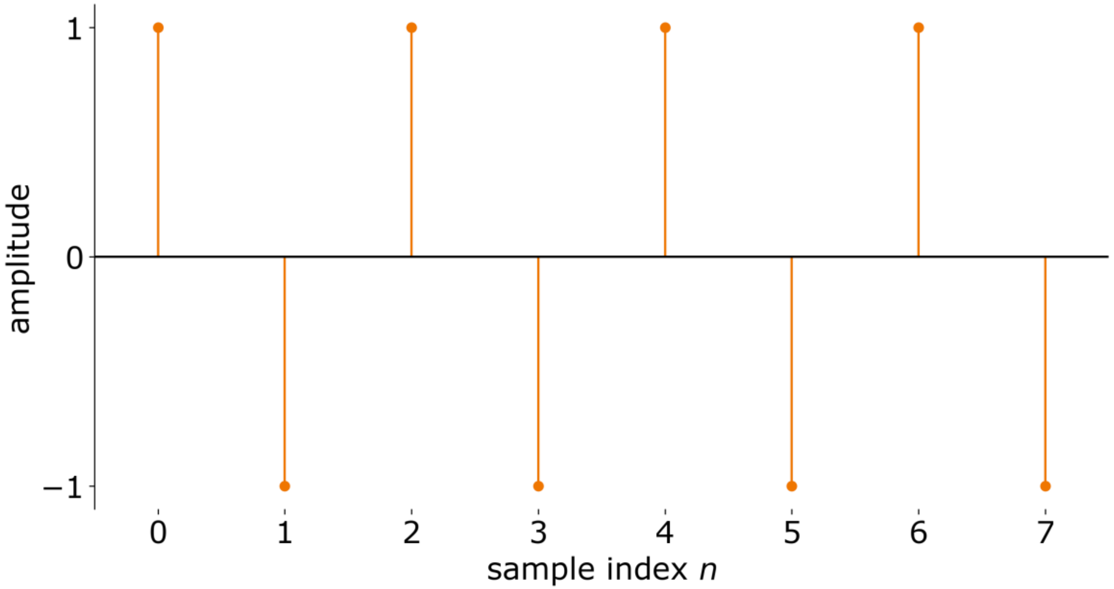
#6: Test for a specific predicted delay

- All deterministic DSP algorithms have a fixed algorithmic delay.



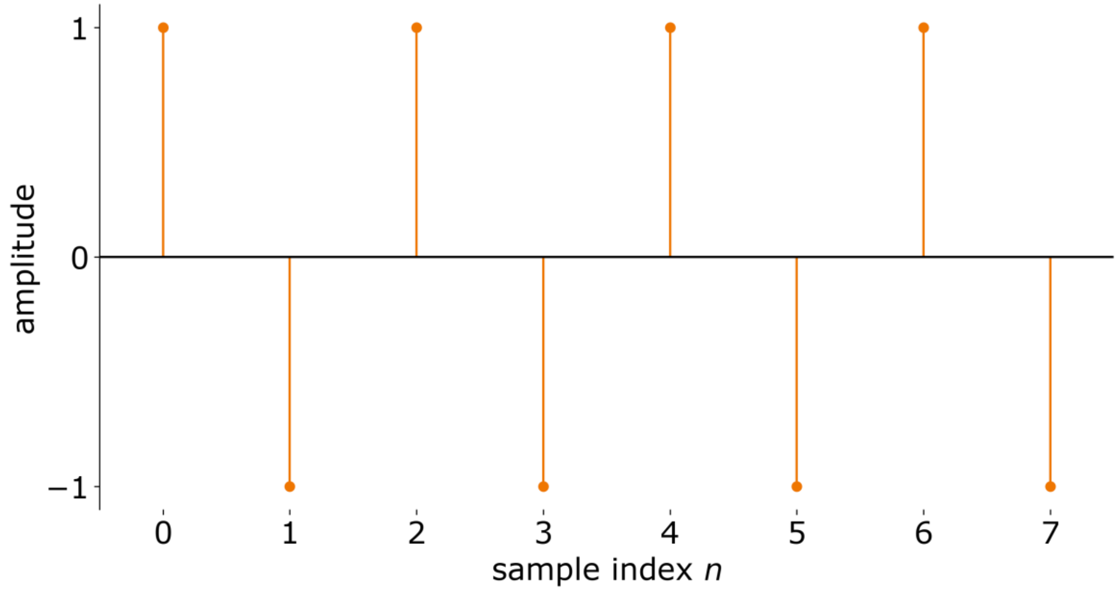


#6: Test for a specific predicted delay

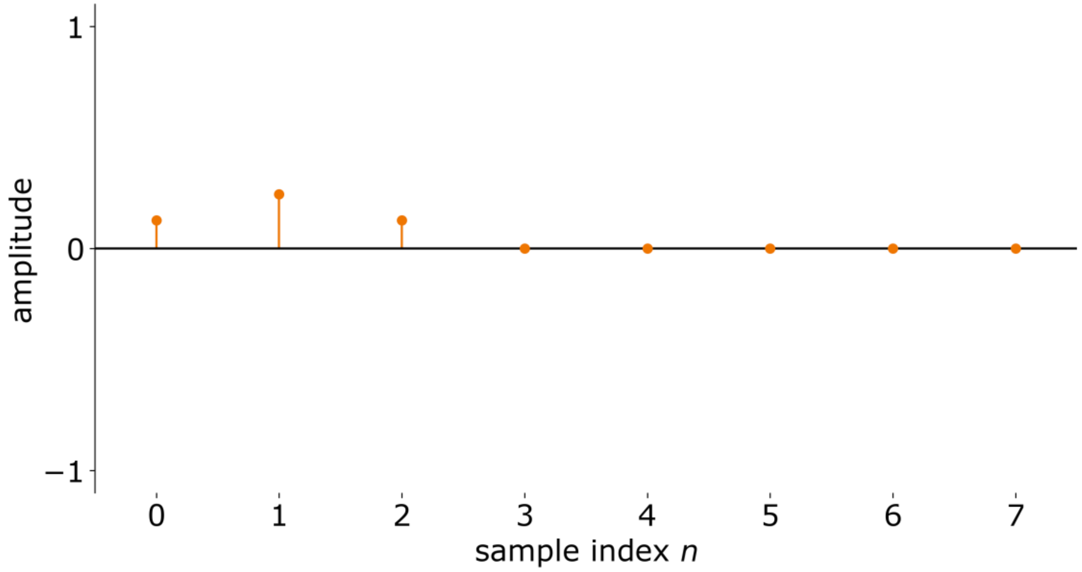


input: Nyquist

#6: Test for a specific predicted delay



input: Nyquist



output: transient, Nyquist response

```cpp
TEST(PhaseVocoderTest, AlgorithmicDelayIsEqualToWindowSize) {
    PhaseVocoder phaseVocoder;

    constexpr auto PHASE_VOCODER_ALGORITHMIC_DELAY = phaseVocoder.WINDOW_SIZE;
    constexpr auto SAMPLES_COUNT = (PHASE_VOCODER_ALGORITHMIC_DELAY / BUFFER_SIZE + 1) * BUFFER_SIZE;

    phaseVocoder.prepareToPlay(SAMPLING_RATE, BUFFER_SIZE);

    const auto ones = std::vector<float>(SAMPLES_COUNT, 1.f);
    auto output = std::vector<float>(SAMPLES_COUNT, 0.f);

    for (auto i = 0; i < SAMPLES_COUNT; i += BUFFER_SIZE) {
        phaseVocoder.process(ones.data() + i, BUFFER_SIZE, output.data() + i);
    }

    for (auto i = 0; i < PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(0., output[i], 1e-3) << " at index " << i;
    }

    for (auto i = 0; i < std::ssize(ones) - PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(ones[i], output[i + PHASE_VOCODER_ALGORITHMIC_DELAY], 3e-3) << " at index " << i;
    }
}
```

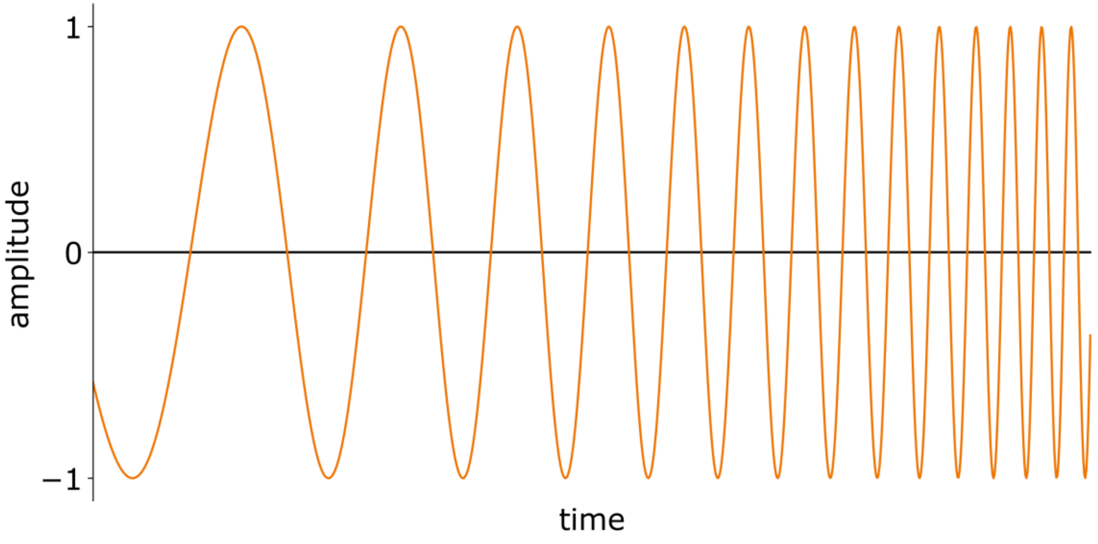
```cpp
TEST(PhaseVocoderTest, AlgorithmicDelayIsEqualToWindowSize) {
    PhaseVocoder phaseVocoder;

    constexpr auto PHASE_VOCODER_ALGORITHMIC_DELAY = phaseVocoder.WINDOW_SIZE;
    constexpr auto SAMPLES_COUNT = (PHASE_VOCODER_ALGORITHMIC_DELAY / BUFFER_SIZE + 1) * BUFFER_SIZE;

    phaseVocoder.prepareToPlay(SAMPLING_RATE, BUFFER_SIZE);

    const auto ones = std::vector<float>(SAMPLES_COUNT, 1.f);
    auto output = std::vector<float>(SAMPLES_COUNT, 0.f);

    for (auto i = 0; i < SAMPLES_COUNT; i += BUFFER_SIZE) {
        phaseVocoder.process(ones.data() + i, BUFFER_SIZE, output.data() + i);
    }

    for (auto i = 0; i < PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(0., output[i], 1e-3) << " at index " << i;
    }

    for (auto i = 0; i < std::ssize(ones) - PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(ones[i], output[i + PHASE_VOCODER_ALGORITHMIC_DELAY], 3e-3) << " at index " << i;
    }
}
```

```cpp
TEST(PhaseVocoderTest, AlgorithmicDelayIsEqualToWindowSize) {
    PhaseVocoder phaseVocoder;

    constexpr auto PHASE_VOCODER_ALGORITHMIC_DELAY = phaseVocoder.WINDOW_SIZE;
    constexpr auto SAMPLES_COUNT = (PHASE_VOCODER_ALGORITHMIC_DELAY / BUFFER_SIZE + 1) * BUFFER_SIZE;

    phaseVocoder.prepareToPlay(SAMPLING_RATE, BUFFER_SIZE);

    const auto ones = std::vector<float>(SAMPLES_COUNT, 1.f);
    auto output = std::vector<float>(SAMPLES_COUNT, 0.f);

    for (auto i = 0; i < SAMPLES_COUNT; i += BUFFER_SIZE) {
        phaseVocoder.process(ones.data() + i, BUFFER_SIZE, output.data() + i);
    }

    for (auto i = 0; i < PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(0., output[i], 1e-3) << " at index " << i;
    }

    for (auto i = 0; i < std::ssize(ones) - PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(ones[i], output[i + PHASE_VOCODER_ALGORITHMIC_DELAY], 3e-3) << " at index " << i;
    }
}
```

```cpp
TEST(PhaseVocoderTest, AlgorithmicDelayIsEqualToWindowSize) {
    PhaseVocoder phaseVocoder;

    constexpr auto PHASE_VOCODER_ALGORITHMIC_DELAY = phaseVocoder.WINDOW_SIZE;
    constexpr auto SAMPLES_COUNT = (PHASE_VOCODER_ALGORITHMIC_DELAY / BUFFER_SIZE + 1) * BUFFER_SIZE;

    phaseVocoder.prepareToPlay(SAMPLING_RATE, BUFFER_SIZE);

    const auto ones = std::vector<float>(SAMPLES_COUNT, 1.f);
    auto output = std::vector<float>(SAMPLES_COUNT, 0.f);

    for (auto i = 0; i < SAMPLES_COUNT; i += BUFFER_SIZE) {
        phaseVocoder.process(ones.data() + i, BUFFER_SIZE, output.data() + i);
    }

    for (auto i = 0; i < PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(0., output[i], 1e-3) << " at index " << i;
    }

    for (auto i = 0; i < std::ssize(ones) - PHASE_VOCODER_ALGORITHMIC_DELAY; ++i) {
        ASSERT_NEAR(ones[i], output[i + PHASE_VOCODER_ALGORITHMIC_DELAY], 3e-3) << " at index " << i;
    }
}
```

#7: Helpful test signals

- Pure sinusoid at various frequencies
  - e.g., cutoff, Nyquist
- Unit step
- Farina sweep (exponentially increasing sine)



Practical examples

What’s wrong with this code?

```
float FractionalDelayLine::readSample() {
  auto readHead = writeHead - 1 - delay;
  if (readHead < 0) {
    readHead += std::ssize(buffer);
  }
  const auto truncatedReadHead = static_cast<int>(std::floor(readHead));
  auto truncatedReadHeadPlusOne = static_cast<int>(std::ceil(readHead));
  const auto truncatedReadHeadWeight = std::abs(readHead - truncatedReadHeadPlusOne);
  const auto truncatedReadHeadPlusOneWeight = std::abs(readHead - truncatedReadHead);
  if (truncatedReadHeadPlusOne >= std::ssize(buffer)) {
    truncatedReadHeadPlusOne -= std::ssize(buffer);
  }
  const auto outputSample = truncatedReadHeadWeight * buffer[truncatedReadHead] +
                    truncatedReadHeadPlusOneWeight * buffer[truncatedReadHeadPlusOne];
  return outputSample;
}
```

```
TEST(Flanger, FractionalDelay) {
  DL::FractionalDelayLine delayLine;

  delayLine.pushSample(1);
  delayLine.pushSample(2);
  delayLine.pushSample(3);
  delayLine.pushSample(4);

  delayLine.setDelay(100);
  ASSERT_FLOAT_EQ(0, delayLine.readSample());

  delayLine.setDelay(1.5f);
  ASSERT_FLOAT_EQ(2.5f, delayLine.readSample());

  delayLine.setDelay(2.3f);
  ASSERT_FLOAT_EQ(1.7f, delayLine.readSample());

  delayLine.setDelay(2.f);
  ASSERT_FLOAT_EQ(2.f, delayLine.readSample());
}
```

```cpp
TEST(Flanger, FractionalDelay) {
  DL::FractionalDelayLine delayLine;

  delayLine.pushSample(1);
  delayLine.pushSample(2);
  delayLine.pushSample(3);
  delayLine.pushSample(4);

  delayLine.setDelay(100);
  ASSERT_FLOAT_EQ(0, delayLine.readSample());

  delayLine.setDelay(1.5f);
  ASSERT_FLOAT_EQ(2.5f, delayLine.readSample());

  delayLine.setDelay(2.3f);
  ASSERT_FLOAT_EQ(1.7f, delayLine.readSample());

  delayLine.setDelay(2.f);
  ASSERT_FLOAT_EQ(2.f, delayLine.readSample()); // 0.f
}
```

What’s wrong with this code?

```
float FractionalDelayLine::readSample() {
  auto readHead = writeHead - 1 - delay;
  if (readHead < 0) {
    readHead += std::ssize(buffer);
  }
  const auto truncatedReadHead = static_cast<int>(std::floor(readHead));
  auto truncatedReadHeadPlusOne = static_cast<int>(std::ceil(readHead));
  const auto truncatedReadHeadWeight = std::abs(readHead - truncatedReadHeadPlusOne);
  const auto truncatedReadHeadPlusOneWeight = std::abs(readHead - truncatedReadHead);
  if (truncatedReadHeadPlusOne >= std::ssize(buffer)) {
    truncatedReadHeadPlusOne -= std::ssize(buffer);
  }
  const auto outputSample = truncatedReadHeadWeight * buffer[truncatedReadHead] +
                  truncatedReadHeadPlusOneWeight * buffer[truncatedReadHeadPlusOne];
  return outputSample;
}
```

What’s wrong with this code?

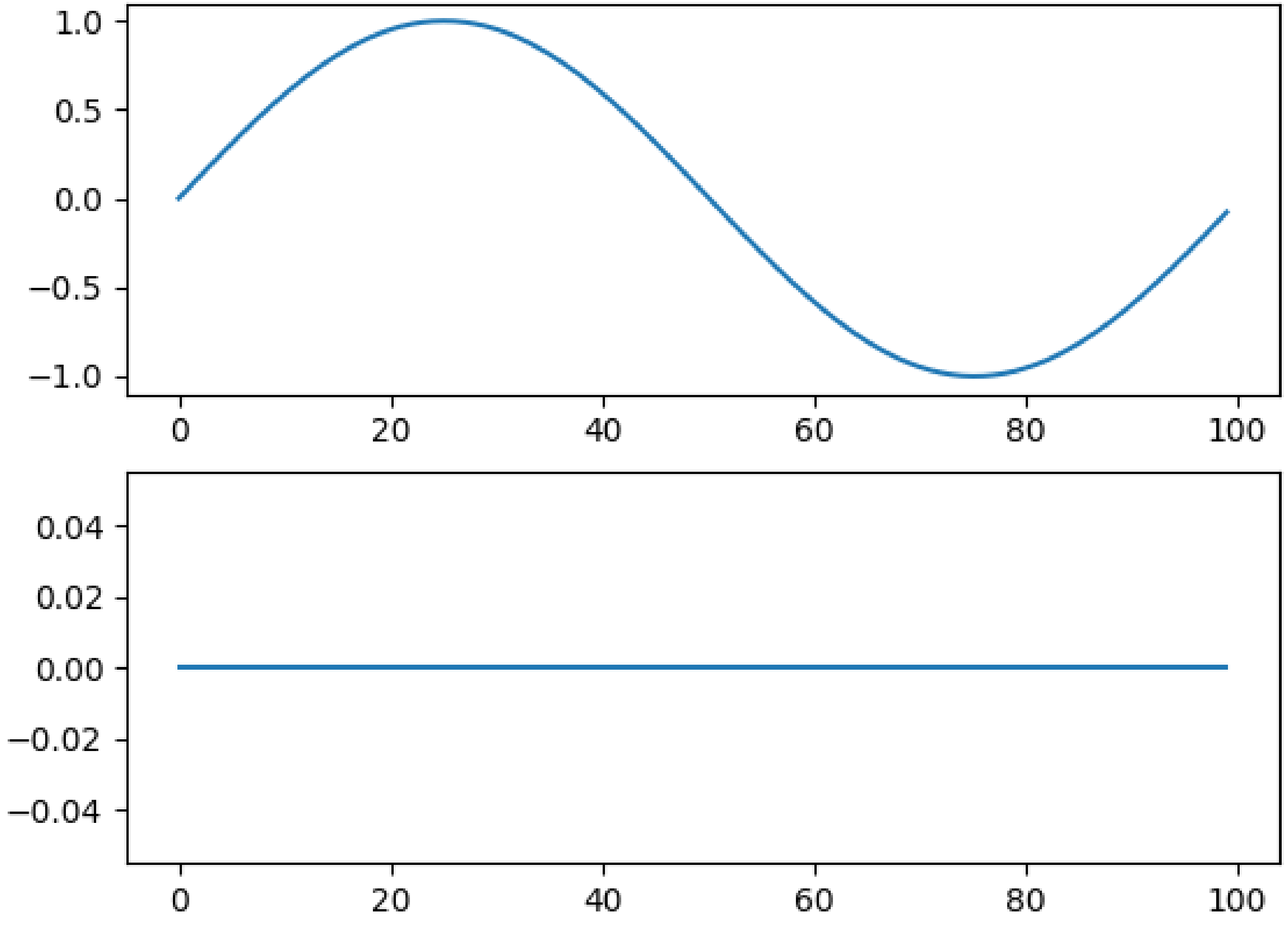
```
float FractionalDelayLine::readSample() {
  auto readHead = writeHead - 1 - delay;
  if (readHead < 0) {
    readHead += std::ssize(buffer);
  }
  const auto truncatedReadHead = static_cast<int>(std::floor(readHead));
  auto truncatedReadHeadPlusOne = truncatedReadHead + 1;
  const auto truncatedReadHeadWeight = std::abs(readHead - truncatedReadHeadPlusOne);
  const auto truncatedReadHeadPlusOneWeight = std::abs(readHead - truncatedReadHead);
  if (truncatedReadHeadPlusOne >= std::ssize(buffer)) {
    truncatedReadHeadPlusOne -= std::ssize(buffer);
  }
  const auto outputSample = truncatedReadHeadWeight * buffer[truncatedReadHead] +
                  truncatedReadHeadPlusOneWeight * buffer[truncatedReadHeadPlusOne];
  return outputSample;
}
```

What’s wrong with this code?

```
DataCallbackResult onAudioReady(AudioStream* audioStream,
                                void* audioData,
                                int32_t framesCount) {
  auto* floatData = reinterpret_cast<float*>(audioData);

  for (auto frame = 0; frame < framesCount; ++frame) {
    const auto sample = getSample();
    for (auto channel = 0; channel < channelCount; ++channel) {
      floatData[frame * channelCount + channelCount] = sample;
    }
  }
  return DataCallbackResult::Continue;
}
```

```cpp
TEST(OutputTest, AudioPlayerExample) {
  constexpr auto channelCount = 2;
  constexpr auto samplesPerChannel = 100;
  constexpr auto sampleRate = 44100.f;

  auto outputData = std::vector<float>(samplesPerChannel * channelCount,
                                       0.f);
  example::AudioPlayer audioPlayer{sampleRate, channelCount};

  audioPlayer.onAudioReady(nullptr,
                           reinterpret_cast<void*>(outputData.data()),
                             samplesPerChannel - 1);

  // write to audio_player_output.wav
  //...
}
```

```python
import matplotlib.pyplot as plt
import soundfile as sf

if __name__ == '__main__':
    root_repo_path = ...
    test_executable_path = ...
    ret = os.system(f'cd {root_repo_path} && cmake -S . -B build && cmake --build build &&'
                    f' {test_executable_path} --gtest_filter="OutputTest.*"')
    data, fs = sf.read(root_repo_path / 'audio_player_output.wav')

    plt.figure()
    plt.subplot(211)
    plt.plot(data[:, 0])
    plt.subplot(212)
    plt.plot(data[:, 1])
    plt.show()
```

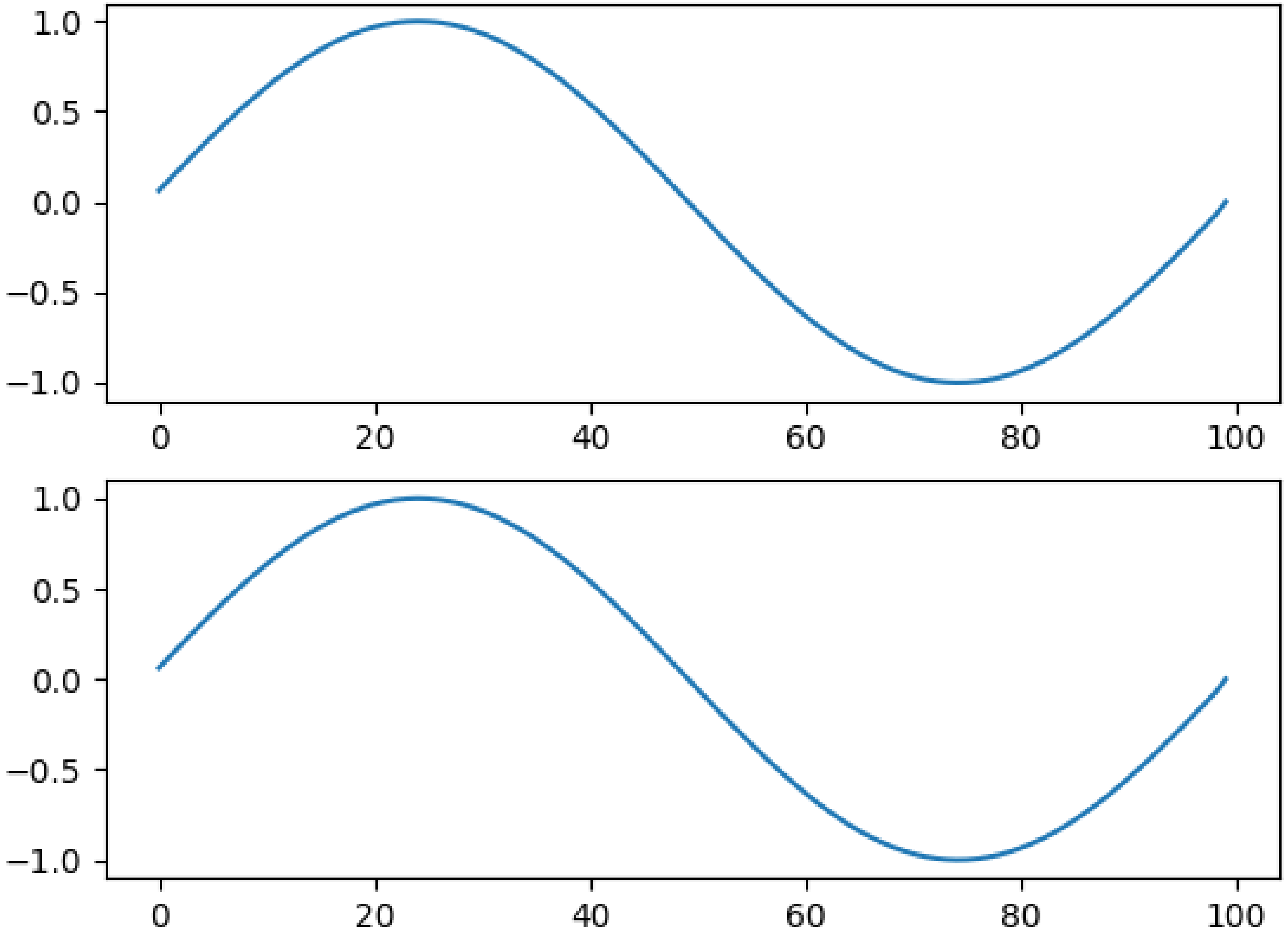

1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
0.04
0.02
0.00
−0.02
−0.04
0
20
40
60
80
100

What’s wrong with this code?

```
DataCallbackResult onAudioReady(AudioStream* audioStream,
                                void* audioData,
                                int32_t framesCount) {
  auto* floatData = reinterpret_cast<float*>(audioData);

  for (auto frame = 0; frame < framesCount; ++frame) {
    const auto sample = getSample();
    for (auto channel = 0; channel < channelCount; ++channel) {
      floatData[frame * channelCount + channelCount] = sample;
    }
  }
  return DataCallbackResult::Continue;
}
```

What’s wrong with this code?

```
DataCallbackResult onAudioReady(AudioStream* audioStream,
                                void* audioData,
                                int32_t framesCount) {
  auto* floatData = reinterpret_cast<float*>(audioData);

  for (auto frame = 0; frame < framesCount; ++frame) {
    const auto sample = getSample();
    for (auto channel = 0; channel < channelCount; ++channel) {
      floatData[frame * channelCount + channel] = sample;
    }
  }
  return DataCallbackResult::Continue;
}
```


1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100
1.0
0.5
0.0
−0.5
−1.0
0
20
40
60
80
100

Other safety measures

1. Testing for out-of-memory access: use at() instead of operator[] on a vector.
2. Use ranges, std::span, and std library algorithms to avoid out-of-range access.
3. Check original authors' implementations.
4. Check textbook sources.
5. Measure performance.
6. Use address sanitizer and undefined behavior sanitizer.
7. Know your DAW.

Summary: your action plan

1. Write reference tests for your software in the current state.
2. Write tests for newly developed audio features using the presented DSP principles.
3. Run these tests on a regular basis.
4. Enjoy fewer bugs in your software!

More resources

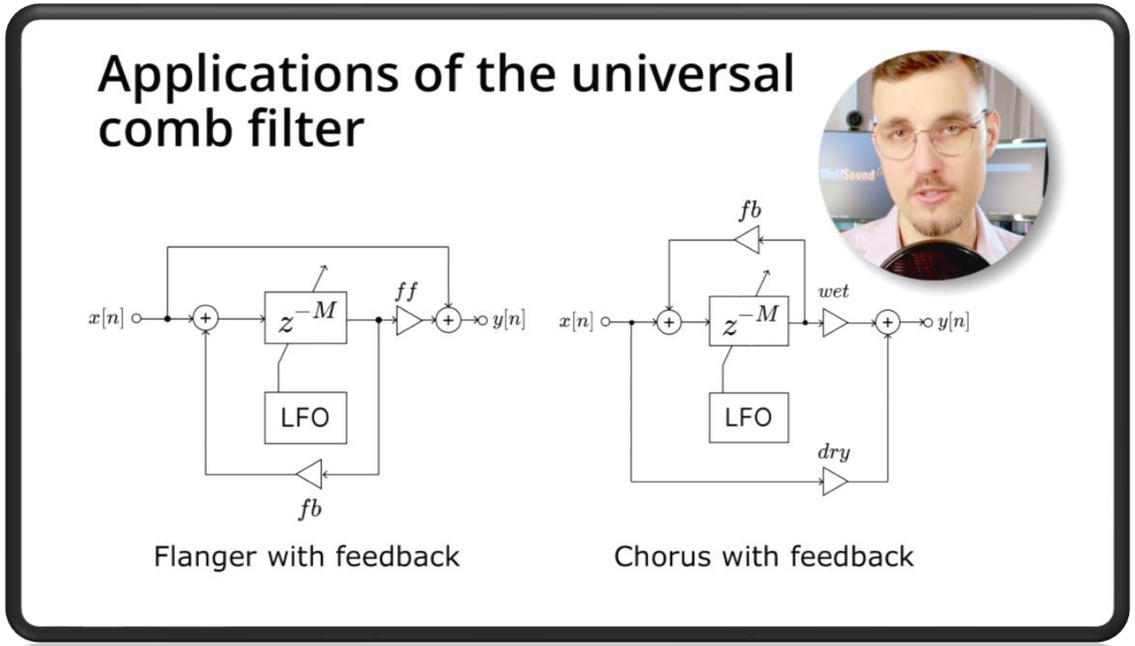
- Audio plugin template with unit tests set up
- Talk by Esa Jääskelä on unit testing
- Talk by Dave Rowland on optimization
- Other ADC talks

How to learn DSP principles?

- Book resources at: thewolfsound.com/resources

How to learn DSP principles?

- DSP Pro online course
- wolfsoundacademy.com/dsp-pro
- 10% off with discount code **ADC23**



DSPPro

Happy dspying!

- (Working) example code & slides: github.com/JanWilczek/adc23
- Contact: jan.wilczek@thewolfsound.com

WolfSound

Become an Audio Programmer