Article

Reducing spectral leakage with windowing

Multiply signal data by window sequence values when performing transforms with noninteger period signals.

Overview

Discrete Fourier and cosine transforms, which decompose a signal into its component frequencies and recreate a signal from a component frequency representation, work over vectors of specific lengths. For example, if you're analyzing audio data, the data might be represented as pages of 1024 samples. Discrete Fourier and cosine transforms can accurately approximate the component frequencies that have an integer number of periods — that is, signals where the start and end points join to form a continuous waveform.

However, with noninteger period signals, where the endpoints don't meet, the discontinuities appear as false frequency components in a forward transform. This smearing of data is called spectral leakage.

You can use an approach called windowing to reduce spectral leakage when performing transforms over data that includes noninteger period signals. *Windowing* multiplies a signal by a vector that represents a smooth curve with boundary values of zero or near zero. This technique ensures that the endpoints of a signal meet and reduces the discontinuities.

Synthesize a test signal

The code examples in this article synthesize the signal data from a series of sine waves. In a real-world app, you'll most likely acquire signal data from a sensor such as a microphone.

Use the `synthesizeSignal` function to generate a composite sine wave from a supplied array of component frequencies and amplitudes:

```
static func synthesizeSignal(frequencyAmplitudePairs: [(f: Float, a: Float)],
                             count: Int) -> [Float] {

    let tau: Float = .pi * 2
    let signal: [Float] = (0 ..< count).map { index in
        frequencyAmplitudePairs.reduce(0) { accumulator, frequenciesAmplitudePair in
            let normalizedIndex = Float(index) / Float(count)
            return accumulator + sin(normalizedIndex * frequenciesAmplitudePair.f *
        }
    }

    return signal
}
```

Create a signal with an integer number of periods

Using the code below, generate a Fourier series approximation of a square wave that's built from a series of sine waves. Each component sine wave has an integer number of periods over the length of the data.

```
let n = 2048

let baseFrequency: Float = 5

let frequencyAmplitudePairs = stride(from: 1, to: 50, by: 2).map { i in
    return(f: baseFrequency * Float(i), a: (1 / Float(i)))
}

var signal = synthesizeSignal(frequencyAmplitudePairs: frequencyAmplitudePairs,
                              count: n)
```

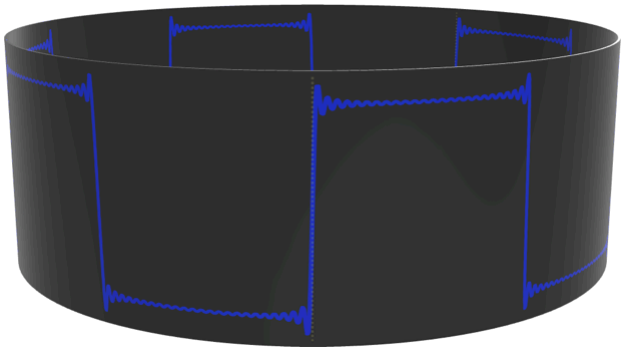
Use the vDSP fast Fourier transform (FFT), like in the example below, to compute the component frequencies of `signal`:

```
let count = n / 2
var realParts = [Float](repeating: 0,
                        count: count)
var imagParts = [Float](repeating: 0,
                        count: count)
```

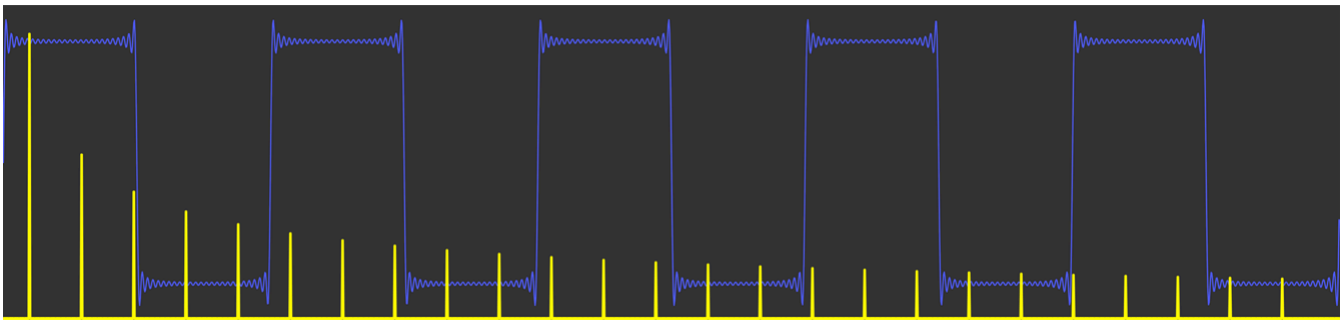
```
realParts.withUnsafeMutableBufferPointer { realPtr in
    imagParts.withUnsafeMutableBufferPointer { imagPtr in

        var complexSignal = DSPSplitComplex(realp: realPtr.baseAddress!,
                                            imagp: imagPtr.baseAddress!)

        signal.withUnsafeBytes {
            vDSP.convert(interleavedComplexVector: [DSPComplex]($0.bindMemory(to: DS
                         toSplitComplexVector: &complexSignal)
        }

        let log2n = vDSP_Length(log2(Float(n)))
        let fft = vDSP.FFT(log2n: log2n,
                           radix: .radix2,
                           ofType: DSPSplitComplex.self)

        fft?.forward(input: complexSignal,
                     output: &complexSignal)
    }
}
```

To learn more about computing the frequency components of a signal, see Finding the component frequencies in a composite sine wave.

The FFT treats the data set as a single period of a continuous signal. The visualization below wraps the signal around a virtual cylinder to illustrate how the FFT interprets the data. This figure also shows that the endpoints meet:



The illustration below shows a representation of the original signal in blue, and the imaginary parts of the frequency-domain data in yellow:



> **Note**
>
> The visualizations of the frequency-domain data in this article are transformed to improve visibility. Each visualization is actually the square root of the absolute value of each element of `imagParts`.

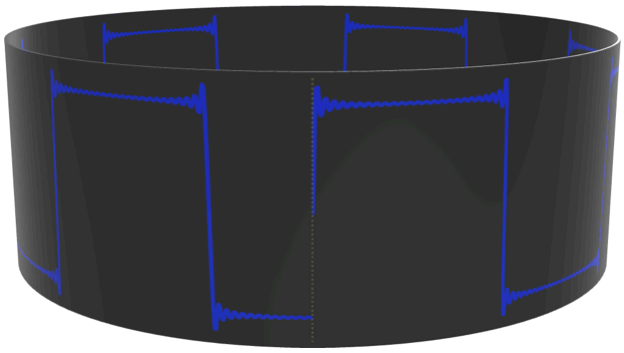
The FFT result shows that the signal is composed of 25 sine waves, represented as spikes in the graph.

Create a signal with a noninteger number of periods

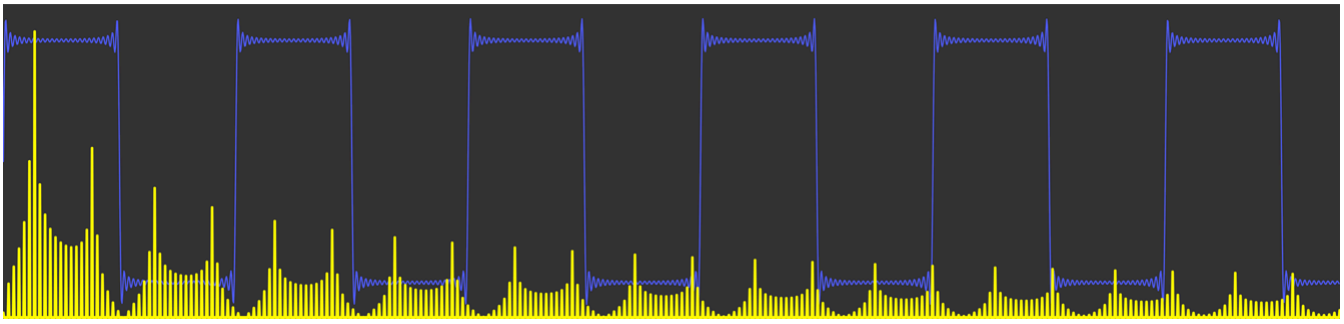
Use the code below to define a series of sine waves with noninteger periods:

```
let n = 2048

let baseFrequency: Float = 5.75

let frequencyAmplitudePairs = stride(from: 1, to: 50, by: 2).map { i in
    return(f: baseFrequency * Float(i), a: (1 / Float(i)))
}

var signal = synthesizeSignal(frequencyAmplitudePairs: frequencyAmplitudePairs,
                              count: n)
```

The visualization below wraps the noninteger-period signal around a virtual cylinder and shows the endpoint discontinuities:



The image below shows the results of a transform of this data. The results shows additional, intermediate values that are the result of spectral leakage.



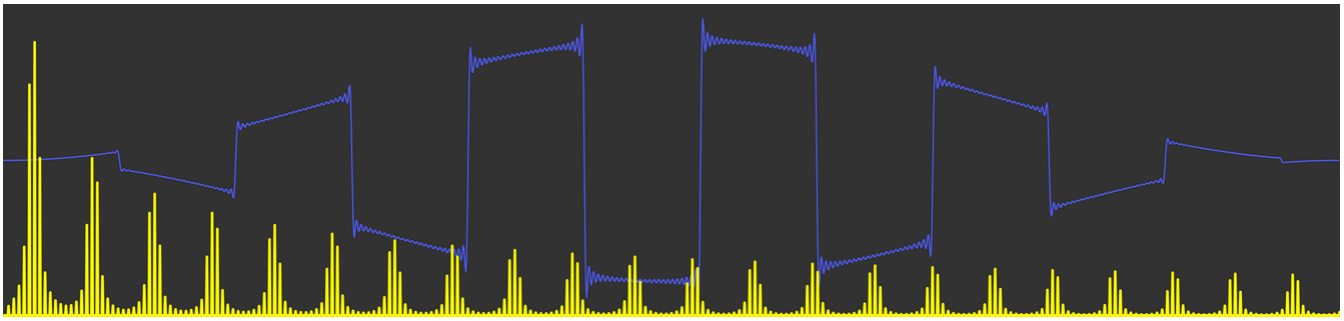
Create a windowed signal with a noninteger number of periods

The code below shows the same noninteger period signal, but in this example, you multiply the signal by the result of `window(ofType:usingSequence:count:isHalfWindow:)`:

```
let n = 2048

let baseFrequency: Float = 5.75

let frequencyAmplitudePairs = stride(from: 1, to: 50, by: 2).map { i in
    return(f: baseFrequency * Float(i), a: (1 / Float(i)))
}

var signal = synthesizeSignal(frequencyAmplitudePairs: frequencyAmplitudePairs,
                              count: n)

let window = vDSP.window(ofType: Float.self,
                         usingSequence: .hanningDenormalized,
                         count: n,
                         isHalfWindow: false)
```

```
signal = vDSP.multiply(signal, window)
```

The illustration below shows the windowed signal in blue, with its boundaries tapered toward zero, and the transformed version with reduced spectral leakage in yellow:



Select a window sequence

vDSP provides functions for generating three different windows:

Hann
: A great-general purpose window that reduces spectral leakage.

Hamming
: Provides better discrimination of component sine waves with close frequencies.

Blackman
: Reduces spectral leakage away from the main frequency compared to Hann and Hamming, but has a wider main peak than Hann.

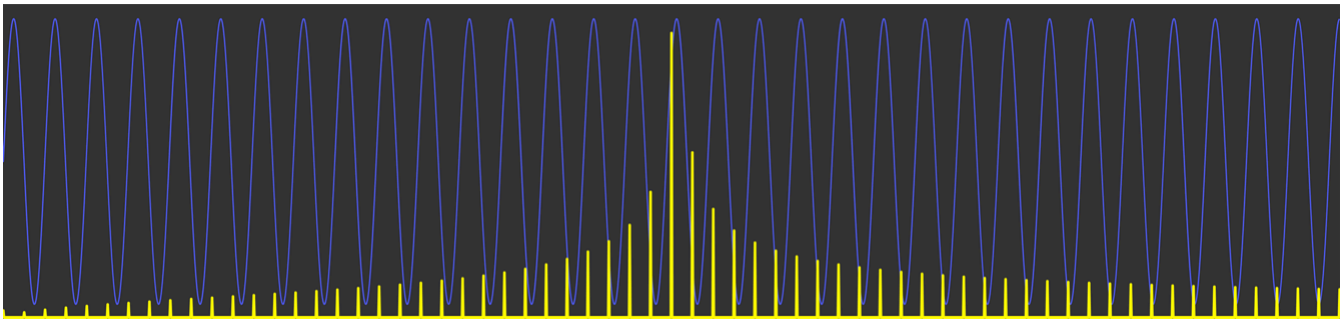
The image below provides a visual comparison of the different window sequence types:

Create a sine wave with a noninteger period

To understand the different effects of the different windows provided by vDSP, create a signal that's composed of a signal sine wave with a noninteger period:

```
let frequencyAmplitudePairs = [(f: Float(32.25), a: Float(1))]
```

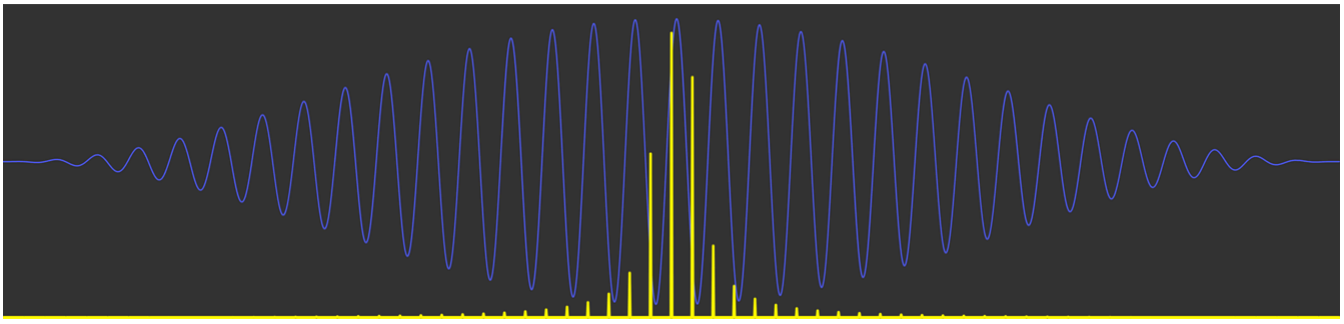
The illustration below shows the sine wave and the frequency-domain result:



Spectral leakage is apparent throughout the rendered FFT result.

Reduce the spectral leakage by using a Hann window

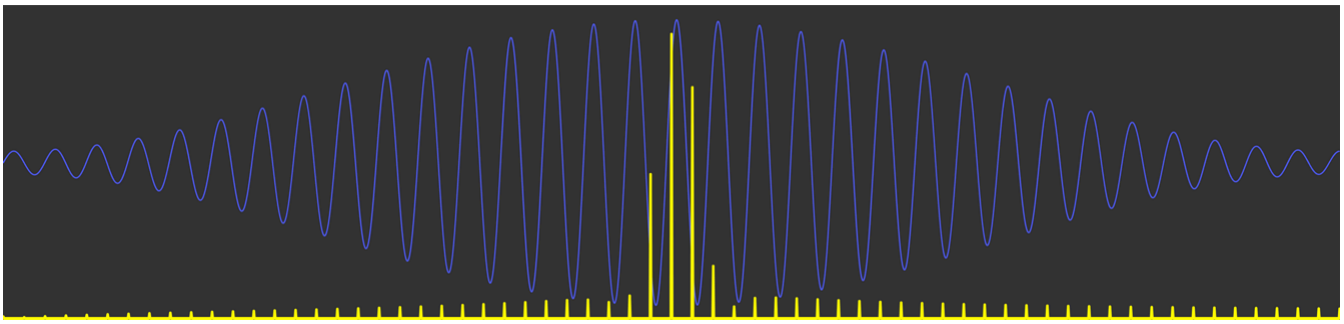
The illustration below shows the time- and frequency-domain representations of the noninteger period sine wave with the Hann window applied:



Reduce the spectral leakage by using a Hamming window

Create a Hamming window by passing `vDSP.WindowSequence.hamming` to the `window(ofType:usingSequence:count:isHalfWindow:)` function. Unlike the Hann window, the Hamming window doesn't reach zero at its boundaries.

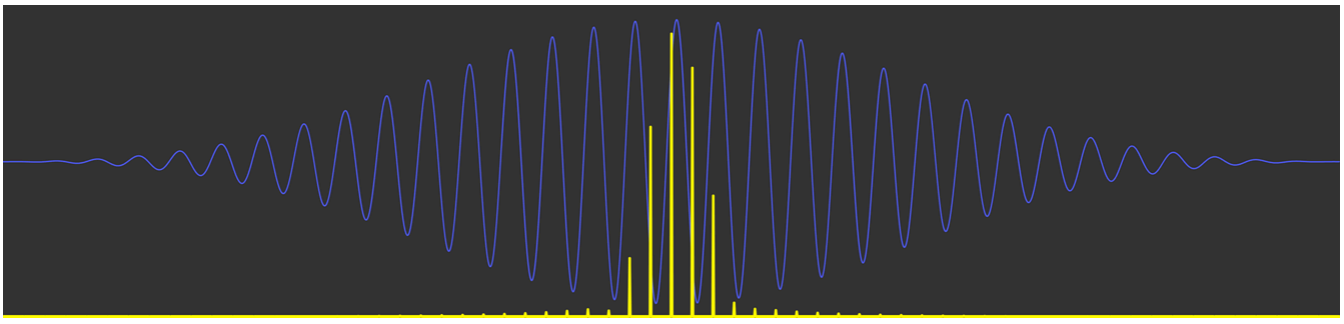
The figure below shows the result of multiplying the signal by a Hamming window: high values around the base frequency in the forward FFT are tighter than the Hann-windowed result, but there's low-level spectral leakage across the entire forward FFT:



Reduce the spectral leakage by using a Blackman window

Create a Blackman window by passing `vDSP.WindowSequence.blackman` to the `window(ofType:usingSequence:count:isHalfWindow:)` function.

The illustration below shows the time- and frequency-domain representations of the noninteger period sine wave with the Blackman window applied:



See Also

Fourier and Cosine Transforms

- Understanding data packing for Fourier transforms
  Format source data for the vDSP Fourier functions, and interpret the results.
- Finding the component frequencies in a composite sine wave
  Use 1D fast Fourier transform to compute the frequency components of a signal.
- Performing Fourier transforms on interleaved-complex data
  Optimize discrete Fourier transform (DFT) performance with the vDSP interleaved DFT routines.
- Signal extraction from noise
  Use Accelerate's discrete cosine transform to remove noise from a signal.

Performing Fourier Transforms on Multiple Signals

Use Accelerate's multiple-signal fast Fourier transform (FFT) functions to transform multiple signals with a single function call.

Halftone descreening with 2D fast Fourier transform

Reduce or remove periodic artifacts from images.

Fast Fourier transforms

Transform vectors and matrices of temporal and spatial domain complex values to the frequency domain, and vice versa.

Discrete Fourier transforms

Transform vectors of temporal and spatial domain complex values to the frequency domain, and vice versa.

Discrete Cosine transforms

Transform vectors of temporal and spatial domain real values to the frequency domain, and vice versa.