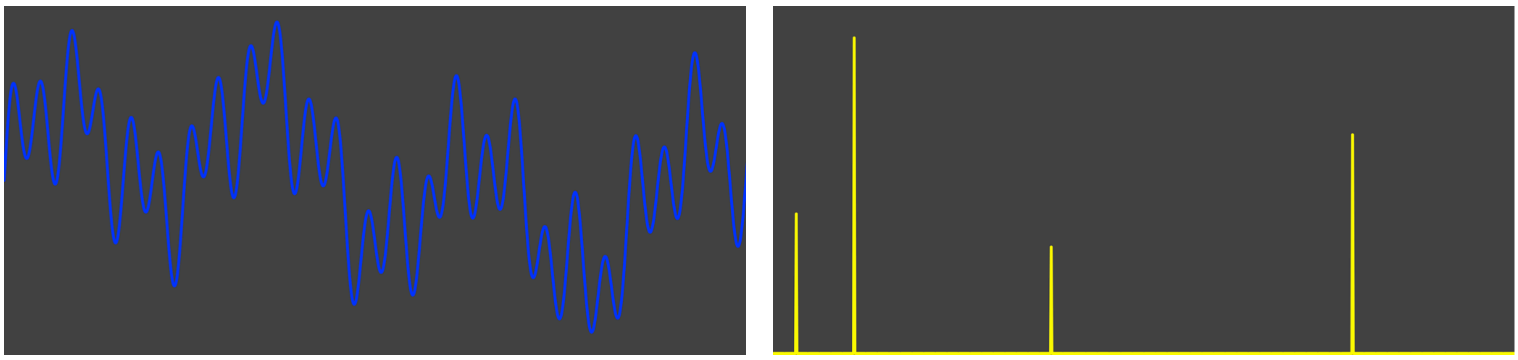
Article

Finding the component frequencies in a composite sine wave

Use 1D fast Fourier transform to compute the frequency components of a signal.

Overview

Accelerate's vDSP module provides functions to perform 1D fast Fourier transforms (FFTs) on vectors of data, such as audio signals. The example below shows an input signal (left) and its frequency domain representation (right) after transforming the signal with a forward FFT.



You can inspect the frequency-domain data of a forward FFT to compute the individual sine wave components of a composite wave. The technique described in this article is applicable to many digital signal processing applications, for example, finding the dominant frequencies in a dual-tone multi-frequency (DTMF) signal or removing noise from a signal.

Synthesize a test signal

The function below generates 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 the composite signal

Create an array that contains frequency-amplitude tuples. You define the frequencies as the number of cycles per n. The highest measurable frequency, known as the Nyquist frequency, is the element with index `n/2`, which is `1023` in a zero-based array that contains `2048` elements.

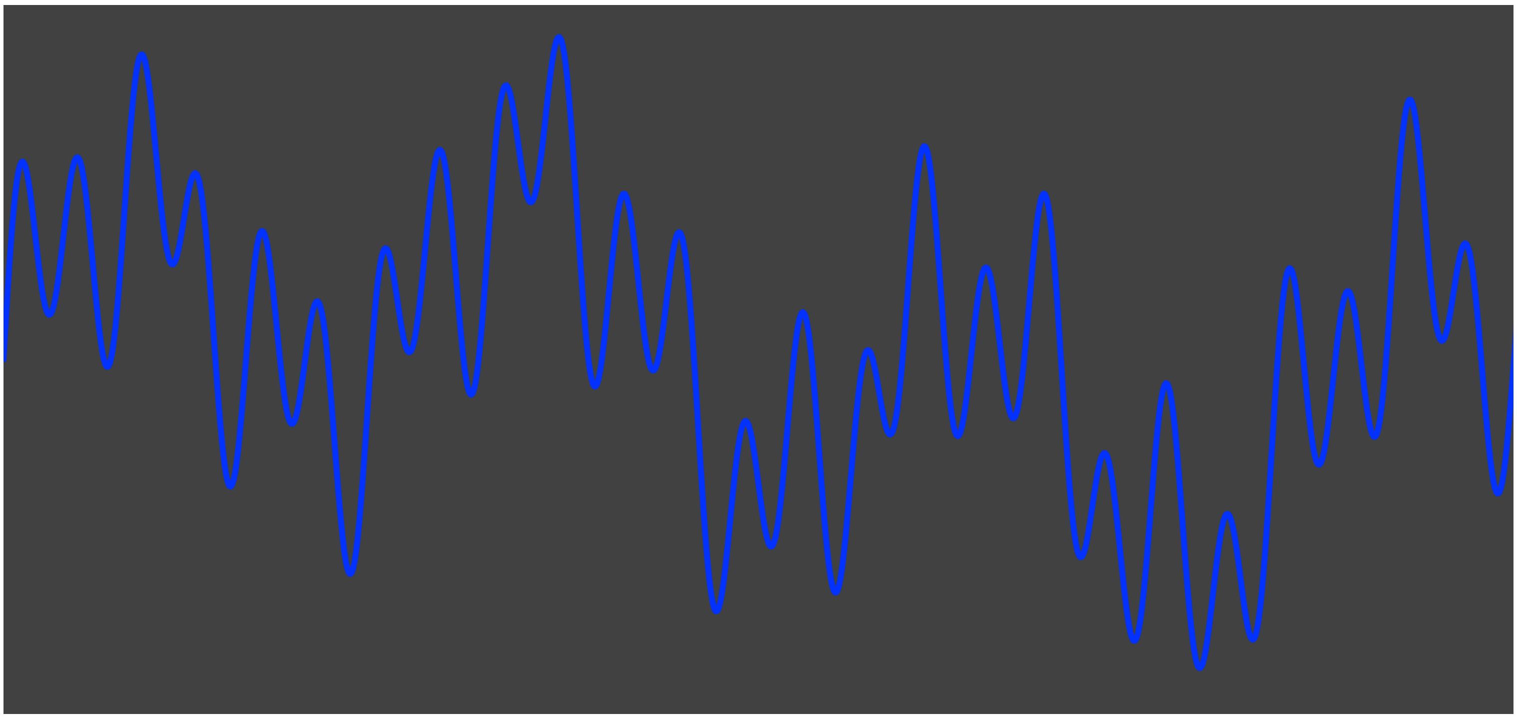
The code below creates the array, `signal`, that contains four component sine waves:

```
let n = vDSP_Length(2048)

let frequencyAmplitudePairs = [(f: Float(2), a: Float(0.8)),
                                (f: Float(7), a: Float(1.2)),
                                (f: Float(24), a: Float(0.7)),
                                (f: Float(50), a: Float(1.0))]

let signal = synthesizeSignal(frequencyAmplitudePairs: frequencyAmplitudePairs,
                              count: Int(n))
```

The image below is a visualization of composite sine waves in `signal`:



Create the FFT setup

Create a setup object that contains a precalculated weights array of complex exponentials required to perform the FFT operations. The values in the weights array simplify the FFT calculation. Creating this setup object can be expensive, so do it only once, for example, when starting your app. After creating the setup object, you can reuse it later.

The code below creates a setup object suitable for performing forward and inverse 1D FFTs on a signal containing n elements:

```
let log2n = vDSP_Length(log2(Float(n)))

guard let fftSetUp = vDSP.FFT(log2n: log2n,
                              radix: .radix2,
                              ofType: DSPSplitComplex.self) else {
                                fatalError("Can't create FFT Setup.")
}
```

You can use this setup object for similarly sized smaller FFTs. However, using a weights array built for an FFT that processes a large number of elements can degrade performance for an FFT that processes a significantly smaller number of elements.

Create the source and destination arrays for the forward FFT

The FFT operates on complex numbers. That is, it operates on numbers that contain a real part and an imaginary part. Create two arrays — one for the real parts and one for the imaginary parts — for the input and output to the FFT operation:

```
let halfN = Int(n / 2)

var forwardInputReal = [Float](repeating: 0,
                               count: halfN)
var forwardInputImag = [Float](repeating: 0,
                               count: halfN)
var forwardOutputReal = [Float](repeating: 0,
                                count: halfN)
var forwardOutputImag = [Float](repeating: 0,
                                count: halfN)
```

Because each complex value stores two real values, the length of each array is half that of `signal`.

Perform the forward FFT

You use `DSPSplitComplex` structures to pass the separate real and imaginary arrays of the input and the output data to the FFT transform function.

The steps below perform the forward FFT:

1. Create a `DSPSplitComplex` structure to store a copy of `signal` that's represented as complex numbers.
2. Use `convert(interleavedComplexVector:toSplitComplexVector:)` to convert the real values in `signal` to complex numbers. The conversion stores the even values in `signal` as the real components in `forwardInput`, and the odd values in `signal` as the imaginary components in `forwardInput`.
3. Create a `DSPSplitComplex` structure with pointers to `forwardOutputReal` and `forwardOutputImag` to receive the FFT result.
4. Perform the forward FFT.

The code below shows how to perform the forward FFT using the steps described above:

```
forwardInputReal.withUnsafeMutableBufferPointer { forwardInputRealPtr in
    forwardInputImag.withUnsafeMutableBufferPointer { forwardInputImagPtr in
        forwardOutputReal.withUnsafeMutableBufferPointer { forwardOutputRealPtr in
            forwardOutputImag.withUnsafeMutableBufferPointer { forwardOutputImagPtr
```

```
                // Create a `DSPSplitComplex` to contain the signal.
                var forwardInput = DSPSplitComplex(realp: forwardInputRealPtr.baseAc
                                                   imagp: forwardInputImagPtr.baseAc

                // Convert the real values in `signal` to complex numbers.
                signal.withUnsafeBytes {
                    vDSP.convert(interleavedComplexVector: [DSPComplex]($0.bindMemor
                                 toSplitComplexVector: &forwardInput)
                }

                // Create a `DSPSplitComplex` to receive the FFT result.
                var forwardOutput = DSPSplitComplex(realp: forwardOutputRealPtr.base
                                                    imagp: forwardOutputImagPtr.base

                // Perform the forward FFT.
                fftSetUp.forward(input: forwardInput,
                                 output: &forwardOutput)
            }
        }
    }
}
```

On return, `forwardOutputReal` contains the real parts of the forward FFT, and `forwardOutputImag` contains the imaginary parts of the frequency-domain representation of the original signal.

Compute component frequencies in the frequency-domain data

Use the vDSP_zaspec function to compute the autospectrum of the frequency-domain data in the `forwardOutputReal` and `forwardOutputImag` arrays. The autospectrum is the sum of squares of the complex and real parts of each complex frequency-domain element. The code below computes the autospectrum:

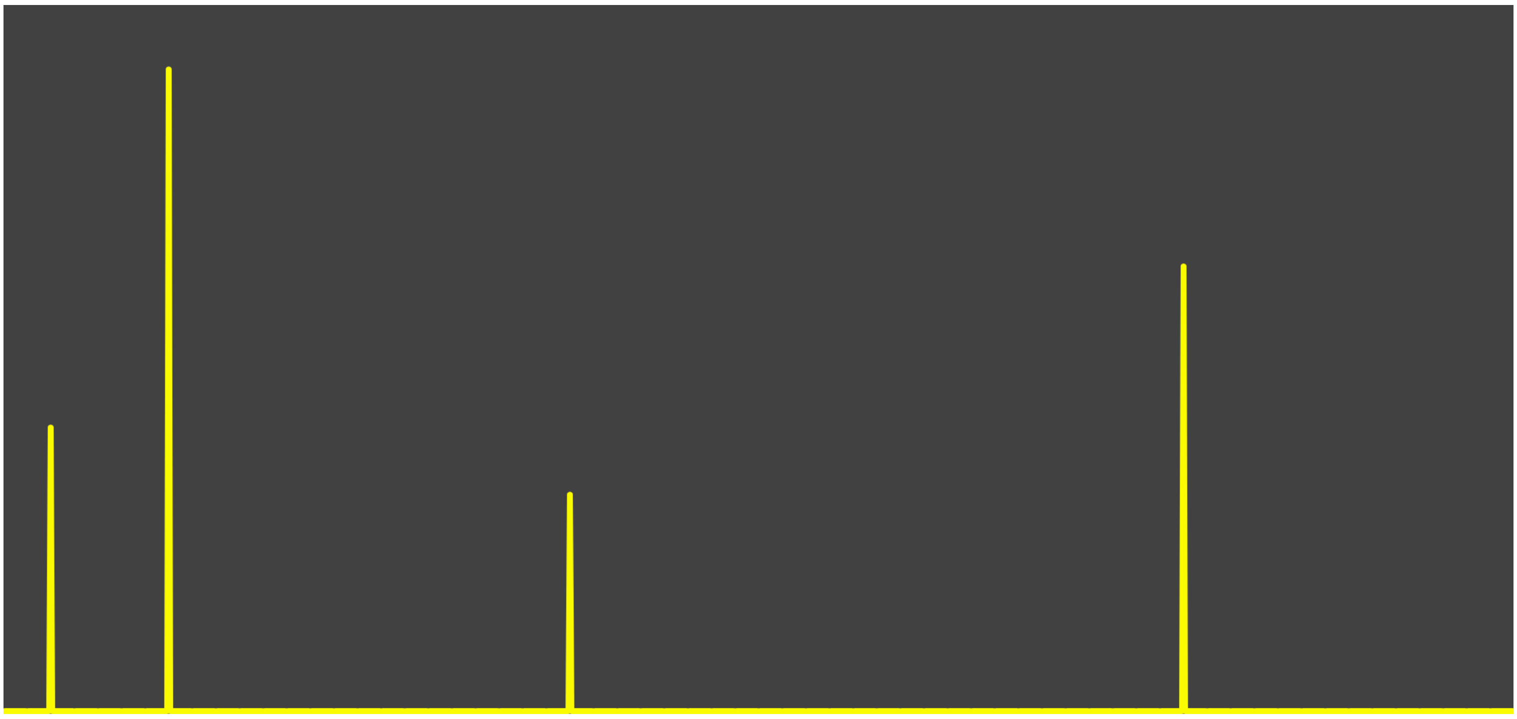
```
let autospectrum = [Float](unsafeUninitializedCapacity: halfN) {
    autospectrumBuffer, initializedCount in

    // The `vDSP_zaspec` function accumulates its output. Clear the
    // uninitialized `autospectrumBuffer` before computing the spectrum.
    vDSP.clear(&autospectrumBuffer)
    forwardOutputReal.withUnsafeMutableBufferPointer { forwardOutputRealPtr in
        forwardOutputImag.withUnsafeMutableBufferPointer { forwardOutputImagPtr in

            var frequencyDomain = DSPSplitComplex(realp: forwardOutputRealPtr.baseAd
                                                  imagp: forwardOutputImagPtr.baseAd

            vDSP_zaspec(&frequencyDomain,
                        autospectrumBuffer.baseAddress!,
                        vDSP_Length(halfN))
        }
    }
    initializedCount = halfN
```

The autospectrum of the forward FFT contains a series of high-magnitude items, rendered as vertical lines in the graph below:



The autospectrum values correspond to the frequencies and amplitudes you specified in the `frequencies` array. The code below scales the amplitudes to consider the autospectrum calculation and the inverse-transform step. To learn more about scaling time- and frequency-domain data, see Understanding data packing for Fourier transforms.

```
let componentFrequencyAmplitudePairs = autospectrum.enumerated().filter {
    $0.element > 1
}.map {
    return ($0.offset, sqrt($0.element) / Float(n))
}
```

```
// Prints:
//     ["frequency: 2 | amplitude: 0.80", "frequency: 7 | amplitude: 1.20",
//      "frequency: 24 | amplitude: 0.70", "frequency: 50 | amplitude: 1.00"]

print(componentFrequencyAmplitudePairs.map {
    "frequency: \($0.0) | amplitude: \(String(format: "%.2f", $0.1))"
})
```

Recreate the original signal

Use an inverse FFT to recreate a signal in the time domain, using the frequency-domain data returned by the forward FFT.

The steps below perform the inverse FFT:

1. Create the source of the inverse FFT, with pointers to `forwardOutputReal` and `forward OutputImag`.
2. Create a `DSPSplitComplex` structure to receive the FFT result.
3. Perform the inverse FFT.
4. Return an array of real values from the FFT result. Because the forward transform has a scaling factor of 2 and the inverse transform has a scaling factor of the number of items, divide each result by 2 `*` n:

```
var inverseOutputReal = [Float](repeating: 0,
                                count: halfN)
var inverseOutputImag = [Float](repeating: 0,
                                count: halfN)

let recreatedSignal: [Float] = forwardOutputReal.withUnsafeMutableBufferPointer { fo
    forwardOutputImag.withUnsafeMutableBufferPointer { forwardOutputImagPtr in
        inverseOutputReal.withUnsafeMutableBufferPointer { inverseOutputRealPtr in
            inverseOutputImag.withUnsafeMutableBufferPointer { inverseOutputImagPtr

                // Create a `DSPSplitComplex` that contains the frequency-domain dat
                let forwardOutput = DSPSplitComplex(realp: forwardOutputRealPtr.base
                                                    imagp: forwardOutputImagPtr.base

                // Create a `DSPSplitComplex` structure to receive the FFT result.
                var inverseOutput = DSPSplitComplex(realp: inverseOutputRealPtr.base
                                                    imagp: inverseOutputImagPtr.base
                // Perform the inverse FFT.
                fftSetUp.inverse(input: forwardOutput,
                                 output: &inverseOutput)

                // Return an array of real values from the FFT result.
                let scale = 1 / Float(n * 2)
                return [Float](fromSplitComplex: inverseOutput,
                               scale: scale,
                               count: Int(n))
            }
        }
    }
}
```

On return, `recreatedSignal` is approximately equal to `signal`.

See Also

Fourier and Cosine Transforms

Understanding data packing for Fourier transforms

Format source data for the vDSP Fourier functions, and interpret the results.

Performing Fourier transforms on interleaved-complex data

Optimize discrete Fourier transform (DFT) performance with the vDSP interleaved DFT routines.

Reducing spectral leakage with windowing

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

{} 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.