Article

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.

Overview

vDSP provides functions for performing fast Fourier transforms (FFTs) on multiple signals with a single function call. Transforming multiple signals is suited to processing stereo audio data or data that's aquired from multiple sources.

Create a Composite Sine Wave

The examples in this article use the following function to create an array with values that represent a composite sine wave:

```
/// Returns an array that contains a composite sine wave from the
/// specified frequency-amplitude pairs.
static func makeCompositeSineWave(from frequencyAmplitudePairs: [(f: Float,
                                                                  a: Float)],
                                  count: Int) -> [Float] {


    return [Float](unsafeUninitializedCapacity: count) {
        buffer, initializedCount in

        /// Fill the buffer with zeros.
        vDSP.fill(&buffer, with: 0)
        /// Create a reusable array to store the sine wave for each iteration.
```

```
        var iterationValues = [Float](repeating: 0, count: count)

        for frequencyAmplitudePair in frequencyAmplitudePairs {
            /// Fill the working array with a ramp in the range `0 ..< frequency`.
            vDSP.formRamp(withInitialValue: 0,
                          increment: frequencyAmplitudePair.f / Float(count / 2),
                          result: &iterationValues)
            /// Compute `sin(x * .pi)` for each element.
            vForce.sinPi(iterationValues, result: &iterationValues)
            if frequencyAmplitudePair.a != 1 {
                /// Mulitply each element by the specified amplitude.
                vDSP.multiply(frequencyAmplitudePair.a, iterationValues,
                              result: &iterationValues)
            }
            /// Add this sine wave iteration to the composite sine wave accumulator.
            vDSP.add(iterationValues, buffer, result: &buffer)
        }

        initializedCount = count
    }
}
```

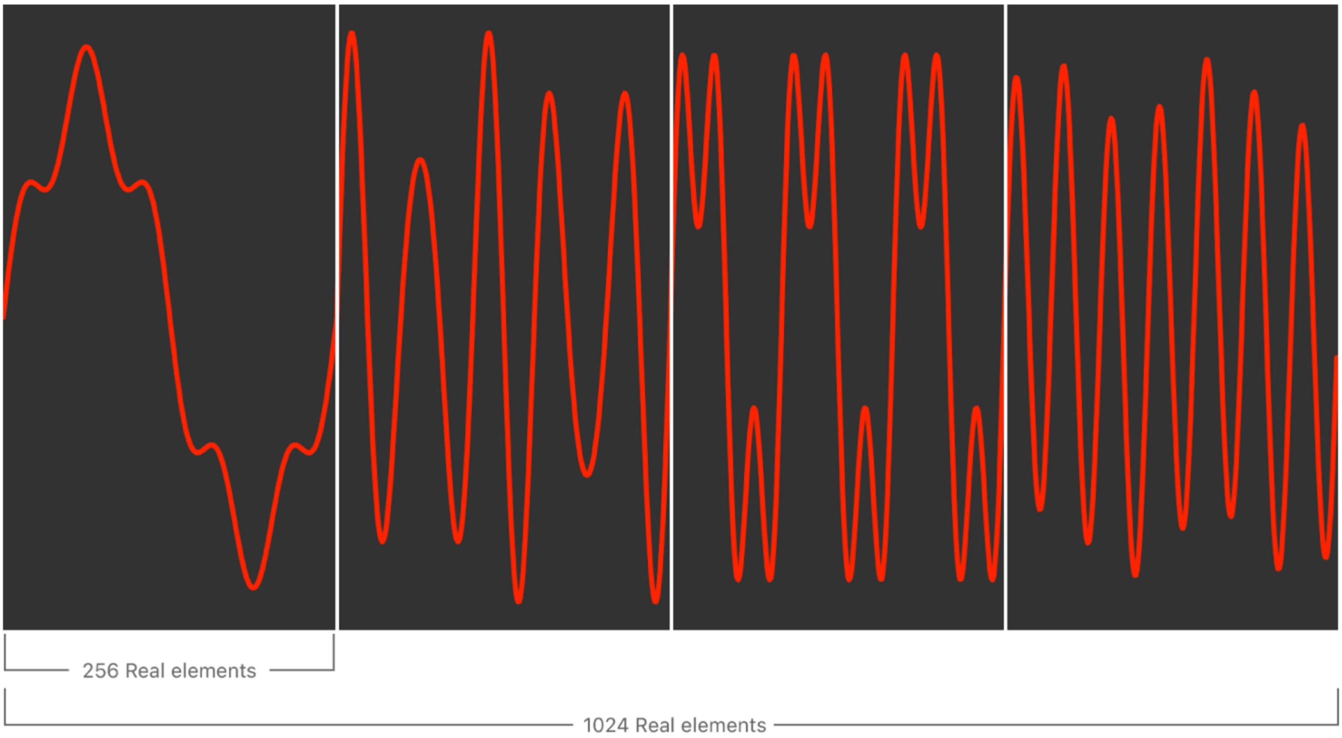
Perform FFT on Multiple Real Signals

The vDSP multiple-signal FFT functions accept multiple signals concatenated together. The following code creates a single 1024 element array from four separate composite sine waves:

```
let realValuesCount = 256

let signal: [Float] = {
    let signal0 = makeCompositeSineWave(from: [(f: 1, a: 1),
                                               (f: 5, a: 0.2)],
                                        count: realValuesCount)

    let signal1 = makeCompositeSineWave(from: [(f: 5, a: 1),
                                               (f: 7, a: 0.3)],
                                        count: realValuesCount)

    let signal2 = makeCompositeSineWave(from: [(f: 3, a: 1),
                                               (f: 9, a: 0.6)],
                                        count: realValuesCount)
```

```
    let signal3 = makeCompositeSineWave(from: [(f: 7, a: 1),
                                               (f: 2, a: 0.15)],
                                        count: realValuesCount)

    return signal0 + signal1 + signal2 + signal3
}()
```

The following image is a visualization of the values in `signal`:



The vDSP FFT and DFT functions work with data in split-complex format. Split-complex format stores the real and imaginary parts of complex numbers in the corresponding elements of two separate arrays.

Use the vDSP_ctoz function to convert the real values in the signal array to split-complex format. The vDSP_ctoz function transforms the real values so that the real array contains even elements, and the imaginary array contains odd elements.

```
let complexValuesCount = signal.count / 2

var complexReals = [Float]()
var complexImaginaries = [Float]()

signal.withUnsafeBytes { signalPtr in
    complexReals = [Float](unsafeUninitializedCapacity: complexValuesCount) {
        realBuffer, realInitializedCount in
        complexImaginaries = [Float](unsafeUninitializedCapacity: complexValuesCount
            imagBuffer, imagInitializedCount in
```

```
            var splitComplex = DSPSplitComplex(realp: realBuffer.baseAddress!,
                                               imagp: imagBuffer.baseAddress!)

            vDSP_ctoz([DSPComplex](signalPtr.bindMemory(to: DSPComplex.self)), 2,
                      &splitComplex, 1,
                      vDSP_Length(complexValuesCount))

            imagInitializedCount = complexValuesCount
        }
        realInitializedCount = complexValuesCount
    }
}
```

The vDSP_fftm_zrip function performs the FFT. Create a DSPSplitComplex structure that acts as a mediatory between the real and imaginary arrays and the FFT function. The third parameter to vDSP_fftm_zrip (the stride between the individual signals) is measured in complex elements.

```
let signalCount = 4

complexReals.withUnsafeMutableBufferPointer { realPtr in
    complexImaginaries.withUnsafeMutableBufferPointer { imagPtr in
        var splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                           imagp: imagPtr.baseAddress!)

        let log2n = vDSP_Length(log2(Float(realValuesCount)))
        if let fft = vDSP_create_fftsetup(log2n, FFTRadix(kFFTRadix2)) {

            vDSP_fftm_zrip(fft,
                           &splitComplex, 1,
                           vDSP_Stride(realValuesCount / 2),
                           log2n,
                           vDSP_Length(signalCount),
                           FFTDirection(kFFTDirection_Forward))

            vDSP_destroy_fftsetup(fft)
        }
    }
}
```

On return, `complexReals` and `complexImaginaries` contain the frequency-domain representation of the four real signals. Call squareMagnitudes(_:result:) to compute the energy at each frequency.

```
let magnitudes = [Float](unsafeUninitializedCapacity: complexValuesCount) {
    buffer, initializedCount in
    complexReals.withUnsafeMutableBufferPointer { realPtr in
        complexImaginaries.withUnsafeMutableBufferPointer { imagPtr in
            
            let splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                               imagp: imagPtr.baseAddress!)
            
            vDSP.squareMagnitudes(splitComplex,
                                  result: &buffer)
        }
    }
    
    initializedCount = complexValuesCount
}
```

Use the magnitudes information to calculate the component frequencies of each of the four signals. The offset of each nonzero magnitude represents the frequency, and the value represents the energy.

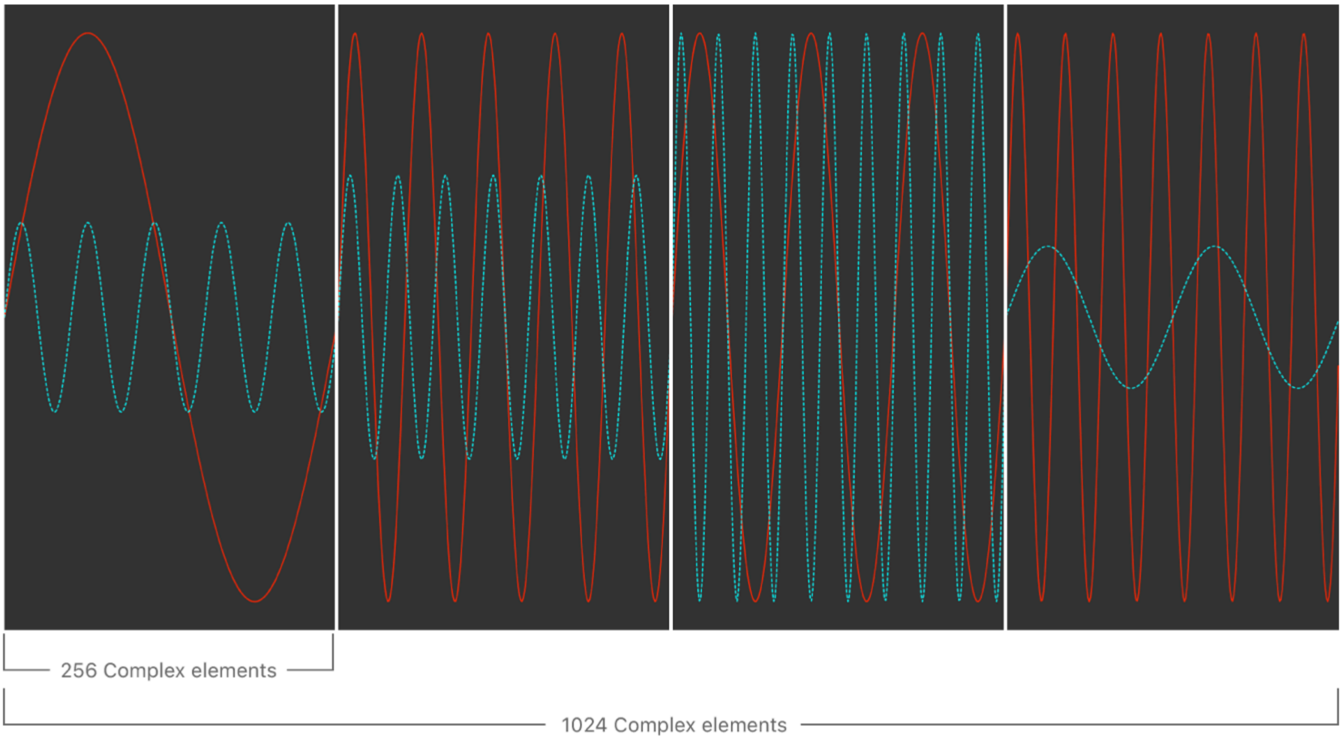
```
for i in 0 ..< signalCount {
    let start = i * (realValuesCount / 2)
    let end = start + (realValuesCount / 2) - 1
    
    let signalMagnitudes = magnitudes[start ..< end]
    
    let components = signalMagnitudes.enumerated().filter {
        $0.element > sqrt(.ulpOfOne)
    }
    
    // Prints
    //  [(offset: 1, element: 65536.0), (offset: 5, element: 2621.4412)]
    //  [(offset: 5, element: 65536.016), (offset: 7, element: 5898.24)]
    //  [(offset: 3, element: 65536.0), (offset: 9, element: 23592.96)]
    //  [(offset: 2, element: 1474.56), (offset: 7, element: 65536.0)]
    print(components)
}
```

Perform FFT on Multiple Complex Signals

A complex signal contains two real signals, one in the real parts and one in the imaginary parts. The following code creates two 1024-element arrays that contain the real and imaginary parts of four 256-element complex signals:

```
let complexValuesCount = 256

var realSignal: [Float] = {
    let signal0 = makeCompositeSineWave(from: [(f: 1, a: 1)],
                                        count: complexValuesCount)

    let signal1 = makeCompositeSineWave(from: [(f: 5, a: 1)],
                                        count: complexValuesCount)

    let signal2 = makeCompositeSineWave(from: [(f: 3, a: 1)],
                                        count: complexValuesCount)

    let signal3 = makeCompositeSineWave(from: [(f: 7, a: 1)],
                                        count: complexValuesCount)

    return signal0 + signal1 + signal2 + signal3
}()

var imaginarySignal: [Float] = {
    let signal0 = makeCompositeSineWave(from: [(f: 5, a: 0.2)],
                                        count: complexValuesCount)

    let signal1 = makeCompositeSineWave(from: [(f: 7, a: 0.3)],
                                        count: complexValuesCount)

    let signal2 = makeCompositeSineWave(from: [(f: 9, a: 0.6)],
                                        count: complexValuesCount)

    let signal3 = makeCompositeSineWave(from: [(f: 2, a: 0.15)],
                                        count: complexValuesCount)

    return signal0 + signal1 + signal2 + signal3
}()
```

The following image is a visualization of the values in `realSignal` as a solid line and the values in `imaginarySignal` as a dashed line:



The vDSP_fftm_zip function performs the FFT in-place on the real and imaginary arrays.

```
let signalCount = 4

realSignal.withUnsafeMutableBufferPointer { realPtr in
    imaginarySignal.withUnsafeMutableBufferPointer { imagPtr in
        var splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                           imagp: imagPtr.baseAddress!)

        let log2n = vDSP_Length(log2(Float(complexValuesCount)))
        if let fft = vDSP_create_fftsetup(log2n, FFTRadix(kFFTRadix2)) {

            vDSP_fftm_zip(fft,
                          &splitComplex, 1,
                          vDSP_Stride(complexValuesCount),
                          log2n,
                          vDSP_Length(signalCount),
                          FFTDirection(kFFTDirection_Forward))

            vDSP_destroy_fftsetup(fft)
        }
    }
}
```

On return, `realSignal` and `imaginarySignal` contain the frequency-domain representation of the four complex signals. Call squareMagnitudes(_:result:) to compute the energy at each frequency.

```
let magnitudesCount = complexValuesCount * signalCount
let magnitudes = [Float](unsafeUninitializedCapacity: magnitudesCount) {
    buffer, initializedCount in
    realSignal.withUnsafeMutableBufferPointer { realPtr in
        imaginarySignal.withUnsafeMutableBufferPointer { imagPtr in

            let splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                               imagp: imagPtr.baseAddress!)

            vDSP.squareMagnitudes(splitComplex,
                                  result: &buffer)
        }
    }

    initializedCount = magnitudesCount
}
```

Use the magnitudes information to calculate the component frequencies of each of the four signals. The offset of each nonzero magnitude represents the frequency, and the value represents the energy.

```
for i in 0 ..< signalCount {
    let start = i * (complexValuesCount)
    let end = start + (complexValuesCount / 2) - 1

    let signalMagnitudes = magnitudes[start ..< end]

    let components = signalMagnitudes.enumerated().filter {
        $0.element > sqrt(.ulpOfOne)
    }

    // Prints
    // [(offset: 1, element: 16384.0), (offset: 5, element: 655.3602)]
    // [(offset: 5, element: 16384.0), (offset: 7, element: 1474.56)]
    // [(offset: 3, element: 16384.0), (offset: 9, element: 5898.24)]
    // [(offset: 2, element: 368.64), (offset: 7, element: 16384.0)]
    print(components)
}
```

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.

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.

{} 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.