Article

Understanding data packing for Fourier transforms

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

Overview

The vDSP real-to-complex fast Fourier transform (FFT) and discrete Fourier transform (DFT) functions write their output in a special packed format to conserve memory. Use the code samples below to understand how to format source data for and interpret the results from the vDSP Fourier transform functions.

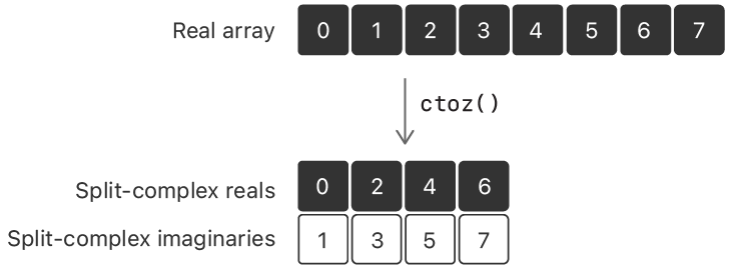
Convert real values to the split-complex format

The vDSP FFT and DFT functions work with data in split-complex format. Split-complex format separates the real and imaginary parts of complex numbers into two separate arrays. Given an array, `signal`, that contains real values, the following code converts the values to split-complex format. Use `vDSP_ctoz` to populate the split collections `complexReals` and `complexImaginaries` with the real values from `signal`:

```
let signal: [Float] = [0, 1, 2, 3, 4, 5, 6, 7]
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
    }
}
```

On return, `complexReals` contains the values `[0.0, 2.0, 4.0, 6.0]`, and `complexImaginaries` contains the values `[1.0, 3.0, 5.0, 7.0]`. The diagram below illustrates how `vDSP_ctoz` converts the real values to the even-odd split configuration:



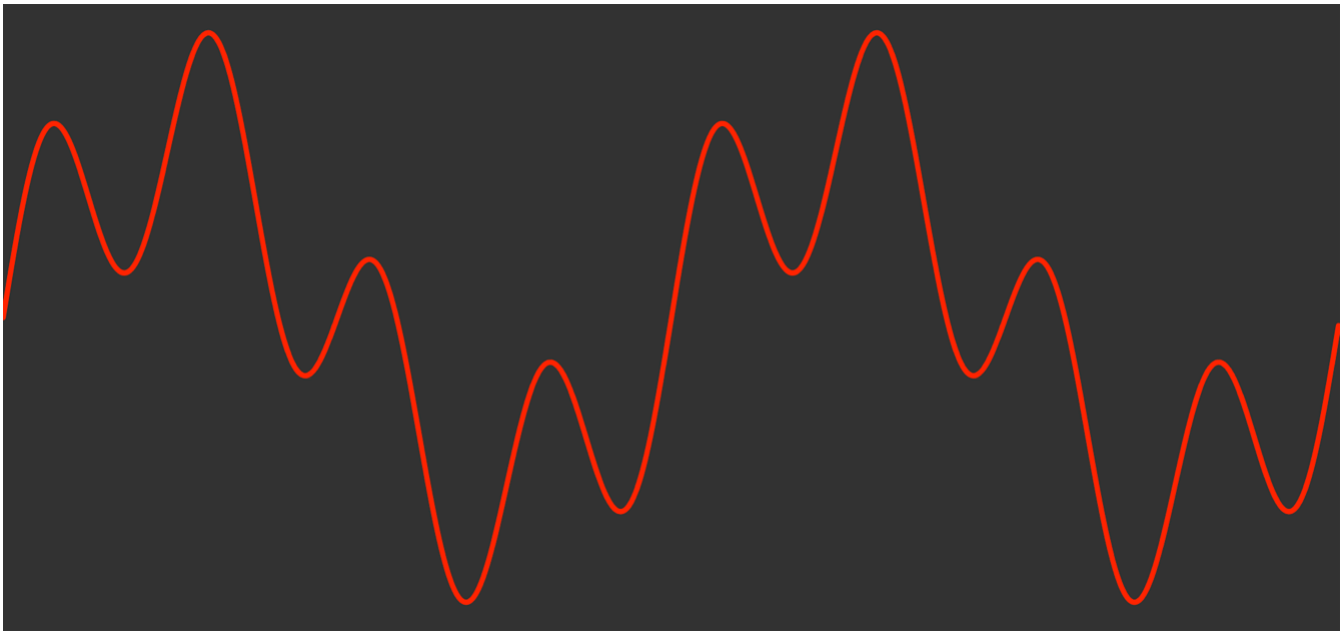
Create a composite sine wave

Use the following function to fill an array with values that represent a composite sine wave:

```
/// Returns an array that contains a composite sine wave from the specified frequenc
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 that the function uses to calculate 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
    }
}
```

The following figure visualizes the values of a 1024-element array that `makeCompositeSineWave()` returns with the frequency-amplitude pairs `[(f: 2, a: 1.5), (f: 8, a: 1.0)]`:



Perform Fourier transform on 1D real data in split-complex format

Use the following code to populate the real `signal` array with a composite of four sine waves and create empty arrays for the complex values:

```
let realValuesCount = 32
let signal: [Float] = makeCompositeSineWave(from: [(f: 1, a: 1),
                                                   (f: 5, a: 1),
                                                   (f: 10, a: 1),
                                                   (f: 15, a: 1)],
                                            count: realValuesCount)

let complexValuesCount = realValuesCount / 2
var complexReals = [Float](repeating: 0,
                           count: complexValuesCount)
var complexImaginaries = [Float](repeating: 0,
                                 count: complexValuesCount)
```

Use vDSP_fft_zrip to perform an in-place fast Fourier transform (FFT) on the real values. In this example, the vDSP_ctoz function uses a DSPSplitComplex structure as an intermediary to populate the `complexReals` and `complexImaginaries` arrays:

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

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

            let log2n = vDSP_Length(log2(Float(realValuesCount)))
            if let fft = vDSP_create_fftsetup(log2n, FFTRadix(kFFTRadix2)) {
                vDSP_fft_zrip(fft,
                              &splitComplex, 1,
                              log2n,
                              FFTDirection(kFFTDirection_Forward))

                vDSP_destroy_fftsetup(fft)
            }
        }
    }
}
```

Alternatively, use vDSP_DFT_zrop_CreateSetup to create an object that performs a discrete Fourier transform on the real values:

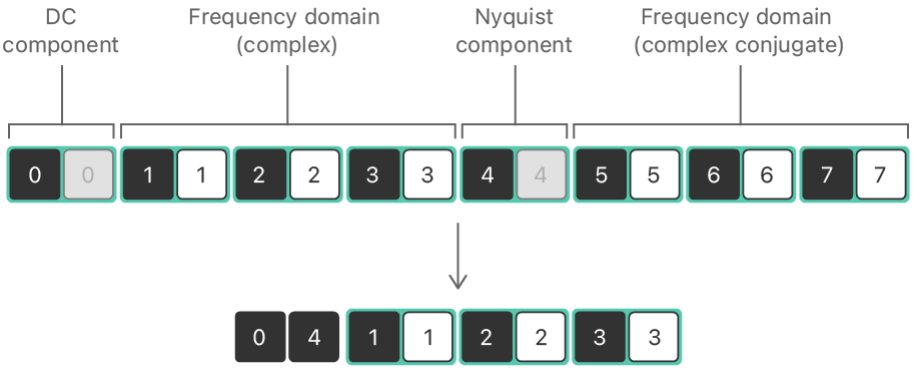
```
signal.withUnsafeBytes { signalPtr in
    complexReals.withUnsafeMutableBufferPointer { realPtr in
        complexImaginaries.withUnsafeMutableBufferPointer { imagPtr in
            var splitComplex = DSPSplitComplex(realp: realPtr.baseAddress!,
                                               imagp: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

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

            if let dft = vDSP_DFT_zrop_CreateSetup(nil,
                                                   vDSP_Length(realValuesCount),
                                                   .FORWARD) {
                vDSP_DFT_Execute(dft,
                                 realPtr.baseAddress!, imagPtr.baseAddress!,
                                 realPtr.baseAddress!, imagPtr.baseAddress!)
```

```
                vDSP_DFT_DestroySetup(dft)
            }
        }
    }
}
```

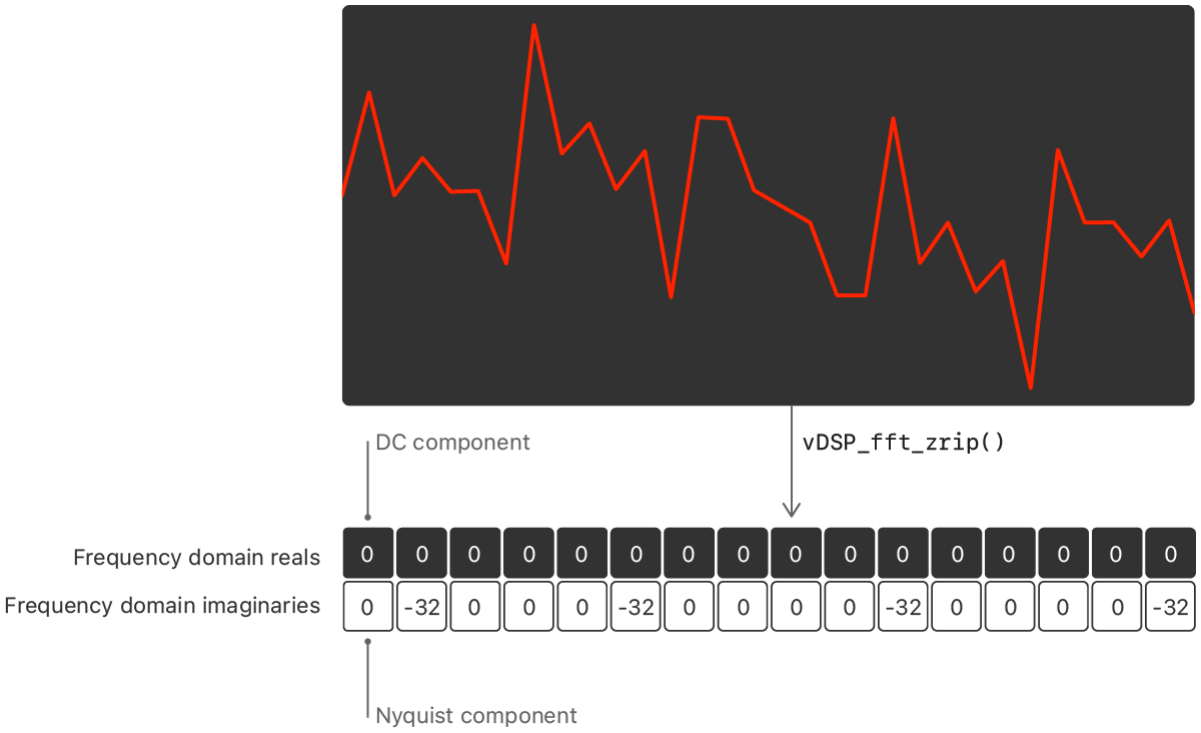
The result of a forward Fourier transform on n real values is n complex values. The list below describes how the vDSP library structures the result:

- Complex element `0` contains the DC component (equal to the sum of the time domain values) in its real part and zero in its imaginary part.
- Complex elements 1 through `n/2 - 1` contain the complex frequency-domain values.
- Complex element `n/2` contains the Nyquist component (the cosine component coefficient at the Nyquist frequency) in its real part and zero in its imaginary part.
- The remaining complex elements contain the complex conjugates of the complex frequency-domain values.

vDSP exploits the zeros in the DC and Nyquist elements and the symmetry of the complex conjugates. The Fourier transform routines represent the frequency-domain data in `n/2` complex values. They achieve this by placing the real Nyquist component in the imaginary part of the DC element and omitting the complex conjugates:



The following figure illustrates the frequency-domain result from performing a forward transform on the signal data that contains the four composite sine waves:



The indices of the nonzero elements in the frequency-domain data are the sine wave frequencies in the original time-domain signal: 1 , 5, 10, and 15.

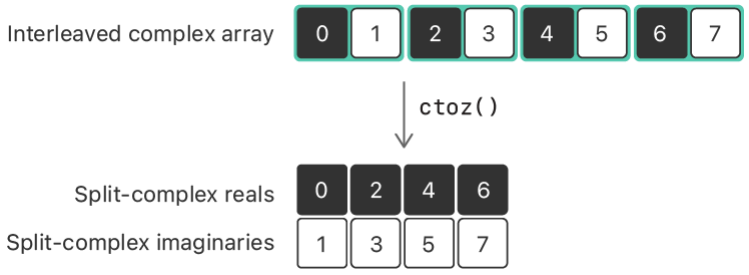
Convert interleaved-complex to the split-complex format

Given an array, `interleavedComplex`, that contains interleaved complex values, the following code converts the values to split-complex format. Use `vDSP_ctoz` to populate the split collections `complexReals` and `complexImaginaries` with the complex values from `interleavedComplex`:

```
let interleavedComplex: [DSPComplex] = [DSPComplex(real: 0, imag: 1),
                                        DSPComplex(real: 2, imag: 3),
                                        DSPComplex(real: 4, imag: 5),
                                        DSPComplex(real: 6, imag: 6)]

let count = interleavedComplex.count

var complexImaginaries = [Float]()
var complexReals = [Float](unsafeUninitializedCapacity: count) {
    realBuffer, realInitializedCount in

    complexImaginaries = [Float](unsafeUninitializedCapacity: count) {
        imagBuffer, imagInitializedCount in

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

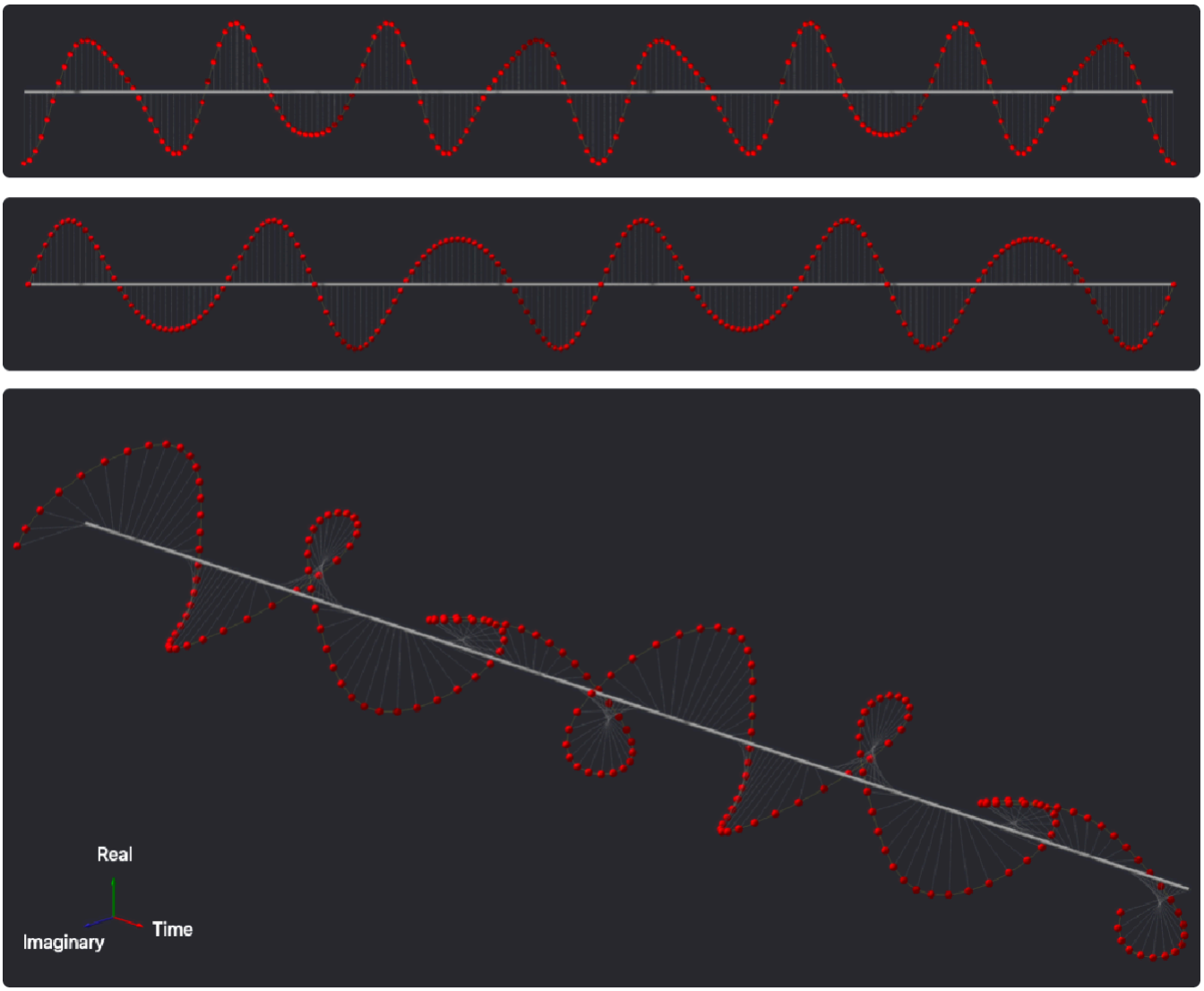
```
        vDSP_ctoz(interleavedComplex, 2,
                  &splitComplex, 1,
                  vDSP_Length(count))

        imagInitializedCount = count
    }

    realInitializedCount = count
}
```

On return, `complexReals` contains the values `[0.0, 2.0, 4.0, 6.0]`, and `complexImaginaries` contains the values `[1.0, 3.0, 5.0, 7.0]`. The following diagram illustrates how `vDSP_ctoz(_:_:_:_:)` converts the interleaved values to the even-odd split configuration:



Perform Fourier transform on 1D complex data

The following figure shows a representation of time-domain complex data. The real part and imaginary part contain distinct composite sine waves:



Use vDSP_DFT_zop_CreateSetup to create a DFT object for a complex transform, or call vDSP_fft_zip to perform a complex FFT in place. The following code creates a complex signal and performs a forward DFT:

```
let complexValuesCount = 16

let realFrequencyAmplitudePairs: [(f: Float, a: Float)] = [(3, 1), (5, 0.2)]
var complexReals = makeCompositeSineWave(from: realFrequencyAmplitudePairs,
                                         count: complexValuesCount)

let imaginaryFrequencyAmplitudePairs: [(f: Float, a: Float)] = [(4, 1), (7, 0.25)]
var complexImaginaries = makeCompositeSineWave(from: imaginaryFrequencyAmplitudePair
                                               count: complexValuesCount)

if let dft = vDSP_DFT_zop_CreateSetup(nil,
                                      vDSP_Length(complexValuesCount),
                                      .FORWARD) {

    vDSP_DFT_Execute(dft,
```

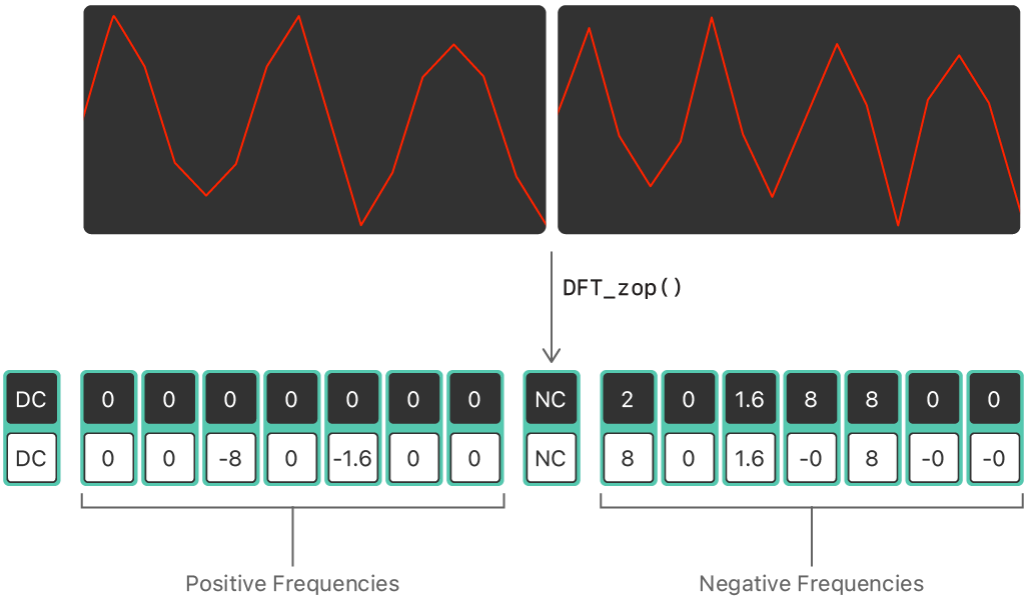
```
                        complexReals,
                        complexImaginaries,
                        &complexReals,
                        &complexImaginaries)

    vDSP_DFT_DestroySetup(dft)
}
```

On return, `complexReals` and `complexImaginaries` contain the following values:

```
Reals:
[ 0.0, 0.0, 0.0,  8.0,  8.0,  1.6,  0.0,  2.0, 0.0,  2.0, 0.0, 1.6,  8.0, 8.0,  0.0,
Imaginaries:
[ 0.0, 0.0, 0.0, -8.0, -0.0, -1.6, -0.0, -0.0, 0.0, -0.0, 0.0, 1.6, -0.0, 8.0, -0.0,
```

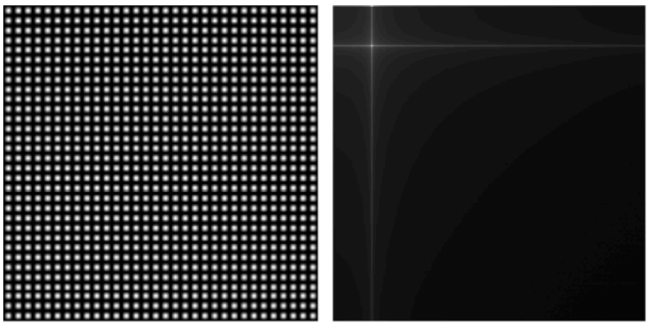
The indices of the nonzero magnitudes indicate the component frequencies 3, 4, 5, and 7.

The following figure shows the layout of the frequency-domain data. The DC and Nyquist components contain real and imaginary parts. The elements `0` to `n/2 -1` contain the positive-frequency values, and the elements `n/2 + 1` to `n - 1` contain the negative-frequency values.

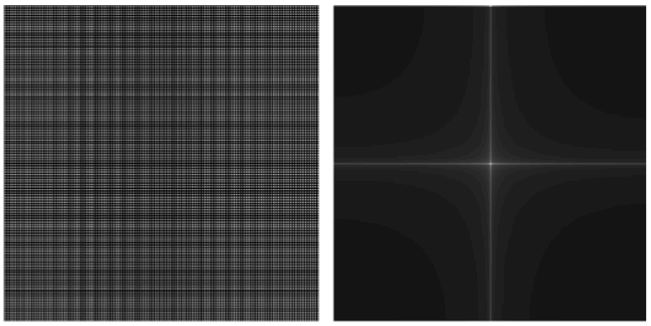


Perform Fourier transform on 2D real data

vDSP provides routines you can use to compute the FFT of 2D real data such as single-channel images. The following figure shows the original, spatial-domain representations of grids of dots with their frequency-domain counterparts:



Frequency = 32



Frequency = 128

Use vDSP_ctoz to convert real spatial-domain values like pixel intensities to complex values. The real part of each complex value contains the even pixel values, and the imaginary part of each complex value contains the odd pixel values. The following code defines the width and height of the matrix of complex values for a 512 x 512 real matrix:

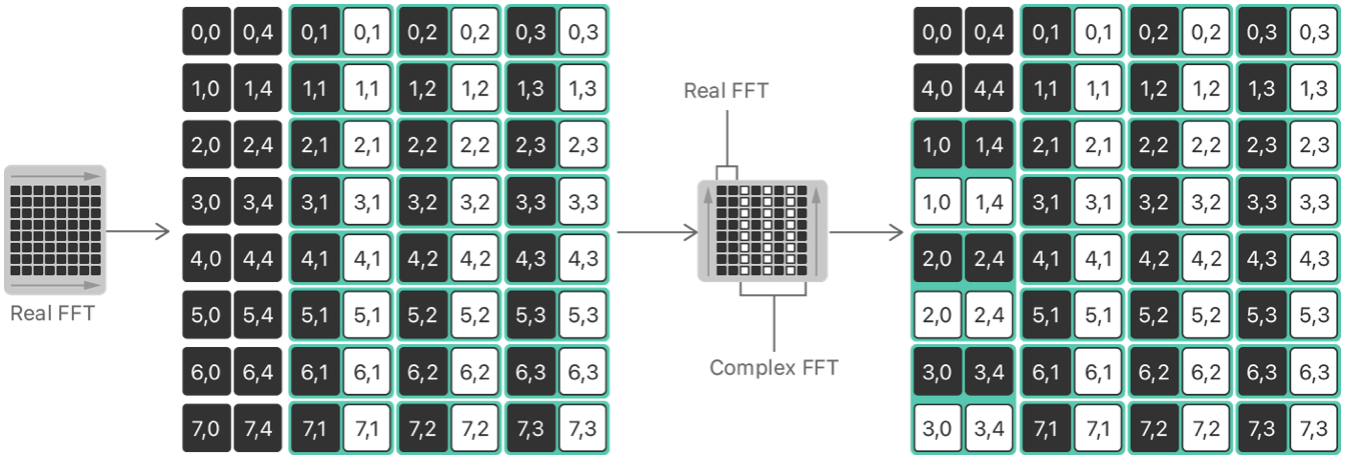
```
let realDimension = 512
let complexValuesWidth = realDimension / 2
let complexValuesHeight = realDimension
```

Use vDSP_fft2d_zrip to perform the forward FFT in place. In this example, `imageData` is an array that contains `realDimension * realDimension` real pixel values:

```
let complexElementCount = complexValuesWidth * complexValuesHeight
var complexReals = [Float]()
var complexImaginaries = [Float]()

imageData.withUnsafeBytes { imageDataPtr in
    complexReals = [Float](unsafeUninitializedCapacity: complexElementCount) {
        realBuffer, realInitializedCount in
        complexImaginaries = [Float](unsafeUninitializedCapacity: complexElementCour
            imagBuffer, imagInitializedCount in

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

            vDSP_ctoz([DSPComplex](imageDataPtr.bindMemory(to: DSPComplex.self)), 2,
                      &splitComplex, 1,
                      vDSP_Length(complexValuesWidth * complexValuesHeight))

            // The binary logarithm of `max(rowCount, columnCount)`.
            let countLog2n = vDSP_Length(log2(Float(realDimension)))
            if let fft = vDSP_create_fftsetup(countLog2n, FFTRadix(kFFTRadix2)) {
```

```
                let dimensionLog2n = vDSP_Length(log2(Float(realDimension)))
                vDSP_fft2d_zrip(fft, &splitComplex,
                                1, 0,
                                dimensionLog2n, dimensionLog2n,
                                FFTDirection(kFFTDirection_Forward))

                vDSP_destroy_fftsetup(fft)
            }

            imagInitializedCount = complexElementCount
        }
        realInitializedCount = complexElementCount
    }
}
```

The 2D FFT operates on real data by first transforming each row. This transform generates real values — the DC and Nyquist component of each row — in the first two elements of each row. The second pass of the transform computes the FFT for each column. The first two columns contain real values, and vDSP uses the real transform routines. Subsequent columns contain complex values, and vDSP uses the complex transform routines:

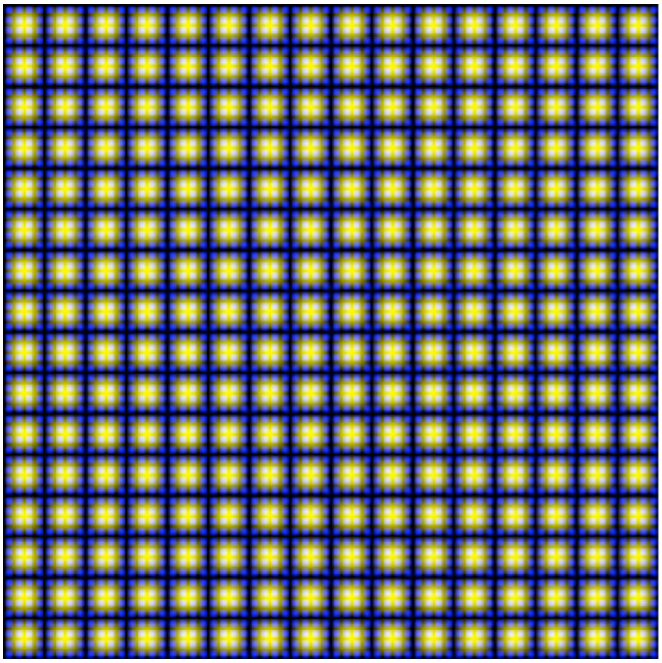


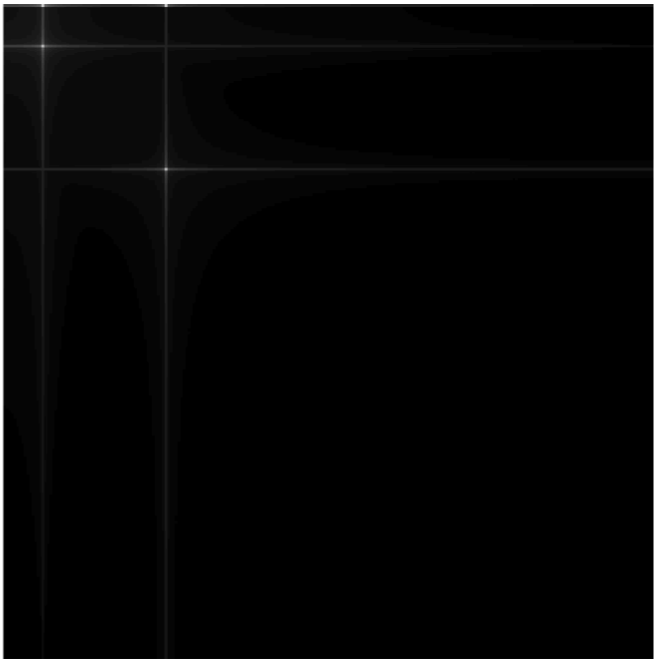
Because the complex elements $n/2 + 1$ to $n - 1$ contain the negative-frequency values, for tasks such as visualizing the frequency-domain representation of 2D real data, you may be able to discard the bottom $n/2$ rows.

Perform Fourier transform on 2D complex data

vDSP provides functions for performing Fourier transforms on 2D complex data, such as two-channel images. The following image shows the spatial-domain representation of 2D complex data

with low-frequency data in the red and green channels, and high-frequency data in the blue channel. The frequency-domain representation shows the magnitude of the complex values:





Use vDSP_fft2d_zip to perform a complex FFT on 2D complex data:

```
let complexDimension = 512

var complexReals = [Float](repeating: 0,
                           count: complexDimension * complexDimension)
var complexImaginaries = [Float](repeating: 0,
                                 count: complexDimension * complexDimension)

// Populate `complexReals` and `complexImaginaries` with 2D complex data.

complexReals.withUnsafeMutableBufferPointer { realPtr in
    complexImaginaries.withUnsafeMutableBufferPointer { imagPtr in

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

        // The binary logarithm of `max(rowCount, columnCount)`.
        let countLog2n = vDSP_Length(log2(Float(complexDimension)))
        if let fft = vDSP_create_fftsetup(countLog2n, FFTRadix(kFFTRadix2)) {

            // The binary logarithm of the width or height of the 2D matrix.
            let dimensionLog2n = vDSP_Length(log2(Float(complexDimension)))
            vDSP_fft2d_zip(fft, &splitComplex,
```

```
                            1, 0,
                            dimensionLog2n, dimensionLog2n,
                            FFTDirection(kFFTDirection_Forward))

            vDSP_destroy_fftsetup(fft)
        }
    }
}
```

On return, `complexReals` and `complexImaginaries` contain the frequency-domain representation of the complex data. Because both the horizontal pass and the vertical pass are complex FFTs, the right n/2 columns and the bottom n/2 rows contain the negative frequencies:

Scale time- and frequency-domain data

To provide the best execution performance, vDSP's Fourier routines don't scale transform results. The following table summarizes the scaling factor for the vDSP FFT and DFT operations:

	1D	2D
Real forward transform	2	2
Real inverse transform	Number of real elements	Number of real elements (rows x columns)
Complex forward transform	1	1
Complex inverse transform	Number of complex elements	Number of complex elements (rows x columns)

For example, the following code performs a forward transform and an inverse transform on eight real elements. The code multiplies the frequency-domain data by 1/2, and the time-domain data by 1/n. The result is identical to the original data:

```
let realValuesCount = 8

// The result of `vDSP_ctoz` on `[0, 1, 2, 3, 4, 5, 6, 7]`.
var complexReals: [Float] = [0, 2, 4, 6]
var complexImaginaries: [Float] = [1, 3, 5, 7]
```

```
// Perform forward transform.
if let dft = vDSP_DFT_zrop_CreateSetup(nil,
                                       vDSP_Length(realValuesCount),
                                       .FORWARD) {
    vDSP_DFT_Execute(dft,
                     complexReals, complexImaginaries,
                     &complexReals, &complexImaginaries)

    vDSP_DFT_DestroySetup(dft)
}

// Apply real forward scaling factor (2).
vDSP.multiply(1 / 2, complexReals, result: &complexReals)
vDSP.multiply(1 / 2, complexImaginaries, result: &complexImaginaries)

// Perform inverse transform.
if let dft = vDSP_DFT_zrop_CreateSetup(nil,
                                       vDSP_Length(realValuesCount),
                                       .INVERSE) {
    vDSP_DFT_Execute(dft,
                     complexReals, complexImaginaries,
                     &complexReals, &complexImaginaries)

    vDSP_DFT_DestroySetup(dft)
}

// Apply real inverse scaling factor (n).
vDSP.multiply(1 / Float(realValuesCount), complexReals, result: &complexReals)
vDSP.multiply(1 / Float(realValuesCount), complexImaginaries, result: &complexImagir

print(complexReals) // Prints "[0.0, 2.0, 4.0, 6.0]".
print(complexImaginaries) // Prints "[1.0, 3.0, 5.0, 7.0]".
```

See Also

Fourier and Cosine Transforms

- 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.

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.