Sample Code

Signal extraction from noise

Use Accelerate's discrete cosine transform to remove noise from a signal.

Download

iOS 13.0+ | iPadOS 13.0+ | Xcode 14.3+

Overview

Accelerate's vDSP module provides functions to perform discrete and fast Fourier transforms (FFTs) on 1D vectors and 2D matrices containing complex numbers. If you want to perform a similar transform on a vector of real numbers, vDSP includes discrete cosine transforms (DCTs).

FFTs and DCTs decompose a signal into its frequency components (known as the *frequency-domain* representation of the signal), and the inverse transform rebuilds a signal into its *time-domain* representation from the frequency components.

By zeroing low-magnitude data, such as noise, from the frequency-domain data, you can reconstruct a signal, leaving only its dominant frequencies. The meaningful signals that you're trying to isolate tend to have their energy packed at a few frequencies. Noise, however, has its energy more uniformly spread across the frequency spectrum (that's what makes it noise). If you zero out low-magnitude frequency components, you can eliminate much of the noise from the spectrum.

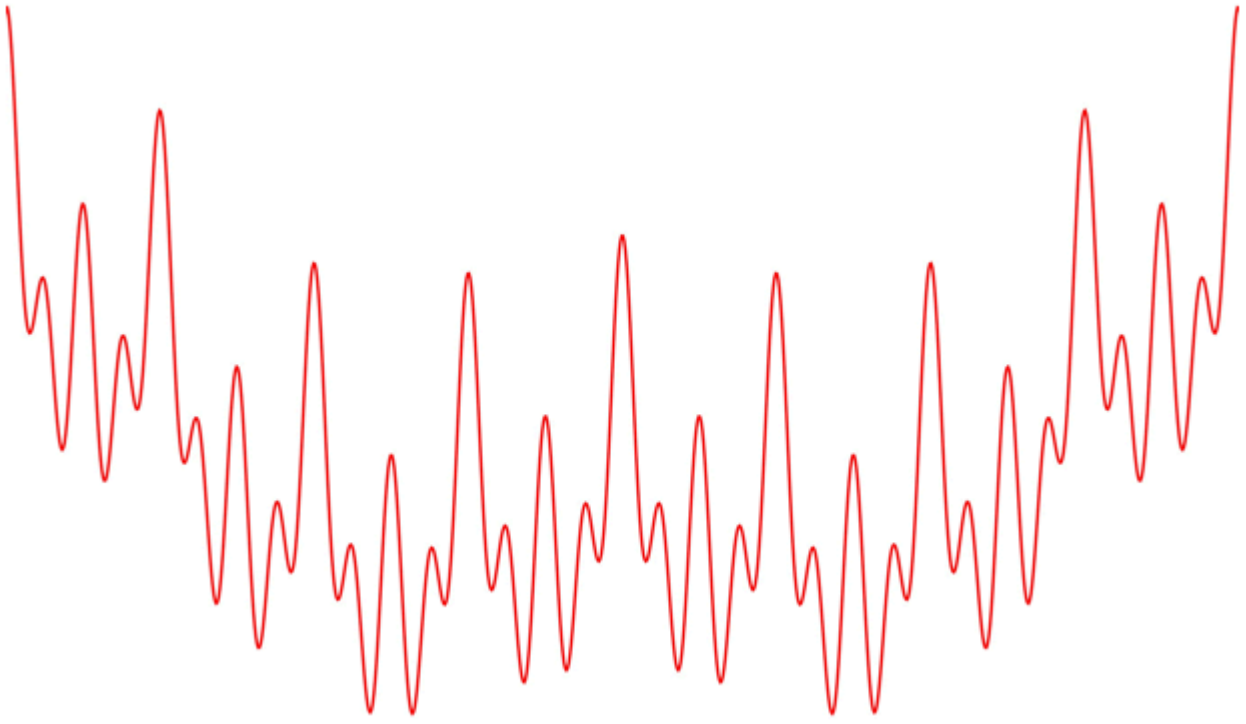
Generate the test signal

The `noisySignal` array contains the noisy signal from which the sample app extracts the underlying signal. The underlying signal is a series of cosine waves that's stored as 1024 samples in an array of single-precision values.

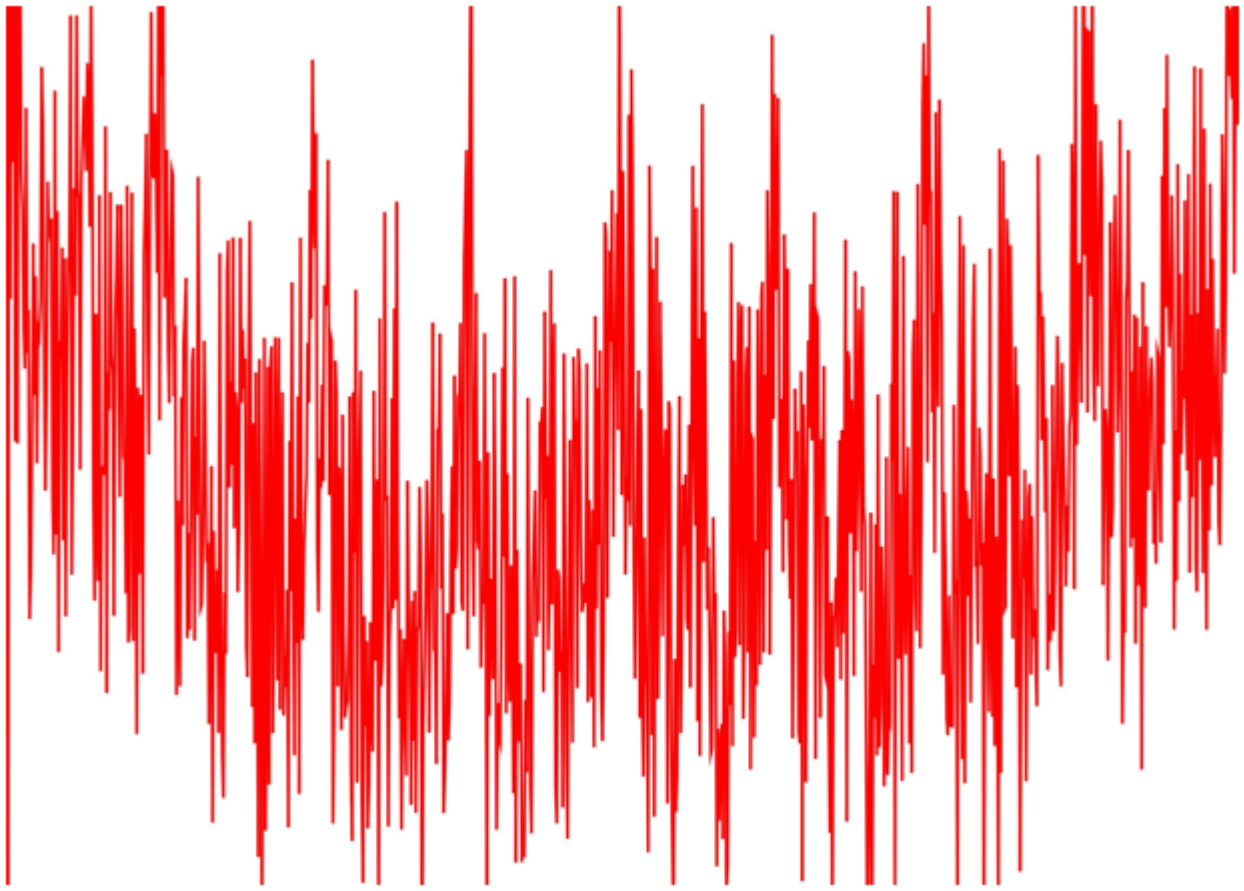
The `static SignalExtractor.generateSignal(noiseAmount:sampleCount:)` function generates a sample at each data point.

```
static func generateSignal(noiseAmount: Double,
                           sampleCount: Int) -> [Float] {

    let tau = Float.pi * 2

    return (0 ..< sampleCount).map { i in
        let phase = Float(i) / Float(sampleCount) * tau

        var signal = cos(phase * 1) * 1.0
            signal += cos(phase * 2) * 0.8
            signal += cos(phase * 4) * 0.4
            signal += cos(phase * 8) * 0.8
            signal += cos(phase * 16) * 1.0
            signal += cos(phase * 32) * 0.8

        return signal + .random(in: -1...1) * Float(noiseAmount)
    }
}
```

When the `noiseAmount` parameter is zero, the values that this code generates return a signal like the one in the image below:



Adding noise to the signal makes it unrecognizable.



Prepare the DCT setups

The sample app creates setup objects that contain all the information required to perform the forward and inverse DCT operations. Because creating these setup objects can be expensive, the sample app creates the DCT setup objects once and reuses them.

The forward transform is a type II DCT.

```
static let forwardDCTSetup = vDSP.DCT(count: sampleCount,
                                      transformType: vDSP.DCTTransformType.II)!
```

The inverse transform is a type III DCT.

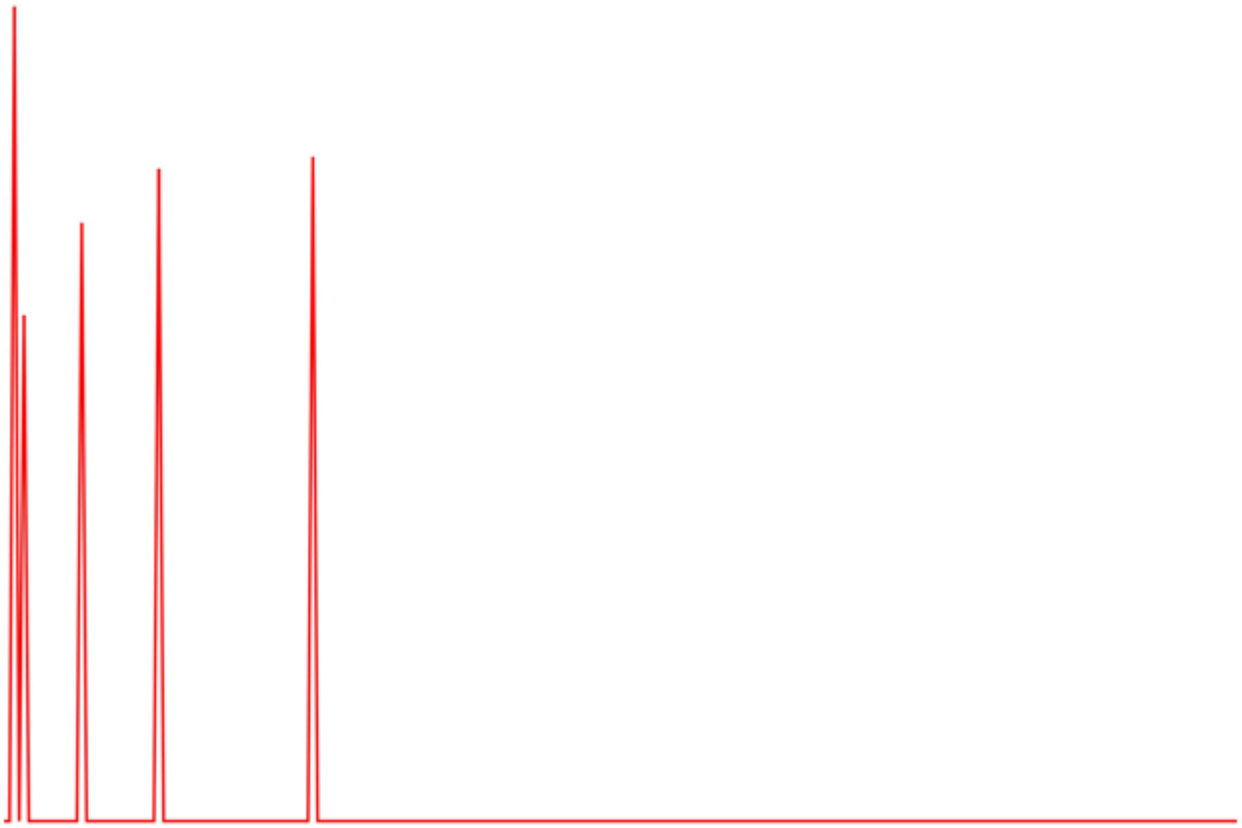
```
static let inverseDCTSetup = vDSP.DCT(count: sampleCount,
                                      transformType: vDSP.DCTTransformType.III)!
```

Perform the DCT

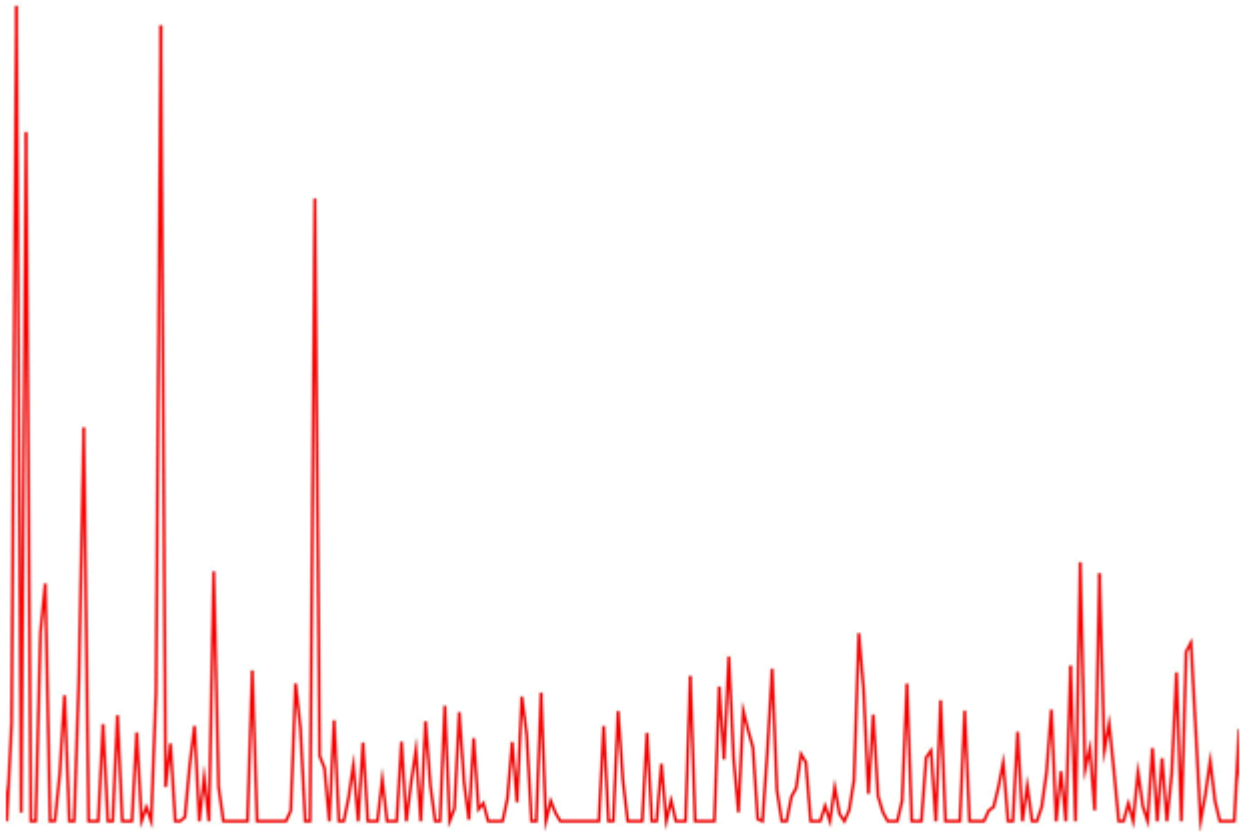
The `transform(_:)` function performs the DCT. This function requires a source array that contains the source signal and a destination array that the function overwrites with the frequency-domain data.

```
forwardDCTSetup.transform(noisySignal,
                          result: &frequencyDomainDestination)
```

The following visualization of the frequency-domain data shows the component cosine parts. The `cos(phase * 1) * 1.0` component is on the left, and `cos(phase * 16) * 1.0` is on the right:



The frequency-domain visualization of the noisy signal shows the dominant frequencies with the noise spread evenly throughout the frequency range. The sample zeroes the low-magnitude data to generate the noise-free signal.



Apply a threshold to the frequency-domain data

Remove the noise from the signal by zeroing all values in the frequency-domain data that are below a specified threshold.

The `threshold(_:to:with:result:)` function sets all values in the frequency-domain array that fall below the threshold to zero.

```
vDSP.threshold(frequencyDomainDestination,
               to: Float(threshold),
               with: .zeroFill,
               result: &frequencyDomainDestination)
```

Recreate the signal

The inverse DCT generates a new signal using the cleaned-up frequency-domain data:

```
inverseDCTSetup.transform(frequencyDomainDestination,
                          result: &timeDomainDestination)
```

The sample app scales the inverse DCT so that it matches the magnitude of the original signal. The scaling factor for the forward transform is 2, and the scaling factor for the inverse transform is the number of samples (in this case, 1024). The `divide(_:_:)` function divides the inverse DCT result by `count / 2` to return a signal with the correct magnitude.

```
let divisor = Float(Self.sampleCount / 2)

vDSP.divide(timeDomainDestination,
            divisor,
            result: &timeDomainDestination)
```

For more information on scaling factors for the vDSP FFT and DFT operations, see Understanding data packing for Fourier transforms.

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.
- 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.