Article

Using vDSP for vector-based arithmetic

Increase the performance of common mathematical tasks with vDSP vector-vector and vector-scalar operations.

Overview

vDSP provides a suite of general-purpose, high-performance arithmetic functions that are alternatives to `for` loops and `map` when applying operations on collections of floating-point values.

For example, the code below multiplies the element-wise sum of two arrays by a scalar value:

```
let a: [Float] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let b: [Float] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
let c: Float = 2

let d = zip(a, b).map {
    ($0.0 + $0.1) * c
}

// d = [22.0, 44.0, 66.0, 88.0, 110.0, 132.0, 154.0, 176.0, 198.0, 220.0]
```

The vDSP version of the same operation runs significantly faster.

```
let d = [Float](unsafeUninitializedCapacity: a.count) {
    buffer, initializedCount in

    vDSP.multiply(addition: (a, b),
                  c,
                  result: &buffer)
```

```
    initializedCount = a.count
}

// d = [22.0, 44.0, 66.0, 88.0, 110.0, 132.0, 154.0, 176.0, 198.0, 220.0]
```

Many vDSP functions have a variant that returns the result.

```
let a: [Float] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let b: [Float] = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
let c: Float = 2

let d = vDSP.multiply(addition: (a, b),
                      c)
```

Available arithmetic functions

The following table summarizes the basic arithmetic functions available in vDSP. All functions are available in single- and double-precision variants.

In the *Operation* column, a subscript (for example, `a[i]`) indicates a vector, and no subscript (for example, a) indicates a scalar value.

Operation	Function
`c[i] = a[i] + b`	`add(_:_:)`
`c[i] = a[i] + b[i]`	`add(_:_:)`
`c[i] = a[i] - b[i]`	`subtract(_:_:)`
`c[i] = a[i] * b`	`multiply(_:_:)`
`c[i] = a[i] * b[i]`	`multiply(_:_:)`
`c[i] = a[i] / b`	`divide(_:_:)`
`c[i] = a / b[i]`	`divide(_:_:)`
`o0[i] = i1[i] + i0[i]` `o1[i] = i1[i] - i0[i]`	`addSubtract(_:_:addResult:subtractResult:)`

Operation	Function
`c[i] = a[i] / b[i]`	`divide(_:_:)`
`d[i] = (a[i] + b[i]) * c`	`multiply(addition:_:)`
`d[i] = (a[i] + b[i]) * c[i]`	`multiply(addition:_:)`
`d[i] = (a[i] - b[i]) * c`	`multiply(subtraction:_:)`
`d[i] = (a[i] - b[i]) * c[i]`	`multiply(subtraction:_:)`
`d[i] = (a[i] * b[i]) + c`	`add(multiplication:_:)`
`d[i] = (a[i] * b) + c[i]`	`add(multiplication:_:)`
`d[i] = (a[i] * b[i]) + c[i]`	`add(multiplication:_:)`
`d[i] = (a[i] * b[i]) - c[i]`	`subtract(multiplication:_:)`
`e[i] = (a[i] * b) + (c[i] * d)`	`add(multiplication:multiplication:)`
`e[i] = (a[i] + b[i]) * (c[i] + d[i])`	`multiply(addition:addition:)`
`e[i] = (a[i] * b[i]) - (c[i] * d[i])`	`subtract(multiplication: multiplication:)`
`e[i] = (a[i] - b[i]) * (c[i] - d[i])`	`multiply(subtraction:subtraction:)`
`e[i] = (a[i] + b[i]) * (c[i] - d[i])`	`multiply(addition:subtraction:)`

See Also

Signal Processing Essentials

- Controlling vDSP operations with stride

  Operate selectively on the elements of a vector at regular intervals.

- Using linear interpolation to construct new data points

Fill the gaps in arrays of numerical data using linear interpolation.

Resampling a signal with decimation

Reduce the sample rate of a signal by specifying a decimation factor and applying a custom antialiasing filter.

vDSP

Perform basic arithmetic operations and common digital signal processing (DSP) routines on large vectors.