vDSP Programming Guide

Developer

Contents

Figures and Listings

Introduction

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

The vDSP API provides mathematical functions for applications such as speech, sound, audio, and video processing, diagnostic medical imaging, radar signal processing, seismic analysis, and scientific data processing.

You should read this document if you need to work with vectors, convolution, Fourier transforms, or perform other similar signal processing tasks.

Organization of This Document

This document is divided into two chapters:

- About the vDSP API (page 5)—Describes calling conventions, naming conventions, and data types used by the vDSP API.
- Using Fourier Transforms (page 9)—Covers the Fourier transform functionality in vDSP at a conceptual level.

See Also

vDSP Examples provides working samples that demonstrate the use of the vDSP API to perform convolutions, Discrete Fourier Transforms (DFTs), or Fast Fourier Transforms (FFTs).

For more information about specific APIs, see *vDSP Reference*.

About the vDSP API

The vDSP API provides mathematical functions for applications such as speech, sound, audio, and video processing, diagnostic medical imaging, radar signal processing, seismic analysis, and scientific data processing.

The vDSP functions operate on real and complex data types. The functions include data type conversions, fast Fourier transforms (FFTs), and vector-to-vector and vector-to-scalar operations.

The vDSP functions have been implemented in two ways: as vectorized code, which uses vector instructions (SSE3, for example) in the processor, and as scalar code, which does not. The vDSP API uses the appropriate version depending on the arguments given and the vector capabilities of the processor.

The header file `vDSP.h` defines nonstandard data types used by the vDSP functions and symbols accepted as flag arguments to vDSP functions. The vDSP API itself is part of vecLib, which in turn is part of the Accelerate framework in OS X. Thus, when actually including the header in your code, you should always use the umbrella framework header (`#include <Accelerate/Accelerate.h>`).

Calling Conventions

This section describes the calling conventions used with the vDSP functions and discusses the use of address strides.

Passing Parameters

Most vDSP functions are provided with input data and produce output data. All such data arguments, whether vector or scalar, are passed by reference; that is, callers pass pointers to memory locations from which input data is read and to which output data is written.

Other argument types passed to vDSP functions include:

- Address strides, which tell a function how to step through input and output data. Stride examples include every element, every other element, every third element, and so forth.
- Flags, which influence a function's behavior in some way. Examples include forward versus inverse directional flags for discrete Fourier transforms.
- Element counts, which tell functions how many elements to process.

Address strides, flags, and element counts are integer values and, unlike input and output data, these values are passed to a function directly, not by reference. For example, the vector multiply command, `vDSP_vmul`, uses vector pointers, address strides, and element counts:

```
void vDSP_vmul(
    float *input_1, /* input vector 1 */
    SInt32 stride_1, /* address stride for input vector 1 */
    float *input_2, /* input vector 2 */
    SInt32 stride_2, /* address stride for input vector 2 */
    float *result, /* output vector */
    SInt32 strideResult, /* address stride for output vector */
    UInt32 size /* real output count */
);
```

A typical call to `vDSP_vmul` illustrates how the input and output data is passed by reference, whereas address strides, flags, and element counts pass directly:

```
/* Multiply sequential values of two 1,024-point vectors */
float a[1024], b[1024], c[1024];
vDSP_vmul( a, 1, b, 1, c, 1, 1024 );
```

Specifying Address Strides

As mentioned earlier, address strides tell functions how to step through input and output data: every element, every second element, every third element, and so on. Though usually positive, address strides can be specified as negative for most vDSP functions. Developers should be aware, however, that negative address strides can often change the arithmetic operation of a function, so experimentation with input values that produce known results is recommended.

When operating on real vectors, address strides of 1 address every element, address strides of 2 address every other element, and so forth. For some functions, address strides of 1 for real vectors result in superior performance over non-unit strides because longer strides require those functions to fall back to using scalar-mode CPU instructions. The use of split complex yields similar performance benefits for complex vectors.

When operating on complex vectors, there are two formats for storing each complex number in a vector element: split complex and interleaved complex.

With split complex vectors (the `DSPSplitComplex` and `DSPDoubleSplitComplex` types), the real parts of the elements in the vector are stored in one array, and the imaginary parts are stored in a separate array. Thus, as with real arrays, an address stride of 1 addresses every element. Most functions use split complex vectors.

With interleaved complex vectors (the `DSPComplex` and `DSPDoubleComplex` types), complex numbers are stored as an ordered pairs of floating point or double values. Therefore, a stride of 2 is specified to process every vector element; a stride of 4 is specified to process every other element; and so forth.

Return Values

Repeating the synopsis of `vDSP_vmul`, the definition line shows `vDSP_vmul` to be of type `void`; that is, it provides no return value to the caller:

```
void vDSP_vmul(a, i, b, j, c, k, n)
```

All data returned by these functions is done inline through overwriting the array arguments, in this case, argument `c`.

IMPORTANT: To achieve the best possible performance, the vDSP routines perform no error checking and return no completion codes. All arguments are assumed to be specified and passed correctly, and these responsibilities lie with the caller.

Naming and Data Type Conventions

Functions in the vDSP API begin with a `vDSP_` prefix. Within this namespace, the vDSP API contains multiple variants of functions depending on whether they use single-precision or double-precision numbers. These variants can be distinguished by their suffix.

Suffix	Meaning
no suffix	Uses single-precision floating point numbers.
`D`	Uses double-precision floating point numbers

For example, the `vDSP_fft2d_zip` and `vDSP_fft2d_zipD` functions are equivalent except that the former works with single-precision values and the latter with double-precision values interleaved in a single array.

The vDSP API is based on standard C integer and floating-point data types. It defines four additional data types to represent vector elements:

- `DSPComplex`—An ordered pair of `float` values stored as a packed data structure. Functions with no suffix may take this type.
- `DSPDoubleComplex`—An ordered pair of `double` values stored as a packed data structure. Functions with a `D` suffix may take this type.
- `DSPSplitComplex`—An ordered pair of `float` values stored as a pair of pointers into an array of real parts and an array of imaginary parts. Functions with no suffix and a `z` at the beginning of the last part of the name typically take this type.
- `DSPDoubleSplitComplex`—An ordered pair of `double` values stored as a pair of pointers into an array of real parts and an array of imaginary parts. Functions with a `D` suffix and a `z` at the beginning of the last part of the name typically take this type.

Using Fourier Transforms

Introduction

The vDSP API provides Fourier transforms for transforming one-dimensional and two-dimensional data between the time domain and the frequency domain.

FFT Weights Arrays

To boost performance, vDSP functions that process frequency-domain data expect an array of complex exponentials (sometimes called twiddle factors) to exist prior to calling the function. Once created, this FFT weights array can be used over and over by the same Fourier function and can be shared by several Fourier functions.

FFT weights arrays are created by calling `vDSP_create_fftsetup` (single-precision) or `vDSP_create_fftsetupD` (double-precision). Before calling a function that processes in the frequency domain, you must call one of these functions. The caller specifies the required array size to the setup function. The setup functions:

- Create a data structure to hold the array.
- Build the array.
- Return a pointer to the data structure (or `NULL` if storage for this structure could not be allocated).

The pointer to the data structure is then passed as an argument to subsequent Fourier functions. You must check the value of the pointer before supplying the pointer to a frequency domain function. A returned value of zero is the only explicit notification that the array allocation failed.

Argument `log2n` to these functions is the base-2 logarithm of `n`, where `n` is the number of complex unit circle divisions the array represents, and thus specifies the largest number of elements that can be processed by a subsequent Fourier function. Argument `log2n` must equal or exceed argument `log2n` supplied to any functions using the weights array. Functions automatically adjust their strides through the array when the table has more resolution, or larger `n`, than required.

Thus, a single call to `vDSP_create_fftsetup` or `vDSP_create_fftsetupD` can serve a series of calls to FFT functions as long as `n` is large enough for each function called and as long as the weights array is preserved.

For example, if you need to call `vDSP_fft_zrip` for 256 data points, you must call `vDSP_create_fftsetup` with `logn` of at least 8. If you also need to call `vDSP_fft_zrip` for a set of 2048 data points, the value of `logn` must be at least 11. So you could generate this table once for both sets as follows:

```
FFTsetup setup; vDSP_create_fftsetup( 11, 0 ); /* supports up to 2048 (2**11)
points  */

...

vDSP_fft_zrip( setup, small_set, 1, 8, FFT_FORWARD); /* 256 (2**8) points  */

...

vDSP_fft_zrip( setup, large_set, 1, 11, FFT_FORWARD); /* 2048 (2**11) points  */
```

Reusing a weights array for similarly sized FFTs is economical. 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 smaller number of elements.

For a more complete example of `vDSP_create_fftsetup` and FFT functions, see *vDSP Examples* .

Data Packing for Real FFTs

The discrete Fourier transform functions in the vDSP API provide a unique case in data formatting to conserve memory. Real-to-complex discrete Fourier transforms write their output data in special packed formats so that the complex output requires no more memory than the real input.

Packing and Transformation Functions

Applications that call the real FFT may have to use two transformation functions, one before the FFT call and one after. This is required if the input array is not in the even-odd split configuration.

A real array `A = {A[0],...,A[n]}` must be transformed into an even-odd array `AEvenOdd = {A[0],A[2],...,A[n-1],A[1],A[3],...A[n]}` by means of the call `vDSP_ctoz`.

The result of a real FFT on `AEvenOdd` of dimension `n` is a complex array of dimension `2n`, with a very special format:

`{[DC,0],C[1],C[2],...,C[n/2],[NY,0],Cc[n/2],...,Cc[2],Cc[1]}`

where:

- Values `DC` and `NY` are the DC and Nyquist components (real values)
- Array `C` is complex in a split representation

- Array `Cc` is the complex conjugate of `C` in a split representation

For a real array `A` of size `n` , the results, which are complex, require `2 * n` spaces. However, much of this data is either always zero or is redundant, so this data can be omitted. Thus, to fit this result into the same space as the input array, the algorithm throws away these unnecessary values. The real FFT stores its results as follows:

```
{[DC,NY],C[1],C[2],...,C[n/2]}.
```

The sections Packing for One-Dimensional Arrays (page 11) and Packing for Two-Dimensional Arrays (page 12) describe the packing formats in more detail.

Packing for One-Dimensional Arrays

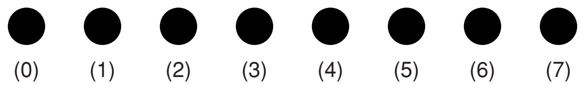
Due to its inherent symmetry, the forward Fourier transform of `n` floating-point inputs from the time domain to the frequency domain produces `n/2 + 1` unique complex outputs. Because data point `n/2 - i` equals the complex conjugate of point `n/2 + i`, the first half of the frequency data by itself (up to and including point `n/2 + 1`) is sufficient to preserve the original information.

In addition, the FFT data packing takes advantage of the fact that the imaginary parts of the first complex output (element zero) and of the last complex output (element `n/2`, the Nyquist frequency), are always zero.

This makes it possible to store the real part of the last output where the imaginary part of the first output would have been. The zero-valued imaginary parts of the first and last outputs are implied.

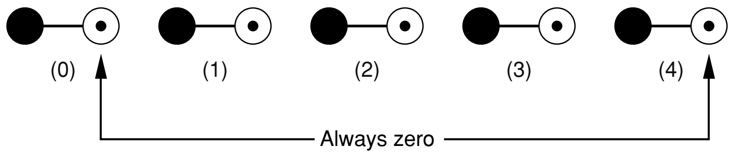
For example, consider this eight-point real vector as it exists prior to being transformed to the frequency domain (Figure 2-1 (page 11)).

Figure 2-1 Eight point real vector



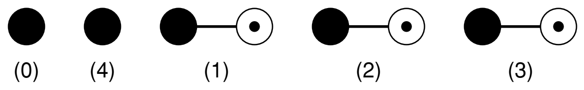
Transforming these eight real points to the frequency domain results in five complex values (Figure 2-2 (page 11)).

Figure 2-2 Results of an eight-point real-to-complex DFT



The five complex values are packed in the output vector shown in Figure 2-3 (page 12)

Figure 2-3 Packed format of an eight-point real-to-complex DFT



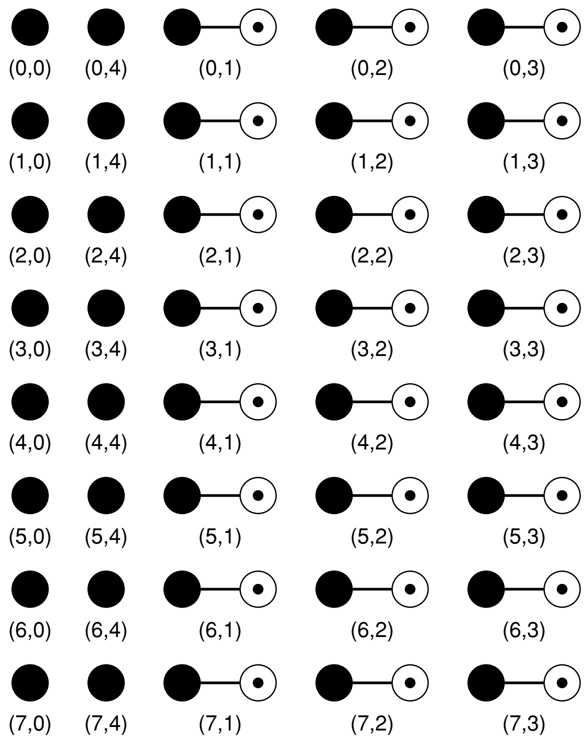
Notice that the real part of output four is stored where the imaginary part of output zero would have been stored with no packing. The imaginary parts of the first output and the last output, always zero, are implied.

Packing for Two-Dimensional Arrays

Two-dimensional real-to-complex DFTs are packed in a similar way. When processing two-dimensional real inputs, the DFTs first transform each row and then transform down columns. As each row is transformed, it is packed as previously described for one-dimensional transforms.

For example, to transform an 8-by-8 real image to the frequency domain, first each row is transformed and packed as shown in Figure 2-4 (page 12)

Figure 2-4 Interim format of an 8-by-8 real-to-complex DFT

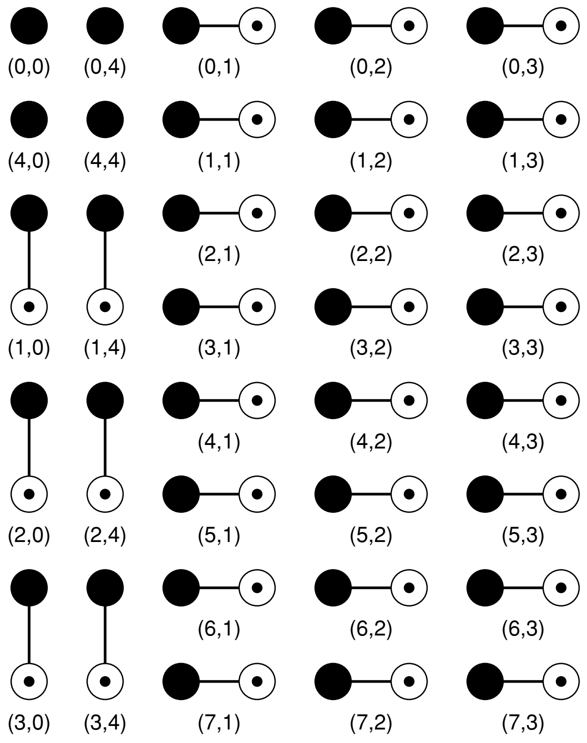


Notice that each row is packed like the one-dimensional transform in the earlier example. Packing yields 16 real values and 24 complex values in a total of 64 `sizeof(float)` memory locations, the size of the original matrix.

After each row is processed and packed, the DFT then transforms each column. Since the first pass leaves real values in columns zero and one, these two columns are transformed from real to complex using the same packing method vertically that was used horizontally on each row. Thus, transforming these 16 real values yields six complex points and four real points, occupying the 16 memory locations in columns zero and one.

The remaining three columns of complex values are transformed complex-to-complex, yielding 24 complex points as output, and requiring 48 memory locations to store. This brings the total storage requirement to 64 locations, the size of the original matrix. The completed 8-by-8 transform's format is shown in Figure 2-5 (page 13)

Figure 2-5 Packed format of an 8-by-8 real-to-complex 2D DFT



Scaling Fourier Transforms

To provide the best possible execution speeds, the vDSP library's functions don't always adhere strictly to textbook formulas for Fourier transforms, and must be scaled accordingly. The following sections specify the scaling for each type of Fourier transform implemented by the vDSP Library. The scaling factors are also stated explicitly in the formulas that accompany the function definitions in the reference chapter.

The scaling factors are summarized in Figure 2-6 (page 14) which shows the implemented values in terms of the mathematical values. The sections that follow describe the scaling factors individually.

Figure 2-6 Summary of the scaling factors

One-Dimensional Transforms

Real forward transforms: $RF_{imp} = RF_{math} * 2$

Real inverse transforms: $RI_{imp} = RI_{math} * n$

Complex forward transforms: $CF_{imp} = CF_{math}$

Complex inverse transforms: $CI_{imp} = CI_{math} * n$

Two-Dimensional Transforms

Real forward transforms: $RF2D_{imp} = RF2D_{math} * 2$

Real inverse transforms: $RI2D_{imp} = RI2D_{math} * mn$

Complex forward transforms: $CF2D_{imp} = CF2D_{math}$

Complex inverse transforms: $CI2D_{imp} = CI2D_{math} * mn$

Real Fourier Transforms

Figure 2-7 (page 14) shows the mathematical formula for the one-dimensional forward Fourier transform.

Figure 2-7 1D forward Fourier transforms, general formula

$$C_m = \sum_{n=0}^{N-1} C_n e^{\left[\frac{-i2\pi}{N}\right]^{nm}}$$

The values of the Fourier coefficients returned by the real forward transform as implemented (RF_{imp}) are equal to the mathematical values (RF_{math}) times 2:

$$RF_{imp} = RF_{math} * 2$$

Figure 2-8 (page 15) shows the mathematical formula for the inverse Fourier transform.

Figure 2-8 1D inverse Fourier transforms, general formula

$$C_m = \sum_{n=0}^{N-1} C_n e^{\left[\frac{i2\pi}{N}\right]^{nm}}$$

The values of the Fourier coefficients returned by the real inverse transform as implemented (`RI`$_{imp}$) are equal to the mathematical values (`RI`$_{math}$) times `N`:

$$\mathrm{RI}_{imp} = \mathrm{RI}_{math} * \mathrm{N}$$

Complex Fourier Transforms

Figure 2-7 (page 14) shows the mathematical formula for the one-dimensional forward Fourier transform.

The values of the Fourier coefficients returned by the complex forward transform as implemented (`CF`$_{imp}$) are equal to the mathematical values (`CF`$_{math}$):

$$\mathrm{CF}_{imp} = \mathrm{CF}_{math}$$

The values of the Fourier coefficients returned by the complex inverse transform as implemented (`CI`$_{imp}$) are equal to the mathematical values (`CI`$_{math}$) times `N`:

$$\mathrm{CI}_{imp} = \mathrm{CI}_{math} * \mathrm{N}$$

Real 2-Dimensional Fourier Transforms

Figure 2-9 (page 15) shows the mathematical formula for the 2-dimensional forward Fourier transform.

Figure 2-9 2D forward Fourier transforms, general formula

$$C_{nm} = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} C_{pq} e^{\left[\frac{-i2\pi}{N}\right]^{pn}} e^{\left[\frac{-i2\pi}{M}\right]^{qm}}$$

The values of the Fourier coefficients returned by the 2-dimensional real forward transform as implemented (`RF2D`$_{\text{imp}}$) are equal to the mathematical values (`R2DF`$_{\text{math}}$) times 2:

$$\mathrm{RF2D}_{\mathrm{imp}} = \mathrm{RF2D}_{\mathrm{math}} * 2$$

Figure 2-10 (page 16) shows the mathematical formula for the 2-dimensional inverse Fourier transform.

Figure 2-10 2D inverse Fourier transforms, general formula

$$C_{nm} = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} C_{pq} e^{\left[\frac{i2\pi}{N}\right]^{pn}} e^{\left[\frac{i2\pi}{M}\right]^{qm}}$$

The values of the Fourier coefficients returned by the 2-dimensional real inverse transform as implemented (`RF2D`$_{\text{imp}}$) are equal to the mathematical values (`R2DF`$_{\text{math}}$) times `MN`:

$$\mathrm{RI2D}_{\mathrm{imp}} = \mathrm{RI2D}_{\mathrm{math}} * \mathrm{MN}$$

Complex 2-Dimensional Fourier Transforms

Figure 2-9 (page 15) shows the mathematical formula for the 2-dimensional forward Fourier transform.

The values of the Fourier coefficients returned by the 2-dimensional complex forward transform as implemented (`CF2D`$_{\text{imp}}$) are equal to the mathematical values (`CF2D`$_{\text{math}}$):

$$\mathrm{CF2D}_{\mathrm{imp}} = \mathrm{CF2D}_{\mathrm{math}}$$

Figure 2-10 (page 16) shows the mathematical formula for the 2-dimensional inverse Fourier transform.

The values of the Fourier coefficients returned by the 2-dimensional complex inverse transform as implemented (`CI2D`$_{\text{imp}}$) are equal to the mathematical values (`CI2D`$_{\text{math}}$) times `MN`:

$$\mathrm{CI2D}_{\mathrm{imp}} = \mathrm{CI2D}_{\mathrm{math}} * \mathrm{mn}$$

Scaling Example

Using complex two-dimensional transforms as an example, when transforming data from the time domain to the frequency domain with `vDSP_fft_zop`, no scaling factor is introduced. Transforming the image from the frequency domain to the time domain with `vDSP_fft_zop`, however, introduces a factor of `N`, where `N` is the vector length.

In the following example, a vector is transformed to the frequency domain, an inverse transform reconstructs it in the time domain, and then it is scaled by `1/N` to recover the original input (perhaps altered by machine rounding errors).

Listing 2-1 Scaling example

```
    ~
    ~
scale = 1.0/((float) FFT_LENGTH) ;
    ~
    ~
/* to the frequency domain */
vDSP_fft_zop(   setup,
           &A,
           1,
           &B,
           1,
           FFT_LENGTH_LOG2,
           FFT_FORWARD,
           ) ;

/* back to the time domain */
vDSP_fft_zop(   setup,
           &B,
           1,
           &A,
           1,
           FFT_LENGTH_LOG2,
           FFT_INVERSE,
           ) ;
```

```
/*scale the result */
vDSP_vsmul(     A.realp,
           1,
           &scale,
           A.realp,
           1,
           FFT_LENGTH
           ) ;

vDSP_vsmul(     A.imagp,
           1,
           &scale,
           A.imagp,
           1,
           FFT_LENGTH
           ) ;
    ~
    ~
```

Document Revision History

This table describes the changes to *vDSP Programming Guide* .

Date	Notes
2013-04-23	Minor revisions and updates.
2011-04-14	Fixed minor technical errors.
2011-01-19	Fixed minor bugs.
2010-03-19	Added document to iOS documentation set.
2010-01-20	New document that describes vDSP, an API for performing math and DSP transforms with vectors.