Accelerate Framework Reference

Mathematical Computation

2010-03-19

Contents

Introduction

Framework:	Accelerate/Accelerate.h
Declared in	cblas.h vDSP.h

This document describes the Accelerate Framework, which contains C APIs for vector and matrix math, digital signal processing, large number handling, and image processing.

Other References

PART I

Other References

BLAS Reference

Framework:	Accelerate
Declared in	cblas.h vBLAS.h

Overview

The vecLib framework contains nine C header files (not counting vecLib.h which merely includes the others).

This document describes the functions declared in the header files `cblas.h` and `vblas.h`, which contain the interfaces for Apple's implementation of the Basic Linear Algebra Subprograms (BLAS) API.

For More Information

Documentation on the BLAS standard, including reference implementations, can be found on the web starting from the BLAS FAQ page at these URLs (verified live as of July 2005): http://www.netlib.org/blas/faq.html and http://www.netlib.org/blas/blast-forum/blast-forum.html

Functions by Task

General Functions

`ATLU_DestroyThreadMemory` (page 18)
Releases per-thread storage associated with BLAS.

`SetBLASParamErrorProc` (page 161)
Sets an error handler function.

`cblas_errprn` (page 91)
Prints an error message.

`cblas_xerbla` (page 126)
The default error handler for BLAS routines.

`cblas_icamax` (page 91)
Returns the index of the element with the largest absolute value in a vector (single-precision complex).

`cblas_idamax` (page 92)
Returns the index of the element with the largest absolute value in a vector (double-precision).

`cblas_isamax` (page 93)
: Returns the index of the element with the largest absolute value in a vector (single-precision).

`cblas_izamax` (page 93)
: Returns the index of the element with the largest absolute value in a vector (double-precision complex).

CATLAS and CBLAS Vector Functions

`catlas_caxpby` (page 18)
: Computes the product of two vectors, scaling each one separately (single-precision complex).

`catlas_cset` (page 19)
: Modifies a vector (single-precision complex) in place, setting each element to a given value.

`catlas_daxpby` (page 19)
: Computes the product of two vectors, scaling each one separately (double-precision).

`catlas_dset` (page 20)
: Modifies a vector (double-precision) in place, setting each element to a given value.

`catlas_saxpby` (page 21)
: Computes the product of two vectors, scaling each one separately (single-precision).

`catlas_sset` (page 21)
: Modifies a vector (single-precision) in place, setting each element to a given value.

`catlas_zaxpby` (page 22)
: Computes the product of two vectors, scaling each one separately (double-precision complex).

`catlas_zset` (page 23)
: Modifies a vector (double-precision complex) in place, setting each element to a given value.

`cblas_sdot` (page 96)
: Computes the dot product of two vectors (single-precision).

`cblas_sdsdot` (page 97)
: Computes the dot product of two single-precision vectors plus an initial single-precision value.

`cblas_cdotc_sub` (page 24)
: Calculates the dot product of the complex conjugate of a single-precision complex vector with a second single-precision complex vector.

`cblas_cdotu_sub` (page 25)
: Computes the dot product of two single-precision complex vectors.

`cblas_ddot` (page 59)
: Computes the dot product of two vectors (double-precision).

`cblas_dsdot` (page 71)
: Computes the double-precision dot product of a pair of single-precision vectors.

`cblas_zdotc_sub` (page 128)
: Calculates the dot product of the complex conjugate of a double-precision complex vector with a second double-precision complex vector.

`cblas_zdotu_sub` (page 129)
: Computes the dot product of two double-precision complex vectors.

Single-Precision Float Matrix Functions

`cblas_sasum` (page 94)
: Computes the sum of the absolute values of elements in a vector (single-precision).

`cblas_saxpy` (page 94)
: Computes a constant times a vector plus a vector (single-precision).

`cblas_scopy` (page 96)
: Copies a vector to another vector (single-precision).

`cblas_sgbmv` (page 98)
: Scales a general band matrix, then multiplies by a vector, then adds a vector (single precision).

`cblas_sgemm` (page 99)
: Multiplies two matrices (single-precision).

`cblas_sgemv` (page 100)
: Multiplies a matrix by a vector (single precision).

`cblas_sger` (page 102)
: Multiplies vector X by the transform of vector Y, then adds matrix A (single precison).

`cblas_snrm2` (page 103)
: Computes the L2 norm (Euclidian length) of a vector (single precision).

`cblas_srot` (page 103)
: Applies a Givens rotation matrix to a pair of vectors.

`cblas_srotg` (page 104)
: Constructs a Givens rotation matrix.

`cblas_srotm` (page 104)
: Applies a modified Givens transformation (single precision).

`cblas_srotmg` (page 106)
: Generates a modified Givens rotation matrix.

`cblas_ssbmv` (page 107)
: Scales a symmetric band matrix, then multiplies by a vector, then adds a vector (single-precision).

`cblas_sscal` (page 108)
: Multiplies each element of a vector by a constant (single-precision).

`cblas_sspmv` (page 108)
: Scales a packed symmetric matrix, then multiplies by a vector, then scales and adds another vector (single precision).

`cblas_sspr` (page 109)
: Rank one update: adds a packed symmetric matrix to the product of a scaling factor, a vector, and its transpose (single precision).

`cblas_sspr2` (page 110)
: Rank two update of a packed symmetric matrix using two vectors (single precision).

`cblas_sswap` (page 111)
: Exchanges the elements of two vectors (single precision).

`cblas_ssymm` (page 111)
: Multiplies a matrix by a symmetric matrix (single-precision).

`cblas_ssymv` (page 113)
: Scales a symmetric matrix, multiplies by a vector, then scales and adds another vector (single precision).

`cblas_ssyr` (page 114)
Rank one update: adds a symmetric matrix to the product of a scaling factor, a vector, and its transpose (single precision).

`cblas_ssyr2` (page 115)
Rank two update of a symmetric matrix using two vectors (single precision).

`cblas_ssyr2k` (page 116)
Performs a rank-2k update of a symmetric matrix (single precision).

`cblas_ssyrk` (page 117)
Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (single precision).

`cblas_stbmv` (page 118)
Scales a triangular band matrix, then multiplies by a vector (single precision).

`cblas_stbsv` (page 119)
Solves a triangular banded system of equations.

`cblas_stpmv` (page 120)
Multiplies a triangular matrix by a vector, then adds a vector (single precision).

`cblas_stpsv` (page 121)
Solves a packed triangular system of equations.

`cblas_strmm` (page 122)
Scales a triangular matrix and multiplies it by a matrix.

`cblas_strmv` (page 123)
Multiplies a triangular matrix by a vector.

`cblas_strsm` (page 124)
Solves a triangular system of equations with multiple values for the right side.

`cblas_strsv` (page 125)
Solves a triangular system of equations with a single value for the right side.

Single-Precision Complex Matrix Functions

`cblas_caxpy` (page 23)
Computes a constant times a vector plus a vector (single-precision complex).

`cblas_ccopy` (page 24)
Copies a vector to another vector (single-precision complex).

`cblas_cgbmv` (page 26)
Scales a general band matrix, then multiplies by a vector, then adds a vector (single-precision complex).

`cblas_cgemm` (page 27)
Multiplies two matrices (single-precision complex).

`cblas_cgemv` (page 29)
Multiplies a matrix by a vector (single-precision complex).

`cblas_cgerc` (page 30)
Multiplies vector X by the conjugate transform of vector Y, then adds matrix A (single-precision complex).

`cblas_cgeru` (page 31)
Multiplies vector X by the transform of vector Y, then adds matrix A (single-precision complex).

`cblas_chbmv` (page 32)
: Scales a Hermitian band matrix, then multiplies by a vector, then adds a vector (single-precision complex).

`cblas_chemm` (page 33)
: Multiplies two Hermitian matrices (single-precision complex), then adds a third (with scaling).

`cblas_chemv` (page 34)
: Scales and multiplies a Hermitian matrix by a vector, then adds a second (scaled) vector.

`cblas_cher` (page 35)
: Hermitian rank 1 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of `X` to matrix `A`.

`cblas_cher2` (page 36)
: Hermitian rank 2 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of vector `Y` to the product of the conjugate of the scaling factor, vector `Y`, and the conjugate transpose of vector `X`, and adds the result to matrix `A`.

`cblas_cher2k` (page 37)
: Performs a rank-2k update of a complex Hermitian matrix (single-precision complex).

`cblas_cherk` (page 38)
: Rank-k update—multiplies a Hermitian matrix by its transpose and adds a second matrix (single precision).

`cblas_chpmv` (page 39)
: Scales a packed hermitian matrix, multiplies it by a vector, and adds a scaled vector.

`cblas_chpr` (page 40)
: Scales and multiplies a vector times its conjugate transpose, then adds a matrix.

`cblas_chpr2` (page 41)
: Multiplies a vector times the conjugate transpose of a second vector and vice-versa, sums the results, and adds a matrix.

`cblas_crotg` (page 42)
: Constructs a complex Givens rotation.

`cblas_cscal` (page 43)
: Multiplies each element of a vector by a constant (single-precision complex).

`cblas_csrot` (page 44)
: Applies a Givens rotation matrix to a pair of complex vectors.

`cblas_csscal` (page 44)
: Multiplies each element of a vector by a constant (single-precision complex).

`cblas_cswap` (page 45)
: Exchanges the elements of two vectors (single-precision complex).

`cblas_csymm` (page 46)
: Multiplies a matrix by a symmetric matrix (single-precision complex).

`cblas_csyr2k` (page 47)
: Performs a rank-2k update of a symmetric matrix (single-precision complex).

`cblas_csyrk` (page 48)
: Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (single-precision complex).

`cblas_ctbmv` (page 49)
: Scales a triangular band matrix, then multiplies by a vector (single-precision compex).

`cblas_ctbsv` (page 50)
: Solves a triangular banded system of equations.

`cblas_ctpmv` (page 51)
: Multiplies a triangular matrix by a vector, then adds a vector (single-precision complex).

`cblas_ctpsv` (page 52)
: Solves a packed triangular system of equations.

`cblas_ctrmm` (page 53)
: Scales a triangular matrix and multiplies it by a matrix.

`cblas_ctrmv` (page 55)
: Multiplies a triangular matrix by a vector.

`cblas_ctrsm` (page 56)
: Solves a triangular system of equations with multiple values for the right side.

`cblas_ctrsv` (page 57)
: Solves a triangular system of equations with a single value for the right side.

`cblas_scasum` (page 95)
: Computes the sum of the absolute values of real and imaginary parts of elements in a vector (single-precision complex).

`cblas_scnrm2` (page 95)
: Computes the unitary norm of a vector (single-precision complex).

Double-Precision Float Matrix Functions

`cblas_dasum` (page 58)
: Computes the sum of the absolute values of elements in a vector (double-precision).

`cblas_daxpy` (page 58)
: Computes a constant times a vector plus a vector (double-precision).

`cblas_dcopy` (page 59)
: Copies a vector to another vector (double-precision).

`cblas_dgbmv` (page 60)
: Scales a general band matrix, then multiplies by a vector, then adds a vector (double precision).

`cblas_dgemm` (page 61)
: Multiplies two matrices (double-precision).

`cblas_dgemv` (page 63)
: Multiplies a matrix by a vector (double precision).

`cblas_dger` (page 64)
: Multiplies vector X by the transform of vector Y, then adds matrix A (double precison).

`cblas_dnrm2` (page 65)
: Computes the L2 norm (Euclidian length) of a vector (double precision).

`cblas_drot` (page 66)
: Applies a Givens rotation matrix to a pair of vectors.

`cblas_drotg` (page 66)
: Constructs a Givens rotation matrix.

`cblas_drotm` (page 67)
: Applies a modified Givens transformation (single precision).

`cblas_drotmg` (page 68)
: Generates a modified Givens rotation matrix.

`cblas_dsbmv` (page 69)
: Scales a symmetric band matrix, then multiplies by a vector, then adds a vector (double precision).

`cblas_dscal` (page 70)
: Multiplies each element of a vector by a constant (double-precision).

`cblas_dspmv` (page 71)
: Scales a packed symmetric matrix, then multiplies by a vector, then scales and adds another vector (double precision).

`cblas_dspr` (page 72)
: Rank one update: adds a packed symmetric matrix to the product of a scaling factor, a vector, and its transpose (double precision).

`cblas_dspr2` (page 73)
: Rank two update of a packed symmetric matrix using two vectors (single precision).

`cblas_dswap` (page 74)
: Exchanges the elements of two vectors (double precision).

`cblas_dsymm` (page 75)
: Multiplies a matrix by a symmetric matrix (double-precision).

`cblas_dsymv` (page 77)
: Scales a symmetric matrix, multiplies by a vector, then scales and adds another vector (single precision).

`cblas_dsyr` (page 78)
: Rank one update: adds a symmetric matrix to the product of a scaling factor, a vector, and its transpose (double precision).

`cblas_dsyr2` (page 78)
: Rank two update of a symmetric matrix using two vectors (single precision).

`cblas_dsyr2k` (page 79)
: Performs a rank-2k update of a symmetric matrix (double precision).

`cblas_dsyrk` (page 81)
: Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (double precision).

`cblas_dtbmv` (page 82)
: Scales a triangular band matrix, then multiplies by a vector (double precision).

`cblas_dtbsv` (page 83)
: Solves a triangular banded system of equations.

`cblas_dtpmv` (page 84)
: Multiplies a triangular matrix by a vector, then adds a vector (double precision).

`cblas_dtpsv` (page 85)
: Solves a packed triangular system of equations.

`cblas_dtrmm` (page 85)
: Scales a triangular matrix and multiplies it by a matrix.

`cblas_dtrmv` (page 87)
: Multiplies a triangular matrix by a vector.

`cblas_dtrsm` (page 88)
: Solves a triangular system of equations with multiple values for the right side.

`cblas_dtrsv` (page 89)
Solves a triangular system of equations with a single value for the right side.

Double-Precision Complex Matrix Functions

`cblas_dzasum` (page 90)
Computes the sum of the absolute values of real and imaginary parts of elements in a vector (single-precision complex).

`cblas_dznrm2` (page 90)
Computes the unitary norm of a vector (double-precision complex).

`cblas_zaxpy` (page 127)
Computes a constant times a vector plus a vector (double-precision complex).

`cblas_zcopy` (page 127)
Copies a vector to another vector (double-precision complex).

`cblas_zdrot` (page 129)
Applies a Givens rotation matrix to a pair of complex vectors.

`cblas_zdscal` (page 130)
Multiplies each element of a vector by a constant (double-precision complex).

`cblas_zgbmv` (page 130)
Scales a general band matrix, then multiplies by a vector, then adds a vector (double-precision complex).

`cblas_zgemm` (page 132)
Multiplies two matrices (double-precision complex).

`cblas_zgemv` (page 133)
Multiplies a matrix by a vector (double-precision complex).

`cblas_zgerc` (page 135)
Multiplies vector X by the conjugate transform of vector Y, then adds matrix A (double-precision complex).

`cblas_zgeru` (page 136)
Multiplies vector X by the transform of vector Y, then adds matrix A (double-precision complex).

`cblas_zhbmv` (page 137)
Scales a Hermitian band matrix, then multiplies by a vector, then adds a vector (double-precision complex).

`cblas_zhemm` (page 138)
Multiplies two Hermitian matrices (double-precision complex).

`cblas_zhemv` (page 139)
Scales and multiplies a Hermitian matrix by a vector, then adds a second (scaled) vector.

`cblas_zher` (page 140)
Adds the product of a scaling factor, vector `X`, and the conjugate transpose of `X` to matrix `A`.

`cblas_zher2` (page 141)
Hermitian rank 2 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of vector `Y` to the product of the conjugate of the scaling factor, vector `Y`, and the conjugate transpose of vector `X`, and adds the result to matrix `A`.

`cblas_zher2k` (page 142)
Performs a rank-2k update of a complex Hermitian matrix (double-precision complex).

`cblas_zherk` (page 143)
Rank-k update—multiplies a Hermitian matrix by its transpose and adds a second matrix (single precision).

`cblas_zhpmv` (page 144)
Scales a packed hermitian matrix, multiplies it by a vector, and adds a scaled vector.

`cblas_zhpr` (page 145)
Scales and multiplies a vector times its conjugate transpose, then adds a matrix.

`cblas_zhpr2` (page 146)
Multiplies a vector times the conjugate transpose of a second vector and vice-versa, sums the results, and adds a matrix.

`cblas_zrotg` (page 147)
Constructs a complex Givens rotation.

`cblas_zscal` (page 147)
Multiplies each element of a vector by a constant (double-precision complex).

`cblas_zswap` (page 148)
Exchanges the elements of two vectors (double-precision complex).

`cblas_zsymm` (page 149)
Multiplies a matrix by a symmetric matrix (double-precision complex).

`cblas_zsyr2k` (page 150)
Performs a rank-2k update of a symmetric matrix (double-precision complex).

`cblas_zsyrk` (page 151)
Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (double-precision complex).

`cblas_ztbmv` (page 152)
Scales a triangular band matrix, then multiplies by a vector (double-precision complex).

`cblas_ztbsv` (page 153)
Solves a triangular banded system of equations.

`cblas_ztpmv` (page 154)
Multiplies a triangular matrix by a vector, then adds a vector (double-precision compex).

`cblas_ztpsv` (page 155)
Solves a packed triangular system of equations.

`cblas_ztrmm` (page 156)
Scales a triangular matrix and multiplies it by a matrix.

`cblas_ztrmv` (page 158)
Multiplies a triangular matrix by a vector.

`cblas_ztrsm` (page 159)
Solves a triangular system of equations with multiple values for the right side.

`cblas_ztrsv` (page 160)
Solves a triangular system of equations with a single value for the right side.

Functions

ATLU_DestroyThreadMemory

Releases per-thread storage associated with BLAS.

```
void ATLU_DestroyThreadMemory (
   void
);
```

Discussion
Many BLAS functions parallelize computation to obtain performance from multiple CPU cores. These functions allocate thread memory. Call this function when you no longer need to call BLAS functions.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_caxpby

Computes the product of two vectors, scaling each one separately (single-precision complex).

```
void catlas_caxpby (
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

N
Number of elements in the vector.

alpha
Scaling factor for `X`.

X
Input vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
Scaling factor for Y.

Y
Input vector `Y`.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector `Y` are replaced with the result.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_cset

Modifies a vector (single-precision complex) in place, setting each element to a given value.

```
void catlas_cset (
   const int N,
   const void *alpha,
   void *X,
   const int incX
);
```

Parameters

N

The number of elements in the vector.

alpha

The new value.

X

The vector to modify.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_daxpby

Computes the product of two vectors, scaling each one separately (double-precision).

```
void catlas_daxpby (
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

N

Number of elements in the vector.

alpha
    Scaling factor for `X`.

X
    Input vector `X`.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
    Scaling factor for Y.

Y
    Input vector `Y`.

incY
    Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector `Y` are replaced with the result.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_dset

Modifies a vector (double-precision) in place, setting each element to a given value.

```
void catlas_dset (
   const int N,
   const double alpha,
   double *X,
   const int incX
);
```

Parameters

N
    The number of elements in the vector.

alpha
    The new value.

X
    The vector to modify.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_saxpby

Computes the product of two vectors, scaling each one separately (single-precision).

```
void catlas_saxpby (
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

N

Number of elements in the vector.

alpha

Scaling factor for `X`.

X

Input vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor for Y.

Y

Input vector `Y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

On return, the contents of vector `Y` are replaced with the result.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

catlas_sset

Modifies a vector (single-precision) in place, setting each element to a given value.

```
void catlas_sset (
   const int N,
   const float alpha,
   float *X,
   const int incX
);
```

Parameters

N

The number of elements in the vector.

alpha
The new value.

X
The vector to modify.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_zaxpby

Computes the product of two vectors, scaling each one separately (double-precision complex).

```
void catlas_zaxpby (
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

N
Number of elements in the vector.

alpha
Scaling factor for `X`.

X
Input vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
Scaling factor for Y.

Y
Input vector `Y`.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector `Y` are replaced with the result.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

catlas_zset

Modifies a vector (double-precision complex) in place, setting each element to a given value.

```
void catlas_zset (
   const int N,
   const void *alpha,
   void *X,
   const int incX
);
```

Parameters

N

The number of elements in the vector.

alpha

The new value.

X

The vector to modify.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_caxpy

Computes a constant times a vector plus a vector (single-precision complex).

```
void cblas_caxpy (
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N

Number of elements in the vectors.

alpha

Scaling factor for the values in `X`.

X

Input vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Input vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector Y are replaced with the result. The value computed is `(alpha * X[i]) + Y[i]`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ccopy

Copies a vector to another vector (single-precision complex).

```
void cblas_ccopy (
   const int N,
   const void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

X
: Source vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Destination vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cdotc_sub

Calculates the dot product of the complex conjugate of a single-precision complex vector with a second single-precision complex vector.

```
void cblas_cdotc_sub (
   const int N,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *dotc
);
```

Parameters

N

Number of elements in vectors `X` and `Y`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `Y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

dotc

The result vector.

Discussion

Computes conjg(X) * Y.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_cdotu_sub

Computes the dot product of two single-precision complex vectors.

```
void cblas_cdotu_sub (
   const int N,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *dotu
);
```

Parameters

N

The length of vectors `X` and `Y`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

dotu
: The result vector.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cgbmv

Scales a general band matrix, then multiplies by a vector, then adds a vector (single-precision complex).

```
void cblas_cgbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const int KL,
   const int KU,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

M
: Number of rows in matrix A.

N
: Number of columns in matrix A.

KL
: Number of subdiagonals in matrix A.

KU
: Number of superdiagonals in matrix A.

alpha
: Scaling factor to multiply matrix `A` by.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`. (Must be at least `KL+KU+1`.)

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor to multiply vector Y by.

Y
: Vector Y.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y`, `alpha*A'*x + beta*y`, or `alpha*conjg(A')*x + beta*y` depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cgemm

Multiplies two matrices (single-precision complex).

```
void cblas_cgemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_TRANSPOSE TransB,
   const int M,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
: Specifies whether to transpose matrix `A`.

TransB
: Specifies whether to transpose matrix B.

M
: Number of rows in matrices A and C.

N
: Number of columns in matrices B and C.

K
: Number of columns in matrix A; number of rows in matrix B.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix A; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix B; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix C; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion
This function multiplies `A * B` and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

`C←αAB +  C`

or

`C←αBA +  C`

with optional use of transposed forms of A, B, or both.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cgemv

Multiplies a matrix by a vector (single-precision complex).

```
void cblas_cgemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
: Specifies whether to transpose matrix `A`.

M
: Number of rows in matrix `A`.

N
: Number of columns in matrix `A`.

alpha
: Scaling factor for the product of matrix `A` and vector `X`.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

X
: Vector X.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor for vector `Y`.

Y
: Vector Y

incY
: Stride within Y. For example, if incY is 7, every 7th element is used.

Discussion

This function multiplies `A * X` (after transposing `A`, if needed) and multiplies the resulting matrix by `alpha`. It then multiplies vector `Y` by `beta`. It stores the sum of these two products in vector `Y`.

Thus, it calculates either

```
Y←αAX +  Y
```

with optional use of the transposed form of `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cgerc

Multiplies vector X by the conjugate transform of vector Y, then adds matrix A (single-precision complex).

```
void cblas_cgerc (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

M
: Number of rows in matrix `A`.

N
: Number of columns in matrix `A`.

alpha
: Scaling factor for vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion
Computes `alpha*x*conjg(y') + A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cgeru

Multiplies vector X by the transform of vector Y, then adds matrix A (single-precision complex).

```
void cblas_cgeru (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

M
: Number of rows in matrix `A`.

N
: Number of columns in matrix `A`.

alpha
: Scaling factor for vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion
Computes `alpha*x*y' + A`.

Availability
Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_chbmv

Scales a Hermitian band matrix, then multiplies by a vector, then adds a vector (single-precision complex).

```
void cblas_chbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

The order of matrix `A`.

K

Half-bandwidth of matrix `A`.

alpha

Scaling value to multiply matrix A by.

A

Matrix `A`.

lda

The leading dimension of array containing matrix `A`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor that vector `Y` is multiplied by.

Y

Vector `Y`. Replaced by results on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and returns the results in vector `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_chemm

Multiplies two Hermitian matrices (single-precision complex), then adds a third (with scaling).

```
void cblas_chemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrices should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
: The number of rows in matrices `A` and `C`.

N
: The number of columns in matrices `B` and `C`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + \ C$

or

$C \leftarrow \alpha BA + \ C$

where

$A = A^H$

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_chemv

Scales and multiplies a Hermitian matrix by a vector, then adds a second (scaled) vector.

```
void cblas_chemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

The order of matrix `A`.

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor for vector `X`.

Y

Vector `Y`. Overwritten by results on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Calculates `Y←αAX +  Y`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_cher

Hermitian rank 1 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of `X` to matrix `A`.

```
void cblas_cher (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const void *X,
   const int incX,
   void *A,
   const int lda
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: The order of matrix `A`.

alpha
: The scaling factor for vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of matrix `A`.

Discussion

Computes `A←αX*conjg(X') +  A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_cher2

Hermitian rank 2 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of vector `Y` to the product of the conjugate of the scaling factor, vector `Y`, and the conjugate transpose of vector `X`, and adds the result to matrix `A`.

```
void cblas_cher2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: The order of matrix `A`.

alpha
: The scaling factor α.

X
: Vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Y
: Vector Y.

incY
: Stride within Y. For example, if incY is 7, every 7th element is used.

A
: Matrix A.

lda
: The leading dimension of matrix A.

Discussion
Computes `A←αX*conjg(Y') + conjg(α)*Y*conjg(X') + A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cher2k

Performs a rank-2k update of a complex Hermitian matrix (single-precision complex).

```
void cblas_cher2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const float beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are 'U' or 'L'.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N
: Order of matrix `C`.

K
: Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

  Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

B
: Matrix `B`.

ldb
: Leading dimension of array containing matrix `B`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion
Computes `alpha*A*B`^{H} `+ alpha*B*A`^{H} `+beta*C`

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_cherk

Rank-k update—multiplies a Hermitian matrix by its transpose and adds a second matrix (single precision).

```
void cblas_cherk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const float alpha,
   const void *A,
   const int lda,
   const float beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`) or the conjugate transpose of A (`'C'` or `'c'`).

N
: Order of matrix `C`.

K
: Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Calculates `alpha*A*A`^{H} `+ beta*C`; if transposed, calculates `alpha*A`^{H}`*A + beta*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_chpmv

Scales a packed hermitian matrix, multiplies it by a vector, and adds a scaled vector.

```
void cblas_chpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *Ap,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha

Scaling factor that matrix `A` is multiplied by.

Ap

Matrix `A`.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor that vector `y` is multiplied by.

Y

Vector `y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_chpr

Scales and multiplies a vector times its conjugate transpose, then adds a matrix.

```
void cblas_chpr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const void *X,
   const int incX,
   void *A
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
Order of matrix `A` and the number of elements in vector `x`.

alpha
Scaling factor that vector `x` is multiplied by.

X
Vector `x`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A
Matrix `A`. Overwritten by results on return.

Discussion
Calculates `alpha*x*conjg(x') + A` and stores the result in `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_chpr2

Multiplies a vector times the conjugate transpose of a second vector and vice-versa, sums the results, and adds a matrix.

```
void cblas_chpr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *Ap
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha
: Scaling factor that vector `x` is multiplied by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Ap
: Matrix `A` in packed storage format. Overwritten by the results on return.

Discussion
Calcuates `alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A`, and stores the result in `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_crotg

Constructs a complex Givens rotation.

```
void cblas_crotg (
   void *a,
   void *b,
   void *c,
   void *s
);
```

Parameters

a

Complex value `a`. Overwritten on output.

b

Complex value `a`.

c

Real value `c`. Unused on entry. Overwritten on return with the value `cos( )`.

s

Complex value `s`. Unused on entry. Overwritten on return with the value `sin( )`.

Discussion

Given a vertical matrix containing `a` and `b`, computes the values of `cos` and `sin` that zero the lower value (`b`). Returns the value of `sin` in `s`, the value of `cos` in `c`, and the upper value (`r`) in `a`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_cscal

Multiplies each element of a vector by a constant (single-precision complex).

```
void cblas_cscal (
   const int N,
   const void *alpha,
   void *X,
   const int incX
);
```

Parameters

N

The number of elements in the vector.

alpha

The constant scaling factor.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_csrot

Applies a Givens rotation matrix to a pair of complex vectors.

```
void cblas_csrot (
   const int N,
   void *X,
   const int incX,
   void *Y,
   const int incY,
   const float c,
   const float s
);
```

Parameters

N
: The number of elements in vectors `X` and `Y`.

X
: Vector `X`. Modified on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`. Modified on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

c
: The value `cos( )` in the Givens rotation matrix.

s
: The value `sin( )` in the Givens rotation matrix.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_csscal

Multiplies each element of a vector by a constant (single-precision complex).

```
void cblas_csscal (
   const int N,
   const float alpha,
   void *X,
   const int incX
);
```

Parameters

N

The number of elements in vector x.

alpha

The constant scaling factor.

X

Vector X.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_cswap

Exchanges the elements of two vectors (single-precision complex).

```
void cblas_cswap (
   const int N,
   void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N

Number of elements in vectors

X

Vector x. On return, contains elements copied from vector y.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

Y

Vector y. On return, contains elements copied from vector x.

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_csymm

Multiplies a matrix by a symmetric matrix (single-precision complex).

```
void cblas_csymm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrices should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + \ C$

or

$C \leftarrow \alpha BA + \ C$

where

$A = A^T$

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_csyr2k

Performs a rank-2k update of a symmetric matrix (single-precision complex).

```
void cblas_csyr2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N
: Order of matrix `C`.

K

Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`.

B

Matrix `B`.

ldb

Leading dimension of array containing matrix `B`.

beta

Scaling factor for matrix `C`.

C

Matrix `C`.

ldc

Leading dimension of array containing matrix `C`.

Discussion

Computes `alpha*A*B`^{T} `+ alpha*B*A`^{T} `+beta*C`

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_csyrk

Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (single-precision complex).

```
void cblas_csyrk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'` or `'t'`).

N
: Order of matrix `C`.

K
: Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion
Calculates `alpha*A*A`^{T} `+ beta*C`; if transposed, calculates `alpha*A`^{T}`*A + beta*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctbmv

Scales a triangular band matrix, then multiplies by a vector (single-precision compex).

```
void cblas_ctbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

The order of matrix `A`.

K

Half-bandwidth of matrix `A`.

A

Matrix `A`.

lda

The leading dimension of array containing matrix `A`.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Computes `A*x` and stores the results in `x`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ctbsv

Solves a triangular banded system of equations.

```
void cblas_ctbsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

Order of matrix `A`.

K

Number of superdiagonals or subdiagonals of matrix `A` (depending on the value of `Uplo`).

A

Matrix `A`.

lda

The leading dimension of matrix `A`.

X

Contains vector `B` on entry. Overwritten with vector `X` on return.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ctpmv

Multiplies a triangular matrix by a vector, then adds a vector (single-precision complex).

```
void cblas_ctpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *Ap,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

Ap
: Matrix `A`.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Computes `A*x`, `A`^{T}`*x`, or `conjg(A`^{T}`)*x` and stores the results in `X`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctpsv

Solves a packed triangular system of equations.

```
void cblas_ctpsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *Ap,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A`.

Ap
: Matrix `A` (in packed storage format).

X
: Contains vector `B` on entry. Overwritten with vector `X` on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctrmm

Scales a triangular matrix and multiplies it by a matrix.

```
void cblas_ctrmm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   void *B,
   const int ldb
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Side

Determines the order in which the matrices should be multiplied.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M

Number of rows in matrix `B`.

N

Number of columns in matrix `B`.

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

B

Matrix `B`. Overwritten by results on return.

ldb

Leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, multiplies `alpha*A*B` or `alpha*A'*B`, depending on `TransA`.

If `Side` is `'R'`, multiplies `alpha*B*A` or `alpha*B*A'`, depending on `TransA`.

In either case, the results are stored in matrix `B`.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctrmv

Multiplies a triangular matrix by a vector.

```
void cblas_ctrmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of matrix `A`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Multiplies `A*X` or `A'*X`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctrsm

Solves a triangular system of equations with multiple values for the right side.

```
void cblas_ctrsm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   void *B,
   const int ldb
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Side

Determines the order in which the matrix and vector should be multiplied.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M

The number of rows in matrix `B`.

N

The number of columns in matrix `B`.

alpha

Scaling factor for matrix `A`.

A

Triangular matrix `A`.

lda

The leading dimension of matrix `B`.

B

On entry, matrix `B`. Overwritten on return by matrix `X`.

ldb

The leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, solves `A*X=alpha*B` or `A'*X=alpha*B`, depending on `TransA`.

If `Side` is `'R'`, solves `X*A=alpha*B` or `X*A'=alpha*B`, depending on `TransA`.

In either case, the results overwrite the values of matrix `B` in `X`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ctrsv

Solves a triangular system of equations with a single value for the right side.

```
void cblas_ctrsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
The order of matrix `A`.

A
Triangular matrix `A`.

lda
The leading dimension of matrix `B`.

X
The vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves `A*x=b` or `A'*x=b` where `x` and `b` are elements in `X` and `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dasum

Computes the sum of the absolute values of elements in a vector (double-precision).

```
double cblas_dasum (
   const int N,
   const double *X,
   const int incX
);
```

Parameters

N
: The number of elements in vector `x`.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value
Returns the sum.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_daxpy

Computes a constant times a vector plus a vector (double-precision).

```
void cblas_daxpy (
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   double *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

alpha
: Scaling factor for the values in `X`.

X
: Input vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Input vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector Y are replaced with the result. The value computed is `(alpha * X[i]) + Y[i]`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dcopy

Copies a vector to another vector (double-precision).

```
void cblas_dcopy (
   const int N,
   const double *X,
   const int incX,
   double *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

X
: Source vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Destination vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ddot

Computes the dot product of two vectors (double-precision).

```
double cblas_ddot (
   const int N,
   const double *X,
   const int incX,
   const double *Y,
   const int incY
);
```

Parameters

N

The number of elements in the vectors.

X

Vector X.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

Y

Vector Y.

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_dgbmv

Scales a general band matrix, then multiplies by a vector, then adds a vector (double precision).

```
void cblas_dgbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const int KL,
   const int KU,
   const double alpha,
   const double *A,
   const int lda,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

TransA

Specifies whether to use matrix A ('N' or 'n') or the transpose of A ('T', 't', 'C', or 'c').

M
Number of rows in matrix A.

N
Number of columns in matrix A.

KL
Number of subdiagonals in matrix A.

KU
Number of superdiagonals in matrix A.

alpha
Scaling factor to multiply matrix `A` by.

A
Matrix `A`.

lda
Leading dimension of array containing matrix `A`. (Must be at least `KL+KU+1`.)

X
Vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
Scaling factor to multiply vector Y by.

Y
Vector Y.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` or `alpha*A'*x + beta*y` depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dgemm

Multiplies two matrices (double-precision).

```
void cblas_dgemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_TRANSPOSE TransB,
   const int M,
   const int N,
   const int K,
   const double alpha,
   const double *A,
   const int lda,
   const double *B,
   const int ldb,
   const double beta,
   double *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
: Specifies whether to transpose matrix `A`.

TransB
: Specifies whether to transpose matrix `B`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

K
: Number of columns in matrix `A`; number of rows in matrix `B`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion
This function multiplies `A * B` and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

`C←αAB +  C`

or

`C←αBA +  C`

with optional use of transposed forms of `A`, `B`, or both.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dgemv

Multiplies a matrix by a vector (double precision).

```
void cblas_dgemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const double alpha,
   const double *A,
   const int lda,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
Specifies whether to transpose matrix `A`.

M
Number of rows in matrix `A`.

N
Number of columns in matrix `A`.

alpha
Scaling factor for the product of matrix `A` and vector `X`.

A
Matrix A.

lda
: The size of the first dimention of matrix A; if you are passing a matrix A[m][n], the value should be m.

X
: Vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

beta
: Scaling factor for vector Y.

Y
: Vector Y

incY
: Stride within Y. For example, if incY is 7, every 7th element is used.

Discussion

This function multiplies A * X (after transposing A, if needed) and multiplies the resulting matrix by alpha. It then multiplies vector Y by beta. It stores the sum of these two products in vector Y.

Thus, it calculates either

$Y \leftarrow \alpha AX + Y$

with optional use of the transposed form of A.

Availability

Available in iOS 4.0 and later.

Declared In

cblas.h

cblas_dger

Multiplies vector X by the transform of vector Y, then adds matrix A (double precison).

```
void cblas_dger (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   const double *Y,
   const int incY,
   double *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

M
: Number of rows in matrix A.

N
: Number of columns in matrix `A`.

alpha
: Scaling factor for vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion

Computes `alpha*x*y' + A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dnrm2

Computes the L2 norm (Euclidian length) of a vector (double precision).

```
double cblas_dnrm2 (
   const int N,
   const double *X,
   const int incX
);
```

Parameters

N
: Length of vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_drot

Applies a Givens rotation matrix to a pair of vectors.

```
void cblas_drot (
   const int N,
   double *X,
   const int incX,
   double *Y,
   const int incY,
   const double c,
   const double s
);
```

Parameters

N
: The number of elements in vectors `X` and `Y`.

X
: Vector `X`. Modified on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`. Modified on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

c
: The value `cos( )` in the Givens rotation matrix.

s
: The value `sin( )` in the Givens rotation matrix.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_drotg

Constructs a Givens rotation matrix.

```
void cblas_drotg (
   double *a,
   double *b,
   double *c,
   double *s
);
```

Parameters

a
: Double-precision value `a`. Overwritten on return with result `r`.

b
: Double-precision value `b`. Overwritten on return with result `z` (zero).

c

Unused on entry. Overwritten on return with the value `cos( )`.

s

Unused on entry. Overwritten on return with the value `sin( )`.

Discussion

Given a vertical matrix containing `a` and `b`, computes the values of `cos` and `sin` that zero the lower value (`b`). Returns the value of `sin` in `s`, the value of `cos` in `c`, and the upper value (`r`) in `a`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_drotm

Applies a modified Givens transformation (single precision).

```
void cblas_drotm (
   const int N,
   double *X,
   const int incX,
   double *Y,
   const int incY,
   const double *P
);
```

Parameters

N

Number of elements in vectors.

X

Vector `X`. Modified on return.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `Y`. Modified on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

P

A 5-element vector:

`P[0]`

Flag value that defines the form of matrix `H`.

`-2.0`: matrix `H` contains the identity matrix.

`-1.0`: matrix `H` is identical to matrix `SH` (defined by the remaining values in the vector).

`0.0`: `H[1,2]` and `H[2,1]` are obtained from matrix `SH`. The remaining values are both `1.0`.

`1.0`: `H[1,1]` and `H[2,2]` are obtained from matrix `SH`. `H[1,2]` is 1.0. `H[2,1]` is -1.0.

`P[1]`

Contains `SH[1,1]`.

`P[2]`

Contains `SH[2,1]`.

`P[3]`

Contains `SH[1,2]`.

`P[4]`

Contains `SH[2,2]`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_drotmg

Generates a modified Givens rotation matrix.

```
void cblas_drotmg (
   double *d1,
   double *d2,
   double *b1,
   const double b2,
   double *P
);
```

Parameters

d1

Scaling factor `D1`.

d2

Scaling factor `D2`.

b1

Scaling factor `B1`.

b2

Scaling factor `B2`.

P

A 5-element vector:

`P[0]`

Flag value that defines the form of matrix `H`.

`-2.0`: matrix `H` contains the identity matrix.

`-1.0`: matrix `H` is identical to matrix `SH` (defined by the remaining values in the vector).

`0.0`: `H[1,2]` and `H[2,1]` are obtained from matrix `SH`. The remaining values are both `1.0`.

`1.0`: `H[1,1]` and `H[2,2]` are obtained from matrix `SH`. `H[1,2]` is 1.0. `H[2,1]` is -1.0.

`P[1]`

Contains `SH[1,1]`.

`P[2]`

Contains `SH[2,1]`.

`P[3]`

Contains `SH[1,2]`.

`P[4]`

Contains `SH[2,2]`.

Discussion

The resulting matrix zeroes the second component of the vector `(sqrt(D1)*B1, sqrt(SD2)*B2)`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dsbmv

Scales a symmetric band matrix, then multiplies by a vector, then adds a vector (double precision).

```
void cblas_dsbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const int K,
   const double alpha,
   const double *A,
   const int lda,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: The order of matrix `A`.

K
: Half-bandwidth of matrix `A`.

alpha
: Scaling value to multiply matrix A by.

A
: Matrix `A`.

lda
: The leading dimension of array containing matrix `A`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor that vector `Y` is multiplied by.

Y
: Vector `Y`. Replaced by results on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and returns the results in vector `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dscal

Multiplies each element of a vector by a constant (double-precision).

```
void cblas_dscal (
   const int N,
   const double alpha,
   double *X,
   const int incX
);
```

Parameters

N
: The number of elements in vector `x`.

alpha
: The constant scaling factor to multiply by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dsdot

Computes the double-precision dot product of a pair of single-precision vectors.

```
double cblas_dsdot (
   const int N,
   const float *X,
   const int incX,
   const float *Y,
   const int incY
);
```

Parameters

N
: The number of elements in the vectors.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dspmv

Scales a packed symmetric matrix, then multiplies by a vector, then scales and adds another vector (double precision).

```
void cblas_dspmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *Ap,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha
: Scaling factor that matrix `A` is multiplied by.

Ap
: Matrix `A` (in packed storage format).

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor that vector `y` is multiplied by.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dspr

Rank one update: adds a packed symmetric matrix to the product of a scaling factor, a vector, and its transpose (double precision).

```
void cblas_dspr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   double *Ap
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
Order of matrix `A`; number of elements in vector `x`.

alpha
Scaling factor to multiply `x` by.

X
Vector `x`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Ap
Matrix `A` (in packed storage format).

Discussion

Calculates `A + alpha*x*x`^{T} and stores the result in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dspr2

Rank two update of a packed symmetric matrix using two vectors (single precision).

```
void cblas_dspr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   const double *Y,
   const int incY,
   double *A
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A`; number of elements in vector `x`.

alpha
: Scaling factor to multiply `x` by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A` (in packed storage format).

Discussion

Calculates `A + alpha*x*y`^{T} `+ alpha*y*x`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dswap

Exchanges the elements of two vectors (double precision).

```
void cblas_dswap (
   const int N,
   double *X,
   const int incX,
   double *Y,
   const int incY
);
```

Parameters

N

Number of elements in vectors

X

Vector `x`. On return, contains elements copied from vector `y`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `y`. On return, contains elements copied from vector `x`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dsymm

Multiplies a matrix by a symmetric matrix (double-precision).

```
void cblas_dsymm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const double alpha,
   const double *A,
   const int lda,
   const double *B,
   const int ldb,
   const double beta,
   double *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Side

Determines the order in which the matrices should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + \; C$

or

$C \leftarrow \alpha BA + \; C$

where

$A = A^T$

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dsymv

Scales a symmetric matrix, multiplies by a vector, then scales and adds another vector (single precision).

```
void cblas_dsymv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *A,
   const int lda,
   const double *X,
   const int incX,
   const double beta,
   double *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

Order of matrix `A`; length of vectors.

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor for vector `y`.

Y

Vector `y`. Contains results on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dsyr

Rank one update: adds a symmetric matrix to the product of a scaling factor, a vector, and its transpose (double precision).

```
void cblas_dsyr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   double *A,
   const int lda
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

Order of matrix `A`; number of elements in vector `x`.

alpha

Scaling factor to multiply `x` by.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A

Matrix `A`.

lda

Leading dimension of array containing matrix A.

Discussion

Calculates `A + alpha*x*x`^{T} and stores the result in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dsyr2

Rank two update of a symmetric matrix using two vectors (single precision).

```
void cblas_dsyr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const double *X,
   const int incX,
   const double *Y,
   const int incY,
   double *A,
   const int lda
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

Order of matrix `A`; number of elements in vector `x`.

alpha

Scaling factor to multiply `x` by.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`.

Discussion

Calculates `A + alpha*x*y`^{T} `+ alpha*y*x`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dsyr2k

Performs a rank-2k update of a symmetric matrix (double precision).

```
void cblas_dsyr2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const double alpha,
   const double *A,
   const int lda,
   const double *B,
   const int ldb,
   const double beta,
   double *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N

Order of matrix `C`.

K

Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`.

B

Matrix `B`.

ldb

Leading dimension of array containing matrix `B`.

beta

Scaling factor for matrix `C`.

C

Matrix `C`.

ldc

Leading dimension of array containing matrix `C`.

Discussion

Computes `alpha*A*B`^{T} `+ alpha*B*A`^{T} `+beta*C`

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dsyrk

Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (double precision).

```
void cblas_dsyrk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const double alpha,
   const double *A,
   const int lda,
   const double beta,
   double *C,
   const int ldc
);
```

Parameters

Order
    Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
    Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
    Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

N
    Order of matrix `C`.

K
    Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha
    Scaling factor for matrix `A`.

A
    Matrix `A`.

lda
    Leading dimension of array containing matrix `A`.

beta
    Scaling factor for matrix `C`.

C
    Matrix `C`.

ldc
    Leading dimension of array containing matrix `C`.

Discussion
Calculates `alpha*A*A`^{T} `+ beta*C`; if transposed, calculates `alpha*A`^{T}`*A + beta*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtbmv

Scales a triangular band matrix, then multiplies by a vector (double precision).

```
void cblas_dtbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const double *A,
   const int lda,
   double *X,
   const int incX
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
The order of matrix `A`.

K
Half-bandwidth of matrix `A`.

A
Matrix `A`.

lda
The leading dimension of array containing matrix `A`.

X
Vector `x`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Computes `A*x` and stores the results in `x`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtbsv

Solves a triangular banded system of equations.

```
void cblas_dtbsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const double *A,
   const int lda,
   double *X,
   const int incX
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
Order of matrix `A`.

K
Number of superdiagonals or subdiagonals of matrix `A` (depending on the value of `Uplo`).

A
Matrix `A`.

lda
The leading dimension of matrix `A`.

X
Contains vector `B` on entry. Overwritten with vector `X` on return.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtpmv

Multiplies a triangular matrix by a vector, then adds a vector (double precision).

```
void cblas_dtpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const double *Ap,
   double *X,
   const int incX
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

Diag
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
Order of matrix `A` and the number of elements in vectors `x` and `y`.

Ap
Matrix `A`.

X
Vector `x`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Computes `A*x`, A^T`*x`, or `conjg(`A^T`)*x` and stores the results in `X`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtpsv

Solves a packed triangular system of equations.

```
void cblas_dtpsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const double *Ap,
   double *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

Order of matrix `A`.

Ap

Matrix `A` (in packed storage format).

X

Contains vector `B` on entry. Overwritten with vector `X` on return.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_dtrmm

Scales a triangular matrix and multiplies it by a matrix.

```
void cblas_dtrmm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const double alpha,
   const double *A,
   const int lda,
   double *B,
   const int ldb
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Side

Determines the order in which the matrices should be multiplied.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M

Number of rows in matrix `B`.

N

Number of columns in matrix `B`.

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

B

Matrix `B`. Overwritten by results on return.

ldb

Leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, multiplies `alpha*A*B` or `alpha*A'*B`, depending on `TransA`.

If `Side` is `'R'`, multiplies `alpha*B*A` or `alpha*B*A'`, depending on `TransA`.

In either case, the results are stored in matrix `B`.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtrmv

Multiplies a triangular matrix by a vector.

```
void cblas_dtrmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const double *A,
   const int lda,
   double *X,
   const int incX
);
```

Parameters

`Order`
Specifies row-major (C) or column-major (Fortran) data ordering.

`Uplo`
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

`TransA`
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

`Diag`
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

`N`
Order of matrix `A`.

`A`
Triangular matrix `A`.

`lda`
Leading dimension of matrix `A`.

`X`
Vector `X`.

`incX`
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Multiplies `A*X` or `A'*X`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtrsm

Solves a triangular system of equations with multiple values for the right side.

```
void cblas_dtrsm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const double alpha,
   const double *A,
   const int lda,
   double *B,
   const int ldb
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrix and vector should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M
: The number of rows in matrix `B`.

N
: The number of columns in matrix `B`.

alpha
: Scaling factor for matrix `A`.

A
: Triangular matrix `A`.

lda
: The leading dimension of matrix `B`.

B
: On entry, matrix `B`. Overwritten on return by matrix `X`.

ldb
: The leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, solves `A*X=alpha*B` or `A'*X=alpha*B`, depending on `TransA`.

If `Side` is `'R'`, solves `X*A=alpha*B` or `X*A'=alpha*B`, depending on `TransA`.

In either case, the results overwrite the values of matrix `B` in `X`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dtrsv

Solves a triangular system of equations with a single value for the right side.

```
void cblas_dtrsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const double *A,
   const int lda,
   double *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: The order of matrix `A`.

A
: Triangular matrix `A`.

lda
: The leading dimension of matrix `B`.

X
: The vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves `A*x=b` or `A'*x=b` where `x` and `b` are elements in `X` and `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dzasum

Computes the sum of the absolute values of real and imaginary parts of elements in a vector (single-precision complex).

```
double cblas_dzasum (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N
: Number of elements in the vector.

X
: Source vector

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value
The return value is a single floating point value that contains the sum of the absolute values of both the real and imaginary parts of the vector.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_dznrm2

Computes the unitary norm of a vector (double-precision complex).

```
double cblas_dznrm2 (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N
: Length of vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_errprn

Prints an error message.

```
int cblas_errprn (
   int ierr,
   int info,
   char *form,
   ...
);
```

Parameters

`ierr`
: Error code number.

`info`
: Exit status returned by the function call. Negative values generally indicate the index of the offending parameter. Positive numbers generally indicate that an algorithm did not converge. Zero indicates success.

`form`
: A printf-style format string.

`...`
: Arguments for the format string.

Return Value
Returns the minimum of `ierr` or `info`.

Discussion
This is commonly called by `cblas_xerbla` (page 126).

Availability
Available in iOS 4.0 and later.

See Also
`cblas_xerbla` (page 126)
`SetBLASParamErrorProc` (page 161)

Declared In
`cblas.h`

cblas_icamax

Returns the index of the element with the largest absolute value in a vector (single-precision complex).

```
int cblas_icamax (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N

Number of elements in the vector.

X

The vector.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value

Returns an index in the range 0..N-1 corresponding with the element with the largest absolute value.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_idamax

Returns the index of the element with the largest absolute value in a vector (double-precision).

```
int cblas_idamax (
   const int N,
   const double *X,
   const int incX
);
```

Parameters

N

Number of elements in the vector.

X

The vector.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value

Returns an index in the range 0..N-1 corresponding with the element with the largest absolute value.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_isamax

Returns the index of the element with the largest absolute value in a vector (single-precision).

```
int cblas_isamax (
   const int N,
   const float *X,
   const int incX
);
```

Parameters

N

Number of elements in the vector.

X

The vector.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value

Returns an index in the range 0..N-1 corresponding with the element with the largest absolute value.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_izamax

Returns the index of the element with the largest absolute value in a vector (double-precision complex).

```
int cblas_izamax (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N

Number of elements in the vector.

X

The vector.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value

Returns an index in the range 0..N-1 corresponding with the element with the largest absolute value.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_sasum

Computes the sum of the absolute values of elements in a vector (single-precision).

```
float cblas_sasum (
   const int N,
   const float *X,
   const int incX
);
```

Parameters

N

Number of elements in the vector.

X

Source vector

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_saxpy

Computes a constant times a vector plus a vector (single-precision).

```
void cblas_saxpy (
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   float *Y,
   const int incY
);
```

Parameters

N

Number of elements in the vectors.

alpha

Scaling factor for the values in `X`.

X

Input vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Input vector `Y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
On return, the contents of vector Y are replaced with the result. The value computed is `(alpha * X[i]) + Y[i]`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_scasum

Computes the sum of the absolute values of real and imaginary parts of elements in a vector (single-precision complex).

```
float cblas_scasum (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N
Number of elements in the vector.

X
Source vector

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Return Value
The return value is a single floating point value that contains the sum of the absolute values of both the real and imaginary parts of the vector.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_scnrm2

Computes the unitary norm of a vector (single-precision complex).

```
float cblas_scnrm2 (
   const int N,
   const void *X,
   const int incX
);
```

Parameters

N
Length of vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_scopy

Copies a vector to another vector (single-precision).

```
void cblas_scopy (
   const int N,
   const float *X,
   const int incX,
   float *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

X
: Source vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Destination vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sdot

Computes the dot product of two vectors (single-precision).

```
float cblas_sdot (
   const int N,
   const float *X,
   const int incX,
   const float *Y,
   const int incY
);
```

Parameters

N
: The number of elements in the vectors.

X
: Vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Y
: Vector Y.

incY
: Stride within Y. For example, if incY is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_sdsdot

Computes the dot product of two single-precision vectors plus an initial single-precision value.

```
float cblas_sdsdot (
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   const float *Y,
   const int incY
);
```

Parameters

N
: The number of elements in the vectors.

alpha
: The initial value to add to the dot product.

X
: Vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Y
: Vector Y.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sgbmv

Scales a general band matrix, then multiplies by a vector, then adds a vector (single precision).

```
void cblas_sgbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const int KL,
   const int KU,
   const float alpha,
   const float *A,
   const int lda,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

M
Number of rows in matrix A.

N
Number of columns in matrix A.

KL
Number of subdiagonals in matrix A.

KU
Number of superdiagonals in matrix A.

alpha
Scaling factor to multiply matrix `A` by.

A
Matrix `A`.

lda
Leading dimension of array containing matrix `A`. (Must be at least `KL+KU+1`.)

X

Vector X.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

beta

Scaling factor to multiply vector Y by.

Y

Vector Y.

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

Discussion

Computes alpha*A*x + beta*y or alpha*A'*x + beta*y depending on the value of TransA.

Availability

Available in iOS 4.0 and later.

Declared In

cblas.h

cblas_sgemm

Multiplies two matrices (single-precision).

```
void cblas_sgemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_TRANSPOSE TransB,
   const int M,
   const int N,
   const int K,
   const float alpha,
   const float *A,
   const int lda,
   const float *B,
   const int ldb,
   const float beta,
   float *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

TransA

Specifies whether to transpose matrix A.

TransB

Specifies whether to transpose matrix B.

M

Number of rows in matrices A and C.

N

Number of columns in matrices `B` and `C`.

K

Number of columns in matrix `A`; number of rows in matrix `B`.

alpha

Scaling factor for the product of matrices A and B.

A

Matrix A.

lda

The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B

Matrix B.

ldb

The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta

Scaling factor for matrix C.

C

Matrix C.

ldc

The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + C$

or

$C \leftarrow \alpha BA + C$

with optional use of transposed forms of `A`, `B`, or both.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_sgemv

Multiplies a matrix by a vector (single precision).

```
void cblas_sgemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

TransA

Specifies whether to transpose matrix `A`.

M

Number of rows in matrix `A`.

N

Number of columns in matrix `A`.

alpha

Scaling factor for the product of matrix `A` and vector `X`.

A

Matrix A.

lda

The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

X

Vector X.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor for vector `Y`.

Y

Vector Y

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

Discussion

This function multiplies `A * X` (after transposing `A`, if needed) and multiplies the resulting matrix by `alpha`. It then multiplies vector `Y` by `beta`. It stores the sum of these two products in vector `Y`.

Thus, it calculates either

`Y←αAX +  Y`

with optional use of the transposed form of `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sger

Multiplies vector X by the transform of vector Y, then adds matrix A (single precison).

```
void cblas_sger (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   const float *Y,
   const int incY,
   float *A,
   const int lda
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

M
Number of rows in matrix `A`.

N
Number of columns in matrix `A`.

alpha
Scaling factor for vector `X`.

X
Vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
Vector `Y`.

incY
Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
Matrix `A`.

lda
Leading dimension of array containing matrix `A`.

Discussion
Computes `alpha*x*y' + A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_snrm2

Computes the L2 norm (Euclidian length) of a vector (single precision).

```
float cblas_snrm2 (
   const int N,
   const float *X,
   const int incX
);
```

Parameters

N
Length of vector `X`.

X
Vector `X`.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_srot

Applies a Givens rotation matrix to a pair of vectors.

```
void cblas_srot (
   const int N,
   float *X,
   const int incX,
   float *Y,
   const int incY,
   const float c,
   const float s
);
```

Parameters

N
The number of elements in vectors `X` and `Y`.

X
Vector `X`. Modified on return.

incX
Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
Vector `Y`. Modified on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

c
: The value `cos( )` in the Givens rotation matrix.

s
: The value `sin( )` in the Givens rotation matrix.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_srotg

Constructs a Givens rotation matrix.

```
void cblas_srotg (
   float *a,
   float *b,
   float *c,
   float *s
);
```

Parameters

a
: Single-precision value `a`. Overwritten on return with result `r`.

b
: Single-precision value `b`. Overwritten on return with result `z` (zero).

c
: Unused on entry. Overwritten on return with the value `cos( )`.

s
: Unused on entry. Overwritten on return with the value `sin( )`.

Discussion
Given a vertical matrix containing `a` and `b`, computes the values of `cos` and `sin` that zero the lower value (`b`). Returns the value of `sin` in `s`, the value of `cos` in `c`, and the upper value (`r`) in `a`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_srotm

Applies a modified Givens transformation (single precision).

```
void cblas_srotm (
   const int N,
   float *X,
   const int incX,
   float *Y,
   const int incY,
   const float *P
);
```

Parameters

N

Number of elements in vectors.

X

Vector `X`. Modified on return.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `Y`. Modified on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

P

A 5-element vector:

`P[0]`

Flag value that defines the form of matrix `H`.

`-2.0`: matrix `H` contains the identity matrix.

`-1.0`: matrix `H` is identical to matrix `SH` (defined by the remaining values in the vector).

`0.0`: `H[1,2]` and `H[2,1]` are obtained from matrix `SH`. The remaining values are both `1.0`.

`1.0`: `H[1,1]` and `H[2,2]` are obtained from matrix `SH`. `H[1,2]` is 1.0. `H[2,1]` is -1.0.

`P[1]`

Contains `SH[1,1]`.

`P[2]`

Contains `SH[2,1]`.

`P[3]`

Contains `SH[1,2]`.

`P[4]`

Contains `SH[2,2]`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_srotmg

Generates a modified Givens rotation matrix.

```
void cblas_srotmg (
   float *d1,
   float *d2,
   float *b1,
   const float b2,
   float *P
);
```

Parameters

d1

Scaling factor `D1`.

d2

Scaling factor `D2`.

b1

Scaling factor `B1`.

b2

Scaling factor `B2`.

P

A 5-element vector:

`P[0]`

Flag value that defines the form of matrix `H`.

`-2.0`: matrix `H` contains the identity matrix.

`-1.0`: matrix `H` is identical to matrix `SH` (defined by the remaining values in the vector).

`0.0`: `H[1,2]` and `H[2,1]` are obtained from matrix `SH`. The remaining values are both `1.0`.

`1.0`: `H[1,1]` and `H[2,2]` are obtained from matrix `SH`. `H[1,2]` is 1.0. `H[2,1]` is -1.0.

`P[1]`

Contains `SH[1,1]`.

`P[2]`

Contains `SH[2,1]`.

`P[3]`

Contains `SH[1,2]`.

`P[4]`

Contains `SH[2,2]`.

Discussion

The resulting matrix zeroes the second component of the vector `(sqrt(D1)*B1, sqrt(SD2)*B2)`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ssbmv

Scales a symmetric band matrix, then multiplies by a vector, then adds a vector (single-precision).

```
void cblas_ssbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const int K,
   const float alpha,
   const float *A,
   const int lda,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

The order of matrix `A`.

K

Half-bandwidth of matrix `A`.

alpha

Scaling value to multiply matrix A by.

A

Matrix `A`.

lda

The leading dimension of array containing matrix `A`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor that vector `Y` is multiplied by.

Y

Vector `Y`. Replaced by results on return.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Computes `alpha*A*x + beta*y` and returns the results in vector `Y`.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sscal

Multiplies each element of a vector by a constant (single-precision).

```
void cblas_sscal (
   const int N,
   const float alpha,
   float *X,
   const int incX
);
```

Parameters

`N`
Number of elements to scale.

`alpha`
The constant to multiply by.

`X`
Vector `x`.

`incX`
Stride within `X`. For example, if `incX` is 7, every 7th element is multiplied by `alpha`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sspmv

Scales a packed symmetric matrix, then multiplies by a vector, then scales and adds another vector (single precision).

```
void cblas_sspmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *Ap,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

`Order`
Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha
: Scaling factor that matrix `A` is multiplied by.

Ap
: Matrix `A` (in packed storage format).

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor that vector `y` is multiplied by.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_sspr

Rank one update: adds a packed symmetric matrix to the product of a scaling factor, a vector, and its transpose (single precision).

```
void cblas_sspr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   float *Ap
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A`; number of elements in vector `x`.

alpha
: Scaling factor to multiply `x` by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Ap
: Matrix `A` (in packed storage format).

Discussion

Calculates `A + alpha*x*x`^{T} and stores the result in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_sspr2

Rank two update of a packed symmetric matrix using two vectors (single precision).

```
void cblas_sspr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   const float *Y,
   const int incY,
   float *A
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A`; number of elements in vector `x`.

alpha
: Scaling factor to multiply `x` by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A

Matrix `A` (in packed storage format).

Discussion

Calculates `A + alpha*x*y`^{T}` + alpha*y*x`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_sswap

Exchanges the elements of two vectors (single precision).

```
void cblas_sswap (
   const int N,
   float *X,
   const int incX,
   float *Y,
   const int incY
);
```

Parameters

N

Number of elements in vectors

X

Vector `x`. On return, contains elements copied from vector `y`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `y`. On return, contains elements copied from vector `x`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ssymm

Multiplies a matrix by a symmetric matrix (single-precision).

```
void cblas_ssymm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   const float *B,
   const int ldb,
   const float beta,
   float *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrices should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

`C←αAB +  C`

or

`C←αBA +  C`

where

`A = A`^{T}

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ssymv

Scales a symmetric matrix, multiplies by a vector, then scales and adds another vector (single precision).

```
void cblas_ssymv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   const float *X,
   const int incX,
   const float beta,
   float *Y,
   const int incY
);
```

Parameters

Order
    Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
    Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
    Order of matrix `A`; length of vectors.

alpha
    Scaling factor for matrix `A`.

A
    Matrix `A`.

lda
    Leading dimension of array containing matrix `A`.

X
    Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor for vector `y`.

Y
: Vector `y`. Contains results on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ssyr

Rank one update: adds a symmetric matrix to the product of a scaling factor, a vector, and its transpose (single precision).

```
void cblas_ssyr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   float *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A`; number of elements in vector `x`.

alpha
: Scaling factor to multiply `x` by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion

Calculates `A + alpha*x*x`^{T} and stores the result in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ssyr2

Rank two update of a symmetric matrix using two vectors (single precision).

```
void cblas_ssyr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const float alpha,
   const float *X,
   const int incX,
   const float *Y,
   const int incY,
   float *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A`; number of elements in vector `x`.

alpha
: Scaling factor to multiply `x` by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion

Calculates `A + alpha*x*y`^{T} ` + alpha*y*x`^{T}.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ssyr2k

Performs a rank-2k update of a symmetric matrix (single precision).

```
void cblas_ssyr2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const float alpha,
   const float *A,
   const int lda,
   const float *B,
   const int ldb,
   const float beta,
   float *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N

Order of matrix `C`.

K

Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

B

Matrix `B`.

ldb
: Leading dimension of array containing matrix `B`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Computes `alpha*A*B`^{T} `+ alpha*B*A`^{T} `+beta*C`

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ssyrk

Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (single precision).

```
void cblas_ssyrk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const float alpha,
   const float *A,
   const int lda,
   const float beta,
   float *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

N
: Order of matrix `C`.

K
: Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Calculates `alpha*A*A`^{T} `+ beta*C`; if transposed, calculates `alpha*A`^{T}`*A + beta*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_stbmv

Scales a triangular band matrix, then multiplies by a vector (single precision).

```
void cblas_stbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const float *A,
   const int lda,
   float *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: The order of matrix `A`.

K
: Half-bandwidth of matrix A.

A
: Triangular matrix A.

lda
: The leading dimension of array containing matrix A.

X
: Vector x.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Discussion
Computes A*x and stores the results in x.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_stbsv

Solves a triangular banded system of equations.

```
void cblas_stbsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const float *A,
   const int lda,
   float *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are 'U' or 'L'.

TransA
: Specifies whether to use matrix A ('N' or 'n') or the transpose of A ('T', 't', 'C', or 'c').

Diag
: Specifies whether the matrix is unit triangular. Possible values are 'U' (unit triangular) or 'N' (not unit triangular).

N
: Order of matrix A.

K

Number of superdiagonals or subdiagonals of matrix `A` (depending on the value of `Uplo`).

A

Triangular matrix `A`.

lda

The leading dimension of matrix `A`.

X

Contains vector `B` on entry. Overwritten with vector `X` on return.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_stpmv

Multiplies a triangular matrix by a vector, then adds a vector (single precision).

```
void cblas_stpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const float *Ap,
   float *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

Order of matrix `A` and the number of elements in vectors `x` and `y`.

Ap

Triangular matrix `A`.

X
: Vector x.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Discussion
Computes A*x, A^T*x, or conjg(A^T)*x and stores the results in X.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_stpsv

Solves a packed triangular system of equations.

```
void cblas_stpsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const float *Ap,
   float *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are 'U' or 'L'.

TransA
: Specifies whether to use matrix A ('N' or 'n') or the transpose of A ('T', 't', 'C', or 'c').

Diag
: Specifies whether the matrix is unit triangular. Possible values are 'U' (unit triangular) or 'N' (not unit triangular).

N
: Order of matrix A.

Ap
: Triangular matrix A (in packed storage format).

X
: Contains vector B on entry. Overwritten with vector X on return.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Discussion
Solves the system of equations A*X=B or A'*X=B, depending on the value of TransA.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_strmm

Scales a triangular matrix and multiplies it by a matrix.

```
void cblas_strmm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   float *B,
   const int ldb
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Side
Determines the order in which the matrices should be multiplied.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M
Number of rows in matrix `B`.

N
Number of columns in matrix `B`.

alpha
Scaling factor for matrix `A`.

A
Triangular matrix `A`.

lda
Leading dimension of matrix `A`.

B
Matrix `B`. Overwritten by results on return.

ldb

Leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, multiplies `alpha*A*B` or `alpha*A'*B`, depending on `TransA`.

If `Side` is `'R'`, multiplies `alpha*B*A` or `alpha*B*A'`, depending on `TransA`.

In either case, the results are stored in matrix `B`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_strmv

Multiplies a triangular matrix by a vector.

```
void cblas_strmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const float *A,
   const int lda,
   float *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

Order of matrix `A`.

A

Triangular matrix `A`.

lda

Leading dimension of matrix `A`.

X

Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Multiplies `A*X` or `A'*X`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_strsm

Solves a triangular system of equations with multiple values for the right side.

```
void cblas_strsm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const float alpha,
   const float *A,
   const int lda,
   float *B,
   const int ldb
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrix and vector should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M
: The number of rows in matrix `B`.

N
: The number of columns in matrix `B`.

alpha
: Scaling factor for matrix `A`.

A
: Triangular matrix `A`.

lda

The leading dimension of matrix `B`.

B

On entry, matrix `B`. Overwritten on return by matrix `X`.

ldb

The leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, solves `A*X=alpha*B` or `A'*X=alpha*B`, depending on `TransA`.

If `Side` is `'R'`, solves `X*A=alpha*B` or `X*A'=alpha*B`, depending on `TransA`.

In either case, the results overwrite the values of matrix `B` in `X`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_strsv

Solves a triangular system of equations with a single value for the right side.

```
void cblas_strsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const float *A,
   const int lda,
   float *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

The order of matrix `A`.

A

Triangular matrix `A`.

lda
: The leading dimension of matrix B.

X
: The vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Discussion
Solves A*x=b or A'*x=b where x and b are elements in X and B.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_xerbla

The default error handler for BLAS routines.

```
void cblas_xerbla (
   int p,
   char *rout,
   char *form,
   ...
);
```

Parameters

p
: The position of the invalid parameter

rout
: The name of the routine that generated this error.

form
: A format string describing the error.

...
: Additional parameters for the format string.

Discussion
This is the default error handler for BLAS functions. You can replace it with another function by calling SetBLASParamErrorProc (page 161).

Availability
Available in iOS 4.0 and later.

See Also
cblas_errprn (page 91)
SetBLASParamErrorProc (page 161)

Declared In
cblas.h

cblas_zaxpy

Computes a constant times a vector plus a vector (double-precision complex).

```
void cblas_zaxpy (
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

alpha
: Scaling factor for the values in `X`.

X
: Input vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Input vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

On return, the contents of vector Y are replaced with the result. The value computed is `(alpha * X[i]) + Y[i]`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zcopy

Copies a vector to another vector (double-precision complex).

```
void cblas_zcopy (
   const int N,
   const void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N
: Number of elements in the vectors.

X
: Source vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Destination vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zdotc_sub

Calculates the dot product of the complex conjugate of a double-precision complex vector with a second double-precision complex vector.

```
void cblas_zdotc_sub (
   const int N,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *dotc
);
```

Parameters

N
: Number of elements in vectors `X` and `Y`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

dotc
: The result vector.

Discussion
Computes conjg(X) * Y.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zdotu_sub

Computes the dot product of two double-precision complex vectors.

```
void cblas_zdotu_sub (
   const int N,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *dotu
);
```

Parameters

N

The length of vectors X and Y.

X

Vector X.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

Y

Vector Y.

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

dotu

The result vector.

Availability
Available in iOS 4.0 and later.

Declared In
cblas.h

cblas_zdrot

Applies a Givens rotation matrix to a pair of complex vectors.

```
void cblas_zdrot (
   const int N,
   void *X,
   const int incX,
   void *Y,
   const int incY,
   const double c,
   const double s
);
```

Parameters

N

The number of elements in vectors X and Y.

X

Vector X. Modified on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`. Modified on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

c
: The value `cos( )` in the Givens rotation matrix.

s
: The value `sin( )` in the Givens rotation matrix.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zdscal

Multiplies each element of a vector by a constant (double-precision complex).

```
void cblas_zdscal (
   const int N,
   const double alpha,
   void *X,
   const int incX
);
```

Parameters

N
: The number of elements in the vector.

alpha
: The constant scaling factor.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zgbmv

Scales a general band matrix, then multiplies by a vector, then adds a vector (double-precision complex).

```
void cblas_zgbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const int KL,
   const int KU,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

M

Number of rows in matrix A.

N

Number of columns in matrix A.

KL

Number of subdiagonals in matrix A.

KU

Number of superdiagonals in matrix A.

alpha

Scaling factor to multiply matrix `A` by.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`. (Must be at least `KL+KU+1`.)

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta

Scaling factor to multiply vector Y by.

Y

Vector Y.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Computes `alpha*A*x + beta*y`, `alpha*A'*x + beta*y`, or `alpha*conjg(A')*x + beta*y` depending on the value of `TransA`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zgemm

Multiplies two matrices (double-precision complex).

```
void cblas_zgemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_TRANSPOSE TransB,
   const int M,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

TransA
: Specifies whether to transpose matrix `A`.

TransB
: Specifies whether to transpose matrix `B`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

K
: Number of columns in matrix `A`; number of rows in matrix `B`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda

The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B

Matrix B.

ldb

The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta

Scaling factor for matrix C.

C

Matrix C.

ldc

The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

`C←αAB +  C`

or

`C←αBA +  C`

with optional use of transposed forms of `A`, `B`, or both.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zgemv

Multiplies a matrix by a vector (double-precision complex).

```
void cblas_zgemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_TRANSPOSE TransA,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

TransA

Specifies whether to transpose matrix A.

M

Number of rows in matrix A.

N

Number of columns in matrix A.

alpha

Scaling factor for the product of matrix A and vector X.

A

Matrix A.

lda

The size of the first dimention of matrix A; if you are passing a matrix A[m][n], the value should be m.

X

Vector X.

incX

Stride within X. For example, if incX is 7, every 7th element is used.

beta

Scaling factor for vector Y.

Y

Vector Y

incY

Stride within Y. For example, if incY is 7, every 7th element is used.

Discussion

This function multiplies A * X (after transposing A, if needed) and multiplies the resulting matrix by alpha. It then multiplies vector Y by beta. It stores the sum of these two products in vector Y.

Thus, it calculates either

$Y \leftarrow \alpha AX + Y$

with optional use of the transposed form of A.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zgerc

Multiplies vector X by the conjugate transform of vector Y, then adds matrix A (double-precision complex).

```
void cblas_zgerc (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

M
: Number of rows in matrix `A`.

N
: Number of columns in matrix `A`.

alpha
: Scaling factor for vector `X`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `Y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

Discussion
Computes `alpha*x*conjg(y') + A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zgeru

Multiplies vector X by the transform of vector Y, then adds matrix A (double-precision complex).

```
void cblas_zgeru (
   const enum CBLAS_ORDER Order,
   const int M,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order
    Specifies row-major (C) or column-major (Fortran) data ordering.

M
    Number of rows in matrix `A`.

N
    Number of columns in matrix `A`.

alpha
    Scaling factor for vector `X`.

X
    Vector `X`.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
    Vector `Y`.

incY
    Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A
    Matrix `A`.

lda
    Leading dimension of array containing matrix `A`.

Discussion
Computes `alpha*x*y' + A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zhbmv

Scales a Hermitian band matrix, then multiplies by a vector, then adds a vector (double-precision complex).

```
void cblas_zhbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: The order of matrix `A`.

K
: Half-bandwidth of matrix `A`.

alpha
: Scaling value to multiply matrix A by.

A
: Matrix `A`.

lda
: The leading dimension of array containing matrix `A`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor that vector `Y` is multiplied by.

Y
: Vector `Y`. Replaced by results on return.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion

Computes `alpha*A*x + beta*y` and returns the results in vector `Y`.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zhemm

Multiplies two Hermitian matrices (double-precision complex).

```
void cblas_zhemm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
Specifies row-major (C) or column-major (Fortran) data ordering.

Side
Determines the order in which the matrices should be multiplied.

Uplo
Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
The number of rows in matrices `A` and `C`.

N
The number of columns in matrices `B` and `C`.

alpha
Scaling factor for the product of matrices A and B.

A
Matrix A.

lda
The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
Matrix B.

ldb
The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
Scaling factor for matrix C.

C

Matrix C.

ldc

The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + \ C$

or

$C \leftarrow \alpha BA + \ C$

where

$A = A^H$

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zhemv

Scales and multiplies a Hermitian matrix by a vector, then adds a second (scaled) vector.

```
void cblas_zhemv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

The order of matrix `A`.

alpha
    Scaling factor for matrix `A`.

A
    Matrix `A`.

lda
    Leading dimension of matrix `A`.

X
    Vector `X`.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
    Scaling factor for vector `X`.

Y
    Vector `Y`. Overwritten by results on return.

incY
    Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Calculates `Y←αAX +  Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zher

Adds the product of a scaling factor, vector `X`, and the conjugate transpose of `X` to matrix `A`.

```
void cblas_zher (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const void *X,
   const int incX,
   void *A,
   const int lda
);
```

Parameters

Order
    Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
    Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
    The order of matrix `A`.

alpha
    The scaling factor for vector `X`.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

Discussion

Computes `A←αX*conjg(X') +  A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zher2

Hermitian rank 2 update: adds the product of a scaling factor, vector `X`, and the conjugate transpose of vector `Y` to the product of the conjugate of the scaling factor, vector `Y`, and the conjugate transpose of vector `X`, and adds the result to matrix `A`.

```
void cblas_zher2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *A,
   const int lda
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N

The order of matrix `A`.

alpha

The scaling factor α.

X

Vector `X`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y

Vector `Y`.

incY

Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

A

Matrix `A`.

lda

The leading dimension of matrix `A`.

Discussion

Computes `A←αX*conjg(Y') + conjg(α)*Y*conjg(X') + A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zher2k

Performs a rank-2k update of a complex Hermitian matrix (double-precision complex).

```
void cblas_zher2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const double beta,
   void *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N

Order of matrix `C`.

K

Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

B
: Matrix `B`.

ldb
: Leading dimension of array containing matrix `B`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Computes `alpha*A*B`^{H}` + alpha*B*A`^{H}` +beta*C`

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zherk

Rank-k update—multiplies a Hermitian matrix by its transpose and adds a second matrix (single precision).

```
void cblas_zherk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const double alpha,
   const void *A,
   const int lda,
   const double beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans
: Specifies whether to use matrix A (`'N'` or `'n'`) or the conjugate transpose of A (`'C'` or `'c'`).

N
: Order of matrix `C`.

K
: Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Calculates `alpha*A*A`^{H}` + beta*C`; if transposed, calculates `alpha*A`^{H}`*A + beta*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zhpmv

Scales a packed hermitian matrix, multiplies it by a vector, and adds a scaled vector.

```
void cblas_zhpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *Ap,
   const void *X,
   const int incX,
   const void *beta,
   void *Y,
   const int incY
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha
: Scaling factor that matrix `A` is multiplied by.

Ap
: Matrix `A`.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

beta
: Scaling factor that vector `y` is multiplied by.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Discussion
Computes `alpha*A*x + beta*y` and stores the results in `Y`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zhpr

Scales and multiplies a vector times its conjugate transpose, then adds a matrix.

```
void cblas_zhpr (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const double alpha,
   const void *X,
   const int incX,
   void *A
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vector `x`.

alpha
: Scaling factor that vector `x` is multiplied by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

A
: Matrix `A`. Overwritten by results on return.

Discussion
Calculates `alpha*x*conjg(x') + A` and stores the result in `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zhpr2

Multiplies a vector times the conjugate transpose of a second vector and vice-versa, sums the results, and adds a matrix.

```
void cblas_zhpr2 (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const int N,
   const void *alpha,
   const void *X,
   const int incX,
   const void *Y,
   const int incY,
   void *Ap
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

alpha
: Scaling factor that vector `x` is multiplied by.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
: Vector `y`.

incY
: Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Ap

Matrix `A` in packed storage format. Overwritten by the results on return.

Discussion

Calcuates `alpha*x*conjg(y') + conjg(alpha)*y*conjg(x') + A`, and stores the result in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zrotg

Constructs a complex Givens rotation.

```
void cblas_zrotg (
   void *a,
   void *b,
   void *c,
   void *s
);
```

Parameters

a

Complex value `a`. Overwritten on return with result `r`.

b

Complex value `a`. Overwritten on return with result `z` (zero).

c

Real value `c`. Unused on entry. Overwritten on return with the value `cos( )`.

s

Complex value `s`. Unused on entry. Overwritten on return with the value `sin( )`.

Discussion

Given a vertical matrix containing `a` and `b`, computes the values of `cos` and `sin` that zero the lower value (`b`). Returns the value of `sin` in `s`, the value of `cos` in `c`, and the upper value (`r`) in `a`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zscal

Multiplies each element of a vector by a constant (double-precision complex).

```
void cblas_zscal (
   const int N,
   const void *alpha,
   void *X,
   const int incX
);
```

Parameters

N
    The number of elements in the vector.

alpha
    The constant scaling factor.

X
    Vector `x`.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zswap

Exchanges the elements of two vectors (double-precision complex).

```
void cblas_zswap (
   const int N,
   void *X,
   const int incX,
   void *Y,
   const int incY
);
```

Parameters

N
    Number of elements in vectors

X
    Vector `x`. On return, contains elements copied from vector `y`.

incX
    Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Y
    Vector `y`. On return, contains elements copied from vector `x`.

incY
    Stride within `Y`. For example, if `incY` is 7, every 7th element is used.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_zsymm

Multiplies a matrix by a symmetric matrix (double-precision complex).

```
void cblas_zsymm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrices should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

M
: Number of rows in matrices `A` and `C`.

N
: Number of columns in matrices `B` and `C`.

alpha
: Scaling factor for the product of matrices A and B.

A
: Matrix A.

lda
: The size of the first dimention of matrix `A`; if you are passing a matrix `A[m][n]`, the value should be `m`.

B
: Matrix B.

ldb
: The size of the first dimention of matrix `B`; if you are passing a matrix `B[m][n]`, the value should be `m`.

beta
: Scaling factor for matrix C.

C
: Matrix C.

ldc
: The size of the first dimention of matrix `C`; if you are passing a matrix `C[m][n]`, the value should be `m`.

Discussion

This function multiplies `A * B` or `B * A` (depending on the value of `Side`) and multiplies the resulting matrix by `alpha`. It then multiplies matrix `C` by `beta`. It stores the sum of these two products in matrix `C`.

Thus, it calculates either

$C \leftarrow \alpha AB + \ C$

or

$C \leftarrow \alpha BA + \ C$

where

$A = A^T$

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zsyr2k

Performs a rank-2k update of a symmetric matrix (double-precision complex).

```
void cblas_zsyr2k (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *B,
   const int ldb,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans

Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

N

Order of matrix `C`.

K
: Specifies the number of columns in matrices `A` and `B` if `trans='N'`.

  Specifies the number of rows if `trans='C'` or `trans='T'`).

alpha
: Scaling factor for matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of array containing matrix `A`.

B
: Matrix `B`.

ldb
: Leading dimension of array containing matrix `B`.

beta
: Scaling factor for matrix `C`.

C
: Matrix `C`.

ldc
: Leading dimension of array containing matrix `C`.

Discussion

Computes `alpha*A*B`^{T} `+ alpha*B*A`^{T} `+beta*C`

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_zsyrk

Rank-k update—multiplies a symmetric matrix by its transpose and adds a second matrix (double-precision complex).

```
void cblas_zsyrk (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE Trans,
   const int N,
   const int K,
   const void *alpha,
   const void *A,
   const int lda,
   const void *beta,
   void *C,
   const int ldc
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

Trans

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'` or `'t'`).

N

Order of matrix `C`.

K

Number of columns in matrix `A` (or number of rows if matrix `A` is transposed).

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of array containing matrix `A`.

beta

Scaling factor for matrix `C`.

C

Matrix `C`.

ldc

Leading dimension of array containing matrix `C`.

Discussion

Calculates `alpha*A*A`^{T} `+ beta*C`; if transposed, calculates `alpha*A`^{T}`*A + beta*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ztbmv

Scales a triangular band matrix, then multiplies by a vector (double-precision complex).

```
void cblas_ztbmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N

The order of matrix `A`.

K

Half-bandwidth of matrix `A`.

A

Matrix `A`.

lda

The leading dimension of array containing matrix `A`.

X

Vector `x`.

incX

Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Computes `A*x` and stores the results in `x`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ztbsv

Solves a triangular banded system of equations.

```
void cblas_ztbsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const int K,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A`.

K
: Number of superdiagonals or subdiagonals of matrix `A` (depending on the value of `Uplo`).

A
: Matrix `A`.

lda
: The leading dimension of matrix `A`.

X
: Contains vector `B` on entry. Overwritten with vector `X` on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ztpmv

Multiplies a triangular matrix by a vector, then adds a vector (double-precision compex).

```
void cblas_ztpmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *Ap,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`), the transpose of A (`'T'` or `'t'`), or the conjugate of A (`'C'` or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A` and the number of elements in vectors `x` and `y`.

Ap
: Matrix `A`.

X
: Vector `x`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion

Computes `A*x`, A^T`*x`, or `conjg(`A^T`)*x` and stores the results in `X`.

Availability

Available in iOS 4.0 and later.

Declared In

`cblas.h`

cblas_ztpsv

Solves a packed triangular system of equations.

```
void cblas_ztpsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *Ap,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A`.

Ap
: Matrix `A` (in packed storage format).

X
: Contains vector `B` on entry. Overwritten with vector `X` on return.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Solves the system of equations `A*X=B` or `A'*X=B`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ztrmm

Scales a triangular matrix and multiplies it by a matrix.

```
void cblas_ztrmm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   void *B,
   const int ldb
);
```

Parameters

Order

Specifies row-major (C) or column-major (Fortran) data ordering.

Side

Determines the order in which the matrices should be multiplied.

Uplo

Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA

Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag

Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M

Number of rows in matrix `B`.

N

Number of columns in matrix `B`.

alpha

Scaling factor for matrix `A`.

A

Matrix `A`.

lda

Leading dimension of matrix `A`.

B

Matrix `B`. Overwritten by results on return.

ldb

Leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, multiplies `alpha*A*B` or `alpha*A'*B`, depending on `TransA`.

If `Side` is `'R'`, multiplies `alpha*B*A` or `alpha*B*A'`, depending on `TransA`.

In either case, the results are stored in matrix `B`.

Availability

Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ztrmv

Multiplies a triangular matrix by a vector.

```
void cblas_ztrmv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

N
: Order of matrix `A`.

A
: Matrix `A`.

lda
: Leading dimension of matrix `A`.

X
: Vector `X`.

incX
: Stride within `X`. For example, if `incX` is 7, every 7th element is used.

Discussion
Multiplies `A*X` or `A'*X`, depending on the value of `TransA`.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ztrsm

Solves a triangular system of equations with multiple values for the right side.

```
void cblas_ztrsm (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_SIDE Side,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int M,
   const int N,
   const void *alpha,
   const void *A,
   const int lda,
   void *B,
   const int ldb
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Side
: Determines the order in which the matrix and vector should be multiplied.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are `'U'` or `'L'`.

TransA
: Specifies whether to use matrix A (`'N'` or `'n'`) or the transpose of A (`'T'`, `'t'`, `'C'`, or `'c'`).

Diag
: Specifies whether the matrix is unit triangular. Possible values are `'U'` (unit triangular) or `'N'` (not unit triangular).

M
: The number of rows in matrix `B`.

N
: The number of columns in matrix `B`.

alpha
: Scaling factor for matrix `A`.

A
: Triangular matrix `A`.

lda
: The leading dimension of matrix `B`.

B
: On entry, matrix `B`. Overwritten on return by matrix `X`.

ldb
: The leading dimension of matrix `B`.

Discussion

If `Side` is `'L'`, solves `A*X=alpha*B` or `A'*X=alpha*B`, depending on `TransA`.

If `Side` is `'R'`, solves `X*A=alpha*B` or `X*A'=alpha*B`, depending on `TransA`.

In either case, the results overwrite the values of matrix B in X.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

cblas_ztrsv

Solves a triangular system of equations with a single value for the right side.

```
void cblas_ztrsv (
   const enum CBLAS_ORDER Order,
   const enum CBLAS_UPLO Uplo,
   const enum CBLAS_TRANSPOSE TransA,
   const enum CBLAS_DIAG Diag,
   const int N,
   const void *A,
   const int lda,
   void *X,
   const int incX
);
```

Parameters

Order
: Specifies row-major (C) or column-major (Fortran) data ordering.

Uplo
: Specifies whether to use the upper or lower triangle from the matrix. Valid values are 'U' or 'L'.

TransA
: Specifies whether to use matrix A ('N' or 'n') or the transpose of A ('T', 't', 'C', or 'c').

Diag
: Specifies whether the matrix is unit triangular. Possible values are 'U' (unit triangular) or 'N' (not unit triangular).

N
: The order of matrix A.

A
: Triangular matrix A.

lda
: The leading dimension of matrix B.

X
: The vector X.

incX
: Stride within X. For example, if incX is 7, every 7th element is used.

Discussion
Solves A*x=b or A'*x=b where x and b are elements in X and B.

Availability
Available in iOS 4.0 and later.

Declared In
`cblas.h`

SetBLASParamErrorProc

Sets an error handler function.

```
void SetBLASParamErrorProc (
   BLASParamErrorProc ErrorProc
);
```

Parameters

`ErrorProc`
The handler function BLAS should call when an error occurs (because of an invalid input, for example).

Availability
Available in iOS 4.0 and later.

See Also
`cblas_errprn` (page 91)
`cblas_xerbla` (page 126)

Declared In
`cblas.h`

Constants

Note: The types given here are valid for C or C++ and for either PowerPC or Intel processors. The typedefs shown are for C++ and PowerPC processors; for other, conditionally compiled typedefs, see the header files.

CBLAS_ORDER

Indicates whether a matrix is in row-major or column-major order.

```
enum CBLAS_ORDER {
    CblasRowMajor=101,
    CblasColMajor=102
};
typedef enum CBLAS_ORDER CBLAS_ORDER;
```

Constants

`CblasRowMajor`
Row-major order.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

`CblasColMajor`
: Column-major order.

  Available in iOS 4.0 and later.

  Declared in `cblas.h`.

CBLAS_TRANSPOSE

Indiates transpose operation to perform on a matrix.

```
enum CBLAS_TRANSPOSE {
    CblasNoTrans=111,
    CblasTrans=112,
    CblasConjTrans=113,
    AtlasConj=114
};
typedef enum CBLAS_TRANSPOSE CBLAS_TRANSPOSE;
```

Constants

`CblasNoTrans`
: No transposition.

  Available in iOS 4.0 and later.

  Declared in `cblas.h`.

`CblasTrans`
: For matrix `X`, use `X(T)`.

  Available in iOS 4.0 and later.

  Declared in `cblas.h`.

`CblasConjTrans`
: For matrix `X`, use `X(H)` (conjugate or Hermitian transposition).

  Available in iOS 4.0 and later.

  Declared in `cblas.h`.

`AtlasConj`
: Available in iOS 4.0 and later.

  Declared in `cblas.h`.

CBLAS_UPLO

Indicates which part of a symmetric matrix to use.

```
enum CBLAS_UPLO {
    CblasUpper=121,
    CblasLower=122
};
typedef enum CBLAS_UPLO CBLAS_UPLO;
```

Constants

`CblasUpper`

Use the upper triangle of the matrix.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

`CblasLower`

Use the lower triangle of the matrix.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

CBLAS_DIAG

Indicates whether a triangular matrix is unit-diagonal (diagonal elements are all equal to 1).

```
enum CBLAS_DIAG {
    CblasNonUnit=131,
    CblasUnit=132
};
typedef enum CBLAS_DIAG CBLAS_DIAG;
```

Constants

`CblasNonUnit`

The triangular matrix is not unit-diagonal.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

`CblasUnit`

The triangular matrix is unit-diagonal.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

CBLAS_SIDE

Indicates the order of a matrix multiplication.

```
enum CBLAS_SIDE {
    CblasLeft=141,
    CblasRight=142
};
typedef enum CBLAS_SIDE CBLAS_SIDE;
```

Constants

`CblasLeft`

Multiply A*B.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

`CblasRight`

Multiply B*A.

Available in iOS 4.0 and later.

Declared in `cblas.h`.

vDSP Reference

Framework: Accelerate/vecLib
Declared in vDSP.h

Overview

This document describes the vDSP portion of the Accelerate framework.

Functions by Task

Single-Vector Absolute Value

The functions in this group compute the absolute value of each element in a vector.

`vDSP_vabs` (page 334)
    Vector absolute values; single precision.

`vDSP_vabsD` (page 334)
    Vector absolute values; double precision.

`vDSP_vabsi` (page 335)
    Integer vector absolute values.

`vDSP_zvabs` (page 524)
    Complex vector absolute values; single precision.

`vDSP_zvabsD` (page 525)
    Complex vector absolute values; double precision.

`vDSP_vnabs` (page 426)
    Vector negative absolute values; single precision.

`vDSP_vnabsD` (page 427)
    Vector negative absolute values; double precision.

Single-Vector Negation

The functions in this group negates each element in a vector.

`vDSP_vneg` (page 428)
    Vector negative values; single precision.

`vDSP_vnegD` (page 429)
: Vector negative values; double precision.

`vDSP_zvneg` (page 539)
: Complex vector negate; single precision.

`vDSP_zvnegD` (page 540)
: Complex vector negate; double precision.

Single-Vector Fill or Clear

The functions in this group fills each element in a vector with a specific value or clears each element.

`vDSP_vfill` (page 362)
: Vector fill; single precision.

`vDSP_vfillD` (page 363)
: Vector fill; double precision.

`vDSP_vfilli` (page 364)
: Integer vector fill.

`vDSP_zvfill` (page 533)
: Complex vector fill; single precision.

`vDSP_zvfillD` (page 533)
: Complex vector fill; double precision.

`vDSP_vclr` (page 349)
: Vector clear; single precision.

`vDSP_vclrD` (page 349)
: Vector clear; double precision.

Single-Vector Generation

The functions in this group build vectors with specific generator functions.

`vDSP_vramp` (page 436)
: Build ramped vector; single precision.

`vDSP_vrampD` (page 437)
: Build ramped vector; double precision.

`vDSP_vrampmul` (page 437)
: Builds a ramped vector and multiplies by a source vector.

`vDSP_vrampmul_s1_15` (page 447)
: Vector fixed-point 1.15 format version of `vDSP_vrampmul` (page 437).

`vDSP_vrampmul_s8_24` (page 448)
: Vector fixed-point 8.24 format version of `vDSP_vrampmul` (page 437).

`vDSP_vrampmul2` (page 438)
: Stereo version of `vDSP_vrampmul` (page 437).

`vDSP_vrampmul2_s1_15` (page 439)
: Vector fixed-point 1.15 format version of `vDSP_vrampmul2` (page 438).

`vDSP_vrampmul2_s8_24` (page 441)
: Vector fixed-point 8.24 format version of `vDSP_vrampmul2` (page 438).

`vDSP_vrampmuladd` (page 442)
: Builds a ramped vector, multiplies it by a source vector, and adds the result to the output vector.

`vDSP_vrampmuladd_s1_15` (page 446)
: Vector fixed-point 1.15 format version of `vDSP_vrampmuladd` (page 442).

`vDSP_vrampmuladd_s8_24` (page 447)
: Vector fixed-point 8.24 format version of `vDSP_vrampmuladd` (page 442).

`vDSP_vrampmuladd2` (page 442)
: Stereo version of `vDSP_vrampmuladd` (page 442).

`vDSP_vrampmuladd2_s1_15` (page 443)
: Vector fixed-point 1.15 format version of `vDSP_vrampmuladd2` (page 442).

`vDSP_vrampmuladd2_s8_24` (page 445)
: Vector fixed-point 8.24 format version of `vDSP_vrampmuladd2` (page 442).

`vDSP_vgen` (page 392)
: Vector tapered ramp; single precision.

`vDSP_vgenD` (page 393)
: Vector tapered ramp; double precision.

`vDSP_vgenp` (page 394)
: Vector generate by extrapolation and interpolation; single precision.

`vDSP_vgenpD` (page 395)
: Vector generate by extrapolation and interpolation; double precision.

`vDSP_vtabi` (page 483)
: Vector interpolation, table lookup; single precision.

`vDSP_vtabiD` (page 484)
: Vector interpolation, table lookup; double precision.

Single-Vector Squaring

The functions in this group computes the square of each element in a vector or the square of the magnitude of each element in a complex vector.

`vDSP_vsq` (page 477)
: Computes the squared values of vector `input` and leaves the result in vector `result`; single precision.

`vDSP_vsqD` (page 477)
: Computes the squared values of vector `signal1` and leaves the result in vector `result`; double precision.

`vDSP_vssq` (page 478)
: Computes the signed squares of vector `signal1` and leaves the result in vector `result`; single precision.

`vDSP_vssqD` (page 478)
: Computes the signed squares of vector `signal1` and leaves the result in vector `result`; double precision.

`vDSP_zvmags` (page 534)
: Complex vector magnitudes squared; single precision.

`vDSP_zvmagsD` (page 535)
: Complex vector magnitudes squared; double precision.

`vDSP_zvmgsa` (page 535)
: Complex vector magnitudes square and add; single precision.

`vDSP_zvmgsaD` (page 536)
: Complex vector magnitudes square and add; double precision.

Single-Vector Polar-Rectangular Conversion

The functions in this group convert each element in a vector between rectangular and polar coordinates.

`vDSP_polar` (page 319)
: Rectangular to polar conversion; single precision.

`vDSP_polarD` (page 320)
: Rectangular to polar conversion; double precision.

`vDSP_rect` (page 321)
: Polar to rectangular conversion; single precision.

`vDSP_rectD` (page 321)
: Polar to rectangular conversion; double precision.

Single-Vector Conversion to Decibel Equivalents

The functions in this group convert power or amplitude values to decibel values.

`vDSP_vdbcon` (page 351)
: Vector convert power or amplitude to decibels; single precision.

`vDSP_vdbconD` (page 352)
: Vector convert power or amplitude to decibels; double precision.

Single-Vector Fractional Part Extraction

The functions in this group remove the whole-number part of each element in a vector, leaving the fractional part.

`vDSP_vfrac` (page 387)
: Vector truncate to fraction; single precision.

`vDSP_vfracD` (page 388)
: Vector truncate to fraction; double precision.

Single-Vector Complex Conjugation

The functions in this group find the complex conjugate of values in a vector.

`vDSP_zvconj` (page 529)
: Complex vector conjugate; single precision.

`vDSP_zvconjD` (page 530)
: Complex vector conjugate; double precision.

Single-Vector Phase Computation

The functions in this group compute the phase values of each element in a complex vector.

`vDSP_zvphas` (page 541)
: Complex vector phase; single precision.

`vDSP_zvphasD` (page 542)
: Complex vector phase; double precision.

Single-Vector Clipping, Limit, and Threshold Operations

The functions in this group restrict the values in a vector so that they fall within a given range or invert values outside a given range.

`vDSP_vclip` (page 344)
: Vector clip; single precision.

`vDSP_vclipD` (page 348)
: Vector clip; double precision.

`vDSP_vclipc` (page 345)
: Vector clip and count; single precision.

`vDSP_vclipcD` (page 347)
: Vector clip and count; double precision.

`vDSP_viclip` (page 397)
: Vector inverted clip; single precision.

`vDSP_viclipD` (page 398)
: Vector inverted clip; double precision.

`vDSP_vlim` (page 402)
: Vector test limit; single precision.

`vDSP_vlimD` (page 403)
: Vector test limit; double precision.

`vDSP_vthr` (page 485)
: Vector threshold; single precision.

`vDSP_vthrD` (page 486)
: Vector threshold; double precision.

`vDSP_vthres` (page 487)
: Vector threshold with zero fill; single precision.

`vDSP_vthresD` (page 488)
: Vector threshold with zero fill; double precision.

`vDSP_vthrsc` (page 488)
: Vector threshold with signed constant; single precision.

`vDSP_vthrscD` (page 489)
: Vector threshold with signed constant; double precision.

Single-Vector Compression

The functions in this group compress the values of a vector.

`vDSP_vcmprs` (page 350)
  Vector compress; single precision.

`vDSP_vcmprsD` (page 350)
  Vector compress; double precision.

Single-Vector Gathering

The functions in this group use either indices or pointers stored within one source vector to generate a new vector containing the chosen elements from either a second source vector or from memory.

`vDSP_vgathr` (page 389)
  Vector gather; single precision.

`vDSP_vgathrD` (page 391)
  Vector gather; double precision.

`vDSP_vgathra` (page 390)
  Vector gather, absolute pointers; single precision.

`vDSP_vgathraD` (page 391)
  Vector gather, absolute pointers; double precision.

`vDSP_vindex` (page 399)
  Vector index; single precision.

`vDSP_vindexD` (page 400)
  Vector index; double precision.

Single-Vector Reversing

The functions in this group reverse the order of the elements in a vector.

`vDSP_vrvrs` (page 451)
  Vector reverse order, in place; single precision.

`vDSP_vrvrsD` (page 452)
  Vector reverse order, in place; double precision.

Single-Vector Copying

The functions in this group copy one vector to another vector.

`vDSP_zvmov` (page 537)
  Complex vector copy; single precision.

`vDSP_zvmovD` (page 538)
  Complex vector copy; double precision.

Single-Vector Zero Crossing Search

The functions in this group find a specified number of zero crossings, returning the last crossing found and the number of crossings found.

`vDSP_nzcros` (page 317)
: Find zero crossings; single precision.

`vDSP_nzcrosD` (page 318)
: Find zero crossings; double precision.

Single-Vector Operations: Linear Averaging

The functions in this group take a vector that contains the linear averages of values from other vectors and adds an additional vector into those averages.

`vDSP_vavlin` (page 342)
: Vector linear average; single precision.

`vDSP_vavlinD` (page 343)
: Vector linear average; double precision.

Single-Vector Linear Interpolation

The functions in this group calculate the linear interpolation between neighboring elements.

`vDSP_vlint` (page 404)
: Vector linear interpolation between neighboring elements; single precision.

`vDSP_vlintD` (page 406)
: Vector linear interpolation between neighboring elements; double precision.

Single-Vector Integration

The functions in this group perform integration on the values in a vector.

`vDSP_vrsum` (page 449)
: Vector running sum integration; single precision.

`vDSP_vrsumD` (page 450)
: Vector running sum integration; double precision.

`vDSP_vsimps` (page 464)
: Simpson integration; single precision.

`vDSP_vsimpsD` (page 465)
: Simpson integration; double precision.

`vDSP_vtrapz` (page 492)
: Vector trapezoidal integration; single precision.

`vDSP_vtrapzD` (page 493)
: Vector trapezoidal integration; double precision.

Single-Vector Sorting

The functions in this group find sort the values in a vector.

`vDSP_vsort` (page 473)
: Vector in-place sort; single precision.

`vDSP_vsortD` (page 474)
: Vector in-place sort; double precision.

`vDSP_vsorti` (page 475)
: Vector index in-place sort; single precision.

`vDSP_vsortiD` (page 475)
: Vector index in-place sort; double precision.

Single-Vector Sliding-Window Summing

The functions in this group calculates a sliding-window sum for a vector.

`vDSP_vswsum` (page 481)
: Vector sliding window sum; single precision.

`vDSP_vswsumD` (page 482)
: Vector sliding window sum; double precision.

Single-Vector Precision Conversion

The functions in this group convert between single-precision and double-precision floating-point vectors.

`vDSP_vdpsp` (page 358)
: Vector convert double-precision to single-precision.

`vDSP_vspdp` (page 476)
: Vector convert single-precision to double-precision.

Single-Vector Floating-Point to Integer Conversion

The functions in this group convert the values in a vector from floating-point values to integer values of a given size.

`vDSP_vfix8` (page 367)
: Converts an array of single-precision floating-point values to signed 8-bit integer values, rounding towards zero.

`vDSP_vfix8D` (page 368)
: Converts an array of double-precision floating-point values to signed 8-bit integer values, rounding towards zero.

`vDSP_vfix16` (page 365)
: Converts an array of single-precision floating-point values to signed 16-bit integer values, rounding towards zero.

`vDSP_vfix16D` (page 365)
: Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards zero.

`vDSP_vfix32` (page 366)
: Converts an array of single-precision floating-point values to signed 32-bit integer values, rounding towards zero.

`vDSP_vfix32D` (page 366)
: Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards zero.

`vDSP_vfixr8` (page 371)
: Converts an array of single-precision floating-point values to signed 8-bit integer values, rounding towards nearest integer.

`vDSP_vfixr8D` (page 372)
: Converts an array of double-precision floating-point values to signed 8-bit integer values, rounding towards nearest integer.

`vDSP_vfixr16` (page 368)
: Converts an array of single-precision floating-point values to signed 16-bit integer values, rounding towards nearest integer.

`vDSP_vfixr16D` (page 369)
: Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards nearest integer.

`vDSP_vfixr32` (page 370)
: Converts an array of single-precision floating-point values to signed 32-bit integer values, rounding towards nearest integer.

`vDSP_vfixr32D` (page 370)
: Converts an array of double-precision floating-point values to signed 32-bit integer values, rounding towards nearest integer.

`vDSP_vfixu8` (page 379)
: Converts an array of single-precision floating-point values to unsigned 8-bit integer values, rounding towards zero.

`vDSP_vfixu8D` (page 380)
: Converts an array of double-precision floating-point values to unsigned 8-bit integer values, rounding towards zero.

`vDSP_vfixu16` (page 376)
: Converts an array of single-precision floating-point values to unsigned 16-bit integer values, rounding towards zero.

`vDSP_vfixu16D` (page 377)
: Converts an array of double-precision floating-point values to unsigned 16-bit integer values, rounding towards zero.

`vDSP_vfixu32` (page 378)
: Converts an array of single-precision floating-point values to unsigned 32-bit integer values, rounding towards zero.

`vDSP_vfixu32D` (page 378)
: Converts an array of double-precision floating-point values to unsigned 32-bit integer values, rounding towards zero.

`vDSP_vfixru8` (page 375)
: Converts an array of single-precision floating-point values to unsigned 8-bit integer values, rounding towards nearest integer.

`vDSP_vfixru8D` (page 376)
: Converts an array of double-precision floating-point values to unsigned 8-bit integer values, rounding towards nearest integer.

`vDSP_vfixru16` (page 372)
: Converts an array of single-precision floating-point values to unsigned 16-bit integer values, rounding towards nearest integer.

`vDSP_vfixru16D` (page 373)
: Converts an array of double-precision floating-point values to unsigned 16-bit integer values, rounding towards nearest integer.

`vDSP_vfixru32` (page 374)
: Converts an array of single-precision floating-point values to unsigned 32-bit integer values, rounding towards nearest integer.

`vDSP_vfixru32D` (page 374)
: Converts an array of double-precision floating-point values to unsigned 32-bit integer values, rounding towards nearest integer.

Single-Vector Integer to Floating-Point Conversion

The functions in this group convert integer values of a given size to floating-point vectors.

`vDSP_vflt8` (page 383)
: Converts an array of signed 8-bit integers to single-precision floating-point values.

`vDSP_vflt8D` (page 383)
: Converts an array of signed 8-bit integers to double-precision floating-point values.

`vDSP_vflt16` (page 380)
: Converts an array of signed 16-bit integers to single-precision floating-point values.

`vDSP_vflt16D` (page 381)
: Converts an array of signed 16-bit integers to double-precision floating-point values.

`vDSP_vflt32` (page 381)
: Converts an array of signed 32-bit integers to single-precision floating-point values.

`vDSP_vflt32D` (page 382)
: Converts an array of signed 32-bit integers to double-precision floating-point values.

`vDSP_vfltu8` (page 386)
: Converts an array of unsigned 8-bit integers to single-precision floating-point values.

`vDSP_vfltu8D` (page 387)
: Converts an array of unsigned 8-bit integers to double-precision floating-point values.

`vDSP_vfltu16` (page 384)
: Converts an array of unsigned 16-bit integers to single-precision floating-point values.

`vDSP_vfltu16D` (page 384)
: Converts an array of unsigned 16-bit integers to double-precision floating-point values.

`vDSP_vfltu32` (page 385)
: Converts an array of unsigned 32-bit integers to single-precision floating-point values.

`vDSP_vfltu32D` (page 386)
: Converts an array of unsigned 32-bit integers to double-precision floating-point values.

Vector-Scalar Addition

These functions add a scalar to each element of a vector.

`vDSP_vsadd` (page 453)
: Vector scalar add; single precision.

`vDSP_vsaddD` (page 453)
: Vector scalar add; double precision.

`vDSP_vsaddi` (page 454)
: Integer vector scalar add.

Vector-Scalar Division

These functions divide each element of a vector by a scalar.

`vDSP_vsdiv` (page 461)
: Vector scalar divide; single precision.

`vDSP_vsdivD` (page 462)
: Vector scalar divide; double precision.

`vDSP_vsdivi` (page 463)
: Integer vector scalar divide.

`vDSP_zvdiv` (page 531)
: Complex vector divide; single precision.

`vDSP_zvdivD` (page 532)
: Complex vector divide; double precision.

`vDSP_svdiv` (page 324)
: Divide scalar by vector; single precision.

`vDSP_svdivD` (page 325)
: Divide scalar by vector; double precision.

Vector-Scalar Multiplication

These functions multiply a scalar by each element of a vector.

`vDSP_vsma` (page 466)
: Vector scalar multiply and vector add; single precision.

`vDSP_vsmaD` (page 467)
: Vector scalar multiply and vector add; double precision.

`vDSP_zvsma` (page 542)
: Complex vector scalar multiply and add; single precision.

`vDSP_zvsmaD` (page 543)
: Complex vector scalar multiply and add; double precision.

`vDSP_vsmul` (page 472)
: Multiplies vector `signal1` by scalar `signal2` and leaves the result in vector `result`; single precision.

`vDSP_vsmulD` (page 473)
: Multiplies vector `signal1` by scalar signal2 and leaves the result in vector `result`; double precision.

`vDSP_zvzsml` (page 546)
Complex vector multiply by complex scalar; single precision.

`vDSP_zvzsmlD` (page 547)
Complex vector multiply by complex scalar; double precision.

Vector-Scalar Multiply and Add

These functions multiply a scalar with each element of a vector, ten add another scalar.

`vDSP_vsmsa` (page 468)
Vector scalar multiply and scalar add; single precision.

`vDSP_vsmsaD` (page 469)
Vector scalar multiply and scalar add; double precision.

`vDSP_vsmsb` (page 470)
Vector scalar multiply and vector subtract; single precision.

`vDSP_vsmsbD` (page 471)
Vector scalar multiply and vector subtract; double precision.

Vector-to-Vector Bitwise Logical Equivalence

`vDSP_veqvi` (page 362)
Vector equivalence, 32-bit logical.

Vector-to-Vector Real Vector Basic Arithmetic

`vDSP_vadd` (page 336)
Adds vector `A` to vector `B` and leaves the result in vector `C`; single precision.

`vDSP_vaddD` (page 336)
Adds vector `A` to vector `B` and leaves the result in vector `C`; double precision.

`vDSP_vsub` (page 478)
Subtracts vector `signal1` from vector `signal2` and leaves the result in vector `result`; single precision.

`vDSP_vsubD` (page 479)
Subtracts vector `signal1` from vector `signal2` and leaves the result in vector `result`; double precision.

`vDSP_vam` (page 337)
Adds vectors `A` and `B`, multiplies the sum by vector `C`, and leaves the result in vector `D`; single precision.

`vDSP_vamD` (page 337)
Adds vectors `A` and `B`, multiplies the sum by vector `C`, and leaves the result in vector `D`; double precision.

`vDSP_vsbm` (page 455)
Vector subtract and multiply; single precision.

`vDSP_vsbmD` (page 456)
Vector subtract and multiply; double precision.

`vDSP_vaam` (page 331)
: Vector add, add, and multiply; single precision.

`vDSP_vaamD` (page 333)
: Vector add, add, and multiply; double precision.

`vDSP_vsbsbm` (page 457)
: Vector subtract, subtract, and multiply; single precision.

`vDSP_vsbsbmD` (page 458)
: Vector subtract, subtract, and multiply; double precision.

`vDSP_vasbm` (page 338)
: Vector add, subtract, and multiply; single precision.

`vDSP_vasbmD` (page 339)
: Vector add, subtract, and multiply; double precision.

`vDSP_vasm` (page 340)
: Vector add and scalar multiply; single precision.

`vDSP_vasmD` (page 341)
: Vector add and scalar multiply; double precision.

`vDSP_vsbsm` (page 460)
: Vector subtract and scalar multiply; single precision.

`vDSP_vsbsmD` (page 461)
: Vector subtract and scalar multiply; double precision.

`vDSP_vmsa` (page 421)
: Vector multiply and scalar add; single precision.

`vDSP_vmsaD` (page 422)
: Vector multiply and scalar add; double precision.

`vDSP_vdiv` (page 355)
: Vector divide; single precision.

`vDSP_vdivD` (page 356)
: Vector divide; double precision.

`vDSP_vdivi` (page 357)
: Vector divide; integer.

`vDSP_vmul` (page 425)
: Multiplies vector `A` by vector `B` and leaves the result in vector `C`; single precision.

`vDSP_vmulD` (page 426)
: Multiplies vector `A` by vector `B` and leaves the result in vector `C`; double precision.

`vDSP_vma` (page 407)
: Vector multiply and add; single precision.

`vDSP_vmaD` (page 408)
: Vector multiply and add; double precision.

`vDSP_vmsb` (page 423)
: Vector multiply and subtract, single precision.

`vDSP_vmsbD` (page 424)
: Vector multiply and subtract; double precision.

`vDSP_vmma` (page 416)
: Vector multiply, multiply, and add; single precision.

`vDSP_vmmaD` (page 418)
    Vector multiply, multiply, and add; double precision.

`vDSP_vmmsb` (page 419)
    Vector multiply, multiply, and subtract; single precision.

`vDSP_vmmsbD` (page 420)
    Vector multiply, multiply, and subtract; double precision.

Vector-to-Vector Complex Vector Basic Arithmetic

`vDSP_zrvdiv` (page 516)
    Divides complex vector `A` by real vector `B` and leaves the result in vector `C`; single precision.

`vDSP_zrvdivD` (page 517)
    Divides complex vector `A` by real vector `B` and leaves the result in vector `C`; double precision.

`vDSP_zrvmul` (page 517)
    Multiplies complex vector `A` by real vector `B` and leaves the result in vector `C`; single precision.

`vDSP_zrvmulD` (page 518)
    Multiplies complex vector `A` by real vector `B` and leaves the result in vector `C`; double precision.

`vDSP_zrvsub` (page 519)
    Subtracts real vector `B` from complex vector `A` and leaves the result in complex vector `C`; single precision.

`vDSP_zrvsubD` (page 520)
    Subtracts real vector `B` from complex vector `A` and leaves the result in complex vector `C`; double precision.

`vDSP_zrvadd` (page 514)
    Adds real vector `B` to complex vector `A` and leaves the result in complex vector `C`; single precision.

`vDSP_zrvaddD` (page 515)
    Adds real vector `B` to complex vector `A` and leaves the result in complex vector `C`; double precision.

`vDSP_zvadd` (page 525)
    Adds complex vectors `A` and `B` and leaves the result in complex vector `C`; single precision.

`vDSP_zvaddD` (page 526)
    Adds complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

`vDSP_zvcmul` (page 528)
    Complex vector conjugate and multiply; single precision.

`vDSP_zvcmulD` (page 528)
    Complex vector conjugate and multiply; double precision.

`vDSP_zvmul` (page 538)
    Multiplies complex vectors `A` and `B` and leaves the result in complex vector `C`; single precision.

`vDSP_zvmulD` (page 539)
    Multiplies complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

`vDSP_zvsub` (page 544)
    Subtracts complex vector `B` from complex vector `A` and leaves the result in complex vector `C`; single precision.

`vDSP_zvsubD` (page 545)
: Subtracts complex vector `B` from complex vector `A` and leaves the result in complex vector `C`; double precision.

`vDSP_zvcma` (page 527)
: Multiplies complex vector `B` by the complex conjugates of complex vector `A`, adds the products to complex vector `C`, and stores the results in complex vector `D`; single precision.

`vDSP_zvcmaD` (page 527)
: Multiplies complex vector `B` by the complex conjugates of complex vector `A`, adds the products to complex vector `C`, and stores the results in complex vector `D`; double precision.

Vector-to-Vector Maxima and Minima

`vDSP_vmax` (page 409)
: Vector maxima; single precision.

`vDSP_vmaxD` (page 409)
: Vector maxima; double precision.

`vDSP_vmaxmg` (page 410)
: Vector maximum magnitudes; single precision.

`vDSP_vmaxmgD` (page 411)
: Vector maximum magnitudes; double precision.

`vDSP_vmin` (page 412)
: Vector minima; single precision.

`vDSP_vminD` (page 413)
: Vector minima; double precision.

`vDSP_vminmg` (page 414)
: Vector minimum magnitudes; single precision.

`vDSP_vminmgD` (page 415)
: Vector minimum magnitudes; double precision.

Vector-to-Vector Distance Computation

`vDSP_vdist` (page 353)
: Vector distance; single precision.

`vDSP_vdistD` (page 354)
: Vector distance; double precision.

Vector-to-Vector Interpolation

`vDSP_vintb` (page 400)
: Vector linear interpolation between vectors; single precision.

`vDSP_vintbD` (page 401)
: Vector linear interpolation between vectors; double precision.

`vDSP_vqint` (page 434)
    Vector quadratic interpolation; single precision.

`vDSP_vqintD` (page 435)
    Vector quadratic interpolation; double precision.

Vector-to-Vector Polynomial Evaluation

`vDSP_vpoly` (page 429)
    Vector polynomial evaluation; single precision.

`vDSP_vpolyD` (page 430)
    Vector polynomial evaluation; double precision.

Vector-to-Vector Pythagoras Computation

The functions in this group apply Pythagoras's theorem to vectors.

`vDSP_vpythg` (page 431)
    Vector Pythagoras; single precision.

`vDSP_vpythgD` (page 433)
    Vector Pythagoras; double precision.

Vector-to-Vector Extrema Finding

`vDSP_venvlp` (page 359)
    Vector envelope; single precision.

`vDSP_venvlpD` (page 360)
    Vector envelope; double precision.

Vector-to-Vector Element Swapping

`vDSP_vswap` (page 479)
    Vector swap; single precision.

`vDSP_vswapD` (page 480)
    Vector swap; double precision.

Vector-to-Vector Merging

`vDSP_vtmerg` (page 490)
    Tapered merge of two vectors; single precision.

`vDSP_vtmergD` (page 491)
    Tapered merge of two vectors; double precision.

Vector-to-Vector Spectra Computation

`vDSP_zaspec` (page 497)
: Computes an accumulating autospectrum; single precision.

`vDSP_zaspecD` (page 498)
: Computes an accumulating autospectrum; double precision.

`vDSP_zcspec` (page 501)
: Accumulating cross-spectrum on two complex vectors; single precision.

`vDSP_zcspecD` (page 502)
: Accumulating cross-spectrum on two complex vectors; double precision.

Vector-to-Vector Coherence Function Computation

`vDSP_zcoher` (page 498)
: Coherence function of two signals; single precision.

`vDSP_zcoherD` (page 499)
: Coherence function of two signals; double precision.

Vector-to-Vector Transfer Function Computation

`vDSP_ztrans` (page 522)
: Transfer function; single precision.

`vDSP_ztransD` (page 523)
: Transfer function; double precision.

Vector-to-Vector Recursive Filtering on Real Vectors

`vDSP_deq22` (page 198)
: Difference equation, 2 poles, 2 zeros; single precision.

`vDSP_deq22D` (page 199)
: Difference equation, 2 poles, 2 zeros; double precision.

Calculating Dot Products

`vDSP_dotpr` (page 207)
: Computes the dot or scalar product of vectors `A` and `B` and leaves the result in scalar `*C`; single precision.

`vDSP_dotpr_s1_15` (page 212)
: Vector fixed-point 1.15 format version of `vDSP_dotpr` (page 207).

`vDSP_dotpr_s8_24` (page 212)
: Vector fixed-point 8.24 format version of `vDSP_dotpr` (page 207).

`vDSP_dotprD` (page 211)
: Computes the dot or scalar product of vectors `A` and `B` and leaves the result in scalar `*C`; double precision.

`vDSP_dotpr2` (page 208)
: Stereo variant of `vDSP_dotpr` (page 207).

`vDSP_dotpr2_s1_15` (page 209)
: Vector fixed-point 1.15 format version of `vDSP_dotpr2` (page 208).

`vDSP_dotpr2_s8_24` (page 210)
: Vector fixed-point 8.24 format version of `vDSP_dotpr2` (page 208).

`vDSP_zdotpr` (page 502)
: Calculates the complex dot product of complex vectors `A` and `B` and leaves the result in complex vector `C`; single precision.

`vDSP_zdotprD` (page 503)
: Calculates the complex dot product of complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

`vDSP_zidotpr` (page 504)
: Calculates the conjugate dot product (or inner dot product) of complex vectors `A` and `B` and leave the result in complex vector `C`; single precision.

`vDSP_zidotprD` (page 504)
: Calculates the conjugate dot product (or inner dot product) of complex vectors `A` and `B` and leave the result in complex vector `C`; double precision.

`vDSP_zrdotpr` (page 513)
: Calculates the complex dot product of complex vector `A` and real vector `B` and leaves the result in complex vector `C`; single precision.

`vDSP_zrdotprD` (page 514)
: Calculates the complex dot product of complex vector `A` and real vector `B` and leaves the result in complex vector `C`; double precision.

Finding Maximums

`vDSP_maxv` (page 297)
: Vector maximum value; single precision.

`vDSP_maxvD` (page 298)
: Vector maximum value; double precision.

`vDSP_maxvi` (page 298)
: Vector maximum value with index; single precision.

`vDSP_maxviD` (page 299)
: Vector maximum value with index; double precision.

`vDSP_maxmgv` (page 294)
: Vector maximum magnitude; single precision.

`vDSP_maxmgvD` (page 294)
: Vector maximum magnitude; double precision.

`vDSP_maxmgvi` (page 295)
: Vector maximum magnitude with index; single precision.

`vDSP_maxmgviD` (page 296)
: Vector maximum magnitude with index; double precision.

Finding Minimums

`vDSP_minv` (page 308)
: Vector minimum value.

`vDSP_minvD` (page 308)
: Vector minimum value; double precision.

`vDSP_minvi` (page 309)
: Vector minimum value with index; single precision.

`vDSP_minviD` (page 310)
: Vector minimum value with index; double precision.

`vDSP_minmgv` (page 304)
: Vector minimum magnitude; single precision.

`vDSP_minmgvD` (page 305)
: Vector minimum magnitude; double precision.

`vDSP_minmgvi` (page 306)
: Vector minimum magnitude with index; single precision.

`vDSP_minmgviD` (page 307)
: Vector minimum magnitude with index; double precision.

Calculating Means

`vDSP_meanv` (page 302)
: Vector mean value; single precision.

`vDSP_meanvD` (page 302)
: Vector mean value; double precision.

`vDSP_meamgv` (page 300)
: Vector mean magnitude; single precision.

`vDSP_meamgvD` (page 301)
: Vector mean magnitude; double precision.

`vDSP_measqv` (page 303)
: Vector mean square value; single precision.

`vDSP_measqvD` (page 304)
: Vector mean square value; double precision.

`vDSP_mvessq` (page 315)
: Vector mean of signed squares; single precision.

`vDSP_mvessqD` (page 316)
: Vector mean of signed squares; double precision.

`vDSP_rmsqv` (page 322)
: Vector root-mean-square; single precision.

`vDSP_rmsqvD` (page 323)
Vector root-mean-square; double precision.

Summing Vectors

`vDSP_sve` (page 326)
Vector sum; single precision.

`vDSP_sveD` (page 326)
Vector sum; double precision.

`vDSP_svemg` (page 327)
Vector sum of magnitudes; single precision.

`vDSP_svemgD` (page 328)
Vector sum of magnitudes; double precision.

`vDSP_svesq` (page 328)
Vector sum of squares; single precision.

`vDSP_svesqD` (page 329)
Vector sum of squares; double precision.

`vDSP_svs` (page 330)
Vector sum of signed squares; single precision.

`vDSP_svsD` (page 331)
Vector sum of signed squares; double precision.

Matrix Multiplication (Real Matrices)

The functions in this group multiply two matrices.

`vDSP_mmul` (page 312)
Performs an out-of-place multiplication of two matrices; single precision.

`vDSP_mmulD` (page 313)
Performs an out-of-place multiplication of two matrices; double precision.

`vDSP_zmma` (page 505)
Multiplies two complex matrices, then adds a third complex matrix; out-of-place; single precision.

`vDSP_zmmaD` (page 505)
Multiplies two complex matrices, then adds a third complex matrix; out-of-place; double precision.

`vDSP_zmms` (page 506)
Multiplies two complex matrices, then subtracts a third complex matrix; out-of-place; single precision.

`vDSP_zmmsD` (page 507)
Multiplies two complex matrices, then subtracts a third complex matrix; out-of-place; double precision.

`vDSP_zmmul` (page 508)
Multiplies two matrices of complex numbers; out-of-place; single precision.

`vDSP_zmmulD` (page 509)
Multiplies two matrices of complex numbers; out-of-place; double precision.

`vDSP_zmsm` (page 510)
: Subtracts the product of two complex matrices from a third complex matrix; out-of-place; single precision.

`vDSP_zmsmD` (page 511)
: Subtracts the product of two complex matrices from a third complex matrix; out-of-place; double precision.

Matrix Transposition

`vDSP_mtrans` (page 314)
: Creates a transposed matrix `C` from a source matrix `A`; single precision.

`vDSP_mtransD` (page 315)
: Creates a transposed matrix `C` from a source matrix `A`; double precision.

Matrix and Submatrix Copying

`vDSP_mmov` (page 311)
: Copies the contents of a submatrix to another submatrix.

`vDSP_mmovD` (page 311)
: Copies the contents of a submatrix to another submatrix.

1D Fast Fourier Transforms (Support Functions)

`vDSP_create_fftsetup` (page 195)
: Builds a data structure that contains precalculated data for use by single-precision FFT functions.

`vDSP_create_fftsetupD` (page 196)
: Builds a data structure that contains precalculated data for use by double-precision FFT functions.

`vDSP_destroy_fftsetup` (page 201)
: Frees an existing single-precision FFT data structure.

`vDSP_destroy_fftsetupD` (page 202)
: Frees an existing double-precision FFT data structure.

1D Fast Fourier Transforms (In-Place Real)

`vDSP_fft_zrip` (page 279)
: Computes an in-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zrip` (page 259)
: Performs the same operation as `vDSP_fft_zrip` (page 279) on multiple signals with a single call.

`vDSP_fft_zripD` (page 281)
: Computes an in-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zripD` (page 261)
: Performs the same operation as `vDSP_fft_zripD` (page 281) on multiple signals with a single call.

`vDSP_fft_zript` (page 282)
: Computes an in-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zript` (page 262)
: Performs the same operation as `vDSP_fft_zript` (page 282) on multiple signals with a single call.

`vDSP_fft_zriptD` (page 283)
: Computes an in-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zriptD` (page 263)
: Performs the same operation as `vDSP_fft_zriptD` (page 283) on multiple signals with a single call.

1D Fast Fourier Transforms (Out-of-Place Real)

`vDSP_fft_zrop` (page 284)
: Computes an out-of-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zrop` (page 264)
: Performs the same operation as `vDSP_fft_zrop` (page 284) on multiple signals with a single call.

`vDSP_fft_zropD` (page 285)
: Computes an out-of-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zropD` (page 265)
: Performs the same operation as `VDSP_fft_zropD` on multiple signals with a single call.

`vDSP_fft_zropt` (page 286)
: Computes an out-of-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zropt` (page 267)
: Performs the same operation as `VDSP_fft_zropt` on multiple signals with a single call.

`vDSP_fft_zroptD` (page 288)
: Computes an out-of-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zroptD` (page 268)
: Performs the same operation as `VDSP_fft_zroptD` on multiple signals with a single call.

1D Fast Fourier Transforms (In-Place Complex)

`vDSP_fft_zip` (page 270)
: Computes an in-place single-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zip` (page 249)
: Performs the same operation as `vDSP_fft_zip` (page 270) on multiple signals with a single call.

`vDSP_fft_zipD` (page 271)
: Computes an in-place double-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zipD` (page 250)
: Performs the same operation as `vDSP_fft_zipD` (page 271) on multiple signals with a single call.

`vDSP_fft_zipt` (page 272)
: Computes an in-place single-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse). A buffer is used for intermediate results.

`vDSP_fftm_zipt` (page 251)
: Performs the same operation as `vDSP_fft_zipt` (page 272) on multiple signals with a single call.

`vDSP_fft_ziptD` (page 273)
: Computes an in-place double-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse). A buffer is used for intermediate results.

`vDSP_fftm_ziptD` (page 252)
: Performs the same operation as `vDSP_fft_ziptD` (page 273) on multiple signals with a single call.

1D Fast Fourier Transforms (Out-of-Place Complex)

`vDSP_fft_zop` (page 274)
: Computes an out-of-place single-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fft_zopD` (page 276)
: Computes an out-of-place double-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zop` (page 253)
: Performs the same operation as `vDSP_fft_zop` (page 274) on multiple signals with a single call.

`vDSP_fftm_zopD` (page 255)
: Performs the same operation as `vDSP_fft_zopD` (page 276) on multiple signals with a single call.

`vDSP_fft_zopt` (page 277)
: Computes an out-of-place single-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fft_zoptD` (page 278)
: Computes an out-of-place double-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

`vDSP_fftm_zopt` (page 256)
: Performs the same operation as `vDSP_fft_zopt` (page 277) on multiple signals with a single call.

`vDSP_fftm_zoptD` (page 258)
: Performs the same operation as `vDSP_fft_zoptD` (page 278) on multiple signals with a single call.

`vDSP_fft3_zop` (page 244)
: Computes an out-of-place radix-3 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 3 times the power of 2 specified by parameter `log2n`; single precision.

`vDSP_fft3_zopD` (page 245)
: Computes an out-of-place radix-3 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 3 times the power of 2 specified by parameter `log2n`; double precision.

`vDSP_fft5_zop` (page 246)
: Computes an out-of-place radix-5 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 5 times the power of 2 specified by parameter `log2n`; single precision.

`vDSP_fft5_zopD` (page 248)
: Computes an out-of-place radix-5 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 5 times the power of 2 specified by parameter `log2n`; double precision.

1D Fast Fourier Transforms (Fixed Length)

`vDSP_FFT16_copv` (page 216)
: Performs a 16-element FFT on interleaved complex data.

`vDSP_FFT32_copv` (page 243)
: Performs a 32-element FFT on interleaved complex data.

`vDSP_FFT16_zopv` (page 217)
: Performs a 16-element FFT on split complex data.

`vDSP_FFT32_zopv` (page 243)
: Performs a 32-element FFT on split complex data.

2D Fast Fourier Transforms (In-Place Complex)

`vDSP_fft2d_zip` (page 217)
: Computes an in-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zipD` (page 219)
: Computes an in-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zipt` (page 220)
: Computes an in-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

`vDSP_fft2d_ziptD` (page 222)
: Computes an in-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

2D Fast Fourier Transforms (Out-of-Place Complex)

`vDSP_fft2d_zop` (page 224)
: Computes an out-of-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zopD` (page 225)
: Computes an out-of-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zopt` (page 226)
: Computes an out-of-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

`vDSP_fft2d_zoptD` (page 228)
: Computes an out-of-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

2D Fast Fourier Transforms (In-Place Real)

`vDSP_fft2d_zrip` (page 230)
: Computes an in-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zripD` (page 231)
: Computes an in-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zript` (page 233)
: Computes an in-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

`vDSP_fft2d_zriptD` (page 234)
: Computes an in-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

2D Fast Fourier Transforms (Out-of-Place Real)

`vDSP_fft2d_zrop` (page 236)
: Computes an out-of-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zropD` (page 238)
: Computes an out-of-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

`vDSP_fft2d_zropt` (page 239)
: Computes an out-of-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

`vDSP_fft2d_zroptD` (page 241)
: Computes an out-of-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

Discrete Fourier Transforms

These functions calculate a discrete Fourier transform of a specified length on a vector.

`vDSP_DFT_CreateSetup` (page 202)
: Old name for `vDSP_DFT_zop_CreateSetup` (page 205).

`vDSP_DFT_DestroySetup` (page 203)
: Releases a setup object.

`vDSP_DFT_Execute` (page 203)
: Calculates the discrete fourier transform for a vector using `vDSP_DFT_zop` (page 204) or `vDSP_DFT_zrop` (page 206).

`vDSP_DFT_zop` (page 204)
: Calculates the discrete fourier transform for a vector (single-precision complex).

`vDSP_DFT_zop_CreateSetup` (page 205)
: Creates data structures for use with `vDSP_DFT_zop` (page 204).

`vDSP_DFT_zrop` (page 206)
: Calculates the discrete fourier transform for a vector (single-precision real).

`vDSP_DFT_zrop_CreateSetup` (page 206)
: Creates data structures for use with `vDSP_DFT_zrop` (page 206).

Correlation and Convolution

These functions perform correlation and convolution operations on real or complex signals in vDSP.

`vDSP_conv` (page 193)
: Performs either correlation or convolution on two vectors; single precision.

`vDSP_convD` (page 194)
: Performs either correlation or convolution on two vectors; double precision.

`vDSP_zconv` (page 499)
: Performs either correlation or convolution on two complex vectors; single precision.

`vDSP_zconvD` (page 500)
: Performs either correlation or convolution on two complex vectors; double precision.

`vDSP_wiener` (page 494)
: Wiener-Levinson general convolution; single precision.

`vDSP_wienerD` (page 496)
: Wiener-Levinson general convolution; double precision.

`vDSP_desamp` (page 199)
: Convolution with decimation; single precision.

`vDSP_desampD` (page 200)
: Convolution with decimation; double precision.

`vDSP_zrdesamp` (page 512)
: Complex/real downsample with anti-aliasing; single precision.

`vDSP_zrdesampD` (page 512)
: Complex/real downsample with anti-aliasing; double precision.

Windowing and Filtering

This section describes the C API for performing filtering operations on real or complex signals in vDSP. It also describes the built-in support for windowing functions such as Blackman, Hamming, and Hann windows.

`vDSP_blkman_window` (page 192)
: Creates a single-precision Blackman window.

`vDSP_blkman_windowD` (page 192)
: Creates a double-precision Blackman window.

`vDSP_hamm_window` (page 289)
: Creates a single-precision Hamming window.

`vDSP_hamm_windowD` (page 290)
: Creates a double-precision Hamming window.

`vDSP_hann_window` (page 290)
: Creates a single-precision Hanning window.

`vDSP_hann_windowD` (page 291)
: Creates a double-precision Hanning window.

`vDSP_f3x3` (page 213)
: Filters an image by performing a two-dimensional convolution with a 3x3 kernel; single precision.

`vDSP_f3x3D` (page 214)
: Filters an image by performing a two-dimensional convolution with a 3x3 kernel; double precision.

`vDSP_f5x5` (page 215)
: Filters an image by performing a two-dimensional convolution with a 5x5 kernel; single precision.

`vDSP_f5x5D` (page 215)
: Filters an image by performing a two-dimensional convolution with a 5x5 kernel; double precision.

`vDSP_imgfir` (page 292)
: Filters an image by performing a two-dimensional convolution with a kernel; single precision.

`vDSP_imgfirD` (page 293)
: Filters an image by performing a two-dimensional convolution with a kernel; double precision.

Complex Vector Conversion

These functions convert complex vectors between interleaved and split forms.

`vDSP_ctoz` (page 197)
: Copies the contents of an interleaved complex vector `C` to a split complex vector `Z`; single precision.

`vDSP_ctozD` (page 197)
: Copies the contents of an interleaved complex vector `C` to a split complex vector `Z`; double precision.

`vDSP_ztoc` (page 521)
: Copies the contents of a split complex vector `A` to an interleaved complex vector `C`; single precision.

`vDSP_ztocD` (page 522)
Copies the contents of a split complex vector `A` to an interleaved complex vector `C`; double precision.

Functions

vDSP_blkman_window

Creates a single-precision Blackman window.

```
void vDSP_blkman_window (
   float *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion
Represented in pseudo-code, this function does the following:

```
for (n=0; n < N; ++n)
{
    C[n] = 0.42 - (0.5 * cos(  2 * pi * n / N ) ) + (0.08 * cos( 4 * pi * n /
N) );
}
```

`vDSP_blkman_window` creates a single-precision Blackman window function `C`, which can be multiplied by a vector using `vDSP_vmul`. Specify the `vDSP_HALF_WINDOW` (page 553) flag to create only the first `(n+1)/2` points, or `0` (zero) for a full-size window.

See also `vDSP_vmul` (page 425).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_blkman_windowD

Creates a double-precision Blackman window.

```
void vDSP_blkman_windowD (
   double *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion
Represented in pseudo-code, this function does the following:

```
for (n=0; n < N; ++n)
{
    C[n] = 0.42 - (0.5 * cos(  2 * pi * n / N ) ) + (0.08 * cos( 4 * pi * n /
N) );
}
```

vDSP_blkman_windowD (page 192) creates a double-precision Blackman window function C, which can be multiplied by a vector using vDSP_vmulD . Specify the vDSP_HALF_WINDOW (page 553) flag to create only the first (n+1)/2 points, or 0 (zero) for a full-size window.

See also vDSP_vmulD (page 426).

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_conv

Performs either correlation or convolution on two vectors; single precision.

```
void vDSP_conv (
   const float __vDSP_signal[],
   vDSP_Stride __vDSP_signalStride,
   const float __vDSP_filter[],
   vDSP_Stride __vDSP_strideFilter,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_lenResult,
   vDSP_Length __vDSP_lenFilter
);
```

Discussion

$$C_{nK} = \sum_{p=0}^{P-1} A_{(n+p)I} B_{pJ} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

If filterStride is positive, vDSP_conv performs correlation. If filterStride is negative,it performs convolution and *filtermust point to the last vector element. The function can run in place, but result cannot be in place with filter.

The value of lenFilter must be less than or equal to 2044. The length of vector signal must satisfy two criteria: it must be

- equal to or greater than 12
- equal to or greater than the sum of N-1 plus the nearest multiple of 4 that is equal to or greater than the value of lenFilter.

Criteria to invoke vectorized code:

- The vectors signal and result must be relatively aligned.
- The value of lenFilter must be between 4 and 256, inclusive.
- The value of lenResult must be greater than 36.

- The values of `signalStride` and `resultStride` must be 1.
- The value of `filterStride` must be either 1 or -1.

If any of these criteria is not satisfied, the function invokes scalar code.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_convD

Performs either correlation or convolution on two vectors; double precision.

```
void vDSP_convD (
   const double __vDSP_signal[],
   vDSP_Stride __vDSP_signalStride,
   const double __vDSP_filter[],
   vDSP_Stride __vDSP_strideFilter,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_lenResult,
   vDSP_Length __vDSP_lenFilter
);
```

Discussion

$$C_{nK} = \sum_{p=0}^{P-1} A_{(n+p)I} B_{pJ} \qquad \text{n = \{0, N-1\}}$$

If `filterStride` is positive, `vDSP_convD` performs correlation. If `filterStride` is negative,it performs convolution and `*filter`must point to the last vector element. The function can run in place, but result cannot be in place with filter.

The value of `lenFilter` must be less than or equal to 2044. The length of vector `signal` must satisfy two criteria: it must be

- equal to or greater than 12
- equal to or greater than the sum of `N-1` plus the nearest multiple of 4 that is equal to or greater than the value of `lenFilter`.

Criteria to invoke vectorized code:

No Altivec support for double precision. On a PowerPC processor, this function always invokes scalar code.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_create_fftsetup

Builds a data structure that contains precalculated data for use by single-precision FFT functions.

```
FFTSetup vDSP_create_fftsetup (
   vDSP_Length __vDSP_log2n,
   FFTRadix __vDSP_radix
);
```

Parameters

log2n

A base 2 exponent that represents the number of divisions of the complex unit circle and thus specifies the largest power of two that can be processed by a subsequent frequency-domain function. Parameter `log2n` must equal or exceed the largest power of 2 that any subsequent function processes using the weights array.

radix

Specifies radix options. Radix 2, radix 3, and radix 5 functions are supported.

Return Value

Returns an `FFTSetup` (page 550) structure for use with one-dimensional FFT functions. Returns 0 on error.

Discussion

This function returns a filled-in `FFTSetup` (page 550) data structure for use by FFT functions that operate on double-precision vectors. Once prepared, the setup structure can be used repeatedly by FFT functions (which read the data in the structure and do not alter it) for any (power of two) length up to that specified when you created the structure.

If an application performs FFTs with diverse lengths, the calls with different lengths can share a single setup structure (created for the longest length), and this saves space over having multiple structures. However, in some cases, notably single-precision FFTs on 32-bit PowerPC, an FFT routine may perform faster if it is passed a setup structure that was created specifically for the length of the transform.

Parameter log2n is a base-two exponent and specifies that the largest transform length that can processed using the resulting setup structure is `2**log2n` (or `3*2**log2n` or `5*2**log2n` if the appropriate flags are passed, as discussed below). That is, the `log2n` parameter must equal or exceed the value passed to any subsequent FFT routine using the setup structure returned by this routine.

Parameter radix specifies radix options. Its value may be the bitwise OR of any combination of `kFFTRadix2` (page 552), `kFFTRadix3` (page 552), or `kFFTRadix5` (page 553). The resulting setup structure may be used with any of the routines for which the respective flag was used. (The radix-3 and radix-5 FFT routines have "fft3" and "fft5" in their names. The radix-2 FFT routines have plain "fft" in their names.)

If zero is returned, the routine failed to allocate storage.

The setup structure is deallocated by calling `vDSP_destroy_fftsetup` (page 201).

Use `vDSP_create_fftsetup` during initialization. It is relatively slow compared to the routines that actually perform FFTs. Never use it in a part of an application that needs to be high performance.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_create_fftsetupD

Builds a data structure that contains precalculated data for use by double-precision FFT functions.

```
FFTSetupD vDSP_create_fftsetupD (
   vDSP_Length __vDSP_log2n,
   FFTRadix __vDSP_radix
);
```

Parameters

log2n

A base 2 exponent that represents the number of divisions of the complex unit circle and thus specifies the largest power of two that can be processed by a subsequent frequency-domain function. Parameter `log2n` must equal or exceed the largest power of 2 that any subsequent function processes using the weights array.

radix

Specifies radix options. Radix 2, radix 3, and radix 5 functions are supported.

Return Value

Returns an `FFTSetupD` (page 550) structure for use with FFT functions, or 0 if there was an error.

Discussion

This function returns a filled-in `FFTSetupD` (page 550) data structure for use by FFT functions that operate on double-precision vectors. Once prepared, the setup structure can be used repeatedly by FFT functions (which read the data in the structure and do not alter it) for any (power of two) length up to that specified when you created the structure.

If an application performs FFTs with diverse lengths, the calls with different lengths can share a single setup structure (created for the longest length), and this saves space over having multiple structures. However, in some cases, notably single-precision FFTs on 32-bit PowerPC, an FFT routine may perform faster if it is passed a setup structure that was created specifically for the length of the transform.

Parameter log2n is a base-two exponent and specifies that the largest transform length that can processed using the resulting setup structure is `2**log2n` (or `3*2**log2n` or `5*2**log2n` if the appropriate flags are passed, as discussed below). That is, the `log2n` parameter must equal or exceed the value passed to any subsequent FFT routine using the setup structure returned by this routine.

Parameter radix specifies radix options. Its value may be the bitwise OR of any combination of `kFFTRadix2` (page 552), `kFFTRadix3` (page 552), or `kFFTRadix5` (page 553). The resulting setup structure may be used with any of the routines for which the respective flag was used. (The radix-3 and radix-5 FFT routines have `fft3` and `fft5` in their names. The radix-2 FFT routines have plain `fft` in their names.)

If zero is returned, the routine failed to allocate storage.

The setup structure is deallocated by calling `vDSP_destroy_fftsetupD` (page 202).

Use `vDSP_create_fftsetupD` during initialization. It is relatively slow compared to the routines that actually perform FFTs. Never use it in a part of an application that needs to be high performance.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_ctoz

Copies the contents of an interleaved complex vector C to a split complex vector Z; single precision.

```
void vDSP_ctoz (
   const DSPComplex __vDSP_C[],
   vDSP_Stride __vDSP_strideC,
   DSPSplitComplex *__vDSP_Z,
   vDSP_Stride __vDSP_strideZ,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$A_{nI} = Re(C_{nK}) \quad ; \quad A_{nK+1} = Im(C_{n;K}) \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

The `strideC` parameter contains an address stride through C. `strideZ` is an address stride through Z. The value of `strideC` must be a multiple of 2.

For best performance, `C.realp`, `C.imagp`, `Z.realp`, and `Z.imagp` should be 16-byte aligned.

See also functionsvDSP_ztoc (page 521) and vDSP_ztocD (page 522).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_ctozD

Copies the contents of an interleaved complex vector C to a split complex vector Z; double precision.

```
void vDSP_ctozD (
   const DSPDoubleComplex __vDSP_C[],
   vDSP_Stride __vDSP_strideC,
   DSPDoubleSplitComplex *__vDSP_Z,
   vDSP_Stride __vDSP_strideZ,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$A_{nI} = Re(C_{nK}) \quad ; \quad A_{nK+1} = Im(C_{n;K}) \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

The `strideC` parameter contains an address stride through C. `strideZ` is an address stride through Z. The value of `strideC` must be a multiple of 2.

For best performance, `C.realp`, `C.imagp`, `Z.realp`, and `Z.imagp` should be 16-byte aligned.

See also functions vDSP_ztoc (page 521) and vDSP_ztocD (page 522).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_deq22

Difference equation, 2 poles, 2 zeros; single precision.

```
void vDSP_deq22 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector; must have at least `N`+2 elements

I
Stride for `A`

B
5 single-precision inputs, filter coefficients

C
Single-precision real output vector; must have at least `N`+2 elements

K
Stride for `C`

N
Number of new output elements to produce

Discussion
Performs two-pole two-zero recursive filtering on real input vector `A`. Since the computation is recursive, the first two elements in vector `C` must be initialized prior to calling `vDSP_deq22`. `vDSP_deq22` creates `N` new values for vector `C` beginning with its third element and requires at least `N`+2 input values from vector `A`. This function can only be done out of place.

$$C_{nk} = \sum_{p=0}^{2} A_{(n-p)i} B_p - \sum_{p=3}^{4} C_{(n-p+2)k} B_p \qquad \text{n} = \{2, \text{N}+1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_deq22D

Difference equation, 2 poles, 2 zeros; double precision.

```
void vDSP_deq22D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector; must have at least `N`+2 elements

I

Stride for `A`

B

5 double-precision inputs, filter coefficients

C

Double-precision real output vector; must have at least `N`+2 elements

K

Stride for `C`

N

Number of new output elements to produce

Discussion

Performs two-pole two-zero recursive filtering on real input vector `A`. Since the computation is recursive, the first two elements in vector `C` must be initialized prior to calling `vDSP_deq22D`. `vDSP_deq22D` creates `N` new values for vector `C` beginning with its third element and requires at least `N`+2 input values from vector `A`. This function can only be done out of place.

$$C_{nk} = \sum_{p=0}^{2} A_{(n-p)i} B_p - \sum_{p=3}^{4} C_{(n-p+2)k} B_p \qquad \mathrm{n} = \{2, \mathrm{N}+1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_desamp

Convolution with decimation; single precision.

```
void vDSP_desamp (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Single-precision real input vector, 8-byte aligned; length of A >= 12

I
: Desampling factor

B
: Single-precision input filter coefficients

C
: Single-precision real output vector

N
: Output count

M
: Filter coefficient count

Discussion

Performs finite impulse response (FIR) filtering at selected positions of vector A. desampx can run in place, but C cannot be in place with B. Length of A must be >=(N-1)*I+(nearest multiple of 4 >=M).

$$C_n = \sum_{p=0}^{P-1} A_{nI+p} B_p \qquad \text{n} = \{0, \text{N-1}\}$$

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_desampD

Convolution with decimation; double precision.

```
void vDSP_desampD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Double-precision real input vector, 8-byte aligned; length of A >= 12

I
: Desampling factor

B
: Double-precision input filter coefficients

C
: Double-precision real output vector

N
: Output count

M
: Filter coefficient count

Discussion

Performs finite impulse response (FIR) filtering at selected positions of vector A. desampx can run in place, but C cannot be in place with B. Length of A must be >=(N-1)*I+(nearest multiple of 4 >=M).

$$C_n = \sum_{p=0}^{P-1} A_{nI+p} B_p \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_destroy_fftsetup

Frees an existing single-precision FFT data structure.

```
void vDSP_destroy_fftsetup (
   FFTSetup __vDSP_setup
);
```

Parameters

setup
: Identifies the weights array, and must point to a data structure previously created by vDSP_create_fftsetup (page 195).

Discussion

`vDSP_destroy_fftsetup` frees an existing weights array. Any memory allocated for the array is released. After the `vDSP_destroy_fftsetup` function returns, the structure should not be used in any other functions.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_destroy_fftsetupD

Frees an existing double-precision FFT data structure.

```
void vDSP_destroy_fftsetupD (
   FFTSetupD __vDSP_setup
);
```

Parameters

setup

Identifies the weights array, and must point to a data structure previously created by `vDSP_create_fftsetupD` (page 196).

Discussion

`vDSP_destroy_fftsetupD` frees an existing weights array. Any memory allocated for the array is released. After the `vDSP_destroy_fftsetupD` function returns, the structure should not be used in any other functions.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_DFT_CreateSetup

Old name for `vDSP_DFT_zop_CreateSetup` (page 205).

```
vDSP_DFT_Setup vDSP_DFT_CreateSetup (
   vDSP_DFT_Setup __vDSP_Previous,
   vDSP_Length __vDSP_Length
);
```

Discussion

Call `vDSP_DFT_zop_CreateSetup` (page 205) instead.

Concurrency Note: This function is not thread-safe. You must not call this function concurrently with any other DFT call.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_DFT_DestroySetup

Releases a setup object.

```
void vDSP_DFT_DestroySetup (
   vDSP_DFT_Setup __vDSP_Setup
);
```

Parameters

__vDSP_Setup

The setup object to destroy (as returned by `vDSP_DFT_CreateSetup` (page 202), `vDSP_DFT_zop_CreateSetup` (page 205), or `vDSP_DFT_zrop_CreateSetup` (page 206).

Discussion

If this setup object shares memory with other setup objects, that memory is not released until the last object is destroyed.

Concurrency Note: This function is not thread-safe. You must not call this function concurrently with any other DFT call.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_DFT_Execute

Calculates the discrete fourier transform for a vector using `vDSP_DFT_zop` (page 204) or `vDSP_DFT_zrop` (page 206).

```
void vDSP_DFT_Execute (
    const struct vDSP_DFT_SetupStruct *__vDSP_Setup,
    const float *__vDSP_Ir,
    const float *__vDSP_Ii,
    float *__vDSP_Or,
    float *__vDSP_Oi
);
```

Parameters

__vDSP_Setup

A DFT setup object returned by a call to `vDSP_DFT_zop_CreateSetup` (page 205).

__vDSP_Ir

A vector containing the real part of the input values.

__vDSP_Ii

A vector containing the imaginary part of the input values.

__vDSP_Or

A vector where the real part of the results is stored on return.

__vDSP_Oi
A vector where the imaginary part of the results is stored on return.

Discussion
This function calculates either a real or complex discrete fourier transform, depending on whether the setup object was created with a call to `vDSP_DFT_zrop_CreateSetup` (page 206) or `vDSP_DFT_zop_CreateSetup` (page 205).

Note: If the setup was created with `vDSP_DFT_zop_CreateSetup` (page 205), each vector should contain `length` elements. If it was created with `vDSP_DFT_zrop_CreateSetup` (page 206), each vector should contain `length/2` elements.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_DFT_zop

Calculates the discrete fourier transform for a vector (single-precision complex).

```
void vDSP_DFT_zop (
   const struct vDSP_DFT_SetupStruct *__vDSP_Setup,
   const float *__vDSP_Ir,
   const float *__vDSP_Ii,
   vDSP_Stride __vDSP_Is,
   float *__vDSP_Or,
   float *__vDSP_Oi,
   vDSP_Stride __vDSP_Os,
   vDSP_DFT_Direction __vDSP_Direction
);
```

Parameters

__vDSP_Setup
A DFT setup object returned by a call to `vDSP_DFT_zop_CreateSetup` (page 205).

__vDSP_InputRe
A vector containing the real part of the input values.

__vDSP_InputIm
A vector containing the imaginary part of the input values.

__vDSP_InputStride
Stride length within the input vector. For example, passing 2 would cause every second value to be used.

__vDSP_OutputRe
A vector where the real part of the results is stored on return.

__vDSP_OutputIm
A vector where the imaginary part of the results is stored on return.

__vDSP_OutputStride
Stride length within the input vector. For example, passing 2 would cause every second value to be overwritten.

__vDSP_Direction
: `vDSP_DFT_FORWARD` (page 551) or `vDSP_DFT_INVERSE` (page 552), indicating whether to perform a forward or inverse transform.

Discussion
The input and output vectors may not overlap, with one exception: if the real and imaginary input vectors are both the same as the corresponding output vector, an in-place algorithm is used.

Note: The length of the transform is specified in the setup object.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_DFT_zop_CreateSetup

Creates data structures for use with `vDSP_DFT_zop` (page 204).

```
vDSP_DFT_Setup vDSP_DFT_zop_CreateSetup (
   vDSP_DFT_Setup __vDSP_Previous,
   vDSP_Length __vDSP_Length,
   vDSP_DFT_Direction __vDSP_Direction
);
```

Parameters

__vDSP_Previous
: An previous result from this function or `NULL`.

__vDSP_Length
: The length of DFT computations with this object.

Return Value
Returns a DFT setup object.

Discussion
This function is designed to share memory between data structures where possible. You should initially create a setup object with the largest length you expect to use, then pass that object as `__vDSP_Previous`. By doing so, this setup object can share the underlying data storage with that object, thus reducing the time needed to construct the tables.

Concurrency Note: This function is not thread-safe. You must not call this function concurrently with any other DFT call.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_DFT_zrop

Calculates the discrete fourier transform for a vector (single-precision real).

```
void vDSP_DFT_zrop (
    vDSP_DFT_Setup __vDSP_Setup,
    const float *__vDSP_InputRe,
    const float *__vDSP_InputIm,
    vDSP_Stride __vDSP_InputStride,
    float *__vDSP_OutputRe,
    float *__vDSP_OutputIm,
    vDSP_Stride __vDSP_OutputStride,
    vDSP_DFT_Direction __vDSP_Direction
);
```

Parameters

__vDSP_Setup
: A DFT setup object returned by a call to `vDSP_DFT_zrop_CreateSetup` (page 206).

__vDSP_InputRe
: A vector containing the real part of the input values.

__vDSP_InputIm
: A vector containing the imaginary part of the input values.

__vDSP_InputStride
: Stride length within the input vector. For example, passing 2 would cause every second value to be used.

__vDSP_OutputRe
: A vector where the real part of the results is stored on return.

__vDSP_OutputIm
: A vector where the imaginary part of the results is stored on return.

__vDSP_OutputStride
: Stride length within the input vector. For example, passing 2 would cause every second value to be overwritten.

__vDSP_Direction
: `vDSP_DFT_FORWARD` (page 551) or `vDSP_DFT_INVERSE` (page 552), indicating whether to perform a forward or inverse transform.

Discussion

The input and output vectors may not overlap unless they are equal, in which case an in-place algorithm is used.

Note: The length of the transform is specified in the setup object.

vDSP_DFT_zrop_CreateSetup

Creates data structures for use with `vDSP_DFT_zrop` (page 206).

```
vDSP_DFT_Setup vDSP_DFT_zrop_CreateSetup (
   vDSP_DFT_Setup __vDSP_Previous,
   vDSP_Length __vDSP_Length,
   vDSP_DFT_Direction __vDSP_Direction
);
```

Parameters

__vDSP_Previous
: An previous result from this function or `NULL`.

__vDSP_Length
: The length of DFT computations with this object.

Return Value

Returns a DFT setup object.

Discussion

This function is designed to share memory between data structures where possible. You should initially create a setup object with the largest length you expect to use, then pass that object as `__vDSP_Previous`. By doing so, this setup object can share the underlying data storage with that object, thus reducing the time needed to construct the tables.

Concurrency Note: This function is not thread-safe. You must not call this function concurrently with any other DFT call.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_dotpr

Computes the dot or scalar product of vectors `A` and `B` and leaves the result in scalar `*C`; single precision.

```
void vDSP_dotpr (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   float *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Parameters

__vDSP_input1
: Input vector `A`.

__vDSP_stride1
: The stride within vector `A`. For example if stride is 2, every second element is used.

__vDSP_input2
: Input vector `B`.

__vDSP_stride2
: The stride within vector `B`. For example if stride is 2, every second element is used.

__vDSP_result
: The dot product (on return).

__vDSP_size
: The number of elements (N).

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_dotpr2

Stereo variant of `vDSP_dotpr` (page 207).

```
void vDSP_dotpr2 (
   const float *__vDSP_A0,
   vDSP_Stride __vDSP_A0Stride,
   const float *__vDSP_A1,
   vDSP_Stride __vDSP_A1Stride,
   const float *__vDSP_B,
   vDSP_Stride __vDSP_BStride,
   float *__vDSP_C0,
   float *__vDSP_C1,
   vDSP_Length __vDSP_Length
);
```

Parameters

__vDSP_A0
: Input vector A0.

__vDSP_A0Stride
: The stride within vector A0. For example if stride is 2, every second element is used.

__vDSP_A1
: Input vector A1.

__vDSP_A1Stride
: The stride within vector A1. For example if stride is 2, every second element is used.

__vDSP_B
: Input vector B.

__vDSP_BStride
: The stride within vector B. For example if stride is 2, every second element is used.

__vDSP_C0
: The dot product of A0 and B (on return).

__vDSP_C1
: The dot product of A1 and B (on return).

__vDSP_Length
: The number of elements.

Discussion

Calculates the dot product of A0 with B, then calculates the dot product of A1 with B. The equation for a single dot product is:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_dotpr2_s1_15

Vector fixed-point 1.15 format version of `vDSP_dotpr2` (page 208).

```
void vDSP_dotpr2_s1_15 (
   const short int *__vDSP_A0,
   vDSP_Stride __vDSP_A0Stride,
   const short int *__vDSP_A1,
   vDSP_Stride __vDSP_A1Stride,
   const short int *__vDSP_B,
   vDSP_Stride __vDSP_BStride,
   short int *__vDSP_C0,
   short int *__vDSP_C1,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_A0
: Input vector `A0`.

__vDSP_A0Stride
: The stride within vector `A0`. For example if stride is 2, every second element is used.

__vDSP_A1
: Input vector `A1`.

__vDSP_A1Stride
: The stride within vector `A1`. For example if stride is 2, every second element is used.

__vDSP_B
: Input vector `B`.

__vDSP_BStride
: The stride within vector `B`. For example if stride is 2, every second element is used.

__vDSP_C0
: The dot product of A0 and B (on return).

__vDSP_C1
The dot product of A1 and B (on return).

__vDSP_Length
The number of elements.

Discussion
Calculates the dot product of A0 with B, then calculates the dot product of A1 with B. The equation for a single dot product is:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad \text{n = \{0, N-1\}}$$

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by `32768.0`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_dotpr2_s8_24

Vector fixed-point 8.24 format version of `vDSP_dotpr2` (page 208).

```
void vDSP_dotpr2_s8_24 (
   const int *__vDSP_A0,
   vDSP_Stride __vDSP_A0Stride,
   const int *__vDSP_A1,
   vDSP_Stride __vDSP_A1Stride,
   const int *__vDSP_B,
   vDSP_Stride __vDSP_BStride,
   int *__vDSP_C0,
   int *__vDSP_C1,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_A0
Input vector `A0`.

__vDSP_A0Stride
The stride within vector `A0`. For example if stride is 2, every second element is used.

__vDSP_A1
Input vector `A1`.

__vDSP_A1Stride
The stride within vector `A1`. For example if stride is 2, every second element is used.

__vDSP_B
Input vector `B`.

__vDSP_BStride
The stride within vector `B`. For example if stride is 2, every second element is used.

`__vDSP_C0`
: The dot product of A0 and B (on return).

`__vDSP_C1`
: The dot product of A1 and B (on return).

`__vDSP_Length`
: The number of elements.

Discussion
Calculates the dot product of A0 with B, then calculates the dot product of A1 with B. The equation for a single dot product is:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad \text{n = \{0, N-1\}}$$

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by `16777216.0`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_dotprD

Computes the dot or scalar product of vectors `A` and `B` and leaves the result in scalar `*C`; double precision.

```
void vDSP_dotprD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   double *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad \text{n = \{0, N-1\}}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_dotpr_s1_15

Vector fixed-point 1.15 format version of vDSP_dotpr (page 207).

```
void vDSP_dotpr_s1_15 (
   const short int *__vDSP_A,
   vDSP_Stride __vDSP_AStride,
   const short int *__vDSP_B,
   vDSP_Stride __vDSP_BStride,
   short int *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_A
Input vector A.

__vDSP_AStride
The stride within vector A. For example if stride is 2, every second element is used.

__vDSP_B
Input vector B.

__vDSP_BStride
The stride within vector B. For example if stride is 2, every second element is used.

__vDSP_C
The dot product (on return).

__vDSP_N
The number of elements.

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad n = \{0, N\text{-}1\}$$

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by 32768.0.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_dotpr_s8_24

Vector fixed-point 8.24 format version of vDSP_dotpr (page 207).

```
void vDSP_dotpr_s8_24 (
   const int *__vDSP_A,
   vDSP_Stride __vDSP_AStride,
   const int *__vDSP_B,
   vDSP_Stride __vDSP_BStride,
   int *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_A
Input vector A.

__vDSP_AStride
The stride within vector A. For example if stride is 2, every second element is used.

__vDSP_B
Input vector B.

__vDSP_BStride
The stride within vector B. For example if stride is 2, every second element is used.

__vDSP_C
The dot product (on return).

__vDSP_N
The number of elements.

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot B_{nJ} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by 16777216.0.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_f3x3

Filters an image by performing a two-dimensional convolution with a 3x3 kernel; single precision.

```
void vDSP_f3x3 (
   float *__vDSP_signal,
   vDSP_Length __vDSP_rows,
   vDSP_Length __vDSP_cols,
   float *__vDSP_filter,
   float *__vDSP_result
);
```

Discussion
This function filters an image by performing a two-dimensional convolution with a 3x3 kernel (`B`) on the input matrix `A` and storing the resulting image in the output matrix `C`.

This performs the following operation:

$$C_{(m+1,n+1)} = \sum_{p=0}^{2} \sum_{q=0}^{2} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad m = \{0, M-1\} \text{ and } n = \{0, N-3\}$$

The function pads the perimeter of the output image with a border of zeros of width 1.

`B` is the 3x3 kernel. `M` and `N` are the number of rows and columns, respectively, of the two-dimensional input matrix `A`. `M` must be greater than or equal to 3. `N` must be even and greater than or equal to 4.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_f3x3D

Filters an image by performing a two-dimensional convolution with a 3x3 kernel; double precision.

```
void vDSP_f3x3D (
   double *__vDSP_signal,
   vDSP_Length __vDSP_rows,
   vDSP_Length __vDSP_cols,
   double *__vDSP_filter,
   double *__vDSP_result
);
```

Discussion
This function filters an image by performing a two-dimensional convolution with a 3x3 kernel on the input matrix `A` and storing the resulting image in the output matrix `C`.

This performs the following operation:

$$C_{(m+1,n+1)} = \sum_{p=0}^{2} \sum_{q=0}^{2} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad m = \{0, M-1\} \text{ and } n = \{0, N-3\}$$

The function pads the perimeter of the output image with a border of zeros of width 1.

B is the 3x3 kernel. M and N are the number of rows and columns, respectively, of the two-dimensional input matrix A. M must be greater than or equal to 3. N must be even and greater than or equal to 4.

Criteria to invoke vectorized code:

- A, B, and C must be 16-byte aligned.
- N must be greater than or equal to 18.

If any of these criteria is not satisfied, the function invokes scalar code.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_f5x5

Filters an image by performing a two-dimensional convolution with a 5x5 kernel; single precision.

```
void vDSP_f5x5 (
   float *__vDSP_signal,
   vDSP_Length __vDSP_rows,
   vDSP_Length __vDSP_cols,
   float *__vDSP_filter,
   float *__vDSP_result
);
```

Discussion
This function filters an image by performing a two-dimensional convolution with a 5x5 kernel (B) on the input matrix A and storing the resulting image in the output matrix C.

This performs the following operation:

$$C_{(m+2,n+2)} = \sum_{p=0}^{4} \sum_{q=0}^{4} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad \text{m = \{0, M-5\} and n = \{0, N-5\}}$$

The function pads the perimeter of the output image with a border of zeros of width 2.

B is the 5x5 kernel. M and N are the number of rows and columns, respectively, of the two-dimensional input matrix A. M must be greater than or equal to 5. N must be even and greater than or equal to 6.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_f5x5D

Filters an image by performing a two-dimensional convolution with a 5x5 kernel; double precision.

```
void vDSP_f5x5D (
   double *__vDSP_signal,
   vDSP_Length __vDSP_rows,
   vDSP_Length __vDSP_cols,
   double *__vDSP_filter,
   double *__vDSP_result
);
```

Discussion
This function filters an image by performing a two-dimensional convolution with a 5x5 kernel (`B`) on the input matrix `A` and storing the resulting image in the output matrix `C`.

This performs the following operation:

$$C_{(m+2,n+2)} = \sum_{p=0}^{4} \sum_{q=0}^{4} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad \mathrm{m} = \{0, \mathrm{M}\text{-}5\} \text{ and } \mathrm{n} = \{0, \mathrm{N}\text{-}5\}$$

The function pads the perimeter of the output image with a border of zeros of width 2.

`B` is the 5x5 kernel. `M` and `N` are the number of rows and columns, respectively, of the two-dimensional input matrix `A`. `M` must be greater than or equal to 5. `N` must be even and greater than or equal to 6.

Criteria to invoke vectorized code:

- `A`, `B`, and `C` must be 16-byte aligned.
- `N` must be greater than or equal to 20.

If any of these criteria is not satisfied, the function invokes scalar code.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_FFT16_copv

Performs a 16-element FFT on interleaved complex data.

```
void vDSP_FFT16_copv (
   float *__vDSP_Output,
   const float *__vDSP_Input,
   FFTDirection __vDSP_Direction
);
```

Parameters

__vDSP_Output
    A vector-block-aligned output array.

__vDSP_Input
    A vector-block-aligned input array.

__vDSP_Direction
: kFFTDirection_Forward (page 552) or kFFTDirection_Inverse (page 552), indicating whether to perform a forward or inverse transform.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_FFT16_zopv

Performs a 16-element FFT on split complex data.

```
void vDSP_FFT16_zopv (
   float *__vDSP_Or,
   float *__vDSP_Oi,
   const float *__vDSP_Ir,
   const float *__vDSP_Ii,
   FFTDirection __vDSP_Direction
);
```

Parameters

__vDSP_Or
: Output vector for real parts.

__vDSP_Oi
: Output vector for imaginary parts.

__vDSP_Ir
: Input vector for real parts.

__vDSP_Ii
: Input vector for imaginary parts.

__vDSP_Direction
: kFFTDirection_Forward (page 552) or kFFTDirection_Inverse (page 552), indicating whether to perform a forward or inverse transform.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zip

Computes an in-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

ioData

A complex vector input.

strideInRow

Specifies a stride across each row of the matrix `signal`. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, parameter `strideInCol` represents the distance between each row of the matrix. If parameter `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for parameter `log2nInCol` and 6 for parameter `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

direction

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

This performs the following operation:

If F = 1 $C_{nm} = \mathrm{FDFT2D}(C_{nm})$ n = {0, N-1} and m = {0, M-1}

If F = -1 $C_{nm} = \mathrm{IDFT2D}(C_{nm}) \bullet MN$ n = {0, N-1} and m = {0, M-1}

$$\mathrm{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(-j2\pi pn)/N} \bullet e^{(-j2\pi qm)/M}$$

$$\mathrm{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(j2\pi pn)/N} \bullet e^{(j2\pi qm)/M}$$

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zipD

Computes an in-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zipD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

ioData

A complex vector input.

strideInRow

Specifies a stride across each row of the matrix `signal`. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, parameter `strideInCol` represents the distance between each row of the matrix. If parameter `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for parameter `log2nInCol` and 6 for parameter `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

direction

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_{nm} = \text{FDFT2D}(C_{nm}) \quad \text{n} = \{0, \text{N-1}\} \text{ and m} = \{0, \text{M-1}\}$$

$$\text{If F = -1} \quad C_{nm} = \text{IDFT2D}(C_{nm}) \bullet MN \quad \text{n} = \{0, \text{N-1}\} \text{ and m} = \{0, \text{M-1}\}$$

$$\text{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(-j2\pi pn)/N} \bullet e^{(-j2\pi qm)/M}$$

$$\text{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(j2\pi pn)/N} \bullet e^{(j2\pi qm)/M}$$

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zipt

Computes an in-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zipt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

ioData

A complex vector input.

strideInRow

Specifies a stride across each row of the matrix `signal`. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, parameter `strideInCol` represents the distance between each row of the matrix. If parameter `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

bufferTemp

A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 16 KB or 4*n, where `log2n = log2nInCol + log2nInRow`.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for parameter `log2nInCol` and 6 for parameter `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

direction

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

This performs the following operation:

If F = 1 $\quad C_{nm} = \mathrm{FDFT2D}(C_{nm}) \quad$ n = {0, N-1} and m = {0, M-1}

If F = -1 $\quad C_{nm} = \mathrm{IDFT2D}(C_{nm}) \bullet MN \quad$ n = {0, N-1} and m = {0, M-1}

$$\mathrm{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(-j2\pi pn)/N} \bullet e^{(-j2\pi qm)/M}$$

$$\mathrm{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(j2\pi pn)/N} \bullet e^{(j2\pi qm)/M}$$

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_ziptD

Computes an in-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_ziptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

`setup`

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

`ioData`

A complex vector input.

`strideInRow`

Specifies a stride across each row of the matrix `signal`. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

`strideInCol`

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, parameter `strideInCol` represents the distance between each row of the matrix. If parameter `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

`bufferTemp`

A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 16 KB or 4*`n`, where `log2n = log2nInCol + log2nInRow`.

`log2nInCol`

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

`log2nInRow`

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for parameter `log2nInCol` and 6 for parameter `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

`direction`

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_{nm} = \text{FDFT2D}(C_{nm}) \quad \text{n = \{0, N-1\} and m = \{0, M-1\}}$$

$$\text{If F = -1} \quad C_{nm} = \text{IDFT2D}(C_{nm}) \bullet MN \quad \text{n = \{0, N-1\} and m = \{0, M-1\}}$$

$$\text{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(-j2\pi pn)/N} \bullet e^{(-j2\pi qm)/M}$$

$$\text{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \bullet e^{(j2\pi pn)/N} \bullet e^{(j2\pi qm)/M}$$

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zop

Computes an out-of-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

signal
: A complex vector signal input.

signalStrideInRow
: Specifies a stride across each row of matrix `a`. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

signalStrideInCol
: If not 0, this parameter represents the distance between each row of the input /output matrix.

result
: The complex vector signal output.

strideResultInRow
: Specifies a row stride for output matrix `result` in the same way that `signalStrideInRow` specifies a stride for input the input /output matrix.

strideResultInCol
: Specifies a column stride for output matrix `result` in the same way that `signalStrideInCol` specifies a stride for input the input /output matrix.

log2nInCol
: The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

flag
: A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

: Results are undefined for other values of `flag`.

Discussion
This performs the following operation:

$$\text{If F = 1} \qquad C_m = \text{FDFT}(A_m) \qquad \text{If F = -1} \qquad C_m = \text{IDFT}\ (A_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zopD

Computes an out-of-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zopD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

`signal`
A complex vector signal input.

`signalStrideInRow`
Specifies a stride across each row of matrix `a`. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

`signalStrideInCol`
If not 0, this parameter represents the distance between each row of the input /output matrix.

`result`
The complex vector signal output.

strideResultInRow
: Specifies a row stride for output matrix `result` in the same way that `signalStrideInRow` specifies a stride for input the input /output matrix.

strideResultInCol
: Specifies a column stride for output matrix `result` in the same way that `signalStrideInCol` specifies a stride for input the input /output matrix.

log2nInCol
: The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

flag
: A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

  Results are undefined for other values of `flag`.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{FDFT}(A_m) \qquad \text{If F = -1} \quad C_m = \text{IDFT}\ (A_m) \bullet N \qquad \text{m = \{0, N-1\}}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zopt

Computes an out-of-place single-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zopt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

signal
: A complex vector signal input.

signalStrideInRow
: Specifies a stride across each row of matrix `a`. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

signalStrideInCol
: If not 0, this parameter represents the distance between each row of the input /output matrix. If parameter signalStrideInCol is 1024, for instance, element 512 equates to element (1,0) of matrix `a`, element 1024 equates to element (2,0), and so forth.

result
: The complex vector signal output.

strideResultInRow
: Specifies a row stride for output matrix `result` in the same way that `signalStrideInRow` specifies a stride for input the input /output matrix.

strideResultInCol
: Specifies a column stride for output matrix `result` in the same way that `signalStrideInCol` specifies a stride for input the input /output matrix.

bufferTemp
: A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 16 KiB or 4*`n`, where `log2n = log2nInCol + log2nInRow`.

log2nInCol
: The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

flag

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `flag`.

Discussion

This performs the following operation:

$$\text{If } F = 1 \quad C_m = \text{FDFT}(A_m) \qquad \text{If } F = -1 \quad C_m = \text{IDFT}\,(A_m) \cdot N \qquad m = \{0, N\text{-}1\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zoptD

Computes an out-of-place double-precision complex discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zoptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters `log2nInCol` and `log2nInRow` of the transform function.

signal

A complex vector signal input.

signalStrideInRow

Specifies a stride across each row of matrix `a`. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

signalStrideInCol

If not 0, this parameter represents the distance between each row of the input /output matrix. If parameter signalStrideInCol is 1024, for instance, element 512 equates to element (1,0) of matrix `a`, element 1024 equates to element (2,0), and so forth.

result

The complex vector signal output.

strideResultInRow

Specifies a row stride for output matrix `result` in the same way that `signalStrideInRow` specifies a stride for input the input /output matrix.

strideResultInCol

Specifies a column stride for output matrix `result` in the same way that `signalStrideInCol` specifies a stride for input the input /output matrix.

bufferTemp

A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 16 KB or 4*`n`, where `log2n = log2nInCol + log2nInRow`.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

flag

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `flag`.

Discussion

This performs the following operation:

$$\text{If F} = 1 \qquad C_m = \text{FDFT}(A_m) \qquad \text{If F} = \text{-1} \qquad C_m = \text{IDFT}\ (A_m) \cdot N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N} \sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zrip

Computes an in-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zrip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters log2nInCol and `log2nInRow` of the transform function.

ioData

A complex vector input.

strideInRow

Specifies a stride across each row of the input matrix signal. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, `strideInCol` represents the distance between each row of the matrix. If `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

direction

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

If F = 1 $C_{nm} = \text{FDFT2D}(C_{nm}) \cdot 2$ n = {0, N-1} and m = {0, M-1}

If F = -1 $C_{nm} = \text{IDFT2D}(C_{nm}) \cdot MN$ n = {0, N-1} and m = {0, M-1}

$$\text{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \cdot e^{(-j2\pi pn)/N} \cdot e^{(-j2\pi qm)/M}$$

$$\text{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \cdot e^{(j2\pi pn)/N} \cdot e^{(j2\pi qm)/M}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zripD

Computes an in-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zripD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters log2nInCol and `log2nInRow` of the transform function.

signal
A complex vector signal input.

strideInRow
Specifies a stride across each row of the input matrix signal. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

`strideInCol`
: Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

  If not 0, `strideInCol` represents the distance between each row of the matrix. If `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

`log2nInCol`
: The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 2 and 10, inclusive.

`log2nInRow`
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 2 and 10, inclusive.

`flag`
: A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

  Results are undefined for other values of `flag`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

$$\text{If F = 1} \qquad C_{nm} = \text{FDFT2D}(C_{nm}) \cdot 2 \qquad \text{n = \{0, N-1\} and m = \{0, M-1\}}$$

$$\text{If F = -1} \qquad C_{nm} = \text{IDFT2D}(C_{nm}) \cdot MN \qquad \text{n = \{0, N-1\} and m = \{0, M-1\}}$$

$$\text{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \cdot e^{(-j2\pi pn)/N} \cdot e^{(-j2\pi qm)/M}$$

$$\text{IDFT2D}(X_{nm}) = \frac{1}{MN} \sum_{p=0}^{N-1} \sum_{q=0}^{M-1} X_{pq} \cdot e^{(j2\pi pn)/N} \cdot e^{(j2\pi qm)/M}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

No Altivec/SSE support for double precision. The function always invokes scalar code.

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zript

Computes an in-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zript (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters log2nInCol and `log2nInRow` of the transform function.

ioData

A complex vector input.

strideInRow

Specifies a stride across each row of the input matrix signal. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol

Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

If not 0, `strideInCol` represents the distance between each row of the matrix. If `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

bufferTemp

A temporary matrix used for storing interim results. The size of temporary memory required is discussed below.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 3 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 3 and 10, inclusive.

`direction`

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `direction`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

If F = 1 $\quad C_{nm} = \mathrm{FDFT2D}(C_{nm}) \cdot 2 \quad$ n = {0, N-1} and m = {0, M-1}

If F = -1 $\quad C_{nm} = \mathrm{IDFT2D}(C_{nm}) \cdot MN \quad$ n = {0, N-1} and m = {0, M-1}

$$\mathrm{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1}\sum_{q=0}^{M-1} X_{pq} \cdot e^{(-j2\pi pn)/N} \cdot e^{(-j2\pi qm)/M}$$

$$\mathrm{IDFT2D}(X_{nm}) = \frac{1}{MN}\sum_{p=0}^{N-1}\sum_{q=0}^{M-1} X_{pq} \cdot e^{(j2\pi pn)/N} \cdot e^{(j2\pi qm)/M}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

The space needed in `bufferTemp` is at most max(9*nr, nc/2) elements in each of `realp` and `imagp`. Here is an example of how to allocate the space:

```
int nr, nc, tempSize;
nr = 1 << log2InRow;
nc = 1 << log2InCol;
tempSize = max(9*nr, nc/2);
bufferTemp.realp = (float *) malloc (tempSize * sizeOf(float) );
bufferTemp.imagp = (float *) malloc (tempSize * sizeOf(float) );
```

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zriptD

Computes an in-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zriptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_strideInRow,
   vDSP_Stride __vDSP_strideInCol,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the values supplied as parameters log2nInCol and `log2nInRow` of the transform function.

signal
: A complex vector signal input.

strideInRow
: Specifies a stride across each row of the input matrix signal. Specifying 1 for `strideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

strideInCol
: Specifies a column stride for the matrix, and should generally be allowed to default unless the matrix is a submatrix. Parameter `strideInCol` can be defaulted by specifying 0. The default column stride equals the row stride multiplied by the column count. Thus, if `strideInRow` is 1 and `strideInCol` is 0, every element of the input /output matrix is processed. If `strideInRow` is 2 and `strideInCol` is 0, every other element of each row is processed.

  If not 0, `strideInCol` represents the distance between each row of the matrix. If `strideInCol` is 1024, for instance, complex element 512 of the matrix equates to element (1,0), element 1024 equates to element (2,0), and so forth.

bufferTemp
: A temporary matrix used for storing interim results. The size of temporary memory required is discussed below.

log2nInCol
: The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 3 and 10, inclusive.

log2nInRow
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 3 and 10, inclusive.

flag
: A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

  Results are undefined for other values of `flag`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

If F = 1 $C_{nm} = \text{FDFT2D}(C_{nm}) \bullet 2$ n = {0, N-1} and m = {0, M-1}

If F = -1 $C_{nm} = \text{IDFT2D}(C_{nm}) \bullet MN$ n = {0, N-1} and m = {0, M-1}

$$\text{FDFT2D}(X_{nm}) = \sum_{p=0}^{N-1}\sum_{q=0}^{M-1} X_{pq} \bullet e^{(-j2\pi pn)/N} \bullet e^{(-j2\pi qm)/M}$$

$$\text{IDFT2D}(X_{nm}) = \frac{1}{MN}\sum_{p=0}^{N-1}\sum_{q=0}^{M-1} X_{pq} \bullet e^{(j2\pi pn)/N} \bullet e^{(j2\pi qm)/M}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

The space needed in `bufferTemp` is at most max(9*nr, nc/2) elements in each of `realp` and `imagp`. Here is an example of how to allocate the space:

```
int nr, nc, tempSize;
nr = 1 << log2InRow;
nc = 1 << log2InCol;
tempSize = max(9*nr, nc/2);
bufferTemp.realp = (float *) malloc (tempSize * sizeOf(float) );
bufferTemp.imagp = (float *) malloc (tempSize * sizeOf(float) );
```

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zrop

Computes an out-of-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zrop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, vDSP_create_fftsetup (page 195). The value supplied as parameter log2n of the setup function must equal or exceed the value supplied as parameter log2n or log2m, whichever is larger, of the transform function.

signal

A complex vector signal input.

signalStrideInRow

Specifies a stride across each row of matrix signal. Specifying 1 for signalStrideInRow processes every element across each row, specifying 2 processes every other element across each row, and so forth.

signalStrideInCol

If not 0, represents the distance between each row of the input /output matrix. If parameter signalStrideInCol is 1024, for instance, element 512 equates to element (1,0) of matrix a, element 1024 equates to element (2,0), and so forth.

result

The complex vector signal output.

strideResultInRow

Specifies a row stride for output matrix c in the same way that signalStrideInRow specifies strides for input the matrix.

strideResultInCol

Specifies a column stride for output matrix c in the same way that signalStrideInCol specify strides for input the matrix.

log2nInCol

The base 2 exponent of the number of columns to process for each row. log2nInCol must be between 3 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for log2nInCol and 6 for log2nInRow. log2nInRow must be between 3 and 10, inclusive.

flag

A forward/inverse directional flag, and must specify kFFTDirection_Forward (page 552) for a forward transform or kFFTDirection_Inverse (page 552) for an inverse transform.

Results are undefined for other values of flag.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetup` (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zropD

Computes an out-of-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse).

```
void vDSP_fft2d_zropD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_Kr,
   vDSP_Stride __vDSP_Kc,
   DSPDoubleSplitComplex *__vDSP_ioData2,
   vDSP_Stride __vDSP_Ir,
   vDSP_Stride __vDSP_Ic,
   vDSP_Length __vDSP_log2nc,
   vDSP_Length __vDSP_log2nr,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` or `log2m`, whichever is larger, of the transform function.

`ioData`

A complex vector input.

`Kr`

Specifies a stride across each row of matrix signal. Specifying 1 for `Kr` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

`Kc`

If not 0, represents the distance between each row of the input /output matrix. If parameter `Kc` is 1024, for instance, element 512 equates to element (1,0) of matrix `a`, element 1024 equates to element (2,0), and so forth.

`ioData2`

The complex vector result.

Ir
: Specifies a row stride for output matrix `ioData2` in the same way that `Kr` specifies strides for input the matrix.

Ic
: Specifies a column stride for output matrix `ioData2` in the same way that `Kc` specify strides for input matrix `ioData`.

log2nc
: The base 2 exponent of the number of columns to process for each row. `log2nc` must be between 3 and 10, inclusive.

log2nr
: The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nc` and 6 for `log2nr`. `log2nr` must be between 3 and 10, inclusive.

flag
: A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

  Results are undefined for other values of `flag`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft2d_zropt

Computes an out-of-place single-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zropt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStrideInRow,
   vDSP_Stride __vDSP_signalStrideInCol,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResultInRow,
   vDSP_Stride __vDSP_strideResultInCol,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2nInCol,
   vDSP_Length __vDSP_log2nInRow,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` or `log2m`, whichever is larger, of the transform function.

signal

A complex vector signal input.

signalStrideInRow

Specifies a stride across each row of matrix `signal`. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

signalStrideInCol

If not 0, represents the distance between each row of matrix `signal`. If parameter `signalStrideInCol` is 1024, for instance, element 512 equates to element (1,0) of matrix `signal`, element 1024 equates to element (2,0), and so forth.

result

The complex vector signal output.

strideResultInRow

Specifies a row stride for output matrix `result` in the same way that `signalStrideInRow` specifies strides for input matrix `result`.

strideResultInCol

Specifies a column stride for output matrix `c` in the same way that `signalStrideInCol` specify strides for input matrix `result`.

bufferTemp

A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) can be calculated using the algorithm shown below.

log2nInCol

The base 2 exponent of the number of columns to process for each row. `log2nInCol` must be between 3 and 10, inclusive.

log2nInRow

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nInCol` and 6 for `log2nInRow`. `log2nInRow` must be between 3 and 10, inclusive.

flag

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `flag`.

Discussion
Here is the `bufferTemp` size algorithm:

```
int nr, nc, tempSize;
nr = 1 << log2InRow;
nc = 1 << log2InCol;
if (( (log2InCol-1) < 3 ) || (log2InRow > 9) {
    tempSize = 9 * nr;
} else {
    tempSize = 17 * nr
}
bufferTemp.realp = (float *) malloc (tempSize * sizeOf(float));
bufferTemp.imagp = (float *) malloc (tempSize * sizeOf(float));
```

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions `vDSP_create_fftsetup` (page 195) and `vDSP_destroy_fftsetupD` (page 202).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft2d_zroptD

Computes an out-of-place double-precision real discrete FFT, either from the spatial domain to the frequency domain (forward) or from the frequency domain to the spatial domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft2d_zroptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_Kr,
   vDSP_Stride __vDSP_Kc,
   DSPDoubleSplitComplex *__vDSP_ioData2,
   vDSP_Stride __vDSP_Ir,
   vDSP_Stride __vDSP_Ic,
   DSPDoubleSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2nc,
   vDSP_Length __vDSP_log2nr,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` or `log2m`, whichever is larger, of the transform function.

ioData

A complex vector input.

Kr

Specifies a stride across each row of matrix signal. Specifying 1 for `signalStrideInRow` processes every element across each row, specifying 2 processes every other element across each row, and so forth.

Kc

If not 0, represents the distance between each row of the input /output matrix. If parameter signalStrideInCol is 1024, for instance, element 512 equates to element (1,0) of matrix `a`, element 1024 equates to element (2,0), and so forth.

ioData2

The complex vector result.

Ir

Specifies a row stride for output matrix `ioData2` in the same way that `Kr` specifies strides for input the matrix.

Ic

Specifies a column stride for output matrix `ioData2` in the same way that `Kc` specify strides for input matrix `ioData`.

temp

A temporary matrix used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 16 KB or 4*`n`, where `log2n = log2nInCol + log2nInRow`.

log2nc

The base 2 exponent of the number of columns to process for each row. `log2nc` must be between 3 and 10, inclusive.

log2nr

The base 2 exponent of the number of rows to process. For example, to process 64 rows of 128 columns, specify 7 for `log2nc` and 6 for `log2nr`. `log2nr` must be between 3 and 10, inclusive.

flag

A forward/inverse directional flag, and must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Results are undefined for other values of `flag`.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, spatial-domain data and its equivalent frequency-domain data have the same storage requirements.

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions `vDSP_create_fftsetupD` (page 196) and `vDSP_destroy_fftsetupD` (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_FFT32_copv

Performs a 32-element FFT on interleaved complex data.

```
void vDSP_FFT32_copv (
   float *__vDSP_Output,
   const float *__vDSP_Input,
   FFTDirection __vDSP_Direction
);
```

Parameters

__vDSP_Output

A vector-block-aligned output array.

__vDSP_Input

A vector-block-aligned input array.

__vDSP_Direction

`kFFTDirection_Forward` (page 552) or `kFFTDirection_Inverse` (page 552), indicating whether to perform a forward or inverse transform.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_FFT32_zopv

Performs a 32-element FFT on split complex data.

```
void vDSP_FFT32_zopv (
   float *__vDSP_Or,
   float *__vDSP_Oi,
   const float *__vDSP_Ir,
   const float *__vDSP_Ii,
   FFTDirection __vDSP_Direction
);
```

Parameters

__vDSP_Or
: Output vector for real parts.

__vDSP_Oi
: Output vector for imaginary parts.

__vDSP_Ir
: Input vector for real parts.

__vDSP_Ii
: Input vector for imaginary parts.

__vDSP_Direction
: kFFTDirection_Forward (page 552) or kFFTDirection_Inverse (page 552), indicating whether to perform a forward or inverse transform.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_fft3_zop

Computes an out-of-place radix-3 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 3 times the power of 2 specified by parameter log2n; single precision.

```
void vDSP_fft3_zop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Use vDSP_create_fftsetup (page 195), to initialize this function. kFFTRadix3 (page 552) must be specified in the call to vDSP_create_fftsetup. setup is preserved for reuse.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through the input vector signal. To process every element of the vector, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

result

The complex vector signal output.

resultStride

Specifies an address stride for the result. The value of `resultStride` should be 1 for best performance.

log2n

The base 2 exponent of the number of elements to process in a single input signal. `log2n` must be between 3 and 15, inclusive.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

Where N = $3(2^m)$ for Radix-3 functions, and N = $5(2^m)$ for Radix-5 functions

If F = 1 $C_m = \mathrm{RDFT}(A_m) \cdot 2$ If F = -1 $C_m = \mathrm{IDFT}\ (A_m) \cdot N$ m = {0, N-1}

$$\mathrm{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \mathrm{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft3_zopD

Computes an out-of-place radix-3 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 3 times the power of 2 specified by parameter `log2n`; double precision.

```
void vDSP_fft3_zopD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_K,
   DSPDoubleSplitComplex *__vDSP_ioData2,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Use `vDSP_create_fftsetupD` (page 196), to initialize this function. `kFFTRadix3` (page 552) must be specified in the call to `vDSP_create_fftsetupD`. `setup` is preserved for reuse.

signal
A complex vector input.

K
Specifies an address stride through the input vector `signal`. To process every element of the vector, specify 1 for parameter `K`; to process every other element, specify 2. The value of `K` should be 1 for best performance.

result
The complex vector result.

L
Specifies an address stride for the result. The value of `L` should be 1 for best performance.

log2n
The base 2 exponent of the number of elements to process in a single input signal. `log2n` must be between 3 and 15, inclusive.

flag
A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
This performs the following operation:

Where $N = 3(2^m)$ for Radix-3 functions, and $N = 5(2^m)$ for Radix-5 functions

If F = 1 $C_m = \mathrm{RDFT}(A_m) \cdot 2$ If F = -1 $C_m = \mathrm{IDFT}(A_m) \cdot N$ m = {0, N-1}

$$\mathrm{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \mathrm{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and “Using Fourier Transforms”.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft5_zop

Computes an out-of-place radix-5 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 5 times the power of 2 specified by parameter `log2n`; single precision.

```
void vDSP_fft5_zop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`
: Use `vDSP_create_fftsetup` (page 195), to initialize this function. `kFFTRadix5` (page 553) must be specified in the call to `vDSP_create_fftsetup`. `setup` is preserved for reuse.

`signal`
: A complex vector signal input.

`signalStride`
: Specifies an address stride through the input vector `signal`. To process every element of the vector, specify 1 for parameter `signalStride`; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

`result`
: The complex vector signal output.

`resultStride`
: Specifies an address stride for the result. The value of `resultStride` should be 1 for best performance.

`log2n`
: The base 2 exponent of the number of elements to process in a single input signal. `log2n` must be between 3 and 15, inclusive.

`flag`
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

Where $N = 3(2^m)$ for Radix-3 functions, and $N = 5(2^m)$ for Radix-5 functions

If $F = 1$ $\quad C_m = \mathrm{RDFT}(A_m) \bullet 2 \qquad$ If $F = -1$ $\quad C_m = \mathrm{IDFT}(A_m) \bullet N \qquad m = \{0, N-1\}$

$$\mathrm{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \mathrm{IDFT}(X_m) = \frac{1}{N} \sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft5_zopD

Computes an out-of-place radix-5 complex Fourier transform, either forward or inverse. The number of input and output values processed equals 5 times the power of 2 specified by parameter `log2n`; double precision.

```
void vDSP_fft5_zopD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_K,
   DSPDoubleSplitComplex *__vDSP_ioData2,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Use `vDSP_create_fftsetupD` (page 196), to initialize this function. `kFFTRadix5` (page 553) must be specified in the call to `vDSP_create_fftsetupD`. `setup` is preserved for reuse.

signal

A complex vector input.

K

Specifies an address stride through the input vector `signal`. To process every element of the vector, specify 1 for parameter `K`; to process every other element, specify 2.

result

The complex vector result.

L

Specifies an address stride for the result.

log2n

The base 2 exponent of the number of elements to process in a single input signal.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

Where $N = 3(2^m)$ for Radix-3 functions, and $N = 5(2^m)$ for Radix-5 functions

If F = 1 $\quad C_m = \mathrm{RDFT}(A_m) \cdot 2 \qquad$ If F = -1 $\quad C_m = \mathrm{IDFT}(A_m) \cdot N \qquad m = \{0, N\text{-}1\}$

$$\mathrm{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \mathrm{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zip

Performs the same operation as `vDSP_fft_zip` (page 270) on multiple signals with a single call.

```
void vDSP_fftm_zip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

`signal`
: A complex vector that stores the input and output signal.

`signalStride`
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2.

`fftStride`
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

`log2n`
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

`numFFT`
: The number of signals.

`flag`
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
This function allows you to perform Discrete Fourier Transforms on multiple input signals using a single call. They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The functions compute in-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zipD

Performs the same operation as `vDSP_fft_zipD` (page 271) on multiple signals with a single call.

```
void vDSP_fftm_zipD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector that stores the input and output signal.

signalStride
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

numFFT
: The number of signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
This function allows you to perform Discrete Fourier Transforms on multiple signals at once, using a single call. It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The function computes in-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zipt

Performs the same operation as `vDSP_fft_zipt` (page 272) on multiple signals with a single call.

```
void vDSP_fftm_zipt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

`signal`
A complex vector that stores the input and output signal.

`signalStride`
Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

`fftStride`
The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

`temp`
A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

`log2n`
The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

`numFFT`
The number of different input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The function allows you to perform Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input/output vector, the parameter `signal`.

The function computes in-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fftm_ziptD

Performs the same operation as `vDSP_fft_ziptD` (page 273) on multiple signals with a single call.

```
void vDSP_fftm_ziptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector that stores the input and output signal.

signalStride
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

temp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

numFFT
: The number of different input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The function allows you to perform Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input/output vector, the parameter `signal`.

The function computes in-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zop

Performs the same operation as `vDSP_fft_zop` (page 274) on multiple signals with a single call.

```
void vDSP_fftm_zop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector that stores the input signal.

signalStride

Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

fftStride

The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result

The complex vector signal output.

resultStride

Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

rfftStride

The number of elements between the first element of one result vector and the next in the output vector `result`.

log2n

The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

numFFT

The number of input signals.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The function allows you to perform Discrete Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input vector, `signal`, and single output vector, `result`.

The function computes out-of-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zopD

Performs the same operation as `vDSP_fft_zopD` (page 276) on multiple signals with a single call.

```
void vDSP_fftm_zopD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195) or `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector signal input.

signalStride
Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

fftStride
The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

`result`

The complex vector signal output.

`resultStride`

Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

`rfftStride`

The number of elements between the first element of one result vector and the next in the output vector `result`.

`log2n`

The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

`numFFT`

The number of different input signals.

`flag`

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The function allows you to perform Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input/output vector, the parameter `signal`.

The function computes out-of-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fftm_zopt

Performs the same operation as `vDSP_fft_zopt` (page 277) on multiple signals with a single call.

```
void vDSP_fftm_zopt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   DSPSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result
: The complex vector signal output.

resultStride
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

rfftStride
: The number of elements between the first element of one result vector and the next in the output vector `result`.

temp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

numFFT
: The number of different input signals.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The function allows you to perform Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input/output vector, the parameter `signal`.

The function computes out-of-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and "Using Fourier Transforms".

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fftm_zoptD

Performs the same operation as `vDSP_fft_zoptD` (page 278) on multiple signals with a single call.

```
void vDSP_fftm_zoptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   DSPDoubleSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector signal input.

signalStride

Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2. The value of signalStride should be 1 for best performance.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result
: The complex vector signal output.

resultStride
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

rfftStride
: The number of elements between the first element of one result vector and the next in the output vector `result`.

temp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`. The value of `log2n` must be between 2 and 12, inclusive.

numFFT
: The number of different input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The function allows you to perform Fourier transforms on a number of different input signals at once, using a single call. It can be used for efficient processing of small input signals (less than 512 points). It will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input/output vector, the parameter `signal`.

The function computes out-of-place complex discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and “Using Fourier Transforms”.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zrip

Performs the same operation as `vDSP_fft_zrip` (page 279) on multiple signals with a single call.

```
void vDSP_fftm_zrip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter `signalStride`; to process every other element, specify 2.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

numFFT
: The number of input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The function allows you to perform Discrete Fourier Transforms on multiple signals using a single call. They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The functions compute in-place real Discrete Fourier Transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fftm_zripD

Performs the same operation as `vDSP_fft_zripD` (page 281) on multiple signals with a single call.

```
void vDSP_fftm_zripD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector signal input.

signalStride

Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter `signalStride`; to process every other element, specify 2.

fftStride

The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

log2n

The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

numFFT

The number of input signals.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The functions allow you to perform Fourier transforms on a number of different input signals at once, using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The functions compute in-place real discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zript

Performs the same operation as `vDSP_fft_zript` (page 282) on multiple signals with a single call.

```
void vDSP_fftm_zript (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter `signalStride`; to process every other element, specify 2.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

temp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

numFFT
: The number of different input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The functions allow you to perform Fourier transforms on a number of different input signals at once, using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The functions compute in-place real Discrete Fourier Transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zriptD

Performs the same operation as `vDSP_fft_zriptD` (page 283) on multiple signals with a single call.

```
void vDSP_fftm_zriptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector signal input.

signalStride
Specifies an address stride through the input signals. To process every element of each signal, specify 1 for parameter signalStride; to process every other element, specify 2.

fftStride
The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

temp
A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
The base 2 exponent of the number of elements to process in a single input signal. For example, to process 512 elements, specify 9 for parameter `log2n`.

numFFT
The number of input signals.

flag
A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The functions allow you to perform Fourier transforms on a number of different input signals at once, using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride.

The functions compute in-place real Discrete Fourier Transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zrop

Performs the same operation as `vDSP_fft_zrop` (page 284) on multiple signals with a single call.

```
void vDSP_fftm_zrop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector signal input.

signalStride
Specifies an address stride through input signals. To process every element of each signal, specify a stride of 1; to process every other element, specify 2.

fftStride
The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result
: The complex vector signal output.

resultStride
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

rfftStride
: The number of elements between the first element of one result vector and the next in the output vector `result`.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

numFFT
: The number of input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This function allows you to perform Discrete Fourier Transforms on multiple input signals using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input vector, `signal`, and a single output vector, `result`.

The functions compute out-of-place real Discrete Fourier Transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and “Using Fourier Transforms”.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fftm_zropD

Performs the same operation as `VDSP_fft_zropD` on multiple signals with a single call.

```
void vDSP_fftm_zropD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through input signals. To process every element of each signal, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

fftStride
: The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result
: The complex vector signal output.

resultStride
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

rfftStride
: The number of elements between the first element of one result vector and the next in the output vector `result`.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

numFFT
: The number of input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This function allows you to perform Discrete Fourier Transforms on multiple input signals using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input vector, the parameter `signal`.

The functions compute out-of-place real Discrete Fourier Transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zropt

Performs the same operation as `VDSP_fft_zropt` on multiple signals with a single call.

```
void vDSP_fftm_zropt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   DSPSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

`setup`
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

`signal`
A complex vector signal input.

`signalStride`
Specifies an address stride through input signals. To process every element of each signal, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

`fftStride`
The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

`result`
The complex vector signal output.

`resultStride`
Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

rfftStride
: The number of elements between the first element of one result vector and the next in the output vector `result`.

temp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

numFFT
: The number of input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
The functions allow you to perform Fourier transforms on a number of different input signals at once, using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input vector, the parameter `signal`.

The functions compute out-of-place real discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetup (page 195), vDSP_destroy_fftsetup (page 201), and "Using Fourier Transforms".

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fftm_zroptD

Performs the same operation as `VDSP_fft_zroptD` on multiple signals with a single call.

```
void vDSP_fftm_zroptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   vDSP_Stride __vDSP_fftStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_resultStride,
   vDSP_Stride __vDSP_rfftStride,
   DSPDoubleSplitComplex *__vDSP_temp,
   vDSP_Length __vDSP_log2n,
   vDSP_Length __vDSP_numFFT,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector signal input.

signalStride

Specifies an address stride through input signals. To process every element of each signal, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

fftStride

The number of elements between the first element of one input signal and the first element of the next (which is also to length of each input signal, measured in elements).

result

The complex vector signal output.

resultStride

Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `resultStride` should be 1 for best performance.

rfftStride

The number of elements between the first element of one result vector and the next in the output vector `result`.

temp

A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `temp.realp` and `temp.imagp` should be 32-byte aligned for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

numFFT

The number of different input signals.

flag
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

The functions allow you to perform Fourier transforms on a number of different input signals at once, using a single call. They can be used for efficient processing of small input signals (less than 512 points). They will work for input signals of 4 points or greater. Each of the input signals processed by a given call must have the same length and address stride. The input signals are concatenated into a single input vector, the parameter `signal`.

The functions compute out-of-place real discrete Fourier transforms of the input signals, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

See also functions vDSP_create_fftsetupD (page 196), vDSP_destroy_fftsetupD (page 202), and "Using Fourier Transforms".

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zip

Computes an in-place single-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector that stores the input and output signal.

stride
: Specifies an address stride through the input/output vector signal. To process every element of the vector, specify 1 for parameter `stride`; to process every other element, specify 2.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \quad C_m = \text{FDFT}(C_m) \qquad \text{If F} = \text{-1} \quad C_m = \text{IDFT}\ (C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zipD

Computes an in-place double-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zipD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector that stores the input and output signal.

stride

Specifies an address stride through the input/output vector signal. To process every element of the vector, specify 1 for parameter `stride`; to process every other element, specify 2. The value of `stride` should be 1 for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{FDFT}(C_m) \qquad \text{If F = -1} \quad C_m = \text{IDFT}\,(C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zipt

Computes an in-place single-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft_zipt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector that stores the input and output signal.

stride

Specifies an address stride through the input/output vector signal. To process every element of the vector, specify 1 for parameter `stride`; to process every other element, specify 2.

bufferTemp

A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `bufferTemp.realp` and `bufferTemp.imagp` should be 32-byte aligned for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \quad C_m = \text{FDFT}(C_m) \qquad \text{If F} = -1 \quad C_m = \text{IDFT}\,(C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_ziptD

Computes an in-place double-precision complex discrete Fourier transform of the input/output vector signal, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse). A buffer is used for intermediate results.

```
void vDSP_fft_ziptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the earlier call to the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector that stores the input and output signal.

stride
: Specifies an address stride through the input/output vector signal. To process every element of the vector, specify 1 for parameter `stride`; to process every other element, specify 2.

bufferTemp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `bufferTemp.realp` and `bufferTemp.imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \quad C_m = \text{FDFT}(C_m) \qquad \text{If F} = -1 \quad C_m = \text{IDFT}\ (C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zop

Computes an out-of-place single-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

`setup`
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

`signal`
: A complex vector that stores the input and output signal.

`signalStride`
: Specifies an address stride through input vector signal. Parameter `strideResult` specifies an address stride through output vector result. Thus, to process every element, specify a signalStride of 1; to process every other element, specify 2.

`result`
: The complex vector signal output.

`strideResult`
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

`log2n`
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

`direction`
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{FDFT}(A_m) \qquad \text{If F = -1} \quad C_m = \text{IDFT}(A_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N} \sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zopD

Computes an out-of-place double-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zopD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195) or `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector that stores the input signal.

signalStride

Specifies an address stride through input vector signal. Parameter `strideResult` specifies an address stride through output vector result. Thus, to process every element, specify a `signalStride` of 1; to process every other element, specify 2. The values of `signalStride` and `strideResult` should be 1 for best performance.

result

The complex vector signal output.

strideResult

Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `strideResult` should be 1 for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{FDFT}(A_m) \qquad \text{If F = -1} \quad C_m = \text{IDFT}(A_m) \cdot N \qquad m = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft_zopt

Computes an out-of-place single-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zopt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector that stores the input and output signal.

signalStride
Specifies an address stride through input vector signal. Parameter `strideResult` specifies an address stride through output vector result. Thus, to process every element, specify a signalStride of 1; to process every other element, specify 2. The values of signalStride and `strideResult` should be 1 for best performance.

result
The complex vector signal output.

strideResult
Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

bufferTemp
A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, tempBuffer`.realp` and tempBuffer`.imagp` should be 32-byte aligned for best performance.

log2n
The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \qquad C_m = \text{FDFT}(A_m) \qquad \text{If F} = -1 \qquad C_m = \text{IDFT}\ (A_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N} \sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zoptD

Computes an out-of-place double-precision complex discrete Fourier transform of the input vector, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zoptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector signal input.

signalStride

Specifies an address stride through input vector signal. Parameter `strideResult` specifies an address stride through output vector result. Thus, to process every element, specify a signalStride of 1; to process every other element, specify 2. The values of signalStride and `strideResult` should be 1 for best performance.

result
The complex vector signal output.

strideResult
Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

bufferTemp
A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, tempBuffer.`realp` and tempBuffer.`imagp` should be 32-byte aligned for best performance.

log2n
The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction
A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{FDFT}(A_m) \qquad \text{If F = -1} \quad C_m = \text{IDFT}(A_m) \bullet N \qquad m = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft_zrip

Computes an in-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zrip (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

C

A complex input/output vector.

K

Specifies an address stride through the input/output vector. To process every element of the vector, specify 1 for parameter `signalStride`; to process every other element, specify 2.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

F

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

Forward transforms read real input and write packed complex output. You can find more details on the packing format in *vDSP Programming Guide*. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements.

$$\text{If F = 1} \quad C_m = \text{RDFT}(C_m) \bullet 2 \qquad \text{If F = -1} \quad C_m = \text{IDFT}\ (C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zripD

Computes an in-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zripD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

C

A complex vector input.

K

Specifies an address stride through the input/output vector. To process every element of the vector, specify 1 for parameter `stride`; to process every other element, specify 2.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

F

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements.

If F = 1 $\quad C_m = \mathrm{RDFT}(C_m) \bullet 2 \qquad$ If F = -1 $\quad C_m = \mathrm{IDFT}\,(C_m) \bullet N \qquad m = \{0, N\text{-}1\}$

$$\mathrm{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \mathrm{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zript

Computes an in-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zript (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, vDSP_create_fftsetup (page 195). The value supplied as parameter log2n of the setup function must equal or exceed the value supplied as parameter log2n of this transform function.

C

A complex vector input.

K

Specifies an address stride through the input/output vector. To process every element of the vector, specify 1 for parameter signalStride; to process every other element, specify 2.

bufferTemp

A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, tempBuffer.realp and tempBuffer.imagp should be 32-byte aligned for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter log2n.

F

A forward/inverse directional flag, which must specify kFFTDirection_Forward (page 552) for a forward transform or kFFTDirection_Inverse (page 552) for an inverse transform.

Discussion

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements.

$$\text{If F} = 1 \quad C_m = \text{RDFT}(C_m) \bullet 2 \qquad \text{If F} = -1 \quad C_m = \text{IDFT}\ (C_m) \bullet N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft_zriptD

Computes an in-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zriptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_ioData,
   vDSP_Stride __vDSP_stride,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

C
: A complex vector input.

K
: Specifies an address stride through the input/output vector. To process every element of the vector, specify 1 for parameter `signalStride`; to process every other element, specify 2.

bufferTemp
: A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, tempBuffer.`realp` and tempBuffer.`imagp` should be 32-byte aligned for best performance.

log2n
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

F
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements.

If F = 1 $C_m = \text{RDFT}(C_m) \bullet 2$ If F = -1 $C_m = \text{IDFT}\ (C_m) \bullet N$ m = {0, N-1}

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft_zrop

Computes an out-of-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zrop (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup
: Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
: A complex vector signal input.

signalStride
: Specifies an address stride through input vector signal. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

result
: The complex vector signal output.

strideResult
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \qquad C_m = \text{RDFT}(A_m) \bullet 2 \qquad \text{If F} = \text{-}1 \qquad C_m = \text{IDFT}\ (A_m) \bullet N \qquad \text{m} = \{0, \text{N-}1\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \bullet e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \bullet e^{(j2\pi nm)/N}$$

Forward transforms read real input and write packed complex output (see *vDSP Programming Guide* for details on the packing format). Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements.

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zropD

Computes an out-of-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zropD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup

Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal

A complex vector signal input.

`signalStride`
: Specifies an address stride through input vector signal. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

`result`
: The complex vector signal output.

`strideResult`
: Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `strideResult` should be 1 for best performance.

`log2n`
: The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

`flag`
: A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F} = 1 \quad C_m = \text{RDFT}(A_m) \cdot 2 \qquad \text{If F} = -1 \quad C_m = \text{IDFT}\ (A_m) \cdot N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements. Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_fft_zropt

Computes an out-of-place single-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zropt (
   FFTSetup __vDSP_setup,
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   DSPSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_direction
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetup` (page 195). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector signal input.

signalStride
Specifies an address stride through input vector signal. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

result
The complex vector signal output.

strideResult
Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `strideResult` should be 1 for best performance.

bufferTemp
A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, tempBuffer.`realp` and tempBuffer.`imagp` should be 32-byte aligned for best performance.

log2n
The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

direction
A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion
This performs the following operation:

$$\text{If F} = 1 \qquad C_m = \text{RDFT}(A_m) \cdot 2 \qquad \text{If F} = -1 \qquad C_m = \text{IDFT}\ (A_m) \cdot N \qquad \text{m} = \{0, \text{N-1}\}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N} \sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements. Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetup (page 195) and vDSP_destroy_fftsetup (page 201).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_fft_zroptD

Computes an out-of-place double-precision real discrete Fourier transform, either from the time domain to the frequency domain (forward) or from the frequency domain to the time domain (inverse).

```
void vDSP_fft_zroptD (
   FFTSetupD __vDSP_setup,
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   DSPDoubleSplitComplex *__vDSP_bufferTemp,
   vDSP_Length __vDSP_log2n,
   FFTDirection __vDSP_flag
);
```

Parameters

setup
Points to a structure initialized by a prior call to the FFT weights array function, `vDSP_create_fftsetupD` (page 196). The value supplied as parameter `log2n` of the setup function must equal or exceed the value supplied as parameter `log2n` of this transform function.

signal
A complex vector signal input.

signalStride
Specifies an address stride through input vector signal. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `signalStride` should be 1 for best performance.

result
The complex vector signal output.

strideResult
Specifies an address stride through output vector result. Thus, to process every element, specify a stride of 1; to process every other element, specify 2. The value of `strideResult` should be 1 for best performance.

bufferTemp

A temporary vector used for storing interim results. The size of temporary memory for each part (real and imaginary) is the lower value of 4*n or 16k for best performance. Or you can simply pass the buffer of size 2^(log2n) for each part (real and imaginary). If possible, `tempBuffer.realp` and `tempBuffer.imagp` should be 32-byte aligned for best performance.

log2n

The base 2 exponent of the number of elements to process. For example, to process 1024 elements, specify 10 for parameter `log2n`.

flag

A forward/inverse directional flag, which must specify `kFFTDirection_Forward` (page 552) for a forward transform or `kFFTDirection_Inverse` (page 552) for an inverse transform.

Discussion

This performs the following operation:

$$\text{If F = 1} \quad C_m = \text{RDFT}(A_m) \cdot 2 \qquad \text{If F = -1} \quad C_m = \text{IDFT}\ (A_m) \cdot N \qquad \text{m = \{0, N-1\}}$$

$$\text{FDFT}(X_m) = \sum_{n=0}^{N-1} X_n \cdot e^{(-j2\pi nm)/N} \qquad \text{IDFT}(X_m) = \frac{1}{N}\sum_{n=0}^{N-1} X_n \cdot e^{(j2\pi nm)/N}$$

Forward transforms read real input and write packed complex output. Inverse transforms read packed complex input and write real output. As a result of packing the frequency-domain data, time-domain data and its equivalent frequency-domain data have the same storage requirements. Real data is stored in split complex form, with odd reals stored on the imaginary side of the split complex form and even reals in stored on the real side.

See also functions vDSP_create_fftsetupD (page 196) and vDSP_destroy_fftsetupD (page 202).

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_hamm_window

Creates a single-precision Hamming window.

```
void vDSP_hamm_window (
   float *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion

$$C_n = 0.54 - 0.46 \cos\frac{2\pi n}{N} \qquad \text{n = \{0, N-1\}}$$

vDSP_hamm_window creates a single-precision Hamming window function C, which can be multiplied by a vector using vDSP_vmul (page 425). Specify the vDSP_HALF_WINDOW (page 553) flag to create only the first (n+1)/2 points, or 0 (zero) for a full-size window.

See also vDSP_vmul (page 425).

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_hamm_windowD

Creates a double-precision Hamming window.

```
void vDSP_hamm_windowD (
   double *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion

$$C_n = 0.54 - 0.46 \cos \frac{2\pi n}{N} \qquad n = \{0, N\text{-}1\}$$

vDSP_hamm_windowD creates a double-precision Hamming window function C, which can be multiplied by a vector using vDSP_vmulD (page 426). Specify the vDSP_HALF_WINDOW (page 553) flag to create only the first (n+1)/2 points, or 0 (zero) for a full-size window.

See also vDSP_vmulD (page 426).

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_hann_window

Creates a single-precision Hanning window.

```
void vDSP_hann_window (
   float *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion

$$C_n = W\left(1.0 - \cos \frac{2\pi n}{N}\right) \qquad n = \{0, N\text{-}1\}$$

vDSP_hann_window creates a single-precision Hanning window function C, which can be multiplied by a vector using vDSP_vmul (page 425).

The FLAG parameter can have the following values:

- vDSP_HANN_DENORM (page 553) creates a denormalized window.
- vDSP_HANN_NORM (page 553) creates a normalized window.
- vDSP_HALF_WINDOW (page 553) creates only the first (N+1)/2 points.

vDSP_HALF_WINDOW can be ORed with any of the other values (i.e., using the C operator |).

See also vDSP_vmul (page 425).

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_hann_windowD

Creates a double-precision Hanning window.

```
void vDSP_hann_windowD (
   double *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_FLAG
);
```

Discussion

$$C_n = W\left(1.0 - \cos\frac{2\pi n}{N}\right) \qquad n = \{0, N\text{-}1\}$$

vDSP_hann_window creates a double-precision Hanning window function C, which can be multiplied by a vector using vDSP_vmulD (page 426).

The FLAG parameter can have the following values:

- vDSP_HANN_DENORM (page 553) creates a denormalized window.
- vDSP_HANN_NORM (page 553) creates a normalized window.
- vDSP_HALF_WINDOW (page 553) creates only the first (N+1)/2 points.

vDSP_HALF_WINDOW can ORed with any of the other values (i.e., using the C operator |).

See also vDSP_vmulD (page 426).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_imgfir

Filters an image by performing a two-dimensional convolution with a kernel; single precision.

```
void vDSP_imgfir (
   float *__vDSP_signal,
   vDSP_Length __vDSP_numRow,
   vDSP_Length __vDSP_numCol,
   float *__vDSP_filter,
   float *__vDSP_result,
   vDSP_Length __vDSP_fnumRow,
   vDSP_Length __vDSP_fnumCol
);
```

Parameters

A
: A real matrix signal input.

M
: Number of rows in `A`.

N
: Number of columns in `A`.

B
: A two-dimensional real matrix containing the filter.

C
: Stores real output matrix.

P
: Number of rows in `B`.

Q
: Number of columns in `B`.

Discussion
The image is given by the input matrix `A`. It has `M` rows and `N` columns.

$$C_{(m+(P\text{-}1)/2,n+(Q\text{-}1)/2)} = \sum_{p=0}^{P-1}\sum_{q=0}^{Q-1} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad \text{m = \{0, M-P\} and n = \{0, N-Q\}}$$

`B` is the filter kernel. It has `P` rows and `Q` columns.

Ensure `Q` >= `P` for best performance.

The filtered image is placed in the output matrix `C`. The function pads the perimeter of the output image with a border of (`P`-1)/2 rows of zeros on the top and bottom and (`Q`-1)/2 columns of zeros on the left and right.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_imgfirD

Filters an image by performing a two-dimensional convolution with a kernel; double precision.

```
void vDSP_imgfirD (
   double *__vDSP_signal,
   vDSP_Length __vDSP_numRow,
   vDSP_Length __vDSP_numCol,
   double *__vDSP_filter,
   double *__vDSP_result,
   vDSP_Length __vDSP_fnumRow,
   vDSP_Length __vDSP_fnumCol
);
```

Parameters

A
: A complex vector signal input.

M
: Number of rows in input matrix.

N
: Number of columns in input matrix.

B
: A two-dimensional real matrix containing the filter.

C
: Stores real output matrix.

P
: Number of rows in `B`.

Q
: Number of columns in `B`.

Discussion
The image is given by the input matrix `A`. It has `M` rows and `N` columns.

$$C_{(m+(P\text{-}1)/2,n+(Q\text{-}1)/2)} = \sum_{p=0}^{P-1}\sum_{q=0}^{Q-1} A_{(m+p,n+q)} \cdot B_{(p,q)} \qquad \text{m} = \{0, \text{M-P}\} \text{ and n} = \{0, \text{N-Q}\}$$

`B` is the filter kernel. It has `P` rows and `Q` columns. For best performance, ensure `Q` >= `P`.

The filtered image is placed in the output matrix `C`. The functions pad the perimeter of the output image with a border of (`P`-1)/2 rows of zeros on the top and bottom and (`Q`-1)/2 columns of zeros on the left and right.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_maxmgv

Vector maximum magnitude; single precision.

```
void vDSP_maxmgv (
   const float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

```
*C = 0;
for (n = 0; n < N; ++n)
    if (*C < abs(A[n*I]))
        *C = abs(A[n*I]);
```

Finds the element with the greatest magnitude in vector A and copies this value to scalar *C.

If N is zero (0), this function returns a value of 0 in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_maxmgvD

Vector maximum magnitude; double precision.

```
void vDSP_maxmgvD (
   const double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

```
*C = 0;
for (n = 0; n < N; ++n)
    if (*C < abs(A[n*I]))
        *C = abs(A[n*I]);
```

Finds the element with the greatest magnitude in vector A and copies this value to scalar *C.

If N is zero (0), this function returns a value of 0 in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_maxmgvi

Vector maximum magnitude with index; single precision.

```
void vDSP_maxmgvi (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A.

C

Output scalar.

IC

Output scalar index.

N

Count of values in A.

Discussion

This performs the following operation:

```
*C = 0;
```

```
for (n = 0; n < N; ++n)
{
    if (*C < abs(A[n*I]))
    {
        *C = abs(A[n*I]);
        *IC = n*I;
    }
}
```

Copies the element with the greatest magnitude from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index.If vector `A` contains more than one instance of the maximum magnitude, `*IC` contains the index of the first instance.

If `N` is zero (`0`), this function returns a value of 0 in `*C`, and the value in `*IC` is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_maxmgviD

Vector maximum magnitude with index; double precision.

```
void vDSP_maxmgviD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector.

I
: Stride for `A`

C
: Output scalar

IC
: Output scalar

N
: Count

Discussion
This performs the following operation:

```
*C = 0;
for (n = 0; n < N; ++n)
{
    if (*C < abs(A[n*I]))
    {
        *C = abs(A[n*I]);
        *IC = n*I;
    }
}
```

Copies the element with the greatest magnitude from real vector A to real scalar *C, and writes its zero-based index to integer scalar *IC. The index is the actual array index, not the pre-stride index. If vector A contains more than one instance of the maximum magnitude, *IC contains the index of the first instance.

If N is zero (0), this function returns a value of 0 in *C, and the value in *IC is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_maxv

Vector maximum value; single precision.

```
void vDSP_maxv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
 Single-precision real input vector.

I
 Stride for A

C
 Output scalar

N
 Count

Discussion
This performs the following operation:

```
*C = -INFINITY;
for (n = 0; n < N; ++n)
    if (*C < A[n*I])
        *C = A[n*I];
```

Finds the element with the greatest value in vector A and copies this value to scalar *C. If N is zero (0), this function returns -INFINITY in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_maxvD

Vector maximum value; double precision.

```
void vDSP_maxvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C
: Output scalar

N
: Count

Discussion

This performs the following operation:

```
*C = -INFINITY;
for (n = 0; n < N; ++n)
    if (*C < A[n*I])
        *C = A[n*I];
```

Finds the element with the greatest value in vector `A` and copies this value to scalar `*C`. If `N` is zero (`0`), this function returns `-INFINITY` in `*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_maxvi

Vector maximum value with index; single precision.

```
void vDSP_maxvi (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector.

I

Stride for `A`

C

Output scalar

IC

Output scalar index

N

Count

Discussion

This performs the following operation:

```
*C = -INFINITY;
for (n = 0; n < N; ++n)
{
    if (*C < A[n * I])
    {
        *C = A[n * I];
        *IC = n * I;
    }
}
```

Copies the element with the greatest value from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the maximum value, `*IC` contains the index of the first instance. If `N` is zero (`0`), this function returns a value of `-INFINITY` in `*C` and `*IC` is undetermined.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_maxviD

Vector maximum value with index; double precision.

```
void vDSP_maxviD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for `A`

C

Output scalar

IC

Output scalar

N

Count

Discussion

This performs the following operation:

```
*C = -INFINITY;
for (n = 0; n < N; ++n)
{
    if (*C < A[n * I])
    {
        *C = A[n * I];
        *IC = n * I;
    }
}
```

Copies the element with the greatest value from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the maximum value, `*IC` contains the index of the first instance. If `N` is zero (`0`), this function returns a value of `-INFINITY` in `*C` and the value in `*IC` is undefined.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_meamgv

Vector mean magnitude; single precision.

```
void vDSP_meamgv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

C

Output scalar

N

Count

Discussion

This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} |A_{ni}|$$

Finds the mean of the magnitudes of elements of vector A and stores this value in scalar *C.

If N is zero (0), this function returns a NaN in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_meamgvD

Vector mean magnitude; double precision.

```
void vDSP_meamgvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for A

C
Output scalar

N
Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} |A_{ni}|$$

Finds the mean of the magnitudes of elements of vector A and stores this value in scalar *C.

If N is zero (0), this function returns a NaN in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_meanv

Vector mean value; single precision.

```
void vDSP_meanv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni}$$

Finds the mean value of the elements of vector A and stores this value in scalar *C.

If N is zero (0), this function returns a NaN in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_meanvD

Vector mean value; double precision.

```
void vDSP_meanvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I
: Stride for `A`

C
: Output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni}$$

Finds the mean value of the elements of vector `A` and stores this value in scalar `*C`.

If `N` is zero (`0`), this function returns a NaN in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_measqv

Vector mean square value; single precision.

```
void vDSP_measqv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector.

I
: Stride for `A`

C
: Output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni}^{2}$$

Finds the mean value of the squares of the elements of vector `A` and stores this value in scalar `*C`.

If `N` is zero (`0`), this function returns a NaN in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_measqvD

Vector mean square value; double precision.

```
void vDSP_measqvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`

C
Output scalar

N
Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni}^2$$

Finds the mean value of the squares of the elements of vector `A` and stores this value in scalar `*C`.

If `N` is zero (`0`), this function returns a NaN. in `*C`

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_minmgv

Vector minimum magnitude; single precision.

```
void vDSP_minmgv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

$$c = |A_0| \quad \text{If} \quad c > |A_{ni}| \quad \text{then} \quad c = |A_{ni}| \qquad n = \{1, N\text{-}1\}$$

Finds the element with the least magnitude in vector A and copies this value to scalar *C.

If N is zero (0), this function returns +INF in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_minmgvD

Vector minimum magnitude; double precision.

```
void vDSP_minmgvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

C

Output scalar

N
Count

Discussion
This performs the following operation:

$$c = |A_0| \quad \text{If} \quad c > |A_{ni}| \quad \text{then} \quad c = |A_{ni}| \qquad n = \{1, N\text{-}1\}$$

Finds the element with the least magnitude in vector `A` and copies this value to scalar `*C`.

If `N` is zero (`0`), this function returns +INF in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_minmgvi

Vector minimum magnitude with index; single precision.

```
void vDSP_minmgvi (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector.

I
Stride for `A`

C
Output scalar

IC
Output scalar index

N
Count

Discussion
This performs the following operation:

$$\begin{matrix} c = |A_0| \\ d = 0 \end{matrix} \quad \text{If} \quad c > |A_{ni}| \quad \text{then} \quad \begin{matrix} c = |A_{ni}| \\ d = ni \end{matrix} \qquad n = \{1, N\text{-}1\}$$

Copies the element with the least magnitude from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the least magnitude, `*IC` contains the index of the first instance.

If `N` is zero (`0`), this function returns a value of +INF in `*C` and the value in `*IC` is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_minmgviD

Vector minimum magnitude with index; double precision.

```
void vDSP_minmgviD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C
: Output scalar

IC
: Output scalar

N
: Count

Discussion
This performs the following operation:

$$\begin{matrix} c = |A_0| \\ d = 0 \end{matrix} \quad \text{If} \quad c > |A_{ni}| \quad \text{then} \quad \begin{matrix} c = |A_{ni}| \\ d = ni \end{matrix} \qquad n = \{1, N\text{-}1\}$$

Copies the element with the least magnitude from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the least magnitude, `*IC` contains the index of the first instance.

If `N` is zero (`0`), this function returns a value of +INF in `*C` and the value in `*IC` is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_minv

Vector minimum value.

```
void vDSP_minv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

$$c = A_0 \quad \text{If} \quad c > A_{ni} \quad \text{then} \quad c = A_{ni} \qquad n = \{1, N\text{-}1\}$$

Finds the element with the least value in vector A and copies this value to scalar *C.

If N is zero (0), this function returns +INFINITY in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_minvD

Vector minimum value; double precision.

```
void vDSP_minvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector.

I

Stride for A

C

Output scalar

N

Count

Discussion

This performs the following operation:

$$c = A_0 \qquad \text{If} \quad c > A_{ni} \quad \text{then} \quad c = A_{ni} \qquad \text{n} = \{1, \text{N-1}\}$$

Finds the element with the least value in vector `A` and copies this value to scalar `*C`.

If `N` is zero (`0`), this function returns +INF in `*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_minvi

Vector minimum value with index; single precision.

```
void vDSP_minvi (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for `A`

C

Output scalar

IC

Output scalar index

N

Count

Discussion

This performs the following operation:

$$\begin{matrix} c = A_0 \\ d = 0 \end{matrix} \qquad \text{If} \quad c > A_{ni} \quad \text{then} \quad \begin{matrix} c = A_{ni} \\ d = ni \end{matrix} \qquad \text{n} = \{1, \text{N-1}\}$$

Copies the element with the least value from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the least value, `*IC` contains the index of the first instance.

If `N` is zero (`0`), this function returns a value of +INF in `*C`, and the value in `*IC` is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_minviD

Vector minimum value with index; double precision.

```
void vDSP_minviD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector.

I
: Stride for `A`

C
: Output scalar

IC
: Output scalar

N
: Count

Discussion
This performs the following operation:

$$\begin{array}{llllll} c = A_0 & & & & c = A_{ni} & \\ & \text{If} & c > A_{ni} & \text{then} & & n = \{1, N\text{-}1\} \\ d = 0 & & & & d = ni & \end{array}$$

Copies the element with the least value from real vector `A` to real scalar `*C`, and writes its zero-based index to integer scalar `*IC`. The index is the actual array index, not the pre-stride index. If vector `A` contains more than one instance of the least value, `*IC` contains the index of the first instance.

If `N` is zero (`0`), this function returns a value of +INF in `*C`, and the value in `*IC` is undefined.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mmov

Copies the contents of a submatrix to another submatrix.

```
void vDSP_mmov (
   float *__vDSP_A,
   float *__vDSP_C,
   vDSP_Length __vDSP_NC,
   vDSP_Length __vDSP_NR,
   vDSP_Length __vDSP_TCA,
   vDSP_Length __vDSP_TCC
);
```

Parameters

A
: Single-precision real input submatrix

C
: Single-precision real output submatrix

NC
: Number of columns in `A` and `C`

NR
: Number of rows in `A` and `C`

TCA
: Number of columns in the matrix of which `A` is a submatrix

TCC
: Number of columns in the matrix of which `C` is a submatrix

Discussion

The matrices are assumed to be stored in row-major order. Thus elements `A[i][j]` and `A[i][j+1]` are adjacent. Elements `A[i][j]` and `A[i+1][j]` are `TCA` elements apart.

This function may be used to move a subarray beginning at any point in a larger embedding array by passing for `A` the address of the first element of the subarray. For example, to move a subarray starting at `A[3][4]`, pass `&A[3][4]`. Similarly, the address of the first destination element is passed for `C`

`NC` may equal `TCA`, and it may equal `TCC`. To copy all of an array to all of another array, pass the number of rows in `NR` and the number of columns in `NC`, `TCA`, and `TCC`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mmovD

Copies the contents of a submatrix to another submatrix.

```
void vDSP_mmovD (
   double *__vDSP_A,
   double *__vDSP_C,
   vDSP_Length __vDSP_NC,
   vDSP_Length __vDSP_NR,
   vDSP_Length __vDSP_TCA,
   vDSP_Length __vDSP_TCC
);
```

Parameters

A
: Double-precision real input submatrix

C
: Double-precision real output submatrix

NC
: Number of columns in `A` and `C`

NR
: Number of rows in `A` and `C`

TCA
: Number of columns in the matrix of which `A` is a submatrix

TCC
: Number of columns in the matrix of which `C` is a submatrix

Discussion

The matrices are assumed to be stored in row-major order. Thus elements `A[i][j]` and `A[i][j+1]` are adjacent. Elements `A[i][j]` and `A[i+1][j]` are `TCA` elements apart.

This function may be used to move a subarray beginning at any point in a larger embedding array by passing for `A` the address of the first element of the subarray. For example, to move a subarray starting at `A[3][4]`, pass `&A[3][4]`. Similarly, the address of the first destination element is passed for `C`

`NC` may equal `TCA`, and it may equal `TCC`. To copy all of an array to all of another array, pass the number of rows in `NR` and the number of columns in `NC`, `TCA`, and `TCC`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mmul

Performs an out-of-place multiplication of two matrices; single precision.

```
void vDSP_mmul (
   float *__vDSP_a,
   vDSP_Stride __vDSP_aStride,
   float *__vDSP_b,
   vDSP_Stride __vDSP_bStride,
   float *__vDSP_c,
   vDSP_Stride __vDSP_cStride,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function multiplies an M-by-P matrix (A) by a P-by-N matrix (B) and stores the results in an M-by-N matrix (C).

This performs the following operation:

$$C_{(mN+n)K} = \sum_{p=0}^{P-1} A_{(mP+p)I} \cdot B_{(pN+n)J} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\} \text{ and } \mathrm{m} = \{0, \mathrm{M}\text{-}1\}$$

Parameters A and B are the matrixes to be multiplied. I is an address stride through A. J is an address stride through B.

Parameter C is the result matrix. K is an address stride through C.

Parameter M is the row count for both A and C. Parameter N is the column count for both B and C. Parameter P is the column count for A and the row count for B.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mmulD

Performs an out-of-place multiplication of two matrices; double precision.

```
void vDSP_mmulD (
   double *__vDSP_a,
   vDSP_Stride __vDSP_aStride,
   double *__vDSP_b,
   vDSP_Stride __vDSP_bStride,
   double *__vDSP_c,
   vDSP_Stride __vDSP_cStride,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function multiplies an M-by-P matrix (A) by a P-by-N matrix (B) and stores the results in an M-by-N matrix (C).

This performs the following operation:

$$C_{(mN+n)K} = \sum_{p=0}^{P-1} A_{(mP+p)I} \bullet B_{(pN+n)J} \qquad n = \{0, N\text{-}1\} \text{ and } m = \{0, M\text{-}1\}$$

Parameters `A` and `B` are the matrixes to be multiplied. `I` is an address stride through `A`. `J` is an address stride through `B`.

Parameter `C` is the result matrix. `K` is an address stride through `C`.

Parameter `M` is the row count for both `A` and `C`. Parameter `N` is the column count for both `B` and `C`. Parameter `P` is the column count for `A` and the row count for `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mtrans

Creates a transposed matrix `C` from a source matrix `A`; single precision.

```
void vDSP_mtrans (
   float *__vDSP_a,
   vDSP_Stride __vDSP_aStride,
   float *__vDSP_c,
   vDSP_Stride __vDSP_cStride,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Discussion
This performs the following operation:

$$C_{(mN+n)K} = A_{(nM+m)I} \qquad n = \{0, N\text{-}1\} \text{ and } m = \{0, M\text{-}1\}$$

Parameter `A` is the source matrix. `I` is an address stride through the source matrix.

Parameter `C` is the resulting transposed matrix. `K` is an address stride through the result matrix.

Parameter `M` is the number of rows in `C` (and the number of columns in `A`).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mtransD

Creates a transposed matrix C from a source matrix A; double precision.

```
void vDSP_mtransD (
   double *__vDSP_a,
   vDSP_Stride __vDSP_aStride,
   double *__vDSP_c,
   vDSP_Stride __vDSP_cStride,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Discussion
This performs the following operation:

$$C_{(mN+n)K} = A_{(nM+m)I} \qquad n = \{0, N\text{-}1\} \text{ and } m = \{0, M\text{-}1\}$$

Parameter A is the source matrix. I is an address stride through the source matrix.

Parameter C is the resulting transposed matrix. K is an address stride through the result matrix.

Parameter M is the number of rows in C (and the number of columns in A).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_mvessq

Vector mean of signed squares; single precision.

```
void vDSP_mvessq (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Single-precision real input vector.

I
    Stride for A

C
    Output scalar

N
    Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni} \cdot \left|A_{ni}\right|$$

Finds the mean value of the signed squares of the elements of vector A and stores this value in *C. If N is zero (0), this function returns a NaN in *C. .

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_mvessqD

Vector mean of signed squares; double precision.

```
void vDSP_mvessqD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for A

C
Output scalar

N
Count

Discussion
This performs the following operation:

$$C = \frac{1}{N} \sum_{n=0}^{N-1} A_{ni} \cdot \left|A_{ni}\right|$$

Finds the mean value of the signed squares of the elements of vector A and stores this value in *C. If N is zero (0), this function returns a NaN in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_nzcros

Find zero crossings; single precision.

```
void vDSP_nzcros (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   vDSP_Length __vDSP_B,
   vDSP_Length *__vDSP_C,
   vDSP_Length *__vDSP_D,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for A

B

Maximum number of crossings to find

C

Index of last crossing found

D

Total number of zero crossings found

N

Count of elements in A

Discussion

This performs the following operation:

```
$d = c = 0$

For integer $n$,   $0 < n < N$;

  If $\text{sign}(A_{nI})! = \text{sign}(A_{(n-1)I})$ then

    $d = d+1$

    If  $d = b$  then

      $c = nI$,

      exit.      $n = \{1, N-1\}$
```

The "function" sign(x) above has the value -1 if the sign bit of x is 1 (x is negative or -0), and +1 if the sign bit is 0 (x is positive or +0).

Scans vector A to locate transitions from positive to negative values and from negative to positive values. The scan terminates when the number of crossings specified by B is found, or the end of the vector is reached. The zero-based index of the last crossing is returned in C. C is the actual array index, not the pre-stride index. If the zero crossing that B specifies is not found, zero is returned in C. The total number of zero crossings found is returned in D.

Note that a transition from -0 to +0 or from +0 to -0 is counted as a zero crossing.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_nzcrosD

Find zero crossings; double precision.

```
void vDSP_nzcrosD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   vDSP_Length __vDSP_B,
   vDSP_Length *__vDSP_C,
   vDSP_Length *__vDSP_D,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for `A`

B

Maximum number of crossings to find

C

Index of last crossing found

D

Total number of zero crossings found

N

Count of elements in `A`

Discussion

This performs the following operation:

```
$d = c = 0$
For integer $n$, $0 < n < N$;
  If $\text{sign}(A_{nI})! = \text{sign}(A_{(n-1)I})$ then
    $d = d+1$
    If $d = b$ then
      $c = nI$,
      exit.      $n = \{1, N-1\}$
```

The "function" sign(x) above has the value -1 if the sign bit of x is 1 (x is negative or -0), and +1 if the sign bit is 0 (x is positive or +0).

Scans vector A to locate transitions from positive to negative values and from negative to positive values. The scan terminates when the number of crossings specified by B is found, or the end of the vector is reached. The zero-based index of the last crossing is returned in C. C is the actual array index, not the pre-stride index. If the zero crossing that B specifies is not found, zero is returned in C. The total number of zero crossings found is returned in D.

Note that a transition from -0 to +0 or from +0 to -0 is counted as a zero crossing.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_polar

Rectangular to polar conversion; single precision.

```
void vDSP_polar (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A, must be even

C
: Single-precision output vector

K
: Stride for C, must be even

N
: Number of ordered pairs processed

Discussion
This performs the following operation:

$$C_{nK} = \sqrt{A_{nI}^2 + A_{nI+1}^2} \qquad C_{nK+1} = \operatorname{atan2}(A_{nI+1}, A_{nI}), \qquad n = \{0, N-1\}$$

Converts rectangular coordinates to polar coordinates. Cartesian (x,y) pairs are read from vector A. Polar (rho, theta) pairs, where rho is the radius and theta is the angle in the range [-pi, pi] are written to vector C. N specifies the number of coordinate pairs in A and C.

Coordinate pairs are adjacent elements in the array, regardless of stride; stride is the distance from one coordinate pair to the next.

This function performs the inverse operation of vDSP_rect, which converts polar to rectangular coordinates.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_polarD

Rectangular to polar conversion; double precision.

```
void vDSP_polarD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`, must be even

C
Double-precision output vector

K
Stride for `C`, must be even

N
Number of ordered pairs processed

Discussion
This performs the following operation:

$$C_{nK} = \sqrt{A_{nI}^2 + A_{nI+1}^2} \qquad C_{nK+1} = \operatorname{atan2}(A_{nI+1}, A_{nI}), \qquad n = \{0, N\text{-}1\}$$

Converts rectangular coordinates to polar coordinates. Cartesian (x,y) pairs are read from vector `A`. Polar (rho, theta) pairs, where rho is the radius and theta is the angle in the range [-pi, pi] are written to vector `C`. `N` specifies the number of coordinate pairs in `A` and `C`.

Coordinate pairs are adjacent elements in the array, regardless of stride; stride is the distance from one coordinate pair to the next.

This function performs the inverse operation of `vDSP_rectD`, which converts polar to rectangular coordinates.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_rect

Polar to rectangular conversion; single precision.

```
void vDSP_rect (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`, must be even

C

Single-precision real output vector

K

Stride for `C`, must be even

N

Number of ordered pairs processed

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} \bullet \cos(A_{nI+1}) \qquad C_{nK+1} = A_{nI} \bullet \sin(A_{nI+1}) \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Converts polar coordinates to rectangular coordinates. Polar (rho, theta) pairs, where rho is the radius and theta is the angle in the range [-pi, pi] are read from vector `A`. Cartesian (x,y) pairs are written to vector `C`. `N` specifies the number of coordinate pairs in `A` and `C`.

Coordinate pairs are adjacent elements in the array, regardless of stride; stride is the distance from one coordinate pair to the next.

This function performs the inverse operation of `vDSP_polar`, which converts rectangular to polar coordinates.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_rectD

Polar to rectangular conversion; double precision.

```
void vDSP_rectD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A, must be even

C

Double-precision real output vector

K

Stride for C, must be even

N

Number of ordered pairs processed

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} \cdot \cos(A_{nI+1}) \qquad C_{nK+1} = A_{nI} \cdot \sin(A_{nI+1}) \qquad n = \{0, N\text{-}1\}$$

Converts polar coordinates to rectangular coordinates. Polar (rho, theta) pairs, where rho is the radius and theta is the angle in the range [-pi, pi] are read from vector A. Cartesian (x,y) pairs are written to vector C. N specifies the number of coordinate pairs in A and C.

Coordinate pairs are adjacent elements in the array, regardless of stride; stride is the distance from one coordinate pair to the next.

This function performs the inverse operation of `vDSP_polarD`, which converts rectangular to polar coordinates.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_rmsqv

Vector root-mean-square; single precision.

```
void vDSP_rmsqv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Single-precision real output scalar

N

Count

Discussion

This performs the following operation:

$$C = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} A_{nI}^{2}}$$

Calculates the root mean square of the elements of A and stores the result in *C.

If N is zero (0), this function returns a NaN in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_rmsqvD

Vector root-mean-square; double precision.

```
void vDSP_rmsqvD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector.

I

Stride for A

C
: Double-precision real output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \sqrt{\frac{1}{N}\sum_{n=0}^{N-1} A_{nI}^{2}}$$

Calculates the root mean square of the elements of `A` and stores the result in `*C`.

If `N` is zero (`0`), this function returns a NaN in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_svdiv

Divide scalar by vector; single precision.

```
void vDSP_svdiv (
   float *__vDSP_A,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input scalar

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
This performs the following operation:

$$C_{nK} = \frac{A}{B_{nJ}}, \qquad n = \{0, N-1\}$$

Divides scalar `A` by each element of vector `B`, storing the results in vector `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_svdivD

Divide scalar by vector; double precision.

```
void vDSP_svdivD (
   double *__vDSP_A,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input scalar

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
This performs the following operation:

$$C_{nK} = \frac{A}{B_{nJ}}, \qquad n = \{0, N-1\}$$

Divides scalar `A` by each element of vector `B`, storing the results in vector `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_sve

Vector sum; single precision.

```
void vDSP_sve (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector.

I
: Stride for `A`

C
: Single-precision real output scalar

N
: Count

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI}$$

Writes the sum of the elements of `A` into `*C`. If `N` is zero (`0`), this function returns 0 in `*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_sveD

Vector sum; double precision.

```
void vDSP_sveD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C

Double-precision real output scalar

N

Count

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI}$$

Writes the sum of the elements of `A` into `*C`. If `N` is zero (`0`), this function returns 0 in `*C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_svemg

Vector sum of magnitudes; single precision.

```
void vDSP_svemg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for `A`

C

Single-precision real output scalar

N

Count

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} |A_{nI}|$$

Writes the sum of the magnitudes of the elements of `A` into `*C`.

If `N` is zero (`0`), this function returns 0 in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_svemgD

Vector sum of magnitudes; double precision.

```
void vDSP_svemgD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector.

I
: Stride for `A`

C
: Double-precision real output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} |A_{nI}|$$

Writes the sum of the magnitudes of the elements of `A` into `*C`.

If `N` is zero (`0`), this function returns 0 in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_svesq

Vector sum of squares; single precision.

```
void vDSP_svesq (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector.

I

Stride for A

C

Single-precision real output scalar

N

Count

Discussion

This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI}^2$$

Writes the sum of the squares of the elements of A into *C.

If N is zero (0), this function returns 0 in *C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_svesqD

Vector sum of squares; double precision.

```
void vDSP_svesqD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector.

I

Stride for A

C
: Double-precision real output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI}^{2}$$

Writes the sum of the squares of the elements of `A` into `C`.

If `N` is zero (`0`), this function returns 0 in `*C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_svs

Vector sum of signed squares; single precision.

```
void vDSP_svs (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector.

I
: Stride for `A`

C
: Single-precision real output scalar

N
: Count

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \bullet \left|A_{nI}\right|$$

Writes the sum of the signed squares of the elements of `A` into `*C`.

If N is zero (0), this function returns 0 in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_svsD

Vector sum of signed squares; double precision.

```
void vDSP_svsD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector.

I
Stride for A

C
Double-precision real output scalar

N
Count

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} \cdot |A_{nI}|$$

Writes the sum of the signed squares of the elements of A into *C.

If N is zero (0), this function returns 0 in *C.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_vaam

Vector add, add, and multiply; single precision.

```
void vDSP_vaam (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real input vector

K
: Stride for C

D
: Single-precision real input vector

L
: Stride for D

E
: Single-precision real output vector

M
: Stride for E

N
: Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$E_{nm} = (A_{ni} + B_{nj})(C_{nk} + D_{nl}) \qquad n = \{0, N\text{-}1\}$$

Multiplies the sum of vectors A and B by the sum of vectors C and D. Results are stored in vector E.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vaamD

Vector add, add, and multiply; double precision.

```
void vDSP_vaamD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Stride for B

C
: Double-precision real input vector

K
: Stride for C

D
: Double-precision real input vector

L
: Stride for D

E
: Double-precision real output vector

M
: Stride for E

N
: Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$E_{nm} = (A_{ni} + B_{nj})(C_{nk} + D_{nl}) \qquad \text{n} = \{0, \text{N-1}\}$$

Multiplies the sum of vectors A and B by the sum of vectors C and D. Results are stored in vector E.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vabs

Vector absolute values; single precision.

```
void vDSP_vabs (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Single-precision real input vector

I
  Stride for `A`

C
  Single-precision real output vector

K
  Stride for `C`

N
  Count

Discussion
This performs the following operation:

$$C_{nK} = |A_{nI}|, \qquad n = \{0, N-1\}$$

Writes the absolute values of the elements of `A` into corresponding elements of `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vabsD

Vector absolute values; double precision.

```
void vDSP_vabsD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = \left|A_{nI}\right|, \qquad n = \{0, N\text{-}1\}$$

Writes the absolute values of the elements of `A` into corresponding elements of `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vabsi

Integer vector absolute values.

```
void vDSP_vabsi (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input vector

I
: Stride for `A`

`C`
: Integer output vector

`K`
: Stride for `C`

`N`
: Count

Discussion

This performs the following operation:

$$C_{nK} = \left|A_{nI}\right| , \qquad n = \{0, N-1\}$$

Writes the absolute values of the elements of `A` into corresponding elements of `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vadd

Adds vector `A` to vector `B` and leaves the result in vector `C`; single precision.

```
void vDSP_vadd (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vaddD

Adds vector `A` to vector `B` and leaves the result in vector `C`; double precision.

```
void vDSP_vaddD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vam

Adds vectors `A` and `B`, multiplies the sum by vector `C`, and leaves the result in vector `D`; single precision.

```
void vDSP_vam (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   const float __vDSP_input3[],
   vDSP_Stride __vDSP_stride3,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$D_{nL} = (A_{nI} + B_{nJ})C_{nK} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vamD

Adds vectors `A` and `B`, multiplies the sum by vector `C`, and leaves the result in vector `D`; double precision.

```
void vDSP_vamD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   const double __vDSP_input3[],
   vDSP_Stride __vDSP_stride3,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion

This performs the following operation:

$$D_{nL} = (A_{nI} + B_{nJ})C_{nK} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vasbm

Vector add, subtract, and multiply; single precision.

```
void vDSP_vasbm (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input vector

J

Stride for `B`

C

Single-precision real input vector

K
Stride for `C`

D
Single-precision real input vector

L
Stride for `D`

E
Single-precision real output vector

M
Stride for `E`

N
Count ; each vector must have at least `N` elements

Discussion
This performs the following operation:

$$E_{nM} = (A_{nI} + B_{nJ})(C_{nK} - D_{nL}) \ , \qquad n = \{0, N-1\}$$

Multiplies the sum of vectors `A` and `B` by the result of subtracting vector `D` from vector `C`. Results are stored in vector `E`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vasbmD

Vector add, subtract, and multiply; double precision.

```
void vDSP_vasbmD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real input vector

K
: Stride for `C`

D
: Double-precision real input vector

L
: Stride for `D`

E
: Double-precision real output vector

M
: Stride for `E`

N
: Count ; each vector must have at least N elements

Discussion
This performs the following operation:

$$E_{nM} = (A_{nI} + B_{nJ})(C_{nK} - D_{nL}) \ , \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies the sum of vectors `A` and `B` by the result of subtracting vector `D` from vector `C`. Results are stored in vector `E`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vasm

Vector add and scalar multiply; single precision.

```
void vDSP_vasm (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input vector

J

Stride for `B`

C

Single-precision real input scalar

D

Single-precision real output vector

L

Stride for `D`

N

Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$D_{nM} = (A_{nI} + B_{nJ})\,c \;, \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies the sum of vectors `A` and `B` by scalar `C`. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vasmD

Vector add and scalar multiply; double precision.

```
void vDSP_vasmD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real input scalar

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$D_{nM} = (A_{nI} + B_{nJ})\, c \;, \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies the sum of vectors `A` and `B` by scalar `C`. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vavlin

Vector linear average; single precision.

```
void vDSP_vavlin (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input scalar

C

Single-precision real input-output vector

K

Stride for `C`

N

Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$C_{nK} = \frac{C_{nK}B + A_{nI}}{B + 1.0} \quad , \qquad n = \{0, N\text{-}1\}$$

Recalculates the linear average of input-output vector `C` to include input vector `A`. Input scalar `B` specifies the number of vectors included in the current average.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vavlinD

Vector linear average; double precision.

```
void vDSP_vavlinD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar

C
: Double-precision real input-output vector

K
: Stride for `C`

N
: Count ; each vector must have at least N elements

Discussion

This performs the following operation:

$$C_{nK} = \frac{C_{nK}B + A_{nI}}{B + 1.0} \quad , \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Recalculates the linear average of input-output vector `C` to include input vector `A`. Input scalar `B` specifies the number of vectors included in the current average.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vclip

Vector clip; single precision.

```
void vDSP_vclip (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input scalar: low clipping threshold

C
: Single-precision real input scalar: high clipping threshold

D
: Single-precision real output vector

L
: Stride for D

N
: Count

Discussion
This performs the following operation:

$$\begin{cases} \text{If } A_{nI} < b & \text{then } D_{nM} = B \\ \text{If } A_{nI} > c & \text{then } D_{nM} = C \ , \qquad n = \{0, N-1\} \\ \text{If } b \le A_{nI} \le c & \text{then } D_{nM} = A_{nI} \end{cases}$$

Elements of A are copied to D while clipping elements that are outside the interval [B, C] to the endpoints.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_vclipc

Vector clip and count; single precision.

```
void vDSP_vclipc (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N,
   vDSP_Length *__vDSP_NLOW,
   vDSP_Length *__vDSP_NHI
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input scalar: low clipping threshold

C
: Single-precision real input scalar: high clipping threshold

D
: Single-precision real output vector

L
: Stride for D

N
: Count of elements in A and D

NLOW
: Number of elements that were clipped to B

NHI
: Number of elements that were clipped to C

Discussion

This performs the following operation:

$$\begin{cases} \text{If } \; A_{nI} < b & \text{then } \; D_{nM} = B \\ \text{If } \; A_{nI} > c & \text{then } \; D_{nM} = C \;, \qquad n = \{0, N\text{-}1\} \\ \text{If } \; b \le A_{nI} \le c & \text{then } \; D_{nM} = A_{nI} \end{cases}$$

Elements of A are copied to D while clipping elements that are outside the interval [B, C] to the endpoints.

The count of elements clipped to B is returned in *NLOW, and the count of elements clipped to C is returned in *NHI

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vclipcD

Vector clip and count; double precision.

```
void vDSP_vclipcD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N,
   vDSP_Length *__vDSP_NLOW,
   vDSP_Length *__vDSP_NHI
);
```

Parameters

A
Double-precision real input vector

I
Stride for A

B
Double-precision real input scalar: low clipping threshold

C
Double-precision real input scalar: high clipping threshold

D
Double-precision real output vector

L
Stride for D

N
Count of elements in A and D

NLOW
Number of elements that were clipped to B

NHI
Number of elements that were clipped to C

Discussion

This performs the following operation:

$$\begin{cases} \text{If } A_{nI} < b & \text{then } D_{nM} = B \\ \text{If } A_{nI} > c & \text{then } D_{nM} = C \ , \qquad n = \{0, N\text{-}1\} \\ \text{If } b \le A_{nI} \le c & \text{then } D_{nM} = A_{nI} \end{cases}$$

Elements of A are copied to D while clipping elements that are outside the interval [B, C] to the endpoints.

The count of elements clipped to B is returned in *NLOW, and the count of elements clipped to C is returned in *NHI

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vclipD

Vector clip; double precision.

```
void vDSP_vclipD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar: low clipping threshold

C
: Double-precision real input scalar: high clipping threshold

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count

Discussion
This performs the following operation:

$$\begin{cases} \text{If } A_{nI} < b & \text{then } D_{nM} = B \\ \text{If } A_{nI} > c & \text{then } D_{nM} = C\ , \qquad n = \{0, \text{N-1}\} \\ \text{If } b \le A_{nI} \le c & \text{then } D_{nM} = A_{nI} \end{cases}$$

Elements of `A` are copied to `D` while clipping elements that are outside the interval [`B`, `C`] to the endpoints.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vclr

Vector clear; single precision.

```
void vDSP_vclr (
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

C

Single-precision real output vector

K

Stride for `C`

N

Count

Discussion
All elements of vector `C` are set to zeros.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vclrD

Vector clear; double precision.

```
void vDSP_vclrD (
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

C

Double-precision real output vector

K

Stride for `C`

N

Count

Discussion
All elements of vector `C` are set to zeros.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vcmprs

Vector compress; single precision.

```
void vDSP_vcmprs (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for A

B

Single-precision real input vector

J

Stride for B

C

Single-precision real output vector

K

Stride for C

N

Count

Discussion

Performs the following operation:

$p = 0$

If $B_{nJ} \neq 0.0$ then $C_{pK} = A_{nI}$; $p = p + 1$; $\quad$ n = {0, N-1}

Compresses vector A based on the nonzero values of gating vector B. For nonzero elements of B, corresponding elements of A are sequentially copied to output vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vcmprsD

Vector compress; double precision.

```
void vDSP_vcmprsD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$p = 0$

If $B_{nJ} \neq 0.0$ then $C_{pK} = A_{nI}$; $p = p + 1$; n = {0, N-1}

Compresses vector `A` based on the nonzero values of gating vector `B`. For nonzero elements of `B`, corresponding elements of `A` are sequentially copied to output vector `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdbcon

Vector convert power or amplitude to decibels; single precision.

```
void vDSP_vdbcon (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   unsigned int __vDSP_F
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input scalar: zero reference

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

F
: Power (0) or amplitude (1) flag

Discussion

Performs the following operation. is 20 if `F` is 1, or 10 if `F` is 0.

$$C_{nK} = \alpha\left(\log_{10}\left(\frac{A_{nI}}{B}\right)\right) \qquad n = \{0, N\text{-}1\}$$

Converts inputs from vector `A` to their decibel equivalents, calculated in terms of power or amplitude according to flag `F`. As a relative reference point, the value of input scalar `B` is considered to be zero decibels.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdbconD

Vector convert power or amplitude to decibels; double precision.

```
void vDSP_vdbconD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   unsigned int __vDSP_F
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input scalar: zero reference

C
: Double-precision real output vector

K
: Stride for C

N
: Count

F
: Power (0) or amplitude (1) flag

Discussion

Performs the following operation:

$$C_{nK} = \alpha\left(\log_{10}\left(\frac{A_{nI}}{B}\right)\right) \qquad n = \{0, N\text{-}1\}$$

The Greek letter alpha equals 20 if F = 1, and 10 if F = 0.

Converts inputs from vector A to their decibel equivalents, calculated in terms of power or amplitude according to flag F. As a relative reference point, the value of input scalar B is considered to be zero decibels.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdist

Vector distance; single precision.

```
void vDSP_vdist (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$C_{nk} = \sqrt{A_{ni}^2 + B_{nj}^2} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Computes the square root of the sum of the squares of corresponding elements of vectors A and B, and stores the result in the corresponding element of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vdistD

Vector distance; double precision.

```
void vDSP_vdistD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Stride for B

C
: Double-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$C_{nk} = \sqrt{A_{ni}^2 + B_{nj}^2} \qquad n = \{0, N-1\}$$

Computes the square root of the sum of the squares of corresponding elements of vectors A and B, and stores the result in the corresponding element of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdiv

Vector divide; single precision.

```
void vDSP_vdiv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = \frac{B_{nJ}}{A_{nI}} \qquad \mathrm{n} = \{0, \mathrm{N\text{-}1}\}$$

Divides elements of vector `B` by corresponding elements of vector `A`, and stores the results in corresponding elements of vector `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdivD

Vector divide; double precision.

```
void vDSP_vdivD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = \frac{B_{nJ}}{A_{nI}} \qquad n = \{0, N\text{-}1\}$$

Divides elements of vector `B` by corresponding elements of vector `A`, and stores the results in corresponding elements of vector `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdivi

Vector divide; integer.

```
void vDSP_vdivi (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_B,
   vDSP_Stride __vDSP_J,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input vector

I
: Stride for A

B
: Integer input vector

J
: Stride for B

C
: Integer output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = \frac{B_{nJ}}{A_{nI}} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Divides elements of vector A by corresponding elements of vector B, and stores the results in corresponding elements of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vdpsp

Vector convert double-precision to single-precision.

```
void vDSP_vdpsp (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Double-precision real input vector

I
    Stride for `A`

C
    Single-precision real output vector

K
    Stride for `C`

N
    Count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI}, \qquad n = \{0, N\text{-}1\}$$

Creates single-precision vector `C` by converting double-precision inputs from vector `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_venvlp

Vector envelope; single precision.

```
void vDSP_venvlp (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Single-precision real input vector: high envelope

I
: Stride for `A`

B
: Single-precision real input vector: low envelope

J
: Stride for `B`

C
: Single-precision real input vector

K
: Stride for `C`

D
: Single-precision real output vector

L
: Stride for `D`

N
: Count

Discussion

This performs the following operation:

$$\text{If } C_{nK} > A_{nI} \text{ or } C_{nK} < B_{nJ} \text{ then } D_{nM} = C_{nK}$$
$$\text{else } D_{nM} = 0.0 \qquad n = \{0, N-1\}$$

Finds the extrema of vector C. For each element of C, the corresponding element of A provides an upper-threshold value, and the corresponding element of B provides a lower-threshold value. If the value of an element of C falls outside the range defined by these thresholds, it is copied to the corresponding element of vector D. If its value is within the range, the corresponding element of vector D is set to zero.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_venvlpD

Vector envelope; double precision.

```
void vDSP_venvlpD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector: high envelope

I

Stride for `A`

B

Double-precision real input vector: low envelope

J

Stride for `B`

C

Double-precision real input vector

K

Stride for `C`

D

Double-precision real output vector

L

Stride for `D`

N

Count

Discussion

This performs the following operation:

$$\text{If} \quad C_{nK} > A_{nI} \quad \text{or} \quad C_{nK} < B_{nJ} \quad \text{then} \quad D_{nM} = C_{nK}$$

$$\text{else} \quad D_{nM} = 0.0 \qquad n = \{0, N\text{-}1\}$$

Finds the extrema of vector C. For each element of C, the corresponding element of A provides an upper-threshold value, and the corresponding element of B provides a lower-threshold value. If the value of an element of C falls outside the range defined by these thresholds, it is copied to the corresponding element of vector D. If its value is within the range, the corresponding element of vector D is set to zero.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_veqvi

Vector equivalence, 32-bit logical.

```
void vDSP_veqvi (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_B,
   vDSP_Stride __vDSP_J,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input vector

I
: Stride for `A`

B
: Integer input vector

J
: Stride for `B`

C
: Integer output vector

K
: Stride for `C`

N
: Count

Discussion

This performs the following operation:

$$C_{nk} = A_{ni} \bullet XNOR \bullet B_{nj} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Outputs the bitwise logical equivalence, exclusive NOR, of the integers of vectors `A` and `B`. For each pair of input values, bits in each position are compared. A bit in the output value is set if both input bits are set, or both are clear; otherwise it is cleared.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vfill

Vector fill; single precision.

```
void vDSP_vfill (
   float *__vDSP_A,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input scalar

C

Single-precision real output vector

K

Stride for `C`

N

Count

Discussion

Performs the following operation:

$$C_{nK} = A \qquad n = \{0, N\text{-}1\}$$

Sets each element of vector `C` to the value of `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vfillD

Vector fill; double precision.

```
void vDSP_vfillD (
   double *__vDSP_A,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input scalar

C

Double-precision real output vector

K

Stride for `C`

N

Count

Discussion
Performs the following operation:

$$C_{nK} = A \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Sets each element of vector `C` to the value of `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfilli

Integer vector fill.

```
void vDSP_vfilli (
   int *__vDSP_A,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input scalar

C
: Integer output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = A \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Sets each element of vector `C` to the value of `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix16

Converts an array of single-precision floating-point values to signed 16-bit integer values, rounding towards zero.

```
void vDSP_vfix16 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix16D

Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards zero.

```
void vDSP_vfix16D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix32

Converts an array of single-precision floating-point values to signed 32-bit integer values, rounding towards zero.

```
void vDSP_vfix32 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix32D

Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards zero.

```
void vDSP_vfix32D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix8

Converts an array of single-precision floating-point values to signed 8-bit integer values, rounding towards zero.

```
void vDSP_vfix8 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfix8D

Converts an array of double-precision floating-point values to signed 8-bit integer values, rounding towards zero.

```
void vDSP_vfix8D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr16

Converts an array of single-precision floating-point values to signed 16-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr16 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr16D

Converts an array of double-precision floating-point values to signed 16-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr16D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr32

Converts an array of single-precision floating-point values to signed 32-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr32 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr32D

Converts an array of double-precision floating-point values to signed 32-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr32D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr8

Converts an array of single-precision floating-point values to signed 8-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr8 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixr8D

Converts an array of double-precision floating-point values to signed 8-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixr8D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru16

Converts an array of single-precision floating-point values to unsigned 16-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru16 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru16D

Converts an array of double-precision floating-point values to unsigned 16-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru16D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru32

Converts an array of single-precision floating-point values to unsigned 32-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru32 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru32D

Converts an array of double-precision floating-point values to unsigned 32-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru32D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru8

Converts an array of single-precision floating-point values to unsigned 8-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru8 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixru8D

Converts an array of double-precision floating-point values to unsigned 8-bit integer values, rounding towards nearest integer.

```
void vDSP_vfixru8D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu16

Converts an array of single-precision floating-point values to unsigned 16-bit integer values, rounding towards zero.

```
void vDSP_vfixu16 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu16D

Converts an array of double-precision floating-point values to unsigned 16-bit integer values, rounding towards zero.

```
void vDSP_vfixu16D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned short *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu32

Converts an array of single-precision floating-point values to unsigned 32-bit integer values, rounding towards zero.

```
void vDSP_vfixu32 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu32D

Converts an array of double-precision floating-point values to unsigned 32-bit integer values, rounding towards zero.

```
void vDSP_vfixu32D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Source vector.

__vDSP_I
  Source vector stride length.

__vDSP_C
  Destination vector.

__vDSP_K
  Destination vector stride length.

__vDSP_N
  Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu8

Converts an array of single-precision floating-point values to unsigned 8-bit integer values, rounding towards zero.

```
void vDSP_vfixu8 (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Source vector.

__vDSP_I
  Source vector stride length.

__vDSP_C
  Destination vector.

__vDSP_K
  Destination vector stride length.

__vDSP_N
  Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfixu8D

Converts an array of double-precision floating-point values to unsigned 8-bit integer values, rounding towards zero.

```
void vDSP_vfixu8D (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   unsigned char *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt16

Converts an array of signed 16-bit integers to single-precision floating-point values.

```
void vDSP_vflt16 (
   short *A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt16D

Converts an array of signed 16-bit integers to double-precision floating-point values.

```
void vDSP_vflt16D (
   short *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt32

Converts an array of signed 32-bit integers to single-precision floating-point values.

```
void vDSP_vflt32 (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt32D

Converts an array of signed 32-bit integers to double-precision floating-point values.

```
void vDSP_vflt32D (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt8

Converts an array of signed 8-bit integers to single-precision floating-point values.

```
void vDSP_vflt8 (
   char *A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Source vector.

__vDSP_I
  Source vector stride length.

__vDSP_C
  Destination vector.

__vDSP_K
  Destination vector stride length.

__vDSP_N
  Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vflt8D

Converts an array of signed 8-bit integers to double-precision floating-point values.

```
void vDSP_vflt8D (
   char *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Source vector.

__vDSP_I
  Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu16

Converts an array of unsigned 16-bit integers to single-precision floating-point values.

```
void vDSP_vfltu16 (
   unsigned short *A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu16D

Converts an array of unsigned 16-bit integers to double-precision floating-point values.

```
void vDSP_vfltu16D (
   unsigned short *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu32

Converts an array of unsigned 32-bit integers to single-precision floating-point values.

```
void vDSP_vfltu32 (
   unsigned int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu32D

Converts an array of unsigned 32-bit integers to double-precision floating-point values.

```
void vDSP_vfltu32D (
   unsigned int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
Destination vector.

__vDSP_K
Destination vector stride length.

__vDSP_N
Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu8

Converts an array of unsigned 8-bit integers to single-precision floating-point values.

```
void vDSP_vfltu8 (
   unsigned char *A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Source vector.

__vDSP_I
Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfltu8D

Converts an array of unsigned 8-bit integers to double-precision floating-point values.

```
void vDSP_vfltu8D (
   unsigned char *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Source vector.

__vDSP_I
: Source vector stride length.

__vDSP_C
: Destination vector.

__vDSP_K
: Destination vector stride length.

__vDSP_N
: Number of elements in vector.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vfrac

Vector truncate to fraction; single precision.

```
void vDSP_vfrac (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = A_{nI} - \text{truncate}\,(A_{nI}) \qquad n = \{0, N\text{-}1\}$$

The "function" truncate(x) is the integer farthest from 0 but not farther than x. Thus, for example, `vDSP_vFrac(-3.25)` produces the result -0.25.

Sets each element of vector `C` to the signed fractional part of the corresponding element of `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vfracD

Vector truncate to fraction; double precision.

```
void vDSP_vfracD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = A_{nI} - \text{truncate}\,(A_{nI}) \qquad n = \{0, N\text{-}1\}$$

The "function" truncate(x) is the integer farthest from 0 but not farther than x. Thus, for example, `vDSP_vFrac(-3.25)` produces the result -0.25.

Sets each element of vector `C` to the signed fractional part of the corresponding element of `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vgathr

Vector gather; single precision.

```
void vDSP_vgathr (
   float *__vDSP_A,
   vDSP_Length *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

B
: Integer vector containing indices

J
: Stride for `B`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = A_{B_{nJ}} \qquad n = \{0, N-1\}$$

Uses elements of vector `B` as indices to copy selected elements of vector `A` to sequential locations in vector `C`. Note that 1, not zero, is treated as the first location in the input vector when evaluating indices. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vgathra

Vector gather, absolute pointers; single precision.

```
void vDSP_vgathra (
   float **A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Pointer input vector

I
: Stride for `A`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = *(A_{nI}) \qquad n = \{0, N-1\}$$

Uses elements of vector `A` as pointers to copy selected single-precision values from memory to sequential locations in vector `C`. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vgathraD

Vector gather, absolute pointers; double precision.

```
void vDSP_vgathraD (
   double **A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Pointer input vector

I
: Stride for `A`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = *(A_{nI}) \qquad n = \{0, N\text{-}1\}$$

Uses elements of vector `A` as pointers to copy selected double-precision values from memory to sequential locations in vector `C`. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vgathrD

Vector gather; double precision.

```
void vDSP_vgathrD (
   double *__vDSP_A,
   vDSP_Length *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

B
: Integer vector containing indices

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = A_{B_{nJ}} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Uses elements of vector `B` as indices to copy selected elements of vector `A` to sequential locations in vector `C`. Note that 1, not zero, is treated as the first location in the input vector when evaluating indices. This function can only be done out of place.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vgen

Vector tapered ramp; single precision.

```
void vDSP_vgen (
   float *__vDSP_A,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input scalar: base value

B

Single-precision real input scalar: end value

C

Single-precision real output vector

K

Stride for `C`

N

Count

Discussion

Performs the following operation:

$$C_{nK} = A + \frac{n(B-A)}{N-1} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Creates ramped vector `C` with element zero equal to scalar `A` and element N-1 equal to scalar `B`. Output values between element zero and element N-1 are evenly spaced and increase or decrease monotonically.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vgenD

Vector tapered ramp; double precision.

```
void vDSP_vgenD (
   double *__vDSP_A,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input scalar: base value

B

Double-precision real input scalar: end value

C

Double-precision real output vector

K

Stride for `C`

N

Count

Discussion

Performs the following operation:

$$C_{nK} = A + \frac{n(B-A)}{N-1} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Creates ramped vector `C` with element zero equal to scalar `A` and element N-1 equal to scalar `B`. Output values between element zero and element N-1 are evenly spaced and increase or decrease monotonically.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vgenp

Vector generate by extrapolation and interpolation; single precision.

```
void vDSP_vgenp (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input vector

J

Stride for `B`

C

Single-precision real output vector

K

Stride for `C`

N
: Count for C

M
: Count for A and B

Discussion
Performs the following operation:

$$C_{nK} = A_0 \qquad \text{for } 0 \le n \le \text{trunc}(B_0)$$

$$C_{nK} = A_{[M-1]I} \qquad \text{for } \text{trunc}(B_{[M-1]J}) < n \le N-1$$

$$C_{nK} = A_{mI} + \frac{A_\Delta\ (n-B_{mJ})}{B_\Delta} \qquad \text{for } \text{trunc}(B_{mJ}) < n \le \ \text{trunc}(B_{[m+1]J})$$

$$\text{where:}\quad A_\Delta = A_{[m+1]I} - A_{mI}$$

$$B_\Delta = B_{[m+1]J} - B_{mI} \qquad \text{m} = \{0, \text{M-2}\}$$

Generates vector C by extrapolation and linear interpolation from the ordered pairs (A,B) provided by corresponding elements in vectors A and B. Vector B provides index values and should increase monotonically. Vector A provides intensities, magnitudes, or some other measurable quantities, one value associated with each value of B. This function can only be done out of place.

Vectors A and B define a piecewise linear function, f(x):

- In the interval [-infinity, trunc(B[0*J]], the function is the constant A[0*I].
- In each interval (trunc(B[m*J]), trunc(B[(m+1)*J])], the function is the line passing through the two points (B[m*J], A[m*I]) and (B[(m+1)*J], A[(m+1)*I]). (This is for each integer m, 0 <= m < M-1.)
- In the interval (B[(M-1)*J], infinity], the function is the constant A[(M-1)*I].
- For 0 <= n < N, C[n*K] = f(n).

This function can only be done out of place.

Output values are generated for integral indices in the range zero through N - 1, deriving output values by interpolating and extrapolating from vectors A and B. For example, if vectors A and B define velocity and time pairs (v, t), vDSP_vgenp writes one velocity to vector C for every integral unit of time from zero to N - 1.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_vgenpD

Vector generate by extrapolation and interpolation; double precision.

```
void vDSP_vgenpD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
  Double-precision real input vector

I
  Stride for A

B
  Double-precision real input vector

J
  Stride for B

C
  Double-precision real output vector

K
  Stride for C

N
  Count for C

M
  Count for A and B

Discussion

Performs the following operation:

$$C_{nK} = A_0 \qquad \text{for } 0 \le n \le \text{trunc}(B_0)$$

$$C_{nK} = A_{[M-1]I} \qquad \text{for } \text{trunc}(B_{[M-1]J}) < n \le N-1$$

$$C_{nK} = A_{mI} + \frac{A_\Delta (n - B_{mJ})}{B_\Delta} \qquad \text{for } \text{trunc}(B_{mJ}) < n \le \text{trunc}(B_{[m+1]J})$$

$$\text{where:} \quad A_\Delta = A_{[m+1]I} - A_{mI}$$

$$B_\Delta = B_{[m+1]J} - B_{mI} \qquad \text{m} = \{0, \text{M-2}\}$$

Generates vector C by extrapolation and linear interpolation from the ordered pairs (A,B) provided by corresponding elements in vectors A and B. Vector B provides index values and should increase monotonically. Vector A provides intensities, magnitudes, or some other measurable quantities, one value associated with each value of B. This function can only be done out of place.

Vectors A and B define a piecewise linear function, f(x):

- In the interval [-infinity, trunc(B[0*J]], the function is the constant A[0*I].

- In each interval (trunc(B[m*J]), trunc(B[(m+1)*J])], the function is the line passing through the two points (B[m*J], A[m*I]) and (B[(m+1)*J], A[(m+1)*I]). (This is for each integer m, 0 <= m < M-1.)
- In the interval (B[(M-1)*J], infinity], the function is the constant A[(M-1)*I].
- For 0 <= n < N, C[n*K] = f(n).

This function can only be done out of place.

Output values are generated for integral indices in the range zero through N - 1, deriving output values by interpolating and extrapolating from vectors A and B. For example, if vectors A and B define velocity and time pairs (v, t), vDSP_vgenp writes one velocity to vector C for every integral unit of time from zero to N - 1.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_viclip

Vector inverted clip; single precision.

```
void vDSP_viclip (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for A

B
Single-precision real input scalar: lower threshold

C
Single-precision real input scalar: upper threshold

D
Single-precision real output vector

L
Stride for D

N
Count

Discussion
Performs the following operation:

$$D_{nj} = A_{ni} \quad \text{if} \quad A_{ni} \leq b$$

$$D_{nj} = A_{ni} \quad \text{if} \quad A_{ni} \geq c$$

$$D_{nj} = b \quad \text{if} \quad b < A_{ni} < 0.0$$

$$D_{nj} = c \quad \text{if} \quad 0.0 \leq A_{ni} < c \qquad n = \{0, N\text{-}1\}$$

Performs an inverted clip of vector `A` using lower-threshold and upper-threshold input scalars `B` and `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_viclipD

Vector inverted clip; double precision.

```
void vDSP_viclipD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`

B
Double-precision real input scalar: lower threshold

C
Double-precision real input scalar: upper threshold

D
Double-precision real output vector

L
Stride for `D`

N
Count

Discussion
Performs the following operation:

$$D_{nj} = A_{ni} \quad \text{if} \quad A_{ni} \le b$$

$$D_{nj} = A_{ni} \quad \text{if} \quad A_{ni} \ge c$$

$$D_{nj} = b \quad \text{if} \quad b < A_{ni} < 0.0$$

$$D_{nj} = c \quad \text{if} \quad 0.0 \le A_{ni} < c \qquad \text{n} = \{0, \text{N-1}\}$$

Performs an inverted clip of vector `A` using lower-threshold and upper-threshold input scalars `B` and `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vindex

Vector index; single precision.

```
void vDSP_vindex (
   float *__vDSP_A,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

B
: Single-precision real input vector: indices

J
: Stride for `B`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = A_{\text{truncate}(B_{nJ})} \qquad \text{n} = \{0, \text{N-1}\}$$

Uses vector `B` as zero-based subscripts to copy selected elements of vector `A` to vector `C`. Fractional parts of vector `B` are ignored.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vindexD

Vector index; double precision.

```
void vDSP_vindexD (
   double *__vDSP_A,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

B
Double-precision real input vector: indices

J
Stride for `B`

C
Double-precision real output vector

K
Stride for `C`

N
Count

Discussion
Performs the following operation:

$$C_{nK} = A_{\text{truncate}(B_{nJ})} \qquad \text{n} = \{0, \text{N-1}\}$$

Uses vector `B` as zero-based subscripts to copy selected elements of vector `A` to vector `C`. Fractional parts of vector `B` are ignored.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vintb

Vector linear interpolation between vectors; single precision.

```
void vDSP_vintb (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real input scalar: interpolation constant

D
: Single-precision real output vector

L
: Stride for `D`

N
: Count

Discussion

This performs the following operation:

$$D_{nK} = A_{nI} + C[B_{nJ} - A_{nI}] \qquad \mathrm{n} = \{0, \mathrm{N\text{-}1}\}$$

Creates vector `D` by interpolating between vectors `A` and `B`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vintbD

Vector linear interpolation between vectors; double precision.

```
void vDSP_vintbD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input vector

J

Stride for B

C

Double-precision real input scalar: interpolation constant

D

Double-precision real output vector

L

Stride for D

N

Count

Discussion

This performs the following operation:

$$D_{nK} = A_{nI} + C[B_{nJ} - A_{nI}] \qquad n = \{0, N-1\}$$

Creates vector D by interpolating between vectors A and B.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vlim

Vector test limit; single precision.

```
void vDSP_vlim (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for A

B

Single-precision real input scalar: limit

C

Single-precision real input scalar

D

Single-precision real output vector

L

Stride for D

N

Count

Discussion

Compares values from vector A to limit scalar B. For inputs greater than or equal to B, scalar C is written to D . For inputs less than B, the negated value of scalar C is written to vector D.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vlimD

Vector test limit; double precision.

```
void vDSP_vlimD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input scalar: limit

C

Double-precision real input scalar

D

Double-precision real output vector

L

Stride for D

N

Count

Discussion

Compares values from vector A to limit scalar B. For inputs greater than or equal to B, scalar C is written to D . For inputs less than B, the negated value of scalar C is written to vector D.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vlint

Vector linear interpolation between neighboring elements; single precision.

```
void vDSP_vlint (
   float *__vDSP_A,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Single-precision real input vector

B
: Single-precision real input vector: integer parts are indices into A and fractional parts are interpolation constants

J
: Stride for B

C
: Single-precision real output vector

K
: Stride for C

N
: Count for C

M
: Length of A

Discussion
Performs the following operation:

$$C_{nK} = A_{\beta} + \alpha(A_{\beta+1} - A_{\beta}) \qquad n = \{0, N\text{-}1\}$$

$$\text{where: } \beta = \text{trunc}(B_{nJ})$$
$$\alpha = B_{nJ} - \text{float}(\beta)$$

Generates vector C by interpolating between neighboring values of vector A as controlled by vector B. The integer portion of each element in B is the zero-based index of the first element of a pair of adjacent values in vector A.

The value of the corresponding element of C is derived from these two values by linear interpolation, using the fractional part of the value in B.

Argument M is not used in the calculation. However, the integer parts of the values in B must be greater than or equal to zero and less than or equal to M - 2.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vlintD

Vector linear interpolation between neighboring elements; double precision.

```
void vDSP_vlintD (
   double *__vDSP_A,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Double-precision real input vector

B
: Double-precision real input vector: integer parts are indices into `A` and fractional parts are interpolation constants

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count for C

M
: Length of A

Discussion

Performs the following operation:

$$C_{nK} = A_{\beta} + \alpha(A_{\beta+1} - A_{\beta}) \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

$$\text{where: } \beta = \mathrm{trunc}(B_{nJ})$$
$$\alpha = B_{nJ} - \mathrm{float}(\beta)$$

Generates vector `C` by interpolating between neighboring values of vector `A` as controlled by vector `B`. The integer portion of each element in `B` is the zero-based index of the first element of a pair of adjacent values in vector `A`.

The value of the corresponding element of `C` is derived from these two values by linear interpolation, using the fractional part of the value in B.

Argument `M` is not used in the calculation. However, the integer parts of the values in `B` must be greater than or equal to zero and less than or equal to `M` - 2.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vma

Vector multiply and add; single precision.

```
void vDSP_vma (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Single-precision real input vector

I
    Stride for `A`

B
    Single-precision real input vector

J
    Stride for `B`

C
    Single-precision real input vector

K
    Stride for `C`

D
    Single-precision real output vector

L
    Stride for `D`

N
    Count

Discussion
This performs the following operation:

$$D_{nM} = A_{nI} \cdot B_{nJ} + C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Multiplies corresponding elements of vectors `A` and `B`, add the corresponding elements of vector `C`, and stores the results in vector `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmaD

Vector multiply and add; double precision.

```
void vDSP_vmaD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real input vector

K
: Stride for `C`

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count

Discussion
This performs the following operation:

$$D_{nM} = A_{nI} \cdot B_{nJ} + C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies corresponding elements of vectors `A` and `B`, add the corresponding elements of vector `C`, and stores the results in vector `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmax

Vector maxima; single precision.

```
void vDSP_vmax (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for `A`

B
Single-precision real input vector

J
Stride for `B`

C
Single-precision real output vector

K
Stride for `C`

N
Count

Discussion
This performs the following operation:

$$\text{If} \quad A_{nI} \geq B_{nJ} \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B_{nJ} \qquad n = \{0, N-1\}$$

Each element of output vector `C` is the greater of the corresponding values from input vectors `A` and `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmaxD

Vector maxima; double precision.

```
void vDSP_vmaxD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for `A`

B

Double-precision real input vector

J

Stride for `B`

C

Double-precision real output vector

K

Stride for `C`

N

Count

Discussion

This performs the following operation:

$$\text{If} \quad A_{nI} \geq B_{nJ} \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B_{nJ} \qquad n = \{0, N-1\}$$

Each element of output vector `C` is the greater of the corresponding values from input vectors `A` and `B`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmaxmg

Vector maximum magnitudes; single precision.

```
void vDSP_vmaxmg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for `A`

B
Single-precision real input vector

J
Stride for `B`

C
Single-precision real output vector

K
Stride for `C`

N
Count

Discussion
This performs the following operation:

$$\text{If} \quad |A_{nI}| \geq |B_{nJ}| \quad \text{then} \quad C_{nK} = |A_{nI}| \quad \text{else} \quad C_{nK} = |B_{nJ}| \qquad n = \{0, N-1\}$$

Each element of output vector `C` is the larger of the magnitudes of corresponding values from input vectors `A` and `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmaxmgD

Vector maximum magnitudes; double precision.

```
void vDSP_vmaxmgD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Stride for B

C
: Double-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$\text{If} \quad |A_{nI}| \geq |B_{nJ}| \quad \text{then} \quad C_{nK} = |A_{nI}| \quad \text{else} \quad C_{nK} = |B_{nJ}| \qquad n = \{0, N-1\}$$

Each element of output vector C is the larger of the magnitudes of corresponding values from input vectors A and B.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vmin

Vector minima; single precision.

```
void vDSP_vmin (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$\text{If} \quad A_{nI} \leq B_{nJ} \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Each element of output vector C is the lesser of the corresponding values from input vectors A and B.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vminD

Vector minima; double precision.

```
void vDSP_vminD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for A

B
Double-precision real input vector

J
Stride for B

C
Double-precision real output vector

K
Stride for C

N
Count

Discussion

This performs the following operation:

$$\text{If} \quad A_{nI} \le B_{nJ} \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Each element of output vector C is the lesser of the corresponding values from input vectors A and B.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vminmg

Vector minimum magnitudes; single precision.

```
void vDSP_vminmg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
This performs the following operation:

$$\text{If} \quad |A_{nI}| \leq |B_{nJ}| \quad \text{then} \quad C_{nK} = |A_{nI}| \quad \text{else} \quad C_{nK} = |B_{nJ}| \qquad n = \{0, N\text{-}1\}$$

Each element of output vector `C` is the smaller of the magnitudes of corresponding values from input vectors `A` and `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vminmgD

Vector minimum magnitudes; double precision.

```
void vDSP_vminmgD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Stride for B

C
: Double-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$\text{If} \quad |A_{nI}| \le |B_{nJ}| \quad \text{then} \quad C_{nK} = |A_{nI}| \quad \text{else} \quad C_{nK} = |B_{nJ}| \qquad n = \{0, N-1\}$$

Each element of output vector C is the smaller of the magnitudes of corresponding values from input vectors A and B.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmma

Vector multiply, multiply, and add; single precision.

```
void vDSP_vmma (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for A

B

Single-precision real input vector

J

Stride for B

C

Single-precision real input vector

K

Stride for C

D

Single-precision real input vector

L

Stride for D

E

Single-precision real output vector

M

Stride for E

N

Count

Discussion

This performs the following operation:

$$E_{nM} = A_{nI} \bullet B_{nJ} + C_{nK} \bullet D_{nL} \qquad \text{n} = \{0, \text{N-1}\}$$

Corresponding elements of A and B are multiplied, corresponding values of C and D are multiplied, and these products are added together and stored in E.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmmaD

Vector multiply, multiply, and add; double precision.

```
void vDSP_vmmaD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real input vector

K
: Stride for `C`

D
: Double-precision real input vector

L
: Stride for `D`

E
: Double-precision real output vector

M
: Stride for `E`

N
: Count

Discussion
This performs the following operation:

$$E_{nM} = A_{nI} \bullet B_{nJ} + C_{nK} \bullet D_{nL} \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Corresponding elements of A and B are multiplied, corresponding values of C and D are multiplied, and these products are added together and stored in E.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_vmmsb

Vector multiply, multiply, and subtract; single precision.

```
void vDSP_vmmsb (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real input vector

K
: Stride for C

D
: Single-precision real input vector

L
: Stride for D

E
: Single-precision real output vector

M
: Stride for E

N
Count

Discussion

This performs the following operation:

$$E_{nM} = A_{nI} B_{nJ} - C_{nK} D_{nL} \qquad n = \{0, N-1\}$$

Corresponding elements of `A` and `B` are multiplied, corresponding values of `C` and `D` are multiplied, and the second product is subtracted from the first. The result is stored in `E`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmmsbD

Vector multiply, multiply, and subtract; double precision.

```
void vDSP_vmmsbD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`

B
Double-precision real input vector

J
Stride for `B`

C
Double-precision real input vector

K
Stride for `C`

D
Double-precision real input vector

L
: Stride for `D`

E
: Double-precision real output vector

M
: Stride for `E`

N
: Count

Discussion

This performs the following operation:

$$E_{nM} = A_{nI} B_{nJ} - C_{nK} D_{nL} \qquad n = \{0, N\text{-}1\}$$

Corresponding elements of `A` and `B` are multiplied, corresponding values of `C` and `D` are multiplied, and the second product is subtracted from the first. The result is stored in `E`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmsa

Vector multiply and scalar add; single precision.

```
void vDSP_vmsa (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real input scalar

D
: Single-precision real output vector

L
: Stride for D

N
: Count

Discussion

This performs the following operation:

$$D_{nK} = A_{nI} \cdot B_{nJ} + C \qquad n = \{0, N\text{-}1\}$$

Corresponding elements of A and B are multiplied and the scalar C is added. The result is stored in D.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmsaD

Vector multiply and scalar add; double precision.

```
void vDSP_vmsaD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Stride for B

C
: Double-precision real input scalar

D
: Double-precision real output vector

L
: Stride for D

N

Count

Discussion

This performs the following operation:

$$D_{nK} = A_{nI} \cdot B_{nJ} + C \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Corresponding elements of `A` and `B` are multiplied and the scalar `C` is added. The result is stored in `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmsb

Vector multiply and subtract, single precision.

```
void vDSP_vmsb (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input vector

J

Stride for `B`

C

Single-precision real input vector

K

Stride for `C`

D

Single-precision real output vector

L

Stride for `D`

N
: Count

Discussion

This performs the following operation:

$$D_{nM} = A_{nI} \cdot B_{nJ} - C_{nK} \qquad n = \{0, N\text{-}1\}$$

Corresponding elements of `A` and `B` are multiplied and the corresponding value of `C` is subtracted. The result is stored in `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vmsbD

Vector multiply and subtract; double precision.

```
void vDSP_vmsbD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real input vector

K
: Stride for `C`

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count

Discussion
This performs the following operation:

$$D_{nM} = A_{nI} \cdot B_{nJ} - C_{nK} \qquad n = \{0, N\text{-}1\}$$

Corresponding elements of `A` and `B` are multiplied and the corresponding value of `C` is subtracted. The result is stored in `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmul

Multiplies vector `A` by vector `B` and leaves the result in vector `C`; single precision.

```
void vDSP_vmul (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex output count

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vmulD

Multiplies vector `A` by vector `B` and leaves the result in vector `C`; double precision.

```
void vDSP_vmulD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex output count

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vnabs

Vector negative absolute values; single precision.

```
void vDSP_vnabs (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = -\left|A_{nI}\right| \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Each value in C is the negated absolute value of the corresponding element in A.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vnabsD

Vector negative absolute values; double precision.

```
void vDSP_vnabsD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

C

Double-precision real output vector

K

Stride for `C`

N

Count

Discussion

This performs the following operation:

$$C_{nK} = -\left|A_{nI}\right| \qquad n = \{0, N\text{-}1\}$$

Each value in `C` is the negated absolute value of the corresponding element in `A`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vneg

Vector negative values; single precision.

```
void vDSP_vneg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

C

Single-precision real output vector

K

Stride for `C`

N

Count

Discussion

Each value in `C` is the negated value of the corresponding element in `A`.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vnegD

Vector negative values; double precision.

```
void vDSP_vnegD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Double-precision real input vector

I
  Stride for `A`

C
  Double-precision real output vector

K
  Stride for `C`

N
  Count

Discussion
Each value in `C` is the negated value of the corresponding element in `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vpoly

Vector polynomial evaluation; single precision.

```
void vDSP_vpoly (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Parameters

A

Single-precision real input vector: coefficients

I

Stride for `A`

B

Single-precision real input vector: variable values

J

Stride for `B`

C

Single-precision real output vector

K

Stride for `C`

N

Count

P

Degree of polynomial

Discussion

This performs the following operation:

$$C_{nK} = \sum_{p=0}^{P} A_{pI} \cdot B_{nJ}^{P-p} \qquad n = \{0, N\text{-}1\}$$

Evaluates polynomials using vector `B` as independent variables and vector `A` as coefficients. A polynomial of degree p requires p+1 coefficients, so vector `A` should contain `P`+1 values.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vpolyD

Vector polynomial evaluation; double precision.

```
void vDSP_vpolyD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Parameters

A
: Double-precision real input vector: coefficients

I
: Stride for `A`

B
: Double-precision real input vector: variable values

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

P
: Degree of polynomial

Discussion

This performs the following operation:

$$C_{nK} = \sum_{p=0}^{P} A_{pI} \cdot B_{nJ}^{P-p} \qquad n = \{0, N\text{-}1\}$$

Evaluates polynomials using vector `B` as independent variables and vector `A` as coefficients. A polynomial of degree p requires p+1 coefficients, so vector `A` should contain `P`+1 values.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vpythg

Vector Pythagoras; single precision.

```
void vDSP_vpythg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real input vector

K
: Stride for C

D
: Single-precision real input vector

L
: Stride for D

E
: Single-precision real output vector

M
: Stride for E

N
: Count

Discussion

This performs the following operation:

$$E_{nM} = \sqrt{(A_{nI} - C_{nK})^2 + (B_{nJ} - D_{nL})^2} \qquad n = \{0, N-1\}$$

Subtracts vector C from A and squares the differences, subtracts vector D from B and squares the differences, adds the two sets of squared differences, and then writes the square roots of the sums to vector E.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vpythgD

Vector Pythagoras; double precision.

```
void vDSP_vpythgD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Double-precision real input vector

I
  Stride for A

B
  Double-precision real input vector

J
  Stride for B

C
  Double-precision real input vector

K
  Stride for C

D
  Double-precision real input vector

L
  Stride for D

E
  Double-precision real output vector

M
  Stride for E

N
  Count

Discussion
This performs the following operation:

$$E_{nM} = \sqrt{(A_{nI} - C_{nK})^2 + (B_{nJ} - D_{nL})^2} \qquad n = \{0, N\text{-}1\}$$

Subtracts vector C from A and squares the differences, subtracts vector D from B and squares the differences, adds the two sets of squared differences, and then writes the square roots of the sums to vector E.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vqint

Vector quadratic interpolation; single precision.

```
void vDSP_vqint (
   float *__vDSP_A,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Single-precision real input vector

B
: Single-precision real input vector: integer parts are indices into A and fractional parts are interpolation constants

J
: Stride for B

C
: Single-precision real output vector

K
: Stride for C

N
: Count for C

M
: Length of A: must be greater than or equal to 3

Discussion
This performs the following operation:

$$C_{nK} = \frac{A_{\beta-1}[\alpha^2-\alpha] + A_{\beta}[2.0-2.0\alpha^2] + A_{\beta+1}[\alpha^2+\alpha]}{2}$$

where: $\beta = \max(\mathrm{trunc}(B_{nJ}), 1)$ $\quad n = \{0, N\text{-}1\}$

$\alpha = B_{nJ} - \mathrm{float}(\beta)$

Generates vector C by interpolating between neighboring values of vector A as controlled by vector B. The integer portion of each element in B is the zero-based index of the first element of a triple of adjacent values in vector A.

The value of the corresponding element of C is derived from these three values by quadratic interpolation, using the fractional part of the value in B.

Argument M is not used in the calculation. However, the integer parts of the values in B must be less than or equal to M - 2.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vqintD

Vector quadratic interpolation; double precision.

```
void vDSP_vqintD (
   double *__vDSP_A,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
Double-precision real input vector

B
Double-precision real input vector: integer parts are indices into A and fractional parts are interpolation constants

J
Stride for B

C
Double-precision real output vector

K
Stride for C

N
Count for C

M
Length of A: must be greater than or equal to 3

Discussion
This performs the following operation:

$$C_{nK} = \frac{A_{\beta-1}[\alpha^2 - \alpha] + A_{\beta}[2.0 - 2.0\alpha^2] + A_{\beta+1}[\alpha^2 + \alpha]}{2}$$

where: $\beta = \max(\mathrm{trunc}(B_{nJ}), 1)$ $\quad$ $n = \{0, N\text{-}1\}$

$\alpha = B_{nJ} - \mathrm{float}(\beta)$

Generates vector C by interpolating between neighboring values of vector A as controlled by vector B. The integer portion of each element in B is the zero-based index of the first element of a triple of adjacent values in vector A.

The value of the corresponding element of C is derived from these three values by quadratic interpolation, using the fractional part of the value in B.

Argument M is not used in the calculation. However, the integer parts of the values in B must be less than or equal to M - 2.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vramp

Build ramped vector; single precision.

```
void vDSP_vramp (
   float *__vDSP_A,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input scalar: initial value

B
Single-precision real input scalar: increment or decrement

C
Single-precision real output vector

K
Stride for C

N
Count

Discussion
Performs the following operation:

$$C_{nk} = a + nb \qquad n = \{0, N\text{-}1\}$$

Creates a monotonically incrementing or decrementing vector. Scalar `A` is the initial value written to vector `C`. Scalar `B` is the increment or decrement for each succeeding element.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampD

Build ramped vector; double precision.

```
void vDSP_vrampD (
   double *__vDSP_A,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input scalar: initial value

B
: Double-precision real input scalar: increment or decrement

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nk} = a + nb \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Creates a monotonically incrementing or decrementing vector. Scalar `A` is the initial value written to vector `C`. Scalar `B` is the increment or decrement for each succeeding element.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmul

Builds a ramped vector and multiplies by a source vector.

```
void vDSP_vrampmul (
   const float *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   float *__vDSP_Start,
   const float *__vDSP_Step,
   float *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
: Input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O
: The output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion
This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] = *Start * I[i*IS];
    *Start += *Step;
}
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmul2

Stereo version of `vDSP_vrampmul` (page 437).

```
void vDSP_vrampmul2 (
   const float *__vDSP_I0,
   const float *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   float *__vDSP_Start,
   const float *__vDSP_Step,
   float *__vDSP_O0,
   float *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
: First input vector, multiplied by the ramp function.

__vDSP_I1
: Second input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O0
: First output vector.

__vDSP_O1
: Second output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] = *Start * I0[i*IS];
    O1[i*OS] = *Start * I1[i*IS];
    *Start += *Step;
}
```

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmul2_s1_15

Vector fixed-point 1.15 format version of `vDSP_vrampmul2` (page 438).

```
void vDSP_vrampmul2_s1_15 (
   const short int *__vDSP_I0,
   const short int *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   short int *__vDSP_Start,
   const short int *__vDSP_Step,
   short int *__vDSP_O0,
   short int *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
: First input vector, multiplied by the ramp function.

__vDSP_I1
: Second input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O0
: First output vector.

__vDSP_O1
: Second output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] = *Start * I0[i*IS];
    O1[i*OS] = *Start * I1[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by `32768.0`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmul2_s8_24

Vector fixed-point 8.24 format version of `vDSP_vrampmul2` (page 438).

```
void vDSP_vrampmul2_s8_24 (
   const int *__vDSP_I0,
   const int *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   int *__vDSP_Start,
   const int *__vDSP_Step,
   int *__vDSP_O0,
   int *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
: First input vector, multiplied by the ramp function.

__vDSP_I1
: Second input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O0
: First output vector.

__vDSP_O1
: Second output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] = *Start * I0[i*IS];
    O1[i*OS] = *Start * I1[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by `16777216.0`.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmuladd

Builds a ramped vector, multiplies it by a source vector, and adds the result to the output vector.

```
void vDSP_vrampmuladd (
   const float *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   float *__vDSP_Start,
   const float *__vDSP_Step,
   float *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
Input vector, multiplied by the ramp function.

__vDSP_IS
Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
The value to increment each subsequent value of the ramp function by.

__vDSP_O
The output vector.

__vDSP_OS
Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
The number of elements to modify.

Discussion
This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] += *Start * I[i*IS];
    *Start += *Step;
}
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmuladd2

Stereo version of `vDSP_vrampmuladd` (page 442).

```
void vDSP_vrampmuladd2 (
   const float *__vDSP_I0,
   const float *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   float *__vDSP_Start,
   const float *__vDSP_Step,
   float *__vDSP_O0,
   float *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
  First input vector, multiplied by the ramp function.

__vDSP_I1
  Second input vector, multiplied by the ramp function.

__vDSP_IS
  Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
  The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
  The value to increment each subsequent value of the ramp function by.

__vDSP_O0
  First output vector.

__vDSP_O1
  Second output vector.

__vDSP_OS
  Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
  The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] += *Start * I0[i*IS];
    O1[i*OS] += *Start * I1[i*IS];
    *Start += *Step;
}
```

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmuladd2_s1_15

Vector fixed-point 1.15 format version of `vDSP_vrampmuladd2` (page 442).

```
void vDSP_vrampmuladd2_s1_15 (
   const short int *__vDSP_I0,
   const short int *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   short int *__vDSP_Start,
   const short int *__vDSP_Step,
   short int *__vDSP_O0,
   short int *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
: First input vector, multiplied by the ramp function.

__vDSP_I1
: Second input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O0
: First output vector.

__vDSP_O1
: Second output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] += *Start * I0[i*IS];
    O1[i*OS] += *Start * I1[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by `32768.0`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmuladd2_s8_24

Vector fixed-point 8.24 format version of `vDSP_vrampmuladd2` (page 442).

```
void vDSP_vrampmuladd2_s8_24 (
   const int *__vDSP_I0,
   const int *__vDSP_I1,
   vDSP_Stride __vDSP_IS,
   int *__vDSP_Start,
   const int *__vDSP_Step,
   int *__vDSP_O0,
   int *__vDSP_O1,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I0
: First input vector, multiplied by the ramp function.

__vDSP_I1
: Second input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vectors. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O0
: First output vector.

__vDSP_O1
: Second output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This function calculates the following:

```
 for (i = 0; i < N; ++i) {
    O0[i*OS] += *Start * I0[i*IS];
    O1[i*OS] += *Start * I1[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by `16777216.0`.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmuladd_s1_15

Vector fixed-point 1.15 format version of `vDSP_vrampmuladd` (page 442).

```
void vDSP_vrampmuladd_s1_15 (
   const short int *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   short int *__vDSP_Start,
   const short int *__vDSP_Step,
   short int *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
Input vector, multiplied by the ramp function.

__vDSP_IS
Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
The value to increment each subsequent value of the ramp function by.

__vDSP_O
The output vector.

__vDSP_OS
Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
The number of elements to modify.

Discussion
This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] += *Start * I[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by `32768.0`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vrampmuladd_s8_24

Vector fixed-point 8.24 format version of `vDSP_vrampmuladd` (page 442).

```
void vDSP_vrampmuladd_s8_24 (
   const int *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   int *__vDSP_Start,
   const int *__vDSP_Step,
   int *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
: Input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O
: The output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] += *Start * I[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by `16777216.0`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmul_s1_15

Vector fixed-point 1.15 format version of `vDSP_vrampmul` (page 437).

```
void vDSP_vrampmul_s1_15 (
   const short int *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   short int *__vDSP_Start,
   const short int *__vDSP_Step,
   short int *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
: Input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O
: The output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] = *Start * I[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with one sign bit and 15 fraction bits. A value in this representation can be converted to floating-point by dividing it by `32768.0`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrampmul_s8_24

Vector fixed-point 8.24 format version of `vDSP_vrampmul` (page 437).

```
void vDSP_vrampmul_s8_24 (
   const int *__vDSP_I,
   vDSP_Stride __vDSP_IS,
   int *__vDSP_Start,
   const int *__vDSP_Step,
   int *__vDSP_O,
   vDSP_Stride __vDSP_OS,
   vDSP_Length __vDSP_N
);
```

Parameters

__vDSP_I
: Input vector, multiplied by the ramp function.

__vDSP_IS
: Stride length in input vector. For example, if `__vDSP_IS` is `2`, every second element is used.

__vDSP_Start
: The initial value for the ramp function. Modified on return to hold the next value (including accumulated errors) so that the ramp function can be continued smoothly.

__vDSP_Step
: The value to increment each subsequent value of the ramp function by.

__vDSP_O
: The output vector.

__vDSP_OS
: Stride length in output vector. For example, if `__vDSP_IS` is `2`, every second element is modified.

__vDSP_N
: The number of elements to modify.

Discussion

This routine calculates the following:

```
for (i = 0; i < N; ++i) {
    O[i*OS] = *Start * I[i*IS];
    *Start += *Step;
}
```

The elements are fixed-point numbers, each with eight integer bits (including the sign bit) and 24 fraction bits. A value in this representation can be converted to floating-point by dividing it by `16777216.0`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrsum

Vector running sum integration; single precision.

```
void vDSP_vrsum (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_S,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

S
: Single-precision real input scalar: weighting factor

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_0 = 0$$

$$C_{mK} = C_{(m-1)K} + SA_{mI} \qquad \mathrm{m} = \{1, \mathrm{N}\text{-}1\}$$

Integrates vector `A` using a running sum from vector `C`. Vector `A` is weighted by scalar `S` and added to the previous output point. The first element from vector `A` is not used in the sum.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrsumD

Vector running sum integration; double precision.

```
void vDSP_vrsumD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_S,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

S
: Double-precision real input scalar: weighting factor

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_0 = 0$$

$$C_{mK} = C_{(m-1)K} + SA_{mI} \qquad \text{m = \{1, N-1\}}$$

Integrates vector `A` using a running sum from vector `C`. Vector `A` is weighted by scalar `S` and added to the previous output point. The first element from vector `A` is not used in the sum.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrvrs

Vector reverse order, in place; single precision.

```
void vDSP_vrvrs (
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

C
: Single-precision real input-output vector

K

Stride for `C`

N

Count

Discussion

Performs the following operation:

$$C_{nK} \leftrightarrow C_{[N-n-1]K} \qquad n = \{0, (N/2)\text{-}1\}$$

Reverses the order of vector C in place.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vrvrsD

Vector reverse order, in place; double precision.

```
void vDSP_vrvrsD (
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

C

Double-precision real input-output vector

K

Stride for `C`

N

Count

Discussion

Performs the following operation:

$$C_{nK} \leftrightarrow C_{[N-n-1]K} \qquad n = \{0, (N/2)\text{-}1\}$$

Reverses the order of vector C in place.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsadd

Vector scalar add; single precision.

```
void vDSP_vsadd (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input scalar

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = A_{nI} + B \qquad n = \{0, N\text{-}1\}$$

Adds scalar B to each element of vector A and stores the result in the corresponding element of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsaddD

Vector scalar add; double precision.

```
void vDSP_vsaddD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = A_{nI} + B \qquad n = \{0, N\text{-}1\}$$

Adds scalar `B` to each element of vector `A` and stores the result in the corresponding element of vector `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsaddi

Integer vector scalar add.

```
void vDSP_vsaddi (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_B,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input vector

I
: Stride for `A`

B
: Integer input scalar

C
: Integer output vector

K
: Stride for `C`

N
: Count

Discussion
Performs the following operation:

$$C_{nK} = A_{nI} + B \qquad n = \{0, N\text{-}1\}$$

Adds scalar `B` to each element of vector `A` and stores the result in the corresponding element of vector `C`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsbm

Vector subtract and multiply; single precision.

```
void vDSP_vsbm (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real input vector

K
: Stride for C

D
: Single-precision real output vector

L
: Stride for D

N
: Count

Discussion

This performs the following operation:

$$D_{nM} = (A_{nI} - B_{nJ})\,C_{nK} \qquad \text{n} = \{0, \text{N-1}\}$$

Subtracts vector B from vector A and then multiplies the differences by vector C. Results are stored in vector D.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vsbmD

Vector subtract and multiply; double precision.

```
void vDSP_vsbmD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for A

B
: Double-precision real input vector

J
: Double for B

C

Double-precision real input vector

K

Stride for `C`

D

Double-precision real output vector

L

Stride for `D`

N

Count

Discussion

This performs the following operation:

$$D_{nM} = (A_{nI} - B_{nJ})\, C_{nK} \qquad \text{n} = \{0, \text{N-1}\}$$

Subtracts vector `B` from vector `A` and then multiplies the differences by vector `C`. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsbsbm

Vector subtract, subtract, and multiply; single precision.

```
void vDSP_vsbsbm (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   float *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for `A`

B

Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real input vector

K
: Stride for `C`

D
: Single-precision real input vector

L
: Stride for `D`

E
: Single-precision real output vector

M
: Stride for `E`

N
: Count

Discussion

This performs the following operation:

$$E_{nM} = (A_{nI} - B_{nJ})(C_{nK} - D_{nL}) \qquad n = \{0, N\text{-}1\}$$

Subtracts vector `B` from `A`, subtracts vector `D` from `C`, and multiplies the differences. Results are stored in vector `E`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsbsbmD

Vector subtract, subtract, and multiply; double precision.

```
void vDSP_vsbsbmD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   double *__vDSP_E,
   vDSP_Stride __vDSP_M,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input vector

J

Stride for B

C

Double-precision real input vector

K

Stride for C

D

Double-precision real input vector

L

Stride for D

E

Double-precision real output vector

M

Stride for E

N

Count

Discussion

This performs the following operation:

$$E_{nM} = (A_{nI} - B_{nJ})(C_{nK} - D_{nL}) \qquad n = \{0, N\text{-}1\}$$

Subtracts vector B from A, subtracts vector D from C, and multiplies the differences. Results are stored in vector E.

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsbsm

Vector subtract and scalar multiply; single precision.

```
void vDSP_vsbsm (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input vector

J
: Stride for `B`

C
: Single-precision real input scalar

D
: Single-precision real output vector

L
: Stride for `D`

N
: Count

Discussion
This performs the following operation:

$$D_{nK} = (A_{nI} - B_{nJ})C \qquad \mathrm{n} = \{0, \mathrm{N\text{-}1}\}$$

Subtracts vector `B` from vector `A` and then multiplies each difference by scalar `C`. Results are stored in vector `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsbsmD

Vector subtract and scalar multiply; double precision.

```
void vDSP_vsbsmD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input vector

J

Stride for B

C

Double-precision real input scalar

D

Double-precision real output vector

L

Stride for D

N

Count

Discussion

This performs the following operation:

$$D_{nK} = (A_{nI} - B_{nJ})C \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Subtracts vector B from vector A and then multiplies each difference by scalar C. Results are stored in vector D.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsdiv

Vector scalar divide; single precision.

```
void vDSP_vsdiv (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input scalar

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = \frac{A_{nI}}{B} \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Divides each element of vector `A` by scalar `B` and stores the result in the corresponding element of vector `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsdivD

Vector scalar divide; double precision.

```
void vDSP_vsdivD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input scalar

C

Double-precision real output vector

K

Stride for C

N

Count

Discussion

Performs the following operation:

$$C_{nK} = \frac{A_{nI}}{B} \qquad n = \{0, N\text{-}1\}$$

Divides each element of vector A by scalar B and stores the result in the corresponding element of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsdivi

Integer vector scalar divide.

```
void vDSP_vsdivi (
   int *__vDSP_A,
   vDSP_Stride __vDSP_I,
   int *__vDSP_B,
   int *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Integer input vector

I
: Stride for A

B
: Integer input scalar

C
: Integer output vector

K
: Stride for C

N
: Count

Discussion

Performs the following operation:

$$C_{nK} = \frac{A_{nI}}{B} \qquad n = \{0, N-1\}$$

Divides each element of vector A by scalar B and stores the result in the corresponding element of vector C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsimps

Simpson integration; single precision.

```
void vDSP_vsimps (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input scalar

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_0 = 0.0$$

$$C_K = \frac{[A_0 + A_I]B}{2}$$

$$C_{nK} = C_{[n-2]K} + \frac{[A_{[n-2]I} + 4.0 \times A_{[n-1]I} + A_{nI}]B}{3} \qquad n = \{2, N\text{-}1\}$$

Integrates vector `A` using Simpson integration, storing results in vector `C`. Scalar `B` specifies the integration step size. This function can only be done out of place.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsimpsD

Simpson integration; double precision.

```
void vDSP_vsimpsD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Double-precision real input vector

I
    Stride for A

B
    Double-precision real input scalar

C
    Double-precision real output vector

K
    Stride for C

N
    Count

Discussion
Performs the following operation:

$$C_0 = 0.0$$

$$C_K = \frac{[A_0 + A_I]B}{2}$$

$$C_{nK} = C_{[n-2]K} + \frac{[A_{[n-2]I} + 4.0 \times A_{[n-1]I} + A_{nI}]B}{3} \qquad \text{n = \{2, N-1\}}$$

Integrates vector A using Simpson integration, storing results in vector C. Scalar B specifies the integration step size. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsma

Vector scalar multiply and vector add; single precision.

```
void vDSP_vsma (
   const float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   const float *__vDSP_B,
   const float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input scalar

C
: Single-precision real input vector

K
: Stride for `C`

D
: Single-precision real output vector

L
: Stride for `D`

N
: Count

Discussion
Performs the following operation:

$$D_{nM} = A_{nI} \bullet B + C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies vector `A` by scalar `B` and then adds the products to vector `C`. Results are stored in vector `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsmaD

Vector scalar multiply and vector add; double precision.

```
void vDSP_vsmaD (
   const double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   const double *__vDSP_B,
   const double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for A

B
Double-precision real input scalar

C
Double-precision real input vector

K
Stride for C

D
Double-precision real output vector

L
Stride for D

N
Count

Discussion
Performs the following operation:

$$D_{nM} = A_{nI} \cdot B + C_{nK} \qquad \text{n} = \{0, \text{N-1}\}$$

Multiplies vector A by scalar B and then adds the products to vector C. Results are stored in vector D.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsmsa

Vector scalar multiply and scalar add; single precision.

```
void vDSP_vsmsa (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for `A`

B
: Single-precision real input scalar

C
: Single-precision real input scalar

D
: Single-precision real output vector

L
: Stride for `D`

N
: Count

Discussion

Performs the following operation:

$$D_{nM} = A_{nI} \cdot B + C \qquad n = \{0, N-1\}$$

Multiplies vector `A` by scalar `B` and then adds scalar `C` to each product. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsmsaD

Vector scalar multiply and scalar add; double precision.

```
void vDSP_vsmsaD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar

C
: Double-precision real input scalar

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count

Discussion

Performs the following operation:

$$D_{nM} = A_{nI} \cdot B + C \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Multiplies vector `A` by scalar `B` and then adds scalar `C` to each product. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsmsb

Vector scalar multiply and vector subtract; single precision.

```
void vDSP_vsmsb (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Single-precision real input vector

I
    Stride for `A`

B
    Single-precision real input scalar

C
    Single-precision real input vector

K
    Stride for `C`

D
    Single-precision real output vector

L
    Stride for `D`

N
    Count

Discussion

Performs the following operation:

$$D_{nM} = A_{nI} \cdot B - C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Multiplies vector `A` by scalar `B` and then subtracts vector `C` from the products. Results are stored in vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsmsbD

Vector scalar multiply and vector subtract; double precision.

```
void vDSP_vsmsbD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision real input vector

I
Stride for `A`

B
Double-precision real input scalar

C
Double-precision real input vector

K
Stride for `C`

D
Double-precision real output vector

L
Stride for `D`

N
Count

Discussion
Performs the following operation:

$$D_{nM} = A_{nI} \bullet B - C_{nK} \qquad \text{n} = \{0, \text{N-1}\}$$

Multiplies vector `A` by scalar `B` and then subtracts vector `C` from the products. Results are stored in vector `D`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsmul

Multiplies vector `signal1` by scalar `signal2` and leaves the result in vector `result`; single precision.

```
void vDSP_vsmul (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float *__vDSP_input2,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot B \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsmulD

Multiplies vector `signal1` by scalar signal2 and leaves the result in vector `result`; double precision.

```
void vDSP_vsmulD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double *__vDSP_input2,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot B \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsort

Vector in-place sort; single precision.

```
void vDSP_vsort (
   float *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_OFLAG
);
```

Parameters

C
: Single-precision real input-output vector

N
: Count

OFLAG
: Flag for sort order: 1 for ascending, -1 for descending

Discussion

Performs an in-place sort of vector `C` in the order specified by parameter `OFLAG`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsortD

Vector in-place sort; double precision.

```
void vDSP_vsortD (
   double *__vDSP_C,
   vDSP_Length __vDSP_N,
   int __vDSP_OFLAG
);
```

Parameters

C
: Double-precision real input-output vector

N
: Count

OFLAG
: Flag for sort order: 1 for ascending, -1 for descending

Discussion

Performs an in-place sort of vector `C` in the order specified by parameter `OFLAG`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vsorti

Vector index in-place sort; single precision.

```
void vDSP_vsorti (
   float *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length *__vDSP_List_addr,
   vDSP_Length __vDSP_N,
   int __vDSP_OFLAG
);
```

Parameters

C

Single-precision real input vector

IC

Integer output vector. Must be initialized with the indices of vector C, from 0 to N-1.

List_addr

Temporary vector. This is currently not used and NULL should be passed.

N

Count

OFLAG

Flag for sort order: 1 for ascending, -1 for descending

Discussion

Leaves input vector C unchanged and performs an in-place sort of the indices in vector IC according to the values in C. The sort order is specified by parameter OFLAG.

The values in C can then be obtained in sorted order, by taking indices in sequence from IC.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vsortiD

Vector index in-place sort; double precision.

```
void vDSP_vsortiD (
   double *__vDSP_C,
   vDSP_Length *__vDSP_IC,
   vDSP_Length *__vDSP_List_addr,
   vDSP_Length __vDSP_N,
   int __vDSP_OFLAG
);
```

Parameters

C

Double-precision real input vector

IC

Integer output vector. Must be initialized with the indices of vector C, from 0 to N-1.

List_addr
Temporary vector. This is currently not used and NULL should be passed.

N
Count

OFLAG
Flag for sort order: 1 for ascending, -1 for descending

Discussion
Leaves input vector `C` unchanged and performs an in-place sort of the indices in vector `IC` according to the values in `C`. The sort order is specified by parameter `OFLAG`.

The values in `C` can then be obtained in sorted order, by taking indices in sequence from `IC`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vspdp

Vector convert single-precision to double-precision.

```
void vDSP_vspdp (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for `A`

C
Double-precision real output vector

K
Stride for `C`

N
Count

Discussion
This performs the following operation:

$$C_{nk} = A_{ni} \qquad n = \{0, N\text{-}1\}$$

Creates double-precision vector `C` by converting single-precision inputs from vector `A`. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsq

Computes the squared values of vector `input` and leaves the result in vector `result`; single precision.

```
void vDSP_vsq (
   const float __vDSP_input[],
   vDSP_Stride __vDSP_strideInput,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI}^2 \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsqD

Computes the squared values of vector `signal1` and leaves the result in vector `result`; double precision.

```
void vDSP_vsqD (
   const double __vDSP_input[],
   vDSP_Stride __vDSP_strideInput,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI}^2 \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vssq

Computes the signed squares of vector `signal1` and leaves the result in vector `result`; single precision.

```
void vDSP_vssq (
   const float __vDSP_input[],
   vDSP_Stride __vDSP_strideInput,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot |A_{nI}| \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vssqD

Computes the signed squares of vector `signal1` and leaves the result in vector `result`; double precision.

```
void vDSP_vssqD (
   const double __vDSP_input[],
   vDSP_Stride __vDSP_strideInput,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} \cdot |A_{nI}| \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsub

Subtracts vector `signal1` from vector `signal2` and leaves the result in vector `result`; single precision.

```
void vDSP_vsub (
   const float __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   float __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vsubD

Subtracts vector `signal1` from vector `signal2` and leaves the result in vector `result`; double precision.

```
void vDSP_vsubD (
   const double __vDSP_input1[],
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   double __vDSP_result[],
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vswap

Vector swap; single precision.

```
void vDSP_vswap (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input-output vector

I

Stride for `A`

B

Single-precision real input-output vector

J

Stride for `B`

N

Count

Discussion

This performs the following operation:

$$C_{nK} \Leftrightarrow A_{nI} \qquad n = \{0, N-1\}$$

Exchanges the elements of vectors `A` and `B`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vswapD

Vector swap; double precision.

```
void vDSP_vswapD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input-output vector

I

Stride for `A`

B
: Double-precision real input-output vector

J
: Stride for B

N
: Count

Discussion
This performs the following operation:

$$C_{nK} \Leftrightarrow A_{nI} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Exchanges the elements of vectors A and B.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vswsum

Vector sliding window sum; single precision.

```
void vDSP_vswsum (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

C
: Single-precision real output vector

K
: Stride for C

N
: Count of output points

P
: Length of window

Discussion
Performs the following operation:

$$C_0(P) = \sum_{p=0}^{P-1} A_{qi} \quad (C_{nk}(P) - C_{(n-1)\,k}(P) + A_{(n+P-1)\,i} - A_{(n-1)\,i}) \qquad \text{n} = \{1, \text{N-1}\}$$

Writes the sliding window sum of `P` consecutive elements of vector `A` to vector `C`, for each of `N` possible starting positions of the `P`-element window in vector `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vswsumD

Vector sliding window sum; double precision.

```
void vDSP_vswsumD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count of output points

P
: Length of window

Discussion
Performs the following operation:

$$C_0(P) = \sum_{p=0}^{P-1} A_{qi} \quad (C_{nk}(P) - C_{(n-1)\,k}(P) + A_{(n+P-1)\,i} - A_{(n-1)\,i}) \qquad \text{n} = \{1, \text{N-1}\}$$

Writes the sliding window sum of `P` consecutive elements of vector `A` to vector `C`, for each of `N` possible starting positions of the `P`-element window in vector `A`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vtabi

Vector interpolation, table lookup; single precision.

```
void vDSP_vtabi (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_S1,
   float *__vDSP_S2,
   float *__vDSP_C,
   vDSP_Length __vDSP_M,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for `A`

S1
Single-precision real input scalar: scale factor

S2
Single-precision real input scalar: base offset

C
Single-precision real input vector: lookup table

M
Lookup table size

D
Single-precision real output vector

L
Stride for `D`

N
Count

Discussion
Performs the following operation:

$$p = F \cdot A_{nI} + G \qquad q = \operatorname{floor}(p) \qquad r = p - \operatorname{float}(q) \qquad D_{nk} = (1.0 - r)C_q + rC_{q-1} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Evaluates elements of vector A for use as offsets into vector `C`. Vector `C` is a zero-based lookup table supplied by the caller that generates output values for vector `D`. Linear interpolation is used to compute output values when offsets do not evaluate integrally. Scale factor `S1` and base offset `S2` map the anticipated range of input values to the range of the lookup table and are typically assigned values such that:

```
    floor(F * minimum input value + G) = 0
    floor(F * maximum input value + G) = M-1
```

Input values that evaluate to zero or less derive their output values from table location zero. Values that evaluate beyond the table, greater than `M`-1, derive their output values from the last table location. For inputs that evaluate integrally, the table location indexed by the integral is copied as the output value. All other inputs derive their output values by interpolation between the two table values surrounding the evaluated input.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vtabiD

Vector interpolation, table lookup; double precision.

```
void vDSP_vtabiD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_S1,
   double *__vDSP_S2,
   double *__vDSP_C,
   vDSP_Length __vDSP_M,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Double-precision real input vector

I
  Stride for `A`

S1
  Double-precision real input scalar: scale factor

S2
  Double-precision real input scalar: base offset

C
  Double-precision real input vector: lookup table

M
  Lookup table size

D
  Double-precision real output vector

L

Stride for D

N

Count

Discussion

Performs the following operation:

$$p = F \bullet A_{nI} + G \qquad q = \text{floor}(p) \qquad r = p - \text{float}(q) \qquad D_{nk} = (1.0 - r)C_q + rC_{q-1} \qquad \text{n} = \{0, \text{N-1}\}$$

Evaluates elements of vector A for use as offsets into vector C. Vector C is a zero-based lookup table supplied by the caller that generates output values for vector D. Linear interpolation is used to compute output values when offsets do not evaluate integrally. Scale factor S1 and base offset S2 map the anticipated range of input values to the range of the lookup table and are typically assigned values such that:

```
    floor(F * minimum input value + G) = 0
    floor(F * maximum input value + G) = M-1
```

Input values that evaluate to zero or less derive their output values from table location zero. Values that evaluate beyond the table, greater than M-1, derive their output values from the last table location. For inputs that evaluate integrally, the table location indexed by the integral is copied as the output value. All other inputs derive their output values by interpolation between the two table values surrounding the evaluated input.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_vthr

Vector threshold; single precision.

```
void vDSP_vthr (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision real input vector

I

Stride for A

B

Single-precision real input scalar: lower threshold

C

Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B \quad n = \{0, N\text{-}1\}$$

Creates vector `C` by comparing each input from vector `A` with scalar `B`. If an input value is less than `B`, `B` is copied to `C`; otherwise, the input value from `A` is copied to `C`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vthrD

Vector threshold; double precision.

```
void vDSP_vthrD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar: lower threshold

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = B \quad n = \{0, N\text{-}1\}$$

Creates vector C by comparing each input from vector A with scalar B. If an input value is less than B, B is copied to C; otherwise, the input value from A is copied to C.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vthres

Vector threshold with zero fill; single precision.

```
void vDSP_vthres (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision real input vector

I
Stride for A

B
Single-precision real input scalar: lower threshold

C
Single-precision real output vector

K
Stride for C

N
Count

Discussion
Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = 0.0 \qquad n = \{0, N\text{-}1\}$$

Creates vector C by comparing each input from vector A with scalar B. If an input value is less than B, zero is written to C; otherwise, the input value from A is copied to C.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vthresD

Vector threshold with zero fill; double precision.

```
void vDSP_vthresD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision real input vector

I

Stride for A

B

Double-precision real input scalar: lower threshold

C

Double-precision real output vector

K

Stride for C

N

Count

Discussion

Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad C_{nK} = A_{nI} \quad \text{else} \quad C_{nK} = 0.0 \qquad n = \{0, N\text{-}1\}$$

Creates vector C by comparing each input from vector A with scalar B. If an input value is less than B, zero is written to C; otherwise, the input value from A is copied to C.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vthrsc

Vector threshold with signed constant; single precision.

```
void vDSP_vthrsc (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   float *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input scalar: lower threshold

C
: Single-precision real input scalar

D
: Single-precision real output vector

L
: Stride for D

N
: Count

Discussion

Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad D_{nM} = C \quad \text{else} \quad D_{nM} = -C \quad n = \{0, N\text{-}1\}$$

Creates vector D using the plus or minus value of scalar C. The sign of the output element is determined by comparing input from vector A with threshold scalar B. For input values less than B, the negated value of C is written to vector D. For input values greater than or equal to B, C is copied to vector D.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vthrscD

Vector threshold with signed constant; double precision.

```
void vDSP_vthrscD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   double *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar: lower threshold

C
: Double-precision real input scalar

D
: Double-precision real output vector

L
: Stride for `D`

N
: Count

Discussion

Performs the following operation:

$$\text{If} \quad A_{nI} \geq B \quad \text{then} \quad D_{nM} = C \quad \text{else} \quad D_{nM} = -C \qquad n = \{0, N\text{-}1\}$$

Creates vector `D` using the plus or minus value of scalar `C`. The sign of the output element is determined by comparing input from vector `A` with threshold scalar `B`. For input values less than `B`, the negotiated value of `C` is written to vector `D`. For input values greater than or equal to `B`, `C` is copied to vector `D`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vtmerg

Tapered merge of two vectors; single precision.

```
void vDSP_vtmerg (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input vector

J
: Stride for B

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} + \frac{n(B_{nJ} - A_{nI})}{N-1} \qquad \mathrm{n} = \{0, \mathrm{N\text{-}1}\}$$

Performs a tapered merge of vectors A and B. Values written to vector C range from element zero of vector A to element N-1 of vector B. Output values between these endpoints reflect varying amounts of their corresponding inputs from vectors A and B, with the percentage of vector A decreasing and the percentage of vector B increasing as the index increases.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_vtmergD

Tapered merge of two vectors; double precision.

```
void vDSP_vtmergD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input vector

J
: Stride for `B`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + \frac{n(B_{nJ} - A_{nI})}{N-1} \qquad \mathrm{n} = \{0, \mathrm{N\text{-}1}\}$$

Performs a tapered merge of vectors `A` and `B`. Values written to vector `C` range from element zero of vector `A` to element `N`-1 of vector `B`. Output values between these endpoints reflect varying amounts of their corresponding inputs from vectors `A` and `B`, with the percentage of vector `A` decreasing and the percentage of vector `B` increasing as the index increases.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_vtrapz

Vector trapezoidal integration; single precision.

```
void vDSP_vtrapz (
   float *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

I
: Stride for A

B
: Single-precision real input scalar: step size

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion
Performs the following operation:

$$C_0 = 0.0$$

$$C_{nK} = C_{[n-1]K} + \frac{B[A_{[n-1]I} + A_{nI}]}{2} \qquad n = \{1, N\text{-}1\}$$

Estimates the integral of vector A using the trapezoidal rule. Scalar B specifies the integration step size. This function can only be done out of place.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_vtrapzD

Vector trapezoidal integration; double precision.

```
void vDSP_vtrapzD (
   double *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

I
: Stride for `A`

B
: Double-precision real input scalar: step size

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Performs the following operation:

$$C_0 = 0.0$$

$$C_{nK} = C_{[n-1]K} + \frac{B[A_{[n-1]I} + A_{nI}]}{2} \qquad n = \{1, N\text{-}1\}$$

Estimates the integral of vector `A` using the trapezoidal rule. Scalar `B` specifies the integration step size. This function can only be done out of place.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_wiener

Wiener-Levinson general convolution; single precision.

```
void vDSP_wiener (
   vDSP_Length __vDSP_L,
   float *__vDSP_A,
   float *__vDSP_C,
   float *__vDSP_F,
   float *__vDSP_P,
   int __vDSP_IFLG,
   int *__vDSP_IERR
);
```

Parameters

L
: Input filter length

A
: Single-precision real input vector: coefficients

C
: Single-precision real input vector: input coefficients

F
: Single-precision real output vector: filter coefficients

P
: Single-precision real output vector: error prediction operators

IFLG
: Not currently used, pass zero

IERR
: Error flag

Discussion

Performs the following operation:

$$\text{Find} \quad C_m \quad \text{such that} \quad F_n = \sum_{m=0}^{L-1} A_{n-m} \cdot C_m \qquad n = \{0, L-1\}$$

solves a set of single-channel normal equations described by:

```
B[n] = C[0] * A[n] + C[1] * A[n-1] +, . . . ,+ C[N-1] * A[n-N+1]
for n = {0, N-1}
```

where matrix A contains elements of the symmetric Toeplitz matrix shown below. This function can only be done out of place.

Note that A[-n] is considered to be equal to A[n].

vDSP_wiener solves this set of simultaneous equations using a recursive method described by Levinson. See Robinson, E.A., *Multichannel Time Series Analysis with Digital Computer Programs*. San Francisco: Holden-Day, 1967, pp. 43-46.

```
|A[0]  A[1]  A[2] ... A[N-1] |    |C[0]  |   |B[0]  |
|A[1]  A[0]  A[1] ... A[N-2] |    |C[1]  |   |B[1]  |
|A[2]  A[1]  A[0] ... A[N-3] |  * |C[2]  | = |B[2]  |
| ...   ...   ... ... ...    |    | ...  |   | ...  |
|A[N-1]A[N-2]A[N-3] ... A[0] |    |C[N-1]|   |B[N-1]|
```

Typical methods for solving N equations in N unknowns have execution times proportional to N³, and memory requirements proportional to N². By taking advantage of duplicate elements, the recursion method executes in a time proportional to N² and requires memory proportional to N. The Wiener-Levinson algorithm recursively builds a solution by computing the m+1 matrix solution from the m matrix solution.

With successful completion, vDSP_wiener returns zero in error flag IERR. If vDSP_wiener fails, IERR indicates in which pass the failure occurred.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_wienerD

Wiener-Levinson general convolution; double precision.

```
void vDSP_wienerD (
   vDSP_Length __vDSP_L,
   double *__vDSP_A,
   double *__vDSP_C,
   double *__vDSP_F,
   double *__vDSP_P,
   int __vDSP_IFLG,
   int *__vDSP_IERR
);
```

Parameters

L
Input filter length

A
Double-precision real input vector: coefficients

C
Double-precision real input vector: input coefficients

F
Double-precision real output vector: filter coefficients

P
Double-precision real output vector: error prediction operators

IFLG
Not currently used, pass zero

IERR
Error flag

Discussion
Performs the following operation:

$$\text{Find} \quad C_m \quad \text{such that} \quad F_n = \sum_{m=0}^{L-1} A_{n-m} \cdot C_m \qquad n = \{0, L\text{-}1\}$$

solves a set of single-channel normal equations described by:

```
B[n] = C[0] * A[n] + C[1] * A[n-1] +, . . . ,+ C[N-1] * A[n-N+1]
for n = {0, N-1}
```

where matrix A contains elements of the symmetric Toeplitz matrix shown below. This function can only be done out of place.

Note that A[-n] is considered to be equal to A[n].

vDSP_wiener solves this set of simultaneous equations using a recursive method described by Levinson. See Robinson, E.A., *Multichannel Time Series Analysis with Digital Computer Programs*. San Francisco: Holden-Day, 1967, pp. 43-46.

```
|A[0]  A[1]  A[2] ... A[N-1] |    |C[0]  |   |B[0]  |
|A[1]  A[0]  A[1] ... A[N-2] |    |C[1]  |   |B[1]  |
|A[2]  A[1]  A[0] ... A[N-3] |  * |C[2]  | = |B[2]  |
| ...   ...   ... ... ...    |    | ...  |   | ...  |
|A[N-1]A[N-2]A[N-3] ... A[0] |    |C[N-1]|   |B[N-1]|
```

Typical methods for solving N equations in N unknowns have execution times proportional to N^3, and memory requirements proportional to N^2. By taking advantage of duplicate elements, the recursion method executes in a time proportional to N^2 and requires memory proportional to N. The Wiener-Levinson algorithm recursively builds a solution by computing the m+1 matrix solution from the m matrix solution.

With successful completion, vDSP_wiener returns zero in error flag IERR. If vDSP_wiener fails, IERR indicates in which pass the failure occurred.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_zaspec

Computes an accumulating autospectrum; single precision.

```
void vDSP_zaspec (
   DSPSplitComplex *__vDSP_A,
   float *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
    Input vector

C
    Input-output vector

N
    Real output count

Discussion

`vDSP_zaspec` multiplies single-precision complex vector `A` by its complex conjugates, yielding the sums of the squares of the complex and real parts: (x + iy) (x - iy) = (x*x + y*y). The results are added to real single-precision input-output vector `C`. Vector `C` must contain valid data from previous processing or should be initialized according to your needs before calling `vDSP_zaspec`.

$$C_n = C_n + (Re(A_n))^2 + (Im(A_n))^2 \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zaspecD

Computes an accumulating autospectrum; double precision.

```
void vDSP_zaspecD (
   DSPDoubleSplitComplex *A,
   double *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Input vector

C

Input-output vector

N

Real output count

Discussion

`vDSP_zaspecD` multiplies double-precision complex vector `A` by its complex conjugates, yielding the sums of the squares of the complex and real parts: `(x + iy) (x - iy) = (x*x + y*y)`. The results are added to real double-precision input-output vector `C`. Vector `C` must contain valid data from previous processing or should be initialized according to your needs before calling `vDSP_zaspec`.

$$C_n = C_n + (Re(A_n))^2 + (Im(A_n))^2 \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zcoher

Coherence function of two signals; single precision.

```
void vDSP_zcoher (
   float *__vDSP_A,
   float *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   float *__vDSP_D,
   vDSP_Length __vDSP_N
);
```

Discussion
Computes the single-precision coherence function D of two signals. The inputs are the signals' autospectra, real single-precision vectors A and B, and their cross-spectrum, single-precision complex vector C.

$$D_n = \frac{[Re(C_n)]^2 + [Im(C_n)]^2}{A_n B_n} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zcoherD

Coherence function of two signals; double precision.

```
void vDSP_zcoherD (
   double *__vDSP_A,
   double *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   double *__vDSP_D,
   vDSP_Length __vDSP_N
);
```

Discussion
Computes the double-precision coherence function D of two signals. The inputs are the signals' autospectra, real double-precision vectors A and B, and their cross-spectrum, double-precision complex vector C.

$$D_n = \frac{[Re(C_n)]^2 + [Im(C_n)]^2}{A_n B_n} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zconv

Performs either correlation or convolution on two complex vectors; single precision.

```
void vDSP_zconv (
   DSPSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPSplitComplex *__vDSP_filter,
   vDSP_Stride __vDSP_strideFilter,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_lenResult,
   vDSP_Length __vDSP_lenFilter
);
```

Discussion

`A` is the input vector, with stride `I`, and `C` is the output vector, with stride `K` and length `N`.

`B` is a filter vector, with stride `I` and length `P`. If `J`is positive,the function performs correlation. If `J`is negative, it performs convolution and `B`must point to the last element in the filter vector. The function can run in place, but `C`cannot be in place with `B`.

$$C_{nK} = \sum_{p=0}^{P-1} A_{(n+p)I} B_{pJ} \qquad \text{n} = \{0, \text{N-1}\}$$

The value of `N` must be less than or equal to 512.

Criteria to invoke vectorized code:

- Both the real parts and the imaginary parts of vectors `A` and `C` must be relatively aligned.
- The values of `I` and `K` must be 1.

If any of these criteria is not satisfied, the function invokes scalar code.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zconvD

Performs either correlation or convolution on two complex vectors; double precision.

```
void vDSP_zconvD (
   DSPDoubleSplitComplex *__vDSP_signal,
   vDSP_Stride __vDSP_signalStride,
   DSPDoubleSplitComplex *__vDSP_filter,
   vDSP_Stride __vDSP_strideFilter,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_lenResult,
   vDSP_Length __vDSP_lenFilter
);
```

Discussion

`A` is the input vector, with stride `I`, and `C` is the output vector, with stride `K` and length `N`.

B is a filter vector, with stride I and length P. If J is positive, the function performs correlation. If J is negative, it performs convolution and B must point to the last element in the filter vector. The function can run in place, but C cannot be in place with B.

$$C_{nK} = \sum_{p=0}^{P-1} A_{(n+p)I} B_{pJ} \qquad n = \{0, N\text{-}1\}$$

The value of N must be less than or equal to 512.

Criteria to invoke vectorized code:

No Altivec support for double precision. On a PowerPC processor, this function always invokes scalar code.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zcspec

Accumulating cross-spectrum on two complex vectors; single precision.

```
void vDSP_zcspec (
   DSPSplitComplex *__vDSP_A,
   DSPSplitComplex *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Single-precision complex input vector

B
Single-precision complex input vector

C
Single-precision complex input-output vector

N
Count

Discussion
Computes the cross-spectrum of complex vectors A and B and then adds the results to complex input-output vector C. Vector C should contain valid data from previous processing or should be initialized with zeros before calling `vDSP_zcspec`.

$$C_n = C_n + A_n^* B_n \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zcspecD

Accumulating cross-spectrum on two complex vectors; double precision.

```
void vDSP_zcspecD (
   DSPDoubleSplitComplex *A,
   DSPDoubleSplitComplex *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision complex input vector

B
Double-precision complex input vector

C
Double-precision complex input-output vector

N
Count

Discussion
Computes the cross-spectrum of complex vectors `A` and `B` and then adds the results to complex input-output vector `C`. Vector `C` should contain valid data from previous processing or should be initialized with zeros before calling `vDSP_zcspecD`.

$$C_n = C_n + A_n^* B_n \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zdotpr

Calculates the complex dot product of complex vectors `A` and `B` and leaves the result in complex vector `C`; single precision.

```
void vDSP_zdotpr (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zdotprD

Calculates the complex dot product of complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zdotprD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zidotpr

Calculates the conjugate dot product (or inner dot product) of complex vectors A and B and leave the result in complex vector C; single precision.

```
void vDSP_zidotpr (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A^{*}_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zidotprD

Calculates the conjugate dot product (or inner dot product) of complex vectors A and B and leave the result in complex vector C; double precision.

```
void vDSP_zidotprD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A^{*}_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zmma

Multiplies two complex matrices, then adds a third complex matrix; out-of-place; single precision.

```
void vDSP_zmma (
   DSPSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an `M`-by-`P` matrix `A` by a `P`-by-`N` matrix `B`, adds the product to `M`-by-`N` matrix `C`, and stores the result in `M`-by-`N` matrix `D`; single precision.

This performs the following operation:

$$D_{(rN+q)L} = C_{(rN+q)K} + \sum_{p=0}^{P-1} A_{(rP+p)I} B_{(pN+q)J}$$

$$0 \le r < M, \quad 0 \le q < N$$

Parameters `A` and `C` are the matrixes to be multiplied, and `C` the matrix to be added. `I` is an address stride through `A`. `J` is an address stride through `B`. `K` is an address stride through `C`. `L` is an address stride through `D`.

Parameter `D` is the result matrix.

Parameter `M` is the row count for `A`, `C` and `D`. Parameter `N` is the column count of `B`, `C`, and `D`. Parameter `P` is the column count of `A` and the row count of `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zmmaD

Multiplies two complex matrices, then adds a third complex matrix; out-of-place; double precision.

```
void vDSP_zmmaD (
   DSPDoubleSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPDoubleSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPDoubleSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPDoubleSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion

This function performs an out-of-place complex multiplication of an M-by-P matrix A by a P-by-N matrix B, adds the product to M-by-N matrix C, and stores the result in M-by-N matrix D; double precision.

This performs the following operation:

$$D_{(rN+q)L} = C_{(rN+q)K} + \sum_{p=0}^{P-1} A_{(rP+p)I} B_{(pN+q)J}$$

$$0 \le r < M, \quad 0 \le q < N$$

Parameters A and C are the matrixes to be multiplied, and C the matrix to be added. I is an address stride through A. J is an address stride through B. K is an address stride through C. L is an address stride through D.

Parameter D is the result matrix.

Parameter M is the row count for A, C and D. Parameter N is the column count of B, C, and D. Parameter P is the column count of A and the row count of B.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zmms

Multiplies two complex matrices, then subtracts a third complex matrix; out-of-place; single precision.

```
void vDSP_zmms (
   DSPSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an M-by-P matrix A by a P-by-N matrix B , subtracts M-by-N matrix C from the product, and stores the result in M-by-N matrix D.

This performs the following operation:

$$D_{(rN+q)L} = \sum_{p=0}^{P-1} A_{(rP+p)I} B_{(pN+q)J} - C_{(rN+q)K}$$

$$0 \le r < M, \quad 0 \le q < N$$

Parameters A and B are the matrixes to be multiplied, and C the matrix to be subtracted. I is an address stride through A. J is an address stride through B. K is an address stride through C. L is an address stride through D.

Parameter D is the result matrix.

Parameter M is the row count for A, C and D. Parameter N is the column count of B, C, and D. Parameter P is the column count of A and the row count of B.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_zmmsD

Multiplies two complex matrices, then subtracts a third complex matrix; out-of-place; double precision.

```
void vDSP_zmmsD (
   DSPDoubleSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPDoubleSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPDoubleSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPDoubleSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an M-by-P matrix A by a P-by-N matrix B, subtracts M-by-N matrix C from the product, and stores the result in M-by-N matrix D.

This performs the following operation:

$$D_{(rN+q)L} = \sum_{p=0}^{P-1} A_{(rP+p)I} B_{(pN+q)J} - C_{(rN+q)K}$$

$$0 \le r < M, \quad 0 \le q < N$$

Parameters A and B are the matrixes to be multiplied, and C the matrix to be subtracted. I is an address stride through A. J is an address stride through B. K is an address stride through C. L is an address stride through D.

Parameter D is the result matrix.

Parameter M is the row count for A, C and D. Parameter N is the column count of B, C, and D. Parameter P is the column count of A and the row count of B.

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_zmmul

Multiplies two matrices of complex numbers; out-of-place; single precision.

```
void vDSP_zmmul (
   DSPSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an `M`-by-`P` matrix `A` by a `P`-by-`N` matrix `B` and stores the results in an `M`-by-`N` matrix `C`.

This performs the following operation:

$$C_{(mN+n)K} = \sum_{p=0}^{P-1} A_{(mP+p)I} \bullet B_{(pN+n)J} \qquad \text{n} = \{0, \text{N-1}\} \text{ and m} = \{0, \text{M-1}\}$$

Parameters `A` and `B` are the matrixes to be multiplied. `I` is an address stride through `A`. `J` is an address stride through `B`.

Parameter `C` is the result matrix. `K` is an address stride through `C`.

Parameter `M` is the row count for both `A` and `C`. Parameter `N` is the column count for both `B` and `C`. Parameter `P` is the column count for `A` and the row count for `B`.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zmmulD

Multiplies two matrices of complex numbers; out-of-place; double precision.

```
void vDSP_zmmulD (
   DSPDoubleSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPDoubleSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPDoubleSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an `M`-by-`P` matrix `A` by a `P`-by-`N` matrix `B` and stores the results in an `M`-by-`N` matrix `C`.

This performs the following operation:

$$C_{(mN+n)\,K} = \sum_{p=0}^{P-1} A_{(mP+p)\,I} \bullet B_{(pN+n)\,J} \qquad \text{n = \{0, N-1\} and m = \{0, M-1\}}$$

Parameters A and B are the matrixes to be multiplied. I is an address stride through A. J is an address stride through B.

Parameter C is the result matrix. K is an address stride through C.

Parameter M is the row count for both A and C. Parameter N is the column count for both B and C. Parameter P is the column count for A and the row count for B.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zmsm

Subtracts the product of two complex matrices from a third complex matrix; out-of-place; single precision.

```
void vDSP_zmsm (
   DSPSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an M-by-P matrix A by a P-by-N matrix B, subtracts the product from M-by-P matrix C, and stores the result in M-by-P matrix *D*.

This performs the following operation:

$$D_{(rN+q)L} = C_{(rN+q)K} - \sum_{p=0}^{P-1} A_{(rP+p)I}\, B_{(pN+q)J}$$

Parameters A and B are the matrixes to be multiplied, and C is the matrix from which the product is to be subtracted. aStride is an address stride through A. bStride is an address stride through B. cStride is an address stride through C. dStride is an address stride through D.

Parameter D is the result matrix.

Parameter M is the row count for A, C and D. Parameter N is the column count of B, C, and D. Parameter P is the column count of A and the row count of B.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zmsmD

Subtracts the product of two complex matrices from a third complex matrix; out-of-place; double precision.

```
void vDSP_zmsmD (
   DSPDoubleSplitComplex *__vDSP_a,
   vDSP_Stride __vDSP_i,
   DSPDoubleSplitComplex *__vDSP_b,
   vDSP_Stride __vDSP_j,
   DSPDoubleSplitComplex *__vDSP_c,
   vDSP_Stride __vDSP_k,
   DSPDoubleSplitComplex *__vDSP_d,
   vDSP_Stride __vDSP_l,
   vDSP_Length __vDSP_M,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_P
);
```

Discussion
This function performs an out-of-place complex multiplication of an M-by-P matrix A by a P-by-N matrix B, subtracts the product from M-by-P matrix C, and stores the result in M-by-P matrix D.

This performs the following operation:

$$D_{(rN+q)L} = C_{(rN+q)K} - \sum_{p=0}^{P-1} A_{(rP+p)I} B_{(pN+q)J}$$

Parameters A and B are the matrixes to be multiplied, and parameter C is the matrix from which the product is to be subtracted. aStride is an address stride through A. bStride is an address stride through B. cStride is an address stride through C. dStride is an address stride through D.

Parameter D is the result matrix.

Parameter M is the row count for A, C and D. Parameter N is the column count of B, C, and D. Parameter P is the column count of A and the row count of B.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zrdesamp

Complex/real downsample with anti-aliasing; single precision.

```
void vDSP_zrdesamp (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Single-precision complex input vector.

I
: Complex decimation factor.

B
: Filter coefficient vector.

C
: Single-precision complex output vector.

N
: Length of output vector.

M
: Length of real filter vector.

Discussion

Performs finite impulse response (FIR) filtering at selected positions of input vector A.

$$C_m = \sum_{p=0}^{P-1} A_{(mi+p)} \cdot B_p \,, \qquad (\text{m} = \{0, \text{N-1}\})$$

Length of A must be at least (N+M-1)*i. This function can run in place, but C cannot be in place with B.

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_zrdesampD

Complex/real downsample with anti-aliasing; double precision.

```
void vDSP_zrdesampD (
   DSPDoubleSplitComplex *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N,
   vDSP_Length __vDSP_M
);
```

Parameters

A
: Double-precision complex input vector.

I
: Complex decimation factor.

B
: Filter coefficient vector.

C
: Double-precision complex output vector.

N
: Length of output vector.

M
: Length of real filter vector.

Discussion

Performs finite impulse response (FIR) filtering at selected positions of input vector `A`.

$$C_m = \sum_{p=0}^{P-1} A_{(mi+p)} \cdot B_p , \qquad (\mathrm{m} = \{0, \mathrm{N}\text{-}1\})$$

Length of `A` must be at least (`N`+`M`-1)*`i`. This function can run in place, but `C` cannot be in place with `B`.

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrdotpr

Calculates the complex dot product of complex vector `A` and real vector `B` and leaves the result in complex vector `C`; single precision.

```
void vDSP_zrdotpr (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zrdotprD

Calculates the complex dot product of complex vector `A` and real vector `B` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zrdotprD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C = \sum_{n=0}^{N-1} A_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zrvadd

Adds real vector `B` to complex vector `A` and leaves the result in complex vector `C`; single precision.

```
void vDSP_zrvadd (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for A

B
: Input vector

J
: Address stride for B

C
: Output vector

K
: Address stride for C

N
: Complex output count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrvaddD

Adds real vector B to complex vector A and leaves the result in complex vector C; double precision.

```
void vDSP_zrvaddD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
Input vector

I
Address stride for A

B
Input vector

J
Address stride for B

C
Output vector

K
Address stride for C

N
Complex output count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad \text{n} = \{0, \text{N-1}\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrvdiv

Divides complex vector `A` by real vector `B` and leaves the result in vector `C`; single precision.

```
void vDSP_zrvdiv (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Discussion
This performs the following operation:

$$C_{nk} = \frac{A_{ni}}{B_{nj}} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zrvdivD

Divides complex vector `A` by real vector `B` and leaves the result in vector `C`; double precision.

```
void vDSP_zrvdivD (
   DSPDoubleSplitComplex *A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Discussion
This performs the following operation:

$$C_{nk} = \frac{A_{ni}}{B_{nj}} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zrvmul

Multiplies complex vector `A` by real vector `B` and leaves the result in vector `C`; single precision.

```
void vDSP_zrvmul (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for A

B
: Input vector

J
: Address stride for B

C
: Output vector

K
: Address stride for C

N
: Complex output count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} \cdot B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrvmulD

Multiplies complex vector `A` by real vector `B` and leaves the result in vector `C`; double precision.

```
void vDSP_zrvmulD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex output count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} \cdot B_{nJ} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrvsub

Subtracts real vector `B` from complex vector `A` and leaves the result in complex vector `C`; single precision.

```
void vDSP_zrvsub (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const float __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex output count

Discussion

This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad \mathrm{n} = \{0, \mathrm{N-1}\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zrvsubD

Subtracts real vector `B` from complex vector `A` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zrvsubD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const double __vDSP_input2[],
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for A

B
: Input vector

J
: Address stride for B

C
: Output vector

K
: Address stride for C

N
: Complex output count

Discussion
This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_ztoc

Copies the contents of a split complex vector A to an interleaved complex vector C; single precision.

```
void vDSP_ztoc (
   const DSPSplitComplex *__vDSP_Z,
   vDSP_Stride __vDSP_strideZ,
   DSPComplex __vDSP_C[],
   vDSP_Stride __vDSP_strideC,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation;

$$C_{nK} = Re(A_{nI})$$

$$C_{nK+1} = Im(A_{nI}) \qquad n = \{0, N\text{-}1\}$$

The `strideC` parameter contains an address stride through `C`. `strideZ` is an address stride through `Z`.

For best performance, `C->realp`, `C->imagp`, `A->realp`, and `A->imagp` should be 16-byte aligned.

See also vDSP_ctoz (page 197) and vDSP_ctozD (page 197).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_ztocD

Copies the contents of a split complex vector `A` to an interleaved complex vector `C`; double precision.

```
void vDSP_ztocD (
   const DSPDoubleSplitComplex *__vDSP_Z,
   vDSP_Stride __vDSP_strideZ,
   DSPDoubleComplex __vDSP_C[],
   vDSP_Stride __vDSP_strideC,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$C_{nK} = Re(A_{nI})$$

$$C_{nK+1} = Im(A_{nI}) \qquad n = \{0, N\text{-}1\}$$

The `strideC` parameter contains an address stride through `C`. `strideZ` is an address stride through `Z`.

For best performance, `C->realp`, `C->imagp`, `A->realp`, and `A->imagp` should be 16-byte aligned.

See also vDSP_ctoz (page 197) and vDSP_ctozD (page 197).

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_ztrans

Transfer function; single precision.

```
void vDSP_ztrans (
   float *__vDSP_A,
   DSPSplitComplex *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision real input vector

B
: Single-precision complex input vector

C
: Single-precision complex output vector

N
: Count

Discussion

This performs the following operation:

$$C_n = \frac{B_n}{A_n} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_ztransD

Transfer function; double precision.

```
void vDSP_ztransD (
   double *__vDSP_A,
   DSPDoubleSplitComplex *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision real input vector

B
: Double-precision complex input vector

C
: Double-precision complex output vector

N
: Count

Discussion

This performs the following operation:

$$C_n = \frac{B_n}{A_n} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvabs

Complex vector absolute values; single precision.

```
void vDSP_zvabs (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision complex input vector

I

Stride for `A`

C

Single-precision real output vector

K

Stride for C

N

Count

Discussion

This performs the following operation:

$$C_{nk} = \sqrt{Re[A_{ni}]^2 + Im[A_{ni}]^2} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvabsD

Complex vector absolute values; double precision.

```
void vDSP_zvabsD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for A

C
: Double-precision real output vector

K
: Stride for C

N
: Count

Discussion
This performs the following operation:

$$C_{nk} = \sqrt{Re[A_{ni}]^2 + Im[A_{ni}]^2} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvadd

Adds complex vectors A and B and leaves the result in complex vector C; single precision.

```
void vDSP_zvadd (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex output count

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvaddD

Adds complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zvaddD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion

This performs the following operation:

$$C_{nK} = A_{nI} + B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvcma

Multiplies complex vector `B` by the complex conjugates of complex vector `A`, adds the products to complex vector `C`, and stores the results in complex vector `D`; single precision.

```
void vDSP_zvcma (
   const DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   const DSPSplitComplex *__vDSP_input3,
   vDSP_Stride __vDSP_stride3,
   const DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$D_{nL} = A^*_{nI} B_{nJ} + C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvcmaD

Multiplies complex vector `B` by the complex conjugates of complex vector `A`, adds the products to complex vector `C`, and stores the results in complex vector `D`; double precision.

```
void vDSP_zvcmaD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_input3,
   vDSP_Stride __vDSP_stride3,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Discussion
This performs the following operation:

$$D_{nL} = A^*_{nI} B_{nJ} + C_{nK} \qquad \mathrm{n} = \{0, \mathrm{N}-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvcmul

Complex vector conjugate and multiply; single precision.

```
void vDSP_zvcmul (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

B
: Single-precision complex input vector

J
: Stride for `B`

B
: Single-precision complex output vector

K
: Stride for `B`

N
: Count

Discussion
Multiplies vector `B` by the complex conjugates of vector `A` and stores the results in vector `B`.

$$C_{nK} = A^{*}_{nI} B_{nJ}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvcmulD

Complex vector conjugate and multiply; double precision.

```
void vDSP_zvcmulD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for A

B
: Double-precision complex input vector

J
: Stride for B

B
: Double-precision complex output vector

K
: Stride for B

N
: Count

Discussion

Multiplies vector B by the complex conjugates of vector A and stores the results in vector B.

$$C_{nK} = A^*_{nI} B_{nJ}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvconj

Complex vector conjugate; single precision.

```
void vDSP_zvconj (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

C
: Single-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion

Conjugates vector `A`.

$$C_{nk} = A*_{ni}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvconjD

Complex vector conjugate; double precision.

```
void vDSP_zvconjD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for `A`

C
: Double-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion

Conjugates vector `A`.

$$C_{nk} = A*_{ni}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvdiv

Complex vector divide; single precision.

```
void vDSP_zvdiv (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

B
: Single-precision complex input vector

J
: Stride for `B`

C
: Single-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion

Divides vector `B` by vector `A`.

$$C_{nK} = \frac{B_{nJ}}{A_{nI}} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvdivD

Complex vector divide; double precision.

```
void vDSP_zvdivD (
   DSPDoubleSplitComplex *A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_B,
   vDSP_Stride __vDSP_J,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for `A`

B
: Double-precision complex input vector

J
: Stride for `B`

C
: Double-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion
Divides vector `B` by vector `A`.

$$C_{nK} = \frac{B_{nJ}}{A_{nI}} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvfill

Complex vector fill; single precision.

```
void vDSP_zvfill (
   DSPSplitComplex *__vDSP_A,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input scalar

C
: Single-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion
Sets each element in complex vector `C` to complex scalar `A`.

$$C_{nK} = A \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvfillD

Complex vector fill; double precision.

```
void vDSP_zvfillD (
   DSPDoubleSplitComplex *__vDSP_A,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input scalar

C
: Double-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion

Sets each element in complex vector `C` to complex scalar `A`.

$$C_{nK} = A \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmags

Complex vector magnitudes squared; single precision.

```
void vDSP_zvmags (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

C
: Single-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Calculates the squared magnitudes of complex vector `A`.

$$C_{nK} = (Re[A_{nI}])^2 + (Im[A_{nI}])^2 \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmagsD

Complex vector magnitudes squared; double precision.

```
void vDSP_zvmagsD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for `A`

C
: Double-precision real output vector

K
: Stride for `C`

N
: Count

Discussion

Calculates the squared magnitudes of complex vector `A`.

$$C_{nK} = (Re[A_{nI}])^2 + (Im[A_{nI}])^2 \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmgsa

Complex vector magnitudes square and add; single precision.

```
void vDSP_zvmgsa (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_B,
   vDSP_Stride __vDSP_J,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
Stride for A

B
Single-precision real input vector

J
Stride for B

C
Single-precision real output vector

K
Stride for C

N
Count

Discussion
Adds the squared magnitudes of complex vector A to real vector B and store the results in real vector C.

$$C_{nK} = [Re[A_{nI}]]^2 + [Im[A_{nI}]]^2 + B_{nJ} \qquad n = \{0, N-1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvmgsaD

Complex vector magnitudes square and add; double precision.

```
void vDSP_zvmgsaD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_B,
   vDSP_Stride __vDSP_J,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
Double-precision complex input vector

I
Stride for A

B
Double-precision real input vector

J
Stride for B

C
Double-precision real output vector

K

Stride for C

N

Count

Discussion

Adds the squared magnitudes of complex vector A to real vector B and store the results in real vector C.

$$C_{nK} = [Re[A_{nI}]]^2 + [Im[A_{nI}]]^2 + B_{nJ} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmov

Complex vector copy; single precision.

```
void vDSP_zvmov (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision complex input vector

I

Stride for A

C

Single-precision complex output vector

K

Stride for C

N

Count

Discussion

Copies complex vector A to complex vector C.

$$C_{nK} = A_{nI}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmovD

Complex vector copy; double precision.

```
void vDSP_zvmovD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for A

C
: Double-precision complex output vector

K
: Stride for C

N
: Count

Discussion

Copies complex vector A to complex vector C.

$$C_{nK} = A_{nI}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvmul

Multiplies complex vectors A and B and leaves the result in complex vector C; single precision.

```
void vDSP_zvmul (
   const DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   const DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   const DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size,
   int __vDSP_conjugate
);
```

Discussion

Pass 1 or -1 for F, for normal or conjugate multiplication, respectively. Results are undefined for other values of F.

$Re[C_{nK}] = Re[A_{nI}]Re[B_{nJ}] - F(Im[A_{nI}]Im[B_{nJ}])$

$Im[C_{nK}] = Re[A_{nI}]Im[B_{nJ}] + F(Im[A_{nI}]Re[B_{nJ}])$

n = {0, N-1}

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvmulD

Multiplies complex vectors `A` and `B` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zvmulD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size,
   int __vDSP_conjugate
);
```

Discussion
Pass 1 or -1 for `F`, for normal or conjugate multiplication, respectively. Results are undefined for other values of `F`.

$Re[C_{nK}] = Re[A_{nI}]Re[B_{nJ}] - F(Im[A_{nI}]Im[B_{nJ}])$

$Im[C_{nK}] = Re[A_{nI}]Im[B_{nJ}] + F(Im[A_{nI}]Re[B_{nJ}])$

n = {0, N-1}

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvneg

Complex vector negate; single precision.

```
void vDSP_zvneg (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Single-precision complex input vector

I

Stride for `A`

C

Single-precision complex output vector

K

Stride for `C`

N

Count

Discussion

Computes the negatives of the values of complex vector `A` and puts them into complex vector `C`.

$$C_{nK} = -A_{nI}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvnegD

Complex vector negate; double precision.

```
void vDSP_zvnegD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A

Double-precision complex input vector

I

Stride for `A`

C

Double-precision complex output vector

K
: Stride for C

N
: Count

Discussion

Computes the negatives of the values of complex vector A and puts them into complex vector C.

$$C_{nK} = -A_{nI}$$

Availability

Available in iOS 4.0 and later.

Declared In

vDSP.h

vDSP_zvphas

Complex vector phase; single precision.

```
void vDSP_zvphas (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   float *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for A

C
: Single-precision real output vector

K
: Stride for C

N
: Count

Discussion

Finds the phase values, in radians, of complex vector A and store the results in real vector C. The results are between -pi and +pi. The sign of the result is the sign of the second coordinate in the input vector.

$$C_{nK} = \operatorname{atan}\frac{Im[A_{nI}]}{Re[A_{nI}]} \qquad n = \{0, N-1\}$$

Availability

Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvphasD

Complex vector phase; double precision.

```
void vDSP_zvphasD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   double *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
  Double-precision complex input vector

I
  Stride for `A`

C
  Double-precision real output vector

K
  Stride for `C`

N
  Count

Discussion
Finds the phase values, in radians, of complex vector `A` and store the results in real vector `C`. The results are between -pi and +pi. The sign of the result is the sign of the second coordinate in the input vector.

$$C_{nK} = \operatorname{atan}\frac{Im[A_{nI}]}{Re[A_{nI}]} \qquad \mathrm{n} = \{0, \mathrm{N}\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvsma

Complex vector scalar multiply and add; single precision.

```
void vDSP_zvsma (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   DSPSplitComplex *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

B
: Single-precision complex input scalar

C
: Single-precision real input vector

K
: Stride for `C`

D
: Single-precision real output vector

L
: Stride for `C`

N
: Count

Discussion

Multiplies vector `A` by scalar `B` and add the products to vector `C`. The result is stored in vector `D`.

$$D_{nL} = A_{nI}B + C_{nK}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvsmaD

Complex vector scalar multiply and add; double precision.

```
void vDSP_zvsmaD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   DSPDoubleSplitComplex *__vDSP_D,
   vDSP_Stride __vDSP_L,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for A

B
: Double-precision complex input scalar

C
: Double-precision real input vector

K
: Stride for C

D
: Double-precision real output vector

L
: Stride for C

N
: Count

Discussion

Multiplies vector A by scalar B and add the products to vector C. The result is stored in vector D.

$$D_{nL} = A_{nI}B + C_{nK}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvsub

Subtracts complex vector B from complex vector A and leaves the result in complex vector C; single precision.

```
void vDSP_zvsub (
   DSPSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
: Input vector

I
: Address stride for `A`

B
: Input vector

J
: Address stride for `B`

C
: Output vector

K
: Address stride for `C`

N
: Complex element count

Discussion

This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad n = \{0, N\text{-}1\}$$

Availability

Available in iOS 4.0 and later.

Declared In

`vDSP.h`

vDSP_zvsubD

Subtracts complex vector `B` from complex vector `A` and leaves the result in complex vector `C`; double precision.

```
void vDSP_zvsubD (
   DSPDoubleSplitComplex *__vDSP_input1,
   vDSP_Stride __vDSP_stride1,
   DSPDoubleSplitComplex *__vDSP_input2,
   vDSP_Stride __vDSP_stride2,
   DSPDoubleSplitComplex *__vDSP_result,
   vDSP_Stride __vDSP_strideResult,
   vDSP_Length __vDSP_size
);
```

Parameters

A
    Input vector

I
    Address stride for A

B
    Input vector

J
    Address stride for B

C
    Output vector

K
    Address stride for C

N
    Complex element count

Discussion
This performs the following operation:

$$C_{nK} = B_{nJ} - A_{nI} \qquad n = \{0, N\text{-}1\}$$

Availability
Available in iOS 4.0 and later.

Declared In
vDSP.h

vDSP_zvzsml

Complex vector multiply by complex scalar; single precision.

```
void vDSP_zvzsml (
   DSPSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPSplitComplex *__vDSP_B,
   DSPSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Single-precision complex input vector

I
: Stride for `A`

B
: Single-precision complex input scalar

C
: Single-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion
This performs the following operation:

$$C_{nK} = A_{nI}B$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_zvzsmlD

Complex vector multiply by complex scalar; double precision.

```
void vDSP_zvzsmlD (
   DSPDoubleSplitComplex *__vDSP_A,
   vDSP_Stride __vDSP_I,
   DSPDoubleSplitComplex *__vDSP_B,
   DSPDoubleSplitComplex *__vDSP_C,
   vDSP_Stride __vDSP_K,
   vDSP_Length __vDSP_N
);
```

Parameters

A
: Double-precision complex input vector

I
: Stride for `A`

B
: Double-precision complex input scalar

C
: Double-precision complex output vector

K
: Stride for `C`

N
: Count

Discussion
This performs the following operation:

$$C_{nK} = A_{nI} B$$

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

Data Types

This document describes the data types used by the vDSP portion of the Accelerate framework.

vDSP_Length

Used for numbers of elements in arrays and indices of elements in arrays. It is also used for the base-two logarithm of numbers of elements.

```
typedef unsigned long vDSP_Length;
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

vDSP_Stride

Used to hold differences between indices of elements, including the lengths of strides.

```
typedef long vDSP_Stride;
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

DSPComplex

Used to hold a complex value.

```
struct DSPComplex {
    float real;
    float imag;
};

typedef struct DSPComplex DSPComplex;
```

Discussion
Complex data are stored as ordered pairs of floating-point numbers. Because they are stored as ordered pairs, complex vectors require address strides that are multiples of two.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

DSPSplitComplex

Used to represent a complex number when the real and imaginary parts are stored in separate arrays.

```
struct DSPSplitComplex {
    float *realp;
    float *imagp;
};

typedef struct DSPSplitComplex DSPSplitComplex;
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

DSPDoubleComplex

Used to hold a double-precision complex value.

```
struct DSPDoubleComplex {
    double real;
    double imag;
};
```

```
typedef struct DSPDoubleComplex         DSPDoubleComplex;
```

Discussion
Double complex data are stored as ordered pairs of doiuble-precision floating-point numbers. Because they are stored as ordered pairs, complex vectors require address strides that are multiples of two.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

DSPDoubleSplitComplex

Used to represent a double-precision complex number when the real and imaginary parts are stored in separate arrays.

```
struct DSPDoubleSplitComplex {
  double *realp;
  double *imagp;
};

typedef struct DSPDoubleSplitComplex DSPDoubleSplitComplex;
```

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

FFTSetup

An opaque type that contains setup information for a given FFT transform.

```
typedef struct OpaqueFFTSetup*          FFTSetup;
```

Discussion
A setup object can be allocated with `vDSP_create_fftsetup` (page 195) and destroyed with `vDSP_destroy_fftsetup` (page 201). The setup object includes, among other things, precomputed tables used in computing an FFT of the specified size.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

FFTSetupD

An opaque type that contains setup information for a given double-precision FFT transform.

```
typedef struct OpaqueFFTSetupD*         FFTSetupD;
```

Discussion
A setup object can be allocated with `vDSP_create_fftsetupD` (page 196) and destroyed with `vDSP_destroy_fftsetupD` (page 202). The setup object includes, among other things, precomputed tables used in computing an FFT of the specified size.

Availability
Available in iOS 4.0 and later.

Declared In
`vDSP.h`

Constants

vDSP Compile-Time Version Information

The version of vDSP (at compile time).

```
#define vDSP_Version0   268
#define vDSP_Version1   0
```

Constants

`vDSP_Version0`

The vDSP major version.

Available in iOS 4.0 and later.

Declared in `vDSP.h`.

`vDSP_Version1`

The vDSP minor version.

Available in iOS 4.0 and later.

Declared in `vDSP.h`.

vDSP_DFT_Direction

Specifies whether to perform a forward or inverse DFT.

```
typedef enum {
    vDSP_DFT_FORWARD = +1,
    vDSP_DFT_INVERSE = -1
} vDSP_DFT_Direction;
```

Constants

`vDSP_DFT_FORWARD`

Specifies a forward transform.

Available in iOS 4.0 and later.

Declared in `vDSP.h`.

`vDSP_DFT_INVERSE`
: Specifies an inverse transform.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

FFTDirection

Specifies whether to perform a forward or inverse FFT.

```
enum {
    kFFTDirection_Forward         = 1,
    kFFTDirection_Inverse         = -1
};
typedef int FFTDirection;
```

Constants

`kFFTDirection_Forward`
: Specifies a forward transform.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

`kFFTDirection_Inverse`
: Specifies an inverse transform.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

FFTRadix

The size of the FFT decomposition.

```
enum {
    kFFTRadix2                    = 0,
    kFFTRadix3                    = 1,
    kFFTRadix5                    = 2
};
typedef int FFTRadix;
```

Constants

`kFFTRadix2`
: Specifies a radix of 2.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

`kFFTRadix3`
: Specifies a radix of 3.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

`kFFTRadix5`
: Specifies a radix of 5.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

Discussion

An `FFTRadix` value is passed as an argument to `vDSP_create_fftsetup` (page 195) or `vDSP_create_fftsetupD` (page 196).

FFTWindow

Specifies the windowing mode for data values in an FFT or reverse FFT.

```
enum {
    vDSP_HALF_WINDOW                = 1,
    vDSP_HANN_DENORM                = 0,
    vDSP_HANN_NORM                  = 2
};
```

Constants

`vDSP_HALF_WINDOW`
: Specifies that the window should only contain the bottom half of the values (`0` to `(N+1)/2`).

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

`vDSP_HANN_DENORM`
: Specifies a denormalized Hann window.

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

`vDSP_HANN_NORM`
: Specifies a normalized Hann window

  Available in iOS 4.0 and later.

  Declared in `vDSP.h`.

Discussion

Passed as a flag to `vDSP_blkman_window` (page 192) or `vDSP_blkman_windowD` (page 192) to specify the desired type of window..

Document Revision History

This table describes the changes to *Accelerate Framework Reference*.

Date	Notes
2010-03-19	Added document to iOS documentation set.
2010-01-19	New document that describes the Accelerate framework, a C-based API for performing matrix math, digital signal processing, and image manipulation.