# vecLib Framework Reference

**Performance**

**2009-01-06**

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

# Contents

# vecLib Framework Reference

| | |
|---|---|
| **Framework:** | vecLib |
| **Declared in** | vBasicOps.h<br>vBigNum.h<br>vForce.h<br>vecLibTypes.h<br>vectorOps.h<br>vfp.h |

## Overview

The vecLib framework contains nine C header files (not counting vecLib.h which merely includes the others). Two of them, vDSP.h and vDSP_translate.h, are covered in *vDSP Library*.

Three of the header files are Apple's versions of well-known libraries which are described in detail in external references:

- cblas.h and vblas.h are the interfaces to Apple's implementations of BLAS. 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
- clapack.h is the interface to Apple's implementation of LAPACK. Documentation of the LAPACK interfaces, including reference implementations, can be found on the web starting from the LAPACK FAQ page at this URL (verified live as of July 2005): http://netlib.org/lapack/faq.html

This document describes the functions declared in the remaining header files: vecLibTypes.h, vfp.h, vForce.h, vBasicOps.h, vectorOps.h, and vBigNum.h.

These files support the vector mathematical functions library (also called "vMathLib"), which runs on vector processing hardware (Altivec or SSE3) if available. This library abstracts the vector processing capability so that code written for it will execute appropriate instructions for the processor available at runtime.

### vecLibTypes.h

The vecLibTypes.h header file defines a set of vector data types (`vFloat`, `vUInt32`, etc.), which represent 128-bit vectors containing values of type `float`, `UInt32`, etc. The vBasicOps.h and vfp.h headers make use of these types.

The type names all begin with the letter "v," followed by a mnemonic for the scalar data type used for elements of the vector. For example, `vUInt32`, `vSInt16`, `vFloat`, etc.

## vBasicOps.h

vBasicOps.h declares a set of basic arithmetic and logical functions on 128-bit vectors, using the integer types from vecLibTypes.h.

The function names begin with "v," followed by a mnemonic for the type of operation, e.g. "S" or "U" for signed or unsigned, then the width of the operation, then the name of the operation. For example, `vS8Divide` performs division of signed 8-bit values packed into 128-bit vectors.

## vfp.h

vBasicOps.h declares a set of floating-point arithmetic, transcendental and trigonometric functions, on 128-bit vectors, using the floating-point types from vecLibTypes.h.

These functions are named with their customary mathematical names, prefixed with the letter "v", and all except `vtablelookup()` have the suffix "f" to indicate that they work with single-precision floating-point data. For example, `vcosf` is the single-precision cosine function.

## vForce.h

vForce.h declares a set of trigonometric and transcendental functions in terms of C arrays (`double *` or `float *`), which can be of any desired length. Internally, the C arrays are converted piecewise into collections of 128-bit vectors, if appropriate for the current architecture.

The functions declared in vForce.h are named with the customary mathematical names, but with the prefix "vv." Each mathematical function is available in two variants: one for single-precision floating-point data and one for double-precision data. The single-precision forms have the suffix "f", while the double-precision forms have no suffix. For example, `vvcosf` is the single-precision cosine function, while `vvcos` is the double-precision variant.

All of the vForce.h functions follow a common format:

- The return type is `void`.
- The first parameter points to an array to hold the results. (The only exceptions are `vvsincosf()` and `vvsincos()`, which have two result arrays pointed to by the first two parameters.)
- One or more parameters point to operand arrays, the same length as the result array.
- The last parameter is the array length.

## vectorOps.h

vectorOps.h declares a set of vector and matrix BLAS functions on arrays of 128-bit vectors containing single-precision floating-point values. The arrays can be of any desired length, but the number of `float` elements must be a multiple of 4.

## vBigNum.h

vBigNum.h provides arithmetic and logical operations on large integers, which may be 128, 256, 512, or 1024 bits in length. It defines types for these values, and internally processes them as collections of 128-bit vectors.

vBigNum.h defines its own set of data types to represent large integer quantities, such as `vS128` for a signed, 128-bit integer or `vU1024` for an unsigned, 1025-bit integer. The function names begin with the data type name, followed by the name of the operation. For example, `vS512Add` performs addition of two 128-bit signed integers.

The functions perform logical and arithmetic operations on scalar values that may be 128, 256, 512, or 1024 bits in width. These values are implemented as structures of one, two, four, or eight 128-bit vectors, and the operations execute on the available vector-processing hardware if possible.

The functions have names that are compatible with those in vBasicOps.h.

## For More Information

For information about membership in Apple's developer program, go to this URL:

http://developer.apple.com/membership/

For information about the Velocity Engine, go to this URL:

http://developer.apple.com/hardwaredrivers/ve/index.html

For general technical support from Apple, go to this URL:

http://developer.apple.com/technicalsupport/index.html

# Functions by Task

### Shift and Rotate Functions (from vBasicOps.h)

`vLL64Shift` (page 36)
64-bit logical left shift.

`vLL64Shift2` (page 37)
64-bit logical left shift with two shift factors.

`vLR64Shift` (page 38)
64-bit logical right shift.

`vLR64Shift2` (page 39)
64-bit logical right shift with two shift factors.

`vA64Shift` (page 28)
64-bit arithmetic shift.

`vA64Shift2` (page 28)
64-bit arithmetic shift with two shift factors.

vA128Shift (page 27)
128-bit arithmetic shift.

vL64Rotate (page 35)
64-bit left rotate.

vR64Rotate (page 40)
64-bit right rotate.

vL64Rotate2 (page 35)
64-bit left rotate with two rotation factors.

vR64Rotate2 (page 41)
64-bit right rotate with two rotation factors.

vL128Rotate (page 34)
128-bit left rotate.

vR128Rotate (page 40)
128-bit right rotate.

## Integer Arithmetic Functions (from vBasicOps.h)

vU64Add (page 83)
Unsigned 64-bit addition (modular arithmetic).

vU64AddS (page 83)
Unsigned 64-bit addition with saturation (clipping).

vS64Add (page 52)
Signed 64-bit addition (modular arithmetic).

vS64AddS (page 53)
Signed 64-bit addition with saturation (clipping).

vU128Add (page 74)
Unsigned 128-bit addition (modular arithmetic).

vU128AddS (page 75)
Unsigned 128-bit addition with saturation (clipping).

vS128Add (page 44)
Signed 128-bit addition (modular arithmetic).

vS128AddS (page 44)
Signed 128-bit addition with saturation (clipping).

vU64Sub (page 84)
Unsigned 64-bit subtraction (modular arithmetic).

vU64SubS (page 84)
Unsigned 64-bit subtraction with saturation (clipping).

vS64Sub (page 54)
Signed 64-bit subtraction (modular arithmetic).

vS64SubS (page 54)
Signed 64-bit subtraction with saturation (clipping).

vU128Sub (page 76)
Unsigned 128-bit subtraction (modular arithmetic).

vU128SubS (page 76)
Unsigned 128-bit subtraction with saturation (clipping).

vS128Sub (page 45)
Signed 128-bit subtraction (modular arithmetic).

vS128SubS (page 45)
Signed 128-bit subtraction with saturation (clipping).

vU8HalfMultiply (page 85)
Unsigned 8-bit multiplication; results are same width as multiplicands.

vS8HalfMultiply (page 54)
Signed 8-bit multiplication; results are same width as multiplicands.

vU16HalfMultiply (page 76)
Unsigned 16-bit multiplication; results are same width as multiplicands.

vS16HalfMultiply (page 46)
Signed 16-bit multiplication; results are same width as multiplicands.

vU32HalfMultiply (page 80)
Unsigned 32-bit multiplication; results are same width as multiplicands.

vS32HalfMultiply (page 49)
Signed 32-bit multiplication; results are same width as multiplicands.

vU64HalfMultiply (page 84)
Unsigned 64-bit multiplication; results are same width as multiplicands.

vS64HalfMultiply (page 54)
Signed 64-bit multiplication; results are same width as multiplicands.

vU128HalfMultiply (page 75)
Unsigned 128-bit multiplication; results are same width as multiplicands.

vS128HalfMultiply (page 45)
Signed 128-bit multiplication; results are same width as multiplicands.

vU32FullMulEven (page 79)
Unsigned 32-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

vU32FullMulOdd (page 80)
Unsigned 32-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

vS32FullMulEven (page 49)
Signed 32-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

vS32FullMulOdd (page 49)
Signed 32-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

vU64FullMulEven (page 83)
Unsigned 64-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

`vU64FullMulOdd` (page 84)
Unsigned 64-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

`vS64FullMulEven` (page 53)
Signed 64-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

`vS64FullMulOdd` (page 53)
Signed 64-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

`vU8Divide` (page 84)
Unsigned 8-bit division.

`vS8Divide` (page 54)
Signed 8-bit division.

`vU16Divide` (page 76)
Unsigned 16-bit division.

`vS16Divide` (page 46)
Signed 16-bit division.

`vU32Divide` (page 79)
Unsigned 32-bit division.

`vS32Divide` (page 49)
Signed 32-bit division.

`vU64Divide` (page 83)
Unsigned 64-bit division.

`vS64Divide` (page 53)
Signed 64-bit division.

`vU128Divide` (page 75)
Unsigned 128-bit division.

`vS128Divide` (page 44)
Signed 128-bit division.

## Floating-Point Arithmetic and Auxiliary Functions (from vfp.h)

`vdivf` (page 31)
For each vector element, calculates `A/B`.

`vsqrtf` (page 68)
For each vector element, calculates the square root of `X`.

`vrsqrtf` (page 41)
For each vector element, calculates the inverse of the square root of `X`.

`vfabf` (page 32)
For each vector element, calculates the absolute value of `v`.

`vcopysignf` (page 30)
For each vector element, produces a value with the magnitude of `arg2` and sign `arg1`. Note that the order of the arguments matches the recommendation of the IEEE 754 floating-point standard, which is opposite from the SANE copysign function.

`vnextafterf` (page 39)
: For each vector element, calculates the next representable value after `x` in the direction of `y`. If `x` is equal to `y`, then `y` is returned.

`vtablelookup` (page 71)
: For each vector element of `Index_Vect`, returns the corresponding value from `Table`.

## Exponential and Logarithmic Functions (from vfp.h)

`vexpf` (page 31)
: For each vector element, calculates the exponential of X.

`vexpm1f` (page 31)
: For each vector element, calculates ExpM1(x) = Exp(x) - 1. But, for small enough arguments, ExpM1(x) is expected to be more accurate than Exp(x) - 1.

`vlogf` (page 37)
: For each vector element, calculates the natural logarithm of `X`.

`vlog1pf` (page 37)
: For each vector element, calculates Log1P = Log(1 + x). But, for small enough arguments, Log1P is expected to be more accurate than Log(1 + x).

`vlogbf` (page 37)
: For each vector element, extracts the exponent of `X`, as a signed integral value. A subnormal argument is treated as though it were first normalized. Thus: 1 <= x * 2^(-logb(x)) < 2.

`vscalbf` (page 56)
: For each vector element, calculates x * 2^n efficiently. This is not normally done by computing 2^n explicitly.

## Trigonometric Functions (from vfp.h)

`vsinf` (page 65)
: For each vector element, calculates the sine.

`vcosf` (page 30)
: For each vector element, calculates the cosine.

`vtanf` (page 71)
: For each vector element, calculates the tangent.

`vasinf` (page 29)
: For each vector element, calculates the arcsine. Results are in the interval [-pi/2, pi/2].

`vacosf` (page 28)
: For each vector element, calculates the arccosine. Results are in the interval [0, pi].

`vatanf` (page 29)
: For each vector element, calculates the arctangent. Results are in the interval [-pi/2, pi/2].

`vatan2f` (page 29)
: For each vector element, calculates the arctangent of `arg2/arg1` in the interval [-pi,pi] using the sign of both arguments to determine the quadrant of the computed value.

## Hyperbolic Functions (from vfp.h)

`vsinhf` (page 65)
For each vector element, calculates the hyperbolic sine of `X`.

`vcoshf` (page 31)
For each vector element, calculates the hyperbolic cosine of `X`.

`vtanhf` (page 72)
For each vector element, calculates the hyperbolic tangent of `X`.

`vasinhf` (page 29)
For each vector element, calculates the inverse hyperbolic sine of `X`.

`vacoshf` (page 28)
For each vector element, calculates the inverse hyperbolic cosine of `X`.

`vatanhf` (page 30)
For each vector element, calculates the inverse hyperbolic tangent of `X`.

## Power Functions (from vfp.h)

`vipowf` (page 32)
For each vector element, calculates `X` to the integer power of `Y`.

`vpowf` (page 39)
For each vector element, calculates `X` to the floating-point power of `Y`. The result is more accurate than using exp(log(`X`)*`Y`).

## Remainder Functions (from vfp.h)

`vfmodf` (page 32)
For each vector element, calculates `X` modulo `Y`.

`vremainderf` (page 41)
For each vector element, calculates the remainder of `X`/`Y`, according to the IEEE 754 floating-point standard.

`vremquof` (page 41)
For each vector element, calculates the remainder of `X`/`Y`, according to the SANE standard. It stores into `QUO` the 7 low-order bits of the integer quotient, such that -127 <= `QUO` <= 127.

## Inquiry Functions (from vfp.h)

`vclassifyf` (page 30)
For each vector element, returns the class of the argument (one of the FP_ ... constants defined in math.h).

`vsignbitf` (page 65)
For each vector element, returns a non-zero value if and only if the sign of `arg` is negative. This includes NaNs, infinities and zeros.

## Array-Oriented Arithmetic and Auxiliary Functions (from vForce.h)

`vvrecf` (page 97)
For each single-precision array element, sets `y` to the reciprocal of `y`.

`vvrec` (page 96)
For each double-precision array element, sets `y` to the reciprocal of `y`.

`vvdivf` (page 92)
For each single-precision array element, sets `z` to `y/x`.

`vvdiv` (page 92)
For each double-precision array element, sets `z` to `y/x`.

`vvsqrtf` (page 100)
For each single-precision array element, sets `y` to the square root of `x`.

`vvsqrt` (page 99)
For each double-precision array element, sets `y` to the square root of `x`.

`vvrsqrtf` (page 97)
For each single-precision array element, sets `y` to the reciprocal of the square root of `x`.

`vvrsqrt` (page 97)
For each double-precision array element, sets `y` to the reciprocal of the square root of `x`.

`vvintf` (page 94)
For each single-precision array element, sets `y` to the integer truncation of `x`.

`vvint` (page 93)
For each double-precision array element, sets `y` to the integer truncation of `x`.

`vvnintf` (page 95)
For each single-precision array element, sets `y` to the nearest integer to `x`.

`vvnint` (page 95)
For each double-precision array element, sets `y` to the nearest integer to `x`.

`vvceilf` (page 89)
For each single-precision array element, sets `y` to the ceiling of `x`.

`vvceil` (page 89)
For each double-precision array element, sets `y` to the ceiling of `x`.

`vvfloorf` (page 93)
For each single-precision array element, sets `y` to the floor of `x`.

`vvfloor` (page 93)
For each double-precision array element, sets `y` to the floor of `x`.

## Array-Oriented Exponential and Logarithmic Functions (from vForce.h)

`vvexpf` (page 92)
For each single-precision array element, sets `y` to the exponential of `x`.

`vvexp` (page 92)
For each double-precision array element, sets `y` to the exponential of `x`.

`vvlogf` (page 95)
For each single-precision array element, sets `y` to the natural logarithm of `x`.

`vvlog` (page 94)
For each double-precision array element, sets `y` to the natural logarithm of `x`.

`vvlog10f` (page 94)
For each single-precision array element, sets `y` to the base 10 logarithm of `x`.

`vvlog10` (page 94)
For each double-precision array element, sets `y` to the base 10 logarithm of `x`.

## Array-Oriented Power Functions (from vForce.h)

`vvpowf` (page 96)
For each single-precision array element, sets `z` to `x` raised to the power of `y`.

`vvpow` (page 96)
For each double-precision array element, sets `z` to `x` raised to the power of `y`.

## Array-Oriented Trigonometric Functions (from vForce.h)

`vvsinf` (page 98)
For each single-precision array element, sets `y` to the sine of `x`.

`vvsin` (page 97)
For each double-precision array element, sets `y` to the sine of `x`.

`vvcosf` (page 90)
For each single-precision array element, sets `y` to the cosine of `x`.

`vvcos` (page 90)
For each double-precision array element, sets `y` to the cosine of `x`.

`vvtanf` (page 100)
For each single-precision array element, sets `y` to the tangent of `x`.

`vvtan` (page 100)
For each double-precision array element, sets `y` to the tangent of `x`.

`vvasinf` (page 86)
For each single-precision array element, sets `y` to the arcsine of `x`.

`vvasin` (page 86)
For each double-precision array element, sets `y` to the arcsine of `x`.

`vvacosf` (page 85)
For each single-precision array element, sets `y` to the arccosine of `x`.

`vvacos` (page 85)
For each double-precision array element, sets `y` to the arccosine of `x`.

`vvatanf` (page 88)
For each single-precision array element, sets `y` to the arctangent of `x`.

`vvatan` (page 87)
For each double-precision array element, sets `y` to the arctangent of `x`.

`vvatan2f` (page 88)
For each single-precision array element, sets `z` to the arctangent of `y/x`.

`vvatan2` (page 88)
For each double-precision array element, sets `z` to the arctangent of `y/x`.

`vvsincosf` (page 98)
For each single-precision array element, sets `z` to the sine of `x` and `y` to the cosine of `x`.

`vvsincos` (page 98)
For each double-precision array element, sets `z` to the sine of `x` and `y` to the cosine of `x`.

`vvcosisinf` (page 91)
For each single-precision array element, sets the real part of `C` to the sine of `x` and the imaginary part of `C` to the cosine of `x`.

`vvcosisin` (page 91)
For each double-precision array element, sets the real part of `C` to the sine of `x` and the imaginary part of `C` to the cosine of `x`.

## Array-Oriented Hyperbolic Functions (from vForce.h)

`vvsinhf` (page 99)
For each single-precision array element, sets `y` to the hyperbolic sine of `x`.

`vvsinh` (page 99)
For each double-precision array element, sets `y` to the hyperbolic sine of `x`.

`vvcoshf` (page 91)
For each single-precision array element, sets `y` to the hyperbolic cosine of `x`.

`vvcosh` (page 90)
For each double-precision array element, sets `y` to the hyperbolic cosine of `x`.

`vvtanhf` (page 101)
For each single-precision array element, sets `y` to the hyperbolic tangent of `x`.

`vvtanh` (page 100)
For each double-precision array element, sets `y` to the hyperbolic tangent of `x`.

`vvasinhf` (page 87)
For each single-precision array element, sets `y` to the inverse hyperbolic sine of `x`.

`vvasinh` (page 87)
For each double-precision array element, sets `y` to the inverse hyperbolic sine of `x`.

`vvacoshf` (page 86)
For each single-precision array element, sets `y` to the inverse hyperbolic cosine of `x`.

`vvacosh` (page 86)
For each double-precision array element, sets `y` to the inverse hyperbolic cosine of `x`.

`vvatanhf` (page 89)
For each single-precision array element, sets `y` to the inverse hyperbolic tangent of `x`.

`vvatanh` (page 89)
For each double-precision array element, sets `y` to the inverse hyperbolic tangent of `x`.

## Shift and Rotate Functions on Big Numbers (from vBigNum.h)

`vLL256Shift` (page 36)
256-bit logical left shift.

`vLR256Shift` (page 38)
256-bit logical right shift.

`vA256Shift` (page 27)
256-bit arithmetic shift.

`vLL512Shift` (page 36)
512-bit logical left shift.

`vLR512Shift` (page 38)
512-bit logical right shift .

`vA512Shift` (page 27)
512-bit arithmetic shift.

`vLL1024Shift` (page 36)
1024-bit logical left shift.

`vLR1024Shift` (page 38)
1024-bit logical right shift .

`vA1024Shift` (page 27)
1024-bit arithmetic shift.

`vL256Rotate` (page 35)
256-bit left rotate.

`vR256Rotate` (page 40)
256-bit right rotate.

`vL512Rotate` (page 35)
512-bit left rotate.

`vR512Rotate` (page 40)
512-bit right rotate.

`vL1024Rotate` (page 34)
1024-bit left rotate.

`vR1024Rotate` (page 39)
1024-bit right rotate.

## Arithmetic Functions on Big Numbers (from vBigNum.h)

`vU256Add` (page 76)
Unsigned 256-bit addition (modular arithmetic).

`vU256AddS` (page 77)
Unsigned 256-bit addition with saturation (clipping).

`vS256Add` (page 46)
Signed 256-bit addition (modular arithmetic).

`vS256AddS` (page 46)
Signed 256-bit addition with saturation (clipping).

vU512Add (page 80)
Unsigned 512-bit addition (modular arithmetic).

vU512AddS (page 80)
Unsigned 512-bit addition with saturation (clipping).

vS512Add (page 50)
Signed 512-bit addition (modular arithmetic).

vS512AddS (page 50)
Signed 512-bit addition with saturation (clipping).

vU1024Add (page 72)
Unsigned 1024-bit addition (modular arithmetic).

vU1024AddS (page 72)
Unsigned 1024-bit addition with saturation (clipping).

vS1024Add (page 42)
Signed 1024-bit addition (modular arithmetic).

vS1024AddS (page 42)
Signed 1024-bit addition with saturation (clipping).

vU256Sub (page 78)
Unsigned 256-bit subtraction (modular arithmetic).

vU256SubS (page 79)
Unsigned 256-bit subtraction with saturation (clipping).

vS256Sub (page 48)
Signed 256-bit subtraction (modular arithmetic).

vS256SubS (page 48)
Signed 256-bit subtraction with saturation (clipping).

vU512Sub (page 82)
Unsigned 512-bit subtraction (modular arithmetic).

vU512SubS (page 82)
Unsigned 512-bit subtraction with saturation (clipping).

vS512Sub (page 52)
Signed 512-bit subtraction (modular arithmetic).

vS512SubS (page 52)
Signed 512-bit subtraction with saturation (clipping).

vU1024Sub (page 74)
Unsigned 1024-bit subtraction (modular arithmetic).

vU1024SubS (page 74)
Unsigned 1024-bit subtraction with saturation (clipping).

vS1024Sub (page 43)
Signed 1024-bit subtraction (modular arithmetic).

vS1024SubS (page 44)
Signed 1024-bit subtraction with saturation (clipping).

vU256Neg (page 78)
Unsigned 256-bit negation.

vS256Neg (page 48)
Signed 256-bit negation.

vU512Neg (page 82)
Unsigned 512-bit negation.

vS512Neg (page 51)
Signed 512-bit negation.

vU1024Neg (page 73)
Unsigned 1024-bit negation.

vS1024Neg (page 43)
Signed 1024-bit negation.

vU256Mod (page 78)
Unsigned 256-bit mod.

vS256Mod (page 48)
Signed 256-bit mod.

vU512Mod (page 81)
Unsigned 512-bit mod.

vS512Mod (page 51)
Signed 512-bit mod.

vU1024Mod (page 73)
Unsigned 1024-bit mod.

vS1024Mod (page 43)
Signed 256-bit Mod.

vU256HalfMultiply (page 78)
Unsigned 256-bit multiplication; result is the same width as multiplicands.

vS256HalfMultiply (page 47)
Signed 256-bit multiplication; result is the same width as multiplicands.

vU512HalfMultiply (page 81)
Unsigned 512-bit multiplication; result is the same width as multiplicands.

vS512HalfMultiply (page 51)
Signed 512-bit multiplication; result is the same width as multiplicands.

vU1024HalfMultiply (page 73)
Unsigned 1024-bit multiplication; result is the same width as multiplicands.

vS1024HalfMultiply (page 42)
Signed 1024-bit multiplication; result is the same width as multiplicands.

vU128FullMultiply (page 75)
Unsigned 128-bit multiplication; result is twice as wide as multiplicands.

vS128FullMultiply (page 45)
Signed 128-bit multiplication; result is twice as wide as multiplicands.

vU256FullMultiply (page 77)
Unsigned 256-bit multiplication; result is twice as wide as multiplicands.

vS256FullMultiply (page 47)
Signed 256-bit multiplication; result is twice as wide as multiplicands.

vU512FullMultiply (page 81)
Unsigned 512-bit multiplication; result is twice as wide as multiplicands.

vS512FullMultiply (page 51)
Signed 512-bit multiplication; result is twice as wide as multiplicands.

`vU256Divide` (page 77)
Unsigned 256-bit division.

`vS256Divide` (page 47)
Signed 256-bit division.

`vU512Divide` (page 81)
Unsigned 512-bit division.

`vS512Divide` (page 50)
Signed 512-bit division.

`vU1024Divide` (page 73)
Unsigned 1024-bit division.

`vS1024Divide` (page 42)
Signed 1024-bit division.

## Vector-Scalar Linear Algebra Functions (from vectorOps.h)

`vIsamax` (page 32)
Finds the position of the first vector element having the largest absolute value.

`vIsamin` (page 33)
Finds the position of the first vector element having the smallest absolute value.

`vIsmax` (page 33)
Finds the position of the first vector element having the maximum value.

`vIsmin` (page 34)
Finds the position of the first vector element having the minimum value.

`vSasum` (page 55)
Finds the sum of the absolute values of the elements in a vector.

`vSsum` (page 69)
Finds the sum of the values of the elements in a vector.

`vSaxpy` (page 55)
Multiplies a vector by a scalar , adds it to a second vector , and stores the result in the second vector.

`vSnaxpy` (page 66)
Performs the computation of `vSaxpy` `n` times, using a different multiplier each time.

`vScopy` (page 56)
Copies one vector to another.

`vSdot` (page 56)
Computes the dot product of two vectors.

`vSndot` (page 66)
Computes the dot products of n pairs of vectors, accumulating or storing the results in an array of `n` `float` values.

`vSnrm2` (page 67)
Finds the Euclidean length of a vector.

`vSnorm2` (page 67)
Finds the Euclidean length of a vector.

vSrot (page 68)
Applies planar rotation to a set of n points whose x and y coordinates are contained in two arrays of vectors.

vSscal (page 69)
Scales a vector in place.

vSswap (page 70)
Interchanges the elements of two vectors.

vSyax() (page 70)
Multiplies each element of a vector and stores the results in a second vector.

vSzaxpy() (page 71)
Multiplies a vector by a scalar, adds it to a second vector, and stores the result in a third vector.

## Matrix-Vector Linear Algebra Functions (from vectorOps.h)

vSgemv() (page 61)
Multiplies a vector by a scalar. Multiplies a matrix by another scalar, then by a second vector, and adds the resulting vector to the first vector. This function can also perform the calculation with the transpose of the original matrix instead of the matrix itself. A selector parameter determines whether the transpose is used.

vSgemx() (page 62)
Multiplies a matrix by a scalar and then by a vector, and adds the resulting vector to a second vector.

vSgemtx() (page 59)
Forms the transpose of a matrix, multiplies it by a scalar and then by a vector, and adds the resulting vector to a second vector.

## Matrix Operations (from vectorOps.h)

vSgeadd() (page 57)
Matrix addition for general matrices or their transposes.

vSgesub() (page 62)
Matrix subtraction for general matrices or their transposes.

vSgemul() (page 60)
Matrix multiplication for general matrices or their transposes.

vSgemm() (page 58)
Combined matrix multiplication and addition for general matrices or their transposes.

vSgetmi() (page 63)
General matrix transpose, in place.

vSgetmo() (page 64)
General matrix transpose, out of place.

vSgevv() (page 64)
Multiplies two matrices and places the results in a third matrix.

# Functions

### vA1024Shift

1024-bit arithmetic shift.

```
extern void vA1024Shift(
    const vS1024 * a,
    UInt32 shiftAmount,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

### vA128Shift

128-bit arithmetic shift.

```
extern vUInt32 vA128Shift(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vA256Shift

256-bit arithmetic shift.

```
extern void vA256Shift(
    const vS256 * a,
    UInt32 shiftAmount,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

### vA512Shift

512-bit arithmetic shift.

```
extern void vA512Shift(
    const vS512 * a,
    UInt32 shiftAmount,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vA64Shift

64-bit arithmetic shift.

```
extern vUInt32 vA64Shift(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vA64Shift2

64-bit arithmetic shift with two shift factors.

```
extern vUInt32 vA64Shift2(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vacosf

For each vector element, calculates the arccosine. Results are in the interval [0, pi].

```
extern vFloat vacosf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vacoshf

For each vector element, calculates the inverse hyperbolic cosine of `X`.

```
extern vFloat vacoshf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vasinf

For each vector element, calculates the arcsine. Results are in the interval [-pi/2, pi/2].

```
extern vFloat vasinf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vasinhf

For each vector element, calculates the inverse hyperbolic sine of `X`.

```
extern vFloat vasinhf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vatan2f

For each vector element, calculates the arctangent of `arg2/arg1` in the interval [-pi,pi] using the sign of both arguments to determine the quadrant of the computed value.

```
extern vFloat vatan2f(vFloat arg1, vFloat arg2);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vatanf

For each vector element, calculates the arctangent. Results are in the interval [-pi/2, pi/2].

```
extern vFloat vatanf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vatanhf

For each vector element, calculates the inverse hyperbolic tangent of `X`.

```
extern vFloat vatanhf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vclassifyf

For each vector element, returns the class of the argument (one of the FP_ ... constants defined in math.h).

```
extern vUInt32 vclassifyf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vcopysignf

For each vector element, produces a value with the magnitude of `arg2` and sign `arg1`. Note that the order of the arguments matches the recommendation of the IEEE 754 floating-point standard, which is opposite from the SANE copysign function.

```
extern vFloat vcopysignf(vFloat arg2, vFloat arg1);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vcosf

For each vector element, calculates the cosine.

```
extern vFloat vcosf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vcoshf

For each vector element, calculates the hyperbolic cosine of `X`.

```
extern vFloat vcoshf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vdivf

For each vector element, calculates `A/B`.

```
extern vFloat vdivf(vFloat A, vFloat B);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vexpf

For each vector element, calculates the exponential of X.

```
extern vFloat vexpf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vexpm1f

For each vector element, calculates ExpM1(x) = Exp(x) - 1. But, for small enough arguments, ExpM1(x) is expected to be more accurate than Exp(x) - 1.

```
extern vFloat vexpm1f(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vfabf

For each vector element, calculates the absolute value of `v`.

```
extern vFloat vfabf(vFloat v);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vfmodf

For each vector element, calculates `X` modulo `Y`.

```
extern vFloat vfmodf(vFloat X, vFloat Y);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vipowf

For each vector element, calculates `X` to the integer power of `Y`.

```
extern vFloat vipowf(vFloat X, vSInt32 Y);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vlsamax

Finds the position of the first vector element having the largest absolute value.

```
extern SInt32 vIsamax(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The index of the first element having the largest absolute value in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vIsamin

Finds the position of the first vector element having the smallest absolute value.

```
extern SInt32 vIsamin(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The index of the first element having the smallest absolute value in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vIsmax

Finds the position of the first vector element having the maximum value.

```
extern SInt32 vIsmax(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The index of the first element having the maximum value in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vIsmin

Finds the position of the first vector element having the minimum value.

```
extern SInt32 vIsmin(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The index of the first element having the minimum value in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vL1024Rotate

1024-bit left rotate.

```
extern void vL1024Rotate(
    const vU1024 * a,
    UInt32 rotateAmount,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vL128Rotate

128-bit left rotate.

```
extern vUInt32 vL128Rotate(vUInt32 vA, vUInt8 vRotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vL256Rotate

256-bit left rotate.

```
extern void vL256Rotate(
    const vU256 * a,
    UInt32 rotateAmount,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vL512Rotate

512-bit left rotate.

```
extern void vL512Rotate(
    const vU512 * a,
    UInt32 rotateAmount,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vL64Rotate

64-bit left rotate.

```
extern vUInt32 vL64Rotate(vUInt32 vA, vUInt8 vRotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vL64Rotate2

64-bit left rotate with two rotation factors.

```
extern vUInt32 vL64Rotate2(vUInt32 vA, vUInt8 vRotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vLL1024Shift

1024-bit logical left shift.

```
extern void vLL1024Shift(
    const vU1024 * a,
    UInt32 shiftAmount,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

### vLL256Shift

256-bit logical left shift.

```
extern void vLL256Shift(
    const vU256 * a,
    UInt32 shiftAmount,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

### vLL512Shift

512-bit logical left shift.

```
extern void vLL512Shift(
    const vU512 * a,
    UInt32 shiftAmount,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

### vLL64Shift

64-bit logical left shift.

```
extern vUInt32 vLL64Shift(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vLL64Shift2

64-bit logical left shift with two shift factors.

```
extern vUInt32 vLL64Shift2(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vlog1pf

For each vector element, calculates Log1P = Log(1 + x). But, for small enough arguments, Log1P is expected to be more accurate than Log(1 + x).

```
extern vFloat vlog1pf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vlogbf

For each vector element, extracts the exponent of `X`, as a signed integral value. A subnormal argument is treated as though it were first normalized. Thus: 1 <= x * 2^(-logb(x)) < 2.

```
extern vFloat vlogbf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vlogf

For each vector element, calculates the natural logarithm of `X`.

```
extern vFloat vlogf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vLR1024Shift

1024-bit logical right shift .

```
extern void vLR1024Shift(
    const vU1024 * a,
    UInt32 shiftAmount,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vLR256Shift

256-bit logical right shift.

```
extern void vLR256Shift(
    const vU256 * a,
    UInt32 shiftAmount,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vLR512Shift

512-bit logical right shift .

```
extern void vLR512Shift(
    const vU512 * a,
    UInt32 shiftAmount,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vLR64Shift

64-bit logical right shift.

```
extern vUInt32 vLR64Shift(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vLR64Shift2

64-bit logical right shift with two shift factors.

```
extern vUInt32 vLR64Shift2(vUInt32 vA, vUInt8 vShiftFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vnextafterf

For each vector element, calculates the next representable value after `x` in the direction of `y`. If `x` is equal to `y`, then `y` is returned.

```
extern vFloat vnextafterf(vFloat x, vFloat y);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vpowf

For each vector element, calculates `X` to the floating-point power of `Y`. The result is more accurate than using exp(log(`X`)*`Y`).

```
extern vFloat vpowf(vFloat X, vFloat Y);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vR1024Rotate

1024-bit right rotate.

```
extern void vR1024Rotate(
    const vU1024 * a,
    UInt32 rotateAmount,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vR128Rotate

128-bit right rotate.

```
extern vUInt32 vR128Rotate(vUInt32 vA, vUInt8 vRotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vR256Rotate

256-bit right rotate.

```
extern void vR256Rotate(const vU256 * a, UInt32 rotateAmount, vU256
* result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vR512Rotate

512-bit right rotate.

```
extern void vR512Rotate(
    const vU512 * a,
    UInt32 rotateAmount,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vR64Rotate

64-bit right rotate.

```
extern vUInt32 vR64Rotate(vUInt32 vA, vUInt8 RotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vR64Rotate2

64-bit right rotate with two rotation factors.

```
extern vUInt32 vR64Rotate2(vUInt32 vA, vUInt8 vRotateFactor);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vremainderf

For each vector element, calculates the remainder of `X/Y`, according to the IEEE 754 floating-point standard.

```
extern vFloat vremainderf(vFloat X, vFloat Y);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vremquof

For each vector element, calculates the remainder of `X/Y`, according to the SANE standard. It stores into `QUO` the 7 low-order bits of the integer quotient, such that -127 <= `QUO` <= 127.

```
extern vFloat vremquof(vFloat X, vFloat Y, vUInt32 *QUO);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vrsqrtf

For each vector element, calculates the inverse of the square root of `X`.

```
extern vFloat vrsqrtf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vfp.h

## vS1024Add

Signed 1024-bit addition (modular arithmetic).

```
extern void vS1024Add(
    const vS1024 * a,
    const vS1024 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS1024AddS

Signed 1024-bit addition with saturation (clipping).

```
extern void vS1024AddS(
    const vS1024 * a,
    const vS1024 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS1024Divide

Signed 1024-bit division.

```
extern void vS1024Divide(
    const vS1024 * numerator,
    const vS1024 * divisor,
    vS1024 * result,
    vS1024 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS1024HalfMultiply

Signed 1024-bit multiplication; result is the same width as multiplicands.

```
extern void vS1024HalfMultiply(
    const vS1024 * a,
    const vS1024 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS1024Mod

Signed 256-bit Mod.

```
extern void vS1024Mod(
    const vS1024 * numerator,
    const vS1024 * divisor,
    vS1024 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS1024Neg

Signed 1024-bit negation.

```
extern void vS1024Neg(
    const vS1024 * a,
     vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS1024Sub

Signed 1024-bit subtraction (modular arithmetic).

```
extern void vS1024Sub(
    const vS1024 * a,
    const vS1024 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS1024SubS

Signed 1024-bit subtraction with saturation (clipping).

```
extern void vS1024SubS(
    const vS1024 * a,
    const vS1024 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS128Add

Signed 128-bit addition (modular arithmetic).

```
extern vSInt32 vS128Add(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS128AddS

Signed 128-bit addition with saturation (clipping).

```
extern vSInt32 vS128AddS(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS128Divide

Signed 128-bit division.

```
extern vSInt32 vS128Divide( vSInt32 vN, vSInt32 vD, vSInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS128FullMultiply

Signed 128-bit multiplication; result is twice as wide as multiplicands.

```
extern void vS128FullMultiply(
    const vS128 * a,
    const vS128 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS128HalfMultiply

Signed 128-bit multiplication; results are same width as multiplicands.

```
extern vSInt32 vS128HalfMultiply(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS128Sub

Signed 128-bit subtraction (modular arithmetic).

```
extern vSInt32 vS128Sub(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS128SubS

Signed 128-bit subtraction with saturation (clipping).

```
extern vSInt32 vS128SubS(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS16Divide

Signed 16-bit division.

```
extern vSInt16 vS16Divide(vSInt16 vN, vSInt16 vD, vSInt16 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS16HalfMultiply

Signed 16-bit multiplication; results are same width as multiplicands.

```
extern vSInt16 vS16HalfMultiply(vSInt16 vA, vSInt16 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS256Add

Signed 256-bit addition (modular arithmetic).

```
extern void vS256Add(
    const vS256 * a,
    const vS256 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS256AddS

Signed 256-bit addition with saturation (clipping).

```
extern void vS256AddS(
    const vS256 * a,
    const vS256 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS256Divide

Signed 256-bit division.

```
extern void vS256Divide(
    const vS256 * numerator,
    const vS256 * divisor,
    vS256 * result,
    vS256 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS256FullMultiply

Signed 256-bit multiplication; result is twice as wide as multiplicands.

```
extern void vS256FullMultiply(
    const vS256 * a,
    const vS256 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS256HalfMultiply

Signed 256-bit multiplication; result is the same width as multiplicands.

```
extern void vS256HalfMultiply(
    const vS256 * a,
    const vS256 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS256Mod

Signed 256-bit mod.

```
extern void vS256Mod(
    const vS256 * numerator,
    const vS256 * divisor,
    vS256 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS256Neg

Signed 256-bit negation.

```
extern void vS256Neg(
    const vS256 * a,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS256Sub

Signed 256-bit subtraction (modular arithmetic).

```
extern void vS256Sub(
    const vS256 * a,
    const vS256 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS256SubS

Signed 256-bit subtraction with saturation (clipping).

```
extern void vS256SubS(
    const vS256 * a,
    const vS256 * b,
    vS256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS32Divide

Signed 32-bit division.

```
extern vSInt32 vS32Divide(vSInt32 vN, vSInt32 vD, vSInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS32FullMulEven

Signed 32-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vSInt32 vS32FullMulEven(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS32FullMulOdd

Signed 32-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vSInt32 vS32FullMulOdd(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS32HalfMultiply

Signed 32-bit multiplication; results are same width as multiplicands.

```
extern vSInt32 vS32HalfMultiply(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS512Add

Signed 512-bit addition (modular arithmetic).

```
extern void vS512Add(
    const vS512 * a,
    const vS512 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512AddS

Signed 512-bit addition with saturation (clipping).

```
extern void vS512AddS(
    const vS512 * a,
    const vS512 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512Divide

Signed 512-bit division.

```
extern void vS512Divide(
    const vS512 * numerator,
    const vS512 * divisor,
    vS512 * result,
    vS512 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512FullMultiply

Signed 512-bit multiplication; result is twice as wide as multiplicands.

```
extern void vS512FullMultiply(
    const vS512 * a,
    const vS512 * b,
    vS1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512HalfMultiply

Signed 512-bit multiplication; result is the same width as multiplicands.

```
extern void vS512HalfMultiply(
    const vS512 * a,
    const vS512 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512Mod

Signed 512-bit mod.

```
extern void vS512Mod(
    const vS512 * numerator,
    const vS512 * divisor,
    vS512 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vS512Neg

Signed 512-bit negation.

```
extern void vS512Neg(
    const vS512 * a,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS512Sub

Signed 512-bit subtraction (modular arithmetic).

```
extern void vS512Sub(
    const vS512 * a,
    const vS512 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS512SubS

Signed 512-bit subtraction with saturation (clipping).

```
extern void vS512SubS(
    const vS512 * a,
    const vS512 * b,
    vS512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vS64Add

Signed 64-bit addition (modular arithmetic).

```
extern vSInt32 vS64Add(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vS64AddS

Signed 64-bit addition with saturation (clipping).

```
extern vSInt32 vS64AddS(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64Divide

Signed 64-bit division.

```
extern vSInt32 vS64Divide(vSInt32 vN, vSInt32 vD, vSInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64FullMulEven

Signed 64-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vSInt32 vS64FullMulEven(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64FullMulOdd

Signed 64-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vSInt32 vS64FullMulOdd(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64HalfMultiply

Signed 64-bit multiplication; results are same width as multiplicands.

```
extern vSInt32 vS64HalfMultiply(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64Sub

Signed 64-bit subtraction (modular arithmetic).

```
extern vSInt32 vS64Sub(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS64SubS

Signed 64-bit subtraction with saturation (clipping).

```
extern vSInt32 vS64SubS(vSInt32 vA, vSInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS8Divide

Signed 8-bit division.

```
extern vSInt8 vS8Divide(vSInt8 vN, vSInt8 vD, vSInt8 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vS8HalfMultiply

Signed 8-bit multiplication; results are same width as multiplicands.

```
extern vSInt8 vS8HalfMultiply(vSInt8 vA, vSInt8 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vSasum

Finds the sum of the absolute values of the elements in a vector.

```
extern float vSasum(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The sum of the absolute values of the elements in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSaxpy

Multiplies a vector by a scalar , adds it to a second vector , and stores the result in the second vector.

```
extern void vSaxpy(
    SInt32 n,
    float alpha,
    const vector float x[],
    vector float y[]);
```

**Parameters**

*n*

Number of elements in each of the vectors *x* and *y*; must be a multiple of 4.

*alpha*

A multiplier for the vector *x*.

*x*

A vector array of `float` values.

*y*

A second vector array of `float` values.

**Discussion**
The elements of *x* are multiplied by *alpha* and added to the corresponding elements of *y*. The results are stored in *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vscalbf

For each vector element, calculates x * 2^n efficiently. This is not normally done by computing 2^n explicitly.

```
extern vFloat vscalbf(vFloat X, vSInt32 n);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vScopy

Copies one vector to another.

```
extern void vScopy(
    SInt32 n,
    const vector float x[],
    vector float y[]);
```

**Parameters**

*n*

Number of elements in vectors *x* and *y*; must be a multiple of 4.

*x*

A vector array of `float` values.

*y*

A second vector array of `float` values.

**Discussion**
The elements of *x* are copied to the corresponding elements of *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSdot

Computes the dot product of two vectors.

```
extern float vSdot(
    SInt32 n,
    const vector float x[],
    const vector float y[]);
```

**Parameters**

*n*

Number of elements in vectors *x* and *y*; must be a multiple of 4.

*x*

A vector array of `float` values.

*y*

A second vector array of `float` values.

**Return Value**
The dot product of the two vectors.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSgeadd()

Matrix addition for general matrices or their transposes.

```
extern void vSgeadd(
    SInt32 height,
    SInt32 width,
    const vector float a[],
    char forma,
    const vector float b[],
    char formb,
    vector float c[]);
```

**Parameters**

*height*

number of rows in the matrices to be added; must be a multiple of 4.

*width*

number of columns in the matrices to be added; must be a multiple of 4.

*a*

a matrix with elements of type `float`. If *forma* = 'n', the matrix itself is used in the calculation and it has *height* rows and *width* columns. If *forma* = 'T', the transpose is used and *a* has *width* rows and *height* columns.

*forma*

selector with a value of 'n' or 'T'.

*b*

a matrix with elements of type `float`. If *formb* = 'n', the matrix itself is used in the calculation and it has *height* rows and *width* columns. If *formb* = 'T', the transpose is used and *b* has *width* rows and *height* columns.

*formb*

selector with a value of 'n' or 'T'.

*c*

destination matrix with *height* rows and *width* columns.

**Discussion**

Matrix *a* (or its transpose) is added to matrix *b* (or its transpose); the result is stored in mactrix *c*.

**Availability**

Mac OS X version 10.0 and later.

**Declared In**

## vSgemm()

Combined matrix multiplication and addition for general matrices or their transposes.

```
extern void vSgemm(
    Sint32     l,
    SInt32 m,
    SInt32 n,
    const vector float a[],
    char forma,
    const vector float b[],
    char formb,
    vector float c,
    float alpha,
    float     beta
    vector float matrix[]);
```

**Parameters**

*l*

number of rows in matrix *c*; must be a multiple of 4.

*m*

if *forma* = 'n', *m* is the number of columns in matrix *a* ; if forma = 'T', *m* is the number of rows in matrix *a*. Also, if *formb* = 'n', *m* is the number of rows in matrix *b*; if formb = 'T', *m* is the number of columns in matrix *b*. *m* must be a multiple of 4.

*n*

number of columns in matrix *c*; must be a multiple of 4.

*a*

a matrix with elements of type `float`. If *forma* = 'n', the matrix itself is used in the calculation and it has *l* rows and *m* columns. If *forma* = 'T', the transpose is used and *a* has *m* rows and *l* columns. Thus the matrix used in the calculation is *l* by *n*.

*forma*

selector with a value of 'n' or 'T'.

*b*

a matrix with elements of type `float`. If *formb* = 'n', the matrix itself is used in the calculation and it has *m* rows and *n* columns. If *formb* = 'T', the transpose is used and *b* has *n* rows and *m* columns. Thus the matrix used in the calculation is *m* by *n*.

*formb*

selector with a value of 'n' or 'T'.

*c*

an *l* by *n* matrix with elements of type `float`.

*alpha*

multiplier for matrix *a*.

*beta*

multiplier for matrix *c*.

*matrix*

destination matrix with *l* rows and *n* columns.

**Discussion**
Matrix *a* (or its transpose) is multiplied by matrix *b* (or its transpose); matrix *c* is multiplied by *beta*, and the result is added to the result of the matrix multiplication; the result is stored in matrix *matrix*

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vSgemtx()

Forms the transpose of a matrix, multiplies it by a scalar and then by a vector, and adds the resulting vector to a second vector.

```
extern void vSgemtx(
    SInt32 m,
    SInt32 n,
    float alpha,
    const vector float a[],
    const vector float x[],
    vector float y[]);
```

**Parameters**

*m*

number of rows in *a*, and the length of vector *y*; must be a multiple of 4.

*n*

number of columns in *a*, and the length of vector *x*; must be a multiple of 4.

*alpha*

scalar multiplier for matrix `a`.

*a*

*m* by *n* matrix with elements of type `float`.

*x*

vector with elements of type `float`.

*y*

destination vector with *n* elements of type `float`.

**Discussion**
The transpose of matrix *a* is multiplied by *alpha* and then by vector *x*; the resulting vector is added to vector *y*, and the results are stored in *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vSgemul()

Matrix multiplication for general matrices or their transposes.

```
extern void vSgemul(
    Sint32 l,
    SInt32 m,
    SInt32 n,
    const vector float a[],
    char forma,
    const vector float b[],
    char formb,
    vector float matrix[]);
```

**Parameters**

*l*

number of rows in matrix *matrix*; must be a multiple of 4.

*m*

if *forma* = 'n', *m* is the number of columns in matrix *a* ; if forma = 'T', *m* is the number of rows in matrix *a*. Also, if *formb* = 'n', *m* is the number of rows in matrix *b*; if formb = 'T', *m* is the number of columns in matrix *b*. *m* must be a multiple of 4.

*n*

number of columns in the matrix *matrix*; must be a multiple of 4.

*a*

a matrix with elements of type `float`. If *forma* = 'n', the matrix itself is used in the calculation and it has *l* rows and *m* columns. If *forma* = 'T', the transpose is used and *a* has *m* rows and *l* columns. Thus the matrix used in the calculation is *l* by *m*.

*forma*

selector with a value of 'n' or 'T'.

*b*

a matrix with elements of type `float`. If *formb* = 'n', the matrix itself is used in the calculation and it has *m* rows and *n* columns. If *formb* = 'T', the transpose is used and *b* has *n* rows and *m* columns. Thus the matrix used in the calculation is *m* by *n*.

*formb*

selector with a value of 'n' or 'T'.

*matrix*

destination matrix with *l* rows and *n* columns.

**Discussion**

Matrix *a* (or its transpose) is multiplied by matrix *b* (or its transpose); the result is stored in matrix *matrix*.

**Availability**

Mac OS X version 10.0 and later.

**Declared In**

## vSgemv()

Multiplies a vector by a scalar. Multiplies a matrix by another scalar, then by a second vector, and adds the resulting vector to the first vector. This function can also perform the calculation with the transpose of the original matrix instead of the matrix itself. A selector parameter determines whether the transpose is used.

```
extern void vSgemv(
    char     forma,
    SInt32     m,
    SInt32 n,
    float     alpha,
    const vector float a[],
    const vector float x[],
    float beta,
    vector float y[]);
```

**Parameters**

*forma*

selects the variant computation to be performed: 'T' causes the transform of matrix *a* to be used, 'n' causes *a* itself to be used.

*m*

number of rows in *a*. If *forma* = 'n', *m* is the length of vector *y*; if *forma* = 'T', *m* is the length of vector *x*; must be a multiple of 4.

*n*

number of columns in *a*. If *forma* = 'n', *m* is the length of vector *x*; if *forma* = 'T', *m* is the length of vector *y*; must be a multiple of 4.

*alpha*

scalar multiplier for matrix `a`.

*a*

*m* by *n* matrix with elements of type `float`.

*x*

vector with elements of type `float`.

*beta*

scalar multiplier for vector `y`.

*y*

destination vector with *n* elements of type `float`.

**Discussion**

Vector *y* is multiplied by *beta*. Matrix *a* is multiplied by *alpha*. Then if *forma* = 'n', *a* is multiplied by vector *x*; if *forma* = 'T', the transpose of *a* is multiplied by *x*. The resulting vector is added to vector *y*, and the results are stored in *y*.

**Availability**

Mac OS X version 10.0 and later.

**Declared In**

## vSgemx()

Multiplies a matrix by a scalar and then by a vector, and adds the resulting vector to a second vector.

```
extern void vSgemx(
    SInt32     m,
    SInt32 n,
    float alpha,
    const vector float                a[],
    const vector float x[],
    vector float y[]);
```

**Parameters**

*m*

number of rows in *a*, and the length of vector *y*; must be a multiple of 4.

*n*

number of columns in *a*, and the length of vector *x*; must be a multiple of 4.

*alpha*

scalar multiplier for matrix `a`.

*a*

*m* by *n* matrix with elements of type `float`.

*x*

vector with elements of type `float`.

*y*

destination vector with *n* elements of type `float`.

**Discussion**

Matrix *a* is multiplied by *alpha* and then by vector *x*; the resulting vector is added to vector *y*, and the results are stored in *y*.

**Availability**

Mac OS X version 10.0 and later.

**Declared In**

## vSgesub()

Matrix subtraction for general matrices or their transposes.

```
extern void vSgesub(
    SInt32 height,
    SInt32 width,
    const vector float a[],
    char forma,
    const vector float b[],
    char formb,
    vector float c[]);
```

**Parameters**

*height*

number of rows in the matrices to be subtracted; must be a multiple of 4.

*width*

number of columns in the matrices to be subtracted; must be a multiple of 4.

*a*

a matrix with elements of type `float`. If *forma* = 'n', the matrix itself is used in the calculation and it has *height* rows and *width* columns. If *forma* = 'T', the transpose is used and *a* has *width* rows and *height* columns.

*forma*

selector with a value of 'n' or 'T'.

*b*

a matrix with elements of type `float`. If *formb* = 'n', the matrix itself is used in the calculation and it has *height* rows and *width* columns. If *formb* = 'T', the transpose is used and *b* has *width* rows and *height* columns.

*formb*

selector with a value of 'n' or 'T'.

*c*

destination matrix with *height* rows and *width* columns.

**Discussion**

Matrix *b* (or its transpose) is subtracted from matrix *a* (or its transpose); the result is stored in mactrix *c*.

**Availability**

Mac OS X version 10.0 and later.

**Declared In**

## vSgetmi()

General matrix transpose, in place.

```
extern void vSgetmi(
    SInt32 size,
    vector float         x[]);
```

**Parameters**

*size*

number of rows and columns in matrix *x*; must be a multiple of 4.

*x*

square matrix with *size* rows and *size* columns.

**Discussion**
The matrix *x* is transposed in place.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vSgetmo()

General matrix transpose, out of place.

```
extern void vSgetmo(
    SInt32 height,
    SInt32 width,
    const vector float x[],
    vector float y[]);
```

**Parameters**

*height*
: number of rows in matrix *x* and number of columns in matrix y; must be a multiple of 4.

*width*
: number of columns in matrix *x* and number of rows in matrix y; must be a multiple of 4.

*x*
: matrix with *height* rows and *width* columns.

*y*
: matrix with *width* rows and *height* columns.

**Discussion**
The matrix *x* is transposed into matrix *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vSgevv()

Multiplies two matrices and places the results in a third matrix.

```
extern void vSgevv(
    SInt32 l,
    SInt32 n,
    const vector float a[],
    const vector float                b[],
    vector float m[]);
```

**Parameters**

*l*
: number of rows in matrix *a* and in matrix *m*; must be a multiple of 4.

*n*
: number of columns in matrix *b* and in matrix *m*; must be a multiple of 4.

*a*
matrix with *l* rows.

*b*
matrix with *n* columns.

*m*
matrix with *l* rows and *n* columns.

**Discussion**
The matrices *a* and *b* are multiplied and the result is stored in matrix *m*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vsignbitf

For each vector element, returns a non-zero value if and only if the sign of `arg` is negative. This includes NaNs, infinities and zeros.

```
extern vUInt32 vsignbitf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vsinf

For each vector element, calculates the sine.

```
extern vFloat vsinf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vsinhf

For each vector element, calculates the hyperbolic sine of `X`.

```
extern vFloat vsinhf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vSnaxpy

Performs the computation of `vSaxpy` `n` times, using a different multiplier each time.

```
extern void vSnaxpy(
    SInt32 n,
    SInt32 m,
    const vector float a[],
    const vector float x[],
    vector float y[]);
```

**Parameters**

*n*

Number of elements in vector *a*; must be a multiple of 4.

*m*

Number of elements in each of the vectors *x* and *y*; must be a multiple of 4.

*x*

A vector array of `float` values.

*y*

A second vector array of `float` values.

**Discussion**
For i = 0 to n-1, the elements of *x* are multiplied by *a*[i] and added to the corresponding elements of *y*. The results are accumulated and stored in *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSndot

Computes the dot products of n pairs of vectors, accumulating or storing the results in an array of `n` `float` values.

```
extern void vSndot(
    SInt32 n,
    SInt32 m,
    float s[],
    SInt32 isw,
    const vector float x[],
    const vector float y[]);
```

**Parameters**

*n*

Number of dot products to compute, and number of elements in vector *s* ; must be a multiple of 4.

*m*

Number of elements in the vectors whose dot products are computed; must be a multiple of 4.

*s*

Destination vector; the *n* dot products are accumulated or stored here.

*isw*

A key that selects one of the four variants of this function: see Discussion below.

*x*

A matrix whose rows are *n* floating-point vectors, each containing *m* values.

*y*

A second matrix whose rows are *n* floating-point vectors, each containing *m* values.

**Discussion**
For i = 0 to n-1, the dot product of vectors *x*[i] and *y*[i] is computed. The dot product is accumulated or stored in *s*[i], according to the value of *isw*:

- if *isw* = 1, the dot product is stored in *s*[i].
- if *isw* = 2, the dot product is negated and then stored in *s*[i].
- if *isw* = 3, the dot product is added to the value in *s*[i].
- if *isw* = 4, the dot product is negated and then added to the value in *s*[i].

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSnorm2

Finds the Euclidean length of a vector.

```
extern float vSnorm2(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The Euclidean length of *x*.

**Discussion**
Input is not scaled.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSnrm2

Finds the Euclidean length of a vector.

```
extern float vSnrm2(SInt32 count, const vector float x[]);
```

**Parameters**

*count*

Number of elements in the vector *x*; must be a multiple of 4.

*x*

A vector array of `float` values.

**Return Value**
The Euclidean length of *x*.

**Discussion**
Input is scaled to avoid destructive underflow and overflow.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vsqrtf

For each vector element, calculates the square root of `X`.

```
extern vFloat vsqrtf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vSrot

Applies planar rotation to a set of n points whose x and y coordinates are contained in two arrays of vectors.

```
extern void vSrot(
    SInt32 n,
    vector float x[],
    vector float y[],
    float c,
    float s);
```

**Parameters**

*n*

number of points to be rotated; must be a multiple of 4.

*x*

vector with *n*/4 elements of type `vector float`, representing the x-coordinates of the points.

*y*

vector with *n*/4 elements of type `vector float`, representing the y-coordinates of the points.

*c*

cosine of the angle of rotation.

*s*
sine of the angle of rotation.

**Discussion**
The coordinates are modified in place in the vectors in arrays *x* and *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSscal

Scales a vector in place.

```
extern void vSscal(SInt32 n, float alpha, vector float x[]);
```

**Parameters**

*n*
number of elements in vector *x*; must be a multiple of 4.

*alpha*
scaling factor.

*v*
vector with *n* elements of type `float`.

**Discussion**
Each element of vector *x* is multiplied in place by *alpha*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSsum

Finds the sum of the values of the elements in a vector.

```
extern float vSsum(SInt32 count, const vector float x[]);
```

**Parameters**

*count*
Number of elements in the vector *x*; must be a multiple of 4.

*x*
A vector array of `float` values.

**Return Value**
The sum of the values of the elements in the vector.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSswap

Interchanges the elements of two vectors.

```
extern void vSswap(SInt32 n,vector float x[],vector float y[]);
```

**Parameters**

*n*

number of elements in vectors *x* and *y*; must be a multiple of 4.

*x*

vector with *n* elements of type `float`.

*y*

vector with *n* elements of type `float`.

**Discussion**
Each element of vector *x* is replaced by the corresponding element of *y*, and vice versa.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vectorOps.h`

## vSyax()

Multiplies each element of a vector and stores the results in a second vector.

```
extern void vSyax(
    SInt32 n,
    float alpha,
    const vector float x[],
    vector float y[]);
```

**Parameters**

*n*

number of elements in vectors *x* and *y*; must be a multiple of 4.

*alpha*

multiplier.

*x*

source vector with *n* elements of type `float`.

*y*

destination vector with *n* elements of type `float`.

**Discussion**
Each element of vector *x* is multiplied by *alpha*, and stored in the corresponding element of *y*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vSzaxpy()

Multiplies a vector by a scalar, adds it to a second vector, and stores the result in a third vector.

```
extern void vSzaxpy(
    SInt32 n,
    float alpha,
    const vector float x[],
    const vector float yY[],
    vector float z[]);
```

**Parameters**

*n*

number of elements in vectors *x*, *y*, and *z*; must be a multiple of 4.

*alpha*

multiplier.

*x*

source vector with *n* elements of type `float`.

*yY*

source vector with *n* elements of type `float`.

*z*

destination vector with *n* elements of type `float`.

**Discussion**
Each element of vector *x* is multiplied by *alpha*, then the corresponding element of *yY* is added. Results are stored in the corresponding elements of *z*.

**Availability**
Mac OS X version 10.0 and later.

**Declared In**

## vtablelookup

For each vector element of `Index_Vect`, returns the corresponding value from `Table`.

```
extern vUInt32 vtablelookup(vSInt32 Index_Vect, UInt32 *Table);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vtanf

For each vector element, calculates the tangent.

```
extern vFloat vtanf(vFloat arg);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vtanhf

For each vector element, calculates the hyperbolic tangent of `X`.

```
extern vFloat vtanhf(vFloat X);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vfp.h`

## vU1024Add

Unsigned 1024-bit addition (modular arithmetic).

```
extern void vU1024Add(
    const vU1024 * a,
    const vU1024 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024AddS

Unsigned 1024-bit addition with saturation (clipping).

```
extern void vU1024AddS(
    const vU1024 * a,
    const vU1024 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024Divide

Unsigned 1024-bit division.

```
extern void vU1024Divide(
    const vU1024 * numerator,
    const vU1024 * divisor,
    vU1024 * result,
    vU1024 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024HalfMultiply

Unsigned 1024-bit multiplication; result is the same width as multiplicands.

```
extern void vU1024HalfMultiply(
    const vU1024 * a,
    const vU1024 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024Mod

Unsigned 1024-bit mod.

```
extern void vU1024Mod(
    const vU1024 * numerator,
    const vU1024 * divisor,
    vU1024 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024Neg

Unsigned 1024-bit negation.

```
extern void vU1024Neg(
    const vU1024 * a,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024Sub

Unsigned 1024-bit subtraction (modular arithmetic).

```
extern void vU1024Sub(
    const vU1024 * a,
    const vU1024 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024SubS

Unsigned 1024-bit subtraction with saturation (clipping).

```
extern void vU1024SubS(
    const vU1024 * a,
    const vU1024 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU128Add

Unsigned 128-bit addition (modular arithmetic).

```
extern vUInt32 vU128Add(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU128AddS

Unsigned 128-bit addition with saturation (clipping).

```
extern vUInt32 vU128AddS(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU128Divide

Unsigned 128-bit division.

```
extern vUInt32 vU128Divide(vUInt32 vN, vUInt32 vD, vUInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU128FullMultiply

Unsigned 128-bit multiplication; result is twice as wide as multiplicands.

```
extern void vU128FullMultiply(
    const vU128 * a,
    const vU128 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU128HalfMultiply

Unsigned 128-bit multiplication; results are same width as multiplicands.

```
extern vUInt32 vU128HalfMultiply(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

### vU128Sub

Unsigned 128-bit subtraction (modular arithmetic).

```
extern vUInt32 vU128Sub(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vU128SubS

Unsigned 128-bit subtraction with saturation (clipping).

```
extern vUInt32 vU128SubS(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vU16Divide

Unsigned 16-bit division.

```
extern vUInt16 vU16Divide(vUInt16 vN, vUInt16 vD, vUInt16 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vU16HalfMultiply

Unsigned 16-bit multiplication; results are same width as multiplicands.

```
extern vUInt16 vU16HalfMultiply(vUInt16 vA, vUInt16 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

### vU256Add

Unsigned 256-bit addition (modular arithmetic).

```
extern void vU256Add(
    const vU256 * a,
    const vU256 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU256AddS

Unsigned 256-bit addition with saturation (clipping).

```
extern void vU256AddS(
    const vU256 * a,
    const vU256 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU256Divide

Unsigned 256-bit division.

```
extern void vU256Divide(
    const vU256 * numerator,
    const vU256 * divisor,
    vU256 * result,
    vU256 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU256FullMultiply

Unsigned 256-bit multiplication; result is twice as wide as multiplicands.

```
extern void vU256FullMultiply(
    const vU256 * a,
    const vU256 * b,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU256HalfMultiply

Unsigned 256-bit multiplication; result is the same width as multiplicands.

```
extern void vU256HalfMultiply(
    const vU256 * a,
    const vU256 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU256Mod

Unsigned 256-bit mod.

```
extern void vU256Mod(
    const vU256 * numerator,
    const vU256 * divisor,
    U256 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU256Neg

Unsigned 256-bit negation.

```
extern void vU256Neg(
    const vU256 * a,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU256Sub

Unsigned 256-bit subtraction (modular arithmetic).

```
extern void vU256Sub(
    const vU256 * a,
    const vU256 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU256SubS

Unsigned 256-bit subtraction with saturation (clipping).

```
extern void vU256SubS(
    const vU256 * a,
    const vU256 * b,
    vU256 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU32Divide

Unsigned 32-bit division.

```
extern vUInt32 vU32Divide(vUInt32 vN, vUInt32 vD, vUInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU32FullMulEven

Unsigned 32-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vUInt32 vU32FullMulEven(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU32FullMulOdd

Unsigned 32-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vUInt32 vU32FullMulOdd(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU32HalfMultiply

Unsigned 32-bit multiplication; results are same width as multiplicands.

```
extern vUInt32 vU32HalfMultiply(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU512Add

Unsigned 512-bit addition (modular arithmetic).

```
extern void vU512Add(
    const vU512 * a,
    const vU512 * b,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512AddS

Unsigned 512-bit addition with saturation (clipping).

```
extern void vU512AddS(
    const vU512 * a,
    const vU512 * b,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512Divide

Unsigned 512-bit division.

```
extern void vU512Divide(
    const vU512 * numerator,
    const vU512 * divisor,
    vU512 * result,
    vU512 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512FullMultiply

Unsigned 512-bit multiplication; result is twice as wide as multiplicands.

```
extern void vU512FullMultiply(
    const vU512 * a,
    const vU512 * b,
    vU1024 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512HalfMultiply

Unsigned 512-bit multiplication; result is the same width as multiplicands.

```
extern void vU512HalfMultiply(
    const vU512 * a,
    const vU512 * b,
    U512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512Mod

Unsigned 512-bit mod.

```
extern void vU512Mod(
    const vU512 * numerator,
    const vU512 * divisor,
    vU512 * remainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512Neg

Unsigned 512-bit negation.

```
extern void vU512Neg(
    const vU512 * a,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512Sub

Unsigned 512-bit subtraction (modular arithmetic).

```
extern void vU512Sub(
    const vU512 * a,
    const vU512 * b,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBigNum.h`

## vU512SubS

Unsigned 512-bit subtraction with saturation (clipping).

```
extern void vU512SubS(
    const vU512 * a,
    const vU512 * b,
    vU512 * result);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBigNum.h

## vU64Add

Unsigned 64-bit addition (modular arithmetic).

```
extern vUInt32 vU64Add(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU64AddS

Unsigned 64-bit addition with saturation (clipping).

```
extern vUInt32 vU64AddS(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU64Divide

Unsigned 64-bit division.

```
extern vUInt32 vU64Divide(vUInt32 vN, vUInt32 vD, vUInt32 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU64FullMulEven

Unsigned 64-bit multiplication; results are twice as wide as multiplicands, even-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vUInt32 vU64FullMulEven(vUInt32vA, vUInt32vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
vBasicOps.h

## vU64FullMulOdd

Unsigned 64-bit multiplication; results are twice as wide as multiplicands, odd-numbered elements of multiplicand vectors are used. Note the big-endian convention: the leftmost element is element 0.

```
extern vUInt32 vU64FullMulOdd(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU64HalfMultiply

Unsigned 64-bit multiplication; results are same width as multiplicands.

```
extern vUInt32 vU64HalfMultiply(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU64Sub

Unsigned 64-bit subtraction (modular arithmetic).

```
extern vUInt32 vU64Sub(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU64SubS

Unsigned 64-bit subtraction with saturation (clipping).

```
extern vUInt32 vU64SubS(vUInt32 vA, vUInt32 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU8Divide

Unsigned 8-bit division.

```
extern vUInt8 vU8Divide(vUInt8 vN, vUInt8 vD, vUInt8 * vRemainder);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vU8HalfMultiply

Unsigned 8-bit multiplication; results are same width as multiplicands.

```
extern vUInt8 vU8HalfMultiply(vUInt8 vA, vUInt8 vB);
```

**Availability**
Mac OS X version 10.0 and later.

**Declared In**
`vBasicOps.h`

## vvacos

For each double-precision array element, sets `y` to the arccosine of `x`.

```
void vvacos (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvacosf

For each single-precision array element, sets `y` to the arccosine of `x`.

```
void vvacosf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvacosh

For each double-precision array element, sets `y` to the inverse hyperbolic cosine of `x`.

```
void vvacosh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvacoshf

For each single-precision array element, sets `y` to the inverse hyperbolic cosine of `x`.

```
void vvacoshf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvasin

For each double-precision array element, sets `y` to the arcsine of `x`.

```
void vvasin (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvasinf

For each single-precision array element, sets `y` to the arcsine of `x`.

```
void vvasinf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvasinh

For each double-precision array element, sets `y` to the inverse hyperbolic sine of `x`.

```
void vvasinh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvasinhf

For each single-precision array element, sets `y` to the inverse hyperbolic sine of `x`.

```
void vvasinhf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvatan

For each double-precision array element, sets `y` to the arctangent of `x`.

```
void vvatan (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvatan2

For each double-precision array element, sets `z` to the arctangent of `y/x`.

```
void vvatan2 (
    double * /* z */,
    const double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvatan2f

For each single-precision array element, sets `z` to the arctangent of `y/x`.

```
void vvatan2f (
    float * /* z */,
    const float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvatanf

For each single-precision array element, sets `y` to the arctangent of `x`.

```
void vvatanf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvatanh

For each double-precision array element, sets `y` to the inverse hyperbolic tangent of `x`.

```
void vvatanh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvatanhf

For each single-precision array element, sets `y` to the inverse hyperbolic tangent of `x`.

```
void vvatanhf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvceil

For each double-precision array element, sets `y` to the ceiling of `x`.

```
void vvceil (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvceilf

For each single-precision array element, sets `y` to the ceiling of `x`.

```
void vvceilf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvcos

For each double-precision array element, sets `y` to the cosine of `x`.

```
void vvcos (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvcosf

For each single-precision array element, sets `y` to the cosine of `x`.

```
void vvcosf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvcosh

For each double-precision array element, sets `y` to the hyperbolic cosine of `x`.

```
void vvcosh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvcoshf

For each single-precision array element, sets `y` to the hyperbolic cosine of `x`.

```
void vvcoshf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvcosisin

For each double-precision array element, sets the real part of `C` to the sine of `x` and the imaginary part of `C` to the cosine of `x`.

```
void vvcosisin (
    __double_complex_t * /* C */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvcosisinf

For each single-precision array element, sets the real part of `C` to the sine of `x` and the imaginary part of `C` to the cosine of `x`.

```
void vvcosisinf (
    __float_complex_t * /* C */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvdiv

For each double-precision array element, sets `z` to `y/x`.

```
void vvdiv (
    double * /* z */,
    const double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvdivf

For each single-precision array element, sets `z` to `y/x`.

```
void vvdivf (
    float * /* z */,
    const float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvexp

For each double-precision array element, sets `y` to the exponential of `x`.

```
void vvexp (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvexpf

For each single-precision array element, sets `y` to the exponential of `x`.

```
void vvexpf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvfloor

For each double-precision array element, sets `y` to the floor of `x`.

```
void vvfloor (double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvfloorf

For each single-precision array element, sets `y` to the floor of `x`.

```
void vvfloorf (float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvint

For each double-precision array element, sets `y` to the integer truncation of `x`.

```
void vvint (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvintf

For each single-precision array element, sets `y` to the integer truncation of `x`.

```
void vvintf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvlog

For each double-precision array element, sets `y` to the natural logarithm of `x`.

```
void vvlog (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvlog10

For each double-precision array element, sets `y` to the base 10 logarithm of `x`.

```
void vvlog10 (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvlog10f

For each single-precision array element, sets `y` to the base 10 logarithm of `x`.

```
void vvlog10f (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvlogf

For each single-precision array element, sets `y` to the natural logarithm of `x`.

```
void vvlogf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvnint

For each double-precision array element, sets `y` to the nearest integer to `x`.

```
void vvnint (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvnintf

For each single-precision array element, sets `y` to the nearest integer to `x`.

```
void vvnintf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvpow

For each double-precision array element, sets `z` to `x` raised to the power of `y`.

```
void vvpow (
    double * /* z */,
    const double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvpowf

For each single-precision array element, sets `z` to `x` raised to the power of `y`.

```
void vvpowf (
    float * /* z */,
    const float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

### vvrec

For each double-precision array element, sets `y` to the reciprocal of `y`.

```
void vvrec (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvrecf

For each single-precision array element, sets `y` to the reciprocal of `y`.

```
void vvrecf(
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvrsqrt

For each double-precision array element, sets `y` to the reciprocal of the square root of `x`.

```
void vvrsqrt (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvrsqrtf

For each single-precision array element, sets `y` to the reciprocal of the square root of `x`.

```
void vvrsqrtf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## vvsin

For each double-precision array element, sets `y` to the sine of `x`.

```
void vvsin (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsincos

For each double-precision array element, sets `z` to the sine of `x` and `y` to the cosine of `x`.

```
void vvsincos (
    double * /* z */,
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsincosf

For each single-precision array element, sets `z` to the sine of `x` and `y` to the cosine of `x`.

```
void vvsincosf (
    float * /* z */,
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsinf

For each single-precision array element, sets `y` to the sine of `x`.

```
void vvsinf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsinh

For each double-precision array element, sets `y` to the hyperbolic sine of `x`.

```
void vvsinh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsinhf

For each single-precision array element, sets `y` to the hyperbolic sine of `x`.

```
void vvsinhf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsqrt

For each double-precision array element, sets `y` to the square root of `x`.

```
void vvsqrt (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvsqrtf

For each single-precision array element, sets `y` to the square root of `x`.

```
void vvsqrtf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvtan

For each double-precision array element, sets `y` to the tangent of `x`.

```
void vvtan (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvtanf

For each single-precision array element, sets `y` to the tangent of `x`.

```
void vvtanf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
vForce.h

## vvtanh

For each double-precision array element, sets `y` to the hyperbolic tangent of `x`.

```
void vvtanh (
    double * /* y */,
    const double * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

### vvtanhf

For each single-precision array element, sets `y` to the hyperbolic tangent of `x`.

```
void vvtanhf (
    float * /* y */,
    const float * /* x */,
    const int * /* n */);
```

**Availability**
Available in Mac OS X v10.4 and later.

**Declared In**
`vForce.h`

## Data Types

**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.

### vUInt8

A 128-bit vector packed with `unsigned char` values.

```
typedef vector unsigned char vUInt8;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

### vSInt8

A 128-bit vector packed with `signed char` values.

```
typedef vector signed char vSInt8;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

### vUInt16

A 128-bit vector packed with `unsigned short` values.

```
typedef vector unsigned short vUInt16;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

### vSInt16

A 128-bit vector packed with `signed short` values.

```
typedef vector signed short vSInt16;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

### vUInt32

A 128-bit vector packed with `unsigned int` values.

```
typedef vector unsigned int vUInt32;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

### vSInt32

A 128-bit vector packed with `signed int` values.

```
typedef vector signed int vSInt32;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

## vFloat

A 128-bit vector packed with `float` values.

```
typedef vector float vFloat;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

## vBool32

A 128-bit vector packed with `bool int` values.

```
typedef vector bool int vBool32;
```

**Availability**
Available in Mac OS X v10.3 and later.

**Declared In**
`vecLibTypes.h`

## __float_complex_t

A single-precision complex number type.

```
typedef complex float __float_complex_t;
```

**Declared In**

## __double_complex_t

A double-precision complex number type.

```
typedef complex double __double_complex_t;
```

**Declared In**

## vU128

A union containing one `vUInt32` vector or four 32-bit integers, representing a 128-bit unsigned integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vU128 {
vUInt32    v;
struct {
  vUInt32  v1;
  }        vs;
struct {
  UInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   LSW;
  }        s;
};
typedef union vU128 vU128;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vS128

A union containing one `vSInt32` vector or four 32-bit integers, representing a 128-bit signed integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vS128 {
vUInt32    v;
struct {
  vUInt32  v1;
  }        vs;
struct {
  SInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   LSW;
  }        s;
};
typedef union vS128 vS128;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vU256

A union containing an array or structure of two `vUInt32` vectors or eight 32-bit integers, representing a 256-bit unsigned integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vU256 {
vUInt32    v[2];
struct {
  vUInt32  v1;
  vUInt32  v2;
  }        vs;
struct {
  UInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   LSW;
  }        s;
};
typedef union vU256 vU256;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vS256

A union containing an array or structure of two `vUInt32` vectors or eight 32-bit integers, representing a 256-bit signed integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vS256 {
vUInt32    v[2];
struct {
  vUInt32  v1;
  vUInt32  v2;
  }        vs;
struct {
  SInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   LSW;
  }        s;
};
typedef union vS256 vS256;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vU512

A union containing an array or structure of four `vUInt32` vectors or sixteen 32-bit integers, representing a 256-bit unsigned integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vU512 {
vUInt32    v[4];
struct {
  vUInt32  v1;
  vUInt32  v2;
  vUInt32  v3;
  vUInt32  v4;
  }        vs;
struct {
  UInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   d8;
  UInt32   d9;
  UInt32   d10;
  UInt32   d11;
  UInt32   d12;
  UInt32   d13;
  UInt32   d14;
  UInt32   d15;
  UInt32   LSW;
  }        s;
};
typedef union vU512 vU512;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vS512

A union containing an array or structure of four `vUInt32` vectors or sixteen 32-bit integers, representing a 256-bit signed integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vS512 {
vUInt32    v[4];
struct {
  vUInt32  v1;
  vUInt32  v2;
  vUInt32  v3;
  vUInt32  v4;
  }        vs;
struct {
  SInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   d8;
  UInt32   d9;
  UInt32   d10;
  UInt32   d11;
  UInt32   d12;
  UInt32   d13;
  UInt32   d14;
  UInt32   d15;
  UInt32   LSW;
  }        s;
};
typedef union vS512 vS512;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

## vU1024

A union containing an array or structure of eight `vUInt32` vectors or thirty-two 32-bit integers, representing a 1024-bit unsigned integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vU1024 {
vUInt32    v[8];
struct {
  vUInt32  v1;
  vUInt32  v2;
  vUInt32  v3;
  vUInt32  v4;
  vUInt32  v5;
  vUInt32  v6;
  vUInt32  v7;
  vUInt32  v8;
  }        vs;
struct {
  UInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   d8;
  UInt32   d9;
  UInt32   d10;
  UInt32   d11;
  UInt32   d12;
  UInt32   d13;
  UInt32   d14;
  UInt32   d15;
  UInt32   d16;
  UInt32   d17;
  UInt32   d18;
  UInt32   d19;
  UInt32   d20;
  UInt32   d21;
  UInt32   d22;
  UInt32   d23;
  UInt32   d24;
  UInt32   d25;
  UInt32   d26;
  UInt32   d27;
  UInt32   d28;
  UInt32   d29;
  UInt32   d30;
  UInt32   d31;
  UInt32   LSW;
  }        s;
};
typedef union vU1024 vU1024;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

### vS1024

A union containing an array or structure of eight `vUInt32` vectors or thirty-two 32-bit integers, representing a 1024-bit signed integer. Conditional definitions provide compatibility with both PowerPC and Intel architectures; see the header file for details.

```
union vS1024 {
vUInt32    v[8];
struct {
  vUInt32  v1;
  vUInt32  v2;
  vUInt32  v3;
  vUInt32  v4;
  vUInt32  v5;
  vUInt32  v6;
  vUInt32  v7;
  vUInt32  v8;
  }        vs;
struct {
  SInt32   MSW;
  UInt32   d2;
  UInt32   d3;
  UInt32   d4;
  UInt32   d5;
  UInt32   d6;
  UInt32   d7;
  UInt32   d8;
  UInt32   d9;
  UInt32   d10;
  UInt32   d11;
  UInt32   d12;
  UInt32   d13;
  UInt32   d14;
  UInt32   d15;
  UInt32   d16;
  UInt32   d17;
  UInt32   d18;
  UInt32   d19;
  UInt32   d20;
  UInt32   d21;
  UInt32   d22;
  UInt32   d23;
  UInt32   d24;
  UInt32   d25;
  UInt32   d26;
  UInt32   d27;
  UInt32   d28;
  UInt32   d29;
  UInt32   d30;
  UInt32   d31;
  UInt32   LSW;
  }        s;
};
typedef union vS1024 vS1024;
```

**Availability**
Available in Mac OS X v10.0 and later.

**Declared In**
`vBigNum.h`

# Document Revision History

This table describes the changes to *vecLib Framework Reference*.

| Date | Notes |
| --- | --- |
| 2009-01-06 | Corrected the Availability of the vForce.h functions. |
| 2005-09-08 | New document that describes the C API for vector mathematical functions in the vecLib subframework of the Accelerate framework. |

Document Revision History

# Index

## Symbols



## V