Swift Standard Library - SIMD Vector Types

SIMD Vector Types

- Documentation
  - **Apple Documentation**: https://developer.apple.com/documentation/swift/swift_standard_library/numbers_and_basic_values/simd_vector_types
  - **apple / swift-evolution - SIMD Vector SE-0229**: https://github.com/apple/swift-evolution/blob/master/proposals/0229-simd.md
- SIMD Vector Types is a feature added in **Swift 5**, **Swift Standard Library**.
  - @frozen struct **SIMD2**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD3**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD4**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD8**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD16**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD32**<Scalar> where Scalar : SIMDScalar
  - @frozen struct **SIMD64**<Scalar> where Scalar : SIMDScalar
- Protocol **SIMDScalar** ··· A type that can be used as an element in a SIMD vector.
  - Conforming Types:
    - **Double**, **Float**, **Float16**
    - **Int**, **Int8**, **Int16**, **Int32**, **Int64**
    - **UInt**, **UInt8**, **UInt16**, **UInt32**, **UInt64**
- Protocol **SIMD** ··· A SIMD vector of a fixed number of elements.

	SIMD2	SIMD3	SIMD4	SIMD8	SIMD16	SIMD32	SIMD64
SIMD Protocol (see the right table)	X	X	X	X	X	X	X
Initializers • **init()** ··· zero in all lanes • **init<Other>(SIMDx<Other>)** • **init(Scalar, ···)** • **init<Other>(SIMDx<Other>, rounding: FloatingPointRoundingRule)** *S: FWI • **init(arrayLiteral: Scalar ···)** • **init<Other>(clamping: SIMDx<Other>)** *S: FWI • **init(from: Decoder)** • **init<Other>(truncatingIfNeeded: SIMDx<Other>)** *S: FWI *S: FWI ··· Scalar: FixedWidthInteger	X init(x:y:)	X init(x:y:z:)	X init(x:y:z:w:) init(lowHalf:highHalf:)	X init(lowHalf:highHalf:)	X init(lowHalf:highHalf:)	X init(lowHalf:highHalf:)	X init(lowHalf:highHalf:)
Instance Properties • **var debugDescription: String** • **var description: String** • **var hashValue: Int** • **var scalarCount: Int**	X var x var y	X var x var y var z	X evenHalf highHalf lowHalf oddHalf x, y, z, w	X evenHalf highHalf lowHalf oddHalf	X evenHalf highHalf lowHalf oddHalf	X evenHalf highHalf lowHalf oddHalf	X evenHalf highHalf lowHalf oddHalf
Type Properties • **static var scalarCount : Int**	X	X	X	X	X	X	X
Instance Methods • **func encode(to: Encoder)** • **func hash(into: inout Hasher)**	X	X	X	X	X	X	X
Operator Functions • **static func != -> Bool**	X	X	X	X	X	X	X

Supporting Functions		
Generic Functions	• func **all**<Storage>(SIMDMask<Storage>) -> Bool • func **any**<Storage>(SIMDMask<Storage>) -> Bool • func **pointwiseMax**<T>(T, T) -> T • func **pointwiseMin**<T>(T, T) -> T	• all: True if every lane of mask is true. • any: True if any lane of mask is true. • pointwise: lanewise max/min of two vectors.

SIMD Protocol		
Initializers	• **init**<S>(S) S: Sequence • **init**(repeating: Self.Scalar)	
Instance Properties	• var **indices**: Range<Int> • var **leadingZeroBitCount**: Self Scalar : FixedWidthInteger • var. **nonzeroBitCount**: Self Scalar : FixedWidthInteger • var. **trailingZeroBitCount**: Self Scalar : FixedWidthInteger	
Type Properties	• static var **one**: Self • static var **zero**: Self	
Instance Methods	• func **addProduct**(Self.Scalar, Self) Scalar : FloatingPoint • func **addProduct**(Self, Self.Scalar) Scalar : FloatingPoint • func **addProduct**(Self, Self) Scalar : FloatingPoint • func **addingProduct**(Self.Scalar, Self) -> Self Scalar : FloatingPoint • func **addingProduct**(Self, Self.Scalar) -> Self Scalar : FloatingPoint • func **addingProduct**(Self, Self) -> Self. Scalar : FloatingPoint • func **clamp**(lowerBound: Self, upperBound: Self) • func **clamped**(lowerBound: Self, upperBound: Self) -> Self • func **formSquareRoot**( ) Scalar : FloatingPoint • func **squareRoot**() -> Self Scalar : FloatingPoint • func **max**() -> Self.Scalar • func **min**() -> Self.Scalar • func **replace**(with: Self.Scalar, where: SIMDMask<Self.MaskStorage>) • func **replace**(with: Self, where: SIMDMask<Self.MaskStorage>) • func **replacing**(with: Self.Scalar, where: SIMDMask<Self.MaskStorage>) -> Self • func **replacing**(with: Self, where: SIMDMask<Self.MaskStorage>) -> Self • func **round**(FloatingPointRoundingRule) Scalar : FloatingPoint • func **rounded**(FloatingPointRoundingRule) -> Self Scalar: FloatingPoint • func **sum**() -> Self.Scalar Scalar : FloatingPoint • func **wrappedSum**() -> Self.Scalar Scalar : FixedWidthInteger	• addProduct: a = a + b x c • clamp: forces between lower and upper. • replace: replaces the element whose mask is true. • round: rounds the floating point value according to the rule. • wrappedSum: uses the wrapping addition.
Type Methods	• static func **random**(in: ClosedRange<Self.Scalar>) -> Self • static func **random**(in: Range<Self.Scalar>) -> Self • static func **random**<T>(in: ClosedRange<Self.Scalar>, using: inout T) -> Self • static func **random**<T>(in: Range<Self.Scalar>, using: inout T) -> Self	
Operator Functions	• static func **&*** (Self.Scalar, Self) -> Self Scalar : FixedWidthInteger • static func **&*** (Self, Self.Scalar) -> Self Scalar : FixedWidthInteger • static func **&*** (Self, Self) -> Self Scalar : FixedWidthInteger • static func **&*=** (inout Self, Self.Scalar) Scalar : FixedWidthInteger • static func **&*=** (inout Self, Self) Scalar : FixedWidthInteger • **&+, &+=** Scalar : FixedWidthInteger ··· pointwise Wrapping addition • **&-, &-=** Scalar : FixedWidthInteger ··· pointwise Wrapping subtraction • **&, &=** Scalar : FixedWidthInteger ··· pointwise bits AND • **|, |=** Scalar : FixedWidthInteger ··· pointwise bits OR • **^, ^=** Scalar : FixedWidthInteger ··· pointwise bits XOR • static func **~** (Self) -> Self Scalar : FixedWidthInteger ··· pointwise bits NOT • **&<<, &<<=** Scalar : FixedWidthInteger ··· pointwise wrapping bits shift • **&>>, &>>=** Scalar : FixedWidthInteger ··· pointwise wrapping bits shift • **%, %=** Scalar : FixedWidthInteger. ··· pointwise modulo • **/, /=** Scalar : FixedWidthInteger ··· pointwise division • ***, *=** Scalar : FloatingPoint ··· pointwise product (Hadamard product) • **+, +=** Scalar : FloatingPoint ··· pointwise addition • **-, -=** Scalar : FloatingPoint ··· pointwise subtraction • **/, /=** Scalar : FloatingPoint ··· pointwise division • static func **==** (Self, Self) -> Bool • **.==, .!=, .<, .<=, .>, .>=** -> SIMDMask<Self.MaskStorage> : pointwise comparison	• &*, &+, &-, &<<, &>>: wrapping operations for FixedWidthInteger • dot comparators (ex. .!=): pointwise comparison

Example

```
// SIMD - Initializers
SIMD3<Int>()                                                                    SIMD3<Int>(0, 0, 0)
SIMD3<Double>(SIMD3<Double>())                                                  SIMD3<Double>(0.0, 0.0, 0.0)
SIMD3<Double>(1, 2, 3)                                                          SIMD3<Double>(1.0, 2.0, 3.0)
SIMD3<Int8>(SIMD3<Double>(1.9, -1.9, -1.1), rounding: .towardZero)              SIMD3<Int8>(1, -1, -1)
SIMD3<Int8>(arrayLiteral: 1, 2, 3)                                              SIMD3<Int8>(1, 2, 3)
SIMD3<Int8>(clamping: SIMD3<Int>(200, -200, 100))                               SIMD3<Int8>(127, -128, 100)
SIMD3<UInt8>(clamping: SIMD3<Int>(200, -200, 100))                              SIMD3<UInt8>(200, 0, 100)
SIMD3<Int8>(truncatingIfNeeded: SIMD3<Int>(512, 511, 100))                      SIMD3<Int8>(0, -1, 100)
SIMD3<Int8>([1, 2, 3])                                                          SIMD3<Int8>(1, 2, 3)
SIMD3<Int8>(repeating: 1)                                                       SIMD3<Int8>(1, 1, 1)

// Type properties
SIMD3<Int8>.one                                                                 SIMD3<Int8>(1, 1, 1)
SIMD3<Double>.zero                                                              SIMD3<Double>(0.0, 0.0, 0.0)
SIMD8<Int>.scalarCount                                                          8

// Type methods
SIMD3<Int8>.random(in: 0...100)                                                 SIMD3<Int8>(8, 87, 25)
SIMD3<Double>.random(in: 0...1)                                                 SIMD3<Double>(0.5457115007719541...

// Instance properties
SIMD3<Int8>.zero.indices                                                        {lowerBound 0, upperBound 3}
SIMD3<Int8>.one.scalarCount                                                     3

// Instance methods
SIMD3<Double>.one.addingProduct(2.0, SIMD3<Double>(1.0, 2.0, 3.0))              SIMD3<Double>(3.0, 5.0, 7.0)
SIMD3<Double>.one.addingProduct(SIMD3<Double>(1, 2, 3), SIMD3<Double>(4, 5, 6)) SIMD3<Double>(5.0, 11.0, 19.0)
SIMD3<Double>(-0.5, 0.5, 1.5).clamped(lowerBound: SIMD3<Double>.zero, upperBound:
    SIMD3<Double>.one)                                                          SIMD3<Double>(0.0, 0.5, 1.0)
SIMD3<Int>(-2, 0, 2).clamped(lowerBound: SIMD3<Int>.zero, upperBound: SIMD3<Int>.one) SIMD3<Int>(0, 0, 1)
SIMD3<Double>(1, 2, 3).squareRoot()                                             SIMD3<Double>(1.0, 1.414213562373...
SIMD3<Int>(1, 2, 3).max()                                                       3
SIMD3<Double>(0.1, 0.2, 0.3).min()                                              0.1
SIMD3<Double>(1, 2, 3).replacing(with: 4, where:
    SIMDMask<SIMD3<Double>.MaskStorage>(arrayLiteral: true, true, false))       SIMD3<Double>(4.0, 4.0, 3.0)
SIMD3<Double>(1.9, -1.9, 1.1).rounded(.towardZero)                              SIMD3<Double>(1.0, -1.0, 1.0)
SIMD3<Double>(1, 2, 3).sum()                                                    6
SIMD3<Int>(1, 2, 3).wrappedSum()                                                6
SIMD3<Int8>(127, 1, 1).wrappedSum()                                             -127
```

```
// Operators
SIMD2<Int8>(2, 64) &* SIMD2<Int8>(3, 4)                                         SIMD2<Int8>(6, 0)
SIMD2<Int8>(2, 64) &+ SIMD2<Int8>(3, 64)                                        SIMD2<Int8>(5, -128)
SIMD2<Int8>(2, -64) &- SIMD2<Int8>(2, 65)                                       SIMD2<Int8>(0, 127)
SIMD2<UInt8>(255, 128) & SIMD2<UInt8>(1, 1)                                     SIMD2<UInt8>(1, 0)
SIMD2<UInt8>(128, 128) | SIMD2<UInt8>(64, 128)                                  SIMD2<UInt8>(192, 128)
SIMD2<UInt8>(255, 128) ^ SIMD2<UInt8>(128, 255)                                 SIMD2<UInt8>(127, 127)
~SIMD2<UInt8>(255, 128)                                                         SIMD2<UInt8>(0, 127)
SIMD2<UInt8>(64, 64) &<< SIMD2<UInt8>(1, 2)                                     SIMD2<UInt8>(128, 0)
SIMD2<UInt8>(128, 128) &>> SIMD2<UInt8>(1, 8)                                   SIMD2<UInt8>(64, 128)
SIMD2<Int8>(100, 99) % Int8(33)                                                 SIMD2<Int8>(1, 0)
SIMD2<Int8>(100, 99) / Int8(33)                                                 SIMD2<Int8>(3, 3)

SIMD2<Double>(1, 2) * Double(10)                                                SIMD2<Double>(10.0, 20.0)
SIMD2<Double>(1, 2) * SIMD2<Double>(10, -10)                                    SIMD2<Double>(10.0, -20.0)

SIMD2<Double>(1, 2) == SIMD2<Double>(1, 2)                                      true
SIMD2<Double>(1, 2) != SIMD2<Double>(1, 3)                                      true
SIMD2<Double>(1, 2) .== SIMD2<Double>(1, 3)                                     SIMDMask<SIMD2<Int64>>(true, false)
SIMD2<Int>(1, 2) .> SIMD2<Int>(1, 1)                                            SIMDMask<SIMD2<Int>>(false, true)
```

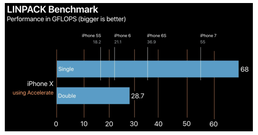
Note

Accelerate simd library　(1) Vectors

Accelerate simd library (iOS 11+)

- Apple References
  - API: Accelerate simd ⋯ Perform computations on small vectors and matrices.
  - Article: Working with Vectors ⋯ Use vectors to calculate geometric values, dot products and cross products.
  - Article: Working with Matrices ⋯ Solve simultaneous equations and transform points in space.
  - Article: Working with Quaternions ⋯ Rotate points around the surface of a sphere, and interpolate between them.
  - Sample Code: Rotating a Cube by Transforming Its Vertices ⋯ Use quaternion interpolation to rotate a cube.
  - WWDC18: Using Accelerate and simd (https://developer.apple.com/videos/play/wwdc2018/701/)

1. Vectors
   - Signed Integer Vectors / Unsigned Integer Vectors
   - Floating-Point Vectors
2. Matrices
   - up to 4x4 (columns x rows : column major naming convention)
3. Quaternions
   - Single-precision quaternion / Double-precision quaternion



Benchmark : Single vs Double using Accelerate (WWDC19)

Signed/Unsigned Packed Vector Data Types

Packed Vector Data Types		typealias	
8-Bit	simd_packed_charN	SIMDN<CChar>	N = 2, 4, 8, 16, 32, 64
16-Bit	simd_packed_shortN	SIMDN<Int16>	N = 2, 4, 8, 16, 32
32-Bit	simd_packed_intN	SIMDN<Int32>	N = 2, 4, 8, 16
64-Bit	simd_packed_longN	SIMDN<simd_long1>	N = 2, 4, 8
8-Bit	simd_packed_ucharN	SIMDN<UInt8>	N = 2, 4, 8, 16, 32, 64
16-Bit	simd_packed_ushortN	SIMDN<UInt16>	N = 2, 4, 8, 16, 32
32-Bit	simd_packed_uintN	SIMDN<UInt32>	N = 2, 4, 8, 16
64-Bit	simd_packed_ulongN	SIMDN<simd_ulong1>	N = 2, 4, 8
32-Bit	simd_packed_floatN	SIMDN<Float>	N = 2, 4, 8, 16
64-Bit	simd_packed_doubleN	SIMDN<Double>	N = 2, 4, 8

Integer Vectors

Functions		Signed Integer Vectors					Unsigned Integer Vectors				
			8-Bit	16-Bit	32-Bit	64-Bit		8-Bit	16-Bit	32-Bit	64-Bit
		simd_char1 simd_short1 simd_int1 simd_long1	simd_char2 simd_char3 simd_char4 simd_char8 simd_char16 simd_char32 simd_char64	simd_short2 simd_short3 simd_short4 simd_short8 simd_short16 simd_short32	simd_int2 simd_int3 simd_int4 simd_int8 simd_int16	simd_long2 simd_long3 simd_long4 simd_long8	simd_uchar1 simd_ushort1 simd_uint1 simd_ulong1	simd_uchar2 simd_uchar3 simd_uchar4 simd_uchar8 simd_uchar16 simd_uchar32 simd_uchar64	simd_ushort2 simd_ushort3 simd_ushort4 simd_ushort8 simd_ushort16 simd_ushort32	simd_uint2 simd_uint3 simd_uint4 simd_uint8 simd_uint16	simd_ulong2 simd_ulong3 simd_ulong4 simd_ulong8
			vector_Y	vector_Y	vector_Y	vector_Y		vector_Y	vector_Y	vector_Y	vector_Y
		CChar Int16 Int32 Int (Int64)	SIMDn<CChar>	SIMDn<Int16>	SIMDn<Int32>	SIMDn<simd_long1>	UInt8 UInt16 UInt32 UInt64	SIMDn<UInt8>	SIMDn<UInt16>	SIMDn<UInt32>	SIMDn<simd_ulong1>
Functions to Create From Other Vectors, Scalar Values, Combinations of Vector and Scalars	simd_make_Y vectorN simd_make_Y_undef		X *Y: char2 /3 /4 / 8 /16 /32 /64 *N: 2, 3, 4, 8, 16, 32	X *Y: short2 /3 / 4 /8 /16 /32 *N: 2, 3, 4, 8, 16, 32	X *Y: int2 /3 /4 /8 /16 *N: 2, 3, 4, 8, 16	X *Y: long2 /3 / 4 /8 *N: 2, 3, 4, 8		X *Y: uchar2 /3 / 4 /8 /16 /32 /64 *N: 2, 3, 4, 8, 16, 32	X *Y: ushort2 /3 / 4 /8 /16 /32 *N: 2, 3, 4, 8, 16, 32	X *Y: uint2 /3 /4 /8 /16 *N: 2, 3, 4, 8, 16	X *Y: ulong2 /3 /4 /8 *N: 2, 3, 4, 8
Functions to Create From Vectors of Other Types	simd_T		X * T: char	X * T: short	X * T: int	X * T: long		X * T: uchar	X * T: ushort	X * T: uint	X * T: ulong
Functions to Perform Saturation Conversion From Vectors of Other Types	simd_T_sat		X * T: char	X * T: short	X * T: int	X * T: long		X * T: uchar	X * T: ushort	X * T: uint	X * T: ulong
Functions to Perform Round-Half-to-Even Conversion From Single-Precision Vectors	simd_T_rte				X * T: int	X * T: long					
Common Functions	simd_abs simd_clamp simd_equal		X	X	X	X		X	X	X (clamp/ equal)	X
	abs clamp				X (2/3/4)					X (clamp: 2/3/4)	
Reduce Functions	simd_reduce_min simd_reduce_max simd_reduce_add		X	X	X	X		X	X	X	X
	reduce_min reduce_max reduce_add				X (2/3/4)					X (2/3/4)	
Extrema Functions	simd_min simd_max		X	X	X	X		X	X	X	X
	min max				X (2/3/4)					X (2/3/4)	
Logic and Bitwise Functions	simd_any simd_all simd_bitselect		X	X	X	X		X	X	X	X

Floating-Point Vectors

Functions			Single-Precision	Double-Precision
		simd_float1 simd_double1	simd_float2, simd_float3, simd_float4, simd_float8, simd_float16	simd_double2, simd_double3, simd_double4, simd_double8
			vector_Y	vector_Y
		Float Double	SIMDn<Float>	SIMDn<Double>
Functions to Create From Other Vectors, Scalar Values, Combinations of Vector and Scalars	simd_make_Y vectorN simd_make_Y_undef		X *Y: float2, float3, float4, float8, float16 *N: 2, 3, 4, 8, 16	X *Y: double2, double3, double4, double8 *N: 2, 3, 4
Functions to Create From Vectors of Other Types	simd_T		X * T: float	X * T: double
Common Functions	simd_abs, simd_equal		X	X
	simd_clamp, simd_fract simd_sign, simd_step	X	X	X
	abs, clamp, fact		X (float2/3/4)	X (double2/3/4)
	sign, step	X	X (float2/3/4)	X (double2/3/4)
Reduce Functions	simd_reduce_min simd_reduce_max simd_reduce_add		X	X
	reduce_min, reduce_max reduce_add		X (float2/3/4)	X (double2/3/4)
Interpolation Functions	simd_mix, simd_smoothstep	X	X	X
	mix, smooth step		X (float2/3/4)	X (double2/3/4)
Extrema Functions	simd_min, simd_max	X	X	X
	min, max	X (double1)	X (float2/3/4)	X (double2/3/4)
	fmin, fmam		X (float2/3/4)	X (double2/3/4)
Reciprocal and Reciprocal Square Root Functions	simd_recip, _precise_, _fast_ simd_rsqrt, _precise_, _fast_	X	X	X
	recip, rsqrt	X	X (float2/3/4)	X (double2/3/4)
Exponential and Logarithmic Functions	exp, exp2, exp10, expm1		X	X
	log, log2, log10, log1p		X	X
Geometry Functions	simd_cross		X (float2/3)	X (double2/3)
	cross		X (float2/3)	X (double2/3)
	simd_dot		X	X
	dot		X (float2/3/4)	X (double2/3/4)
	simd_incircle, simd_insphere		X (float2: incircle, float3: insphere)	X (float2: incircle, float3: insphere)
	simd_normalize, _precise_, _fast_ simd_project, _precise_, _fast_		X	X
	simd_orient, simd_reflect, simd_refract		X (float2/3/4)	X (double2/3/4)
	normalize, project, reflect, refract		X (float2/3/4)	X (double2/3/4)
Vector Norm Functions	simd_norm_one, simd_norm_inf		X	X
	norm_one, norm_inf		X (float2/3/4)	X (double2/3/4)
Length and Distance Functions	simd_length, _precise_, _fast_ simd_distance, _precise_, _fast_		X	X
	simd_length_squared, simd_distance_squared		X	X
	length, length_squared, distance, distance_squared		X (float2/3/4)	X (double2/3/4)
Hyperbolic Functions	acosh, asinh, atanh, cosh, sinh, tanh		X	X
Logic Functions	simd_select, simd_bitselect		X	X
Math Functions	cbrt, ceil, ert, erfc, floor, fma, fmod, hypot, lgamma, nextafter, pow, remainder, round, simd_muladd, tgamma, trunc	X (only simd_muladd)	X (float8/16: a part of)	X (double8: a part of)
Trigonometric Functions	acos, asin, atan, atan2, cos, cospi, sin, sinpi, tan, tanpi		X	X
Classification Functions	isfinite, isinf, isnan, isnormal		X	X

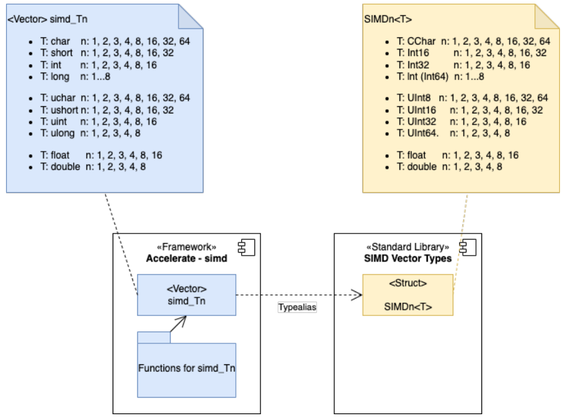
Note

Accelerate simd library　(1) Vectors

Accelerate simd library (iOS 11+)

- **Vectors: simd_Tn**
  - Signed Integer Vectors / Unsigned Integer Vectors
  - Floating-Point Vectors

- Since simd types are typealias of SIMD types, you can take advantage of SIMD type functions.



Sample: Integer Vectors

accelerate_simd_vector

```
import simd

// Create vectors
vector_uchar2(0, 0) // typealias                                                  SIMD2<UInt8>(0, 0)
simd_make_uchar4(255) // (Uint8)                                                 SIMD4<UInt8>(255, 0, 0, 0)
simd_make_uchar4(1, 2, 3, 4) // (Uint8 x4)                                       SIMD4<UInt8>(1, 2, 3, 4)
vector4(1, 2, 3, 4) // (Uint8 x4)                                               SIMD4<Int>(1, 2, 3, 4)
simd_make_uchar4(simd_make_uchar3(1, 2, 3), 4)                                   SIMD4<UInt8>(1, 2, 3, 4)
simd_uchar(simd_int4(1, 2, 3, 4))                                                SIMD4<UInt8>(1, 2, 3, 4)
simd_uchar(simd_int4(255, 256, 257, -1))                                         SIMD4<UInt8>(255, 0, 1, 255)
simd_uchar_sat(simd_int4(255, 256, 257, -1))                                     SIMD4<UInt8>(255, 255, 255, 0)
simd_clamp(simd_uchar4(10, 20, 30, 40), simd_uchar4(repeating: 20),              SIMD4<UInt8>(20, 20, 30, 30)
    simd_uchar4(repeating: 30)) // x, min, max
simd_equal(simd_uchar4(1, 2, 3, 4), simd_uchar4(1, 2, 3, 4))                    true

// Extrema Functions
simd_min(simd_uchar4(1, 2, 3, 4), simd_uchar4(4, 3, 2, 1))                      SIMD4<UInt8>(1, 2, 2, 1)
simd_max(simd_uchar4(1, 2, 3, 4), simd_uchar4(4, 3, 2, 1))                      SIMD4<UInt8>(4, 3, 3, 4)

// Logic and Bitwise Functions
simd_any(simd_uchar4(128, 0, 0, 0))                                             true
simd_all(simd_uchar4(128, 128, 128, 128))                                       true
simd_bitselect(simd_uchar4(1, 2, 3, 4), simd_uchar4(11, 12, 13, 14),            SIMD4<UInt8>(11, 2, 13, 4)
    simd_char4(-1, 0, -1, 0)) // x, y, mask

// init, methods, and operators From SIMD Vector Types
simd_uchar4(repeating: 0)                                                        SIMD4<UInt8>(0, 0, 0, 0)
simd_uchar4.zero                                                                 SIMD4<UInt8>(0, 0, 0, 0)
simd_uchar4.random(in: 0...200)                                                  SIMD4<UInt8>(113, 45, 185, 154)
simd_uchar4(1, 2, 3, 4) &* 2                                                     SIMD4<UInt8>(2, 4, 6, 8)
simd_uchar4(1, 2, 3, 4) / 2                                                      SIMD4<UInt8>(0, 1, 1, 2)
simd_uchar4(1, 2, 3, 4) &+ 10                                                    SIMD4<UInt8>(11, 12, 13, 14)
simd_uchar4(1, 2, 3, 4) == simd_uchar4(1, 2, 3, 4)                               true
simd_uchar4(1, 2, 3, 4) .== simd_uchar4(1, 2, 10, 20)                            SIMDMask<SIMD4<Int8>>(true, true, false, false)
```

Sample: Floating-point Vectors

```
import simd

// Create floating-point vectors
vector_float2(0, 0) // typealias                                                 SIMD2<Float>(0.0, 0.0)
simd_make_float2(1)                                                              SIMD2<Float>(1.0, 0.0)
simd_make_float2(1, 2)                                                           SIMD2<Float>(1.0, 2.0)
simd_make_float4(simd_make_float3(1, 2, 3), 4)                                   SIMD4<Float>(1.0, 2.0, 3.0, 4.0)
simd_make_float4_undef(1)                                                        SIMD4<Float>(1.0, 1e-45, 3.1405114e-36, -1.6947658e+37)
simd_float(simd_int4(1, 2, 3, 4))                                                SIMD4<Float>(1.0, 2.0, 3.0, 4.0)

// Common Functions
simd_abs(simd_float2(-1, -2))                                                    SIMD2<Float>(1.0, 2.0)
simd_equal(simd_float2(1, 2), simd_float2(1, 3))                                 false
simd_clamp(simd_float2(-1, 2), simd_float2.zero, simd_float2.one) // x, min, max SIMD2<Float>(0.0, 1.0)
simd_fract(simd_float2(1.2, 1.3)) // fractional part                            SIMD2<Float>(0.20000005, 0.29999995)
simd_sign(simd_float2(-2.5, 2.5)) // sign                                       SIMD2<Float>(-1.0, 1.0)
simd_step(simd_float2(10, 20), simd_float2(8, 21)) // edge, x (0: less than edge) SIMD2<Float>(0.0, 1.0)

// Reduce Functions
simd_reduce_add(simd_float2(1, 2)) // sum of all elements                        3
simd_reduce_max(simd_float2(-1, 1)) // max value of all elements                1
simd_reduce_min(simd_float2(-1, 1)) // min value of all elements                -1

// Interpolation Functions
simd_mix(simd_float2.zero, simd_float2.one, simd_float2(0.3, 0.5)) // linear: x, y, t SIMD2<Float>(0.3, 0.5)
simd_smoothstep(simd_float2.zero, simd_float2.one, simd_float2(0.1, 0.9)) // smooth: edge0, edge1, x SIMD2<Float>(0.028, 0.97199994)

// Extrema Functions
simd_max(simd_float2(1, 2), simd_float2(3, 1))                                   SIMD2<Float>(3.0, 2.0)
simd_min(simd_float2(-1, -2), simd_float2(0, -3))                                SIMD2<Float>(-1.0, -3.0)

// Reciprocal and Reciprocal Square Root Functions
simd_recip(simd_float2(2, 3)) // reciprocal                                      SIMD2<Float>(0.5, 0.33333334)
simd_fast_recip(simd_float2(2, 3)) // reciprocal                               SIMD2<Float>(0.4999981, 0.33333302)
simd_rsqrt(simd_float2(2, 3)) // reciprocal square root                         SIMD2<Float>(0.7071068, 0.57735026)
simd_fast_rsqrt(simd_float2(2, 3)) // reciprocal square root                    SIMD2<Float>(0.70709807, 0.5773467)

// Exponential and Logarithmic Functions
exp(simd_float2(1, 2)) // e raised to the power                                 SIMD2<Float>(2.7182817, 7.389056)
exp2(simd_float2(1, 2)) // 2 raised to the power                               SIMD2<Float>(2.0, 4.0)
exp10(simd_float2(-1, 2)) // 10 raised to the power                            SIMD2<Float>(0.1, 100.0)
expm1(simd_float2(1, 2)) // e^x −1                                             SIMD2<Float>(1.7182817, 6.389056)
log(simd_float2(1, 2)) // natural logarithm                                     SIMD2<Float>(0.0, 0.6931472)
log2(simd_float2(1, 256)) // base 2 logarithm                                  SIMD2<Float>(0.0, 8.0)
log10(simd_float2(1, 100)) // base 10 logarithm                                SIMD2<Float>(0.0, 2.0)
log1p(simd_float2(0, 1)) // log(1+x)                                           SIMD2<Float>(0.0, 0.6931472)

// Geometry Functions
simd_cross(simd_float2(1, 0), simd_float2(0, 1)) // cross product               SIMD3<Float>(0.0, 0.0, 1.0)
simd_dot(simd_float2(1, 2), simd_float2(3, 4)) // dot product                   11
simd_incircle(simd_float2.zero, simd_float2(0, 1), simd_float2(1, 0), simd_float2(0, -1)) // x, a, b, c: tests if in circle -2
simd_incircle(simd_float2(-1, 0), simd_float2(0, 1), simd_float2(1, 0), simd_float2(0, -1)) // x, a, b, c 0
simd_incircle(simd_float2(2, 0), simd_float2(0, 1), simd_float2(1, 0), simd_float2(0, -1)) // x, a, b, c 6
simd_normalize(simd_float2(2, 3)) // normalizes                               SIMD2<Float>(0.5547002, 0.8320503)
simd_orient(simd_float2(1, 0), simd_float2(0.5, 0.5)) // x, y: tests the orientation 0.5
simd_orient(simd_float2(1, 0), simd_float2(0.5, -0.5))                           -0.5
simd_project(simd_float2(0.5, 0.5), simd_float2(1, 0)) // 1st projected onto 2nd SIMD2<Float>(0.5, 0.0)
simd_reflect(simd_float2(0.5, -0.5), simd_float2(0, 1)) // x, n (unit normal): reflection direction SIMD2<Float>(0.5, 0.5)
simd_refract(simd_float2(0.5, -0.5), simd_float2(0, 1), 0.5) // x, n(unit normal), eta: refraction direction SIMD2<Float>(0.25, -0.9013878)

// Vector Norm Functions
simd_norm_one(simd_float2(-2, 2)) // sum of absolute values                     4
simd_norm_inf(simd_float2(-10, 2)) // max absolute value                        10

// Length and Distance Functions
simd_length(simd_float2(-3, 4)) // length                                       5
simd_length_squared(simd_float2(-3, 4)) // square of the length                25
simd_distance(simd_float2.zero, simd_float2.one) // distance of the two vectors 1.414214
simd_distance_squared(simd_float2.zero, simd_float2.one) // square of the distance 2

// Hyperbolic Functions
acosh(simd_float2(5, 3)) // inverse hyperbolic cosine                           SIMD2<Float>(2.2924316, 1.7627472)
asinh(simd_float2(-5, 5)) // inverse hyperbolic sine                           SIMD2<Float>(-2.3124385, 2.3124385)
atanh(simd_float2(-0.1, 0.1)) // inverse hyperbolic tangent                     SIMD2<Float>(-0.10033535, 0.10033535)
cosh(simd_float2(-1, 1)) // hyperbolic cosine                                  SIMD2<Float>(1.5430806, 1.5430806)
sinh(simd_float2(-1, 1)) // hyperbolic sine                                    SIMD2<Float>(-1.1752012, 1.1752012)
tanh(simd_float2(-10, 10)) // hyperbolic tangent                               SIMD2<Float>(-1.0, 1.0)

// Logic Functions
simd_select(simd_float2.zero, simd_float2.one, simd_int2(-1, 1)) // x, y, mask(high-order bit) SIMD2<Float>(1.0, 0.0)
simd_bitselect(simd_float2.zero, simd_float2.one, simd_int2(-1, 1)) // x, y, mask SIMD2<Float>(1.0, 0.0)

// Methods for SIMD Vector Type
simd_float2.one.addingProduct(simd_float2.one, 2) // a + b x c                  SIMD2<Float>(3.0, 3.0)
simd_float2(2, 5).squareRoot()                                                   SIMD2<Float>(1.4142135, 2.236068)
simd_float2(2, 5).squareRoot().rounded(.towardZero)                              SIMD2<Float>(1.0, 2.0)
simd_float2.random(in: 0...0.5)                                                  SIMD2<Float>(0.476167, 0.094211966)
simd_float2(1, 2) * simd_float2(3, 4)                                            SIMD2<Float>(3.0, 8.0)
simd_float2(1, 1) == simd_float2.one                                             true
```

Accelerate simd library　(2) Matrices

Accelerate simd library (iOS 11+)

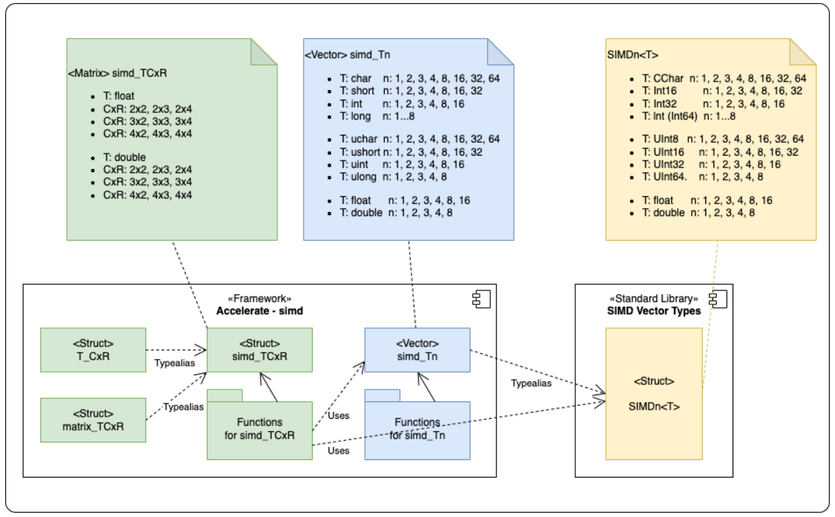
- Apple References
  - API: Accelerate simd ⋯ Perform computations on small vectors and matrices.
  - Article: Working with Matrices ⋯ Solve simultaneous equations and transform points in space.
  - WWDC18: Using Accelerate and simd (https://developer.apple.com/videos/play/wwdc2018/701/)
- **Matrices**
  - up to 4x4 (columns x rows : column major naming convention)
  - Single-Precision Floating-Point : **simd_floatCxR**
    - CxR: 2x2, 2x3, 2x4
    - CxR: 3x2, 3x3, 3x4
    - CxR: 4x2, 4x3, 4x4
  - Double-Precision Floating-Point Matrices : **simd_doubleCxR**
    - CxR: 2x2, 2x3, 2x4
    - CxR: 3x2, 3x3, 3x4
    - CxR: 4x2, 4x3, 4x4

Floating-Point Matrices

Members * T: float / double * CxR (cXr): 2x2 ⋯ 4x4 * TCxR: float2x2 ⋯ / double2x2 ⋯ * SIMDn<T>: SIMD2<float> ⋯ SIMD4<float> / SIMD2<double> ⋯ SIMD4<double> * quatP: quatf / quatd			Single-Precision <struct> simd_float2x2 /2x3 /2x4 simd_float3x2 /3x3 /3x4 simd_float4x2 /4x3 /4x4 <Typealias> floatCxR matrix_floatCxR	Double-Precision <struct> simd_double2x2 /2x3 /2x4 simd_double3x2 /3x3 /3x4 simd_double4x2 /4x3 /4x4 <Typealias> doubleCxR matrix_doubleCxR
Initializers	init()	Creates a new matrix.	X	X
	init(T)　T: Float / Double	Creates a new matrix with the scalar value on the main diagonal.	X	X
	init(diagonal: SIMDn<T>)	Creates a new matrix with the vector on the main diagonal.	X	X
	init([SIMDr<T>])	Creates a new matrix with the columns.	X	X
	init(SIMDr<T>, ⋯)	Creates a new matrix with the vectors as columns.	X	X
	init(columns: (simd_floatR, ⋯))	Creates a new matrix with the columns.	X	X
	init(rows: [SIMDc<T>])	Creates a new matrix with the rows.	X	X
	init(simd_quatP)	Creates a new matrix from the quarternion.	X (3x3, 4x4)	X (3x3, 4x4)
Matrix Constants	let matrix_identity_TCxR	Global Variable: A CxR identity matrix.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
Matix Properties	var determinant: T	The determinant of the matrix.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
	var inverse: simd_TCxR	The inverse of the matrix.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
	var transpose: TCxR	The transpose of the matrix.	X	X
	var columns: (simd_Tc, ⋯)	The columns of the matrix.	X	X
	var debugDescription: String	A representation of the matrix.	X	X
Matrix Creation Functions	simd_matrix(simd_Tr, ⋯) -> simd_TCxR	Returns a new matrix with the columns.	X	X
	simd_matrix_from_rows(simd_Tc, ⋯) -> simd_TCxR	Returns a new matrix with the rows.	X	X
	matrix_from_rows(SIMDc<T>, ⋯) -> simd_TCxR	Returns a new matrix with the rows.	X	X
	simd_diagonal_matrix(simd_Tn) -> simd_TCxR	Returns a new matrix with the vector on the main diagonal.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
	simd_matrixCxR(simd_quatP)	Returns a new matrix with the quarternion.	X (3x3, 4x4)	X (3x3, 4x4)
Math Functions	simd_add(simd_TCxR, simd_TCxR) -> simd_TCxR	Returns the sum of two matrices.	X	X
	simd_sub(simd_TCxR, simd_TCxR) -> simd_TCxR	Returns the difference of two matrices.	X	X
	matrix_scale(T, simd_TCxR) -> simd_TCxR	Returns the product of a scalar value and a matrix.	X	X
Matrix-Scalar Multiplication Functions	simd_mul(T, simd_TCxR) -> simd_TCxR	Returns the product of a scalar value and a matrix.	X	X
Matrix-Vector Multiplication Functions	simd_mul(simd_Tr, simd_TCxR) -> simd_Tc	Returns the product of a vector and a matrix.	X	X
	matrix_multiply(simd_Tr, simd_TCxR) -> simd_Tc	Returns the product of a vector and a matrix.	X	X
	simd_mul(simd_TCxR, simd_Tc) -> simd_Tr	Returns the product of a matrix and a vector.	X	X
	matrix_multiply(simd_TCxR, simd_Tc) -> simd_Tr	Returns the product of a matrix and a vector.	X	X

Floating-Point Matrices (cont)

Members * T: float / double * CxR: 2x2 ⋯ 4x4 * TCxR: float2x2 ⋯ / double2x2 ⋯ * SIMDn<T>: SIMD2<float> ⋯ SIMD4<float> / SIMD2<double> ⋯ SIMD4<double>			Single-Precision <struct> simd_float2x2 /2x3 /2x4 simd_float3x2 /3x3 /3x4 simd_float4x2 /4x3 /4x4 <Typealias> floatCxR matrix_floatCxR	Double-Precision <struct> simd_double2x2 /2x3 /2x4 simd_double3x2 /3x3 /3x4 simd_double4x2 /4x3 /4x4 <Typealias> doubleCxR matrix_doubleCxR
Matrix-Matrix Multiplication Functions	simd_mul(simd_Txxx, simd_Tyyy) -> simd_Tzzz	Returns the product of two matrices.	X	X
	matrix_multiply(simd_Txxx, simd_Tyyy) -> simd_Tzzz	Returns the product of two matrices.	X	X
Equality Functions	simd_equal(simd_TCxR, simd_TCxR) -> simd_bool	Returns true if every element is equal.	X	X
	simd_almost_equal_elements(simd_TCxR, simd_TCxR, T) -> simd_bool	Returns true if every element is within the tolerance.	X	X
	simd_almost_equal_elements_relative(simd_TCxR, simd_TCxR, T) -> simd_bool	Returns true if every element is within the tolerance.	X	X
Determinant and Inverse Functions	simd_determinant(simd_TCxR) -> T	Returns the determinant of the matrix.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
	simd_inverse(simd_TCxR) -> simd_TCxR	Returns the inverse of the matrix.	X (2x2, 3x3, 4x4)	X (2x2, 3x3, 4x4)
Linear Combination Function	simd_linear_combination(T, simd_TCxR, T, simd_TCxR) -> simd_TCxR	Returns the linear combination of two scalar values and two matrices.	X	X
Transpose Function	simd_transpose(simd_TCxR) -> simd_TRxC	Returns the transpose of a matrix.	X	X
Operators	* (simd_TCxR, SIMDc<T>) -> SIMDn<T>		X	X
	* (SIMDr<T>, simd_TCxR) -> SIMDn<T>		X	X
	* (simd_TCxR, Txxx) -> Tyyy		X	X
	* (T, simd_TCxR) -> simd_TCxR		X	X
	* (simd_TCxR, T) -> simd_TCxR		X	X
	*= (simd_TCxR, T)		X	X
	*= (simd_TCxR, Txxx)		X	X
	+ (simd_TCxR, simd_TCxR)		X	X
	+= (simd_TCxR, simd_TCxR)		X	X
	- (simd_TCxR) -> simd_TCxR		X	X
	- (simd_TCxR, simd_TCxR) -> simdTCxR		X	X
	-= (simd_TCxR, simd_TCxR)		X	X
	== (simd_TCxR, simd_TCxR) -> Bool		X	X



Accelerate simd library　(2) Matrices

Accelerate simd library (iOS 11+)

- Apple References
  - API: Accelerate simd ⋯ Perform computations on small vectors and matrices.
  - Article: Working with Matrices ⋯ Solve simultaneous equations and transform points in space.
  - WWDC18: Using Accelerate and simd (https://developer.apple.com/videos/play/wwdc2018/701/)
- **Matrices**
  - up to 4x4 (columns x rows : column major naming convention)
  - Single-Precision Floating-Point : **simd_floatCxR**
    - CxR: 2x2, 2x3, 2x4
    - CxR: 3x2, 3x3, 3x4
    - CxR: 4x2, 4x3, 4x4
  - Double-Precision Floating-Point Matrices : **simd_doubleCxR**
    - CxR: 2x2, 2x3, 2x4
    - CxR: 3x2, 3x3, 3x4
    - CxR: 4x2, 4x3, 4x4

Sample: Floating-point Matrices

```
import simd

// Initializers
simd_float2x3()                                                                 // simd_float2x3([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
simd_float2x3(1)                                                                // simd_float2x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0]])
simd_float2x2(diagonal: SIMD2<Float>.one * 2)                                   // simd_float2x2([[2.0, 0.0], [0.0, 2.0]])
simd_float2x3([SIMD3<Float>(1, 2, 3), SIMD3<Float>(4, 5, 6)])                   // simd_float2x3([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
simd_float2x3(SIMD3<Float>(1, 2, 3), SIMD3<Float>(4, 5, 6))                     // simd_float2x3([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
simd_float2x3(columns: (simd_float3.zero, simd_float3.one))                     // simd_float2x3([[0.0, 0.0, 0.0], [1.0, 1.0, 1.0]])
simd_float2x3(rows: [SIMD2<Float>.zero, SIMD2<Float>.one, SIMD2<Float>.zero])   // simd_float2x3([[0.0, 1.0, 0.0], [0.0, 1.0, 0.0]])
simd_float3x3(simd_quatf()) // from a quarternion                               // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

// Matrix Constants
matrix_identity_float3x3 // Global Variable                                     // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

// Matrix Properties
simd_float3x3(1).determinant                                                    // 1
simd_float3x3(1).inverse                                                        // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
simd_float3x3(1).transpose                                                      // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
simd_float3x3(1).columns.0                                                      // SIMD3<Float>(1.0, 0.0, 0.0)
simd_float3x3(1).debugDescription                                               // "simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])"

// Matrix Creation Functions
simd_matrix(simd_float2(1, 2), simd_float2(3, 4)) // columns                    // simd_float2x2([[1.0, 2.0], [3.0, 4.0]])
simd_matrix_from_rows(simd_float2(1, 2), simd_float2(3, 4)) // rows             // simd_float2x2([[1.0, 3.0], [2.0, 4.0]])
matrix_from_rows(SIMD2<Float>(1, 2), SIMD2<Float>(3, 4)) // rows               // simd_float2x2([[1.0, 3.0], [2.0, 4.0]])
simd_diagonal_matrix(simd_float2(1, 2))                                         // simd_float2x2([[1.0, 0.0], [0.0, 2.0]])
simd_matrix3x3(simd_quatf()) // from a quarternion                             // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])

// Math Functions
simd_add(simd_float3x3(1), simd_float3x3(2))                                    // simd_float3x3([[3.0, 0.0, 0.0], [0.0, 3.0, 0.0], [0.0, 0.0, 3.0]])
simd_sub(simd_float3x3(2), simd_float3x3(1))                                    // simd_float3x3([[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]])
matrix_scale(2, simd_float3x3(1))                                               // simd_float3x3([[2.0, 0.0, 0.0], [0.0, 2.0, 0.0], [0.0, 0.0, 2.0]])

// Matrix-Scalar Multiplication Functions
simd_mul(2, simd_float3x3(1))                                                   // simd_float3x3([[2.0, 0.0, 0.0], [0.0, 2.0, 0.0], [0.0, 0.0, 2.0]])

// Matrix-Vector Multiplication Functions
simd_mul(simd_float2(1, 2), simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)))                // SIMD2<Float>(5.0, 11.0)
matrix_multiply(simd_float2(1, 2), simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)))         // SIMD2<Float>(5.0, 11.0)
simd_mul(simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)), simd_float2(1, 2))                // SIMD2<Float>(7.0, 10.0)
matrix_multiply(simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)), simd_float2(1, 2))         // SIMD2<Float>(7.0, 10.0)

// Matrix-Matrix Multiplication Functions
simd_mul(simd_float2x3(simd_float3(1, 2, 3), simd_float3(4, 5, 6)), simd_float3x2(simd_float2(1, 2),
    simd_float2(3, 4), simd_float2(5, 6)))                                      // simd_float3x3([[9.0, 12.0, 15.0], [19.0, 26.0, 33.0], [29.0, 40.0, 51.0]])

```

```
// Equality Functions
simd_equal(simd_float3x3(1), simd_float3x3(1))                                  // true
simd_almost_equal_elements(simd_float3x3(1), simd_float3x3(1.09), 0.1)  // x, y, tolerance   // true
simd_almost_equal_elements(simd_float3x3(1), simd_float3x3(0.91), 0.1)  // x, y, tolerance   // true
simd_almost_equal_elements_relative(simd_float3x3(1), simd_float3x3(0.91), 0.1) // x, y, tol  // true

// Determinant and Inverse Functions
simd_determinant(simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)))           // -2
simd_inverse(simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)))               // simd_float2x2([[-2.0, 1.0], [1.5, -0.5]])

// Linear Combination Function
simd_linear_combination(0.1, simd_float2x2(simd_float2.one, simd_float2.one), 0.9,  // simd_float2x2([[1.9, 1.9], [1.9, 1.9]])
    simd_float2x2(simd_float2.one, simd_float2.one)*2) // a, x, b, y

// Transpose Function
simd_transpose(simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)))             // simd_float2x2([[1.0, 3.0], [2.0, 4.0]])

// Element Access (Subscript)
simd_float2x2(simd_float2(1, 2), simd_float2(3, 4))[0] // column                 // SIMD2<Float>(1.0, 2.0)
simd_float2x2(simd_float2(1, 2), simd_float2(3, 4))[0][1]                       // 2

// Typealias
float3x3()                                                                      // simd_float3x3([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])
matrix_float3x3()                                                               // simd_float3x3([[0.0, 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]])

// Operators
simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)) * 2                         // simd_float2x2([[2.0, 4.0], [6.0, 8.0]])
simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)) * simd_float2(5, 6)         // SIMD2<Float>(23.0, 34.0)
simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)) + simd_float2x2(1)          // simd_float2x2([[2.0, 2.0], [3.0, 5.0]])
-simd_float2x2(simd_float2(1, 2), simd_float2(3, 4)) - simd_float2x2(1)         // simd_float2x2([[-2.0, -2.0], [-3.0, -5.0]])
simd_float3x3(1) == simd_float3x3(1)                                            // true
simd_float3x3(1) != simd_float3x3(1)                                            // false
```

Hide or show the Navigator

Note

Accelerate simd library　(3) Quaternions

Accelerate simd library (iOS 11+)

- Apple References
  - API: Accelerate simd ⋯ Perform computations on small vectors and matrices.
  - Article: Working with Quaternions ⋯ Rotate points around the surface of a sphere, and interpolate between them.
  - Sample Code: Rotating a Cube by Transforming Its Vertices ⋯ Use quaternion interpolation to rotate a cube.
  - WWDC18: Using Accelerate and simd (https://developer.apple.com/videos/play/wwdc2018/701/)
- **Quaternions**
  - Rotate points around the surface of a sphere, and interpolate between them.
  - Quaternions are defined by a scalar (real) part, and three imaginary parts collectively called the vector part. Quaternions are often used in graphics programming as a compact representation of the rotation of an object in three dimensions.
  - The length of a quaternion is the square root of the sum of the squares of its components.
  - Quaternions have some advantages over matrices. For example, they're smaller: A 3 x 3 matrix of floats is 48 bytes, and a single-precision quaternion is 16 bytes. They also can offer better performance: Although a single rotation using a quaternion is a little slower than one using a matrix, when combining actions, quaternions can be up to 30% faster.
  - Single-Precision Quaternion : **simd_quatf**
  - Double-Precision Quaternion : **simd_quatd**

Quaternions

Properties and Support Functions * t: float / double * T: Float / Double * quatP: quatf / quatd			Single-Precision <struct> simd_quatf	Double-Precision <struct> simd_quatd
Initializers	init()	Creates a new quaternion.	X	X
	init(vector: simd_t4)	Creates a new quaternion from a vector. [0, 1, 2]: img, [3]: real	X	X
	init(simd_t3x3)	Creates a new quaternion from a 3x3 rotation matrix, which must be orthogonal and have a determinant of 1.	X	X
	init(simd_t4x4)	Creates a new quaternion from a 4x4 rotation matrix. The last row and the last column of rotation matrix are ignored.	X	X
	int(angle: Float, axis: SIMD3<T>)	Creates a new quaternion with an action that's a rotation about an axis.	X	X
	init(from: SIMD3<T>, to: SIMD3<T>)	Creates a new quaternion with an action that's a rotation between two vectors. Vectors must be normalized.	X	X
	init(ix: T, iy: T, iz: T, r: T)	Creates a new quaternion from scalar values.	X	X
	init(real: T, imag: SIMD3<T>)	Creates a new quaternion from a scalar value and a vector.	X	X
Quaternion Properties	var angle: Float { get }	The angle, in radians, by which the quaternion's action rotates.	X	X
	var axis: SIMD3<T>　{ get }	The normalized axis about which the quaternion's action rotates.	X	X
	var conjugate:　simd_quatP　{ get }	The conjugate of the quaternion.	X	X
	var imag: SIMD3<T>　{ get set }	The imaginary part of the quaternion.	X	X
	var real: T　{ get set }	The real part of the quaternion.	X	X
	var inverse: simd_quatP　{ get }	The inverse of the quaternion.	X	X
	var length: T　{ get }	The length of the quaternion.	X	X
	var normalized: simd_quatP　{ get }	The unit quaternion of the quaternion.	X	X
	var vector: simd_f4	The underlying vector of the quaternion.	X	X
	var debugDescription: String { get }	A representation of the quaternion.	X	X
Creation Functions	simd_quaternion(T, T, T, T) -> simd_quatP	Returns a new quaternion from scalar values. (ix, iy, iz, r)	X	X
	simd_quaternion(T, simd_t3) -> simd_quatP	Returns a new quaternion from a scalar and a vector. (angle, axis)	X	X
	simd_quaternion(UnsafePointer<T>!) -> simd_quatP	Return a new quaternion from a pointer to scalar values.	X	X
	simd_quaternion(simd_t3, simd_t3) -> simd_quatP	Return a new quaternion from two vectors. (from, to)	X	X
	simd_quaternion(simd_t3x3) -> simd_quatP	Returns a new quaternion from a matrix.	X	X
	simd_quaternion(simd_t4) -> simd_quatP	Returns a new quaternion from a vector.	X	X
	simd_quaternion(simd_t4x4) -> simd_quatP	Returns a new quaternion form a matrix.	X	X

Quaternions (cont)

Properties and Support Functions * t: float / double * T: Float / Double * quatP: quatf / quatd			Single-Precision <struct> simd_quatf	Double-Precision <struct> simd_quatd
Math Functions	simd_add(simd_quatP, simd_quatP) -> simd_quatP	Returns the sum of two quaternions.	X	X
	simd_mul(simd_quatP, simd_quatP) -> simd_quatP	Returns the product of two quaternions.	X	X
	simd_mul(T, simd_quatP) -> simd_quatP		X	X
	simd_mul(simd_quatP, T) -> simd_quatP		X	X
	simd_sub(simd_quatP, simd_quatP) -> simd_quatP	Returns the difference between two quaternions.	X	X
Quaternion Functions	simd_act(simd_quatP, simd_t3) -> simd_t3	Returns a vector rotated by a quaternion.	X	X
	simd_angle(simd_quatP) -> T	Returns the angle by which a quaternion rotates.	X	X
	simd_axis(simd_quatP) -> simd_t3	Returns the axis about which a quaternion rotates.	X	X
	simd_bezier(simd_quatP, simd_quatP, simd_quatP, quatP, Float) -> simd_quatP	Returns the spherical cubic Bezier interpolation between quaternions. (q0, q1, q2, q3, t)	X	X
	simd_conjugate(simd_quatP) -> simd_quatP	Returns the conjugate of a quaternion.	X	X
	simd_imag(simd_quatP) -> simd_t3	Returns the imaginary (vector) part.	X	X
	simd_negate(simd_quatP) -> simd_quatP	Returns the negation.	X	X
	simd_real(simd_quatP) -> T	Returns the real (scalar) part.	X	X
	simd_slerp(simd_quatP, simd_quatP, T) -> simd_quatP	Returns a spherical linearly interpolated value along the shortest arc between two quaternions. (q0, q1, t)	X	X
	simd_slerp_longest(simd_quatP, simd_quatP, T) -> simd_quatP	Returns a spherical linearly interpolated value along the longest arc between two quaternions. (q0, q1, t)	X	X
	simd_spline(simd_quatP, simd_quatP, simd_quatP, simd_quatP, T) -> simd_quatP	Returns an interpolated value between two quaternions along a spherical cubic spline. (q0, q1, q2, q3, t)	X	X
Geometry Functions	simd_dot(simd_quatP, simd_quatP) -> T	Returns the dot product of two quaternions.	X	X
	dot(simd_quatP, simd_quatP) -> T		X	X
	simd_length(simd_quatP) -> T	Returns the length of a quaternion.	X	X
	simd_normalize(simd_quatP) -> T	Returns a quaternion of length 1.	X	X
	exp(simd_quatP) -> simd_quatP	e^q	X	X (Float)
	log(simd_quatP) -> simd_quatP	log q	X	X (Float)
Inverse Function	simd_inverse(simd_quatP) -> simd_quatP	Returns the inverse of a quaternion.	X	X
Instance Methods	act(SIMD3<T>) -> SIMD3<T>	Returns the vector rotated by the quaternion.	X	X
Operators	* (T, simd_quatP) -> simd_quatP * (simd_quatP, T) -> simd_quatP * (simd_quatP, simd_quatP) -> simd_quatP		X	X
	*= (inout simd_quatP, T) *= (inout simd_quatP, simd_quatP)		X	X
	+ (simd_quatP, simd_quatP) -> simd_quatP		X	X
	+= (inout simd_quatP, simd_quatP)		X	X
	- (simd_quatP) -> simd_quatP		X	X
	- (simd_auatP, simd_quatP) -> simd_quatP		X	X
	-= (inout simd_quatP, simd_quatP)		X	X
	/ (simd_quatP, simd_quatP) -> simd_quatP / (simd_quatP, T) -> simd_quatP		X	X
	/= (inout simd_quatP, T) /= (inout simd_quatP, simd_quatP)		X	X
	== (simd_quatP, simd_quatP) -> Bool		X	X

Accelerate simd library　(3) Quaternions

Accelerate simd library (iOS 11+)

- Apple References
  - API: Accelerate simd ⋯ Perform computations on small vectors and matrices.
  - Article: Working with Quaternions ⋯ Rotate points around the surface of a sphere, and interpolate between them.
  - Sample Code: Rotating a Cube by Transforming Its Vertices ⋯ Use quaternion interpolation to rotate a cube.
  - WWDC18: Using Accelerate and simd (https://developer.apple.com/videos/play/wwdc2018/701/)
- **Quaternions**
  - Rotate points around the surface of a sphere, and interpolate between them.
  - Quaternions are defined by a scalar (real) part, and three imaginary parts collectively called the vector part. Quaternions are often used in graphics programming as a compact representation of the rotation of an object in three dimensions.
  - The length of a quaternion is the square root of the sum of the squares of its components.
  - Quaternions have some advantages over matrices. For example, they're smaller: A 3 x 3 matrix of floats is 48 bytes, and a single-precision quaternion is 16 bytes. They also can offer better performance: Although a single rotation using a quaternion is a little slower than one using a matrix, when combining actions, quaternions can be up to 30% faster.
  - Single-Precision Quaternion : **simd_quatf**
  - Double-Precision Quaternion : **simd_quatd**

Sample: Quaternions

```
import simd

// Initializers
simd_quatf()                                                                                                    // simd_quatf(real: 0.0, imag: SIMD3<Float>(0.0, 0.0, 0.0))
simd_quatf(vector: simd_float4(0, 0, 0, 1))                                                                     // simd_quatf(real: 1.0, imag: SIMD3<Float>(0.0, 0.0, 0.0))
simd_quatf(simd_float3x3(SIMD3<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0),                                    // simd_quatf(real: 0.70710677, imag: SIMD3<Float>(0.0, 0.0, -0.70710677))
    SIMD3<Float>(sin(Float.pi/2), cos(Float.pi/2), 0), SIMD3<Float>(0, 0, 1))) // from 3x3
    rotation matrix, which must be orthogonal and have a determinant of 1
simd_float3x3(SIMD3<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0),                                               // 1
    SIMD3<Float>(sin(Float.pi/2), cos(Float.pi/2), 0), SIMD3<Float>(0, 0, 1)).determinant
simd_quatf(simd_float4x4(SIMD4<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0, 0),                                 // simd_quatf(real: 0.70710677, imag: SIMD3<Float>(0.0, 0.0, -0.70710677))
    SIMD4<Float>(sin(Float.pi/2), cos(Float.pi/2), 0, 0), SIMD4<Float>(0, 0, 1, 0),
    SIMD4<Float>(0, 0, 0, 1))) // from 4x4 rotation matrix, whose 3x3 part must be
    orthogonal and have a determinant of 1
simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 0, -1)) // rotation about an axis                            // simd_quatf(real: 0.7071068, imag: SIMD3<Float>(0.0, 0.0, -0.70710677))
simd_quatf(from: SIMD3<Float>(0, 1, 0), to: SIMD3<Float>(1, 0, 0)) // rotation between two                       // simd_quatf(real: 0.7071068, imag: SIMD3<Float>(0.0, 0.0, -0.7071068))
    vectors. They should be normalized.
simd_quatf(ix: 0, iy: 0, iz: 0, r: 1) // from scalar values                                                     // simd_quatf(real: 1.0, imag: SIMD3<Float>(0.0, 0.0, 0.0))
simd_quatf(real: 1, imag: SIMD3<Float>.zero) // from a scalar value and a vector                                // simd_quatf(real: 1.0, imag: SIMD3<Float>(0.0, 0.0, 0.0))

// Quaternion Properties
simd_quatf(simd_float3x3(SIMD3<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0),                                    // 1.570796
    SIMD3<Float>(sin(Float.pi/2), cos(Float.pi/2), 0), SIMD3<Float>(0, 0, 1))).angle
simd_quatf(simd_float3x3(SIMD3<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0),                                    // SIMD3<Float>(0.0, 0.0, -0.99999994)
    SIMD3<Float>(sin(Float.pi/2), cos(Float.pi/2), 0), SIMD3<Float>(0, 0, 1))).axis
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).conjugate // conjugate                                                     // simd_quatf(real: 4.0, imag: SIMD3<Float>(-1.0, -2.0, -3.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).imag                                                                      // SIMD3<Float>(1.0, 2.0, 3.0)
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).real                                                                      // 4
simd_quatf(simd_float3x3(SIMD3<Float>(cos(Float.pi/2), -sin(Float.pi/2), 0),                                    // simd_quatf(real: 0.70710677, imag: SIMD3<Float>(-0.0, -0.0, 0.70710677))
    SIMD3<Float>(sin(Float.pi/2), cos(Float.pi/2), 0), SIMD3<Float>(0, 0, 1))).inverse
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).length                                                                    // 5.477226
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).normalized                                                                // simd_quatf(real: 0.73029673, imag: SIMD3<Float>(0.18257418, 0.36514837, 0.5477226))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).normalized.length                                                         // 1
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4).vector                                                                    // SIMD4<Float>(1.0, 2.0, 3.0, 4.0)

// Math Functions
simd_add(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4), simd_quatf(ix: 1, iy: 2, iz: 3, r: 4))                          // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_mul(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4), simd_quatf(ix: 5, iy: 6, iz: 7, r: 8))                          // simd_quatf(real: -6.0, imag: SIMD3<Float>(24.0, 48.0, 48.0))
simd_mul(simd_quatf(ix: 5, iy: 6, iz: 7, r: 8), simd_quatf(ix: 1, iy: 2, iz: 3, r: 4))                          // simd_quatf(real: -6.0, imag: SIMD3<Float>(32.0, 32.0, 56.0))
simd_mul(2, simd_quatf(ix: 1, iy: 2, iz: 3, r: 4))                                                              // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_mul(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4), 2)                                                              // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_sub(simd_quatf(ix: 1, iy: 1, iz: 1, r: 1), simd_quatf(ix: 3, iy: 4, iz: 5, r: 2))                          // simd_quatf(real: -1.0, imag: SIMD3<Float>(-2.0, -3.0, -4.0))

```

Note

```
// Quaternion Functions
simd_act(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 0, -1)), simd_float3(1, 0, 0))                    // SIMD3<Float>(7.671892e-08, -1.0, 0.0)
simd_angle(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 0, -1)))                                         // 1.570796
simd_axis(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 0, -1)))                                          // SIMD3<Float>(0.0, 0.0, -0.99999994)
simd_negate(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4))                                                              // simd_quatf(real: -4.0, imag: SIMD3<Float>(-1.0, -2.0, -3.0))
simd_slerp(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(1, 0, 0)) ,simd_quatf(angle:                        // simd_quatf(real: 0.7071068, imag: SIMD3<Float>(0.70710677, 0.0, 0.0))
    Float.pi/2, axis: SIMD3<Float>(0, 1, 0)), 0) // (q0, q1, t)
simd_slerp(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(1, 0, 0)) ,simd_quatf(angle:                        // simd_quatf(real: 0.8164966, imag: SIMD3<Float>(0.40824828, 0.40824828, 0.0))
    Float.pi/2, axis: SIMD3<Float>(0, 1, 0)), 0.5) // (q0, q1, t)
simd_slerp(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(1, 0, 0)) ,simd_quatf(angle:                        // simd_quatf(real: 0.7071068, imag: SIMD3<Float>(0.0, 0.70710677, 0.0))
    Float.pi/2, axis: SIMD3<Float>(0, 1, 0)), 1.0) // (q0, q1, t)
simd_slerp_longest(simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(1, 0, 0))                                   // simd_quatf(real: 0.0, imag: SIMD3<Float>(0.7071068, -0.7071068, 0.0))
    ,simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 1, 0)), 0.5) // (q0, q1, t)

// Geometry Functions
simd_dot(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4), simd_quatf(ix: 5, iy: 6, iz: 7, r: 8))                          // 70
dot(simd_quatf(ix: 1, iy: 2, iz: 3, r: 4), simd_quatf(ix: 5, iy: 6, iz: 7, r: 8))                               // 70
exp(simd_quatf(ix: 1, iy: 1, iz: 1, r: 1))                                                                      // simd_quatf(real: -0.43643785, imag: SIMD3<Float>(1.5490403, 1.5490403, 1.5490403))

// Instance Methods
simd_quatf(angle: Float.pi/2, axis: SIMD3<Float>(0, 0, -1)).act(SIMD3<Float>(1, 0, 0))                          // SIMD3<Float>(7.671892e-08, -1.0, 0.0)

// Operators
2 * simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                                                       // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) * 2                                                                       // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) * simd_quatf(ix: 5, iy: 6, iz: 7, r: 8)                                   // simd_quatf(real: -6.0, imag: SIMD3<Float>(24.0, 48.0, 48.0))
simd_quatf(ix: 5, iy: 6, iz: 7, r: 8) * simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                   // simd_quatf(real: -6.0, imag: SIMD3<Float>(32.0, 32.0, 56.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) + simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                   // simd_quatf(real: 8.0, imag: SIMD3<Float>(2.0, 4.0, 6.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) - simd_quatf(ix: 1, iy: 1, iz: 1, r: 1)                                   // simd_quatf(real: 3.0, imag: SIMD3<Float>(0.0, 1.0, 2.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) / 2                                                                       // simd_quatf(real: 2.0, imag: SIMD3<Float>(0.5, 1.0, 1.5))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) / simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                   // simd_quatf(real: 0.9999999, imag: SIMD3<Float>(0.0, 0.0, 0.0))
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) == simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                  // true
simd_quatf(ix: 1, iy: 2, iz: 3, r: 4) != simd_quatf(ix: 1, iy: 2, iz: 3, r: 4)                                  // false
```