Sourcery VSIPL++

Reference Manual

Version 2.2-9

Sourcery VSIPL++: Reference Manual: Version 2.2-9

CodeSourcery, Inc.

Table of Contents

Chapter 1
VSIPL++ File Structure and Namespaces

Abstract

This chapter provides an introduction to the basic structure of a program which uses the VSIPL++ library, as well as the layout of header files and C++ namespaces within the library.

1.1. Initialization and Basic Program Structure

The minimum requirements for a VSIPL++ program are to include the `vsip/initfin.hpp` header, and to create a `vsip::vsipl` object before doing any VSIPL++ operations. For example:

```
#include <vsip/initfin.hpp>
int main()
{
  vsip::vsipl vsipl_initialization_object;
  // ...
}
```

Creating the `vsip::vsipl` object initializes the library, sets up common storage, and allocates coprocessors (if applicable). These are then freed when the object is destroyed. Typically, the `vsip::vsipl` object is declared as a local variable at the beginning of `main` and implicitly destroyed at the end of it, but this is not a requirement.

When multiple `vsip::vsipl` objects are created, the library is initialized when the first one is created, and subsequent object creations do nothing. Similarly, the library resources are only released when all of the existing `vsip::vsipl` objects have been destroyed. Thus, multiple components of a program can access the library independently without needing to explicitly coordinate a single VSIPL++ library initialization and finalization.

The behavior of the library can also be controlled by command-line arguments passed to the executable, as described in the User's Guide and Tutorial. To enable this, the `argc` and `argv` arguments to `main` should be passed to the contructor for the `vsip::vsipl` object, as in this example:

```
#include <vsip/initfin.hpp>
int main(int argc, char **argv)
{
  vsip::vsipl vsipl_initialization_object(argc, argv);
  // ...
}
```

The constructor then modifies the argument count and argument list, removing the arguments which it recognizes. Thus, when a program will be processing additional command-line arguments, it can be advantageous to call the `vsip::vsipl` constructor first; then, the code to process the additional arguments will not need to be modified to ignore the VSIPL++ arguments.

1.2. Namespaces

The VSIPL++ library uses a variety of C++ namespaces. Features that are part of the VSIPL++ standard are in the `vsip` namespace, while features that are CodeSourcery extensions to the standard are in the `vsip_csl` namespace. Both of these contain `impl` sub-namespaces, which hold implementation-specific features and functions.

The `vsip_csl` namespace also contains sub-namespaces for specific sets of features. The `vsip_csl::img` namespace contains features for image processing; the `vsip_csl::output` namespace contains features for data output, and the `vsip_csl::stencil` namespace contains features for stencil operators.

1.3. Header Files

Much like the C++ standard library, the VSIPL++ library contains separate header files for each feature set. Thus, a program unit should include the header files necessary for the features it uses. For example, as we have already seen, the `vsip::vsipl` initialization object requires the `vsip/initfin.hpp` header file.

The following header files are available:

`vsip/complex.hpp` defines VSIPL++'s complex number facilities.

`vsip/dense.hpp` defines VSIPL++'s `Dense` block type. This file is included automatically by the view header files.

`vsip/domain.hpp` defines VSIPL++'s `Domain` and `Index` handling.

`vsip/initfin.hpp` defines VSIPL++'s library initialization and finalization.

`vsip/map.hpp` defines VSIPL++'s `Map` class template for distributing data across multiple processors.

`vsip/math.hpp` defines VSIPL++'s mathematical operations on views.

`vsip/matrix.hpp` defines VSIPL++'s `Matrix` view class template

`vsip/parallel.hpp` defines VSIPL++'s parallel suppport functions.

`vsip/random.hpp` defines VSIPL++'s random number generator.

`vsip/selgen.hpp` defines VSIPL++'s selection and generation functions.

`vsip/signal.hpp` defines VSIPL++'s signal processing objects and functions.

`vsip/solvers.hpp` defines VSIPL++'s linear algebra solvers.

`vsip/support.hpp` defines VSIPL++'s support types.

`vsip/tensor.hpp` defines VSIPL++'s `Tensor` view class template.

`vsip/vector.hpp` defines VSIPL++'s `Vector` view class template.

The directory structure for the header files parallels the namespace structure. Thus, header files defining things in the `vsip` namespace will be in the `vsip` directory; header files defining things in the `vsip_csl::img` namespace will be in the `vsip_csl/img` directory, and so forth.

Chapter 2
Basic VSIPL++ Data Types and Constants

Abstract

This chapter describes the basic VSIPL++ data structures for representing 1, 2, and 3-dimensional sets of data, as well as supporting data types and constants.

2.1. Introduction

In VSIPL++, objects containing sets of data points are called "views". There are three primary classes of views, depending on the dimensionality of the data: `Vector` objects contain one-dimensional data sets, `Matrix` objects contain two-dimensional data sets, and `Tensor` objects contain three-dimensional data sets.

The following conventions are used within this documentation:

- Capitalized variables (`A`,`B`,`Bool_view`) refer to views: vector, matrix, or tensor values.
- Lower-case variables (`a`,`scale`) refer to scalar values.

View declarations are templated in terms of the type of data values that the view contains, and the block type which describes the storage of the data (e.g., dense, sparse, etc.). For example, a dense `Matrix` variable containing `float` data could be declared as

```
Matrix<float, Dense<2, float> > A(n_rows, n_cols);
```

Because `Dense` is the default storage type, this could also be declared as simply

```
Matrix<float> A(n_rows, n_cols);
```

2.2. Scalar Data Types

Sourcery VSIPL++ defines default types for scalar floating-point and integer data, in both real and complex forms:

```
typedef float scalar_f;
typedef int scalar_i;
typedef std::complex<scalar_f> cscalar_f;
typedef std::complex<scalar_i> cscalar_i;
```

Sourcery VSIPL++ also supports double-precision scalar and complex data types.

In addition, Sourcery VSIPL++ defines a number of integer types for indices, strides, and other related values.

`dimension_type`	Unsigned integer for dimension numbers (0, 1, or 2).
`index_type`	Unsigned integer for view indices.
`index_difference_type`	Signed integer for index differences.
`stride_type`	Signed integer for strides in index ranges.
`length_type`	Unsigned integer for lengths of index ranges.

2.3. Indexes and Domains

2.3.1. The Index type

An Index is a coordinate-tuple, representing a position in a View. An `Index<1>` is used for `Vector`s, an `Index<2>` for `Matrix`, and `Index<3>` for `Tensor`.

2.3.1.1. Constructors

```
template <dimension_type D> Index<D>::Index();
template <> Index<1>::Index(index_type x);
template <> Index<2>::Index(index_type y,
                            index_type x);
template <> Index<3>::Index(index_type z,
                            index_type y,
                            index_type x);
```

Description: Create an Index<D> with the given coordinates.

2.3.1.2. Accessors

```
template <dimension_type D>
index_type Index<D>::operator[](dimension_type d);
```

Description: Return the coordinate in the given dimension `d`.

2.3.2. The Domain type

A Domain is a non-empty set of non-negative indices.

2.3.2.1. Constructors

```
template <dimension_type D> Domain<D>::Domain();
template <> Domain<1>::Domain(index_type length);
```

Description: Create a Domain<1> of length `length`, starting at position `0`, with stride=1.

```
template <> Domain<1>::Domain(index_type i, stride_type s, \
length_type len);
```

Description: Create a Domain<1> of length `len`, starting at position `i`, with stride `s`.

```
template <dimension_type D> Domain<D>::Domain();
template <> Domain<2>::Domain(Domain<1> &y, Domain<1> &x);
template <> Domain<3>::Domain(Domain<1> &z, Domain<1> &y, Domain<1> \
&x);
```

Description: Create 2D and 3D domains out of 2 an 3 1D domains, respectively.

2.3.2.2. Accessors

```
template <dimension_type D>
Domain<1> &Domain<D>::operator[](dimension_type d);
```

Description: Return the 1D Domain in the given dimension.

```
index_type Domain<1>::first() const;
```

Description: Return the starting position of the domain.

```
stride_type Domain<1>::stride() const;
```

Description: Return the stride of the domain>

```
length_type Domain<1>::length() const;
```

Description: Return the length of the domain>

```
template <dimension_type D>
length_type Domain<D>::size() const;
```

Description: Return the size of the domain. For one-dimensional domains, this is the same as its length. For multi-dimensional domains, it is the product of the sizes of its (1D) constituent domains.

2.3.2.3. Arithmetic operations

```
template <dimension_type D>
Domain<D> Domain<D>::operator+(index_difference_type d);
```

Description: Increment the start index by `d`.

```
template <dimension_type D>
Domain<D> Domain<D>::operator-(index_difference_type d);
```

Description: Decrement the start index by `d`.

```
template <dimension_type D>
Domain<D> Domain<D>::operator*(stride_scalar_type s);
```

Description: Multiply the domain's stride by `s`.

```
template <dimension_type D>
Domain<D> Domain<D>::operator/(stride_scalar_type s);
```

Description: Divide the domain's stride by `s`.

2.4. Vector Objects

VSIPL++ Vector objects represent one-dimensional sets of data.

2.4.1. Vector Declarations

The type of values stored in the vector is given by the first template argument.

Examples:

`Vector<scalar_f>`	A vector of default (single-precision) floating-point values.
`Vector<double>`	A vector of double-precision floating-point values.
`Vector<cscalar_f>`	A vector of default (single-precision) complex values.
`Vector<complex<double> >`	A vector of double-precision complex values.

Optionally, the physical storage format can be controlled by the second template argument, which specifies the block type used to represent the data. If this is not specified, the default block type of `Dense` is used, which represents data stored in contiguous memory.

Example:

`Vector<float, Dense<1, float> >`	A vector of single-precision floating-point values, explicitly specified with `Dense<1, float>` storage.

Vectors can also be of type `const_Vector`, in which case their values cannot be modified directly.

2.4.2. Vector Constructors

Vectors are created by declaring an object of type `Vector`. The following constructors exist:

`Vector<float> A(size)`	Creates a vector with given size, with its values uninitialized.
`Vector<float> A(size, value)`	Creates a vector with given size, with its Values initialized to `value`.
`Vector<float> A(block)`	Creates a vector which is associated with the given data block.
`Vector<float> A(vector)`	Creates a vector which is a either a copy or an alias of the given vector, with type casts as necessary. (If `A` and `vector` have the same block type, a reference (alias) is created. Otherwise a copy is performed.)

Vectors of `const_Vector` type can only be constructed from existing Vectors with the block and vector constructor forms.

2.4.3. Vector Attributes

The following operations can be performed on a vector in order to determine various attributes.

`A.size()`	Returns the total number of elements in a vector.
`A.size(d)`	Returns the number of elements in the vector's `d`th dimension. Since vectors are one-dimensional, this is only defined for `d = 0`, and `size(0) == size()`.
`A.length()`	Equivalent to `A.size()`.
`A.block()`	Returns the underlying data-storage block for the vector.

2.4.4. Vector Elements

The following operations can be performed on a vector to read values from or write values to particular elements of the vector.

`A(n)`	Returns an lvalue reference to the `n`th value of a vector. Unless the vector is a `const_vector`, this can be used both to read values from the vector and to write values to it.
`A.get(n)`	Returns the `n`th value of a vector. This is generally more efficient than `A(n)` for retrieving the values of an elements.
`A.put(n, value)`	Sets the `n`th value of a vector to `value`. As with `A.get(n)`, this is generally more efficient than using `A(n)`.

2.4.5. Vector Subviews

VSIPL++ allows subviews of vectors to be created. A subview represents a subset of the original vector. The subview aliases the original vector, that is changes to the subview will be reflected in the original vector, and visa versa.

`A(Domain<1>(start, stride, size))`	Return a subview of `A`. The subview is of size `size`. The nth element of the subview refers to the `start + n*stride` element of `A`.
`A.get(Domain<1>(start, stride, size))`	Return a `const_Vector` subview of `A`.

2.4.6. Subview Vector Variables

The subview type of a vector type allows for the creation of variables that reference subviews of a vector. For example,

```
Vector<scalar_f>::subview_type A = view(Domain<1>(f, s, l));
```

will create a variable `A` that references the given subdomain of `view`. Thus, modifying elements of `A` will modify the corresponding elements of `view` and visa versa.

The underlying storage is reference-counted, and it will not be deallocated until all references have been destroyed. Thus, even if `view` is destroyed, the elements of `A` will continue to be valid until it is destroyed as well.

It is also possible to declare constant subview variables that cannot be modified directly, thus preventing unexpected alterations to the primary vector. These are declared with `const_subview_type`, as (for instance)

```
Vector<scalar_f>::const_subview_type A = view(Domain<1>(f, s, l));
```

2.4.7. Special Vector Subviews of Complex Vectors.

There are two additional subview functions for vectors of complex numbers:

`real(A)`	returns subview of real values in complex vector A
`imag(A)`	returns subview of imaginary values in complex vector A

Reference variables for the real and imaginary variables have the types `Vector<T>::realview_type` and `Vector<T>::imagview_type`, respectively, or `Vector<T>::const_realview_type` and `Vector<T>::const_imagview_type` for subviews that cannot be directly modified.

2.5. Matrix Objects

VSIPL++ Matrix objects represent two-dimensional sets of data. Their use is similar to Vectors.

2.6. Tensor Objects

VSIPL++ Tensor objects represent three-dimensional sets of data. Their use is similar to Vectors.

2.7. Parameter Enumerations for Signal Processing

VSIPL++ defines several enumerations to aid in the construction and use of signal processing functions and objects.

2.7.1. alg_hint_type

Use to indicate a preference on type of algorithm, if library has multiple algorithms. If library does not have multiple algorithms, preference will be ignored.

alg_time Prefer fastest algorithm.

alg_space Prefer most memory efficient algorithm.

alg_noise Prefer most accurate algorithm.

2.7.2. bias_type

Some filters, for example `Correlation`, can scale the output. Control this behavior with **`bias_type`** .

biased Do not scale.

unbiased Divide each output value by the number of input values.

2.7.3. mat_op_type

Linear equation solvers and generalized matrix products use **`mat_op_type`** to indicate the matrix operation type.

mat_ntrans Indicates the matrix should not be transposed.

mat_trans Indicates the matrix should be transposed.

mat_herm Indicates the Hermitian transpose or conjugate transpose of the matrix should be taken.

mat_conj Indicates the conjugate of the matrix should be taken.

2.7.4. obj_state

Some filters, for example `Fir`, can maintain state between invocations. Use **`obj_state`** to control this behavior.

state_no_save Do not save state between successive invocation of filter.

state_save Save state between successive invocations of filter so that output is continuous.

2.7.5. product_side_type

Linear equation solvers, specifically `chold`, `qrd` and `svd`, use **`product_side_type`** to indicate whether to use left or right multiplication in matrix products.

mat_lside Indicates `prod(A, B)` yields the product `A B`.

mat_rside Indicates `prod(A, B)` yields the product `B A`.

2.7.6. mat_uplo

The Cholesky linear equation solver `chold` uses **`mat_uplo`** to indicate which half of a symmetric or Hermitian matrix is referenced.

lower Indicates the lower LU decomposition is performed.

upper Indicates the upper LU decomposition is performed.

2.7.7. return_mechanism_type

Fast Fourier Transforms and Linear equation solvers, specifically `chold`, `lud`, `qrd` and `svd`, use **`return_mechanism_type`** to indicate the return mechanism format for matrices containing results. The former is generally easier to code, though the latter is generally faster and preferred for larger data sets.

by_value Indicates a function returns a computed value.

by_reference Indicates a function requires a parameter where the computed value is saved

2.7.8. storage_type

Linear equation solvers, specifically `qrd` and `svd`, use `storage_type` to indicate the storage format for decomposed matrices.

qrd_nosaveq The `qrd` object does not store Q.

qrd_saveq1 The `qrd` object stores `Q` using the same amount of space as the matrix given for decomposition.

qrd_saveq The `qrd` object stores `Q` using the same number of rows as the matrix given for decomposition.

svd_uvfull The `svd` object stores all of the decomposed matrix.

svd_uvnos The `svd` object does not store the decomposed matrix.

svd_uvpart Given an `N` by `M` matrix, where `p = min(M, N)`, the `svd` object stores either the first `p` columns of `U` (in the case of type `ustorage`) or the first `p` rows of V^T or V^H (in the case of type `vstorage`).

2.7.9. support_region_type

`support_region_type` describes how to handle edge conditions for convolution and correlation filter objects.

support_full Compute output wherever kernel has overlap with input support, treating values outside input as zero.

support_same Compute output with same size as input, treating values outside input as zero.

support_min Compute output only where kernel is entirely within the input support.

support_min_zeropad	Compute output only where kernel is entirely within the input support, with zero padding of output so sizes matches input.

2.7.10. symmetry_type

The convolution algorithm uses a kernel whose size is determined from the size of a view of coefficients and from an indication of its symmetry given by a member of **`symmetry_type`** .

nonsym	The kernel has the same size as the coefficient view.
sym_even_len_odd	The kernel size is one less than twice the size of the coefficient view.
sym_even_len_even	The kernel size is twice the size of the coefficient view.

Chapter 3
Overview of VSIPL++ Operations

Abstract

This chapter summarizes the functions and operations that can be applied to `Vector`, `Matrix`, and `Tensor` objects.

3.1. Functions and Function Objects

In VSIPL++, simple operators are typically defined as functions that act on view objects, and either modify their arguments or return the result as another view object. For example, the elementwise addition operator can be invoked on two vectors by

```
Vector<float> A(10, 3.0f), B(10, 5.0f), Z(10);
Z = add(A, B);
```

Some simple operations are also defined by overloading the arithmetic operators; for instance, the above addition could also have been written as

```
Z = A + B;
```

VSIPL++ also includes several operator classes. These are more complicated operators that must allocate internal working memory, or contain of a setup phase that may be reusable. These include FFTs (for which the allocation of working memory and the definition of the twiddle factors can be reused for repeated applications to views with the same dimensions) and matrix solvers (which may decompose a matrix and can then use that decomposition repeatedly with differing right-hand sides). Constructing an object of an operator class performs this setup work, and the object can then be applied to view objects in a manner analogous to a function.

For example, an object corresponding to a 1024-element FFT operator with complex `float` arguments and no scaling can be defined by

```
Fft<const_Vector, complex<float>, complex<float>, fft_fwd, by_value>
   fft_obj(1024, 1.f);
```

Once defined, this can be used (to compute Z as the Fourier transform of A) as

```
Vector<complex<float> > A(1024, 1.0f), Z(1024);
fft_obj(A, Z);
```

3.2. Elementwise Operations

VSIPL++ provides a number of operations that operate on the elements of views (Vector, Matrix, or Tensor objects) independently.

3.2.1. Assignment Operators

Two types of assignment operators are supported: assignment of a view to another view of the same size and dimensionality (that is, assignment of a Vector to a Vector, a Matrix to a Matrix, or a Tensor to a Tensor), and assignment of a scalar value to a view.

`A = B`	Assigns the values of the elements of `B` to the corresponding elements of `A`.
`A = b`	Assigns the value of the scalar `b` to all of the elements of `A`.

Note that the assignment operator always produces a copy of the values being assigned; it does not produce a reference to the same data. To create a view referencing data in an existing view, create a subview instead (see section 2.4.6).

Other assignment operators are also supported for both types of assignment. For the case of assigning a view to another view, these operators are:

`A += B`	Adds the elements of `B` to the elements of `A`.
`A -= B`	Subtracts the elements of `B` from the elements of `A`.
`A *= B`	Multiplies the elements of `A` by the elements of `B`.
`A /= B`	Divides the elements of `A` by the elements of `B`.
`A &= B`	Assigns the boolean "and" of `A` and`B` to the elements of `A`.
`A |= B`	Assigns the boolean "or" of `A` and`B` to the elements of `A`.
`A ^= B`	Assigns the boolean "exclusive or" of `A` and `B` to the elements of `A`.

The assignment operators for assigning a scalar value to a view are equivalent. Here, `A += b` adds the value of `b` to each element of `A`, and the other assignment operators are defined analogously for the scalar-valued case.

3.3. Vector Operations

VSIPL++ Vector objects represent one-dimensional sets of data. The type of values stored in the vector is given by the first template argument.

3.3.1. Vector Generation Functions

`A = 0`	Clears vector to value 0.
`A = value`	Fills vector with scalar value `value`.
`A = ramp(init, step, size)`	Fills vector with ramp function. The `n`th of element of `A` is set to `init + n * step`.

3.3.2. Vector Copy

`Z = A`	Copies value from vector `A` into vector `Z`.

3.3.3. Vector Arithmetic Elementwise Unary Operations and Functions

The following elementwise unary operations can be performed on vectors, producing a vector result:

`acos(A)`	Trigonometric arc cosine (section 4.2.1)
`arg(A)`	Phase angle of complex (section 4.2.4)
`asin(A)`	Trigonometric arc sine (section 4.2.5)
`atan(A)`	Trigonometric arc tangent (section 4.2.6)
`bnot(A)`	Boolean not (section 4.2.35)
`ceil(A)`	Round floating-point value up to next integral value (section 4.2.12)
`conj(A)`	Complex conjugate (section 4.2.13)

`cos(A)`	Trigonometric cosine (section 4.2.14)
`cosh(A)`	Hyperbolic cosine (section 4.2.15)
`euler(A)`	Rotate complex unit vector by angle (section 4.2.18)
`exp(A)`	Natural exponential (section 4.2.19)
`exp10(A)`	Base-10 exponential (section 4.2.20)
`floor(A)`	Round floating-point value down to next integral value (section 4.2.22)
`imag(A)`	Imaginary part of complex (section 4.2.27)
`is_finite(A)`	Is floating-point value finite (section 4.2.29)
`is_nan(A)`	Is floating-point value not a number (NaN) (section 4.2.29)
`is_normal(A)`	Is floating-point value normal (section 4.2.30)
`lnot(A)`	Logical not (section 4.2.35)
`log(A)`	Base-e logarithm (section 4.2.36)
`log10(A)`	Base-10 logarithm (section 4.2.37)
`mag(A)`	Magnitude (section 4.2.42)
`magsq(A)`	Magnitude squared (section 4.2.43)
`neg(A)`	Negation (section 4.2.53)
`real(A)`	Real part of complex (section 4.2.55)
`recip(A)`	Recipricol (section 4.2.56)
`rsqrt(A)`	Recipricol square root (section 4.2.57)
`sin(A)`	Trigonometric sine (section 4.2.59)
`sinh(A)`	Hyperbolic sine (section 4.2.60)
`sq(A)`	Square (section 4.2.61)
`sqrt(A)`	Square root (section 4.2.62)
`tan(A)`	Trigonometric tangent (section 4.2.64)
`tanh(A)`	Hyperbolic tangent (section 4.2.65)

3.3.4. Vector Arithmetic Elementwise Binary Operations and Functions

The following elementwise binary operations can be performed on vectors, producing a vector result:

`add(A)`	Addition (section 4.2.2)
`atan2(A)`	Arc tangent of quotient (section 4.2.7)

`band(A)`	Bitwise and (section 4.2.8)
`bor(A)`	Bitwise or (section 4.2.9)
`bxor(A)`	Bitwise exclusive or (section 4.2.11)
`div(A)`	Division (section 4.2.16)
`eq(A)`	Equality comparison (section 4.2.17)
`fmod(A)`	Floating-point modulo (remainder after division) (section 4.2.23)
`ge(A)`	Greater-than or equal comparison (section 4.2.24)
`gt(A)`	Greater-than comparison (section 4.2.25)
`hypot(A)`	Hypotenuse of right triangle (section 4.2.26)
`jmul(A)`	Conjugate multiply (section 4.2.32)
`land(A)`	Logical and (section 4.2.33)
`le(A)`	Less-than or equal comparison (section 4.2.34)
`lor(A)`	Logical or (section 4.2.38)
`lt(A)`	Less-than comparison (section 4.2.39)
`lxor(A)`	Logical exclusive or (section 4.2.40)
`max(A)`	Maxima (section 4.2.44)
`maxmg(A)`	Magnitude maxima (section 4.2.45)
`maxmgsq(A)`	Magnitude squared maxima (section 4.2.46)
`min(A)`	Minima (section 4.2.47)
`minmg(A)`	Magnitude minima (section 4.2.48)
`minmgsq(A)`	Magnitude squared minima (section 4.2.49)
`mul(A)`	Multiplication (section 4.2.51)
`ne(A)`	Not equal comparison (section 4.2.52)
`pow(A)`	Raise to power (section 4.2.54)
`sub(A)`	Subtract (section 4.2.63)

3.3.5. Vector Arithmetic Elementwise Ternary Operations and Functions

The following elementwise ternary operations can be performed on vectors, producing a vector result:

`am(A)`	Fused addition-multiplication (section 4.2.3)
`expoavg(A)`	Exponential average (section 4.2.21)

`ite(A)`	Addition (section 4.2.31)
`ma(A)`	Fused multiplication-addition (section 4.2.41)
`msb(A)`	Fused multiplication-subtraction (section 4.2.50)
`sbm(A)`	Fused subtraction-multiplication (section 4.2.58)

3.3.6. Vector Type Conversions

A vector with one type of values can be converted a vector with another type of values using `view_cast`.

For example, to convert a vector of floats `A` into a vector of ints `Z`:

```
Vector<float> A(size);
Vector<int>   Z(size);
Z = view_cast<int>(A)
```

3.3.7. Vector Arithmetic Elementwise Binary Operations and Functions

The following arithmetic elementwise binary operations and functions are available on vectors:

`Z = add(A, B)`	Addition, `Z(n) = A(n) + B(n)`(section 4.2.2)
`Z = div(A, B)`	Division, `Z(n) = A(n) / B(n)`(section 4.2.16)
`Z = max(A, B)`	Maximum of `A(n)` and `B(n)`
`Z = min(A, B)`	Minimum of `A(n)` and `B(n)`
`Z = mul(A, B)`	Multiplication, `Z(n) = A(n) * B(n)`(section 4.2.51)
`Z = sub(A, B)`	Subtraction, `Z(n) = A(n) - B(n)`(section 4.2.63)

Addition, subtraction, multiplication, and division can also be written in operator form:

`Z = A + B`	equivalent to `Z = add(A, B)`
`Z = A - B`	equivalent to `Z = sub(A, B)`
`Z = A * B`	equivalent to `Z = mul(A, B)`
`Z = A / B`	equivalent to `Z = div(A, B)`

In all the preceding functions and operations, either of the vector operands can be replaced with scalar operands.

For example, to perform a scalar-vector multiply:

```
Z = a * B;
```

or

```
Z = mul(a, B);
```

3.3.8. Vector Logical Elementwise Binary Operations and Functions

`Z = eq(A, B)`	`Z(n) = A(n) == B(n)`
`Z = gt(A, B)`	`Z(n) = A(n) > B(n)`
`Z = gte(A, B)`	`Z(n) = A(n) >= B(n)`
`Z = lt(A, B)`	`Z(n) = A(n) < B(n)`
`Z = lte(A, B)`	`Z(n) = A(n) <= B(n)`
`Z = ne(A, B)`	`Z(n) = A(n) != B(n)`

3.3.9. Vector Arithmetic Elementwise Ternary Operations and Functions

The following arithmetic elementwise ternary operations and functions are available on vectors:

`Z = ma(A, B, C)`	Multiply-add, `Z(n) = A(n) * B(n) + C(n)`
`Z = am(A, B, C)`	Add-multiply, `Z(n) = A(n) + B(n) * C(n)`

In all the preceding functions and operations, one or more of the vector operands can be replaced with scalar operands.

For example, to scale a vector, then apply an offset:

```
Z = scale * A + offset;
```

or

```
Z = ma(scale, A, offset);
```

(where `scale` and `offset` are scalar values)

3.3.10. Vector Non-Arithmetic Elementwise Ternary Operations and Functions

`Z = ite(bool_vector, A, B)`	For the nth element of `Z`, sets value to nth element of`A` if nth element of bool_vector is true, otherwise sets value to the nth element of `B`.Notionally equivalent to C `?:` operator.`(Foreach n) Z[n] = bool_vector[n] ? A[n] : B[n]`

In all the preceding functions and operations, one or more of the vector operands can be replaced with scalar operands.

For example, the apply a scalar threshold `b` to a vector:

```
Z = ite(A > b, A, b);
```

3.3.11. Vector Reductions

The following functions reduce a vector to a single value:

`z = alltrue(A)`	When the element type of `A` is `bool`, the function returns `true` if all the elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.1) for more information.
`z = anytrue(A)`	When the element type of `A` is `bool`, the function returns `true` if any elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.2) for more information.
`z = sumval(A)`	Return the sum of A's values.
`z = sumsqval(A)`	Return the sum of the squares of A's values.
`z = meanval(A)`	Return the mean of A's values.
`z = meansqval(A)`	Return the mean of squares of A's values.

The following functions reduce a vector to a single value that corresponds to an element within the vector.

`z = maxval(A, idx)`	return the maximum value of A. Set `idx` to the index of this element (`A.get(idx) == z`).
`z = maxmgval(A, idx)`	return the maximum value of the magnitude of A. Set `idx` to the index of this element (`mag(A.get(idx)) == z`).
`z = maxmgsqval(A, idx)`	return the maximum value of the magnitude squared of A. Set `idx` to the index of this element (`magsq(A.get(idx)) == z`).
`z = minval(A, idx)`	return the minimum value of A. Set `idx` to the index of this element (`A.get(idx) == z`).
`z = minmgval(A, idx)`	return the minimum value of the magnitude of A. Set `idx` to the index of this element (`mag(A.get(idx)) == z`).
`z = minmgsqval(A, idx)`	return the minimum value of the magnitude squared of A. Set `idx` to the index of this element (`magsq(A.get(idx)) == z`).

3.3.12. Vector Linear Algebra

`z = cvjdot(A, B)`	conjugate dot-product (section 4.4.2)
`Z = kron(A, B)`	kronecker-product (section 4.4.7)
`z = dot(A, B)`	dot-product (section 4.4.3)
`Z = outer(A, B)`	outer-product (section 4.4.9)

3.3.13. Vector Window Functions

`Z = blackman(len)`	Construct and return a vector containing a Blackman window of length `len`. (section 4.7.1)

`Z = cheby(len, ripple)`	Construct and return a vector containing Dolph-Chebyshev window weights with user-specified `ripple` and having length `len`. (section 4.7.2)
`Z = hanning(len)`	Construct and return a vector containing a Hanning window of length `len`. (section 4.7.4)
`Z = kaiser(len, beta)`	Construct and return a vector containing Kaiser window weights with transition width `beta` and having length `len`. (section 4.7.5)

3.3.14. Vector Convolution

VSIPL++ provides facilities to perform convolutions on vectors through the class template `Convolution` (section 4.8.1). Once constructed with compile-time selections, a convolution object can be applied to an input vector to produce results in an output vector.

3.3.15. Vector Correlation

VSIPL++ provides facilities to perform correlations on vectors through the class template `Correlation` (section 4.8.2). Once constructed with compile-time selections, a correlation object can be applied to an input vector and a kernel vector to produce results in an output vector.

3.3.16. Vector FIR Filter

VSIPL++ provides facilities to perform an FIR filter on vectors through the class template `Fir` (section 4.8.3). Once constructed with compile-time selections, an FIR object can be applied to an input vector to produce results in an output vector.

3.3.17. Vector Histogram

VSIPL++ provides facilities to perform an histogram on vectors through the class template `Histogram`. Once constructed with compile-time selections, a histogram object can be applied to an input vector to produce results in an output vector.

3.3.18. Vector Random Number Generation

VSIPL++ provides facilities to generate vectors of random number through the class template `Rand`. Once constructed with compile-time selections, a `Rand` object can generate a vector of random numbers through the invocation of one of its member functions..

3.3.19. Vector Frequency Swap

VSIPL++ provides a function `freqswap` to swap halves of vectors. (section 4.7.3)

3.3.20. Vector Subviews

VSIPL++ allows subviews of vectors to be created. A subview represents a subset of the original vector. The subview aliases the original vector, that is changes to the subview will be reflected in the original vector, and visa versa.

`A(Domain<1>(start, stride, size))`	Create subview of `vector`. Subview is of size `size`. The nth element of the subview refers to the `start + n*stride` element of `A`.

3.3.21. Subview Vector Variables

To represent a vector subview in a variable, it is necessary to use the correct block type. Otherwise the variable will copy the values in the subview.

```
Vector<T>::subview_type A = view(Domain<1>(f, s, l));
```

3.3.22. Special Vector Subviews of Complex Vectors.

`real(A)`	returns subview of real values in complex vector A
`imag(A)`	returns subview of real values in complex vector A

3.3.23. User-Defined Functions on Vectors

User-defined functions that accept a vector as a parameter should use a template parameter to represent the vector's block type. This allows vectors with different block types, such as those created by subview operators, to be handled by the function.

For example, to write a function that accepts a vector of floating-point values:

```
template <typename BlockT>
...
function(Vector<float, BlockT> vector)
{
   ...
}
```

If the function can handle different value types (such as single- and double-precision), the value type can also be made a template parameter:

```
template <typename, T
          typename BlockT>
...
function(Vector<T, BlockT> vector)
{
   ...
}
```

3.4. Matrix Operations

VSIPL++ Matrix objects represent two-dimensional sets of data. The type of values stored in the matrix is given by the first template argument.

3.4.1. Matrix Generation Functions

`A = 0`	Clears matrix to value 0.
`A = value`	Fills matrix with scalar value `value`.

3.4.2. Matrix Copy

`Z = A`	Copies value from matrix `A` into matrix `Z`.

3.4.3. Matrix Arithmetic Elementwise Unary Operations and Functions

The following elementwise unary operations can be performed on matrices, producing a matrix result:

`acos(A)`	Trigonometric arc cosine (section 4.2.1)
`arg(A)`	Phase angle of complex (section 4.2.4)
`asin(A)`	Trigonometric arc sine (section 4.2.5)
`atan(A)`	Trigonometric arc tangent (section 4.2.6)
`bnot(A)`	Boolean not (section 4.2.35)
`ceil(A)`	Round floating-point value up to next integral value (section 4.2.12)
`conj(A)`	Complex conjugate (section 4.2.13)
`cos(A)`	Trigonometric cosine (section 4.2.14)
`cosh(A)`	Hyperbolic cosine (section 4.2.15)
`euler(A)`	Rotate complex unit vector by angle (section 4.2.18)
`exp(A)`	Natural exponential (section 4.2.19)
`exp10(A)`	Base-10 exponential (section 4.2.20)
`floor(A)`	Round floating-point value down to next integral value (section 4.2.22)
`imag(A)`	Imaginary part of complex (section 4.2.27)
`is_finite(A)`	Is floating-point value finite (section 4.2.29)
`is_nan(A)`	Is floating-point value not a number (NaN) (section 4.2.29)
`is_normal(A)`	Is floating-point value normal (section 4.2.30)
`lnot(A)`	Logical not (section 4.2.35)
`log(A)`	Base-e logarithm (section 4.2.36)
`log10(A)`	Base-10 logarithm (section 4.2.37)
`mag(A)`	Magnitude (section 4.2.42)
`magsq(A)`	Magnitude squared (section 4.2.43)
`neg(A)`	Negation (section 4.2.53)
`real(A)`	Real part of complex (section 4.2.55)

`recip(A)`	Recipricol (section 4.2.56)
`rsqrt(A)`	Recipricol square root (section 4.2.57)
`sin(A)`	Trigonometric sine (section 4.2.59)
`sinh(A)`	Hyperbolic sine (section 4.2.60)
`sq(A)`	Square (section 4.2.61)
`sqrt(A)`	Square root (section 4.2.62)
`tan(A)`	Trigonometric tangent (section 4.2.64)
`tanh(A)`	Hyperbolic tangent (section 4.2.65)

3.4.4. Matrix Arithmetic Elementwise Binary Operations and Functions

The following elementwise binary operations can be performed on matrices, producing a matrix result:

`add(A)`	Addition (section 4.2.2)
`atan2(A)`	Arc tangent of quotient (section 4.2.7)
`band(A)`	Bitwise and (section 4.2.8)
`bor(A)`	Bitwise or (section 4.2.9)
`bxor(A)`	Bitwise exclusive or (section 4.2.11)
`div(A)`	Division (section 4.2.16)
`eq(A)`	Equality comparison(section 4.2.17)
`fmod(A)`	Floating-point modulo (remainder after division) (section 4.2.23)
`ge(A)`	Greater-than or equal comparison (section 4.2.24)
`gt(A)`	Greater-than comparison (section 4.2.25)
`hypot(A)`	Hypotenuse of right triangle (section 4.2.26)
`jmul(A)`	Conjugate multiply (section 4.2.32)
`land(A)`	Logical and (section 4.2.33)
`le(A)`	Less-than or equal comparison (section 4.2.34)
`lor(A)`	Logical or (section 4.2.38)
`lt(A)`	Less-than comparison (section 4.2.39)
`lxor(A)`	Logical exclusive or (section 4.2.40)
`max(A)`	Maxima (section 4.2.44)

`maxmg(A)`	Magnitude maxima (section 4.2.45)
`maxmgsq(A)`	Magnitude squared maxima (section 4.2.46)
`min(A)`	Minima (section 4.2.47)
`minmg(A)`	Magnitude minima (section 4.2.48)
`minmgsq(A)`	Magnitude squared minima (section 4.2.49)
`mul(A)`	Multiplication (section 4.2.51)
`ne(A)`	Not equal comparison (section 4.2.52)
`pow(A)`	Raise to power (section 4.2.54)
`sub(A)`	Subtract (section 4.2.63)

3.4.5. Matrix Arithmetic Elementwise Ternary Operations and Functions

The following elementwise ternary operations can be performed on matrices, producing a matrix result:

`am(A)`	Fused addition-multiplication (section 4.2.3)
`expoavg(A)`	Exponential average (section 4.2.21)
`ite(A)`	Addition (section 4.2.31)
`ma(A)`	Fused multiplication-addition (section 4.2.41)
`msb(A)`	Fused multiplication-subtraction (section 4.2.50)
`sbm(A)`	Fused subtraction-multiplication (section 4.2.58)

3.4.6. Matrix Type Conversions

A matrix with one type of values can be converted a matrix with another type of values using `view_cast`.

For example, to convert a matrix of floats `A` into a matrix of ints `Z`:

```
Matrix<float> A(size);
Matrix<int>   Z(size);
Z = view_cast<int>(A)
```

3.4.7. Matrix Arithmetic Elementwise Binary Operations and Functions

The following arithmetic elementwise binary operations and functions are available on matrices:

`Z = add(A, B)`	Addition, `Z(n) = A(n) + B(n)`(section 4.2.2)
`Z = div(A, B)`	Division, `Z(n) = A(n) / B(n)`(section 4.2.16)
`Z = max(A, B)`	Maximum of `A(n)` and `B(n)`
`Z = min(A, B)`	Minimum of `A(n)` and `B(n)`

`Z = mul(A, B)` Multiplication, `Z(n) = A(n) * B(n)` (section 4.2.51)

`Z = sub(A, B)` Subtraction, `Z(n) = A(n) - B(n)` (section 4.2.63)

Addition, subtraction, multiplication, and division can also be written in operator form:

`Z = A + B` equivalent to `Z = add(A, B)`

`Z = A - B` equivalent to `Z = sub(A, B)`

`Z = A * B` equivalent to `Z = mul(A, B)`

`Z = A / B` equivalent to `Z = div(A, B)`

In all the preceding functions and operations, either of the vector operands can be replaced with scalar operands.

For example, to perform a scalar-vector multiply:

```
Z = a * B;
```

or

```
Z = mul(a, B);
```

3.4.8. Matrix Logical Elementwise Binary Operations and Functions

`Z = eq(A, B)` `Z(n) = A(n) == B(n)`

`Z = gt(A, B)` `Z(n) = A(n) > B(n)`

`Z = gte(A, B)` `Z(n) = A(n) >= B(n)`

`Z = lt(A, B)` `Z(n) = A(n) < B(n)`

`Z = lte(A, B)` `Z(n) = A(n) <= B(n)`

`Z = ne(A, B)` `Z(n) = A(n) != B(n)`

3.4.9. Matrix Arithmetic Elementwise Ternary Operations and Functions

The following arithmetic elementwise ternary operations and functions are available on matrices:

`Z = ma(A, B, C)` Multiply-add, `Z(n) = A(n) * B(n) + C(n)`

`Z = am(A, B, C)` Add-multiply, `Z(n) = A(n) + B(n) * C(n)`

In all the preceding functions and operations, one or more of the matrix operands can be replaced with scalar operands.

For example, to scale a matrix, then apply an offset:

```
Z = scale * A + offset;
```

or

```
Z = ma(scale, A, offset);
```

(where `scale` and `offset` are scalar values)

3.4.10. Matrix Non-Arithmetic Elementwise Ternary Operations and Functions

`Z = ite(bool_vector, A, B)`	For the m, nth element of `Z`, sets value to m, nth element of`A` if m, nth element of bool_vector is true, otherwise sets value to the m, nth element of `B`. Notionally equivalent to C `?:` operator. `(Foreach n) Z[n] = bool_vector[n] ? A[n] : B[n]`

In all the preceding functions and operations, one or more of the matrix operands can be replaced with scalar operands.

For example, the apply a scalar threshold `b` to a matrix:

```
Z = ite(A > b, A, b);
```

3.4.11. Matrix Reductions

The following functions reduce a matrix to a single value:

`z = alltrue(A)`	When the element type of `A` is `bool`, the function returns `true` if all the elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.1) for more information.
`z = anytrue(A)`	When the element type of `A` is `bool`, the function returns `true` if any elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.2) for more information.
`z = sumval(A)`	Return the sum of `A`'s values.
`z = sumsqval(A)`	Return the sum of the squares of `A`'s values.
`z = meanval(A)`	Return the mean of A's values.
`z = meansqval(A)`	Return the mean of the squares of `A`'s values.

The following functions reduce a matrix to a single value that corresponds to an element within the matrix.

`z = maxval(A, idx)`	return the maximum value of `A`. Set `idx` to the index of this element (`A.get(idx[0], idx[1]) == z`).
`z = maxmgval(A, idx)`	return the maximum value of the magnitude of `A`. Set `idx` to the index of this element (`mag(A.get(idx[0], idx[1])) == z`).
`z = maxmgsqval(A, idx)`	return the maximum value of the magnitude squared of `A`. Set `idx` to the index of this element (`magsq(A.get(idx[0], idx[1])) == z`).
`z = minval(A, idx)`	return the minimum value of `A`. Set `idx` to the index of this element (`A.get(idx) == z`).

`z = minmgval(A, idx)`	return the minimum value of the magnitude of `A`. Set `idx` to the index of this element (`mag(A.get(idx[0], idx[1])) == z`).
`z = minmgsqval(A, idx)`	return the minimum value of the magnitude squared of `A`. Set `idx` to the index of this element (`magsq(A.get(idx[0], idx[1])) == z`).

3.4.12. Matrix Linear Algebra

`Z = conj(A)`	conjugate (section 4.2.13)
`Z = herm(A)`	hermetian (conjugate-transpose) (section 4.4.6)
`Z = prod(A, B)`	product (section 4.4.10)
`Z = prod3(A, B)`	3x3 product (section 4.4.11)
`Z = prod4(A, B)`	4x4 product (section 4.4.12)
`Z = prodh(A, B)`	hermetian product (section 4.4.13)
`Z = prodj(A, B)`	conjugate product (section 4.4.14)
`Z = prodt(A, B)`	transpose product (section 4.4.15)
`Z = trans(A)`	transpose (section 4.4.16)

3.4.13. Matrix Convolution

VSIPL++ provides facilities to perform convolutions on matrices through the class template `Convolution`. Once constructed with compile-time selections, a convolution object can be applied to an input matrix to produce results in an output matrix.

3.4.14. Matrix Correlation

VSIPL++ provides facilities to perform correlations on matrices through the class template `Correlation`. Once constructed with compile-time selections, a correlation object can be applied to an input matrix and a kernel matrix. to produce results in an output matrix.

3.4.15. Matrix Histogram

VSIPL++ provides facilities to perform an histogram on matrices through the class template `Histogram`. Once constructed with compile-time selections, a histogram object can be applied to an input matrix to produce results in an output vector.

3.4.16. Matrix Random Number Generation

VSIPL++ provides facilities to generate matrices of random number through the class template `Rand`. Once constructed with compile-time selections, a `Rand` object can generate a matrix of random numbers through the invocation of one of its member functions.

3.4.17. Matrix Frequency Swap

VSIPL++ provides a function `freqswap` to swap quadrants of matrices. (section 4.7.3)

3.5. Tensor Operations

VSIPL++ Tensor objects represent three-dimensional sets of data. The type of values stored in the tensor is given by the first template argument.

3.5.1. Tensor Generation Functions

`A = 0`	Clears tensor to value 0.
`A = value`	Fills tensor with scalar value `value`.

3.5.2. Tensor Copy

`Z = A`	Copies value from tensor `A` into tensor `Z`.

3.5.3. Tensor Arithmetic Elementwise Unary Operations and Functions

The following elementwise unary operations can be performed on tensors, producing a tensor result:

`acos(A)`	Trigonometric arc cosine (section 4.2.1)
`arg(A)`	Phase angle of complex (section 4.2.4)
`asin(A)`	Trigonometric arc sine (section 4.2.5)
`atan(A)`	Trigonometric arc tangent (section 4.2.6)
`bnot(A)`	Boolean not (section 4.2.35)
`ceil(A)`	Round floating-point value up to next integral value (section 4.2.12)
`conj(A)`	Complex conjugate (section 4.2.13)
`cos(A)`	Trigonometric cosine (section 4.2.14)
`cosh(A)`	Hyperbolic cosine (section 4.2.15)
`euler(A)`	Rotate complex unit vector by angle (section 4.2.18)
`exp(A)`	Natural exponential (section 4.2.19)
`exp10(A)`	Base-10 exponential (section 4.2.20)
`floor(A)`	Round floating-point value down to next integral value (section 4.2.22)
`imag(A)`	Imaginary part of complex (section 4.2.27)
`is_finite(A)`	Is floating-point value finite (section 4.2.29)
`is_nan(A)`	Is floating-point value not a number (NaN) (section 4.2.29)
`is_normal(A)`	Is floating-point value normal (section 4.2.30)
`lnot(A)`	Logical not (section 4.2.35)

`log(A)`	Base-e logarithm (section 4.2.36)
`log10(A)`	Base-10 logarithm (section 4.2.37)
`mag(A)`	Magnitude (section 4.2.42)
`magsq(A)`	Magnitude squared (section 4.2.43)
`neg(A)`	Negation (section 4.2.53)
`real(A)`	Real part of complex (section 4.2.55)
`recip(A)`	Recipricol (section 4.2.56)
`rsqrt(A)`	Recipricol square root (section 4.2.57)
`sin(A)`	Trigonometric sine (section 4.2.59)
`sinh(A)`	Hyperbolic sine (section 4.2.60)
`sq(A)`	Square (section 4.2.61)
`sqrt(A)`	Square root (section 4.2.62)
`tan(A)`	Trigonometric tangent (section 4.2.64)
`tanh(A)`	Hyperbolic tangent (section 4.2.65)

3.5.4. Tensor Arithmetic Elementwise Binary Operations and Functions

The following elementwise binary operations can be performed on tensors, producing a tensor result:

`add(A)`	Addition (section 4.2.2)
`atan2(A)`	Arc tangent of quotient (section 4.2.7)
`band(A)`	Bitwise and (section 4.2.8)
`bor(A)`	Bitwise or (section 4.2.9)
`bxor(A)`	Bitwise exclusive or (section 4.2.11)
`div(A)`	Division (section 4.2.16)
`eq(A)`	Equality comparison (section 4.2.17)
`fmod(A)`	Floating-point modulo (remainder after division) (section 4.2.23)
`ge(A)`	Greater-than or equal comparison (section 4.2.24)
`gt(A)`	Greater-than comparison (section 4.2.25)
`hypot(A)`	Hypotenuse of right triangle (section 4.2.26)
`jmul(A)`	Conjugate multiply (section 4.2.32)
`land(A)`	Logical and (section 4.2.33)

`le(A)`	Less-than or equal comparison (section 4.2.34)
`lor(A)`	Logical or (section 4.2.38)
`lt(A)`	Less-than comparison (section 4.2.39)
`lxor(A)`	Logical exclusive or (section 4.2.40)
`max(A)`	Maxima (section 4.2.44)
`maxmg(A)`	Magnitude maxima (section 4.2.45)
`maxmgsq(A)`	Magnitude squared maxima (section 4.2.46)
`min(A)`	Minima (section 4.2.47)
`minmg(A)`	Magnitude minima (section 4.2.48)
`minmgsq(A)`	Magnitude squared minima (section 4.2.49)
`mul(A)`	Multiplication (section 4.2.51)
`ne(A)`	Not equal comparison (section 4.2.52)
`pow(A)`	Raise to power (section 4.2.54)
`sub(A)`	Subtract (section 4.2.63)

3.5.5. Tensor Arithmetic Elementwise Ternary Operations and Functions

The following elementwise ternary operations can be performed on tensors, producing a tensor result:

`am(A)`	Fused addition-multiplication (section 4.2.3)
`expoavg(A)`	Exponential average (section 4.2.21)
`ite(A)`	Addition (section 4.2.31)
`ma(A)`	Fused multiplication-addition (section 4.2.41)
`msb(A)`	Fused multiplication-subtraction (section 4.2.50)
`sbm(A)`	Fused subtraction-multiplication (section 4.2.58)

3.5.6. Tensor Type Conversions

A tensor with one type of values can be converted a tensor with another type of values using `view_cast`.

For example, to convert a tensor of floats `A` into a tensor of ints `Z`:

```
Tensor<float> A(size);
Tensor<int>   Z(size);
Z = view_cast<int>(A)
```

3.5.7. Tensor Arithmetic Elementwise Binary Operations and Functions

The following arithmetic elementwise binary operations and functions are available on tensors:

`Z = add(A, B)`	Addition, `Z(n) = A(n) + B(n)` (section 4.2.2)
`Z = div(A, B)`	Division, `Z(n) = A(n) / B(n)` (section 4.2.16)
`Z = max(A, B)`	Maximum of `A(n)` and `B(n)`
`Z = min(A, B)`	Minimum of `A(n)` and `B(n)`
`Z = mul(A, B)`	Multiplication, `Z(n) = A(n) * B(n)` (section 4.2.51)
`Z = sub(A, B)`	Subtraction, `Z(n) = A(n) - B(n)` (section 4.2.63)

Addition, subtraction, multiplication, and division can also be written in operator form:

`Z = A + B`	equivalent to `Z = add(A, B)`
`Z = A - B`	equivalent to `Z = sub(A, B)`
`Z = A * B`	equivalent to `Z = mul(A, B)`
`Z = A / B`	equivalent to `Z = div(A, B)`

In all the preceding functions and operations, either of the tensor operands can be replaced with scalar operands.

For example, to perform a scalar-tensor multiply:

```
Z = a * B;
```

or

```
Z = mul(a, B);
```

3.5.8. Tensor Logical Elementwise Binary Operations and Functions

`Z = eq(A, B)`	`Z(n) = A(n) == B(n)`
`Z = gt(A, B)`	`Z(n) = A(n) > B(n)`
`Z = gte(A, B)`	`Z(n) = A(n) >= B(n)`
`Z = lt(A, B)`	`Z(n) = A(n) < B(n)`
`Z = lte(A, B)`	`Z(n) = A(n) <= B(n)`
`Z = ne(A, B)`	`Z(n) = A(n) != B(n)`

3.5.9. Tensor Arithmetic Elementwise Ternary Operations and Functions

The following arithmetic elementwise ternary operations and functions are available on tensors:

`Z = ma(A, B, C)`	Multiply-add, `Z(n) = A(n) * B(n) + C(n)`
`Z = am(A, B, C)`	Add-multiply, `Z(n) = A(n) + B(n) * C(n)`

In all the preceding functions and operations, one or more of the tensor operands can be replaced with scalar operands.

For example, to scale a tensor, then apply an offset:

```
Z = scale * A + offset;
```

or

```
Z = ma(scale, A, offset);
```

(where `scale` and `offset` are scalar values)

3.5.10. Tensor Non-Arithmetic Elementwise Ternary Operations and Functions

`Z = ite(bool_tensor, A, B)`	For the m,n,pth element of `Z`, sets value to nth element of`A` if m,n,pth element of bool_tensor is true, otherwise sets value to the m,n,pth element of `B`. Notionally equivalent to C `?:` operator. `(Foreach n) Z[m, n, p] = bool_tensor[m, n, p] ? A[m, n, p] : B[m, n, p]`

In all the preceding functions and operations, one or more of the tensor operands can be replaced with scalar operands.

For example, the apply a scalar threshold `b` to a tensor:

```
Z = ite(A > b, A, b);
```

3.5.11. Tensor Reductions

The following functions reduce a tensor to a single value:

`z = alltrue(A)`	When the element type of `A` is `bool`, the function returns `true` if all the elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.1) for more information.
`z = anytrue(A)`	When the element type of `A` is `bool`, the function returns `true` if any elements are `true`; otherwise `false`. When the element type is something else, see (section 4.3.2) for more information.
`z = sumval(A)`	Return the sum of `A`'s values.
`z = sumsqval(A)`	Return the sum of the squares of `A`'s values.
`z = meanval(A)`	Return the mean of `A`'s values.
`z = meansqval(A)`	Return the mean of the squares of `A`'s values.

The following functions reduce a tensor to a single value that corresponds to an element within the vector.

`z = maxval(A, idx)`	return the maximum value of `A`. Set `idx` to the index of this element (`A.get(idx[0], idx[1], idx[2]) == z`).

`z = maxmgval(A, idx)`	return the maximum value of the magnitude of `A`. Set `idx` to the index of this element (`mag(A.get(idx[0], idx[1], idx[2])) == z`).
`z = maxmgsqval(A, idx)`	return the maximum value of the magnitude squared of `A`. Set `idx` to the index of this element (`magsq(A.get(idx[0], idx[1], idx[2])) == z`).
`z = minval(A, idx)`	return the minimum value of `A`. Set `idx` to the index of this element (`A.get(idx[0], idx[1], idx[2]) == z`).
`z = minmgval(A, idx)`	return the minimum value of the magnitude of `A`. Set `idx` to the index of this element (`mag(A.get(idx[0], idx[1], idx[2])) == z`).
`z = minmgsqval(A, idx)`	return the minimum value of the magnitude squared of `A`. Set `idx` to the index of this element (`magsq(A.get(idx[0], idx[1], idx[2])) == z`).

3.5.12. Tensor Random Number Generation

VSIPL++ provides facilities to generate tensors of random number through the class template `Rand`. Once constructed with compile-time selections, a `Rand` object can generate a tensor of random numbers through the invocation of one of its member functions.

Chapter 4
Function Reference

Abstract

This chapter contains detailed references for all of the VSIPL++ functions that can be applied to vectors, matrices, and tensors.

4.1. Introduction

The following man pages describe the operation of each VSIPL++ function and class.

4.2. Elementwise Functions

4.2.1. acos

Description: Elementwise arc cosine (inverse cosine).

Syntax:

```
Vector<T> acos (  Vector<T> A  );

Matrix<T> acos (  Matrix<T> A  );

Tensor<T> acos (  Tensor<T> A  );
```

Result: Each element of the result view is set to arc or inverse cosine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = acos(A)` produces a result equivalent to `Z(i) = acos(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = acos(A);
```

See Also: `asin` (section 4.2.5) `atan` (section 4.2.6) `cos` (section 4.2.14) `sin` (section 4.2.59) `tan` (section 4.2.64)

4.2.2. add

Description: Elementwise addition.

Syntax:

```
Vector<T> add (  Vector<T> A ,  Vector<T> B  );

Vector<T> add (  T a ,  Vector<T> B  );

Vector<T> add (  Vector<T> A ,  T b  );

Matrix<T> add (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> add (  T a ,  Matrix<T> B  );

Matrix<T> add (  Matrix<T> A ,  T b  );

Tensor<T> add (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> add (  T a ,  Tensor<T> B  );

Tensor<T> add (  Tensor<T> A ,  T b  );
```

Operator Syntax: Addition can also be written in operator form. `add(A, B)` is equivalent to `A + B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the sum of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = add(A, B)` produces a result equivalent to `Z(i) = A(i) + B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is added to all of the elements of the other argument; for example, `Z = add(A, b)` produces `Z(i) = A(i) + b`.

Example:

```
Vector<float> Z, A, B;
Z = add(A, B);
```

See Also: `div` (section 4.2.16) `mul` (section 4.2.51) `sub` (section 4.2.63)

4.2.3. am

Description: Elementwise addition-multiplication.

Syntax:

```
Vector<T> am (  Vector<T> A ,  Vector<T> B ,  Vector<T> C  );

Matrix<T> am (  Matrix<T> A ,  Matrix<T> B ,  Matrix<T> C  );

Tensor<T> am (  Tensor<T> A ,  Tensor<T> B ,  Tensor<T> C  );
```

Operator Syntax: Addition can also be written in operator form. `am(A, B, C)` is equivalent to `(A + B) * C`.

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the sum-product of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = am(A, B, C)` produces a result equivalent to `Z(i) = (A(i) + B(i)) * C(i)` for all of the elements of the vector. If any of the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = am(A, b, C)` produces `Z(i) = (A(i) + b) * C`.

Example:

```
Vector<float> Z, A, B, C;
Z = am(A, B, C);
```

See Also: `ma` (section 4.2.41) `msb` (section 4.2.50) `sbm` (section 4.2.58)

4.2.4. arg

Description: Elementwise phase angle of complex.

Syntax:

```
Vector<T> arg (  Vector<complex<T> > A  );

Vector<T> arg (  Matrix<complex<T> > A  );
```

```
Vector<T> arg (  Tensor<complex<T> > A  );
```

Result: Each element of the result view is set to the phase angle of the corresponding element of the argument. For instance, if the argument is a vector, `Z = arg(A)` produces a result equivalent to `Z(i) = atan2(imag(A(i)), real(A(i)))` for all the elements of the vector.

Example:

```
Vector<complex<float> > A;
Vector<float> Z;
Z = arg(A);
```

See Also: `imag` (section 4.2.27) `real` (section 4.2.55)

4.2.5. `asin`

Description: Elementwise arc sine (inverse sine).

Syntax:

```
Vector<T> asin (  Vector<T> A  );

Matrix<T> asin (  Matrix<T> A  );

Tensor<T> asin (  Tensor<T> A  );
```

Result: Each element of the result view is set to arc or inverse sine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = asin(A)` produces a result equivalent to `Z(i) = asin(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = asin(A);
```

See Also: `acos` (section 4.2.1) `atan` (section 4.2.6) `cos` (section 4.2.14) `sin` (section 4.2.59) `tan` (section 4.2.64)

4.2.6. `atan`

Description: Elementwise arc tangent (inverse tangent).

Syntax:

```
Vector<T> atan (  Vector<T> A  );

Matrix<T> atan (  Matrix<T> A  );

Tensor<T> atan (  Tensor<T> A  );
```

Result: Each element of the result view is set to arc or inverse tangent of the corresponding element of the argument. For instance, if the argument is a vector, `Z = atan(A)` produces a result equivalent to `Z(i) = atan(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = atan(A);
```

See Also: `acos` (section 4.2.1) `asin` (section 4.2.5) `cos` (section 4.2.14) `sin` (section 4.2.59) `tan` (section 4.2.64)

4.2.7. atan2

Description: Elementwise arc tangent of a quotient.

Syntax:

```
Vector<T> atan2 ( Vector<T> A , Vector<T> B );

Vector<T> atan2 ( T a , Vector<T> B );

Vector<T> atan2 ( Vector<T> A , T b );

Matrix<T> atan2 ( Matrix<T> A , Matrix<T> B );

Matrix<T> atan2 ( T a , Matrix<T> B );

Matrix<T> atan2 ( Matrix<T> A , T b );

Tensor<T> atan2 ( Tensor<T> A , Tensor<T> B );

Tensor<T> atan2 ( T a , Tensor<T> B );

Tensor<T> atan2 ( Tensor<T> A , T b );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the arc tangent of the quotient of the elements of the two arguments. For instance, if the arguments are vectors, `Z = atan2(A, B)` produces a result equivalent to `Z(i) = atan2(A(i), B(i)) = atan(A(i) / B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is used as part of the quotient for all of the elements of the other argument; for example, `Z = atan2(A, b)` produces `Z(i) = atan2(A(i), b)`.

Example:

```
Vector<float> Z, A, B;
Z = atan2(A, B);
```

See Also: `atan` (section 4.2.6) `tan` (section 4.2.64)

4.2.8. band

Description: Elementwise bitwise and.

Syntax:

```
Vector<T> band ( Vector<T> A , Vector<T> B );

Vector<T> band ( T a , Vector<T> B );

Vector<T> band ( Vector<T> A , T b );
```

```
Matrix<T> band (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> band (  T a ,  Matrix<T> B  );

Matrix<T> band (  Matrix<T> A ,  T b  );

Tensor<T> band (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> band (  T a ,  Tensor<T> B  );

Tensor<T> band (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Value type T must support bitwise conjunction (bool, char, int, and so on).

Result: Each element of the result value is set to the bitwise and of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = band(A, B)` produces a result equivalent to `Z(i) = band(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is bitwise anded to all all of the elements of the other argument; for example, `Z = band(A, b)` produces `Z(i) = band(A(i), b)`.

Example:

```
Vector<int> Z, A, B;
Z = band(A, B);
```

See Also: `bnot` (section 4.2.10) `bor` (section 4.2.9) `bxor` (section 4.2.11)

4.2.9. bor

Description: Elementwise bitwise or.

Syntax:

```
Vector<T> bor (  Vector<T> A ,  Vector<T> B  );

Vector<T> bor (  T a ,  Vector<T> B  );

Vector<T> bor (  Vector<T> A ,  T b  );

Matrix<T> bor (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> bor (  T a ,  Matrix<T> B  );

Matrix<T> bor (  Matrix<T> A ,  T b  );

Tensor<T> bor (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> bor (  T a ,  Tensor<T> B  );

Tensor<T> bor (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Value type T must support bitwise disjunction (bool, char, int, and so on).

Result: Each element of the result value is set to the bitwise or of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = bor(A, B)` produces a result equivalent to `Z(i) = bor(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is bitwise ored to all all of the elements of the other argument; for example, `Z = bor(A, b)` produces `Z(i) = bor(A(i), b)`.

Example:

```
Vector<int> Z, A, B;
Z = bor(A, B);
```

See Also: `band` (section 4.2.8) `bnot` (section 4.2.10) `bxor` (section 4.2.11)

4.2.10. bnot

Description: Elementwise bitwise negation.

Syntax:

```
Vector<T> bnot (  Vector<T> A  );

Matrix<T> bnot (  Matrix<T> A  );

Tensor<T> bnot (  Tensor<T> A  );
```

Result: Each element of the result view is set to bitwise negation of the corresponding element of the argument. For instance, if the argument is a vector of int, `Z = bnot(A)` produces a result equivalent to `Z(i) = ~A(i)` for all the elements of the vector. Valid only on value types supporting bitwise negation (bool, char, int, and so on).

Example:

```
Vector<bool> Z, A;
Z = neg(A);
```

See Also: `lnot` (section 4.2.35) `neg` (section 4.2.53)

4.2.11. bxor

Description: Elementwise bitwise exclusive or.

Syntax:

```
Vector<T> bxor (  Vector<T> A ,  Vector<T> B  );

Vector<T> bxor (  T a ,  Vector<T> B  );

Vector<T> bxor (  Vector<T> A ,  T b  );

Matrix<T> bxor (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> bxor (  T a ,  Matrix<T> B  );

Matrix<T> bxor (  Matrix<T> A ,  T b  );

Tensor<T> bxor (  Tensor<T> A ,  Tensor<T> B  );
```

```
Tensor<T> bxor (  T a ,  Tensor<T> B  );

Tensor<T> bxor (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Value type T must support bitwise exlusive-or (bool, char, int, and so on).

Result: Each element of the result value is set to the bitwise exclusive or of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = bxor(A, B)` produces a result equivalent to `Z(i) = bxor(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is bitwise exclusive ored to all all of the elements of the other argument; for example, `Z = bxor(A, b)` produces `Z(i) = bxor(A(i), b)`.

Example:

```
Vector<int> Z, A, B;
Z = bxor(A, B);
```

See Also: `band` (section 4.2.8) `bor` (section 4.2.9) `bnot` (section 4.2.10)

4.2.12. `ceil`

Description: Elementwise floating-point ceiling.

Syntax:

```
Vector<T> ceil (  Vector<T> A  );

Matrix<T> ceil (  Matrix<T> A  );

Tensor<T> ceil (  Tensor<T> A  );
```

Result: Each element of the result view is set to the floating-point value of the argument view rounded up to the next integral value. For instance, if the argument is a vector, `Z = ceil(A)` produces a result equivalent to `Z(i) = ceil(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = ceil(A);
```

See Also: `floor` (section 4.2.22)

4.2.13. `conj`

Description: Elementwise complex conjugate.

Syntax:

```
Vector<T> conj (  Vector<T> A  );

Matrix<T> conj (  Matrix<T> A  );

Tensor<T> conj (  Tensor<T> A  );
```

Result: Each element of the result view is set to the complex conjugate of the corresponding element of the argument. For instance, if the argument is a vector, `Z = conj(A)` produces a result equivalent to `Z(i) = conj(A(i))` for all the elements of the vector.

Example:

```
Vector<complex<float> > Z, A;
Z = conj(A);
```

See Also: `real` (section 4.2.55) `imag` (section 4.2.27)

4.2.14. cos

Description: Elementwise trigonometric cosine.

Syntax:

```
Vector<T> cos (  Vector<T> A  );

Matrix<T> cos (  Matrix<T> A  );

Tensor<T> cos (  Tensor<T> A  );
```

Result: Each element of the result view is set to cosine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = cos(A)` produces a result equivalent to `Z(i) = cos(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = cos(A);
```

See Also: `sin` (section 4.2.59) `tan` (section 4.2.64)

4.2.15. cosh

Description: Elementwise hyperbolic cosine.

Syntax:

```
Vector<T> cosh (  Vector<T> A  );

Matrix<T> cosh (  Matrix<T> A  );

Tensor<T> cosh (  Tensor<T> A  );
```

Result: Each element of the result view is set to hyperbolic cosine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = cosh(A)` produces a result equivalent to `Z(i) = cosh(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = cosh(A);
```

See Also: `sinh` (section 4.2.60) `tanh` (section 4.2.65)

4.2.16. div

Description: Elementwise division.

Syntax:

```
Vector<T> div (  Vector<T> A ,  Vector<T> B  );

Vector<T> div (  T a ,  Vector<T> B  );

Vector<T> div (  Vector<T> A ,  T b  );

Matrix<T> div (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> div (  T a ,  Matrix<T> B  );

Matrix<T> div (  Matrix<T> A ,  T b  );

Tensor<T> div (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> div (  T a ,  Tensor<T> B  );

Tensor<T> div (  Tensor<T> A ,  T b  );
```

Operator Syntax: Division can also be written in operator form. `div(A, B)` is equivalent to `A / B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the fraction of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = div(A, B)` produces a result equivalent to `Z(i) = A(i) / B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it either divides or is divided by all of the elements of the other argument; for example, `Z = div(A, b)` produces `Z(i) = A(i) / b`.

Example:

```
Vector<float> Z, A, B;
Z = div(A, B);
```

See Also: `add` (section 4.2.2) `mul` (section 4.2.51) `sub` (section 4.2.63)

4.2.17. eq

Description: Elementwise equality comparison.

Syntax:

```
Vector<bool> eq (  Vector<T> A ,  Vector<T> B  );

Vector<bool> eq (  T a ,  Vector<T> B  );

Vector<bool> eq (  Vector<T> A ,  T b  );

Matrix<bool> eq (  Matrix<T> A ,  Matrix<T> B  );

Matrix<bool> eq (  T a ,  Matrix<T> B  );
```

```
Matrix<bool> eq ( Matrix<T> A , T b );

Tensor<bool> eq ( Tensor<T> A , Tensor<T> B );

Tensor<bool> eq ( T a , Tensor<T> B );

Tensor<bool> eq ( Tensor<T> A , T b );
```

Operator Syntax: Equality comparison can also be written in operator form. `eq(A, B)` is equivalent to `A == B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the two arguments are equal, false otherwise. For instance, if the arguments are vectors, `Z = eq(A, B)` produces a result equivalent to `Z(i) = A(i) == B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = eq(A, b)` produces `Z(i) = A(i) == b`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size), B(size);
Z = eq(A, B);
```

See Also: `ge` (section 4.2.24) `gt` (section 4.2.25) `le` (section 4.2.34) `lt` (section 4.2.39) `ne` (section 4.2.52)

4.2.18. `euler`

Description: Elementwise euler function.

Syntax:

```
Vector<complex<T> > euler ( Vector<T> A );

Matrix<complex<T> > euler ( Matrix<T> A );

Tensor<complex<T> > euler ( Tensor<T> A );
```

Result: Each element of the result view is a complex unit vector rotated by the angle given in the corresponding element of the argument. For instance, if the argument is a vector, `Z = euler(A)` produces a result equivalent to `Z(i) = polar(1, A(i))` for all the elements of the vector.

Example:

```
length_type size = 32;
Vector<float> A(size);
Vector<complex<float> > Z(size);
Z = euler(A);
```

See Also: `arg` (section 4.2.4)

4.2.19. `exp`

Description: Elementwise natural exponential.

Syntax:

```
Vector<T> exp (  Vector<T> A  );

Matrix<T> exp (  Matrix<T> A  );

Tensor<T> exp (  Tensor<T> A  );
```

Result: Each element of the result view is set to natural exponential of the corresponding element of the argument. For instance, if the argument is a vector, `Z = exp(A)` produces a result equivalent to `Z(i) = exp(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = exp(A);
```

See Also: `exp10` (section 4.2.20) `log` (section 4.2.36) `log10` (section 4.2.37)

4.2.20. exp10

Description: Elementwise base-10 exponential.

Syntax:

```
Vector<T> exp10 (  Vector<T> A  );

Matrix<T> exp10 (  Matrix<T> A  );

Tensor<T> exp10 (  Tensor<T> A  );
```

Result: Each element of the result view is set to base-10 exponential of the corresponding element of the argument. For instance, if the argument is a vector, `Z = exp10(A)` produces a result equivalent to `Z(i) = exp10(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = exp10(A);
```

See Also: `exp` (section 4.2.19) `log` (section 4.2.36) `log10` (section 4.2.37)

4.2.21. expoavg

Description: Elementwise exponential average.

Syntax:

```
Vector<T> expoavg (  Vector<T> A ,  Vector<T> B ,  Vector<T> C  );

Matrix<T> expoavg (  Matrix<T> A ,  Matrix<T> B ,  Matrix<T> C  );

Tensor<T> expoavg (  Tensor<T> A ,  Tensor<T> B ,  Tensor<T> C  );
```

Operator Syntax: Addition can also be written in operator form. `expoavg(A, B, C)` is equivalent to `A*B + (1-A)*C`.

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the exponential average of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = expoavg(A, B, C)` produces a result equivalent to `Z(i) = A(i) * B(i)) + (1 - A(i)) * C(i)` for all of the elements of the vector. If any of the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = expoavg(A, b, C)` produces `Z(i) = A(i) * b + (1 - A(i)) * C(i)`.

Example:

```
Vector<float> Z, A, B, C;
Z = expoavg(A, B, C);
```

See Also:

4.2.22. `floor`

Description: Elementwise floating-point floor.

Syntax:

```
Vector<T> floor (  Vector<T> A  );

Matrix<T> floor (  Matrix<T> A  );

Tensor<T> floor (  Tensor<T> A  );
```

Result: Each element of the result view is set to the floating-point value of the argument view rounded down to the next integral value. For instance, if the argument is a vector, `Z = floor(A)` produces a result equivalent to `Z(i) = floor(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = floor(A);
```

See Also: `ceil` (section 4.2.12)

4.2.23. `fmod`

Description: Floating-point modulo (remainder after division).

Syntax:

```
Vector<T> fmod (  Vector<T> A ,  Vector<T> B  );

Vector<T> fmod (  T a ,  Vector<T> B  );

Vector<T> fmod (  Vector<T> A ,  T b  );

Matrix<T> fmod (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> fmod (  T a ,  Matrix<T> B  );
```

```
Matrix<T> fmod (  Matrix<T> A ,  T b  );

Tensor<T> fmod (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> fmod (  T a ,  Tensor<T> B  );

Tensor<T> fmod (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the sum of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = fmod(A, B)` produces a result equivalent to `Z(i) = fmod(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is fmoded to all of the elements of the other argument; for example, `Z = fmod(A, b)` produces `Z(i) = fmod(A(i), b)`.

Example:

```
Vector<float> Z, A, B;
Z = fmod(A, B);
```

See Also:

4.2.24. ge

Description: Elementwise greater-than or equal comparison.

Syntax:

```
Vector<bool> ge (  Vector<T> A ,  Vector<T> B  );

Vector<bool> ge (  T a ,  Vector<T> B  );

Vector<bool> ge (  Vector<T> A ,  T b  );

Matrix<bool> ge (  Matrix<T> A ,  Matrix<T> B  );

Matrix<bool> ge (  T a ,  Matrix<T> B  );

Matrix<bool> ge (  Matrix<T> A ,  T b  );

Tensor<bool> ge (  Tensor<T> A ,  Tensor<T> B  );

Tensor<bool> ge (  T a ,  Tensor<T> B  );

Tensor<bool> ge (  Tensor<T> A ,  T b  );
```

Operator Syntax: Greater-than or equal comparison can also be written in operator form. `ge(A, B)` is equivalent to `A >= B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the first argument is greater-than or equal the second argument, false otherwise. For instance, if the arguments are vectors, `Z = ge(A, B)` produces a result equivalent to `Z(i) = A(i) >= B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = ge(A, b)` produces `Z(i) = A(i) >= b`.

Example:

```
Vector<bool> Z;
Vector<float> A, B;
Z = ge(A, B);
```

See Also: `eq` (section 4.2.17) `gt` (section 4.2.25) `le` (section 4.2.34) `lt` (section 4.2.39) `ne` (section 4.2.52)

4.2.25. gt

Description: Elementwise greater-than comparison.

Syntax:

```
Vector<bool> gt (  Vector<T> A ,  Vector<T> B  );

Vector<bool> gt (  T a ,  Vector<T> B  );

Vector<bool> gt (  Vector<T> A ,  T b  );

Matrix<bool> gt (  Matrix<T> A ,  Matrix<T> B  );

Matrix<bool> gt (  T a ,  Matrix<T> B  );

Matrix<bool> gt (  Matrix<T> A ,  T b  );

Tensor<bool> gt (  Tensor<T> A ,  Tensor<T> B  );

Tensor<bool> gt (  T a ,  Tensor<T> B  );

Tensor<bool> gt (  Tensor<T> A ,  T b  );
```

Operator Syntax: Greater-than comparison can also be written in operator form. `gt(A, B)` is equivalent to `A > B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the first argument is greater-than the second argument, false otherwise. For instance, if the arguments are vectors, `Z = gt(A, B)` produces a result equivalent to `Z(i) = A(i) > B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = gt(A, b)` produces `Z(i) = A(i) > b`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size), B(size);
Z = gt(A, B);
```

See Also: `eq` (section 4.2.17) `ge` (section 4.2.24) `le` (section 4.2.34) `lt` (section 4.2.39) `ne` (section 4.2.52)

4.2.26. hypot

Description: Hypotenuse of right triangle.

Syntax:

```
Vector<T> hypot (  Vector<T> A ,  Vector<T> B  );

Vector<T> hypot (  T a ,  Vector<T> B  );

Vector<T> hypot (  Vector<T> A ,  T b  );

Matrix<T> hypot (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> hypot (  T a ,  Matrix<T> B  );

Matrix<T> hypot (  Matrix<T> A ,  T b  );

Tensor<T> hypot (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> hypot (  T a ,  Tensor<T> B  );

Tensor<T> hypot (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the square-root of the sum of sqaures of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = hypot(A, B)` produces a result equivalent to `Z(i) = sqrt(sq(A(i)) + sq(B(i)))` for all of the elements of the vector. If either of the arguments is a scalar, it is used with all of the elements of the other argument; for example, `Z = hypot(A, b)` produces `Z(i) = sqrt(sq(A(i)) + sq(b))`.

Example:

```
Vector<float> Z, A, B;
Z = hypot(A, B);
```

See Also: `atan2` (section 4.2.7)

4.2.27. `imag`

Description: Elementwise imaginary part of complex.

Syntax:

```
Vector<T> imag (  Vector<complex<T> > A  );

Vector<T> imag (  Matrix<complex<T> > A  );

Vector<T> imag (  Tensor<complex<T> > A  );
```

Result: Each element of the result view is set to the imaginary component of the corresponding element of the argument. For instance, if the argument is a vector, `Z = imag(A)` produces a result equivalent to `Z(i) = imag(A(i))` for all the elements of the vector.

Example:

```
Vector<complex<float> > A;
Vector<float> Z;
Z = imag(A);
```

See Also: `arg` (section 4.2.4) `real` (section 4.2.55)

4.2.28. `is_finite`

Description: Elementwise check for finite floating-point value.

Syntax:

```
Vector<bool> is_finite (  Vector<T> A  );

Matrix<bool> is_finite (  Matrix<T> A  );

Tensor<bool> is_finite (  Tensor<T> A  );
```

Result: Each element of the result view is set to true of the corresponding element of the argument is a finite floating-point value, false otherwise. For instance, if the argument is a vector, `Z = is_finite(A)` produces a result equivalent to `Z(i) = is_finite(A(i))` for all the elements of the vector. For arguments with complex value type, output is conjunction of is_finite for real and imaginary components. `Z(i) = is_finite(real(A(i))) && is_finite(imag(A(i)))`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size);
Z = is_finite(A);
```

See Also: `is_nan` (section 4.2.29) `is_normal` (section 4.2.30)

4.2.29. `is_nan`

Description: Elementwise check for floating-point NaN (not a number).

Syntax:

```
Vector<bool> is_nan (  Vector<T> A  );

Matrix<bool> is_nan (  Matrix<T> A  );

Tensor<bool> is_nan (  Tensor<T> A  );
```

Result: Each element of the result view is set to true of the corresponding element of the argument is a NaN (not a number), false otherwise. For instance, if the argument is a vector, `Z = is_nan(A)` produces a result equivalent to `Z(i) = is_nan(A(i))` for all the elements of the vector. For arguments with complex value type, output is conjunction of is_nan for real and imaginary components. `Z(i) = is_nan(real(A(i))) && is_nan(imag(A(i)))`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size);
Z = is_nan(A);
```

See Also: `is_finite` (section 4.2.28) `is_normal` (section 4.2.30)

4.2.30. is_normal

Description: Elementwise check for floating-point normal value.

Syntax:

```
Vector<bool> is_normal ( Vector<T> A );

Matrix<bool> is_normal ( Matrix<T> A );

Tensor<bool> is_normal ( Tensor<T> A );
```

Result: Each element of the result view is set to true of the corresponding element of the argument is a normal floating-point value, false otherwise. For instance, if the argument is a vector, `Z = is_normal(A)` produces a result equivalent to `Z(i) = is_normal(A(i))` for all the elements of the vector. For arguments with complex value type, output is conjunction of is_normal for real and imaginary components. `Z(i) = is_normal(real(A(i))) && is_normal(imag(A(i)))`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size);
Z = is_normal(A);
```

See Also: `is_finite` (section 4.2.28) `is_nan` (section 4.2.29)

4.2.31. ite

Description: Elementwise if-then-else.

Syntax:

```
Vector<T> ite ( Vector<bool> A , Vector<T> B , Vector<T> C );

Matrix<T> ite ( Matrix<bool> A , Matrix<T> B , Matrix<T> C );

Tensor<T> ite ( Tensor<bool> A , Tensor<T> B , Tensor<T> C );
```

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the if-then-else evaluation of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = ite(A, B, C)` produces a result equivalent to `Z(i) = A(i) ? B(i) : C(i)` for all of the elements of the vector. If any of the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = ite(A, b, C)` produces `Z(i) = A(i) ? b : C(i)`.

Example:

```
Vector<float> Z, A, B, C;
Z = ite(A, B, C);
```

See Also:

4.2.32. jmul

Description: Elementwise multiplication by conjugate.

Syntax:

```
Vector<T> jmul ( Vector<T> A , Vector<T> B );

Vector<T> jmul ( T a , Vector<T> B );

Vector<T> jmul ( Vector<T> A , T b );

Matrix<T> jmul ( Matrix<T> A , Matrix<T> B );

Matrix<T> jmul ( T a , Matrix<T> B );

Matrix<T> jmul ( Matrix<T> A , T b );

Tensor<T> jmul ( Tensor<T> A , Tensor<T> B );

Tensor<T> jmul ( T a , Tensor<T> B );

Tensor<T> jmul ( Tensor<T> A , T b );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension. Value type must be complex.

Result: Each element of the result value is set to the product of the corresponding element of the first argument with the conjugate of the second argument. For instance, if the arguments are vectors, `Z = jmul(A, B)` produces a result equivalent to `Z(i) = A(i) * conj(B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is scales all of the elements of the other argument; for example, `Z = jmul(A, b)` produces `Z(i) = A(i) * conj(b)`.

Example:

```
Vector<complex<float> > Z, A, B;
Z = jmul(A, B);
```

See Also: `mul` (section 4.2.51)

4.2.33. land

Description: Elementwise logical and.

Syntax:

```
Vector<T> land ( Vector<T> A , Vector<T> B );

Vector<T> land ( T a , Vector<T> B );

Vector<T> land ( Vector<T> A , T b );

Matrix<T> land ( Matrix<T> A , Matrix<T> B );

Matrix<T> land ( T a , Matrix<T> B );

Matrix<T> land ( Matrix<T> A , T b );
```

```
Tensor<T> land (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> land (  T a ,  Tensor<T> B  );

Tensor<T> land (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension. Value type T must be bool.

Result: Each element of the result value is set to the logical and of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = land(A, B)` produces a result equivalent to `Z(i) = land(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is logical anded to all all of the elements of the other argument; for example, `Z = land(A, b)` produces `Z(i) = land(A(i), b)`.

Example:

```
Vector<bool> Z, A, B;
Z = land(A, B);
```

See Also: `bnot` (section 4.2.35) `bor` (section 4.2.38) `bxor` (section 4.2.40)

4.2.34. `le`

Description: Elementwise less-than or equal comparison.

Syntax:

```
Vector<bool> le (  Vector<T> A ,  Vector<T> B  );

Vector<bool> le (  T a ,  Vector<T> B  );

Vector<bool> le (  Vector<T> A ,  T b  );

Matrix<bool> le (  Matrix<T> A ,  Matrix<T> B  );

Matrix<bool> le (  T a ,  Matrix<T> B  );

Matrix<bool> le (  Matrix<T> A ,  T b  );

Tensor<bool> le (  Tensor<T> A ,  Tensor<T> B  );

Tensor<bool> le (  T a ,  Tensor<T> B  );

Tensor<bool> le (  Tensor<T> A ,  T b  );
```

Operator Syntax: less-than or equal comparison can also be written in operator form. `le(A, B)` is equivalent to `A <= B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the first argument is less-than or equal the second argument, false otherwise. For instance, if the arguments are vectors, `Z = le(A, B)` produces a result equivalent to `Z(i) = A(i) <= B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = le(A, b)` produces `Z(i) = A(i) <= b`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size), B(size);
Z = le(A, B);
```

See Also: `eq` (section 4.2.17) `ge` (section 4.2.24) `gt` (section 4.2.25) `lt` (section 4.2.39) `ne` (section 4.2.52)

4.2.35. `lnot`

Description: Elementwise logical negation.

Syntax:

```
Vector<T> lnot (  Vector<T> A  );

Matrix<T> lnot (  Matrix<T> A  );

Tensor<T> lnot (  Tensor<T> A  );
```

Result: Each element of the result view is set to logical negation of the corresponding element of the argument. For instance, if the argument is a vector of bool, `Z = lnot(A)` produces a result equivalent to `Z(i) = !A(i)` for all the elements of the vector. Valid only on value types supporting logical negation (bool, char, int, and so on).

Example:

```
Vector<bool> Z, A;
Z = lnot(A);
```

See Also: `bnot` (section 4.2.10) `neg` (section 4.2.53)

4.2.36. `log`

Description: Elementwise natural logarithm.

Syntax:

```
Vector<T> log (  Vector<T> A  );

Matrix<T> log (  Matrix<T> A  );

Tensor<T> log (  Tensor<T> A  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result view is set to natural logarithm of the corresponding element of the argument. For instance, if the argument is a vector, `Z = log(A)` produces a result equivalent to `Z(i) = log(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = log(A);
```

See Also: `exp` (section 4.2.19) `exp10` (section 4.2.20) `log10` (section 4.2.37)

4.2.37. `log10`

Description: Elementwise base-10 logarithm.

Syntax:

```
Vector<T> log10 (  Vector<T> A  );

Matrix<T> log10 (  Matrix<T> A  );

Tensor<T> log10 (  Tensor<T> A  );
```

Result: Each element of the result view is set to base-10 logarithm of the corresponding element of the argument. For instance, if the argument is a vector, `Z = log10(A)` produces a result equivalent to `Z(i) = log10(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = log10(A);
```

See Also: `exp` (section 4.2.19) `exp10` (section 4.2.20) `log` (section 4.2.36)

4.2.38. `lor`

Description: Elementwise logical or.

Syntax:

```
Vector<T> lor (  Vector<T> A ,  Vector<T> B  );

Vector<T> lor (  T a ,  Vector<T> B  );

Vector<T> lor (  Vector<T> A ,  T b  );

Matrix<T> lor (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> lor (  T a ,  Matrix<T> B  );

Matrix<T> lor (  Matrix<T> A ,  T b  );

Tensor<T> lor (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> lor (  T a ,  Tensor<T> B  );

Tensor<T> lor (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension. Value type T must support logical negation (bool, char, int, and so on).

Result: Each element of the result value is set to the logical or of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = lor(A, B)` produces a result equivalent to `Z(i) = lor(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is logical ored to all all of the elements of the other argument; for example, `Z = lor(A, b)` produces `Z(i) = lor(A(i), b)`.

Example:

```
Vector<bool> Z, A, B;
Z = lor(A, B);
```

See Also: `land` (section 4.2.33) `lnot` (section 4.2.35) `lxor` (section 4.2.40)

4.2.39. `lt`

Description: Elementwise less-than comparison.

Syntax:

```
Vector<bool> lt ( Vector<T> A , Vector<T> B );

Vector<bool> lt ( T a , Vector<T> B );

Vector<bool> lt ( Vector<T> A , T b );

Matrix<bool> lt ( Matrix<T> A , Matrix<T> B );

Matrix<bool> lt ( T a , Matrix<T> B );

Matrix<bool> lt ( Matrix<T> A , T b );

Tensor<bool> lt ( Tensor<T> A , Tensor<T> B );

Tensor<bool> lt ( T a , Tensor<T> B );

Tensor<bool> lt ( Tensor<T> A , T b );
```

Operator Syntax: Less-than comparison can also be written in operator form. `lt(A, B)` is equivalent to `A < B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the first argument is less-than the second argument, false otherwise. For instance, if the arguments are vectors, `Z = lt(A, B)` produces a result equivalent to `Z(i) = A(i) < B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = lt(A, b)` produces `Z(i) = A(i) < b`.

Example:

```
length_type size = 32;
Vector<bool> Z(size);
Vector<float> A(size), B(size);
Z = lt(A, B);
```

See Also: `eq` (section 4.2.17) `ge` (section 4.2.24) `gt` (section 4.2.25) `le` (section 4.2.34) `ne` (section 4.2.52)

4.2.40. `lxor`

Description: Elementwise bitwise exclusive or.

Syntax:

```
Vector<T> lxor (  Vector<T> A ,  Vector<T> B  );

Vector<T> lxor (  T a ,  Vector<T> B  );

Vector<T> lxor (  Vector<T> A ,  T b  );

Matrix<T> lxor (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> lxor (  T a ,  Matrix<T> B  );

Matrix<T> lxor (  Matrix<T> A ,  T b  );

Tensor<T> lxor (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> lxor (  T a ,  Tensor<T> B  );

Tensor<T> lxor (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension. Value type T must support bitwise negation (bool, char, int, and so on).

Result: Each element of the result value is set to the bitwise exclusive or of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = lxor(A, B)` produces a result equivalent to `Z(i) = lxor(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is bitwise exclusive ored to all all of the elements of the other argument; for example, `Z = lxor(A, b)` produces `Z(i) = lxor(A(i), b)`.

Example:

```
Vector<bool> Z, A, B;
Z = lxor(A, B);
```

See Also: `land` (section 4.2.33) `lor` (section 4.2.38) `lnot` (section 4.2.35)

4.2.41. `ma`

Description: Elementwise multiplication-addition.

Syntax:

```
Vector<T> ma (  Vector<T> A ,  Vector<T> B ,  Vector<T> C  );

Matrix<T> ma (  Matrix<T> A ,  Matrix<T> B ,  Matrix<T> C  );

Tensor<T> ma (  Tensor<T> A ,  Tensor<T> B ,  Tensor<T> C  );
```

Operator Syntax: Addition can also be written in operator form. `ma(A, B, C)` is equivalent to `(A * B) + C`.

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the product-sum of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = ma(A, B, C)` produces a result equivalent to `Z(i) = (A(i) * B(i)) + C(i)` for all of the elements of the vector. If any of

the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = ma(A, b, C)` produces `Z(i) = (A(i) * b) + C(i)`.

Example:

```
Vector<float> Z, A, B, C;
Z = ma(A, B, C);
```

See Also: `am` (section 4.2.3) `msb` (section 4.2.50) `sbm` (section 4.2.58)

4.2.42. `mag`

Description: Elementwise magnitude.

Syntax:

```
Vector<T> mag (  Vector<T> A  );

Vector<T> mag (  Vector<complex<T> > A  );

Matrix<T> mag (  Matrix<T> A  );

Matrix<T> mag (  Matrix<complex<T> > A  );

Tensor<T> mag (  Tensor<T> A  );

Tensor<T> mag (  Tensor<complex<T> > A  );
```

Result: Each element of the result view is set to the magnitude (equivalently the absolute value) of the corresponding element of the argument. For instance, if the argument is a vector, `Z = mag(A)` produces a result equivalent to `Z(i) = mag(A(i))` for all the elements of the vector. If argument is a view of scalars, return type is a view of scalars. If argument is a view of complex, return type is a view of scalars.

Example:

```
Vector<float> Z, A;
Z = mag(A);
```

See Also: `magsq` (section 4.2.43)

4.2.43. `magsq`

Description: Elementwise magnitude squared.

Syntax:

```
Vector<T> magsq (  Vector<T> A  );

Vector<T> magsq (  Vector<complex<T> > A  );

Matrix<T> magsq (  Matrix<T> A  );

Matrix<T> magsq (  Matrix<complex<T> > A  );

Tensor<T> magsq (  Tensor<T> A  );
```

```
Tensor<T> magsq (  Tensor<complex<T> > A  );
```

Result: Each element of the result view is set to the magnitude squared of the corresponding element of the argument. For instance, if the argument is a vector, `Z = magsq(A)` produces a result equivalent to `Z(i) = magsq(A(i))` for all the elements of the vector. For views of scalars, return type is a view of scalars. For views of complex, return type is a view of scalars.

Example:

```
Vector<float> Z, A;
Z = magsq(A);
```

See Also: `mag` (section 4.2.42)

4.2.44. max

Description: Elementwise maxima.

Syntax:

```
Vector<T> max (  Vector<T> A ,  Vector<T> B  );

Vector<T> max (  T a ,  Vector<T> B  );

Vector<T> max (  Vector<T> A ,  T b  );

Matrix<T> max (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> max (  T a ,  Matrix<T> B  );

Matrix<T> max (  Matrix<T> A ,  T b  );

Tensor<T> max (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> max (  T a ,  Tensor<T> B  );

Tensor<T> max (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the maxima of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = max(A, B)` produces a result equivalent to `Z(i) = max(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = max(A, b)` produces `Z(i) = max(A(i), b)`.

Example:

```
Vector<float> Z, A, B;
Z = max(A, B);
```

See Also: `maxmg` (section 4.2.45) `maxmgsq` (section 4.2.46) `min` (section 4.2.47) `minmg` (section 4.2.48) `minmgsq` (section 4.2.49)

4.2.45. maxmg

Description: Elementwise magnitude maxima.

Syntax:

```
Vector<T> maxmg ( Vector<T> A , Vector<T> B );

Vector<T> maxmg ( T a , Vector<T> B );

Vector<T> maxmg ( Vector<T> A , T b );

Matrix<T> maxmg ( Matrix<T> A , Matrix<T> B );

Matrix<T> maxmg ( T a , Matrix<T> B );

Matrix<T> maxmg ( Matrix<T> A , T b );

Tensor<T> maxmg ( Tensor<T> A , Tensor<T> B );

Tensor<T> maxmg ( T a , Tensor<T> B );

Tensor<T> maxmg ( Tensor<T> A , T b );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the maxima of the magnitudes of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = maxmg(A, B)` produces a result equivalent to `Z(i) = max(mag(A(i)), mag(B(i)))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = maxmg(A, b)` produces `Z(i) = max(mag(A(i)), mag(b))`.

Example:

```
Vector<float> Z, A, B;
Z = maxmg(A, B);
```

See Also: `max` (section 4.2.44) `maxmgsq` (section 4.2.46) `min` (section 4.2.47) `minmg` (section 4.2.48) `minmgsq` (section 4.2.49)

4.2.46. `maxmgsq`

Description: Elementwise magnitude-squared maxima.

Syntax:

```
Vector<T> maxmgsq ( Vector<T> A , Vector<T> B );

Vector<T> maxmgsq ( T a , Vector<T> B );

Vector<T> maxmgsq ( Vector<T> A , T b );

Matrix<T> maxmgsq ( Matrix<T> A , Matrix<T> B );

Matrix<T> maxmgsq ( T a , Matrix<T> B );

Matrix<T> maxmgsq ( Matrix<T> A , T b );

Tensor<T> maxmgsq ( Tensor<T> A , Tensor<T> B );
```

```
Tensor<T> maxmgsq (  T a ,  Tensor<T> B  );

Tensor<T> maxmgsq (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the maxima of the magnitudes squared of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = maxmgsq(A, B)` produces a result equivalent to `Z(i) = max(mag(sq(A(i))), mag(sq(B(i))))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = maxmgsq(A, b)` produces `Z(i) = max(mag(sq(A(i))), mag(sq(b)))`.

Example:

```
Vector<float> Z, A, B;
Z = maxmgsq(A, B);
```

See Also: `max` (section 4.2.44) `maxmg` (section 4.2.45) `min` (section 4.2.47) `minmg` (section 4.2.48) `minmgsq` (section 4.2.49)

4.2.47. `min`

Description: Elementwise minima.

Syntax:

```
Vector<T> min (  Vector<T> A ,  Vector<T> B  );

Vector<T> min (  T a ,  Vector<T> B  );

Vector<T> min (  Vector<T> A ,  T b  );

Matrix<T> min (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> min (  T a ,  Matrix<T> B  );

Matrix<T> min (  Matrix<T> A ,  T b  );

Tensor<T> min (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> min (  T a ,  Tensor<T> B  );

Tensor<T> min (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the minima of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = min(A, B)` produces a result equivalent to `Z(i) = min(A(i), B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = min(A, b)` produces `Z(i) = min(A(i), b)`.

Example:

```
Vector<float> Z, A, B;
Z = min(A, B);
```

See Also: `max` (section 4.2.44) `maxmg` (section 4.2.45) `maxmgsq` (section 4.2.46) `minmg` (section 4.2.48) `minmgsq` (section 4.2.49)

4.2.48. `minmg`

Description: Elementwise magnitude minima.

Syntax:

```
Vector<T> minmg (  Vector<T> A ,  Vector<T> B  );

Vector<T> minmg (  T a ,  Vector<T> B  );

Vector<T> minmg (  Vector<T> A ,  T b  );

Matrix<T> minmg (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> minmg (  T a ,  Matrix<T> B  );

Matrix<T> minmg (  Matrix<T> A ,  T b  );

Tensor<T> minmg (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> minmg (  T a ,  Tensor<T> B  );

Tensor<T> minmg (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the minima of the magnitudes of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = minmg(A, B)` produces a result equivalent to `Z(i) = min(mag(A(i)), mag(B(i)))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = minmg(A, b)` produces `Z(i) = min(mag(A(i)), mag(b))`.

Example:

```
Vector<float> Z;
Vector<complex<float> > A, B;
Z = minmg(A, B);
```

See Also: `max` (section 4.2.44) `maxmg` (section 4.2.45) `maxmgsq` (section 4.2.46) `min` (section 4.2.47) `minmgsq` (section 4.2.49)

4.2.49. `minmgsq`

Description: Elementwise magnitude-squared minima.

Syntax:

```
Vector<T> minmgsq (  Vector<T> A ,  Vector<T> B  );

Vector<T> minmgsq (  T a ,  Vector<T> B  );

Vector<T> minmgsq (  Vector<T> A ,  T b  );
```

```
Matrix<T> minmgsq (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> minmgsq (  T a ,  Matrix<T> B  );

Matrix<T> minmgsq (  Matrix<T> A ,  T b  );

Tensor<T> minmgsq (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> minmgsq (  T a ,  Tensor<T> B  );

Tensor<T> minmgsq (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the minima of the magnitudes squared of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = minmgsq(A, B)` produces a result equivalent to `Z(i) = min(mag(sq(A(i))), mag(sq(B(i))))` for all of the elements of the vector. If either of the arguments is a scalar, it is compared with all of the elements of the other argument; for example, `Z = minmgsq(A, b)` produces `Z(i) = min(mag(sq(A(i))), mag(sq(b)))`.

Example:

```
Vector<float> Z;
Vector<complex<float> > A, B;
Z = minmgsq(A, B);
```

See Also: `max` (section 4.2.44) `maxmg` (section 4.2.45) `maxmgsq` (section 4.2.46) `min` (section 4.2.47) `minmg` (section 4.2.48)

4.2.50. `msb`

Description: Elementwise multiplication-addition.

Syntax:

```
Vector<T> msb (  Vector<T> A ,  Vector<T> B ,  Vector<T> C  );

Matrix<T> msb (  Matrix<T> A ,  Matrix<T> B ,  Matrix<T> C  );

Tensor<T> msb (  Tensor<T> A ,  Tensor<T> B ,  Tensor<T> C  );
```

Operator Syntax: Addition can also be written in operator form. `msb(A, B, C)` is equivalent to `(A * B) - C`.

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the product-difference of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = msb(A, B, C)` produces a result equivalent to `Z(i) = (A(i) * B(i)) - C(i)` for all of the elements of the vector. If any of the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = msb(A, b, C)` produces `Z(i) = (A(i) * b) - C(i)`.

Example:

```
Vector<float> Z, A, B, C;
Z = msb(A, B, C);
```

See Also: `am` (section 4.2.3) `msb` (section 4.2.41) `sbm` (section 4.2.58)

4.2.51. `mul`

Description: Elementwise multiplication.

Syntax:

```
Vector<T> mul (  Vector<T> A ,  Vector<T> B  );

Vector<T> mul (  T a ,  Vector<T> B  );

Vector<T> mul (  Vector<T> A ,  T b  );

Matrix<T> mul (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> mul (  T a ,  Matrix<T> B  );

Matrix<T> mul (  Matrix<T> A ,  T b  );

Tensor<T> mul (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> mul (  T a ,  Tensor<T> B  );

Tensor<T> mul (  Tensor<T> A ,  T b  );
```

Operator Syntax: Multiplication can also be written in operator form. `mul(A, B)` is equivalent to `A * B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the product of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = mul(A, B)` produces a result equivalent to `Z(i) = A(i) * B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is scales all of the elements of the other argument; for example, `Z = mul(A, b)` produces `Z(i) = A(i) * b`.

Example:

```
Vector<float> Z, A, B;
Z = mul(A, B);
```

See Also: `add` (section 4.2.2) `div` (section 4.2.16) `sub` (section 4.2.63)

4.2.52. ne

Description: Elementwise not-equal comparison.

Syntax:

```
Vector<bool> ne (  Vector<T> A ,  Vector<T> B  );

Vector<bool> ne (  T a ,  Vector<T> B  );
```

```
Vector<bool> ne (  Vector<T> A ,  T b  );

Matrix<bool> ne (  Matrix<T> A ,  Matrix<T> B  );

Matrix<bool> ne (  T a ,  Matrix<T> B  );

Matrix<bool> ne (  Matrix<T> A ,  T b  );

Tensor<bool> ne (  Tensor<T> A ,  Tensor<T> B  );

Tensor<bool> ne (  T a ,  Tensor<T> B  );

Tensor<bool> ne (  Tensor<T> A ,  T b  );
```

Operator Syntax: Not-equal comparison can also be written in operator form. `ne(A, B)` is equivalent to `A != B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to true if the corresponding elements of the two arguments are not equal, false otherwise. For instance, if the arguments are vectors, `Z = ne(A, B)` produces a result equivalent to `Z(i) = A(i) != B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it is compared to all of the elements of the other argument; for example, `Z = ne(A, b)` produces `Z(i) = A(i) != b`.

Example:

```
Vector<bool> Z;
Vector<float> A, B;
Z = ne(A, B);
```

See Also: `ge` (section 4.2.24) `gt` (section 4.2.25) `le` (section 4.2.34) `lt` (section 4.2.39) `ne` (section 4.2.52)

4.2.53. neg

Description: Elementwise arithmetic negation.

Syntax:

```
Vector<T> neg (  Vector<T> A  );

Matrix<T> neg (  Matrix<T> A  );

Tensor<T> neg (  Tensor<T> A  );
```

Result: Each element of the result view is set to arithmetic negation of the corresponding element of the argument. For instance, if the argument is a vector, `Z = neg(A)` produces a result equivalent to `Z(i) = -A(i)` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = neg(A);
```

See Also: `bnot` (section 4.2.10) `lnot` (section 4.2.35)

4.2.54. pow

Description: Elementwise raise to power.

Syntax:

```
Vector<T> pow (  Vector<T> A ,  Vector<T> B  );

Vector<T> pow (  T a ,  Vector<T> B  );

Vector<T> pow (  Vector<T> A ,  T b  );

Matrix<T> pow (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> pow (  T a ,  Matrix<T> B  );

Matrix<T> pow (  Matrix<T> A ,  T b  );

Tensor<T> pow (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> pow (  T a ,  Tensor<T> B  );

Tensor<T> pow (  Tensor<T> A ,  T b  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is set to the power of the corresponding elements of the first argument to the second argument. For instance, if the arguments are vectors, `Z = pow(A, B)` produces a result equivalent to `Z(i) = pow(A(i) ** B(i))` for all of the elements of the vector. If either of the arguments is a scalar, it is used with all of the elements of the other argument; for example, `Z = pow(A, b)` produces `Z(i) = A(i) ** b`.

Example:

```
Vector<float> Z, A, B;
Z = pow(A, B);
```

See Also: `div` (section 4.2.16) `mul` (section 4.2.51) `sub` (section 4.2.63)

4.2.55. real

Description: Elementwise real part of complex.

Syntax:

```
Vector<T> real (  Vector<complex<T> > A  );

Vector<T> real (  Matrix<complex<T> > A  );

Vector<T> real (  Tensor<complex<T> > A  );
```

Result: Each element of the result view is set to the real component of the corresponding element of the argument. For instance, if the argument is a vector, `Z = real(A)` produces a result equivalent to `Z(i) = real(A(i))` for all the elements of the vector.

Example:

```
Vector<complex<float> > A;
Vector<float> Z;
Z = real(A);
```

See Also: `arg` (section 4.2.4) `imag` (section 4.2.27)

4.2.56. `recip`

Description: Elementwise recipricol.

Syntax:

```
Vector<T> recip (  Vector<T> A  );

Matrix<T> recip (  Matrix<T> A  );

Tensor<T> recip (  Tensor<T> A  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result view is set to the recipricol square root of the corresponding element of the argument. For instance, if the argument is a vector, `Z = recip(A)` produces a result equivalent to `Z(i) = 1 / A(i)` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = recip(A);
```

See Also: `rsqrt` (section 4.2.57)

4.2.57. `rsqrt`

Description: Elementwise recipricol square root.

Syntax:

```
Vector<T> rsqrt (  Vector<T> A  );

Matrix<T> rsqrt (  Matrix<T> A  );

Tensor<T> rsqrt (  Tensor<T> A  );
```

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result view is set to the recipricol square root of the corresponding element of the argument. For instance, if the argument is a vector, `Z = rsqrt(A)` produces a result equivalent to `Z(i) = 1 / sqrt(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = rsqrt(A);
```

See Also: `sqrt` (section 4.2.62)

4.2.58. sbm

Description: Elementwise subtraction-multiplication.

Syntax:

```
Vector<T> sbm (  Vector<T> A ,  Vector<T> B ,  Vector<T> C  );

Matrix<T> sbm (  Matrix<T> A ,  Matrix<T> B ,  Matrix<T> C  );

Tensor<T> sbm (  Tensor<T> A ,  Tensor<T> B ,  Tensor<T> C  );
```

Operator Syntax: Addition can also be written in operator form. `sbm(A, B, C)` is equivalent to `(A + B) * C`.

Requirements: It is permissible for arguments to be scalar instead of a view. Scalars are treated a view with constant value. If multiple arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the difference-product of the corresponding elements of the arguments. For instance, if the arguments are vectors, `Z = sbm(A, B, C)` produces a result equivalent to `Z(i) = (A(i) - B(i)) * C(i)` for all of the elements of the vector. If any of the arguments are scalar, they are processed with all of the elements of the other arguments; for example, `Z = sbm(A, b, C)` produces `Z(i) = (A(i) - b) * C`.

Example:

```
Vector<float> Z, A, B, C;
Z = sbm(A, B, C);
```

See Also: `am` (section 4.2.3) `ma` (section 4.2.41) `msb` (section 4.2.50)

4.2.59. sin

Description: Elementwise trigonometric sine.

Syntax:

```
Vector<T> sin (  Vector<T> A  );

Matrix<T> sin (  Matrix<T> A  );

Tensor<T> sin (  Tensor<T> A  );
```

Result: Each element of the result view is set to sine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = sin(A)` produces a result equivalent to `Z(i) = sin(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = sin(A);
```

See Also: `cos` (section 4.2.14) `tan` (section 4.2.64)

4.2.60. sinh

Description: Elementwise hyberbolic sine.

Syntax:

```
Vector<T> sinh (  Vector<T> A  );

Matrix<T> sinh (  Matrix<T> A  );

Tensor<T> sinh (  Tensor<T> A  );
```

Result: Each element of the result view is set to hyberbolic sine of the corresponding element of the argument. For instance, if the argument is a vector, `Z = sinh(A)` produces a result equivalent to `Z(i) = sinh(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = sinh(A);
```

See Also: `cosh` (section 4.2.15) `tanh` (section 4.2.65)

4.2.61. sq

Description: Elementwise square.

Syntax:

```
Vector<T> sq (  Vector<T> A  );

Matrix<T> sq (  Matrix<T> A  );

Tensor<T> sq (  Tensor<T> A  );
```

Result: Each element of the result view is set to the square of the corresponding element of the argument. For instance, if the argument is a vector, `Z = sq(A)` produces a result equivalent to `Z(i) = A(i) * A(i)` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = sq(A);
```

See Also: `sqrt` (section 4.2.62)

4.2.62. sqrt

Description: Elementwise square root.

Syntax:

```
Vector<T> sqrt (  Vector<T> A  );

Matrix<T> sqrt (  Matrix<T> A  );

Tensor<T> sqrt (  Tensor<T> A  );
```

Result: Each element of the result view is set to the square root of the corresponding element of the argument. For instance, if the argument is a vector, `Z = sqrt(A)` produces a result equivalent to `Z(i) = sqrt(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = sqrt(A);
```

See Also: `sq` (section 4.2.61)

4.2.63. sub

Description: Elementwise subtraction.

Syntax:

```
Vector<T> sub (  Vector<T> A ,  Vector<T> B  );

Vector<T> sub (  T a ,  Vector<T> B  );

Vector<T> sub (  Vector<T> A ,  T b  );

Matrix<T> sub (  Matrix<T> A ,  Matrix<T> B  );

Matrix<T> sub (  T a ,  Matrix<T> B  );

Matrix<T> sub (  Matrix<T> A ,  T b  );

Tensor<T> sub (  Tensor<T> A ,  Tensor<T> B  );

Tensor<T> sub (  T a ,  Tensor<T> B  );

Tensor<T> sub (  Tensor<T> A ,  T b  );
```

Operator Syntax: Subtraction can also be written in operator form. `sub(A, B)` is equivalent to `A - B`.

Requirements: If both arguments are non-scalar, they must be the same size in each dimension.

Result: Each element of the result value is equal to the difference of the corresponding elements of the two arguments. For instance, if the arguments are vectors, `Z = sub(A, B)` produces a result equivalent to `Z(i) = A(i) - B(i)` for all of the elements of the vector. If either of the arguments is a scalar, it's difference with all of the elements of the other argument is computed; for example, `Z = sub(A, b)` produces `Z(i) = A(i) - b`.

Example:

```
Vector<float> Z, A, B;
Z = sub(A, B);
```

See Also: `add` (section 4.2.2) `div` (section 4.2.16) `mul` (section 4.2.51)

4.2.64. tan

Description: Elementwise trigonometric tangent.

Syntax:

```
Vector<T> tan (  Vector<T> A  );

Matrix<T> tan (  Matrix<T> A  );

Tensor<T> tan (  Tensor<T> A  );
```

Result: Each element of the result view is set to tangent of the corresponding element of the argument. For instance, if the argument is a vector, `Z = tan(A)` produces a result equivalent to `Z(i) = tan(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = tan(A);
```

See Also: `cos` (section 4.2.14) `sin` (section 4.2.59)

4.2.65. `tanh`

Description: Elementwise hyperbolic tangent.

Syntax:

```
Vector<T> tanh (  Vector<T> A  );

Matrix<T> tanh (  Matrix<T> A  );

Tensor<T> tanh (  Tensor<T> A  );
```

Result: Each element of the result view is set to hyperbolic tangent of the corresponding element of the argument. For instance, if the argument is a vector, `Z = tanh(A)` produces a result equivalent to `Z(i) = tanh(A(i))` for all the elements of the vector.

Example:

```
Vector<float> Z, A;
Z = tanh(A);
```

See Also: `cosh` (section 4.2.15) `sinh` (section 4.2.60)

4.3. Reduction Functions

4.3.1. `alltrue`

Syntax:

```
template <typename                          T,
          template <typename, typename> class ViewT,
          typename                          BlockT>
T
alltrue(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: When *T* is `bool`, the function returns `true` if all elements are `true`, otherwise `false`. When *T* is another type, `alltrue` returns the application of accumulation operator `band` with base value `~(T())` applied to *v*.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `alltrue` returns a scalar value of type *T*.

Example:

```
Vector<bool> bvec(4, true);

std::cout << alltrue(bvec) << std::endl; // prints 1

Vector<int> vec(3);

vec(0) = 0x00ff;
vec(1) = 0x119f;
vec(2) = 0x92f7;

std::cout << alltrue(vec) == 0x0097 << std::endl; // prints 1
```

See Also: `band` (section 4.2.8)

4.3.2. anytrue

Syntax:

```
    template <typename                        T,
              template <typename, typename> class ViewT,
              typename                        BlockT>
    T
    anytrue(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*.

Description: When *T* is `bool`, the function returns `true` if at least one element is `true`, otherwise `false`. When *T* is another type, `anytrue` returns the application of accumulation operator `bor` with base value `T()` applied to *v*.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `anytrue` returns a scalar value of type *T*.

Example:

```
Vector<bool> bvec(4, true);

std::cout << anytrue(bvec) << std::endl; // prints 1

Vector<int> vec(3);

vec(0) = 0x00ff;
vec(1) = 0x119f;
vec(2) = 0x92f7;

std::cout << anytrue(vec) == 0x93ff << std::endl; // prints 1
```

See Also: `bor` (section 4.2.9)

4.3.3. `maxmgsqval`

Syntax:

```
    template <typename                          T,
              template <typename, typename> class ViewT,
              typename                          BlockT>
    T
    maxmgsqval(
        ViewT<complex<T>, BlockT>         v,
        Index<ViewT<complex<T>, BlockT>::dim>& idx);
```

Template parameters

`T` is the type of the elements in the view and of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxmgsqval` computes the maximum squared magnitude of all the elements of the view and returns that value. It also returns, through the parameter *`idx`*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`. The function parameters *`v`* and *`idx`* are defined in terms of `complex<T>`.

Result: `maxmgsqval` returns a scalar value of type *`T`*.

Example:

```
   length_type         size = 13;
   typedef float T;
   Vector<complex<T> > vec(size, complex<T>(3, 4));
   Index<1>            idx;
   T                   val;

   vec(1) = complex<T>(6, 8);
   vec(2) = complex<T>(0.3, 0.4);
```

```
val = maxmgsqval(vec, idx);
cout << val << endl;  // prints 100
cout << idx << endl;  // prints 1
```

4.3.4. `maxmgval`

Syntax:

```
template <typename                          T,
          template <typename, typename> class ViewT,
          typename                          BlockT>
T
maxmgval(
    ViewT<T, BlockT>         v,
    Index<ViewT<T, BlockT>::dim>& idx);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxmgval` computes the maximum magnitude of all the elements of the view and returns that value. It also returns, through the parameter *idx*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: When *T* is `complex<W>`, `maxmgval` returns a scalar value of type *W*; otherwise it returns a value of type *T*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
Index<1>            idx;
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = maxmgval(vec, idx);
cout << val << endl;  // prints 10
cout << idx << endl;  // prints 1
```

4.3.5. `maxval`

Syntax:

```
template <typename                          T,
          template <typename, typename> class ViewT,
          typename                          BlockT>
```

```
    T
    maxval(
        ViewT<T, BlockT>         v,
        Index<ViewT<T, BlockT>::dim>& idx);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxval` computes the maximum of all the elements of the view and returns that value. It also returns, through the parameter *idx*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `maxval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 3.;
vec(3) = 2.;

Index<1>  idx;
float val = maxval(vec,idx);
std::cout << val << std::endl;  // prints 3.0
std::cout << idx << std::endl;  // prints 2
```

4.3.6. `meansqval`

Syntax:

```
    template <typename                      T,
              template <typename, typename> class ViewT,
              typename                      BlockT>
    T
    meansqval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `meansqval` sums the squares of all the elements of the view and returns that value divided by the number of elements.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `meansqval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 2.;
vec(3) = 3.;

std::cout << meansqval(vec) << std::endl;  // prints 3.5
```

See Also: `add` (section 4.2.2)

4.3.7. `meanval`

Syntax:

```
    template <typename                       T,
              template <typename, typename> class ViewT,
              typename                       BlockT>
    T
    meanval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `meanval` sums all the elements of the view and returns that value divided by the number of elements.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `meanval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 2.;
vec(3) = 3.;

std::cout << meanval(vec) << std::endl;  // prints 1.5
```

See Also: `add` (section 4.2.2)

4.3.8. `minmgsqval`

Syntax:

```
template <typename                        T,
          template <typename, typename> class ViewT,
          typename                        BlockT>
T
minmgsqval(
    ViewT<complex<T>, BlockT>         v,
    Index<ViewT<complex<T>, BlockT>::dim>& idx);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minmgsqval` computes the minimum squared magnitude of all the elements of the view and returns that value. It also returns, through the parameter *idx*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`. The function parameters *v* and *idx* are defined in terms of `complex<T>`.

Result: `minmgsqval` returns a scalar value of type *T*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
Index<1>            idx;
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = minmgsqval(vec, idx);
cout << val << endl;  // prints 0.25
cout << idx << endl;  // prints 2
```

4.3.9. `minmgval`

Syntax:

```
template <typename                        T,
          template <typename, typename> class ViewT,
          typename                        BlockT>
T
minmgval(
    ViewT<T, BlockT>         v,
    Index<ViewT<T, BlockT>::dim>& idx);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minmgval` computes the minimum magnitude of all the elements of the view and returns that value. It also returns, through the parameter *idx*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: When *T* is `complex<W>`, `minmgval` returns a scalar value of type *W*; otherwise it returns a value of type *T*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
Index<1>            idx;
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = minmgval(vec, idx);
cout << val << endl;  // prints 0.5
cout << idx << endl;  // prints 2
```

4.3.10. `minval`

Syntax:

```
template <typename                          T,
          template <typename, typename> class ViewT,
          typename                          BlockT>
T
minval(
    ViewT<T, BlockT>         v,
    Index<ViewT<T, BlockT>::dim>& idx);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minval` computes the minimum of all the elements of the view and returns that value. It also returns, through the parameter *idx*, the indices to locate that value in the view.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `minval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 3.;
vec(1) = 1.;
vec(2) = 0.;
vec(3) = 2.;

Index<1>  idx;
float val = minval(vec,idx);
std::cout << val << std::endl;  // prints 0.0
std::cout << idx << std::endl;  // prints 2
```

4.3.11. `sumsqval`

Syntax:

```
    template <typename                        T,
              template <typename, typename> class ViewT,
              typename                        BlockT>
    T
    sumsqval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `sumsqval` sums the squares of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `sumsqval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 2.;
vec(3) = 3.;

std::cout << sumsqval(vec) << std::endl;  // prints 14.0
```

See Also: `add` (section 4.2.2)

4.3.12. sumval

Syntax:

```
template <typename                          T,
          template <typename, typename> class ViewT,
          typename                          BlockT>
T
sumval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `sumval` sums all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `sumval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 2.;
vec(3) = 3.;

std::cout << sumval(vec) << std::endl;  // prints 6.0
```

See Also: `add` (section 4.2.2)

4.4. Linear Algebra Matrix-Vector Functions

4.4.1. cumsum

Description: Cumulative sum of matrix rows or columns

Syntax:

`void` **`cumsum<d>`** `(` *`const_Vector<T> A , Vector<T> B`* `);`

`void` **`cumsum<d>`** `(` *`const_Matrix<T> A , Matrix<T> B`* `);`

Requirements: Arguments `A` and `B` must be views of the same dimension. If matrices are passed, template parameter `d` must be either `row` (`0`) or `col` (`1`).

Result: If arguments are vectors, the template parameter `d` is ignored and each element `i` of result is set to the the sum of the `0` through `i`th elements of the input vector: `B(i) = sum(j = 0 ... i) of A(j)`.

If arguments are matrices, the template parameter `d`, controls whether summation is done along rows (if `d == row == 0`) or columns (if `d == col == 1`).

If `d == row`, then each result element `B(r, c)` is set to the sum of the `0` through `c`th elements of row `r` in matrix `A`: `B(r, c) = sum(j = 0 ... c) of A(r, c)`

If `d == col`, then each result element `B(r, c)` is set to the sum of the `0` through `r`th elements of column `c` in matrix `A`: `B(r, c) = Sum(j = 0 .. r) of A(r, c)`

Example:

```
length_type m = 32, n = 16;
Matrix<float> A(m, n), B(m, n);
cumsum<row>(A, B);
```

4.4.2. `cvjdot`

Description: Linear algebra conjugate dot-product.

Syntax:

`T` **`cvjdot`** `(` *`Vector<T> A`* `,` *`Vector<T> B`* `);`

Requirements: `A` and `B` must be vectors of the same size.

Result: Returns the conjugate dot-product (inner-product) of the argument vectors:

result = sum(i = 0 .. size-1) : A(i) * conj(B(i))

Example:

```
length_type n = 16;
Vector<complex<float> > A(n), B(n);
complex<float> result = cvjdot(A, B);
```

See Also: `dot` (section 4.4.3) `outer` (section 4.4.9)

4.4.3. `dot`

Description: Linear algebra dot-product.

Syntax:

`T` **`dot`** `(` *`Vector<T> A`* `,` *`Vector<T> B`* `);`

Requirements: `A` and `B` must be vectors of the same size.

Result: Returns the dot-product (inner-product) of the argument vectors. result = sum(i = 0 ... size-1) : A(i) * B(i)

Example:

```
length_type n = 16;
Vector<float> A(n), B(n);
float result = prod(A, B);
```

See Also: `cvjdot`(section 4.4.2)`outer`(section 4.4.9)

4.4.4. gemp

Description: Linear algebra generalized matrix product.

Syntax:

```
template <mat_op_type OpA,
          mat_op_type OpB,
          typename T0,
          typename ConstMatrix1T,
          typename ConstMatrix2T,
          typename T3,
          typename Matrix4T>
void
gemp(
   T0            alpha,
   ConstMatrix1T A,
   ConstMatrix2T B,
   T3            beta,
   Matrix4T      C)
```

Requirements: The number of columns of `OpA(A)` must equal the number of rows of `OpB(B)`.

Result: Computes the expression: C = alpha * OpA(A) * OpB(B) + beta * C

Template parameters `OpA` and `OpB` are of type `mat_op_type` (section 2.7.3) and specify operations on matrices `A` and `B`: `mat_ntrans` indicates no transpose, `mat_trans` indicates transpose, `mat_herm` indicates hermetian, and `mat_conj` indicates conjugation.

Example:

```
length_type m = 48, p = 16, n = 32;
Matrix<float> C(m, p), A(m, n), B(n, p);
gemp(0.5, A, B, 0.5, C);
```

See Also: `gems`(section 4.4.5)`prod`(section 4.4.10)`prodh`(section 4.4.13)`prodj`(section 4.4.14)`prodt`(section 4.4.15)

4.4.5. gems

Description: Linear algebra generalized matrix sum.

Syntax:

```
template <mat_op_type OpA,
          typename T0,
          typename ConstMatrix1T,
          typename T2,
          typename Matrix3>
T1
gems(
   T0            alpha,
   ConstMatrix1T A
   T2            beta,
   Matrix3T      C)
```

Requirements: If OpA equal mat_ntrans or mat_conj, A and B must have the same size.

If OpA equal mat_trans or mat_herm, the number of rows of A must equal the number of columns of B, and the number of columns of A must equal the number of rows of B.

Result: Computes the expression: C = alpha * OpA(A) + beta * C

Template parameter OpA is of type mat_op_type (section 2.7.3) and specify an operation on matrix A: mat_ntrans indicates no transpose, mat_trans indicates transpose, mat_herm indicates hermetian, and mat_conj indicates conjugation.

Example:

```
length_type m = 48, p = 16, n = 32;
Matrix<float> C(m, p), A(m, n), B(n, p);
gemp(0.5, A, B, 0.5, C);
```

See Also: gemp(section 4.4.4)

4.4.6. herm

Description: Matrix hermetian (conjugate-transpose).

Syntax:

Matrix<T> **herm** (*Matrix<T> A*);

Result: Returns a matrix hermetian view. The i, jth element of the result are the conjugated j, ith element of the argument.

result(i, j) = conj(A(j, i))

Note that herm() does itself not rearrange or modify data in memory. However, if the result is assigned to a destination matrix, during the copy into the destination, a corner-turn and conjugation may be performed.

Example:

```
length_type m = 32, n = 16;
Matrix<complex<float> > Z(m, n), A(n, m);
Z = herm(A);
```

See Also: trans(section 4.4.16)

4.4.7. kron

Description: Linear algebra kronecker product.

Syntax:

```
template <typename T0,
          typename View1,
          typename View2>
const_Matrix<PromotedT, unspecified>
kron(
   T0    alpha,
```

```
   View1 v,
   View2 w)
```

Requirements: Arguments `v` and `w` must have the same dimensionality.

Result: Returns the kronecker product of the two view parameters, scaled by `alpha`.

The number of output matrix rows equals the product of `v` and `w`'s rows. The number of ouput matrix columns equal the product of `v` and `w`'s columns.

The value type of the result matrix is the promoted value type of `alpha`, `v`, and `w`,

Example:

```
length_type m = 4, p = 16, n = 8, q = 12;
Matrix<float> C(m*p, n*q), V(m, n), W(p, q);
C = kron(0.5, V, W);
```

4.4.8. `modulate`

Description: Modulate vector with baseband frequency.

Syntax:

```
template <typename ConstVectorT0,
          typename T1,
          typename T2,
          typename VectorT3>
T1
modulate(
   ConstVector0T v
   T1            nu,
   T2            phi,
   Vector3T      w)
```

Result: Sets `w` to value of input `v` modulated with complex frequency `phi + i*nu`.

`w(i) = v(i) * exp((i * nu + phi)`*i*`)`

Where *i* is sqrt(-1)

Returns `v.size() * nu + phi`.

Example:

```
float nu = 3.14/16, phi = 3.14/2;
length_type size = 16;
Vector<complex<float> > Z(size), A(size);
modulate(A, nu, phi, B);
```

4.4.9. `outer`

Description: Linear algebra outer-product.

Syntax:

```
Matrix<T> outer (  Vector<T> A ,  Vector<T> B  );
```

Result: Returns the outer-product of the argument vectors. Each element `result(i, j)` is set to the product of the `i`th element of A and `j`th element of B.

result(i, j) = A(i) * B(j)

If the argument vectors have a complex value type, the conjugate of B is used in the product:

result(i, j) = A(i) * conj(B(j))

Example:

```
length_type m = 32, n = 16;
Vector<float> A(m), B(n);
Matrix<float> Z(m, n);
Z = outer(A, B);
```

See Also: `cvjdot` (section 4.4.2) `dot` (section 4.4.3)

4.4.10. `prod`

Description: Linear algebra product.

Syntax:

```
Matrix<T> prod (  Matrix<T> A ,  Matrix<T> B  );

Vector<T> prod (  Matrix<T> A ,  Vector<T> B  );

Vector<T> prod (  Vector<T> A ,  Matrix<T> B  );
```

Requirements: If both arguments are matrices, the number of columns of A must equal the number of rows of B.

If A is a matrix and B is a vector, the number of columns of A must equal the size of B.

If A is a vector and B is a matrix, the size of A must equal the number of rows of B.

Result: The result is equal to the linear algebraic product of the arguments.

If A is a `m` by `n` matrix and B is a `n` by `p` matrix, a `m` x `p` matrix is returned.

If A is a `m` by `n` matrix, and B is a `n` element vector, a `m` element vector is returned.

If A is a `n` element vector, and B is a `n` x `p` matrix, a `p` element vector is returned.

Example:

```
Matrix<float> Z, A, B;
Z = prod(A, B);
```

See Also: `prodh`(section 4.4.13)`prodj`(section 4.4.14)`prodt`(section 4.4.15)

4.4.11. `prod3`

Description: Linear algebra 3x3 product.

Syntax:

```
Matrix<T> prod3 (  Matrix<T> A ,  Matrix<T> B  );

Vector<T> prod3 (  Matrix<T> A ,  Vector<T> B  );
```

Requirements: All matrix arguments must be size 3 by 3. All vector arguments must be of size 3.

Result: The result is equal to the linear algebraic product of the arguments.

If A is a 3 by 3 matrix and B is a 3 by 3 matrix, a 3 x 3 matrix is returned.

If A is a 3 by 3 matrix, and B is a 3 element vector, a 3 element vector is returned.

Example:

```
Matrix<float> Z(3, 3), A(3, 3), B(3, 3);
Z = prod(A, B);
```

See Also: `prod` (section 4.4.10) `prod4` (section 4.4.12)

4.4.12. `prod4`

Description: Linear algebra 4x4 product.

Syntax:

```
Matrix<T> prod4 (  Matrix<T> A ,  Matrix<T> B  );

Vector<T> prod4 (  Matrix<T> A ,  Vector<T> B  );
```

Requirements: All matrix arguments must be size 4 by 4. All vector arguments must be of size 4.

Result: The result is equal to the linear algebraic product of the arguments.

If A is a 4 by 4 matrix and B is a 4 by 4 matrix, a 4 by 4 matrix is returned.

If A is a 4 by 4 matrix, and B is a 4 element vector, a 4 element vector is returned.

Example:

```
Matrix<float> Z(4, 4), A(4, 4), B(4, 4);
Z = prod(A, B);
```

See Also: `prod` (section 4.4.10) `prod4` (section 4.4.12)

4.4.13. `prodh`

Description: Linear algebra product, with hermetian.

Syntax:

```
Matrix<T> prodh (  Matrix<T> A ,  Matrix<T> B  );
```

Requirements: The number of columns of A must equal the number of columns of B.

Result: The result is equal to the linear algebraic product of the first argument with the hermetian (conjugate transpose) of the second argument. `Z = prodh(A, B) = prod(A, conj(trans(B)))`

If A is a `m` by `n` matrix and B is a `p` by `n` matrix, a `m` x `p` matrix is returned.

Example:

```
length_type m = 64, n = 32, p = 48;
Matrix<float> Z(m, p), A(m, n), B(p, n);
Z = prodh(A, B);
```

See Also: `prod` (section 4.4.10) `prodj` (section 4.4.14) `prodt` (section 4.4.15)

4.4.14. `prodj`

Description: Linear algebra product, with conjuage.

Syntax:

`Matrix<T>` **`prodj`** `(` *`Matrix<T> A`* `,` *`Matrix<T> B`* `);`

Requirements: The number of columns of `A` must equal the number of rows of `B`.

Result: The result is equal to the linear algebraic product of the first argument with the conjugate of the second argument. `Z = prodj(A, B) = prod(A, conj(B))`

If A is a `m` by `n` matrix and B is a `n` by `p` matrix, a `m` x `p` matrix is returned.

Example:

```
length_type m = 64, n = 32, p = 48;
Matrix<float> Z(m, p), A(m, n), B(n, p);
Z = prodj(A, B);
```

See Also: `prod`(section 4.4.10)`prodh`(section 4.4.13)`prodt`(section 4.4.15)

4.4.15. `prodt`

Description: Linear algebra product, with transpose.

Syntax:

`Matrix<T>` **`prodt`** `(` *`Matrix<T> A`* `,` *`Matrix<T> B`* `);`

Requirements: The number of columns of `A` must equal the number of rows of `B`.

Result: The result is equal to the linear algebraic product of the first argument with the transpose of the second argument.

If A is a `m` by `n` matrix and B is a `p` by `n` matrix, a `m` x `p` matrix is returned.

Example:

```
length_type m = 64, n = 32, p = 48;
Matrix<float> Z(m, p), A(m, n), B(p, n);
Z = prod(A, B);
```

See Also: `prod`(section 4.4.10)`prodh`(section 4.4.13)`prodj`(section 4.4.14)

4.4.16. `trans`

Description: Matrix transpose.

Syntax:

`Matrix<T>` **`trans`** `(` *`Matrix<T> A`* `);`

Result: Returns a matrix transpose view. The `i, j`th element of the result are the `j, i`th element of the argument. result(i, j) = A(j, i)

Note that `trans()` does itself not rearrange data in memory. However, if the result is assigned to a destination matrix, during the copy into the destination, a corner-turn may be performed.

Example:

```
length_type m = 32, n = 16;
Matrix<float> Z(m, n), A(n, m);
Z = trans(A);
```

See Also: `herm` (section 4.4.6)

4.5. Linear System Solvers

These functions and classes solve linear systems and also perform singular value decomposition.

4.5.1. Cholesky Decomposition Solver

This section describes the Cholesky decomposition processing object provided by VSIPL++.

4.5.1.1. Class template chold<>

The class `chold` uses Cholesky decomposition to solve linear systems.

```
template <typename T,
          return_mechanism_type ReturnMechanism = by_value>
class chold;
```

Template parameters

`T`	The value type used for the decomposition object. May be single- or double-precision floating-point, and either real or complex.
`ReturnMechanism`	The return mechanism type indicates whether the Cholesky decomposition object's `solve()` function returns results by value (`by_value`) or by reference (`by_reference`) into matrices provided by the caller. See section 2.7.7 for details.

4.5.1.2. Constructor

```
chold(mat_uplo uplo, length_type len);
```

Description: Constructs a `chold` object that will decompose *`len`* by *`len`* positive definite matrices. The parameter *`uplo`* controls whether an upper LU or lower LU decomposition will be

performed (see section 2.7.6). Note also that `chold` objects may also be copied (constructed) from other `chold` objects.

Requirements: The parameter *`len`* must be greater than zero.

4.5.1.3. Accessor functions

```
mat_uplo uplo() const;
length_type length() const;
```

Description: Report the attributes of this `chold` object. The `length()` function returns the length, equal to the number of rows as well as the number of columns in the decomposed matrix. The `uplo()` function indicates whether the lower half or upper half of a decomposed matrix is referenced.

4.5.1.4. Solve Systems

```
template <typename Block>
bool
decompose(Matrix<T, Block> A);
```

Description: Performs Cholesky decomposition of *A*. When it can be used, Cholesky decomposition is twice as efficient as normal LU decomposition.

Requirements: The matrix *A* must be the same size as specified in the constructor. Note that the contents may be overwritten, therefore *A* should not be modified until all `solve()` calls have been performed. The matrix *A* must be both symmetric and positive definite for Cholesky decomposition to work.

Result: False is returned if the decomposition fails.

4.5.1.5. Solve Systems (by_value)

This function is available only if the `chold` class template is parameterized with `ReturnMechanism=by_value`.

```
template <mat_op_type tr,
          typename    Block>
const_Matrix<T, unspecified>
solve(const_Matrix<T, Block> B);
```

Description: Solves a linear system.

The parameter `tr` controls what type of operation is performed on *A* when solving the system (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $A^T X = B$ is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $A^H X = B$ is solved.

Otherwise, $AX = B$ is solved.

Requirements: The number of rows of *B* must be equal to the value returned by `length()`. A successful call to `decompose()` must have occurred.

Result: Returns the solution to the linear system. The returned matrix's block type may be a different type from `Block`.

4.5.1.6. Solve Systems (by_reference)

This function is available only if the `chold` class template is parameterized with `ReturnMechanism=by_reference`.

```
template <mat_op_type tr,
          typename    Block0,
          typename    Block1>
bool
solve(
  const_Matrix<T, Block0> B,
  Matrix<T, Block1>       X);
```

Description: Solves a linear system.

The parameter `tr` controls what type of operation is performed on *A* when solving the system (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $A^T X = B$ is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $A^H X = B$ is solved.

Otherwise, $AX = B$ is solved.

Requirements: The number of rows of both *B* and *X* must be equal to the value returned by `length()`. A successful call to `decompose()` must have occurred.

Result: Stores the solution in *X*. True is returned if the computation succeeds.

4.5.2. Covariance Solver

This section describes the covariance linear system solver function provided by VSIPL++.

4.5.2.1. Solve Systems (return by value)

```
template <typename T,
          typename Block0,
          typename Block1>
Matrix<T, unspecified>
covsol(
  Matrix<T, Block0>       A,
  const_Matrix<T, Block1> B);
```

Description: Solves the covariance linear system $A^T AX = B$ for *X* if type `T` is real and $A^H AX = B$ for *X* if type `T` is complex.

Requirements: The matrix *A* is of size `M` by `N` (where `M >= N`) and is of rank `N`. The matrix *B* is of size `N` by `K`. The type `T` may be single- or double-precision floating-point, and either real or complex. Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines.

Result: The solution `X` is returned and is of size `N` by `K`. Note that *A* may be overwritten. The returned matrix's block type may be a different type from `Block0` or `Block1`.

4.5.2.2. Solve Systems (return by reference)

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2>
Matrix<T, unspecified>
covsol(
  Matrix<T, Block0>       A,
  const_Matrix<T, Block1> B,
  Matrix<T, Block2>       X);
```

Description: Solves the covariance linear system $A^TAX = B$ for *X* if type T is real and $A^HAX = B$ for *X* if type T is complex.

Requirements: The matrix *A* is of size M by N (where M >= N) and is of rank N. The matrix *B* is of size N by K. The matrix *X* is also of size N by K. The type T may be single- or double-precision floating-point, and either real or complex. Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines.

Result: The solution is placed in *X*. Note that *A* may be overwritten.

4.5.3. Linear Least Squares Solver

This section describes the linear least squares solver function provided by VSIPL++.

4.5.3.1. Solve Systems (return by value)

```
template <typename T,
          typename Block0,
          typename Block1>
Matrix<T, unspecified>
llsqsol(
  Matrix<T, Block0>       A,
  const_Matrix<T, Block1> B);
```

Description: Solves the linear least squares problem $\min_X ||AX - B||_2$ for *X*.

Requirements: The matrix *A* is of size M by N (where M >= N) and is of rank N. The matrix *B* is of size M by K. The type T may be single- or double-precision floating-point, and either real or complex. Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines.

Result: Returns the solution X which is of size N by K. Note that *A* may be overwritten. The returned matrix's block type may be a different type from Block0 or Block1.

4.5.3.2. Solve Systems (return by reference)

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2>
Matrix<T, Block2>
llsqsol(
  Matrix<T, Block0>       A,
```

```
  const_Matrix<T, Block1> B,
  Matrix<T, Block2>       X);
```

Description: Solves the linear least squares problem $\min_X ||AX - B||_2$ for X.

Requirements: The matrix A is of size M by N (where M >= N) and is of rank N. The matrix B is of size M by K. The matrix X is of size N by K. The type T may be single- or double-precision floating-point, and either real or complex. Temporary workspace may be allocated, which may result in nondeterministic execution time. As an alternative, use the QR routines.

Result: Stores the solution in X and returns it. Note that A may be overwritten.

4.5.4. LU Decomposition Solver

This section describes the LU (lower and upper triangular) decomposition processing object provided by VSIPL++.

4.5.4.1. Class template lud<>

The class lud performs LU decomposition to solve linear systems.

```
template <typename T,
          return_mechanism_type ReturnMechanism = by_value>
class lud;
```

Template parameters

T	The value type used for the decomposition object. May be real or complex, single- or double-precision floating-point types.
ReturnMechanism	The return mechanism type indicates whether the LU decomposition object's solve() function returns results by value (by_value) or by reference (by_reference) into matrices provided by the caller. See section 2.7.7 for details.

4.5.4.2. Constructor

```
lud(length_type len);
```

Description: Constructs an lud object that will decompose *len* by *len* matrices. Note also that lud objects may also be copied (constructed) from other lud objects.

Requirements: The parameter *len* must be greater than zero.

4.5.4.3. Accessor functions

```
length_type length() const;
```

Description: Report the length attribute of this lud object, equal to the number of rows as well as the number of columns in the decomposed matrix.

4.5.4.4. Solve Systems

```
template <typename Block>
bool
decompose(Matrix<T, Block> A);
```

Description: Performs LU decomposition of *A*.

Requirements: The matrix *A* must be the same size as specified in the constructor. Note that the contents may be overwritten, therefore *A* should not be modified until all `solve()` calls have been performed.

Result: False is returned if the decomposition fails.

4.5.4.5. Solve Systems (by_value)

This function is available only if the `lud` class template is parameterized with `ReturnMechanism=by_value`.

```
template <mat_op_type tr,
          typename    Block>
const_Matrix<T, unspecified>
solve(const_Matrix<T, Block> B);
```

Description: Solves a linear system. The parameter `tr` controls what type of operation is performed on *A* when solving the system (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $A^T X = B$ is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $A^H X = B$ is solved.

Otherwise, $AX = B$ is solved.

Requirements: The number of rows of *B* must be equal to the value returned by `length()`. A successful call to `decompose()` must have occurred.

Result: Returns the solution to the linear system.

4.5.4.6. Solve Systems (by_reference)

This function is available only if the `lud` class template is parameterized with `ReturnMechanism=by_reference`.

```
template <mat_op_type tr,
          typename    Block0,
          typename    Block1>
bool
solve(
  const_Matrix<T, Block0> B,
  Matrix<T, Block1>       X);
```

Description: Solves a linear system. The parameter `tr` controls what type of operation is performed on *A* when solving the system (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $A^T X = B$ is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $A^H X = B$ is solved.

Otherwise, $AX = B$ is solved.

Requirements: The number of rows of both *B* and *X* must be equal to the value returned by `length()`. A successful call to `decompose()` must have occurred.

Result: Stores the solution in *X*. True is returned if the computation succeeds.

Exceptions: If the backends enabled do not support the requested LU decomposition, a `vsip::impl::unimplemented` exception will be thrown. This is a deviation from the VSIPL++ spec.

4.5.5. QR Decomposition Solver

This section describes the QR decomposition processing object provided by VSIPL++.

4.5.5.1. Class template qrd<>

The class `qrd` performs QR decomposition and solves linear systems.

```
template <typename T,
          return_mechanism_type ReturnMechanism = by_value>
class qrd;
```

Template parameters

`T`	The value type used for the decomposition object. May be real or complex, single- or double-precision floating-point types.
`ReturnMechanism`	The return mechanism type indicates whether the QR solver and product functions return results by value (`by_value`) or by reference (`by_reference`) into matrices provided by the caller. See section 2.7.7 for details.

4.5.5.2. Constructor

```
qrd(length_type rows, length_type columns, storage_type st);
```

Description: Constructs a `qrd` object. The parameters *`rows`* and *`columns`* refer to the size of the `Q` matrix. The parameter *`st`* controls how much of the `Q` matrix is stored (see section 2.7.8). Note also that `qrd` objects may also be copied (constructed) from other `qrd` objects.

Requirements: The number of rows must be greater than or equal to the number of columns.

4.5.5.3. Accessor functions

```
length_type rows() const;
length_type columns() const;
storage_type qstorage() const;
```

Description: Report the various attributes of this `qrd` object. The number of rows is returned by `rows()` and the number of columns by `columns()`. The function `st()` returns how the decomposition matrix `Q` is stored by this object, if at all.

4.5.5.4. Solve Systems

```
template <typename Block>
bool
decompose(Matrix<T, Block> A);
```

Description: Performs a QR decomposition of the matrix *`A`* into matrices `Q` and `R`.

Requirements: The matrix *A* must be the same size as specified in the constructor. Note that the contents may be overwritten therefore *A* should not be modified prior to calling any other function.

Result: False is returned if the decomposition fails because *A* does not have full column rank. Note: If T is a specialization of complex, Q is unitary. Otherwise, Q is orthogonal. R is an upper triangular matrix. If *A* has full rank, then R is a nonsingular matrix. No column interchanges are performed.

4.5.5.5. Solve Systems (by_value)

This function is available only if the `qrd` class template is parameterized with `ReturnMechanism=by_value`

```
template <mat_op_type       tr,
          product_side_type ps,
          typename          Block>
const_Matrix<T, unspecified>
prodq(const_Matrix<T, Block> C);
```

Description: Calculates the product of Q and *C*. The parameter `tr` controls what type of operation is performed on *C* before the product is computed (see section 2.7.3) and `ps` determines what side of the product Q is placed on (see section 2.7.5). The actual product and its number of rows and columns (shown in the table below) depends on the values of `tr`, `ps`, and `qstorage()` and whether T is not or is a specialization of complex.

For `qstorage() == qrd_saveq1`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	QC, rows(), s	CQ, s, columns()
tr == mat_trans, T	Q^TC, columns(), s	CQ^T, s, rows()
tr == mat_herm, complex<T>	Q^HC, columns(), s	CQ^H, s, rows()

where *s* is an arbitrary positive value.

For `qstorage() == qrd_saveq`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	QC, rows(), s	CQ, s, rows()
tr == mat_trans, T	Q^TC, rows(), s	CQ^T, s, rows()
tr == mat_herm, complex<T>	Q^HC, rows(), s	CQ^H, s, rows()

Requirements: A successful call to `decompose()` must have occurred for this object with `qstorage()` equaling either `qrd_saveq1` or `qrd_saveq`. Otherwise, the behavior is undefined. The number of rows and columns (shown in the table below) of *C* depend on the values of `tr`, `ps`, and `qstorage()`.

For `qstorage() == qrd_saveq1`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	columns(), s	s, rows()
tr == mat_trans	rows(), s	s, columns()

	ps == mat_lside	**ps == mat_rside**
tr == mat_herm	rows(), s	s, columns()

where *s* is the same variable as above.

For `qstorage() == qrd_saveq`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, rows()
tr == mat_trans	rows(), s	s, rows()
tr == mat_herm	rows(), s	s, rows()

Result: Returns the product of `Q` and *C*. The returned matrix's block type may be a different type from `Block`.

```
template <mat_op_type tr,
          typename    Block>
const_Matrix<T, unspecified>
rsol(
  const_Matrix<T, Block> B,
  T const                alpha);
```

Description: Solves a linear system for *X*. The parameter `tr` controls what type of operation is performed on *R* before the system is solved (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $R^T X$ `= alpha B` is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $R^H X$ `= alpha B` is solved.

Otherwise, `RX = alpha B` is solved.

Requirements: The number of rows in *B* must be equal to the value returned by `columns()`. A successful call to `decompose()` must have occurred.

Result: Returns a constant matrix *X* containing the solution.

```
template <typename Block>
const_Matrix<T, unspecified>
covsol(const_Matrix<T, Block> B);
```

Description: Solves a covariance linear system for *X*.

If `T` is not a specialization of complex, then $A^T A X$ `= B` is solved, where `A` is the matrix given to the most recent call to `decompose()`.

If `T` is a specialization of complex, then $A^H A X$ `= B` is solved.

Requirements: The number of rows in *B* must be equal to the value returned by `columns()`. Note also that *X* and *B* are element-conformant

Result: Returns a matrix *X* containing the solution.

```
template <typename Block>
Matrix<T, unspecified>
lsqsol(const_Matrix<T, Block> B)
```

Description: Solves the linear least squares problem $\min_X ||AX - B||_2$ for X, where A is the matrix given to the most recent call to `decompose()`.

Requirements: The number of rows in B must be equal to the value returned by `rows()`. The number of rows in X will equal the value returned by `columns()`.

Result: Returns a constant matrix X containing the solution.

4.5.5.6. Solve Systems (by_reference)

This function is available only if the `qrd` class template is parameterized with `ReturnMechanism=by_reference`

```
template <mat_op_type       tr,
          product_side_type ps,
          typename          Block0,
          typename          Block1>
bool
prodq(
  const_Matrix<T, Block0> C,
  Matrix<T, Block1>       X);
```

Description: Calculates the product of `Q` and C. The parameter `tr` controls what type of operation is performed on C before the product is computed (see section 2.7.3) and `ps` determines what side of the product `Q` is placed on (see section 2.7.5). The actual product depends on the values of `tr`, and whether `T` is not or is a specialization of complex:

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	QC	CQ
tr == mat_trans, T	$Q^T C$	CQ^T
tr == mat_herm, complex<T>	$Q^H C$	CQ^H

Requirements: A successful call to `decompose()` must have occurred for this object with `qstorage()` equaling either `qrd_saveq1` or `qrd_saveq`. Otherwise, the behavior is undefined. The number of rows and columns (shown in the table below) of C depend on the values of `tr`, `ps`, and `qstorage()`.

For `qstorage() == qrd_saveq1`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	columns(), s	s, rows()
tr == mat_trans	rows(), s	s, columns()
tr == mat_herm	rows(), s	s, columns()

where s is an arbitrary positive value.

For `qstorage() == qrd_saveq`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, rows()
tr == mat_trans	rows(), s	s, rows()
tr == mat_herm	rows(), s	s, rows()

The number of rows and columns of *X* (shown in the table below) depend on the values of `tr`, `ps`, and `qstorage()`.

For `qstorage() == qrd_saveq1`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, columns()
tr == mat_trans	columns(), s	s, rows()
tr == mat_herm	columns(), s	s, rows()

where *s* is the same variable as above.

For `qstorage() == qrd_saveq`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, rows()
tr == mat_trans	rows(), s	s, rows()
tr == mat_herm	rows(), s	s, rows()

Result: Calculates the product of `Q` and *C* stores it in *X*.

```
template <mat_op_type tr,
          typename    Block0,
          typename    Block1>
bool
rsol(
  const_Matrix<T, Block0> B,
  T const                 alpha,
  Matrix<T, Block1>       X);
```

Description: Solves a linear system for *X*. The parameter `tr` controls what type of operation is performed on *R* before the system is solved (see section 2.7.3).

If `tr == mat_trans` and `T` is not a specialization of complex, then $R^T X$ `= alpha B` is solved.

If `tr == mat_herm` and `T` is a specialization of complex, then $R^H X$ `= alpha B` is solved.

Otherwise, `RX = alpha * B` is solved.

Requirements: The number of rows in *B* must be equal to the value returned by `columns()`. A successful call to `decompose()` must have occurred.

Result: Stores the solution in *X*.

```
template <typename Block0,
          typename Block1>
bool
```

```
covsol(
  const_Matrix<T, Block0> B,
  Matrix<T, Block1>       X);
```

Description: Solves a covariance linear system for *X*. If `T` is not a specialization of complex, then $A^T AX = B$ is solved, where `A` is the matrix given to the most recent call to `decompose()`. If `T` is a specialization of complex, then $A^H AX = B$ is solved.

Requirements: The number of rows in *B* must be equal to the value returned by `columns()`. Note also that *X* is modifiable and element-conformant with *B*.

Result: The solution is stored in *X*

```
template <typename Block0,
          typename Block1>
bool
lsqsol(
  const_Matrix<T, Block0> B,
  Matrix<T, Block1>       X)   \
```

Description: Solves the linear least squares problem $\min_X ||AX - B||_2$ for *X*, where `A` is the matrix given to the most recent call to `decompose()`.

Requirements: The number of rows in *B* must be equal to the value returned by `rows()`. The number of rows in *X* must equal the value returned by `columns()`.

Result: Stores the solution in the matrix *X*.

4.5.6. Singular Value Decomposition

This section describes the singular value decomposition processing object provided by VSIPL++.

4.5.6.1. Class template svd<>

The class `svd` performs singular value decomposition to decompose a matrix into orthogonal or unitary matrixes and singular values.

```
template <typename T,
          return_mechanism_type ReturnMechanism = by_value>
class svd;
```

Template parameters

`T`	The value type used for the decomposition object. May be real or complex, single- or double-precision floating-point types.
`ReturnMechanism`	The return mechanism type indicates whether the SV decomposition and product functions return results by value (`by_value`) or by reference (`by_reference`) into matrices provided by the caller. See section 2.7.7 for details.

4.5.6.2. Constructor

```
svd(
  length_type  rows,
  length_type  columns,
```

```
  storage_type ustorage,
  storage_type vstorage);
```

Description: Constructs an `svd` object. The parameters *`rows`* and *`columns`* refer to the size of the matrix to be decomposed. The parameters *`ustorage`* and *`vstorage`* control how much of the `U` and `V` matrices are stored, respectively (see section 2.7.8). Note also that `svd` objects may also be copied (constructed) from other `svd` objects.

Requirements: The number of rows and columns must both be positive.

4.5.6.3. Accessor functions

```
length_type rows() const;
length_type columns() const;
storage_type ustorage() const;
storage_type vstorage() const;
```

Description: Report the various attributes of this `svd` object. The number of rows is returned by `rows()` and the number of columns by `columns()`. The function `ustorage()` returns how the decomposition matrix `U` is stored by this object, if at all. The function `vstorage()` returns how the decomposition matrix V^T or V^H is stored by this object, if at all.

4.5.6.4. Solve Systems (by_value)

This function is available only if the `svd` class template is parameterized with `ReturnMechanism=by_value`

```
template <typename Block>
const_Vector<T, unspecified>
decompose(Matrix<T, Block> A);
```

Description: Performs a singular value decomposition of the matrix *`A`* containing `M` rows and `N` columns.

If T is not a specialization of complex, $A = U\ SV^T$, where square orthogonal matrix `U` has the same number of rows as *`A`*, `S` is a matrix with the same shape as *`A`* and all zero values except its first `p` diagonal elements are real, nonincreasing, nonnegative values, and V is a square orthogonal matrix with the same number of columns as *`A`*. The number of diagonal elements are given by `p = min(M, N)`.

If T is a specialization of complex, $A = U\ SV^H$, where `U`, `S`, and `V`, are similar to those described above except `U` and `V` are unitary, not orthogonal, matrices.

Requirements: The matrix *`A`* must be the same size as specified in the constructor. Note that the contents may be overwritten therefore *`A`* should not be modified prior to calling any other function.

Result: Returns a vector of length p containing the sinugular values of *`A`* in non-increasing order. Note that memory may be allocated. The returned vector's block type may be a different type from `Block`.

```
template <mat_op_type       tr,
          product_side_type ps,
          typename          Block>
const_Matrix<T, unspecified>
produ(const_Matrix<T, Block> C) const;
```

Description: Calculates the product of `U` and *C*. The parameter `tr` controls what type of operation is performed on *C* before the product is computed (see section 2.7.3) and `ps` determines what side of the product `U` is placed on (see section 2.7.5). The actual product and its number of rows and columns (shown in the table below) depends on the values of `tr`, `ps`, and `ustorage()` and whether `T` is not or is a specialization of complex.

For `ustorage() == qrd_uvpart`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	UC, columns(), s	CU, s, p
tr == mat_trans, T	U^TC, p, s	CU^T, s, columns()
tr == mat_herm, complex<T>	U^HC, p, s	CU^H, s, columns()

where *s* is an arbitrary positive value and `p = min(M, N)`.

For `ustorage() == svd_uvfull`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	UC, columns(), s	CU, s, columns()
tr == mat_trans, T	U^TC, columns(), s	CU^T, s, columns()
tr == mat_herm, complex<T>	U^HC, columns(), s	CU^H, s, columns()

Requirements: A successful call to `decompose()` must have occurred for this object with `ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. Otherwise, the behavior is undefined. The number of rows and columns (shown in the table below) of *C* depend on the values of `tr`, `ps`, and `ustorage()`.

For `ustorage() == svd_uvpart`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	p, s	s, columns()
tr == mat_trans	columns(), s	s, p
tr == mat_herm	columns(), s	s, p

where *s* is an arbitrary positive value and `p = min(M, N)`.

For `ustorage() == svd_uvfull`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	columns(), s	s, columns()
tr == mat_trans	columns(), s	s, columns()
tr == mat_herm	columns(), s	s, columns()

Result: Returns the product of `U` and *C*. The returned matrix's block type may be a different type from `Block`.

```
template <mat_op_type       tr,
          product_side_type ps,
          typename          Block>
```

```
const_Matrix<T, unspecified>
prodv(const_Matrix<T, Block> C) const;
```

Description: Calculates the product of V and *C*. The parameter tr controls what type of operation is performed on *C* before the product is computed (see section 2.7.3) and ps determines what side of the product V is placed on (see section 2.7.5). The actual product and its number of rows and columns (shown in the table below) depends on the values of tr, ps, and vstorage() and whether T is not or is a specialization of complex.

For vstorage() == qrd_uvpart,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	VC, columns(), s	CV, s, p
tr == mat_trans, T	$V^T C$, p, s	CV^T, s, columns()
tr == mat_herm, complex<T>	$V^H C$, p, s	CV^H, s, columns()

where *s* is an arbitrary positive value and p = min(M, N).

For vstorage() == svd_uvfull,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	VC, columns(), s	CV, s, columns()
tr == mat_trans, T	$V^T C$, columns(), s	CV^T, s, columns()
tr == mat_herm, complex<T>	$V^H C$, columns(), s	CV^H, s, columns()

Requirements: A successful call to decompose() must have occurred for this object with vstorage() equaling either svd_uvpart or svd_uvfull. Otherwise, the behavior is undefined. The number of rows and columns (shown in the table below) of *C* depend on the values of tr, ps, and vstorage().

For vstorage() == svd_uvpart,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	p, s	s, columns()
tr == mat_trans	columns(), s	s, p
tr == mat_herm	columns(), s	s, p

where *s* is an arbitrary positive value and p = min(M, N).

For vstorage() == svd_uvfull,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	columns(), s	s, columns()
tr == mat_trans	columns(), s	s, columns()
tr == mat_herm	columns(), s	s, columns()

Result: Returns the product of V and *C*. The returned matrix's block type may be a different type from Block.

```
const_Matrix<T, unspecified>
u(
  index_type low,
  index_type high) const;
```

Description: Returns consecutive columns in the matrix U from a singular value decomposition.

Requirements: A successful call to `decompose()` must have occurred with `ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. Otherwise, the behavior is undefined.

Result: Returns the constant matrix U containing columns low, low+1, ..., high, inclusive.

```
const_Matrix<T, unspecified>
v(
  index_type low,
  index_type high) const;
```

Description: Returns consecutive columns in the matrix V from a singular value decomposition.

Requirements: A successful call to `decompose()` must have occurred with `vstorage()` equaling either `svd_uvpart` or `svd_uvfull`. Otherwise, the behavior is undefined.

Result: Returns the constant matrix V containing columns low, low+1, ..., high, inclusive.

4.5.6.5. Solve Systems (by_reference)

This function is available only if the `svd` class template is parameterized with `ReturnMechanism=by_reference`

```
template <typename Block0,
          typename Block1>
bool
decompose(
  Matrix<T, Block0> A,
  Vector<T, Block1> x);
```

Description: Performs a singular value decomposition of the matrix *A* containing `M` rows and `N` columns.

If T is not a specialization of complex, $A = U\ SV^T$, where square orthogonal matrix U has the same number of rows as *A*, S is a matrix with the same shape as *A* and all zero values except its first p diagonal elements are real, nonincreasing, nonnegative values, and V is a square orthogonal matrix with the same number of columns as *A*. The number of diagonal elements are given by `p = min(M, N)`.

If T is a specialization of complex, $A = U\ SV^H$, where U, S, and V, are similar to those described above except U and V are unitary, not orthogonal, matrices.

Requirements: The matrix *A* may be overwritten. It must be the same size as specified in the constructor. The vector *x* must be of length p.

Result: Returns true if the decomposition succeeds. The vector *x* is filled with the sinugular values of *A* in non-increasing order. Note that memory may be allocated.

```
template <mat_op_type       tr,
          product_side_type ps,
```

```
          typename          Block0,
          typename          Block1>
bool
produ(
  const_Matrix<T, Block0> C,
  Matrix<T, Block1>       X) const;
```

Description: Calculates the product of `U` and *`C`*. The parameter `tr` controls what type of operation is performed on *`C`* before the product is computed (see section 2.7.3) and `ps` determines what side of the product `U` is placed on (see section 2.7.5). The actual product (shown in the table below) depends on the values of `tr`, `ps`, and whether `T` is not or is a specialization of complex.

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	UC	CU
tr == mat_trans, T	$U^{T}C$	CU^{T}
tr == mat_herm, complex<T>	$U^{H}C$	CU^{H}

Requirements: A successful call to `decompose()` must have occurred for this object with `ustorage()` equaling either `svd_uvpart` or `svd_uvfull`. Otherwise, the behavior is undefined. The number of rows and columns (shown in the table below) of *`C`* depend on the values of `tr`, `ps`, and `ustorage()`.

For `ustorage() == svd_uvpart`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	p, s	s, rows()
tr == mat_trans	rows(), s	s, p
tr == mat_herm	rows(), s	s, p

where *`s`* is an arbitrary positive value and `p = min(M, N)`.

For `ustorage() == svd_uvfull`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, rows()
tr == mat_trans	rows(), s	s, rows()
tr == mat_herm	rows(), s	s, rows()

The number of rows and columns of *`X`* (shown in the table below) depend on the values of `tr`, `ps`, and `qstorage()`.

For `ustorage() == svd_uvpart`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, p
tr == mat_trans	p, s	s, rows()
tr == mat_herm	p, s	s, rows()

where *`s`* is the same variable as above.

For `ustorage() == svd_uvfull`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	rows(), s	s, rows()
tr == mat_trans	rows(), s	s, rows()
tr == mat_herm	rows(), s	s, rows()

Result: Returns true if the product succeeds. The product of `U` and *C* is stored in *X*.

```
template <mat_op_type       tr,
          product_side_type ps,
          typename          Block0,
          typename          Block1>
bool
prodv(
  const_Matrix<T, Block0> C,
  Matrix<T, Block1>       X) const;
```

Description: Calculates the product of `V` and *C*. The parameter `tr` controls what type of operation is performed on *C* before the product is computed (see section 2.7.3) and `ps` determines what side of the product `V` is placed on (see section 2.7.5). The actual product (shown in the table below) depends on the values of `tr`, `ps`, and whether `T` is not or is a specialization of complex.

For `vstorage() == qrd_uvpart`,

	ps == mat_lside	**ps == mat_rside**
tr == mat_ntrans	VC	CV
tr == mat_trans, T	$V^{T}C$	CV^{T}
tr == mat_herm, complex<T>	$V^{H}C$	CV^{H}

```
template <typename Block>
bool
u(
  index_type       low,
  index_type       high
  Matrix<T, Block> X) const;
```

Description: Returns consecutive columns in the matrix U from a singular value decomposition.

Requirements: A successful call to `decompose()` must have occurred with `ustorage()` equaling either `svd_uvpart` or `svd_uvfull`.

Result: Returns true if the matrix is stored. Stores the constant matrix `U` containing columns low, low+1, ..., high, inclusive in *X*.

```
template <typename Block>
bool
v(
  index_type       low,
  index_type       high
  Matrix<T, Block> X) const;
```

Description: Returns consecutive columns in the matrix V from a singular value decomposition.

Requirements: A successful call to `decompose()` must have occurred with `vstorage()` equaling either `svd_uvpart` or `svd_uvfull`.

Result: Returns true if the matrix is stored. Stores the constant matrix `V` containing columns low, low+1, ..., high, inclusive in *X*.

4.5.7. Toeplitz Solver

This section describes the Toeplitz linear system solver function provided by VSIPL++.

4.5.7.1. Solve Systems (return by value)

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2>
const_Vector<T, unspecified>
toepsol(
  const_Vector<T, Block0> a,
  const_Vector<T, Block1> b,
  Vector<T, Block2>       w);
```

Description: Solve a real symmetric or complex Hermitian positive definite Toeplitz linear system `Ax = b`, where *a* specifies the Toeplitz matrix `A`. The Toeplitz linear system is real symmetric if type `T` is real and is Hermitian if type `T` is complex.

Requirements: The N by N Toeplitz matrix formed from *a* must have full rank and be positive definite. The sizes of *a*, *b* and *w* are equal to N. The type `T` may be single- or double-precision floating-point, and either real or complex. The vector *w* is used as a temporary workspace.

Result: The solution `x` is returned. The returned matrix's block type may be a different type from `Block0`, `Block1` or `Block2`.

4.5.7.2. Solve Systems (return by reference)

```
template <typename T,
          typename Block0,
          typename Block1,
          typename Block2,
          typename Block3>
const_Vector<T, Block3>
toepsol(
  const_Vector<T, Block0> a,
  const_Vector<T, Block1> b,
  Vector<T, Block2>       w,
  Vector<T, Block3>       x);
```

Description: Solve a real symmetric or complex Hermitian positive definite Toeplitz linear system `Ax = b`, where *a* specifies the Toeplitz matrix `A`. The Toeplitz linear system is real symmetric if type `T` is real and is Hermitian if type `T` is complex.

Requirements: The N by N Toeplitz matrix formed from *a* must have full rank and be positive definite. The sizes of *a*, *b*, *w* and *x* are equal to N. The type `T` may be single- or double-precision floating-point, and either real or complex. The vector *w* is used as a temporary workspace.

Result: The solution is stored in *x* and returned.

4.6. Selection, generation, and manipulation functions

4.6.1. Generation functions

4.6.1.1. Random Number Generation

This section describes the `Rand` class provided by VSIPL++.

A `Rand` object provides member functions to generate scalar random numbers and views of random numbers.

```
template <typename T>
class Rand;
```

Template parameters

T The type of the random numbers generated by this class.

`Rand` objects may not be assigned or copied.

4.6.1.1.1. Public types

```
typedef Vector<T> vector_type;
typedef Matrix<T> matrix_type;
typedef Tensor<T> tensor_type;
```

Description: These types specify the return values for the non-scalar number generators.

4.6.1.1.2. Constructors

```
Rand(
  index_type seed
  index_type numprocs
  index_type id,
  bool portable = true);
```

Requires: 0 < `id` <= `numprocs` <= $2^{31} - 1$

Description: See [VSPEC101] section *Random number generation* for more information and this quote. “Constructs a random number generator object using the specified seed `seed` . If `portable` == false, the random number generator characteristics are implementation defined. Otherwise, the random number generator object obeys the VSIPL 1.1 API and guidelines in VSIPL 1.1 API sections “Random Numbers,” “VSIPL Random Number Generator Functions,” and “Sample Implementation.”” When `portable` == `false`, the implementation may select an algorithm that yields performance better than the portable method.

```
Rand(index_type seed, bool portable = true);
```

Description: Using this constructor is short-hand for using the previous constructor as follows:

```
Rand(seed, 1, 1, portable);
```

4.6.1.1.3. Number Generators

```
        T randu()
        T randn()
```

Description: Return scalar random numbers.

```
        const_Vector<T, unspecified> randu(length_type len)
        const_Vector<T, unspecified> randn(length_type len)
```

Description: Construct and return vectors of random numbers.

```
        const_Matrix<T, unspecified> randu(
            length_type rows,
            length_type columns)
        const_Matrix<T, unspecified> randn(
            length_type rows,
            length_type columns)
```

Description: Construct and return matrices of random numbers.

```
        const_Tensor<T, unspecified> randu(
            length_type z,
            length_type y,
            length_type x)
        const_Tensor<T, unspecified> randn(
            length_type z,
            length_type y,
            length_type x)
```

Description: Construct and return tensors of random numbers.

4.6.1.1.4. Example

```
// Construct a random number generator for floats.
Rand<float> vgen(0, 0);

// Create a vector of 35 uniform random numbers.
Rand<float>::vector_type v1 = vgen.randu(35);
```

4.6.1.2. ramp()

```
    template <typename T>
    const_Vector<T, unspecified >
    ramp(T a, T b, length_type len);
    \
```

Returns: A Vector of size `len`. For `0 <= i < len, w.get(i) == a + i * b`.

4.6.2. Selection functions

4.6.2.1. first()

```
template <typename Predicate, typename Vector1, typename Vector2>
index_type
first(index_type begin, Predicate p, Vector1 v, Vector2 w);
```

Returns: The smallest index k >= j such that `f(v.get(k), w.get(k))`. A return value at least `v.size()` indicates `f(v.get(k), w.get(k))` is `false` for all k >= j. This value will equal `v.size()` if `j < v.size()`.

4.6.2.2. indexbool()

```
template <typename VectorT>
length_type
index_bool(VectorT source, Vector<Index<1> > indices);
template <typename MatrixT>
length_type
index_bool(MatrixT source, Vector<Index<2> > indices);
template <typename TensorT>
length_type
index_bool(TensorT source, Vector<Index<3> > indices);
```

Description: Obtain all indices for which `source` evaluates to `true`, in lexicographical order.

Returns: The number of elements returned.

4.6.2.3. gather()

```
template <typename T, typename B0, typename B1>
Vector<T, unspecified >
gather(const_Vector<T, B0> source, Vector<Index<1>, B1> indices);
template <typename T, typename B0, typename B1>
Vector<T, unspecified >
gather(const_Matrix<T, B0> source, Vector<Index<2>, B1> indices);
template <typename T, typename B0, typename B1>
Vector<T, unspecified >
gather(const_Tensor<T, B0> source, Vector<Index<3>, B1> indices);
```

Description: Returns value from `source`, at positions given by `indices`.

4.6.2.4. scatter()

```
template <typename T, typename B0, typename B1, typename B2>
void
scatter(const_Vector<T, B0> source,
        Vector<Index<1>, B1> indices,
        Vector<T, B1> destination);
template <typename T, typename B0, typename B1, typename B2>
void
scatter(const_Vector<T, B0> source,
        Vector<Index<2>, B1> indices,
        Matrix<T, B1> destination);
template <typename T, typename B0, typename B1, typename B2>
```

```
void
scatter(const_Vector<T, B0> source,
        Vector<Index<3>, B1> indices,
        Tensor<T, B1> destination);
```

Description: Copies all values from `source` into `destination`, at the index given by `indices`.

4.7. Signal Processing Functions

4.7.1. `blackman`

Description: Construct a vector containing Blackman window weights.

Syntax:

`Vector<float>` **`blackman`** `(` *`length_type LEN`* `);`

Requirements: The parameter *`LEN`* must be greater than one.

Result: Each element `n`, from 0 through `LEN-1`, of the result vector is

```
0.42 -
0.50 * cos(temp1 * n) +
0.08 * cos(temp2 * n)
```

where the `float` values `temp1` is `2 * VSIP_IMPL_PI / (len - 1)` and `temp2` is `2 * temp1`.

Example:

```
Vector<float> Z;
Z = blackman(200);
```

See Also:

`cheby` (section 4.7.2)

`hanning` (section 4.7.4)

`kaiser` (section 4.7.5)

4.7.2. `cheby`

Description: Construct a vector containing Chebyshev window weights.

Syntax:

`Vector<float>` **`cheby`** `(` *`length_type LEN , scalar_f ripple`* `);`

Requirements: The parameter *`LEN`* must be greater than one.

Result: The function returns a vector of length *`LEN`* initialized with Dolph-Chebyshev window weights using the specified *`ripple`*.

Example:

```
Vector<float> Z;
Z = cheby(200,0.5);
```

See Also:

`blackman` (section 4.7.1)

`hanning` (section 4.7.4)

`kaiser` (section 4.7.5)

4.7.3. `freqswap`

Syntax:

```
    Vector<T> freqswap( Vector<T> A );
    Matrix<T> freqswap( Matrix<T> A );
```

Description: Swaps halves of a vector, or quadrants of a matrix, to remap zero frequencies from the origin to the middle.

Result: `freqswap` returns a new vector (matrix) with *A*'s halves (quadrants) swapped.

Example:

```
Vector<int> vec(4);

vec(0) = 10;
vec(1) = 11;
vec(2) = 12;
vec(3) = 13;

std::cout << freqswap(vec) << std::endl; // prints: 12 13 10 11
```

4.7.4. `hanning`

Description: Construct a vector containing Hanning window weights.

Syntax:

`Vector<float>` **`hanning`** `(` *`length_type LEN`* `);`

Requirements: The parameter *`LEN`* must be greater than one.

Result: Each element `n`, from 0 through `LEN-1`, of the result vector is

```
0.5 * (1 - (cos(temp * (n + 1))))
```

where the `float` value `temp` is `2 * VSIP_IMPL_PI / (len + 1)`.

Example:

```
Vector<float> Z;
Z = hanning(200);
```

See Also:

`blackman` (section 4.7.1)

`cheby` (section 4.7.2)

`kaiser` (section 4.7.5)

4.7.5. `kaiser`

Description: Construct a vector containing Kaiser window weights.

Syntax:

```
Vector<float> kaiser (  length_type LEN ,  scalar_f beta  );
```

Requirements: The parameter *`LEN`* must be greater than one.

Result: The function returns a vector initialized with Kaiser window weights with transition width parameter *`beta`* and having length *`LEN`*.

Example:

```
Vector<float> Z;
Z = kaiser(200,0.5);
```

See Also:

`blackman` (section 4.7.1)

`cheby` (section 4.7.2)

`hanning` (section 4.7.4)

4.8. Signal Processing Objects

4.8.1. Convolution

This section describes the `Convolution` class provided by VSIPL++.

Applying a convolution object on a view performs a convolution on the view. `Convolution` supports different computations, depending on the element type and dimensionalities of the kernel, input and output views.

```
template <template <typename, typename> class ConstViewT,
          symmetry_type                       Symm,
          support_region_type                 Supp,
          typename                            T,
          unsigned                            N_times = 0,
          alg_hint_type                       A_hint = alg_time>
class Convolution;
```

Template parameters

`ConstViewT` The convolution dimensionality: `const_Vector` for 1D convolutions; `const_Matrix` for 2D convolutions.

`Symm` The symmetry of the kernel. See `symmetry_type` (section 2.7.10).

`Supp` The support region of the convolution algorithm. See `support_region_type` (section 2.7.9).

`T` The type of the elements in the views.

`N_times` The expected number of times this object will be used. This is a hint how much effort to spend on upfront optimization. A value of `0` stands for infinity, and thus results in the most effort.

`A_hint` One of `alg_time`, `alg_space`, or `alg_none`. This indicates how the implementation should optimize its computation or resource use. See alg_hint_type (section 2.7.1).

4.8.1.1. Constructor

Construct a 1D object

```
template <unspecified>
Convolution(const_Vector<T, unspecified> filter_coeffs,
            Domain<1> const&             input_size,
            length_type                  decimation = 1);
```

Construct a 2D object

```
template <unspecified>
Convolution(const_Matrix<T, unspecified> filter_coeffs,
            Domain<2> const&             input_size,
            length_type                  decimation = 1);
```

Description: Create a `Convolution` object of the given kernel view, input size, and decimation. The first version is available when the class's `ConstViewT` template parameter is `const_Vector<T, unspecified>`; the second when it is `const_Matrix<T,unspecified>`.

4.8.1.2. Call operators

```
template <unspecified>
Vector<T, unspecified>
operator()(
   const_Vector<T, unspecified> in,
   Vector<T, unspecified>       out);
```

Description: Calls the Convolution on the given input vector. This version is available only when the class's `ConstViewT` template parameter is `const_Vector<T, unspecified>.`

Result: The result view is stored into the `out` argument, and is returned as return-value for convenience, too.

```
template <unspecified>
Matrix<T, unspecified>
operator()(
   const_Matrix<T, unspecified> in,
   Matrix<T, unspecified>       out);
```

Description: Calls the Convolution on the given input matrix. This version is available only when the class's `ConstViewT` template parameter is `const_Matrix<T, unspecified>.`

Result: The result view is stored into the `out` argument, and is returned as return-value for convenience, too.

4.8.1.3. Example

```
// Declare a vector of kernel coefficients.
Vector<float> coeff(5);
...  // Initialize the coefficients.

// Declare a convolution object.
Convolution<const_Vector, nonsym, support_min, float>
   conv(coeff, Domain<1>(100), 1);

// Declare the input vector.
Vector<float> in(100);
...  // Initialize the input vector.

// Declare the output vector.
Vector<float> out(96);

// Perform the convolution of the input into the output.
conv(in, out);
```

See Also:

support_region_type	(section 2.7.9)
symmetry_type	(section 2.7.10)

4.8.2. Correlation

This section describes the `Correlation` class provided by VSIPL++.

Applying a correlation object on a view performs a correlation on the view. `Correlation` supports different computations, depending on the element type and dimensionalities of the kernel, input and output views.

```
template <template <typename, typename> class ConstViewT,
          support_region_type                 Supp,
          typename                            T,
          unsigned                            N_times = 0,
          alg_hint_type                       A_hint = alg_time>
class Correlation;
```

Template parameters

`ConstViewT`	The correlation dimensionality: `const_Vector` for 1D correlations; `const_Matrix` for 2D correlations.
`Supp`	The support region of the correlation algorithm. See `support_region_type` (section 2.7.9).
`T`	The type of the elements in the views.

`N_times` The expected number of times this object will be used. This is a hint how much effort to spend on upfront optimization. A value of `0` stands for infinity, and thus results in the most effort.

`A_hint` One of `alg_time`, `alg_space`, or `alg_none`. This indicates how the implementation should optimize its computation or resource use. See alg_hint_type xx (section 2.7.1).

4.8.2.1. Constructor

Construct a 1D object

```
Correlation(Domain<1> const&   ref_size,
            Domain<1> const&   input_size);
```

Construct a 2D object

```
Correlation(Domain<2> const&   ref_size,
            Domain<2> const&   input_size);
```

Description: Creates a `Correlation` object with the given kernel size and input size. The first version is available when the class's `ConstViewT` template parameter is `const_Vector<T, unspecified>`; the second when it is `const_Matrix<T,unspecified>`.

4.8.2.2. Call operators

```
template <unspecified>
Vector<T, unspecified>
operator()(
    bias_type                    bias,
    const_Vector<T, unspecified> ref,
    const_Vector<T, unspecified> in,
    Vector<T, unspecified>       out);
```

Description: Calls the Correlation on the given kernel, input and output vectors. This version is available only when the class's `ConstViewT` template parameter is `const_Vector<T, unspecified>`. The parameter *`bias`* must be a member of the enumeration `bias_type`. See (section 2.7.2).

Result: The result view is stored into the `out` argument, and is returned as return-value for convenience, too.

```
template <unspecified>
Matrix<T, unspecified>
operator()(
    bias_type                    bias,
    const_Matrix<T, unspecified> ref,
    const_Matrix<T, unspecified> in,
    Matrix<T, unspecified>       out);
```

Description: Calls the Correlation on the given kernel, input and output matrices. This version is available only when the class's `ConstViewT` template parameter is `const_Matrix<T, unspecified>`. The parameter *`bias`* must be a member of the enumeration `bias_type`. See (section 2.7.2).

Result: The result view is stored into the `out` argument, and is returned as return-value for convenience, too.

4.8.2.3. Example

```
// Declare a correlation object.
Correlation<const_Vector, support_min, float>
    corr(Domain<1>(5), Domain<1>(100));

// Declare a vector of kernel items.
Vector<float> kernel(5);
...  // Initialize the kernel.

// Declare the input vector.
Vector<float> in(100);
...  // Initialize the input vector.

// Declare the output vector.
Vector<float> out(96);

// Perform the correlation of the input into the output.
corr(biased, kernel, in, out);
```

See Also:

support_region_type	(section 2.7.9)
bias_type	(section 2.7.2)

4.8.3. `FIR Filter`

This section describes the `Fir` class provided by VSIPL++.

Applying a `Fir` object to a view performs a FIR filter on the view.

```
template <typename  T,
          symmetry_type S = nonsym,
          obj_state C = state_save,
          unsigned  N = 0,
          alg_hint_type H = alg_time>
class Fir;
```

Template parameters

T The type of the elements in the views.

S The symmetry of the kernel. See `symmetry_type` (section 2.7.10).

C `state_no_save` or `state_save`. An `Fir` object can maintain state between invocations of its call operator. See obj_state (section 2.7.4).

N The expected number of times this object will be used. This is a hint how much effort to spend on upfront optimization. A value of 0 stands for infinity, and thus results in the most effort.

H One of `alg_time`, `alg_space`, or `alg_none`. This indicates how the implementation should optimize its computation or resource use. See alg_hint_type (section 2.7.1).

4.8.3.1. Constructor

```
template <unspecified>
    Fir(
        const_Vector<T,unspecified> kernel,
        length_type input_size,
        length_type decimation = 1)
```

Description: Construct an `Fir` object with a vector of kernel coefficients and the indicated size of input vector and decimation.

```
Fir(Fir const &fir)
```

Description: Construct a new `Fir` object from an existing one..

4.8.3.2. Accessor Functions

```
length_type kernel_size()
length_type filter_order()
length_type input_size()
length_type output_size()
length_type decimation()
obj_state continuous_filtering()
```

Description: Report the various attributes of this `Fir` object.

4.8.3.3. Call operator

```
template <unspecified>
length_type operator()(
    const_Vector<T, unspecified> in,
    Vector<T, unspecified>       out
    )
```

Description: Apply the `Fir` object to the vector `in` and write results in the vector `out`. The length of the output vector is returned.

4.8.3.4. Assignment operator

```
Fir &operator= (Fir const &fir)
```

Description: Copy one `Fir` object into another.

4.8.3.5. Example

```
// Declare the kernel vector.
Vector<float> kernel(5);
...  // Initialize the kernel vector.

// Declare an FIR object.
Fir<float,nonsym,state_save> fir(kernel, 100, 1);

// Declare the input vector.
```

```
Vector<float> in(100);
...  // Initialize the input vector.

// Declare the output vector.
Vector<float> out(96);

// Perform the FIR filter on the input into the output.
fir(in, out);
```

See Also:

obj_state (section 2.7.4)

symmetry_type (section 2.7.10)

4.8.4. Fft - Fast Fourier Transform

Applying an FFT object on a view performs a single Fast Fourier Transform on the entire view. `Fft` supports different computations, dependent on the input element type, output element type, a specified direction or a special dimension, and the dimensionalities of the input and output views.

```
template <template <typename, typename> class ViewT,
          typename InputT,
          typename OutputT,
          int SD = 0,
          return_mechanism_type ReturnMechanism = by_value,
          unsigned Number = 0,
          alg_hint_type Hint = alg_time>
class Fft;
```

Template parameters

ViewT	Used to indicate the FFT dimensionality: `const_Vector` for 1D FFTs, `const_Matrix` for 2D FFTs, and `const_Tensor` for 3D FFTs.
InputT, OutputT	The input and output value-types of the Fourier transform, respectively. For a complex transform, both types need to be identical. For a real transform, one of them is complex, the other real.
SD	The special dimension / direction. In case of a real FFT, its value indicates which dimension has different input and output sizes. In case of a complex FFT, its value indicates whether to perform a forward or inverse transform.
ReturnMechanism	The return-mechanism-type indicates whether to return the output-view by-value or by-reference.
Number	The expected number of times this object will be used. This is a hint how much effort to spend on upfront optimization. A value of `0` stands for infinity, and thus results in the most effort.
Hint	One of `alg_time`, `alg_space`, or `alg_none`. This indicates how the implementation should optimize its computation or resource use.

FFT parameters should fulfill the following requirements. Input and output sizes are specified during construction, the other parameters are template arguments.

View	input-type / output-type	SD	input size	output size
Vector	complex<T> / complex<T>	fft_fwd	M	M
	complex<T> / complex<T>	fft_inv	M	M
	T / complex<T>	0	M	M/2 + 1
	complex<T> / T	0	M/2 + 1	M
Matrix	complex<T> / complex<T>	fft_fwd	MxN	MxN
	complex<T> / complex<T>	fft_inv	MxN	MxN
	T / complex<T>	0	MxN	Mx(N/2 + 1)
	T / complex<T>	1	MxN	(M/2 + 1)xN
	complex<T> / T	0	Mx(N/2 + 1)	MxN
	complex<T> / T	1	(M/2 + 1)xN	MxN
Tensor	complex<T> / complex<T>	fft_fwd	MxNxP	MxNxP
	complex<T> / complex<T>	fft_inv	MxNxP	MxNxP
	T / complex<T>	0	MxNxP	MxNx(P/2 + 1)
	T / complex<T>	1	MxNxP	Mx(N/2 + 1)xP
	T / complex<T>	2	MxNxP	MxNx(P/2 + 1)
	complex<T> / T	0	MxNx(P/2 + 1)	MxNxP
	complex<T> / T	1	Mx(N/2 + 1)xP	MxNxP
	complex<T> / T	2	MxNx(P/2 + 1)	MxNxP

4.8.4.1. Constructor

```
Fft(Domain<dim> const& dom, scalar_type scale);
```

Description: Creates an Fft object of the given size, with the given scaling factor.

Requirements: `dim` is the Fft dimensionality.

4.8.4.2. Accessor functions

```
Domain<dim> const& input_size() const;
Domain<dim> const& output_size() const;
scalar_type scale() const;
bool forward() const;
```

Description: Report the various attributes of this Fft object. `forward()` returns true for a forward transform, false otherwise.

4.8.4.3. Call operators

```
template <class ViewT>
<undefined> operator()(ViewT in);
```

Description: Calls the Fft on the given input, out-of-place.

Requirements: This operator is available for `ReturnMechanismType=by_value` only. `ViewT` has to obey they above requirements for type and dimensions.

Result: Returns a new view containing the output.

```
template <class InViewT, class OutViewT>
OutViewT operator()(InViewT in, OutViewT out);
```

Description: Calls the Fft on the given input, out-of-place

Requirements: This operator is available for `ReturnMechanismType=by_reference` only. `InViewT` and `OutViewT` have to obey the above requirements for type and dimensions.

Result: The result view is stored into the `out` argument, and is returned as return-value for convenience, too.

```
template <class ViewT>
ViewT operator()(ViewT inout);
```

Description: Calls the Fft on the given input, in-place.

Requirements: This operator is available only for complex FFT objects, with `ReturnMechanismType=by_reference`. `ViewT` has to obey the above requirements for type and dimensions.

Result: The result is stored in-place into the `inout` view, and is also returned as return-value for convenience

4.8.5. Class template Fftm<>

Applying an `Fftm` object on a Matrix performs multiple fast Fourier transforms on the rows or columns of a Matrix. A Multiple FFT treats a matrix as a collection of either rows or columns and applies an FFT to each row or column.

```
template <typename InputT,
          typename OutputT,
          int A = row,
          int D = fft_fwd,
          return_mechanism_type ReturnMechanism = by_value,
          unsigned Number = 0,
          alg_hint_type Hint = alg_time>
class Fftm;
```

Template parameters

InputT, OutputT	The input and output value-types of the Fourier transform, respectively. For a complex transform, both types need to be identical. For a real transform, one of them is complex, the other real.
Axis	The dimension along which to apply the Ffts.
Direction	The direction of the Ffts, either `fft_fwd` or `fft_inv`.
ReturnMechanism	The return-mechanism-type indicates whether to return the output-view by-value or by-reference.

Number — The expected number of times this object will be used. This is a hint how much effort to spend on upfront optimization. A value of `0` stands for infinity, and thus results in the most effort.

Hint — A `alg_hint_type` (section 2.7.1), indicating how the implementation should optimize its computation or resource use.

FFTM parameters should fulfill the following requirements. Input and output sizes are specified during construction, the other parameters are template arguments.

input-type / output-type	axis	direction	input size	output size
`complex<T>` / `complex<T>`	`0` or `1`	`fft_fwd`	MxN	MxN
`complex<T>` / `complex<T>`	`0` or `1`	`fft_inv`	MxN	MxN
`T` / `complex<T>`	`0`	`fft_fwd`	MxN	`Mx(N/2 + 1)`
`T` / `complex<T>`	`1`	`fft_fwd`	MxN	`(M/2 + 1)xN`
`complex<T>` / `T`	`0`	`fft_inv`	`Mx(N/2 + 1)`	MxN
`complex<T>` / `T`	`1`	`fft_inv`	`(M/2 + 1)xN`	MxN

4.8.5.1. Constructor

```
Fftm(Domain<2> const& dom, scalar_type scale);
```

Description: Creates an `Fftm` object of the given size, with the given scaling factor.

4.8.5.2. Accessor functions

```
Domain<2> const& input_size() const;
Domain<2> const& output_size() const;
scalar_type scale() const;
bool forward() const;
```

Description: Report the various attributes of this Fftm object. `forward()` returns true for a forward transform, false otherwise.

4.8.5.3. Call operators

```
template <class MatrixT>
<undefined> operator()(MatrixT in);
```

Description: Calls the Fftm on the given input, out-of-place.

Requirements: This operator is available for `ReturnMechanismType=by_value` only. `MatrixT` has to obey the above requirements for type and dimensions.

Result: Returns a new matrix containing the output.

```
template <class InMatrixT, class OutMatrixT>
OutMatrixT operator()(InMatrixT in, OutMatrixT out);
```

Description: Calls the Fftm on the given input, out-of-place.

Requirements: This operator is available for `ReturnMechanismType=by_reference` only. `InMatrixT` and `OutMatrixT` have to obey the above requirements for type and dimensions.

Result: The result matrix is stored into the `out` argument, and is returned as return-value for convenience, too.

```
template <class MatrixT>
MatrixT operator()(MatrixT inout);
```

Description: Calls the Fftm on the given input, in-place.

Requirements: This operator is available only for complex FFTM objects, with `ReturnMechanismType=by_reference`. `MatrixT` has to obey the above requirements for type and dimensions.

Result: The result is stored in-place into the `inout` matrix, and is also returned as return-value for convenience.

4.8.6. Histogram

This section describes the `Histogram` class provided by VSIPL++.

Applying a histogram object to a view computes a histogram of the view. `Histogram` supports different computations, depending on the indicated element type and dimensionality.

```
template <template <unspecified>
          class const_View = const_Vector,
          typename T>
class Histogram;
```

Template parameters

`const_View` The histogram dimensionality: `const_Vector` for 1D histograms; `const_Matrix` for 2D histograms.

`T` The type of the elements in the views.

4.8.6.1. Constructor

```
Histogram(T min_value, T max_value, length_type num_bin);
```

Description: Create a `Histogram` object to collect results constrained by these values:

`min_value` The first element of the result vector accumulates results for all input items that are less than this value.

`max_value` The last element of the result vector accumulates results for all input items that are greater than or equal to this value.

`num_bin` The number of elements in the result vector.

For each input element `V` such that *`min_value`* <= `V` < *`max_value`*, the result element `I` is incremented. `I` is computed as `((V-min_value)/delta)+1`, where `delta` is `(max_value-min_value)/(num_bin-2)`.

4.8.6.2. Call operators

```
template <unspecified>
const_Vector<scalar_i>
```

```
        operator()(const_Vector<T, Block> data,
                   bool accumulate = false)
```

Description: Calls the Histogram on the given input vector. When the parameter *`accumulate`* is `true`, the results for this call are added to previous results.

Requirements: This call operator is available only when the template parameter *`const_View`* is `const_Vector`.

Result: The result vector view is returned as return-value.

```
        template <unspecified>
        const_Vector<scalar_i>
        operator()(const_Matrix<T, Block> data,
                   bool accumulate = false)
```

Description: Calls the Histogram on the given input matrix. When the parameter *`accumulate`* is `true`, the results for this call are added to previous results.

Requirements: This call operator is available only when the template parameter *`const_View`* is `const_Matrix`.

Result: The result vector view is returned as return-value.

4.8.6.3. Example

```
// Declare a histogram object.
Histogram<const_Vector, float> h(0, 8, 10);

// Declare the input vector.
Vector<float> in(100);
...  // Initialize the input vector.

// Declare the output vector.
Vector<scalar_i> out(10);

// Compute the histogram of the input into the output.
out = h(in);
```

4.8.7. IIR Filter

This section describes the Infinite Impulse Response filter provided by VSIPL++.

4.8.7.1. Class template Iir<>

```
template <typename T = float,
   object_state Save = state_save,
   unsigned Number = 0,
   alg_hint_type Hint = 0>
class Iir;
```

Template parameters

T The value-type of the `Iir` filter

obj_state — If this object is to be called repeatedly on subsequent chunks of a long vector, use `state_save`, else `state_nosave`

Number — The expected number of times this object will be used. This is a hint to the library how much effort to spend on upfront optimization. A value of `0` stands for infinity, and thus results in the most effort.

Hint — One of `alg_time`, `alg_space`, or `alg_none`. This indicates how the library should optimize its computation or resource use.

4.8.7.1.1. Constructor

```
Iir(const_Matrix<T, unspecified> b,
    const_Matrix<T, unspecified> a,
    length_type i);
```

Description: Creates an Iir object...

4.8.7.1.2. Accessor functions

```
length_type kernel_size() const;
length_type filter_order() const;
length_type input_size() const;
length_type output_size() const;
```

Description: Report the various attributes of this Iir filter.

4.8.7.1.3. Call operators

```
template <class InViewT, class OutViewT>
OutViewT operator()(InViewT in, OutViewT out);
```

Description: Calls the Iir filter on `in`, storing the result in `out`.

Result: Returns a new view containing the output.

Chapter 5
Advanced VSIPL++ Data Types

5.1. Blocks

Description. Every block is a logically contiguous array of data. Blocks provide element-wise operations to access the data. Blocks do not, in general, provide data-parallel access to the data.

> **Note**
> There is no requirement that a block store data by allocating memory to hold the data. For example, a block may compute the data dynamically.

Valid expressions.

Expression	Requirements	Semantics
Block::value_type		The value type of this block.
Block::reference_type		The reference type of this block.
Block::const_reference_type		The const reference type of this block.
Block::map_type		The map type of this block.
b.size()		Return the number of elements in this block.
b.size(dimension_type X, dimension_type d)		Return the extent of this block in the *d* dimension.
b.map()		
b.increment()		
b.decrement()		
b.get(index_type i_1,..., index_type i_x)	The arity of this method corresponds to the dimensionality of this block.	Returns the value of the given element.

5.1.2. Writable Block concept

Description. A WritableBlock is a Block that can be written to.

Valid expressions.

Expression	Requirements	Semantics
b.put(i_1,i_2,...,i_x,t)		Sets the value of element (i_1,...i_x) to t.

5.1.3. Allocatable Block concept

Description. An AllocatableBlock is a Block that can be allocated. It may be writable.

Valid expressions.

Expression	Requirements	Semantics
Block(dom, map)		construct a block of type Block with the given domain and the given map.
Block(dom, value, map)		construct a block of type Block with the given domain, the given value for all its elements, and the given map.

5.2. The Layout template

Description. The Layout template is simple tuple representing compile-time information about a block's data (storage) layout.

```
template <dimension_type D,
          typename Order,
```

```
          typename PackType,
          typename ComplexType>
{
  static dimension_type const dim = D;
  typedef PackType pack_type;
  typedef Order order_type;
  typedef ComplexType complex_type;
};
```

Template parameters.

D	The block's dimension.
Order	The dimension ordering. This is expressed using tuples, i.e. tuple<0,1,2>, tuple<1,0,2>, etc., or aliases such as row2_type or col2_type.
PackType	One of `Stride_unit`, `Stride_unit_dense`, `Stride_unit_align<...>`, or `Stride_unknown`.
ComplexType	One of `Any_type`, `Cmplx_inter_fmt`, or `Cmplx_split_fmt`.

5.3. The Dense class template

Description. Dense models the Section 5.1.3, “Allocatable Block concept” concept. It explicitly stores one value for each index in its domain, in a dense memory block.

```
template <dimension_type D = 1,
          typename T = VSIP_DEFAULT_VALUE_TYPE,
   typename Order = tuple<0,1,2>,
   typename Map = Local_map>
class Dense
{
public:
  Dense(Domain<D> const &dom, T value, Map const &map);
  Dense(Domain<D> const &dom, Map const &map);
  Dense(Domain<D> const &dom, T *data, Map const &map);

  user_storage_type user_storage() const;
  bool admitted() const;
  void admit(bool update);
  void release(bool update);
  void release(bool update, T *&data);
  void find(T *&data);
  void rebind(T *data);
  void rebind(T *data, Domain<D> const &dom);

};
```

The interface for complex Dense blocks has some additional member functions to handle user storage.

```
template <dimension_type D,
          typename T,
     typename Order,
   typename Map>
```

```
class Dense<D, complex<T>, Order, Map>
{
public:
  Dense(Domain<D> const &dom, complex<T> value, Map const &map);
  Dense(Domain<D> const &dom, Map const &map);
  Dense(Domain<D> const &dom, complex<T> *data, Map const &map);
  Dense(Domain<D> const &dom, T *data, Map const &map);
  Dense(Domain<D> const &dom, T *real, T *imag, Map const &map);

  user_storage_type user_storage() const;
  bool admitted() const;
  void admit(bool update);
  void release(bool update);
  void release(bool update, complex<T> *&data);
  void release(bool update, T *&data);
  void release(bool update, T *&real, T *&imag);
  void find(complex<T> *&data);
  void find(T *&data);
  void find(T *&real, T *&imag);
  void rebind(complex<T> *data);
  void rebind(T *data);
  void rebind(T *real, T *imag);
  void rebind(complex<T> *data, Domain<D> const &dom);
  void rebind(T *data, Domain<D> const &dom);
  void rebind(T *real, T *imag, Domain<D> const &dom);

};
```

Template parameters.

D The block's dimension.

T The block's value-type.

Order The dimension ordering. This is expressed using tuples, i.e. tuple<0,1,2>, tuple<1,0,2>, etc., or aliases such as row2_type or col2_type.

Map The block's map-type.

5.3.1. Constructors

```
Dense(Domain<D> const &dom, Map const &map);
```

Description. Construct a Dense block.

```
Dense(Domain<D> const &dom, value_type value, Map const &map);
```

Description. Construct a Dense block, with all values initialized to *value*

```
Dense(Domain<D> const &dom, value_type *data, Map const &map);
```

Description. Construct a Dense block using user-storage. The block's data may only be accessed after a call to `admit()`.

```
Dense(Domain<D> const &dom, scalar_type *data, Map const &map);
Dense(Domain<D> const &dom, scalar_type *real, scalar_type *imag, \
Map const &map);
```

Description. Construct a complex Dense block using user-storage. In the first case the data is passed in as an interleaved array. In the second case the data is passed in as a split pair of real arrays, holding the real and imaginary parts of the data. The block's data may only be accessed after a call to `admit()`.

5.3.2. User-storage functions

```
user_storage_type user_storage() const;
```

Description. Return the type of user-storage of this block.

```
void admit(bool update);
```

Description. Admit the user-storage, allowing the block to access the data. If *`update`* is `true`, this operation may perform a copy into the block's own storage.

```
bool admitted() const;
```

Description. Return `true` if the block is being admitted.

```
void release(bool update);
void release(bool update, value_type *&data);
```

Description. Release the user storage. If *`update`* is `true`, this operation may perform a copy into the user storage. If a value-type pointer is provided, it is set to the start of the user storage block. If the block doesn't use user storage, set *`data`* to `0`.

```
void release(bool update, scalar_type *&data);
void release(bool update, scalar_type *&real, scalar_type *&imag);
```

Description. If this is a complex block, two additional `release()` functions are provided. The first returns a pointer to interleaved-complex array, the second the two pointers to the split complex pair of arrays.

```
void find(value_type *&data);
```

Description. Return the start of the user-storage of this block. If the block does not use user-storage, return `0`.

```
void find(scalar_type *&data);
void find(scalar_type *&real, scalar_type *&imag);
```

Description. These two variants are only available for complex blocks. The first returns the pointer to the interleaved complex array, the second the two pointers to the split complex pair of arrays.

```
void rebind(value_type *data);
void rebind(value_type *data, Domain<D> const &dom);
```

Description. Rebind the block to a new user-storage array. If a `Domain` is provided, reset the block's size accordingly.

```
void rebind(scalar_type *data);
void rebind(scalar_type *real, scalar_type *imag);
void rebind(scalar_type *data, Domain<D> const &dom);
void rebind(scalar_type *real, scalar_type *imag, Domain<D> const \
&dom);
```

Description. These variants are only available for complex blocks. The first rebinds the block to the interleaved complex array, the second to two split-complex arrays. If a `Domain` is provided, reset the block's size accordingly.

Chapter 6
Extension Reference

Abstract

This chapter contains detailed references for extensions that Codesourcery has added to VSIPL++.

6.1. Introduction

The following man pages describe the operation of each VSIPL++ extension, function and class.

Unless otherwise noted, the following names are in the *vsip_csl* namespace.

6.2. Sort Functions

6.2.1. sort_indices

Syntax:

```
template <typename T,
          typename Block1,
          typename Block2,
          typename FunctorT>
void
sort_indices(
        Vector<index_type, Block1> indices,
  const_Vector<T,          Block2> data,
      FunctorT                    sort_functor
  );
```

Template parameters

T the type of the elements in vector *data*.

Block1 the block type of vector *indices*.

Block2 the block type of vector *data*.

FunctorT a class type with a member `bool operator()(T, T)`. The default value is `std::less<T>`.

Description: `sort_indices` overwrites *indices* with index values such that `sort_functor(data(indices(i)), data(indices(j)))` is true iff `i <= j`. The vector *data* is not modifed in any way.

Requirements: Vectors *indices* and *data* must have the same size.

Example:

```
Vector<float> vec(4);
Vector<index_type> inx(4);

vec(0) = 11.;
vec(1) = 14.;
vec(2) = 13.;
vec(3) = 12.;

sort_indices(inx, vec);

std::cout << inx << std::endl;  // prints 0 3 2 1
```

```
sort_indices(inx, vec, greater<float>());

std::cout << inx << std::endl;  // prints 1 2 3 0

std::cout << vec << std::endl;  // prints 11 14 13 12
```

See Also:

`sort_data` (section 6.2.2)

`sort_data (in place)` (section 6.2.3)

6.2.2. `sort_data (out of place)`

Syntax:

```
    template <typename T,
              typename Block1,
              typename Block2,
              typename FunctorT>
    void
    sort_data(
      const_Vector<T, Block1> data_in,
            Vector<T, Block2> data_out,
          FunctorT            sort_functor
      );
```

Template parameters

`T` the type of the elements in vectors *`data_in`* and *`data_out`*.

`Block1` the block type of vector *`data_in`*.

`Block2` the block type of vector *`data_out`*.

`FunctorT` a class type with a member `bool operator()(T, T)`. The default value is `std::less<T>`.

Description: After copying *`data_in`* to *`data_out`*, `sort_data` rearranges the values such that `sort_functor(data_out(i), data_out(j))` is true iff `i <= j`. The vector *`data_in`* is not modifed in any way.

Requirements: Vectors *`data_in`* and *`data_out`* must have the same size.

Example:

```
Vector<float> vec(4);
Vector<float> out(4);

vec(0) = 11.;
vec(1) = 14.;
vec(2) = 13.;
vec(3) = 12.;

sort_data(vec, out);
```

```
std::cout << out << std::endl;  // prints 11 12 13 14

sort_data(vec, out, greater<float>());

std::cout << out << std::endl;  // prints 11 12 13 14

std::cout << vec << std::endl;  // prints 11 14 13 12
```

See Also:

`sort_data (in place)` (section 6.2.3)

`sort_indices` (section 6.2.1)

6.2.3. `sort_data (in place)`

Syntax:

```
    template <typename T,
              typename BlockT,
              typename FunctorT>
    void
    sort_data(
            Vector<T, BlockT> data,
          FunctorT            sort_functor
      );
```

Template parameters

`T` the type of the elements in vector *`data`*.

`BlockT` the block type of vector *`data`*.

`FunctorT` a class type with a member `bool operator()(T, T)`. The default value is `std::less<T>`.

Description: `sort_data` rearranges the values such that `sort_functor(data(i), data(j))` is true iff `i <= j`.

Requirements: N/A

Example:

```
Vector<float> vec(4);
Vector<float> out(4);

vec(0) = 11.; vec(1) = 14.; vec(2) = 13.; vec(3) = 12.;

sort_data(vec);

std::cout << vec << std::endl;  // prints 11 12 13 14

vec(0) = 11.; vec(1) = 14.; vec(2) = 13.; vec(3) = 12.;
```

```
sort_data(vec, greater<float>());

std::cout << vec << std::endl;  // prints 14 13 12 11
```

See Also:

`sort_data` (section 6.2.2)

`sort_indices` (section 6.2.1)

6.3. Reduction Functions

6.3.1. `maxmgsqval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                       T,
          template <typename, typename> class ViewT,
          typename                       BlockT>
T
maxmgsqval(ViewT<complex<T>, BlockT> v);
```

Template parameters

`T` is the type of the elements in the view and of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxmgsqval` computes the maximum squared magnitude of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`. The function parameters *`v`* and *`idx`* are defined in terms of `complex<T>`.

Result: `maxmgsqval` returns a scalar value of type *`T`*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = maxmgsqval(vec);
cout << val << endl;  // prints 100
```

See Also: `maxmgsqval` (section 4.3.3)

6.3.2. `maxmgval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                      T,
          template <typename, typename> class ViewT,
          typename                      BlockT>
T
maxmgval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxmgval` computes the maximum magnitude of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: When *T* is `complex<W>`, `maxmgval` returns a scalar value of type *W*; otherwise it returns a value of type *T*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = maxmgval(vec);
cout << val << endl;  // prints 10
```

See Also: `maxmgval` (section 4.3.4)

6.3.3. `maxval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                      T,
          template <typename, typename> class ViewT,
          typename                      BlockT>
T
maxval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `maxval` computes the maximum of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `maxval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 0.;
vec(1) = 1.;
vec(2) = 3.;
vec(3) = 2.;

float val = maxval(vec);
std::cout << val << std::endl;  // prints 3.0
```

See Also: `maxval` (section 4.3.5)

6.3.4. `meansqval`

Syntax:

```
    template <typename                        T,
              template <typename, typename> class ViewT,
              typename                        BlockT,
              typename                        ResultT>
    ResultT
    meansqval(ViewT<T, BlockT> v, ResultT);
```

Template parameters

T is the type of the elements in the view.

ResultT is the type of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `meansqval` sums the squares of all the elements of the view and returns that value divided by the number of elements. The return type may be different from the element type in cases where summing may overflow an accumulator of the element type. To specify the return type, use an *exemplar* as the second parameter to the function. See the example.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `meansqval` returns a scalar value of type *`ResultT`*.

Example:

```
Vector<unsigned short> vec(4);

vec(0) = 256;
vec(1) = 1;
vec(2) = 2;
vec(3) = 3;

typedef unsigned long W;

std::cout << meansqval(vec, W()) << std::endl;  // prints 16387

// This version displays a wrong answer.

std::cout << meansqval(vec) << std::endl;  // prints 3
```

See Also: `meansqval` (section 4.3.6)

6.3.5. `meanval`

Syntax:

```
    template <typename                        T,
              template <typename, typename> class ViewT,
              typename                        BlockT,
              typename                        ResultT>
    ResultT
    meanval(ViewT<T, BlockT> v, ResultT);
```

Template parameters

`T` is the type of the elements in the view.

`ResultT` is the type of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `meanval` sums all the elements of the view and returns that value divided by the number of elements. The return type may be different from the element type in cases where summing may overflow an accumulator of the element type. To specify the return type, use an *exemplar* as the second parameter to the function. See the example.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `meanval` returns a scalar value of type *`ResultT`*.

Example:

```
Vector<unsigned short> vec(4);
```

```
vec(0) = 65535;
vec(1) = 1;
vec(2) = 2;
vec(3) = 3;

typedef unsigned long W;

std::cout << meanval(vec, W()) << std::endl;  // prints 16385

// This version displays a wrong answer.

std::cout << meanval(vec) << std::endl;  // prints 1
```

See Also: `meanval` (section 4.3.7)

6.3.6. `minmgsqval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                      T,
          template <typename, typename> class ViewT,
          typename                      BlockT>
T
minmgsqval(ViewT<complex<T>, BlockT> v);
```

Template parameters

`T` is the type of the elements in the view and of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minmgsqval` computes the minimum squared magnitude of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`. The function parameters *`v`* and *`idx`* are defined in terms of `complex<T>`.

Result: `minmgsqval` returns a scalar value of type *`T`*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = minmgsqval(vec);
cout << val << endl;  // prints 0.25
```

See Also: `minmgsqval` (section 4.3.8)

6.3.7. `minmgval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                       T,
          template <typename, typename> class ViewT,
          typename                       BlockT>
T
minmgval(ViewT<T, BlockT> v;
```

Template parameters

`T` is the type of the elements in the view and of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minmgval` computes the minimum magnitude of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: When *`T`* is `complex<W>`, `minmgval` returns a scalar value of type *`W`*; otherwise it returns a value of type *`T`*.

Example:

```
length_type         size = 13;
typedef float T;
Vector<complex<T> > vec(size, complex<T>(3, 4));
T                   val;

vec(1) = complex<T>(6, 8);
vec(2) = complex<T>(0.3, 0.4);

val = minmgval(vec);
cout << val << endl;  // prints 0.5
```

See Also: `minmgval` (section 4.3.9)

6.3.8. `minval`

Syntax:

```
#include <vsip_vsl/math.hpp>

template <typename                       T,
          template <typename, typename> class ViewT,
          typename                       BlockT>
```

```
    T
    minval(ViewT<T, BlockT> v);
```

Template parameters

T is the type of the elements in the view and of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `minval` computes the minimum of all the elements of the view and returns that value.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `minval` returns a scalar value of type *T*.

Example:

```
Vector<float> vec(4);

vec(0) = 3.;
vec(1) = 1.;
vec(2) = 0.;
vec(3) = 2.;

float val = minval(vec);
std::cout << val << std::endl;  // prints 0.0
```

See Also: `minval` (section 4.3.10)

6.3.9. `sumsqval`

Syntax:

```
    template <typename                       T,
              template <typename, typename> class ViewT,
              typename                       BlockT,
              typename                       ResultT>
    ResultT
    sumsqval(ViewT<T, BlockT> v, ResultT);
```

Template parameters

T is the type of the elements in the view.

ResultT is the type of the return value.

ViewT is the type of the parameter *v*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `sumsqval` sums the squares of all the elements of the view and returns that value. The return type may be different from the element type in cases where summing may overflow an

accumulator of the element type. To specify the return type, use an *exemplar* as the second parameter to the function. See the example.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `sumsqval` returns a scalar value of type *`ResultT`*.

Example:

```
Vector<unsigned short> vec(4);

vec(0) = 256;
vec(1) = 1;
vec(2) = 2;
vec(3) = 3;

typedef unsigned long W;

std::cout << sumsqval(vec, W()) << std::endl;  // prints 65550

// This version displays a wrong answer.

std::cout << sumsqval(vec)      << std::endl;  // prints    14
```

See Also: `sumsqval` (section 4.3.11)

6.3.10. `sumval`

Syntax:

```
    template <typename                        T,
              template <typename, typename> class ViewT,
              typename                        BlockT,
              typename                        ResultT>
    ResultT
    sumval(ViewT<T, BlockT> v, ResultT);
```

Template parameters

`T` is the type of the elements in the view.

`ResultT` is the type of the return value.

`ViewT` is the type of the parameter *`v`*. It must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Description: `sumval` sums all the elements of the view and returns that value. The return type may be different from the element type in cases where summing may overflow an accumulator of the element type. To specify the return type, use an *exemplar* as the second parameter to the function. See the example.

Requirements: The template parameter `ViewT` must be `const_Vector`, `const_Matrix` or `const_Tensor`.

Result: `sumval` returns a scalar value of type *`ResultT`*.

Example:

```
Vector<unsigned short> vec(4);

vec(0) = 65535;
vec(1) = 1;
vec(2) = 2;
vec(3) = 3;

typedef unsigned long W;

std::cout << sumval(vec, W()) << std::endl;  // prints 65541

// This version displays a wrong answer.

std::cout << sumval(vec)      << std::endl;  // prints     5
```

See Also: `sumval` (section 4.3.12)

6.4. View Cast

6.4.1. `view_cast`

Syntax:

```
    template <typename T,
              typename ViewT>
    typename View_of_dim<ViewT::dim, T, ...>::type
    view_cast(ViewT view);
```

Template parameters

`T` value type to cast elements of view *`view`* to. This must be explicit.

`ViewT` view type. Implied by *`view`*.

Description: `view_cast` casts the value type of *`view`* to new value type *`T`*. Data is not copied and original view is not modified in any way.

Header File / Namespace: `view_cast` is defined in the header `vsip_csl/view_cast.hpp` and is in the *`vsip_csl`* namespace.

Example: The following example casts the values in a view of 16-bit ints to avoid overflow for summation.

```
Vector<uint16_t> vec(4);
vec(0) = 1 << 15;
vec(1) = 1 << 15;
vec(2) = 1 << 15;
vec(3) = 1 << 15;
uint32_t sum = sumval(view_cast<uint32_t>(vec));
```

6.5. Dispatcher - related types

Unless otherwise noted, all types described in this section live in the `vsip_csl::dispatcher` namespace.

6.5.1. Evaluator class template

Description. Evaluators are used by the Dispatcher harness to express whether a given *backend* may process a particular argument. (For a description of the Dispatcher harness, see Chapter 3, "Using the Dispatch Framework".

```
template <typename Operation, typename Backend, typename Signature>
struct Evaluator;
```

Template parameters.

Operation	A tag used to identify the operation this Evaluator performs.
Backend	A tag used to identify a particular backend. The tag needs to appear in the List<Operation> specialization to participate in the dispatch process.
Signature	The signature of the operation to be performed. For example, to dispatch an operation `float compute(Block const &)` one would use the signature `float(Block const &)`.

Evaluators are specialized for particular operations and backends. The following requirements need to be fulfilled:

Valid expressions.

Expression	Requirements	Semantics
Evaluator::backend_type	This is only required when the Signature template argument is void(void).	The backend type this evaluator provides.
Evaluator::ct_valid		A boolean value expressing the result of the compile-time evaluation of this evaluator.
Evaluator::name()		Return the name (C string) of this evaluator, suitable for diagnostics and profiling purposes.
Evaluator::rt_valid(...)	This is only required in case the above ct_valid is `true`. The sig-ments allow this Evaluator to be nature of this function corres-used. ponds to the signature provided as third template argument to this Evaluator template specialization.	Return true if the particular argu-
Evaluator::exec(...)	This is only required in case the above ct_valid is `true`. The signature of this function corresponds to the signature provided as third template argument to this Evaluator template specialization.	Execute this Evaluator.

Example.

```
template <typename LHS, typename RHS>
struct Evaluator<op::assign<1>, be::user, void(LHS &, RHS const &)>
{
  char const *name() { return "my custom evaluator";}
  static bool const ct_valid = is_valid_expression<RHS>::value;
  static bool rt_valid(LHS &lhs, RHS const &rhs);
  static void expr(LHS &, RHS const &);
};
```

6.5.2. `dispatch`

Description. Dispatch an operation to an appropriate backend.

```
template <typename O, typename R, typename... Args>
R dispatch(Args... a);
```

Template arguments.

O The operation tag

R The return type

Args The argument types

Examples.

Example 6.1. Simple case

```
A a = ...;
B b = ...;
C c = ...;
result_type r = dispatch<operation_tag, result_type>(a, b, c);
```

Sometimes it is necessary to specify the argument types explicitly, for example to avoid qualifiers from being stripped off.

Example 6.2. Dispatch with block arguments

```
Vector<> argument = ...;
typedef Vector<>::block_type block_type;
result_type r = dispatch<operation_tag, result_type, block_type \
const &>(argument.block());
```

Description. Print out information about available backends for the given operation, without actually performing it.

```
template <typename O, typename... Args>
void dispatch_diagnostics(Args... a);
```

Template arguments.

O The operation tag

Args The argument types

Examples.

Example 6.3. Simple case

```
A a = ...;
B b = ...;
C c = ...;
dispatch_diagnostics<operation_tag>(a, b, c);
```

6.6. Expression block types

Unless otherwise noted, all types described in this section live in the `vsip_csl::expr` namespace.

6.6.1. The UnaryFunctor concept

Description. A Unary functor computes a result block from a single argument block. It is most frequently used in conjunction with the `Unary` template to represent non-elementwise unary block expressions.

Valid expressions.

Expression	Requirements	Semantics
F<B>	B is a valid Block model.	F needs to be instantiable with arbitrary block types.
F<B>::dim		The dimensionality of the result block
F<B>::result_type		The value-type of the result block
F<B>::map_type		The map-type of the result block
f.arg()		Return a const reference to the argument block.
f.size()		Return the total size of the result block.
f.size(X, d)		Return the size of the result block.
f.map()		Return the map of the result block.
f.apply(result)		Apply the functor, and store the result in *`result`*.

6.6.2. The Unary class template

Description. A Unary block is an expression block with a single argument block. It models the `Block` concept.

```
template <template <typename> class Operation,
          typename ArgumentBlock,
          bool Elementwise = false>
class Unary;
```

Template parameters.

Operation	If `Elementwise == true`, a model of the ElementwiseUnaryFunctor concept, otherwise a model of the `UnaryFunctor` concept
ArgumentBlock	

The interface (and implementation) of the `Unary` template depends slightly on whether the expression is element-wise. Thus two specializations are provided.

```
template <template <typename> class Operation, typename Block>
class Unary<Operation, Block, true>
```

```
{
public:
  Unary(Block const &);
  Unary(Operation<Block> const &, Block const &);

  operation_type const &operation() const;
  Block const &arg() const;
};
```

```
template <template <typename> class Operation, typename Block>
class Unary<Operation, Block, false>
{
public:
  Unary(Operation<Block> const &);

  operation_type const &operation() const;
  Block const &arg() const;
  void evaluate() const;
  template <typename ResultBlock>
  void apply(ResultBlock &result) const;
};
```

Member functions.

operation()	Return the operation associated with this expression.
arg()	Return the argument block associated with this expression.
evaluate()	Evaluate the operation, storing the result in the state of this expression block.
apply()	Evaluate the operation, storing the result in the provided block argument.

6.6.3. The Unary_functor class template

Description. Unary_functor models the UnaryFunctor concept. To use it, derive from it and provide an `apply()` that performs the desired operation.

```
template <typename ArgumentBlockType>
    class Unary_functor
{
public:
  Unary_functor(ArgumentBlockType const &);
  template <typename ResultBlockType>
  void apply(ResultBlockType &r) const {} // implement in derived \
class
};
```

Example 6.4. Use of Unary_functor

```
template <typename ArgumentBlockType>
struct Operation : Unary_functor<ArgumentBlockType>
{
  Operation(ArgumentBlockType const &arg)
    : Unary_functor<ArgumentBlockType>(arg) {}
  template <typename ResultBlockType>
  void apply(ResultBlockType &result) const
  { compute(result, this->arg());}
};

// Implement 'operate' as a lazy function
template <typename T, typename BlockType>
lazy_Vector<T, Unary<Operation, BlockType> const>
operate(const_Vector<T, BlockType> input)
{
  Operation<BlockType> operation(input.block());
  Unary<Operation, BlockType> block(operation);
  return lazy_Vector<T, Unary<Operation, BlockType> const>(block);
};
```

6.6.4. The BinaryFunctor concept

Description. A Binary functor computes a result block from two argument blocks. It is most frequently used in conjunction with the `Binary` template to represent non-elementwise binary block expressions.

Valid expressions.

Expression	Requirements	Semantics
F<B1, B2>	B1 and B2 are valid Block models.	F needs to be instantiable with arbitrary block types.
F<B1, B2>::dim		The dimensionality of the result block
F<B1, B2>::result_type		The value-type of the result block
F<B1, B2>::map_type		The map-type of the result block
f.arg1(), f.arg2()		Return const references to the argument blocks.
f.size()		Return the total size of the result block.
f.size(X, d)		Return the size of the result block.
f.map()		Return the map of the result block.
f.apply(result)		Apply the functor, and store the result in *`result`*.

6.6.5. The Binary class template

Description. A Binary block is an expression block with two argument blocks. It models the `Block` concept.

```
template <template <typename, typename> class Operation,
          typename Argument1Block, typename Argument2Block,
          bool Elementwise = false>
class Binary;
```

Template parameters.

Operation	If `Elementwise == true`, a model of the Elementwise-BinaryFunctor concept, otherwise a model of the `BinaryFunctor` concept

Argument1Block, Argument2Block

The interface (and implementation) of the `Binary` template depends slightly on whether the expression is element-wise. Thus two specializations are provided.

```
template <template <typename, typename> class Operation,
          typename Arg1, typename Arg2>
class Binary<Operation, Arg1, Arg2, true>
{
public:
  Unary(Arg1 const &, Arg2 const &);
  Unary(Operation<Arg1, Arg2> const &, Arg1 const &, Arg2 const &);

  operation_type const &operation() const;
  Arg1 const &arg1() const;
  Arg2 const &arg2() const;
};
```

```
template <template <typename, typename> class Operation,
          typename Arg1, typename Arg2>
class Binary<Operation, Arg1, Arg2, false>
{
public:
  Binary(Operation<Arg1, Arg2> const &);

  operation_type const &operation() const;
  Arg1 const &arg1() const;
  Arg2 const &arg2() const;
  void evaluate() const;
  template <typename ResultBlock>
  void apply(ResultBlock &result) const;
};
```

Member functions.

operation()	Return the operation associated with this expression.
arg1(), arg2()	Return the argument blocks associated with this expression.
evaluate()	Evaluate the operation, storing the result in the state of this expression block.
apply()	Evaluate the operation, storing the result in the provided block argument.

Chapter 7
Sourcery VSIPL API extensions

Abstract

This chapter contains detailed references for Sourcery VSIPL API extensions.

7.1. Introduction

The following man pages describe the Sourcery VSIPL API not covered by the VSIPL specification.

7.2. Direct Data Access to real vector views

7.2.1. vsip_csl_vattr_<type>

```
typedef struct
{
  vsip_scalar_<type> *data;
  vsip_stride  stride;
  vsip_length  length;
} vsip_csl_vattr_<type>;
```

where *<type>* can be one of `f`, `d`, `i`, `bl`, `si`, `uc`, `vi`, or `mi` to represent float, double, int, boolean, short int, unsigned char, vector index, or matrix index, respectively.

Member description

data — A pointer to the data array of the appropriate type (see the VSIPL specification for definitions of `vsip_scalar_<type>`).

stride — The stride of the array, i.e. the distance (in units of `vsip_scalar_<type>`) between two adjacent elements.

length — Number of elements in the array.

7.2.2. vsip_csl_vgetattrib_<type>

```
void
vsip_csl_vgetattrib_f(vsip_vview_f const *vector, vsip_csl_vattr_f \
*attrib);
void
vsip_csl_vgetattrib_d(vsip_vview_d const *vector, vsip_csl_vattr_d \
*attrib);
void
vsip_csl_vgetattrib_i(vsip_vview_i const *vector, vsip_csl_vattr_i \
*attrib);
void
vsip_csl_vgetattrib_si(vsip_vview_si const *vector, \
vsip_csl_vattr_si *attrib);
void
vsip_csl_vgetattrib_uc(vsip_vview_uc const *vector, \
vsip_csl_vattr_uc *attrib);
void
vsip_csl_vgetattrib_bl(vsip_vview_bl const *vector, \
vsip_csl_vattr_bl *attrib);
void
vsip_csl_vgetattrib_vi(vsip_vview_vi const *vector, \
vsip_csl_vattr_vi *attrib);
void
```

```
vsip_csl_vgetattrib_mi(vsip_vview_mi const *vector, \
vsip_csl_vattr_mi *attrib);
```

Description: Query the direct data access attributes for a real vector.

Result: The struct pointed to by `attrib` is filled with the direct-data-access attributes for `vector`.

Note: The pointer is *undefined* if the `block` is currently *released*. A valid pointer returned when the `block` is currently *admitted* becomes undefined if the `block` is subsequently *released*.

Example:

```
vsip_vview_f *vector = ...;
vsip_csl_vattr_f dda;
vsip_csl_vgetattrib_f(src, &dda);
if (dda.data)
{
  for (i = 0; i != dda.length; ++i)
    dda.data[i * dda.stride] = i;
}
  \
```

7.3. Direct Data Access to complex vector views

7.3.1. vsip_csl_cvattr_<type>

```
typedef struct
{
  vsip_scalar_<type> *data_r;
  vsip_scalar_<type> *data_i;
  vsip_stride  stride;
  vsip_length  length;
} vsip_csl_cvattr_<type>;
```

where <*type*> can be one of `f` or `d` to represent float or double, respectively.

Member description

data_r, data_i — Pointers to the data arrays of the appropriate type (see the VSIPL specification for definitions of `vsip_scalar_<type>`).

If `data_i=NULL`, `data_r` refers to interleaved complex data. Otherwise `data_r` refers to the real part and `data_i` to the imaginary part of split complex data.

stride — The stride of the array, i.e. the distance (in units of `vsip_scalar_<type>`) between two adjacent elements.

length — Number of elements in the array.

7.3.2. vsip_csl_cvgetattrib_<type>

```
void
vsip_csl_cvgetattrib_f(vsip_cvview_f const *vector, \
```

```
vsip_csl_cvattr_f *attrib);
void
vsip_csl_cvgetattrib_d(vsip_cvview_d const *vector, \
vsip_csl_cvattr_d *attrib);
```

Description: Query the direct data access attributes for a complex vector.

Result: The struct pointed to by `attrib` is filled with the direct-data-access attributes for `vector`. If `vector` holds interleaved complex data, `data_i` will be `NULL` and `data_r` will refer to interleaved complex data. If `vector` holds split-complex data, `data_r` and `data_i` will refer to the real and imaginary data arrays respectively.

Note: The pointer is *undefined* if the `block` is currently *released*. A valid pointer returned when the `block` is currently *admitted* becomes undefined if the `block` is subsequently *released*.

Example:

```
vsip_cvview_f *vector = ...;
vsip_csl_cvattr_f dda;
vsip_csl_cvgetattrib_f(src, &dda);
if (dda.data_r && dda.data_i) /* split-complex data */
{
  for (i = 0; i != dda.length; ++i)
  {
    dda.data_r[i * dda.stride] = i;
    dda.data_i[i * dda.stride] = i;
  }
}
else if (dda.data_r) /* interleaved-complex data */
{
  for (i = 0; i != dda.length; ++i)
  {
    dda.data_r[i * dda.stride] = i; /* real */
    dda.data_r[i * dda.stride+1] = i; /* imag */
  }
}
  \
```

7.4. Direct Data Access to real matrix views

7.4.1. vsip_csl_mattr_<type>

```
typedef struct
{
  vsip_scalar_<type> *data;
  vsip_stride  row_stride;
  vsip_length  row_length;
  vsip_stride  col_stride;
  vsip_length  col_length;
} vsip_csl_mattr_<type>;
```

where <*type*> can be one of `f`, `d`, `i`, `bl`, `si`, `uc`, `vi`, or `mi` to represent float, double, int, boolean, short int, unsigned char, vector index, or matrix index, respectively.

Member description

data	A pointer to the data array of the appropriate type (see the VSIPL specification for definitions of `vsip_scalar_<type>`).
row_stride, col_stride	The row-stride and column-stride of the array, i.e. the distance (in units of `vsip_scalar_<type>`) between two adjacent elements within a row and within a column, respectively.
row_length, col_length	Number of elements per row (i.e., number of columns), and number of elements per column (i.e., number of rows) in the array.

7.4.2. `vsip_csl_mgetattrib_<type>`

```
void
vsip_csl_mgetattrib_f(vsip_mview_f const *matrix, vsip_csl_mattr_f \
*attrib);
void
vsip_csl_mgetattrib_d(vsip_mview_d const *matrix, vsip_csl_mattr_d \
*attrib);
void
vsip_csl_mgetattrib_i(vsip_mview_i const *matrix, vsip_csl_mattr_i \
*attrib);
void
vsip_csl_mgetattrib_si(vsip_mview_si const *matrix, \
vsip_csl_mattr_si *attrib);
void
vsip_csl_mgetattrib_uc(vsip_mview_uc const *matrix, \
vsip_csl_mattr_uc *attrib);
void
vsip_csl_mgetattrib_bl(vsip_mview_bl const *matrix, \
vsip_csl_mattr_bl *attrib);
void
vsip_csl_mgetattrib_vi(vsip_mview_vi const *matrix, \
vsip_csl_mattr_vi *attrib);
void
vsip_csl_mgetattrib_mi(vsip_mview_mi const *matrix, \
vsip_csl_mattr_mi *attrib);
```

Description: Query the direct data access attributes for a real matrix.

Result: The struct pointed to by `attrib` is filled with the direct-data-access attributes for `matrix`.

Note: The pointer is *undefined* if the `block` is currently *released*. A valid pointer returned when the `block` is currently *admitted* becomes undefined if the `block` is subsequently *released*.

Example:

```

vsip_mview_f *matrix = ...;
vsip_csl_mattr_f dda;
vsip_csl_mgetattrib_f(src, &dda);
if (dda.data)
{
```

```
  for (i = 0; i != dda.col_length; ++i)
    for (j = 0; j != dda.row_length; ++i)
      dda.data[i * dda.col_stride + j * dda.row_stride] = i;
}
  \
```

7.5. Direct Data Access to complex matrix views

7.5.1. vsip_csl_cmattr_<type>

```
typedef struct
{
  vsip_scalar_<type> *data_r;
  vsip_scalar_<type> *data_i;
  vsip_stride  row_stride;
  vsip_length  row_length;
  vsip_stride  col_stride;
  vsip_length  col_length;
} vsip_csl_cmattr_<type>;
```

where <*type*> can be one of `f` or `d` to represent float or double, respectively.

Member description

data_r, data_i	Pointers to the data arrays of the appropriate type (see the VSIPL specification for definitions of `vsip_scalar_<type>`). If `data_i=NULL`, `data_r` refers to interleaved complex data. Otherwise `data_r` refers to the real part and `data_i` to the imaginary part of split complex data.
row_stride, col_stride	The row-stride and column-stride of the array, i.e. the distance (in units of `vsip_scalar_<type>`) between two adjacent elements within a row and within a column, respectively.
row_length, col_length	Number of elements per row (i.e., number of columns), and number of elements per column (i.e., number of rows) in the array.

7.5.2. vsip_csl_cmgetattrib_<type>

```
void
vsip_csl_cmgetattrib_f(vsip_cmview_f const *vector, \
vsip_csl_cmattr_f *attrib);
void
vsip_csl_cmgetattrib_d(vsip_cmview_d const *vector, \
vsip_csl_cmattr_d *attrib);
```

Description: Query the direct data access attributes for a complex matrix.

Result: The struct pointed to by `attrib` is filled with the direct-data-access attributes for `matrix`. If `matrix` holds interleaved complex data, `data_i` will be `NULL` and `data_r` will refer to interleaved complex data. If `matrix` holds split-complex data, `data_r` and `data_i` will refer to the real and imaginary data arrays respectively.

Note: The pointer is *undefined* if the `block` is currently *released*. A valid pointer returned when the `block` is currently *admitted* becomes undefined if the `block` is subsequently *released*.

Example:

```
vsip_cmview_f *matrix = ...;
vsip_csl_cmattr_f dda;
vsip_csl_cmgetattrib_f(src, &dda);
if (dda.data_r && dda.data_i) /* split-complex data */
{
  for (i = 0; i != dda.col_length; ++i)
    for (j = 0; j != dda.row_length; ++j)
    {
      dda.data_r[i * dda.col_stride + j * dda.row_stride] = i;
      dda.data_i[i * dda.col_stride + j * dda.row_stride] = i;
    }
}
else if (dda.data_r) /* interleaved-complex data */
{
  for (i = 0; i != dda.col_length; ++i)
    for (j = 0; j != dda.row_length; ++j)
    {
     dda.data_r[i * dda.col_stride + j * dda.row_stride] = i; /* \
real */
     dda.data_r[i * dda.col_stride + j * dda.row_stride + 1] = i; \
/* imag */
    }
}
  \
```

References

[VSPEC101] CodeSourcery, LLC . “VSIPL++ Specification 1.01”. Developed under subcontract 601-02-S-0109 under U.S. Government contract F30602-00-D-0221, 2005

Index

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W